Batch Matrix-form Equations and Implementation of Multilayer Perceptrons

Wieger Wesselink[1], Bram Grooten[1], Huub van de Wetering[1], Qiao Xiao[1], and Decebal Constantin Mocanu[1,2]

[1] *Eindhoven University of Technology, The Netherlands*
[2] *University of Luxembourg, Luxembourg*

{j.w.wesselink,b.grooten,h.v.d.wetering,q.xiao}@tue.nl, decebal.mocanu@uni.lu

Abstract

Multilayer perceptrons (MLPs) remain fundamental to modern deep learning, yet their algorithmic details are rarely presented in complete, explicit *batch matrix-form*. Rather, most references express gradients per sample or rely on automatic differentiation. Although automatic differentiation can achieve equally high computational efficiency, the usage of batch matrix-form makes the computational structure explicit, which is essential for transparent, systematic analysis, and optimization in settings such as sparse neural networks. This paper fills that gap by providing a mathematically rigorous and implementation-ready specification of MLPs in batch matrix-form. We derive forward and backward equations for all standard and advanced layers, including batch normalization and softmax, and validate all equations using the symbolic mathematics library SymPy. From these specifications, we construct uniform reference implementations in NumPy, PyTorch, JAX, TensorFlow, and a high-performance C++ backend optimized for sparse operations. Our main contributions are: (1) a complete derivation of batch matrix-form backpropagation for MLPs, (2) symbolic validation of all gradient equations, (3) uniform Python and C++ reference implementations grounded in a small set of matrix primitives, and (4) demonstration of how explicit formulations enable efficient sparse computation. Together, these results establish a validated, extensible foundation for understanding, teaching, and researching neural network algorithms.

Keywords: batch matrix-form backpropagation, multilayer perceptrons, symbolic equation validation, sparse neural networks, reference implementations

1 Introduction

The inner workings of multilayer perceptrons (MLPs) are often treated as implementation details hidden within large frameworks. As a result, the precise mathematical structure of forward and backward computations, especially in batch matrix-form, is rarely documented explicitly. This creates a gap between conceptual understanding and practical implementation, limiting both educational use and the development of optimized or sparse neural architectures.

Neural networks are widely implemented in frameworks such as PyTorch (Paszke et al., 2019), JAX (Bradbury et al., 2018), and TensorFlow (Abadi et al., 2016). While these frameworks perform efficiently for end users, they abstract away the underlying algorithms. This lack of transparency can indeed hinder students who aim to understand neural network mechanics or researchers who wish

to modify and experiment with them. In particular, backpropagation equations are usually implicit, hidden behind automatic differentiation tools (Baydin et al., 2017), which can limit applications requiring explicit matrix computations. Explicit derivations of batch matrix-form backpropagation equations are rarely provided in the literature and, to our knowledge, are absent for advanced layers such as batch normalization and softmax. To address this, the Nerva project (Wesselink, 2023) provides a collection of C++ and Python libraries that implement multilayer perceptrons (MLPs) using explicit batch matrix-form equations and readable reference implementations.

Sparse neural networks provide a clear example of why explicit backpropagation equations are useful. In Wesselink et al. (2024), we evaluate the performance of truly sparse layers. Since the support for sparse tensors in Python frameworks is still limited or experimental, we opt for a native C++ implementation using the Intel MKL Library for efficient sparse matrix operations. By employing explicit backpropagation equations, we not only circumvent the insufficient support for automatic differentiation in sparse computations, but also gain additional advantages: precise identification of the sparse and dense operations that dominate performance, and systematic exploration of optimizations.

The main focus of this paper remains the practical implementation of MLPs. We provide precise mathematical specifications of MLPs in explicit batch matrix-form, including complete backpropagation equations for standard and advanced layers, and produce straightforward, readable implementations that closely reflect these specifications. We include full matrix-form equations for commonly used layers, activation functions, and loss functions, along with derivations. Correctness is crucial; we therefore rely on the symbolic Python library SymPy (Meurer et al., 2017), which enables early validation of equations and precise localization of errors in formulas. This goes beyond standard numerical gradient checking, which only indicates that an error exists without identifying its source.

We express the execution of MLPs using a carefully selected set of matrix operations. Table 1 provides an overview of these operations, and Table 3 shows their realizations in Python frameworks such as NumPy (Harris et al., 2020), PyTorch, JAX, and TensorFlow, as well as in C++ using Eigen (Guennebaud et al., 2010). These tables serve as a bridge from mathematical equations to code, yielding implementations that are uniform, maintainable, and readily extensible to other frameworks. All implementations presented in this paper are part of the Nerva project (Wesselink, 2023) and are derived from explicit batch matrix-form equations, expressed in the set of matrix operations in Table 1. The C++ backends target high-performance computation with support for truly sparse networks, while the Python backends emphasize clarity and readability to support education and experimentation. Together, they provide complementary perspectives: performance-oriented implementations in C++ and transparent reference implementations in Python. Summarizing, this paper makes the following contributions:

1. We present a complete derivation of batch matrix-form backpropagation for multilayer perceptrons (MLPs), covering both standard and advanced layers such as batch normalization and softmax.

2. We symbolically validate all gradient equations using the SymPy library, providing mathematical assurance beyond numerical gradient checking.

3. We provide uniform reference implementations across major Python frameworks (NumPy, PyTorch, JAX, TensorFlow) and a high-performance C++ backend, all grounded in the same small set of matrix primitives.

4. We demonstrate how explicit matrix-form formulations make the computational structure

of MLPs visible, enabling systematic optimization, exploration of sparsity, and support for specialized architectures.

The remainder of this paper is structured as follows: Section 2 provides mathematical background, Section 3 introduces MLPs and their training via stochastic gradient descent. Section 4 covers sparse neural networks and their implementation in Nerva, while the design and implementation details are described in Section 5. Section 6 presents experimental results and we conclude the paper in Section 7. Appendices include layer equations (A), matrix operation implementations (B), activation functions (C), loss functions (D), weight initialization (E), optimization functions (F), and learning rate schedulers (G), with derivations provided where relevant.

2 Mathematical Background

In this section we provide notational conventions, we provide definitions for the gradient and the Jacobian, and a comprehensive table with matrix operations that are necessary for the execution of multilayer perceptrons.

In the rest of this paper we use the following notations. Vectors are denoted using lowercase symbols x, y, z, and matrices using uppercase symbols X, Y, Z. The columns of a matrix $X \in \mathbb{R}^{m \times n}$ are denoted as $x^1, \ldots, x^n$, while the rows are denoted as $x_1, \ldots, x_m$. Occasionally the subscript notation x_i is also used to denote the i^{th} element of vector x, but this will be clear from context. To distinguish between row and column vectors, we denote their domains as $\mathbb{R}^{1 \times n}$ and $\mathbb{R}^{m \times 1}$, respectively. We occasionally use the dot symbol $\cdot$ to denote matrix multiplication; we never use it for the vector dot product.

Having established these general notations, we now apply them in the context of neural networks. Throughout the paper, x, X denote inputs, y, Y outputs, z, Z intermediate values, and t, T targets. The number of inputs of a layer is denoted by D and the number of outputs by K. N indicates the number of samples in a mini-batch. We follow the convention of the major neural network frameworks where data is stored using a row layout. This means that by default x, y and z are considered to be row vectors, and that each row of an input matrix X represents an example of a dataset. For example, if $X \in \mathbb{R}^{\mathrm{N} \times \mathrm{D}}$, then $x_i \in \mathbb{R}^{1 \times \mathrm{D}}$ represents the i-th example. Figure 1 provides a graphical representation of a multilayer perceptron, illustrating the notation introduced above.

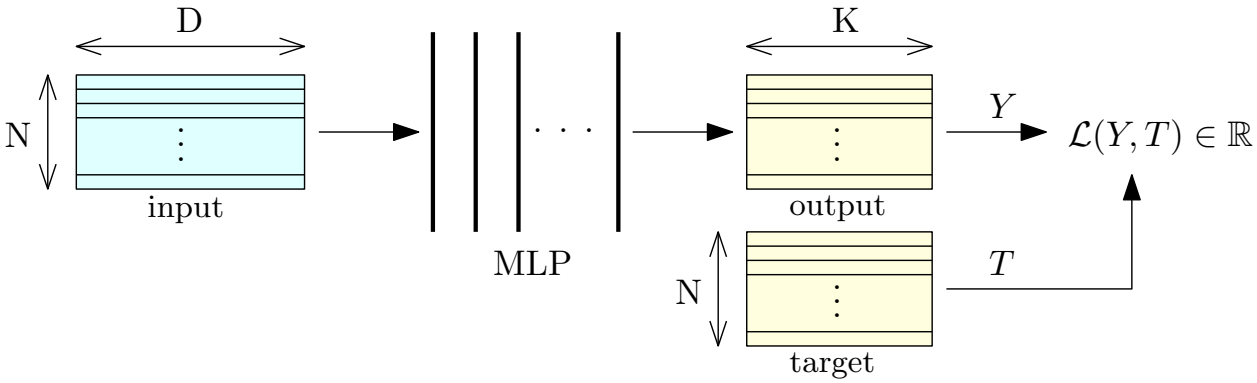

Figure 1: Graphical representation of a multilayer perceptron. The input is a mini-batch, represented by an $\mathrm{N} \times \mathrm{D}$ matrix consisting of N examples with D features. In the feedforward pass this input is passed through an MLP with a series of layers, represented by vertical bars. This results in an $\mathrm{N} \times \mathrm{K}$ output matrix Y. From the output Y and the expected output T the loss $\mathcal{L}(Y, T)$ is computed.

We express core concepts used throughout our work in the following definitions.

Definition 1 (Gradient)**.** *Let $f : \mathbb{R}^{m\times n} \to \mathbb{R}$ be a function with input X that has elements x_{ij}, with $m, n \in \mathbb{N}^+$. Then the gradient $\nabla_X f$ is defined as*

$$\nabla_X f(X) = \begin{bmatrix} \frac{\partial f(X)}{\partial x_{11}} & \cdots & \frac{\partial f(X)}{\partial x_{1n}} \\ \vdots & \ddots & \vdots \\ \frac{\partial f(X)}{\partial x_{m1}} & \cdots & \frac{\partial f(X)}{\partial x_{mn}} \end{bmatrix}. \tag{1}$$

To streamline later derivations, we introduce a shorthand notation for the gradient of the loss.

Definition 2 (Gradient of the loss function)**.** *For neural networks with output $Y \in \mathbb{R}^{\mathrm{N}\times\mathrm{K}}$ and target $T \in \mathbb{R}^{\mathrm{N}\times\mathrm{K}}$, let $\mathcal{L} : \mathbb{R}^{\mathrm{N}\times\mathrm{K}} \times \mathbb{R}^{\mathrm{N}\times\mathrm{K}} \to \mathbb{R}$ be a fixed loss function. The gradient of the loss with respect to the output is written as*

$$DY = \nabla_Y \mathcal{L}(Y, T). \tag{2}$$

More generally, if Y depends on a parameter Z (i.e., $Y = Y(Z)$), we define

$$DZ = \nabla_Z \mathcal{L}(Y(Z), T). \tag{3}$$

Definition 3 (Jacobian)**.** *Let $f : \mathbb{R}^n \to \mathbb{R}^m$ be a function with input $x \in \mathbb{R}^n$. Then the Jacobian $\frac{\partial f}{\partial x}$ is defined as follows.*

$$\frac{\partial f}{\partial x}(x) = \begin{bmatrix} \frac{\partial f_1(x)}{\partial x_1} & \cdots & \frac{\partial f_1(x)}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m(x)}{\partial x_1} & \cdots & \frac{\partial f_m(x)}{\partial x_n} \end{bmatrix}. \tag{4}$$

We adopt the convention that the Jacobian does not depend on whether x and $f(x)$ are represented as row or column vectors. This choice is consistent with the symbolic Python library SymPy, which we use for validation, although other conventions exist in the literature.

The execution of MLPs depends on a small number of matrix operations. In Table 1 on page 5 we provide a mathematical notation, a code representation, and a definition for the most important operations. These operations are used in the implementation of activation functions, loss functions, and the feedforward and backpropagation equations of neural network layers. Basic operations like matrix assignment, matrix indexing and matrix dimensions are omitted, to keep the resulting code familiar to users of the respective Python frameworks. There is some redundancy in the table, to allow for more efficient, or numerically stable implementations. We refrain from using broadcasting notation, as commonly found in frameworks like NumPy, where arrays of different dimensions can be added and use the standard notation of matrix calculus instead. This approach with explicit dimensions is needed for validating the correctness using SymPy. For instance, to compute the sum of elements in a column vector $x \in \mathbb{R}^{n\times 1}$, we express it as the dot product $1_n^\top \cdot x$, with $1_n = (1, \ldots, 1)^\top$ a column vector of ones.

3 Multilayer Perceptrons

A multilayer perceptron is an artificial neural network comprised of layers, each containing neurons, which are the basic processing units, see Fig. 2. The network typically consists of three types of

Table 1: An overview of matrix operations that are needed for the implementation of the class of MLPs described in this paper. We assume that $X, Y \in \mathbb{R}^{m \times n}$ and $Z \in \mathbb{R}^{n \times k}$, where $k, m, n \in \mathbb{N}^+$. Wherever possible, we use m to denote the number of rows and n to denote the number of columns of a matrix or vector. The function σ is the sigmoid function as defined in Appendix C. In the second column we use the unified Nerva API notation, which is implemented consistently across the Python and C++ backends, ensuring a one-to-one correspondence between equations and code.

OPERATION	NERVA API	DEFINITION
0_m	`zeros(m)`	$m \times 1$ column vector with elements equal to 0
0_{mn}	`zeros(m, n)`	$m \times n$ matrix with elements equal to 0
1_m	`ones(m)`	$m \times 1$ column vector with elements equal to 1
1_{mn}	`ones(m, n)`	$m \times n$ matrix with elements equal to 1
$\mathbb{I}_n$	`identity(n)`	$n \times n$ identity matrix
$X^\top$	`X.T`	transposition
cX	`c * X`	scalar multiplication, $c \in \mathbb{R}$
$X + Y$	`X + Y`	addition
$X - Y$	`X - Y`	subtraction
$X \cdot Z$	`X @ Z` or `X * Z`	matrix multiplication, also denoted as XZ
$x^\top y$ or $xy^\top$	`dot(x,y)`	dot product, $x, y \in \mathbb{R}^{m \times 1}$ or $x, y \in \mathbb{R}^{1 \times n}$
$X \odot Y$	`hadamard(X,Y)`	element-wise product of X and Y
$\mathsf{diag}(X)$	`diag(X)`	column vector that contains the diagonal of X
$\mathsf{Diag}(x)$	`Diag(x)`	diagonal matrix with x as diagonal, $x \in \mathbb{R}^{1 \times n}$ or $x \in \mathbb{R}^{m \times 1}$
$1_m^\top \cdot X \cdot 1_n$	`elements_sum(X)`	sum of the elements of X
$x \cdot 1_n^\top$	`column_repeat(x, n)`	n copies of column vector $x \in \mathbb{R}^{m \times 1}$
$1_m \cdot x$	`row_repeat(x, m)`	m copies of row vector $x \in \mathbb{R}^{1 \times n}$
$1_m^\top \cdot X$	`columns_sum(X)`	$1 \times n$ row vector with sums of the columns of X
$X \cdot 1_n$	`rows_sum(X)`	$m \times 1$ column vector with sums of the rows of X
$\max(X)_{\text{col}}$	`columns_max(X)`	$1 \times n$ row vector with maximum values of the columns of X
$\max(X)_{\text{row}}$	`rows_max(X)`	$m \times 1$ column vector with maximum values of the rows of X
$(1_m^\top \cdot X)/n$	`columns_mean(X)`	$1 \times n$ row vector with mean values of the columns of X
$(X \cdot 1_n)/m$	`rows_mean(X)`	$m \times 1$ column vector with mean values of the rows of X
$f(X)$	`apply(f, X)`	element-wise application of $f : \mathbb{R} \to \mathbb{R}$ to X
e^X	`exp(X)`	element-wise application of $f : x \to e^x$ to X
$\log(X)$	`log(X)`	element-wise application of the natural logarithm $f : x \to \ln(x)$ to X
$1/X$	`reciprocal(X)`	element-wise application of $f : x \to 1/x$ to X
$\sqrt{X}$	`sqrt(X)`	element-wise application of $f : x \to \sqrt{x}$ to X
$X^{-1/2}$	`inv_sqrt(`X`)`	element-wise application of $f : x \to x^{-1/2}$ to X
$\log(\sigma(X))$	`log_sigmoid(`X`)`	element-wise application of $f : x \to \log(\sigma(x))$ to X,

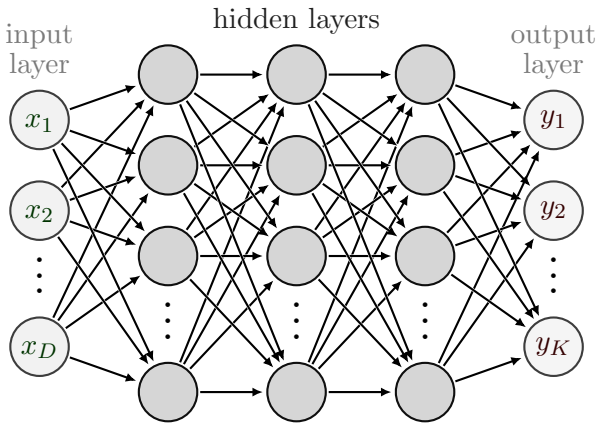

Figure 2: A multilayer perceptron with three hidden layers. The input vector x has D elements, and the output vector y has K elements. Neurons in each layer are represented by circles, and connections between neurons are indicated by arrows. The weights associated with the connections form the parameters of the linear layers.

layers: an input layer, a number of hidden layers, and an output layer. Neurons in the input layer represent features of the input data, while neurons in subsequent layers capture increasingly complex patterns. At the core of MLPs lies the concept of linear layers. In a linear layer, the input row vector x undergoes a linear transformation represented by $xW^\top + b$, where W is a weight matrix[1] and b is a bias vector. In matrix-form, where each row of a batch X is an input, this turns into

$$Y = X \cdot W^\top + 1_\mathrm{N} \cdot b. \tag{5}$$

Since linear transformations are limited in their expressive power, activation functions (Dubey et al., 2022) can be applied to the output of linear layers to introduce non-linearity to the model. Common activation functions include the sigmoid, hyperbolic tangent (tanh), and rectified linear unit (ReLU). These functions allow the network to learn complex, non-linear patterns in the data. Beyond linear and activation layers, additional layers like dropout and batch normalization contribute to the performance and stability of MLPs. A loss function is used to quantify the disparity between the predicted outputs and the true targets. Common loss functions include mean squared error for regression tasks and cross-entropy loss for classification tasks. The term *feedforward* describes the process of passing data through the network from input to output. *Backpropagation* (Kelley, 1960) is the mechanism used to iteratively adjust the weights and biases, making the neural network "learn". This is done based on the gradients with respect to the loss function that are computed during a backward pass. This iterative process refines the model's ability to make accurate predictions. This adjustment of the network parameters is called optimization. Gradient Descent, a popular optimization method, minimizes the loss function by iteratively moving in the direction of steepest descent in the parameter space. Modern variations, like Adam (Kingma and Ba, 2015), incorporate adaptive learning rates, enhancing convergence speed and stability. In summary, MLPs constitute a powerful framework for modeling complex relationships within data. MLPs are universal function approximators, meaning they can theoretically learn and approximate any continuous function. An extensive treatment of MLPs (without matrix-form equations) can be found in (Zhang et al., 2024).

Mathematically, a neural network can be viewed as a function $f_\Theta : \mathbb{R}^\mathrm{D} \to \mathbb{R}^\mathrm{K}$ that depends on a set of parameters Θ (including the weights and biases of linear layers), and that maps an input

[1]Matrix W is transposed to be consistent with existing frameworks.

vector $x \in \mathbb{R}^{\mathrm{D}}$ to an output vector $y \in \mathbb{R}^{\mathrm{K}}$. We consider the supervised learning regime, where we have a dataset of inputs $\{x_i\}_{1 \leq i \leq \mathrm{N}}$, and a corresponding set of targets $\{t_i\}_{1 \leq i \leq \mathrm{N}}$. The output is denoted as $y_i = f_\Theta(x_i)$. The goal of training a neural network is to find values for Θ that minimize the loss $\sum_{i=1}^{\mathrm{N}} \mathcal{L}(y_i, t_i)$, where $\mathcal{L}$ is a loss function that measures a distance between its arguments y and t. In practical applications, a neural network can handle multiple inputs at once. This is done by taking a matrix $X \in \mathbb{R}^{\mathrm{N} \times \mathrm{D}}$ as input (a mini-batch) together with a matrix $T \in \mathbb{R}^{\mathrm{N} \times \mathrm{K}}$, such that the rows of X and T are the inputs with their corresponding targets. We can lift the function f_Θ to multiple inputs using the matrix-form notation $Y = f_\Theta(X)$, where $y_i = f_\Theta(x_i)$ for $1 \leq i \leq \mathrm{N}$. The function $\mathcal{L}$ is expanded to multiple inputs using

$$\mathcal{L}(Y, T) = \sum_{i=1}^{\mathrm{N}} \mathcal{L}(y_i, t_i).$$

When using mini-batches as input, the training of a neural network can be expressed using matrix-matrix products instead of matrix-vector products. This generally increases the performance significantly, since there are specialized libraries available that apply massive parallelism and hardware optimizations like vectorization. For this reason we will provide specifications of neural networks that handle inputs in matrix-form.

3.1 Batch normalization

Batch normalization (Ioffe and Szegedy, 2015) can be used to normalize the inputs of neural network layers by adjusting the mean and variance of the data. This can help to speed up and stabilize training by mitigating issues such as vanishing or exploding gradients. We will slightly rewrite the formulation of Ioffe and Szegedy (2015), and convert it into matrix-form. Note that the input and output size of a batch normalization layer are the same, i.e., $\mathrm{D} = \mathrm{K}$. Let $X \in \mathbb{R}^{\mathrm{N} \times \mathrm{D}}$ be an input batch. Each row of X corresponds with an example, and each column with a feature. Let $\mu = (1_{\mathrm{N}}^\top \cdot X)/\mathrm{N}$, i.e., the row vector $\mu \in \mathbb{R}^{1 \times \mathrm{D}}$ contains the averages of the columns in the batch. For a given example x_i, we write $r_i = x_i - \mu \in \mathbb{R}^{1 \times \mathrm{D}}$, which represents the centered feature vector of that example. For a given feature j, we define $\sigma_j^2 = ((r^j)^\top \cdot r^j)/\mathrm{N}$, which is the variance of the j-th feature across the batch. The variance vector $\Sigma \in \mathbb{R}^{1 \times \mathrm{D}}$ that contains σ_j^2 as its elements can then be formulated as $\Sigma = \mathsf{diag}(R^\top R)^\top/\mathrm{N}$, where R is the $\mathrm{N} \times \mathrm{D}$ matrix with rows $r_1, \ldots, r_{\mathrm{N}}$. The first part of batch normalization, which standardizes the input X to a matrix Z with mean 0 and standard deviation 1, is then given by the equations

$$\begin{aligned} R &= X - \frac{1_{\mathrm{N}} \cdot 1_{\mathrm{N}}^\top}{\mathrm{N}} \cdot X \\ \Sigma &= \frac{1}{\mathrm{N}} \cdot \mathsf{diag}(R^\top R)^\top \\ Z &= (1_{\mathrm{N}} \cdot \Sigma^{-\frac{1}{2}}) \odot R. \end{aligned} \tag{6}$$

The second part of batch normalization consists of a scale and a shift operation that transforms the rows $z_1, \ldots, z_{\mathrm{N}}$ of Z using $y_i = \gamma \odot z_i + \beta$, with $\beta, \gamma \in \mathbb{R}^{1 \times \mathrm{D}}$. In matrix-form this becomes

$$Y = (1_{\mathrm{N}} \cdot \gamma) \odot Z + 1_{\mathrm{N}} \cdot \beta. \tag{7}$$

The parameters β and γ are learnable parameters, meaning that they are periodically updated using an optimizer function in order to minimize the loss. Note that each column of R sums to zero, which is a direct consequence of subtracting the average μ from the rows of X. Since Z scales the columns of R with a constant factor, the matrix Z has the same property. This property will be exploited in the derivation of the backpropagation equations in Appendix A.

3.2 Dropout

Dropout (Srivastava et al., 2014) is a technique that can be used to prevent *overfitting* of a neural network. Multiple variants of dropout exists. In this section we discuss the DropConnect variant (Wan et al., 2013). During training of the network it randomly replaces a fraction $0 < p < 1$ of the weights of a linear layer (i.e., the elements of the matrix W) by zero, for example at the start of each epoch of training. An easy way to implement this is to define a matrix R (not to be confused with the R used in batch normalization) with the same dimensions as the weight matrix W and that contains the value 0 on all positions that are dropped, and the value 1 on all other positions. The effect of dropping the weights can then be achieved by using the weight matrix $W' = W \odot R$ instead of W. However, doing this has an unwanted side effect: it will cause the sum of the absolute values of the weights to decrease with a factor $1 - p$ on average. To compensate for this, the non-zero values of R should get the value $1/(1 - p)$ instead of 1. The feedforward equation of dropout that we use is therefore

$$Y = X(W \odot R)^\top + 1_N \cdot b. \tag{8}$$

Note that this variation of dropout is applied to the connections between the neurons. It is also possible to apply dropout to the neurons of a layer.

3.3 Training a multilayer perceptron

The first step of training an MLP is to choose an architecture for the network that is suitable for a given task. In case of an MLP this comes down to selecting a sequence of layers, their sizes and activation functions, and a loss function. How to do this is outside the scope of this paper. However, in the appendices we provide detailed descriptions of several layer types (A), activation functions (C) and loss functions (D). The next step is to initialize the parameters of the layers, in particular the weights and biases of the linear layers. This too can have a large impact on performance (Hayou et al., 2018). In Appendix E we give a few common weight initialization strategies. Training an MLP is usually done with (a variant of) stochastic gradient descent and consists of three steps that are performed repeatedly:

1. (feedforward) Given an input batch X and the neural network parameters Θ, compute the output Y.

2. (backpropagation) Given outputs Y corresponding to inputs X and expected outputs T, compute the gradient of the loss $\mathsf{D}Y$. Then from Y and $\mathsf{D}Y$, compute the gradient $\mathsf{D}\Theta$ of the parameters Θ.

3. (optimization) Given the gradient $\mathsf{D}\Theta$, update the parameters Θ.

These steps are performed for each input batch X with target T of a dataset, and this process is repeated a given number of times (the number of epochs). For the optimization step a learning rate (Wu et al., 2019) is used to determine the size of the steps taken. It controls how much the model's weights should be adjusted with respect to the gradient of the loss function. The learning rate is often decreased during training, in Appendix G we provide an overview of some learning rate schedulers.

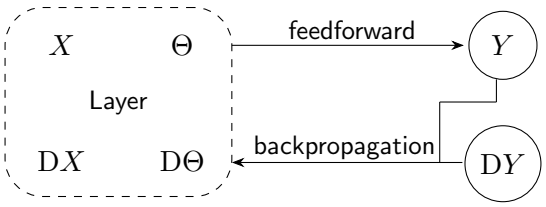

Figure 3: Data flow of a layer. A layer stores the input X and the layer parameters θ (e.g., the weight matrix W and the bias vector b in case of a linear layer), and the corresponding gradients $\mathsf{D}X$ and $\mathsf{D}\theta$ with respect to the loss function. In the feedforward step the output $Y = f_\theta(X)$ is calculated and forwarded to the next layer. In the backpropagation step the output Y and its gradient $\mathsf{D}Y$ are obtained from the next layer and used to calculate the gradients $\mathsf{D}X$ and $\mathsf{D}\theta$. X and $\mathsf{D}X$ are then passed back to the previous layer. In the optimization step the gradient $\mathsf{D}\theta$ is used to update the value θ.

4 Sparse Neural Networks

This section presents a case study for assessing the performance of truly sparse neural networks, using the C++ backend of the Nerva libraries (Wesselink, 2023). The study reported in (Wesselink et al., 2024) serves as an example of the approach advocated in this paper: implementing neural networks with explicit backpropagation equations in batch matrix-form and clearly defined matrix operations.

The practical implementation of sparse neural networks remains a significant challenge. While major frameworks are actively extending sparse tensor support, limitations in operator coverage, efficiency, and autograd capabilities persist. Researchers often resort to workarounds such as binary masks applied to dense tensors (Liu and Wang, 2023), which do not deliver the true computational and memory savings of sparsity. As a result, evaluating the performance of truly sparse networks within these ecosystems is difficult.

The Nerva backend forgoes automatic differentiation not only because Eigen does not provide native autograd support, but also because existing C++ autodiff libraries are not well-optimized for sparse parameters. Instead, it uses manually derived backpropagation equations in batch matrix-form. Using these explicit equations has two important benefits:

1. **Correctness:** By using validated equations and expressing their implementation in a small set of well-defined matrix operations, the correctness of the implementation follows directly from the validated equations.

2. **Profiling and optimization:** Each equation corresponds to one or more matrix operations, enabling fine-grained timing and analysis of sub-steps (e.g., in batch normalization) and core sparse/dense products.

Fine-grained profiling reveals that the most critical and performance-sensitive operations in sparse networks are:

$$
\begin{aligned}
A &= SB && \text{(sparse-dense product, feedforward)} \\
A &= S^\top B && \text{(sparse-dense product, backward pass)} \\
S &= AB^\top && \text{(dense-dense product, backward pass)}
\end{aligned}
$$

where A and B are dense matrices, and S and T are sparse matrices. Among these, the product $S = AB^\top$ is particularly challenging. Because S is sparse, only a small subset of values from the

dense product $AB^\top$ are retained. In practice, this operation often dominates backpropagation in linear layers, consuming over half of the backpropagation time in experiments such as training on CIFAR-10 at 95% sparsity. The right-hand side is a dense matrix product, but only a fraction of its entries are actually used to update the sparse weight matrix. Explicitly forming and storing $AB^\top$ is therefore computationally expensive and negates the memory advantages of sparsity. Several approaches can be used to compute this product directly in sparse form, and we experimented with a number of them in the Nerva libraries. Additional performance issues appeared in various parts of the implementation. Dense products were sometimes not forwarded by Eigen's expression trees to MKL, resulting in slower execution, and some MKL sparse kernels exhibited poor efficiency for large matrices. The advantage of expressing the implementation through explicit matrix-form equations is that such bottlenecks become straightforward to detect and analyze.

Empirical results confirm the practical benefits of this approach. While dense MLPs implemented with Nerva perform on par with PyTorch, the advantage becomes clear in sparse settings: PyTorch's masking strategy maintains nearly constant runtime across sparsity levels, whereas Nerva's truly sparse implementation accelerates as sparsity increases. At 99% sparsity on CIFAR-10, Nerva is about four times faster and uses dramatically less memory, enabling training of networks too large for dense frameworks. These results illustrate how explicit matrix-form equations combined with an efficient sparse backend, as promoted in this paper, provide a transparent path to studying, optimizing, and scaling truly sparse neural networks beyond the limits of dense frameworks.

5 Design and implementation of multilayer perceptrons

In this section we explain our implementation. We have chosen for an object oriented design in which layers are represented by classes. The main algorithmic structure is presented using our NumPy framework, but it is similar to all others.

Both an MLP and its layers support a feedforward computation, a backpropagation and an optimization step, see Listing 1. Each layer of an MLP depends on a number of parameters θ. In a layer we store the input X, the parameters θ, their gradients $\mathsf{D}X$ and $\mathsf{D}\theta$, and a function object for the optimization step. During the feedforward step the output Y is calculated and stored as input in the next layer. During the backpropagation step the gradients $\mathsf{D}X$ and $\mathsf{D}\theta$ are computed from the output Y and its gradient $\mathsf{D}Y$, that are obtained from the next layer. This is depicted in Fig. 3. Listing 2 shows how stochastic gradient descent can be implemented using our NumPy framework. Here `train_loader` is an object that extracts batches with their corresponding targets from a dataset, similar to the PyTorch `DataLoader`. In case of classification tasks, the targets T are often represented by a vector of integers, instead of a matrix with the same dimensions as X. In such a case a *one-hot encoding* is needed that expands T to a matrix containing zeros and ones. This is taken care of by the `train_loader` object. Note that in our design the neural network parameters Θ and the optimization functions are stored inside the layers of the MLP `M`. The gradient descent optimizer is demonstrated in Listing 3. It stores a reference to a parameter `x` and the corresponding gradient `Dx`. The update to the parameters is specified in Appendix F. A layer can contain multiple parameters. We support different optimization functions for different parameters. This is achieved using the composite design pattern, as shown in Listing 3. An MLP can be constructed as shown in Listing 4.

5.1 The softmax layer

In this section we show how to design and implement the softmax layer. We use it to demonstrate a non-trivial derivation of a backpropagation equation. For an overview of other layers see Appendix

Listing 1 Multilayer perceptron implementation. Only the fundamental methods needed for the execution are given. For completeness the interface of the layer classes is added, which has the same structure. The attributes of a layer are the input `X` and its gradient `DX`, and a function pointer `optimizer` that takes care of the optimization step. A multilayer perceptron has only one attribute, `layers`, for storing the list of layers.

```
class Layer(object):
    def feedforward(self, X: Matrix) -> Matrix:
    def backpropagate(self, Y: Matrix, DY: Matrix):
    def optimize(self, eta: float):

class MultilayerPerceptron(object):
    def feedforward(self, X: Matrix) -> Matrix:
        for layer in self.layers:
            X = layer.feedforward(X)
        return X

    def backpropagate(self, Y: Matrix, DY: Matrix):
        for layer in reversed(self.layers):
            layer.backpropagate(Y, DY)
            Y, DY = layer.X, layer.DX

    def optimize(self, eta: float):
        for layer in self.layers:
            layer.optimize(eta)
```

Listing 2 A minimal implementation of stochastic gradient descent in NumPy.

```
def stochastic_gradient_descent(
        M: MultilayerPerceptron,
        epochs: int,
        loss: LossFunction,
        learning_rate: LearningRateScheduler,
        train_loader: DataLoader):

    for epoch in range(epochs):
        lr = learning_rate(epoch)
        for (X, T) in train_loader:
            Y = M.feedforward(X)
            # take the average of the gradients
            N = X.shape[0]  # the batch size
            DY = loss.gradient(Y, T) / N
            M.backpropagate(Y, DY)
            M.optimize(lr)
```

A. Besides this we show how the backpropagation equations are transformed into code, and how their correctness is validated using SymPy.

A softmax layer is a linear layer that uses the softmax function (10) as its activation function. It normalizes the output of the linear layer into a probability distribution. The feedforward equations for a single input row vector $x \in \mathbb{R}^{1\times \mathrm{D}}$ are given by

$$\begin{cases} z &= xW^\top + b \\ y &= \mathsf{softmax}(z), \end{cases} \tag{9}$$

Listing 3 Optimizer Implementation. An optimizer stores references to a parameter `x` and its gradient `Dx` with respect to the loss function. Optimizers for multiple parameters are joined using the composite design pattern. The method `update` performs the actual optimization step. The parameter `eta` is the step size or learning rate, and `L` is an instance of a `Layer`.

```
class GradientDescentOptimizer(Optimizer):
    def __init__(self, x, Dx):
        self.x = x
        self.Dx = Dx

    def update(self, eta):
        self.x -= eta * self.Dx

optimizer_W = MomentumOptimizer(layer.W, layer.DW, 0.9)
optimizer_b = NesterovOptimizer(layer.b, layer.Db, 0.9)
L.optimizer = CompositeOptimizer(optimizer_W, optimizer_b)
```

Listing 4 Multilayer perceptron construction. The functions `set_optimizer` and `set_weigths` are offered as convenience functions to simplify the work. It is also possible to initialize the attributes manually.

```
    M = MultilayerPerceptron()
    input_size = 3072
    output_size = 1024
    act = ReLUActivation()
    layer = ActivationLayer(input_size, output_size, act)
    layer.set_optimizer('Momentum(0.9)')
    layer.set_weights('Xavier')
    M.layers.append(layer)
    ...
```

with $y, z \in \mathbb{R}^{1\times \mathrm{K}}$ and with

$$\mathsf{softmax}(z) = \frac{e^z}{e^z \cdot 1_{\mathrm{K}}}. \tag{10}$$

Here, $e^z/(e^z \cdot 1_{\mathrm{K}})$ is in the matrix notation that we use in this paper, see Table 1. It is equivalent to the more common element-wise definition, expressed in the coordinates z_i:

$$\mathsf{softmax}(z)_i = \frac{e^{z_i}}{\sum_{k=1}^{\mathrm{K}} e^{z_k}} \qquad (1 \le i \le \mathrm{K}).$$

For a mini-batch $X \in \mathbb{R}^{\mathrm{N}\times \mathrm{D}}$ the feedforward equations are given by

$$\begin{cases} Z &= XW^\top + 1_{\mathrm{N}} \cdot b \\ Y &= \mathsf{softmax}(Z), \end{cases} \tag{11}$$

with $Y, Z \in \mathbb{R}^{\mathrm{N}\times \mathrm{K}}$ and

$$\mathsf{softmax}(Z) = e^Z \odot \left(\frac{1}{e^Z \cdot 1_{\mathrm{K}}} \cdot 1_{\mathrm{K}}^\top \right). \tag{12}$$

As an example we will now derive the equation for the gradient $\mathsf{D}Z$. Whenever we apply a rule in a derivation, the number of the corresponding equation is put above the equal sign between parentheses. By the product rule we have

$$\begin{aligned}
&\frac{\partial}{\partial z}\mathsf{softmax}(z) = \frac{\partial}{\partial z}\left(\frac{e^z}{e^z \cdot 1_{\mathrm{K}}}\right) \\
&= \frac{\partial}{\partial z}\left(e^z \cdot \frac{1}{e^z \cdot 1_{\mathrm{K}}}\right) \\
&\overset{(19)}{=} \frac{\partial e^z}{\partial z} \cdot \frac{1}{e^z \cdot 1_{\mathrm{K}}} + (e^z)^\top \frac{\partial}{\partial z}\left(\frac{1}{e^z \cdot 1_{\mathrm{K}}}\right) \\
&= \mathsf{Diag}(e^z) \cdot \frac{1}{e^z \cdot 1_{\mathrm{K}}} - (e^z)^\top \cdot \frac{e^z}{(e^z \cdot 1_{\mathrm{K}})^2} \\
&= \mathsf{Diag}\left(\frac{e^z}{e^z \cdot 1_{\mathrm{K}}}\right) - \left(\frac{e^z}{e^z \cdot 1_{\mathrm{K}}}\right)^\top \cdot \frac{e^z}{e^z \cdot 1_{\mathrm{K}}} \\
&= \mathsf{Diag}(y) - y^\top y,
\end{aligned} \tag{13}$$

with $y = \mathsf{softmax}(z)$. Now let y_i, z_i be the i^{th} row of Y, Z for $1 \leq i \leq \mathrm{N}$, hence $y_i = \mathsf{softmax}(z_i)$. Using the chain rule we find

$$\frac{\partial \mathcal{L}}{\partial z_i} \overset{(22)}{=} \frac{\partial \mathcal{L}}{\partial y_i}\frac{\partial}{\partial z_i}\mathsf{softmax}(z_i),$$

hence

$$\begin{aligned}
\mathsf{D}z_i &= \mathsf{D}y_i \cdot \Big(\mathsf{Diag}(y_i) - y_i^\top y_i\Big) \\
&= \mathsf{D}y_i \odot y_i - \mathsf{D}y_i \cdot y_i^\top y_i.
\end{aligned}$$

Using property (26) we can generalize this into matrix-form:

$$\mathsf{D}Z = (\mathsf{D}Y - \mathsf{diag}(\mathsf{D}Y \cdot Y^\top) \cdot 1_{\mathrm{K}}^\top) \odot Y. \tag{14}$$

The remaining backpropagation equations for the softmax layer are

$$\begin{aligned}
\mathsf{D}W &= \mathsf{D}Z^\top \cdot X \\
\mathsf{D}b &= 1_{\mathrm{N}}^\top \cdot \mathsf{D}Z \\
\mathsf{D}X &= \mathsf{D}Z \cdot W.
\end{aligned} \tag{15}$$

They are derived in Appendix A. Listing 5 shows how the feedforward and backpropagation steps are implemented in NumPy, where we have used the matrix operations from Table 3 to transform the equations into code. In Listing 6 we demonstrate how the backpropagation equations are validated in SymPy, for a small instance ($\mathrm{D} = 3, \mathrm{K} = 2, \mathrm{N} = 2$) with a simplified squared error loss function. While this does not result in a complete proof, it turns out to be sufficient to detect errors in the equations. Moreover, if an error is detected, it is easy to locate the source of the error by checking the individual steps of the derivation in SymPy. This is a significant improvement over gradient checking using numerical approximations, which does not help to locate the source. Listing 7 shows how the derivation of equation 13 can be validated using SymPy.

Listing 5 Softmax layer implementation in NumPy. We use Table 3 to transform equations 9, 14 and 15 into code.

```
class SoftmaxLayer(LinearLayer):
  def feedforward(self, X: Matrix) -> Matrix:
    ...
    Z = X @ W.T + row_repeat(b, N)
    Y = softmax(Z)
    ...

  def backpropagate(self, Y: Matrix, DY: Matrix) -> None:
    ...
    DZ = hadamard(Y, DY - column_repeat(diag(DY @ Y.T), K))
    DW = DZ.T @ X
    Db = columns_sum(DZ)
    DX = DZ @ W
    ...
```

Listing 6 Validating gradients in SymPy using symbolic differentiation. The function `gradient` computes the gradient of a matrix expression (of dimension 1×1 in case of the loss) with respect to a vector of variables, and is implemented using the SymPy symbolic differentiation operator `diff`. The variables `w`, `b`, `x`, and `z` contain the elements of W, b, X and Z in equation 9. The function `equal_matrices` checks equality of two matrices using the function `sympy.simplify`.

```
    # backpropagation
    DZ = hadamard(Y, DY - column_repeat(diag(DY * Y.T), N))
    DW = DZ.T * X
    Db = columns_sum(DZ)
    DX = DZ * W

    # symbolic differentiation followed by gradient check
    DW1 = gradient(loss(Y), w)
    Db1 = gradient(loss(Y), b)
    DX1 = gradient(loss(Y), x)
    DZ1 = gradient(loss(Y), z)
    self.assertTrue(equal_matrices(DW, DW1))
    self.assertTrue(equal_matrices(Db, Db1))
    self.assertTrue(equal_matrices(DX, DX1))
    self.assertTrue(equal_matrices(DZ, DZ1))
```

Listing 7 Checking the derivation in (13) using SymPy. The derivation is checked for an arbitrary vector z, but with a concrete value for its size K. Note that $e^z \cdot 1_\mathrm{K}$ corresponds to `rows_sum(exp(z))` according to Table 1.

```
K = 3
# create a symbolic 1 x K vector
z = Matrix(symarray('z', (1, K), real=True))
y = softmax(z)
e = exp(z)
# rows_sum returns a 1 x 1 Matrix, so we extract the scalar element
f = rows_sum(exp(z))[0, 0]
self.assertTrue(equal_matrices(softmax(z).jacobian(z), Diag(e) / f - (e.T * e) / (f * f)))
self.assertTrue(equal_matrices(softmax(z).jacobian(z), Diag(y) - y.T * y))
```

5.2 Implementation details

In the implementation of MLPs, one has to take into account that some functions are prone to numerical errors. In this section we list a few of them, and discuss how to solve those issues. Furthermore, we discuss the runtime efficiency of the implementation.

The softmax function in (10) is prone to underflow and overflow. In practice an equivalent, but numerically more stable version called stable-softmax is therefore used, see e.g., (Blanchard et al., 2020). For single inputs we have

$$\mathsf{stable\text{-}softmax}(z) = \frac{e^y}{e^y \cdot 1_{\mathrm{K}}} \tag{16}$$

and for multiple outputs

$$\mathsf{stable\text{-}softmax}(Z) = e^Y \odot \left(\frac{1}{e^Y \cdot 1_{\mathrm{K}}} \cdot 1_{\mathrm{K}}^\top \right), \tag{17}$$

where

$$\begin{cases} y = z - \max(z) \cdot 1_{\mathrm{K}}^\top \\ Y = Z - \max(Z)_{\mathrm{row}} \cdot 1_{\mathrm{K}}^\top. \end{cases}$$

Note that 16 is the stable version of equation 10 and 17 is the stable version of equation 12. Another example is the operation $X^{-1/2}$ that is needed for batch normalization (Appendix A). This operation is unstable for values near zero. A common solution for this problem is to compute $(X + \varepsilon)^{-1/2}$ instead, meaning that a small positive value ε is added[2] to the entries of X. Finally, the operation $\log(\sigma(X))$ in the logistic cross-entropy loss function (47) needs a special implementation. For this reason the functions inv_sqrt and log_sigmoid were added to the matrix operations in Table 1.

To create a multilayer perceptron implementation based on this paper, consider the following. Firstly, choose a suitable matrix library, and use it to implement and test the matrix operations in Table 1. Secondly, literally implement the layer equations given in Appendix A. This should result in a functionally correct implementation. However, for an efficient implementation the matrix library should support hardware acceleration, and preferably also techniques like lazy evaluation and removal of temporaries. The latter are needed to avoid explicit calculation of intermediate values like $1_{\mathrm{N}} \cdot b$ in the feedforward pass of a linear layer (5). Furthermore, profiling of the performance is needed. In particular, more complicated expressions, e.g., in batch normalization (40), may result in bottlenecks that can be avoided by rewriting them. The equations and the corresponding code as presented here do give good results for the platforms we tried (see next section), but there may be room for improvement.

6 Experimental Results

Our Python implementations are available as PyPI (Python Software Foundation, n.d.) packages `nerva-jax`, `nerva-numpy`, `nerva-sympy`, `nerva-tensorflow`, and `nerva-torch`, and on GitHub (Wesselink, 2023). To validate these implementations, we perform experiments with the MNIST (Deng, 2012) and CIFAR-10 (Krizhevsky, 2009) datasets. As a baseline we use a native PyTorch model derived from `nn.Module`, in combination with the `nn.CrossEntropyLoss` loss function. It is compared with six of our own implementations: our C++ implementation (Nerva-C++) and its Python bindings (Nerva-Python), and our Python implementations in PyTorch, TensorFlow, JAX and NumPy. Note that all our implementations are based on Table 1 with matrix-form operations.

[2] In Keras the value ε is a parameter of batch normalization.

Table 2: Performance comparison of our implementations using the MNIST and CIFAR-10 datasets. The baseline is a native implementation in PyTorch based on an `nn.Module`. In all cases an MLP is used with two hidden layers of sizes 1024 and 512. The results are for one epoch of training with batch size 100 and a constant learning rate of 0.01. The initial weights are the same for each experiment. The experiments were executed on an an Intel Core i7-6700 system with four CPU cores.

Tool	Dataset	Loss	Train Accuracy	Test Accuracy	Time (s)
PyTorch-Native	MNIST	0.204	0.940	0.939	3.518
Nerva-C++	MNIST	0.203	0.940	0.939	3.138
Nerva-Python	MNIST	0.204	0.940	0.939	3.507
Nerva-PyTorch	MNIST	0.204	0.940	0.939	4.909
Nerva-TensorFlow	MNIST	0.204	0.940	0.939	12.818
Nerva-JAX	MNIST	0.204	0.940	0.939	11.863
Nerva-NumPy	MNIST	0.204	0.940	0.939	10.630
PyTorch-Native	CIFAR-10	1.649	0.410	0.412	7.885
Nerva-C++	CIFAR-10	1.640	0.417	0.418	7.423
Nerva-Python	CIFAR-10	1.650	0.413	0.413	8.073
Nerva-PyTorch	CIFAR-10	1.643	0.415	0.417	11.277
Nerva-TensorFlow	CIFAR-10	1.657	0.407	0.407	18.941
Nerva-JAX	CIFAR-10	1.646	0.417	0.415	19.707
Nerva-NumPy	CIFAR-10	1.670	0.398	0.398	20.985

We use an MLP with two hidden layers of sizes 1024 and 512. The first two layers of the MLP use the ReLU activation function, while the last layer is a linear layer without activation function. Since both experiments are about classification tasks, we use softmax cross-entropy for the loss function. In all cases we have used our own stochastic gradient descent implementation, similar to Listing 2. The initial weights are generated using Xavier, and they are the same for each experiment. We measure the net training time of one epoch, loading of the dataset is not included.

The results of the experiments can be found in Table 2. In both cases our C++ implementation is slightly faster than the native PyTorch implementation. This should not come as a surprise, since the usage of a scripting language like Python usually incurs overhead. However, this demonstrates that an implementation in terms of our table with matrix operations does not necessarily lead to a performance loss. Our reference implementation in PyTorch is about 40% slower than a native PyTorch script that uses a `torch.nn.Module`. This can be seen as the price one has to pay for an explicit implementation of backpropagation in Python. The NumPy, TensorFlow and JAX implementations are significantly slower. The most likely explanation for this is that they are not using a highly optimized library like Intel MKL for matrix computations on the CPU.

7 Discussion and Conclusions

This paper presents an implementation of multilayer perceptrons (MLPs) with a focus on clarity and transparency. We provide detailed specifications of all the necessary equations in explicit batch matrix-form, which ease implementation and facilitate analysis. The mathematical equations are realized through a small, consistent set of matrix primitives, summarized in tables that bridge

formal equations and code. This uniform approach contributes to more standardized, maintainable, and transparent implementations of neural networks.

Implementations were made in popular Python frameworks, including PyTorch, TensorFlow, and JAX, and in C++ using Eigen combined with the Intel MKL library. All these implementations adhere closely to the mathematical specifications, ensuring correctness and consistency across frameworks.

Our Python implementations fully expose the internal execution of multilayer perceptrons, including all steps of backpropagation. Care has been taken to make the code as simple and readable as possible, in order to provide a valuable resource for students and researchers. A large part of the equations has been derived and validated symbolically using the SymPy library, which accelerates development and ensures mathematical correctness. Experiments with our PyTorch implementation show that it runs approximately 40% slower than a native PyTorch `nn.Module` implementation. For research and educational use, this is a reasonable trade-off for gaining full control and transparency over every computational detail.

We provide an efficient C++ implementation as part of the Nerva library. Initial experiments indicate that this approach yields competitive performance. In general, the Eigen library produces efficient code for matrix expressions, though some expressions required slight reformulation to ensure forwarding of matrix products to optimized MKL routines. The structured, equation-driven approach advocated in this paper is valuable for understanding and improving computational efficiency, and it naturally extends to other backends, such as GPUs.

A particularly demanding domain for efficient implementation is that of sparse neural networks. Current mainstream frameworks typically emulate sparsity by applying binary masks to dense tensors, which provides no real savings in memory or computation. In contrast, our C++ backend implements true sparse matrix operations, guided by the same batch matrix-form equations used for dense networks. This makes it possible to analyze correctness and performance at the level of individual matrix operations and to identify bottlenecks in Eigen and MKL. As shown in our previous work (Wesselink et al., 2024), this approach enables targeted optimizations and substantial speedups at high sparsity levels. Explicit matrix-form backpropagation thus forms a solid foundation not only for transparent dense implementations, but also for scaling truly sparse architectures.

In summary, this work contributes: (1) a complete derivation of batch matrix-form backpropagation for MLPs, (2) symbolic validation of all gradient equations using SymPy, (3) uniform implementations across major Python frameworks and a high-performance C++ backend, and (4) demonstration of how explicit matrix formulations enable structured reasoning and efficient sparse computation. Together, these results provide a validated, extensible foundation for both education and research in neural network implementation.

The scope of this paper is limited to standard neural network layers. Future extensions may include convolutional, pooling, and transformer layers, continuing the same philosophy of explicit matrix-form specification, symbolic validation, and uniform implementation across frameworks. The minimal set of matrix primitives developed here provides a basis for such extensions. Further work may also explore GPU implementations and distributed computations, where explicit control over matrix operations can guide new optimizations for both dense and sparse networks.

References

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath

Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation*, pages 265–283, 2016.

Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of Machine Learning Research*, 18:153:1–153:43, 2017. URL `https://jmlr.org/papers/v18/17-468.html`.

Pierre Blanchard, Desmond J Higham, and Nicholas J Higham. Accurately computing the log-sum-exp and softmax functions. *IMA Journal of Numerical Analysis*, 41(4):2311–2330, 08 2020. 10.1093/imanum/draa038.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL `https://github.com/google/jax`.

François Chollet et al. Keras. `https://keras.io`, 2015.

Li Deng. The MNIST database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012. 10.1109/MSP.2012.2211477.

Shiv Ram Dubey, Satish Kumar Singh, and Bidyut Baran Chaudhuri. Activation functions in deep learning: A comprehensive survey and benchmark. *Neurocomputing*, 503:92–108, 2022. 10.1016/J.NEUCOM.2022.06.111.

K. Fukushima. Cognitron: a self-organizing multilayered neural network. *Biological Cybernetics*, 20: 121–136, 1975.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, 2010. URL `https://proceedings.mlr.press/v9/glorot10a.html`.

Gaël Guennebaud, Benoît Jacob, et al. Eigen v3, 2010. URL `https://eigen.tuxfamily.org`.

Charles R. Harris et al. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020. 10.1038/s41586-020-2649-2.

Soufiane Hayou, Arnaud Doucet, and Judith Rousseau. On the selection of initialization and activation function for deep neural networks. *arXiv preprint arXiv:1805.08266*, 2018. URL `https://arxiv.org/abs/1805.08266`.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 1026–1034, 2015. 10.1109/ICCV.2015.123.

Geoffrey Hinton, Li Deng, Dong Yu, George E. Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N. Sainath, and Brian Kingsbury. Deep

neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012. 10.1109/MSP.2012.2205597.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*, volume 37 of *JMLR Workshop and Conference Proceedings*, pages 448–456, 2015. URL https://proceedings.mlr.press/v37/ioffe15.html.

Henry J Kelley. Gradient theory of optimal flight paths. *Ars Journal*, 30(10):947–954, 1960.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*, 2015. URL https://arxiv.org/abs/1412.6980.

Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009. URL https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf.

Shiwei Liu and Zhangyang Wang. Ten lessons we have learned in the new "sparseland": A short handbook for sparse neural network researchers. *arXiv preprint arXiv:2302.02596*, 2023. 10.48550/arXiv.2302.02596.

Andrew L. Maas, Awni Y. Hannun, and Andrew Y. Ng. Rectifier nonlinearities improve neural network acoustic models. In *ICML Workshop on Deep Learning for Audio, Speech and Language Processing*, volume 30, page 3, 2013.

Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. Sympy: symbolic computing in Python. *PeerJ Computer Science*, 3:e103, 2017. ISSN 2376-5992. 10.7717/peerj-cs.103.

Meenal V. Narkhede, Prashant P. Bartakke, and Mukul S. Sutaone. A review on weight initialization strategies for neural networks. *Artificial Intelligence Review*, 55(1):291–322, 2022. 10.1007/S10462-021-10033-Z.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035, 2019. URL https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html.

Python Software Foundation. Python package index - PyPI, n.d. URL https://pypi.org/.

Nitish Srivastava, Geoffrey E. Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014. URL https://jmlr.org/papers/v15/srivastava14a.html.

Li Wan, Matthew D. Zeiler, Sixin Zhang, Yann LeCun, and Rob Fergus. Regularization of neural networks using dropconnect. In *International Conference on Machine Learning*, volume 28 of

JMLR Workshop and Conference Proceedings, pages 1058–1066, 2013. URL https://proceedings.mlr.press/v28/wan13.html.

Wieger Wesselink. The Nerva Libraries, March 2023. URL https://github.com/wiegerw/nerva.

Wieger Wesselink, Bram Grooten, Qiao Xiao, Cassio de Campos, and Mykola Pechenizkiy. Nerva: A truly sparse implementation of neural networks. *arXiv preprint arXiv:2407.17437*, 2024. URL https://arxiv.org/abs/2407.17437. Presented at the ICLR 2023 Workshop on Sparsity in Neural Networks.

Yanzhao Wu, Ling Liu, Juhyun Bae, Ka Ho Chow, Arun Iyengar, Calton Pu, Wenqi Wei, Lei Yu, and Qi Zhang. Demystifying learning rate policies for high accuracy training of deep neural networks. In *IEEE International Conference on Big Data*, pages 1971–1980, 2019. 10.1109/BIGDATA47090.2019.9006104.

Chris Yeh. Deriving Batch-Norm Backprop Equations, 2017. URL https://chrisyeh96.github.io/2017/08/28/deriving-batchnorm-backprop.html.

Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. *Dive into Deep Learning*. Cambridge University Press, Cambridge, United Kingdom, 2024. URL https://D2L.ai.

A Layer equations

A major contribution of this paper is the following overview of the feedforward and backpropagation equations of layers **in matrix-form**. For each of the equations, the corresponding Python implementation is given. For several equations a derivation is included, while the correctness of the equations and derivations has been validated using SymPy. All layers are implemented in our Nerva Python packages, and a validation of all equations and derivations can be found in the `tests` directory of the `nerva-sympy` package. We consider input batches X in row layout, i.e., each row represents a single example. We use the following notations:

- $X \in \mathbb{R}^{\mathrm{N} \times \mathrm{D}}$ is the input batch, where N is the number of examples and D is the input dimension.
- $Y \in \mathbb{R}^{\mathrm{N} \times \mathrm{K}}$ is the output batch, where K is the output dimension.
- $W \in \mathbb{R}^{\mathrm{K} \times \mathrm{D}}$ is the weight matrix, which maps the input features to the output features.
- $b \in \mathbb{R}^{1 \times \mathrm{K}}$ is the bias vector.
- $Z \in \mathbb{R}^{\mathrm{N} \times \mathrm{K}}$ is a matrix with intermediate values.
- $\beta, \gamma, \Sigma \in \mathbb{R}^{1 \times \mathrm{K}}$ are the parameters of batch normalization.
- $R \in \mathbb{R}^{\mathrm{K} \times \mathrm{D}}$ is a dropout matrix.

Each of these matrices has a gradient with the same dimensions, denoted using the same symbol preceded by D, e.g., $\mathsf{D}X$ is the gradient corresponding to X. The implementation uses the same names. The input `X`, and layer parameters like `W`, `b` and their gradients `DX`, `DW`, `Db` are stored in the attributes of a layer. The only exception is the output `Y` and its gradient `DY`, which are stored in the `X` and `DX` attributes of the next layer. Fig. 3 explains this in more detail.

linear layer

FEEDFORWARD EQUATIONS

$$Y = XW^\top + 1_\mathrm{N} \cdot b$$

```
Y = X * W.T + row_repeat(b, N)
```

BACKPROPAGATION EQUATIONS

$$\begin{aligned} \mathsf{D}W &= \mathsf{D}Y^\top \cdot X \\ \mathsf{D}b &= 1_\mathrm{N}^\top \cdot \mathsf{D}Y \\ \mathsf{D}X &= \mathsf{D}Y \cdot W \end{aligned}$$

```
DW = DY.T * X
Db = columns_sum(DY)
DX = DY * W
```

activation layer Let $\mathsf{act} : \mathbb{R} \to \mathbb{R}$ be an activation function, for example relu.

FEEDFORWARD EQUATIONS

$$Z = XW^\top + 1_\mathrm{N} \cdot b$$
$$Y = \mathsf{act}(Z)$$

BACKPROPAGATION EQUATIONS

$$\mathsf{D}Z = \mathsf{D}Y \odot \mathsf{act}'(Z)$$
$$\mathsf{D}W = \mathsf{D}Z^\top \cdot X$$
$$\mathsf{D}b = 1_\mathrm{N}^\top \cdot \mathsf{D}Z$$
$$\mathsf{D}X = \mathsf{D}Z \cdot W$$

```
Z = X * W.T + row_repeat(b, N)
Y = act(Z)
```

```
DZ = hadamard(DY, act.gradient(Z))
DW = DZ.T * X
Db = columns_sum(DZ)
DX = DZ * W
```

softmax layer

FEEDFORWARD EQUATIONS

$$Z = XW^\top + 1_\mathrm{N} \cdot b$$
$$Y = \mathsf{softmax}(Z)$$

BACKPROPAGATION EQUATIONS

$$\mathsf{D}Z = Y \odot (\mathsf{D}Y - \mathsf{diag}(\mathsf{D}Y \cdot Y^\top) \cdot 1_\mathrm{K}^\top)$$
$$\mathsf{D}W = \mathsf{D}Z^\top \cdot X$$
$$\mathsf{D}b = 1_\mathrm{N}^\top \cdot \mathsf{D}Z$$
$$\mathsf{D}X = \mathsf{D}Z \cdot W$$

```
Z = X * W.T + row_repeat(b, N)
Y = softmax(Z)
```

```
DZ = hadamard(Y, DY - column_repeat(diag(DY * Y.T),
    K))
DW = DZ.T * X
Db = columns_sum(DZ)
DX = DZ * W
```

log-softmax layer

FEEDFORWARD EQUATIONS

$$Z = XW^\top + 1_\mathrm{N} \cdot b$$
$$Y = \mathsf{logsoftmax}(Z)$$

BACKPROPAGATION EQUATIONS

$$\mathsf{D}Z = \mathsf{D}Y - \mathsf{softmax}(Z) \odot (\mathsf{D}Y \cdot 1_\mathrm{K} \cdot 1_\mathrm{K}^\top)$$
$$\mathsf{D}W = \mathsf{D}Z^\top \cdot X$$
$$\mathsf{D}b = 1_\mathrm{N}^\top \cdot \mathsf{D}Z$$
$$\mathsf{D}X = \mathsf{D}Z \cdot W$$

```
Z = X * W.T + row_repeat(b, N)
Y = log_softmax(Z)
```

```
DZ = DY - hadamard(softmax(Z), column_repeat(
    rows_sum(DY), K))
DW = DZ.T * X
Db = columns_sum(DZ)
DX = DZ * W
```

batch normalization layer

FEEDFORWARD EQUATIONS

$$
\begin{aligned}
R &= X - \frac{1_{\mathrm{N}} \cdot 1_{\mathrm{N}}^{\top}}{\mathrm{N}} \cdot X \\
\Sigma &= \frac{1}{\mathrm{N}} \cdot \mathsf{diag}(R^{\top} R)^{\top} \\
Z &= (1_{\mathrm{N}} \cdot \Sigma^{-\frac{1}{2}}) \odot R \\
Y &= (1_{\mathrm{N}} \cdot \gamma) \odot Z + 1_{\mathrm{N}} \cdot \beta
\end{aligned}
$$

```
R = X - row_repeat(columns_mean(X), N)
Sigma = diag(R.T * R).T / N
inv_sqrt_Sigma = inv_sqrt(Sigma)
Z = hadamard(row_repeat(inv_sqrt_Sigma, N), R)
Y = hadamard(row_repeat(gamma, N), Z) + row_repeat(
    beta, N)
```

BACKPROPAGATION EQUATIONS

$$
\begin{aligned}
\mathsf{D}Z &= (1_{\mathrm{N}} \cdot \gamma) \odot \mathsf{D}Y \\
\mathsf{D}\beta &= 1_{\mathrm{N}}^{\top} \cdot \mathsf{D}Y \\
\mathsf{D}\gamma &= 1_{\mathrm{N}}^{\top} \cdot (Z \odot \mathsf{D}Y) \\
\mathsf{D}X &= \left(\frac{1}{\mathrm{N}} \cdot 1_{\mathrm{N}} \cdot \Sigma^{-\frac{1}{2}} \right) \\
&\quad \odot \Big((\mathrm{N} \cdot \mathbb{I}_{\mathrm{N}} - 1_{\mathrm{N}} \cdot 1_{\mathrm{N}}^{\top}) \cdot \mathsf{D}Z \\
&\qquad - Z \odot (1_{\mathrm{N}} \cdot \mathsf{diag}(Z^{\top} \cdot \mathsf{D}Z)^{\top}) \Big)
\end{aligned}
$$

```
DZ = hadamard(row_repeat(gamma, N), DY)
Dbeta = columns_sum(DY)
Dgamma = columns_sum(hadamard(DY, Z))
DX = hadamard(row_repeat(inv_sqrt_Sigma / N, N), (N
    * identity(N) - ones(N, N)) * DZ - hadamard(Z,
    row_repeat(diag(Z.T * DZ).T, N)))
```

linear dropout layer

FEEDFORWARD EQUATIONS

$$
Y = X(W \odot R)^{\top} + 1_{\mathrm{N}} \cdot b
$$

```
Y = X * hadamard(W, R).T + row_repeat(b, N)
```

BACKPROPAGATION EQUATIONS

$$
\begin{aligned}
\mathsf{D}W &= (\mathsf{D}Y^{\top} \cdot X) \odot R \\
\mathsf{D}b &= 1_{\mathrm{N}}^{\top} \cdot \mathsf{D}Y \\
\mathsf{D}X &= \mathsf{D}Y(W \odot R)
\end{aligned}
$$

```
DW = hadamard(DY.T * X, R)
Db = columns_sum(DY)
DX = DY * hadamard(W, R)
```

activation dropout layer

FEEDFORWARD EQUATIONS

$$
\begin{aligned}
Z &= X(W \odot R)^{\top} + 1_{\mathrm{N}} \cdot b \\
Y &= \mathsf{act}(Z)
\end{aligned}
$$

```
Z = X * hadamard(W, R).T + row_repeat(b, N)
Y = act(Z)
```

BACKPROPAGATION EQUATIONS

$$
\begin{aligned}
\mathsf{D}Z &= \mathsf{D}Y \odot \mathsf{act}'(Z) \\
\mathsf{D}W &= (\mathsf{D}Z^{\top} \cdot X) \odot R \\
\mathsf{D}b &= 1_{\mathrm{N}}^{\top} \cdot \mathsf{D}Z \\
\mathsf{D}X &= \mathsf{D}Z(W \odot R)
\end{aligned}
$$

```
DZ = hadamard(DY, act.gradient(Z))
DW = hadamard(DZ.T * X, R)
Db = columns_sum(DZ)
DX = DZ * hadamard(W, R)
```

A.1 Derivations

In this section we give some derivations of the backpropagation equations. We give some applications of the product rule and chain rule for vector functions, and show how some properties on the rows and columns of a matrix can be generalized into matrix-form.

Lemma 1 (Product Rule for vector functions)**.** *The product rule for scalar functions u and v is given by*

$$(u \cdot v)' = u' \cdot v + u \cdot v'.$$

It can be generalized to vector functions, but the result is sensitive to the orientation of the operands. Below we give four concrete applications of the product rule for vector functions. Let $x \in \mathbb{R}^p$, $A \in \mathbb{R}^{m \times n}$, and $h(x) = f(x)g(x)$ for $m, n, p \in \mathbb{N}^+$.

$$\text{Let } f(x) \in \mathbb{R}^{n \times 1}, \text{ and let } g(x) \in \mathbb{R}, \text{ then } \frac{\partial h}{\partial x} = \frac{\partial f}{\partial x} g + f \frac{\partial g}{\partial x}. \tag{18}$$

$$\text{Let } f(x) \in \mathbb{R}^{1 \times n}, \text{ and let } g(x) \in \mathbb{R}, \text{ then } \frac{\partial h}{\partial x} = \frac{\partial f}{\partial x} g + f^\top \frac{\partial g}{\partial x}. \tag{19}$$

$$\text{Let } f(x) = A, \text{ and let } g(x) \in \mathbb{R}^{n \times 1}, \text{ then } \frac{\partial h}{\partial x} = A \frac{\partial g}{\partial x}. \tag{20}$$

$$\text{Let } f(x) \in \mathbb{R}^{1 \times m}, \text{ and let } g(x) = A, \text{ then } \frac{\partial h}{\partial x} = A^\top \frac{\partial f}{\partial x}. \tag{21}$$

Lemma 2 (Chain rule for vector functions)**.** *Let $f : \mathbb{R}^n \to \mathbb{R}^m$, let $g : \mathbb{R}^m \to \mathbb{R}^p$, and let $h(x) = g(y)$ with $y = f(x)$. Then we have*

$$\frac{\partial h}{\partial x} = \frac{\partial g}{\partial y} \cdot \frac{\partial f}{\partial x} \tag{22}$$

Note that this equations holds irrespective of whether $f(x)$ is a row or a column vector.

Property 1 (Matrix properties)**.** *Let $X, Y, Z \in \mathbb{R}^{m \times n}$. We denote the i^{th} row of matrix A as a_i and the j^{th} column of matrix A as a^j. The element at position (i, j) of A is denoted as a_{ij}. Below we give a number of properties on the rows and columns of matrices, and the generalization of these properties into matrix-form.*

$$\text{If } z^j = x^j \cdot (x^j)^\top \cdot y^j \quad (1 \leq j \leq n), \text{ then } Z = X \odot (1_m \cdot \mathit{diag}(X^\top \cdot Y)^\top). \tag{23}$$

$$\text{If } z^j = 1_m \cdot (x^j)^\top \cdot y^j \quad (1 \leq j \leq n), \text{ then } Z = 1_m \cdot \mathit{diag}(X^\top Y)^\top. \tag{24}$$

$$\text{If } z^j = x^j \cdot 1_m^\top \cdot y^j \quad (1 \leq j \leq n), \quad \text{then } Z = X \odot (1_m \cdot 1_m^\top \cdot Y) \tag{25}$$

$$\text{If } z_i = x_i \cdot y_i^\top \cdot y_i \quad (1 \leq i \leq m), \quad \text{then } Z = (\mathit{diag}(X \cdot Y^\top) \cdot 1_n^\top) \odot Y \tag{26}$$

$$\text{If } z_i = x_i \cdot y_i^\top \cdot 1_n^\top \quad (1 \leq i \leq m), \quad \text{then } Z = \mathit{diag}(X \cdot Y^\top) \cdot 1_n^\top \tag{27}$$

$$\text{If } z_i = x_i \cdot 1_n \cdot y_i \quad (1 \leq i \leq m), \quad \text{then } Z = (X \cdot 1_n \cdot 1_n^\top) \odot Y \tag{28}$$

All properties have been validated with SymPy. Naturally there is a lot of symmetry between the row and column properties. By means of example we will prove equation (26). From $z_i = x_i \cdot y_i^\top \cdot y_i$

we derive that $z_{ij} = (x_i \cdot y_i^\top) y_{ij}$ for $1 \leq j \leq n$. Hence we can write $Z = R \odot Y$, where R is defined using $r_{ij} = (x_i \cdot y_i^\top)$. We observe that $\mathsf{diag}(X \cdot Y^\top) = (x_1 \cdot y_1^\top, \ldots, x_m \cdot y_m^\top)^\top$. From the definition of R we can see that it consists of n copies of the column vector $\mathsf{diag}(X \cdot Y^\top)$. Hence we have $R = \mathsf{diag}(X \cdot Y^\top) \cdot 1_n^\top$.

linear layer We have $Y = X \cdot W^\top + 1_\mathrm{N} \cdot b$. Let x_i, y_i, b_i be the i^{th} row of X, Y, $1_\mathrm{N} \cdot b$ for $1 \leq i \leq \mathrm{N}$, hence $y_i = x_i W^\top + b_i$, where $b_i = b$. Furthermore, let w_j be the j^{th} row of W, and let e^i be the i^{th} column of the unit matrix $\mathbb{I}_\mathrm{K}$. Let $\mathcal{L}(Y) = \sum_{i=1}^{\mathrm{N}} \mathcal{L}(y_i)$ be the corresponding loss. We calculate

$$\frac{\partial \mathcal{L}}{\partial x_i} \overset{(22)}{=} \sum_{n=1}^{\mathrm{N}} \frac{\partial \mathcal{L}}{\partial y_n} \frac{\partial y_n}{\partial x_i} = \frac{\partial \mathcal{L}}{\partial y_i} \frac{\partial y_i}{\partial x_i} \overset{(21)}{=} \frac{\partial \mathcal{L}}{\partial y_i} \cdot W, \text{ hence } \mathsf{D}x_i = \mathsf{D}y_i \cdot W \tag{29}$$

$$\frac{\partial \mathcal{L}}{\partial b} \overset{(22)}{=} \sum_{n=1}^{\mathrm{N}} \frac{\partial \mathcal{L}}{\partial y_n} \frac{\partial y_n}{\partial b} \overset{(21)}{=} \sum_{n=1}^{\mathrm{N}} \frac{\partial \mathcal{L}}{\partial y_n} \cdot \mathbb{I}_K = \sum_{n=1}^{\mathrm{N}} \frac{\partial \mathcal{L}}{\partial y_n}, \text{ hence } \mathsf{D}b = 1_\mathrm{N}^\top \cdot \mathsf{D}Y \tag{30}$$

For the gradient of W we calculate the derivative with respect to an arbitrary entry w_{ij}. From the definition of Y we derive

$$\frac{\partial y_{nk}}{\partial w_{ij}} = \frac{\partial (\sum_{d=1}^{D} x_{nd}\, w_{kd} + b_k)}{\partial w_{ij}} = \begin{cases} x_{nj}, & \text{if } k = i, \\ 0, & \text{if } k \neq i. \end{cases}$$

Using this, we calculate

$$\frac{\partial \mathcal{L}}{\partial w_{ij}} = \sum_{n=1}^{N} \sum_{k=1}^{K} \frac{\partial \mathcal{L}}{\partial y_{nk}} \frac{\partial y_{nk}}{\partial w_{ij}} = \sum_{n=1}^{N} \frac{\partial \mathcal{L}}{\partial y_{ni}}\, x_{nj}. \tag{31}$$

The equations 29, 30 and 31 can be generalized to matrix equations: $\mathsf{D}X = \mathsf{D}Y \cdot W$, $\mathsf{D}b = 1_\mathrm{N}^\top \cdot \mathsf{D}Y$, and $\mathsf{D}W = \mathsf{D}Y^\top \cdot X$.

log-softmax layer We have $Y = \mathsf{log\text{-}softmax}(Z)$. Let y_i, z_i be the i^{th} row of Y, Z for $1 \leq i \leq \mathrm{N}$, hence $y_i = \mathsf{log\text{-}softmax}(z_i)$. We calculate

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial z_i} &\overset{(22)}{=} \frac{\partial \mathcal{L}}{\partial y_i} \frac{\partial}{\partial z_i} \mathsf{log\text{-}softmax}(z_i) \overset{(43)}{=} \frac{\partial \mathcal{L}}{\partial y_i} (\mathbb{I}_K - 1_\mathrm{K} \cdot \mathsf{softmax}(z_i)) \\ &\iff \mathsf{D}z_i = \mathsf{D}y_i \cdot (\mathbb{I}_K - 1_\mathrm{K} \cdot \mathsf{softmax}(z_i)) = \mathsf{D}y_i - \mathsf{D}y_i \cdot 1_\mathrm{K} \cdot \mathsf{softmax}(z_i). \end{aligned}$$

This can be generalized to matrices using property 28:

$$\mathsf{D}Z = \mathsf{D}Y - \mathsf{softmax}(Z) \odot (\mathsf{D}Y \cdot 1_\mathrm{K} \cdot 1_\mathrm{K}^\top) \tag{32}$$

batch normalization layer We will derive the equation for $\mathsf{D}X$, which is the most complicated one. Following the approach of (Yeh, 2017), we first derive the equations for a single column. Let $x^j, r^j, z^j \in \mathbb{R}^{\mathrm{N} \times 1}$ be the j^{th} column of X, R and Z, and let $\sigma \in \mathbb{R}$ be the j^{th} element of Σ, with $1 \leq j \leq \mathrm{D}$. Then we obtain the following equations:

$$r^j = x^j - \frac{1_\mathrm{N} \cdot 1_\mathrm{N}^\top}{\mathrm{N}} \cdot x^j = (\mathbb{I}_\mathrm{N} - \frac{1_\mathrm{N} \cdot 1_\mathrm{N}^\top}{\mathrm{N}}) \cdot x^j \tag{33}$$

$$\sigma = \frac{(r^j)^\top r^j}{\mathrm{N}} \tag{34}$$

$$z^j = r^j \cdot \sigma^{-\frac{1}{2}} \tag{35}$$

We calculate

$$\frac{\partial z^j}{\partial r^j} \overset{(18)}{=} \frac{\partial r^j}{\partial r^j} \cdot \sigma^{-\frac{1}{2}} + r^j \cdot \frac{\partial \sigma^{-\frac{1}{2}}}{\partial r^j} \overset{(22)}{=} \sigma^{-\frac{1}{2}} - r^j \cdot \frac{\sigma^{-\frac{3}{2}}}{2} \cdot \frac{\partial \sigma}{\partial r^j} = \sigma^{-\frac{1}{2}} - r^j \cdot \frac{\sigma^{-\frac{3}{2}}}{2} \cdot \left(\frac{2(r^j)^\top}{\mathrm{N}} \right) \tag{36}$$

$$= \frac{\sigma^{-\frac{1}{2}}}{\mathrm{N}} \left(\mathrm{N} \cdot \mathbb{I}_\mathrm{N} - \sigma^{-1} r^j (r^j)^\top \right) = \frac{\sigma^{-\frac{1}{2}}}{\mathrm{N}} \cdot \left(\mathrm{N} \cdot \mathbb{I}_\mathrm{N} - z^j (z^j)^\top \right). \tag{37}$$

Using the chain rule we find

$$\frac{\partial \mathcal{L}}{\partial r^j} \overset{(22)}{=} \frac{\partial \mathcal{L}}{\partial z^j} \frac{\partial z^j}{\partial r^j}, \text{ hence } \mathsf{D}r^j = \frac{\sigma^{-\frac{1}{2}}}{\mathrm{N}} \left(\mathrm{N} \cdot \mathbb{I}_\mathrm{N} - z^j (z^j)^\top \right) \cdot \mathsf{D}z^j \tag{38}$$

$$\frac{\partial \mathcal{L}}{\partial x^j} \overset{(22)}{=} \frac{\partial \mathcal{L}}{\partial r^j} \frac{\partial r^j}{\partial x^j}, \text{ hence } \mathsf{D}x^j = (\mathbb{I}_\mathrm{N} - \frac{1_\mathrm{N} \cdot 1_\mathrm{N}^\top}{\mathrm{N}}) \cdot \mathsf{D}r^j, \tag{39}$$

where one needs to take into account that for a column vector x we have $\mathsf{D}x = (\frac{\partial \mathcal{L}}{\partial x})^\top$. Hence

$$\begin{aligned}
\mathsf{D}x^j &= \frac{\sigma^{-\frac{1}{2}}}{\mathrm{N}} \cdot (\mathbb{I}_\mathrm{N} - \frac{1_\mathrm{N} \cdot 1_\mathrm{N}^\top}{\mathrm{N}}) \cdot \left(\mathrm{N} \cdot \mathbb{I}_\mathrm{N} - z^j \cdot (z^j)^\top \right) \cdot \mathsf{D}z^j \\
&= \frac{\sigma^{-\frac{1}{2}}}{\mathrm{N}} \cdot (\mathrm{N} \cdot \mathbb{I}_\mathrm{N} - z^j \cdot (z^j)^\top - 1_\mathrm{N} \cdot 1_\mathrm{N}^\top + \frac{1_\mathrm{N} \cdot 1_\mathrm{N}^\top \cdot z^j \cdot (z^j)^\top}{\mathrm{N}}) \cdot \mathsf{D}z^j \\
&\quad \{ \text{ In batch normalization the column sum } 1_\mathrm{N}^\top \cdot z^j \text{ evaluates to zero. } \} \\
&= \frac{\sigma^{-\frac{1}{2}}}{\mathrm{N}} \cdot ((\mathrm{N} \cdot \mathbb{I}_\mathrm{N} - 1_\mathrm{N} \cdot 1_\mathrm{N}^\top) \cdot \mathsf{D}z^j - z^j \cdot (z^j)^\top \cdot \mathsf{D}z^j)
\end{aligned}$$

Using property (23) we generalize this into matrix-form:

$$\mathsf{D}X = (\frac{1}{\mathrm{N}} \cdot 1_\mathrm{N} \cdot \Sigma^{-\frac{1}{2}}) \odot \left((\mathrm{N} \cdot \mathbb{I}_\mathrm{N} - 1_\mathrm{N} \cdot 1_\mathrm{N}^\top) \cdot \mathsf{D}Z - Z \odot (1_\mathrm{N} \cdot \mathsf{diag}(Z^\top \cdot \mathsf{D}Z)^\top) \right). \tag{40}$$

For the remaining equations, let $y_i, z_i, \beta_i, \gamma_i$ be the i^{th} row of Y, Z, $1_\mathrm{N} \cdot \beta$, and $1_\mathrm{N} \cdot \gamma$ for $1 \leq i \leq \mathrm{N}$, where $\beta_i = \beta$, and $\gamma_i = \gamma$. Hence $y_i = \gamma_i \odot z_i + \beta_i$. From this it follows

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \beta_i} &= \frac{\partial \mathcal{L}}{\partial y_i} \\
\frac{\partial \mathcal{L}}{\partial \gamma_i} &= \frac{\partial \mathcal{L}}{\partial y_i} \odot z_i,
\end{aligned}$$

which we can generalize into matrix-form: $\mathsf{D}\beta = 1_\mathrm{N}^\top \cdot \mathsf{D}Y$ and $\mathsf{D}\gamma = 1_\mathrm{N}^\top \cdot (\mathsf{D}Y \odot Z)$.

B Implementation of matrix operations

In Table 3 we provide implementations in several frameworks of the fundamental matrix-form operations for multilayer perceptrons. See Table 1 in the main body of the paper for the corresponding equations and definitions.

Operation	NumPy + JAX	PyTorch	TensorFlow	Eigen
zeros(m,n)	zeros((m,n))	zeros(m,n)	zeros([m,n])	Zero(m,n)
ones(m,n)	ones((m,n))	ones(m,n)	ones([m,n])	Ones(m,n)
identity(n)	eye(n)	eye(n)	eye(n)	Identity(n,n)
X.T	X.T	X.T	X.T	X.transpose()
c * X	c * X	c * X	c * X	c * X
X + Y	X + Y	X + Y	X + Y	X + Y
X - Y	X - Y	X - Y	X - Y	X - Y
X * Z	X @ Z	X @ Z	X @ Z	X * Z
hadamard(X,Y)	X * Y	X * Y	multiply(X,Y)	X.array() * Y.array()
diag(X)	diag(X)	diag(X)	diag_part(X)	X.diagonal()
Diag(x)	diag(x)	diag(x.flatten())	diag(reshape(x,[-1]))	x.asDiagonal()
elements_sum(X)	sum(X)	sum(X)	reduce_sum(X)	X.sum()
column_repeat(x,n)	tile(x,(1,n))	x.repeat(1,n)	tile(x,[1,n])	x.replicate(1,n)
row_repeat(x,m)	tile(x,(m,1))	x.repeat(m,1)	tile(x,[m,1])	x.replicate(m,1)
columns_sum(X)	sum(X,axis=0)	sum(X,dim=0)	reduce_sum(X,axis=0)	X.colwise().sum()
rows_sum(X)	sum(X,axis=1)	sum(X,dim=1)	reduce_sum(X,axis=1)	X.rowwise().sum()
columns_max(X)	max(X,axis=0)	max(X,dim=0).values	reduce_max(X,axis=0)	X.colwise().maxCoeff()
rows_max(X)	max(X,axis=1)	max(X,dim=1).values	reduce_max(X,axis=1)	X.rowwise().maxCoeff()
columns_mean(X)	mean(X,axis=0)	mean(X,dim=0)	reduce_mean(X,axis=0)	X.colwise().mean()
rows_mean(X)	mean(X,axis=1)	mean(X,dim=1)	reduce_mean(X,axis=1)	X.rowwise().mean()
apply(f,X)	f(X)	f(X)	f(X)	X.unaryExpr(f)
exp(X)	exp(X)	exp(X)	exp(X)	X.array().exp()
log(X)	log(X)	log(X)	log(X)	X.array().log()
reciprocal(X)	1 / X	1 / X	reciprocal(X)	X.array().inverse()
sqrt(X)	sqrt(X)	sqrt(X)	sqrt(X)	X.array().sqrt()
inv_sqrt(X)	reciprocal(sqrt(X+e))	reciprocal(sqrt(X+e))	reciprocal(sqrt(X+e))	reciprocal(sqrt(X.array()+e))
log_sigmoid(X)	-logaddexp(0,-X)	-softplus(-X)	-softplus(-X)	-log1p(exp(-X.array()))

Table 3: Implementation of matrix operations in NumPy, JAX, PyTorch, TensorFlow and Eigen. We assume that e is a given small positive constant that is used to avoid division by zero. Note that some of the operations are located in Python submodules.

C Activation functions

In this section we give an overview of some commonly used univariate activation functions. These activation functions are typically applied element-wise to the output matrix Y of a neural network.

NAME	FUNCTION	DERIVATIVE
ReLU (Fukushima, 1975)	$\mathsf{relu}(x) = \max(0, x)$	$\mathsf{relu}'(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{otherwise} \end{cases}$
Leaky ReLU (Maas et al., 2013)	$\mathsf{leaky\text{-}relu}(x) = \max(\alpha x, x)$	$\mathsf{leaky\text{-}relu}'(x) = \begin{cases} \alpha & \text{if } x < \alpha x \\ 1 & \text{otherwise} \end{cases}$
Hyperbolic tangent	$\mathsf{tanh}(x) = \dfrac{e^x - e^{-x}}{e^x + e^{-x}}$	$\mathsf{tanh}'(x) = 1 - \mathsf{tanh}(x)^2$
Sigmoid / logistic function (Hinton et al., 2012)	$\sigma(x) = \dfrac{1}{1 + e^{-x}}$	$\sigma'(x) = \sigma(x)(1 - \sigma(x))$

Note that leaky ReLU has a parameter $\alpha \in \mathbb{R}$, usually $0 < \alpha < 1$.

C.1 Softmax functions

In this section we give an overview of softmax functions and their derivatives, both for single inputs and for batches. These equations are present in the activation function of softmax layers and in some of the loss functions in Appendix D. We use the same notation as before, where $x \in \mathbb{R}^{1\times \mathrm{D}}$ is a single input with dimension D, and $X \in \mathbb{R}^{\mathrm{N}\times \mathrm{D}}$ is an input batch, where N is the number of samples in the batch. We denote the rows of X as $x_1, \ldots, x_{\mathrm{N}}$.

Below an overview of softmax functions and their derivatives is given. Equations in matrix-form are only given if they are needed later on. Note that the function log-softmax is defined using

$$\textsf{log-softmax}(x) = \log(\textsf{softmax}(x)).$$

VECTOR FUNCTIONS	MATRIX FUNCTIONS
$\textsf{softmax}(x) = \dfrac{e^x}{e^x \cdot 1_{\mathrm{D}}}$	$\textsf{softmax}(X) = e^X \odot \left(\dfrac{1}{e^X \cdot 1_{\mathrm{D}}} \cdot 1_{\mathrm{D}}^\top\right)$
$\textsf{stable-softmax}(x) = \dfrac{e^z}{e^z \cdot 1_{\mathrm{D}}}$	$\textsf{stable-softmax}(X) = e^Z \odot \left(\dfrac{1}{e^Z \cdot 1_{\mathrm{D}}} \cdot 1_{\mathrm{D}}^\top\right)$
$\textsf{log-softmax}(x) = x - \log(e^x \cdot 1_{\mathrm{D}}) \cdot 1_{\mathrm{D}}^\top$	$\textsf{log-softmax}(X) = X - \log(e^X \cdot 1_{\mathrm{D}}) \cdot 1_{\mathrm{D}}^\top$
$\textsf{stable-log-softmax}(x) = z - \log(e^z \cdot 1_{\mathrm{D}}) \cdot 1_{\mathrm{D}}^\top$	$\textsf{stable-log-softmax}(X) = Z - \log(e^Z \cdot 1_{\mathrm{D}}) \cdot 1_{\mathrm{D}}^\top,$

where

$$\begin{cases} z = x - \max(x) \cdot 1_{\mathrm{D}}^\top \\ Z = X - \max(X)_{\text{row}} \cdot 1_{\mathrm{D}}^\top. \end{cases}$$

The derivatives of the softmax functions are given by

$$\frac{\partial}{\partial x}\textsf{softmax}(x) = \frac{\partial}{\partial x}\textsf{stable-softmax}(x) = \textsf{Diag}(\textsf{softmax}(x)) - \textsf{softmax}(x)^\top \cdot \textsf{softmax}(x), \tag{41}$$

$$\frac{\partial}{\partial x}\textsf{log-softmax}(x) = \frac{\partial}{\partial x}\textsf{stable-log-softmax}(x) = \mathbb{I}_D - 1_{\mathrm{D}} \cdot \textsf{softmax}(x) \tag{42}$$

C.2 Derivations

log-softmax Let $y(x) = \textsf{softmax}(x)$, then we have

$$\begin{aligned} \frac{\partial}{\partial x}\log\left(\textsf{softmax}(x)\right) &\overset{(22)}{=} \frac{\partial}{\partial y}\log(y)\frac{\partial}{\partial x}\textsf{softmax}(x) \\ &\overset{(13)}{=} \textsf{Diag}\left(\frac{1}{y}\right) \cdot \left(\textsf{Diag}(y) - y^\top y\right) \\ &= \mathbb{I}_D - \textsf{Diag}\left(\frac{1}{y}\right) y^\top y \\ &= \mathbb{I}_D - 1_{\mathrm{D}} \cdot y \\ &= \mathbb{I}_D - 1_{\mathrm{D}} \cdot \textsf{softmax}(x). \end{aligned} \tag{43}$$

D Loss functions

In this section we give an overview of some loss functions and their gradients, both for single inputs and for batches. We use the following notations:

1. $y \in \mathbb{R}^{1\times \mathrm{K}}$ is a single output, where K is the output dimension.
2. $t \in \mathbb{R}^{1\times \mathrm{K}}$ is a target for a single output y.
3. $Z \in \mathbb{R}^{\mathrm{N}\times \mathrm{K}}$ is an output batch, where N is the number of examples in the batch.
4. $T \in \mathbb{R}^{\mathrm{N}\times \mathrm{K}}$ contains the targets for an output batch Y.

squared error loss

$$\begin{aligned} \mathcal{L}_{\mathrm{SE}}(y,t) &= (y-t)(y-t)^\top \\ \nabla_y \mathcal{L}_{\mathrm{SE}}(y,t) &= 2(y-t) \\ \mathcal{L}_{\mathrm{SE}}(Y,T) &= 1_{\mathrm{N}}^\top \cdot ((Y-T)\odot(Y-T)) \cdot 1_{\mathrm{K}} \\ \nabla_Y \mathcal{L}_{\mathrm{SE}}(Y,T) &= 2(Y-T) \end{aligned} \tag{44}$$

The mean squared error loss is a scaled version of the squared error loss.

$$\mathcal{L}_{\mathrm{MSE}}(Y,T) = \frac{\mathcal{L}_{\mathrm{SE}}(Y,T)}{\mathrm{K}\cdot\mathrm{N}}$$

cross-entropy loss

$$\begin{aligned} \mathcal{L}_{\mathrm{CE}}(y,t) &= -t\cdot\log(y)^\top \\ \nabla_y \mathcal{L}_{\mathrm{CE}}(y,t) &= -t\odot\frac{1}{y} \\ \mathcal{L}_{\mathrm{CE}}(Y,T) &= -1_{\mathrm{N}}^\top \cdot (T\odot\log(Y))\cdot 1_{\mathrm{K}} \\ \nabla_Y \mathcal{L}_{\mathrm{CE}}(Y,T) &= -T\odot\frac{1}{Y} \end{aligned} \tag{45}$$

softmax cross-entropy loss

$$\begin{aligned} \mathcal{L}_{\mathrm{SCE}}(y,t) &= -t\cdot\mathsf{log\text{-}softmax}(y)^\top \\ \nabla_y \mathcal{L}_{\mathrm{SCE}}(y,t) &= t\cdot 1_{\mathrm{K}}\cdot\mathsf{softmax}(y) - t \\ \mathcal{L}_{\mathrm{SCE}}(Y,T) &= -1_{\mathrm{N}}^\top\cdot(T\odot\mathsf{log\text{-}softmax}(Y))\cdot 1_{\mathrm{K}} \\ \nabla_Y \mathcal{L}_{\mathrm{SCE}}(Y,T) &= \mathsf{softmax}(Y)\odot(T\cdot 1_{\mathrm{K}}\cdot 1_{\mathrm{K}}^\top) - T \end{aligned} \tag{46}$$

Note that if t is a one-hot encoded target (e.g., in case of a classification problem), it is a vector consisting of one value 1 and all others values 0. In other words we have $t\cdot 1_{\mathrm{K}} = 1$, hence in this case the gradients can be simplified to

$$\begin{aligned} \nabla_y \mathcal{L}_{\mathrm{SCE}}(y,t) &= \mathsf{softmax}(y) - t \\ \nabla_Y \mathcal{L}_{\mathrm{SCE}}(Y,T) &= \mathsf{softmax}(Y) - T \end{aligned}$$

logistic cross-entropy loss

$$\begin{aligned}
\mathcal{L}_{\text{LCE}}(y,t) &= -t \cdot \log(\sigma(y))^\top \\
\nabla_y \mathcal{L}_{\text{LCE}}(y,t) &= t \odot \sigma(y) - t \\
\mathcal{L}_{\text{LCE}}(Y,T) &= -1_{\text{N}}^\top \cdot (T \odot \log(\sigma(Y))) \cdot 1_{\text{K}} \\
\nabla_Y \mathcal{L}_{\text{LCE}}(Y,T) &= T \odot \sigma(Y) - T
\end{aligned} \tag{47}$$

negative log-likelihood loss

$$\begin{aligned}
\mathcal{L}_{\text{NLL}}(y,t) &= -\log(y \cdot t^\top) \\
\nabla_y \mathcal{L}_{\text{NLL}}(y,t) &= -\frac{1}{y \cdot t^\top} \cdot t \\
\mathcal{L}_{\text{NLL}}(Y,T) &= -1_{\text{N}}^\top \cdot (\log((Y \odot T) \cdot 1_{\text{K}})) \\
\nabla_Y \mathcal{L}_{\text{NLL}}(Y,T) &= -\left(\frac{1}{(Y \odot T) \cdot 1_{\text{K}}} \cdot 1_{\text{K}}^\top\right) \odot T
\end{aligned} \tag{48}$$

D.1 Derivations

cross-entropy loss

$$\nabla_y \mathcal{L}_{\text{CE}}(y,t) = -\frac{\partial}{\partial y}(t \cdot \log(y)^\top) \overset{(20)}{=} -t \cdot \mathsf{Diag}\left(\frac{1}{y}\right) = -t \odot \frac{1}{y} \tag{49}$$

softmax cross-entropy loss

$$\begin{aligned}
\nabla_y \mathcal{L}_{\text{SCE}}(y,t) &= -\frac{\partial}{\partial y}(t \cdot \mathsf{log\text{-}softmax}(y)^\top) \overset{(20)}{=} -t \cdot \frac{\partial}{\partial y}\mathsf{log\text{-}softmax}(y) \\
&\overset{(43)}{=} -t \cdot (\mathbb{I}_K - 1_{\text{K}} \cdot \mathsf{softmax}(y)) = t \cdot 1_{\text{K}} \cdot \mathsf{softmax}(y) - t
\end{aligned} \tag{50}$$

If the target t is a one-hot encoded vector, we have $t \cdot 1_{\text{K}} = 1$. In that case the gradient simplifies to

$$\nabla_y \mathcal{L}_{\text{SCE-one-hot}}(y,t) = \mathsf{softmax(y)} - t.$$

Using property (28) we can generalize equation (50) to

$$\nabla_Y \mathcal{L}_{\text{SCE}}(Y,T) = \mathsf{softmax}(Y) \odot (T \cdot 1_{\text{K}} \cdot 1_{\text{K}}^\top) - T.$$

logistic cross-entropy loss

$$\begin{aligned}
\nabla_y \mathcal{L}_{\text{LCE}}(y,t) &= \frac{\partial}{\partial y}(-t \cdot \log(\sigma(y))^\top) \overset{(20)}{=} -t \cdot \frac{\partial}{\partial y}\log(\sigma(y)) \\
&= -t \cdot \mathsf{Diag}(1_{\text{K}}^\top - \sigma(y)) = t \odot \sigma(y) - t,
\end{aligned}$$

where we use the fact that in the univariate case:

$$\frac{d}{dt}\log(\sigma(t)) = \frac{\sigma'(t)}{\sigma(t)} = \frac{\sigma(t) \cdot (1 - \sigma(t))}{\sigma(t)} = 1 - \sigma(t).$$

E Weight initialization

The initial values of the weights in linear layers need to be chosen carefully, since they may have a large impact on the performance of a neural network (Narkhede et al., 2022). Typically, these values are randomly generated based on specific probability distributions. In this section we give a few commonly used distributions.

NAME	DISTRIBUTION
Uniform	$U(a, b)$
Xavier (Glorot and Bengio, 2010)	$U(-\frac{1}{\sqrt{\mathrm{D}}}, \frac{1}{\sqrt{\mathrm{D}}})$
Normalized Xavier (Glorot and Bengio, 2010)	$U(-\frac{\sqrt{6}}{\sqrt{\mathrm{D}+\mathrm{K}}}, \frac{\sqrt{6}}{\sqrt{\mathrm{D}+\mathrm{K}}})$,
He (He et al., 2015)	$\mathcal{N}(0, \sqrt{\frac{2}{\mathrm{D}}})$,

where D is the number of inputs and K the number of outputs of the layer to which the weight matrix belongs. Furthermore $U(a, b)$ is the uniform distribution on a given interval $[a, b]$, and $\mathcal{N}(\mu, \sigma)$ is the normal distribution with mean μ and standard deviation σ.

Unlike weights, the initial values of bias vectors are typically set to a small constant or zero rather than drawn from probability distributions. However, the Xavier weight distribution can also be used.

F Optimization

In the optimization step the parameters θ of a layer are updated based on the value of their gradient $\mathsf{D}\theta$ with respect to a given loss function. The goal of this step is to decrease the value of the loss. In this section we give three common choices for optimization methods: gradient descent, momentum and Nesterov. All of them take a learning rate parameter η as input, which is used to control the size of the optimization step. Our equations are equivalent to the ones in Keras (Chollet et al., 2015), but presented in matrix-form. We use a prime symbol to denote updated values.

GRADIENT DESCENT	MOMENTUM	NESTEROV
$\theta' = \theta - \eta \cdot \mathsf{D}\theta$	$\Delta'_\theta = \mu \cdot \Delta_\theta - \eta \cdot \mathsf{D}\theta$	$\Delta'_\theta = \mu \cdot \Delta_\theta - \eta \cdot \mathsf{D}\theta$
	$\theta' = \theta + \Delta'_\theta$	$\theta' = \theta + \mu \cdot \Delta'_\theta - \eta \cdot \mathsf{D}\theta$

Both momentum and Nesterov depend on a parameter $0 < \mu < 1$. Furthermore, they store an additional parameter Δ_θ with the same dimensions as θ. This parameter is updated in each optimization call, and initially contains only zeroes.

G Learning rate schedulers

We define a learning rate scheduler as a function $\eta : \mathbb{N} \to \mathbb{R}$ that returns the learning rate at optimization step i. We assume that η_0 is a given initial learning rate.

Formula	Description
$\eta_i = \eta_0$	A constant scheduler with initial value η_0.
$\eta_{i+1} = \dfrac{\eta_i}{1 + d \cdot i}$	A time-based scheduler with decay parameter d.
$\eta_i = \eta_0 \cdot d^{\lfloor \frac{1+i}{r} \rfloor}$	A step-based scheduler with change rate d and drop rate r.
$\eta_i = \eta_0 e^{-d \cdot i}$	An exponential scheduler with decay parameter d.
$\eta_i = \eta_0 \Gamma^{\sum_{j=1}^{k} \lfloor \frac{i}{m_j} \rfloor}$	A multi-step scheduler with decay parameter Γ and milestones $\{m_1, \dots m_k\}$.