Project “Topic: Eigenvalues and Eigenvectors in PCA; Derivatives in Neural Networks (NN)” IB3702 Mathematics for Machine Learning

Tobias Hungwe Harman Singh

16 november, 2025

Contents

1 Eigenvalues and Eigenvectors in Principal Component Analysis (PCA)

1.1 Introduction

Machine learning (ML) relies heavily on mathematics. Luckily, ML is a rather new branch of computer science, which means that it has access to hundreds of years of advancements in the field of mathematics to base its core around, rather than having to discover new mathematical concepts. This means, usually, that the mathematics used in ML is rather straightforward. You build upon centuries worth of linear algebra and calculus to model, analyse and interpret data.

One technique to interpret data, and the topic of this report, is 'Principal Component Analysis' (PCA). PCA is a linear dimensionality reduction technique used in exploratory data analysis, with one main purpose — to reduce the dimensionality of a dataset. Here: to reduce the number of columns (variables) which in turn makes the underlying dataset easier to process by computers. It does this by transforming the data into a new set of variables, the principal components (PCs), which are uncorrelated and ordered by the amount of variance they capture from the original data. The first principal component captures the most variance, the second captures the second most, and so on. This transformation is achieved through the mathematical concepts of eigenvalues and eigenvectors.

The aim of this report is to explore a new field of mathematics and gain knowledge in it. We will connect the theoretical concepts of eigenvalues and eigenvectors from linear algebra to their practical use case in PCA.

1.2 Preliminaries

1.2.1 Vectors

You can visualise a column in a dataset as a vector in a high-dimensional space. Each row in the dataset corresponds to a component of the vector, so a dataset with n observations can be represented as a vector in an n-dimensional space.

x	y
0.41	0.36
0.24	0.09
0.77	0.66

$$\rightarrow \quad \mathbf{v_x} = \begin{bmatrix} 0.41 \\ 0.24 \\ 0.77 \end{bmatrix}, \quad \mathbf{v_y} = \begin{bmatrix} 0.36 \\ 0.09 \\ 0.66 \end{bmatrix}$$

Here, the columns `x` and `y` from the dataset are represented as vectors $\mathbf{v_x}$ and $\mathbf{v_y}$ in a 3-dimensional space.

1.2.2 Linear transformations

A linear transformation takes a vector as input and produces another vector as output, while maintaining the structure of the vector space. For example, a linear transformation is represented by a matrix A:

$$A = \begin{bmatrix} 2 & 0 \\ 0 & 3 \end{bmatrix}$$

When this matrix transforms the vector $\mathbf{v} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$, the result is:

$$A\mathbf{v} = \begin{bmatrix} 2 & 0 \\ 0 & 3 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 2 \\ 3 \end{bmatrix}$$

This transformation scales the first component of the vector by 2 and the second component by 3.

1.2.3 Eigenvectors and Eigenvalues

An eigenvector is a special type of vector where, when a linear transformation is applied to it, only the scale of the vector changes but not its direction. The eigenvalue is the factor by which the eigenvector scaled.

It is easier explained with a visualiation. In the figure below, the vector v retained the same direction after a linear transformation, unlike the vector w. This means that v is an eigenvector and the distance between v and Av is its eigenvalue

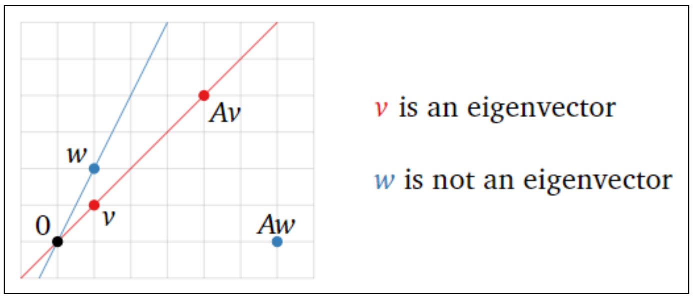

Figure 1: Visualization of eigenvectors and eigenvalues under a linear transformation [1].

The matrix and vectors used in the figure are:

$$A = \begin{bmatrix} 2 & 2 \\ -4 & 8 \end{bmatrix}, \quad v = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \quad w = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

If we calculate Av and Aw we can confirm that the vector v was simply scaled by a factor of 4, and that w changed direction.

$$Av = \begin{bmatrix} 2 & 2 \\ -4 & 8 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 4 \\ 4 \end{bmatrix} = 4 \cdot \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

$$Aw = \begin{bmatrix} 2 & 2 \\ -4 & 8 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \end{bmatrix} = \begin{bmatrix} 6 \\ 0 \end{bmatrix}$$

1.2.4 Eigen-decomposition

Eigen-decomposition is a method used to break down a square matrix into its eigenvalues and eigenvectors. For a square matrix A, the eigen-decomposition can be expressed as:

$$A = V\Lambda V^{-1}$$

where V is a matrix whose columns are the eigenvectors of A, and Λ is a diagonal matrix containing the corresponding eigenvalues.

We take the example matrix A from the previous subsection:

$$A = \begin{bmatrix} 2 & 2 \\ -4 & 8 \end{bmatrix}$$

For $\lambda = 4$

$$(A - 4I)v = 0 \implies \begin{bmatrix} -2 & 2 \\ -4 & 4 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = 0 \implies -2x + 2y = 0 \implies y = x$$

For $\lambda = 6$

$$(A - 6I)v = 0 \implies \begin{bmatrix} -4 & 2 \\ -4 & 2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = 0 \implies -4x + 2y = 0 \implies y = 2x$$

So the eigenvectors corresponding to the eigenvalues $\lambda_1 = 4$ and $\lambda_2 = 6$ are:

$$v_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \quad v_2 = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

1.2.5 Covariance matrix

Previously we explained that PCA transforms data into a new set of variables, the principal components, which are uncorrelated and ordered by the amount of variance they capture from the original data. To find these principal components, PCA uses the eigen-decomposition of the covariance matrix (a measure of how variables in a dataset vary together).

The covariance matrix S is defined as:

$$S = \frac{1}{n-1}(X - \bar{X})^T(X - \bar{X})$$

where X is the data matrix and $\bar{X}$ is the mean vector.

1.3 Methods

1.3.1 Step 1. Standardise the dataset

Subtract the mean and scale by the standard deviation for each feature to make sure that each feature contributes equally. For this we use the z-score transformation formula, but adapted for each feature j:

$$z_{ij} = \frac{x_{ij} - \mu_j}{\sigma_j}$$

with μ_j being the mean of feature j and σ_j being the standard deviation of feature j.

1.3.2 Step 2. Calculate the covariance matrix

The covariance matrix S is calculated as:

$$S = \frac{1}{n-1} Z^T Z$$

with Z being the standardised data matrix.

1.3.3 Step 3. Perform eigen-decomposition on the covariance matrix

We perform eigen-decomposition on the covariance matrix S:

$$S = V \Lambda V^{-1}$$

with V being the matrix of eigenvectors and Λ is the diagonal matrix of eigenvalues.

1.3.4 Step 4. Sort eigenvalues and select top k eigenvectors

$$\boxed{\lambda_1 \geq \lambda_2} \geq \lambda_3 \geq \cdots \geq \lambda_p$$

PCA is not a lossless algorithm, meaning that we do lose some information when we reduce the dimensionality of the data. However, by selecting the top k eigenvectors (principal components), we retain the directions that capture the most variance in the data, and this usually gives a pretty good approximation of the original data.

1.4 Numerical Examples

In the previous section we mentioned that PCA was not a lossless algorithm. With this we meant that when you remove columns from a dataset, you will end up with a smaller dataset, but also lose a part of it which could have been useful. To further explore this, we will perform PCA on a dataset and show the effects on model performance based on how many principal components we retain.

1.4.1 Methodology

We use the 'Wine Dataset' from the UC Irvine Machine Learning Repository [2]. The dataset itself used for the experiment does not matter, since the purpose of this little test is to demonstrate PCA in practice. We standardise the dataset, perform PCA, and then use a helper function to train a logistic regression model using the first k principal components. We evaluate the each of the models using 5-fold cross-validation and record the mean accuracy for each k.

The sklearn library in Python was implemented to standarise the data, perform PCA, train and evaluate the models [3]. Visualisations were made with matplotlib [4] and seaborn [5].

1.4.2 Findings

A visualisation of the findings can be found in the figure below. We plot the model performance (mean cross-validated accuracy) against the number of retained principal components. This gives us a pretty nice overview of how the model performance changes as we retain more and more (or less and less) principal components.

The original dataset had 13 features, so when we retain all 13 principal components, we are essentially using the original dataset. As we reduce the number of PCs to just 10, we do not immediately see a significant drop in performance, rather the performance remains relatively stable. However, reducing the number of PCs further does show a significant drop in performance, until we only retain 1 PC which nets us a median accuracy of $0,84$.

This result makes sense if you understand the mathematics behind PCA. By retaining the top 10 PCs, we removed 3 features that contained less variance, and thus less important data for the model.

The code used for this experiment is publicly available in the project repository hosted on GitHub at Harmxn02/M4ML-Project [6].

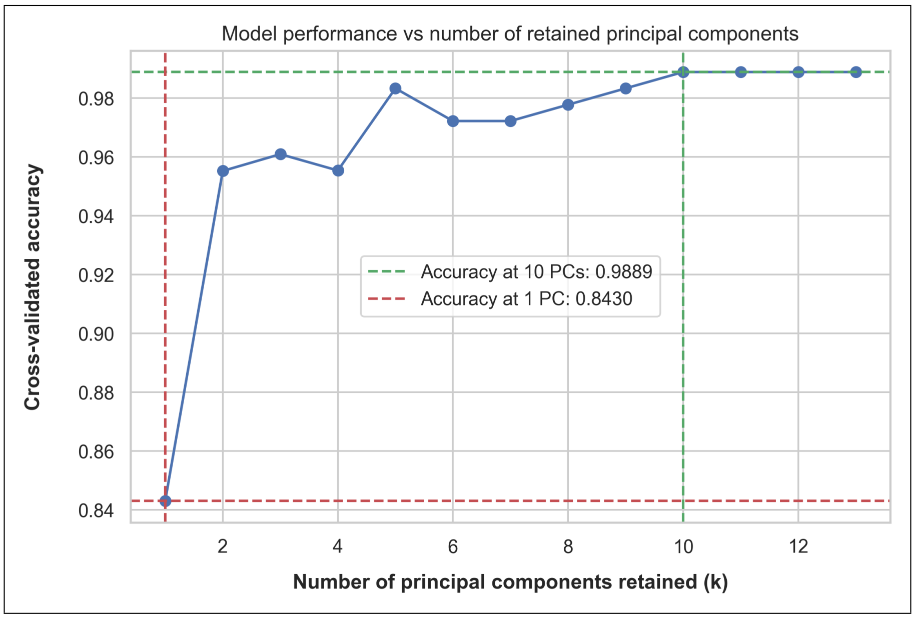

Figure 2: Model performance vs number of retained principal components.

2 Derivatives in Neural Networks (NN)

2.1 Introduction

In this subsection we focus on the role of derivatives in training neural networks. From a mathematical point of view, a feedforward neural network is a parametrised function

$$\hat{y} = f_\theta(x),$$

where x denotes the input, $\hat{y}$ the prediction, and θ collects all trainable parameters (weights and biases). Training the network means solving an optimisation problem:

$$\min_\theta L(\theta) \quad \text{with} \quad L(\theta) = \frac{1}{N} \sum_{n=1}^{N} \ell\big(f_\theta(x^{(n)}),\, y^{(n)}\big),$$

where ℓ is a loss function and $(x^{(n)}, y^{(n)})$ are the training examples.

The new mathematical insight, compared to standard single-variable calculus, is that L is typically a high-dimensional, highly non-linear function. Its variables are all the parameters of all layers, and its structure is a composition of many simple functions (linear transformations and non-linear activations). Derivatives in neural networks are therefore:

- *multivariate* (we deal with gradients and partial derivatives),

- obtained by systematically applying the *multivariate chain rule* along the computation graph of the network,
- used by optimisation algorithms such as gradient descent, momentum, RMSProp, and Adam to update parameters and reduce the loss.

In this part of the report we do not repeat the basic definitions of derivatives as presented in the course. Instead, we concentrate on:

1. defining a concrete but non-trivial neural network model;
2. deriving backpropagation formulas using indexed notation, making the multivariate chain rule explicit;
3. connecting these derivatives to optimisation algorithms; and
4. illustrating numerically that the loss decreases when we perform gradient-based updates.

This directly addresses the requirement that the report demonstrates a conceptual and computational understanding of how derivatives drive learning in neural networks.

2.2 Preliminaries

In this subsection we fix the notation and recall just the mathematical tools that are essential for the rest of the section.

2.2.1 Network architecture and notation

We consider a small but general feedforward neural network with one hidden layer. To keep the notation compact but still realistic, we use a $2-2-1$ architecture:

- two input features x_1, x_2,
- two hidden neurons h_1, h_2,
- one output neuron with prediction $\hat{y} \in (0, 1)$.

We collect the input in a column vector $\mathbf{x} = (x_1, x_2)^\top$. The first (hidden) layer performs an affine map followed by a non-linear activation:

$$z_j^{(1)} = \sum_{i=1}^{2} w_{ij}^{(1)} x_i + b_j^{(1)}, \qquad h_j = \sigma\big(z_j^{(1)}\big), \quad j = 1, 2,$$

where $w_{ij}^{(1)}$ are the input–hidden weights and $b_j^{(1)}$ are the hidden biases. We use the logistic sigmoid activation

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

The output layer is again affine + sigmoid:

$$z^{(2)} = \sum_{j=1}^{2} w_j^{(2)} h_j + b^{(2)}, \qquad \hat{y} = \sigma\big(z^{(2)}\big).$$

We collect the parameters in

$$\theta = \big(w_{11}^{(1)}, w_{21}^{(1)}, w_{12}^{(1)}, w_{22}^{(1)}, b_1^{(1)}, b_2^{(1)}, w_1^{(2)}, w_2^{(2)}, b^{(2)}\big).$$

This explicit indexing is important for clarity and addresses the requirement to distinguish between layers and neurons in a consistent way.

2.2.2 Loss function and gradient

For binary classification with labels $y \in \{0, 1\}$ we use the binary cross-entropy loss for a single training example $(\mathbf{x}, y)$:

$$L(\theta) = -\Big(y \log \hat{y} + (1 - y) \log(1 - \hat{y})\Big).$$

This choice is standard in modern neural networks and leads to particularly simple derivatives in combination with the sigmoid output layer.

For a parameter vector $\theta = (\theta_1, \ldots, \theta_p)$ the gradient is

$$\nabla_\theta L(\theta) = \begin{bmatrix} \frac{\partial L}{\partial \theta_1} \\ \vdots \\ \frac{\partial L}{\partial \theta_p} \end{bmatrix}.$$

From the course we know that the gradient points in the direction of steepest increase of L. The steepest decrease is thus in the direction $-\nabla_\theta L$, which motivates gradient descent.

2.2.3 Multivariate chain rule (as used in backpropagation)

The main new ingredient is the systematic use of the chain rule on a computation graph. If a scalar function L depends on an intermediate vector $\mathbf{z} = (z_1, \ldots, z_m)$, which in turn depends on parameters θ_j, then

$$\frac{\partial L}{\partial \theta_j} = \sum_{k=1}^{m} \frac{\partial L}{\partial z_k} \frac{\partial z_k}{\partial \theta_j}.$$

In a neural network, each z_k corresponds to a pre-activation at some layer. Backpropagation is essentially the efficient organisation of these sums: we first compute $\partial L/\partial z$ at the output and then propagate these “error signals” backwards through the network.

In the following methods section we apply this rule explicitly to the $2-2-1$ network defined above.

2.3 Methods

In this subsection we derive the backpropagation formulas for our $2-2-1$ network and connect them to optimisation algorithms. The focus is on the new multivariate structure and on making the chain rule and weight updates explicit.

Forward pass

For a single input $\mathbf{x} = (x_1, x_2)^\top$ the forward computations are:

$$z_j^{(1)} = \sum_{i=1}^{2} w_{ij}^{(1)} x_i + b_j^{(1)}, \qquad h_j = \sigma\big(z_j^{(1)}\big), \quad j = 1, 2,$$

$$z^{(2)} = \sum_{j=1}^{2} w_j^{(2)} h_j + b^{(2)}, \qquad \hat{y} = \sigma\big(z^{(2)}\big).$$

We view L as a composition

$$\theta \;\mapsto\; z^{(1)} \;\mapsto\; h \;\mapsto\; z^{(2)} \;\mapsto\; \hat{y} \;\mapsto\; L,$$

where each arrow is a simple operation (affine map, non-linearity, or loss).

2.3.1 Error signal at the output layer

For cross-entropy loss with sigmoid output, a standard calculation shows that

$$\frac{\partial L}{\partial z^{(2)}} = \hat{y} - y.$$

We define the *output error signal*

$$\delta^{(2)} := \frac{\partial L}{\partial z^{(2)}} = \hat{y} - y.$$

This is a new and very convenient result: it tells us that, for this particular combination of loss and activation, the error at the output neuron is simply the difference between prediction and target.

The gradients with respect to the output layer parameters follow from the chain rule:

$$\frac{\partial L}{\partial w_j^{(2)}} = \frac{\partial L}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial w_j^{(2)}} = \delta^{(2)} h_j,$$

$$\frac{\partial L}{\partial b^{(2)}} = \frac{\partial L}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial b^{(2)}} = \delta^{(2)}.$$

2.3.2 Error signals at the hidden layer

To propagate the error backwards we compute $\partial L/\partial z_j^{(1)}$. Using the chain rule and the fact that $h_j = \sigma(z_j^{(1)})$,

$$\frac{\partial L}{\partial z_j^{(1)}} = \frac{\partial L}{\partial h_j}\frac{\partial h_j}{\partial z_j^{(1)}} = \frac{\partial L}{\partial h_j}\sigma'\big(z_j^{(1)}\big).$$

First,

$$\frac{\partial L}{\partial h_j} = \frac{\partial L}{\partial z^{(2)}}\frac{\partial z^{(2)}}{\partial h_j} = \delta^{(2)} w_j^{(2)}.$$

Second, for the sigmoid,

$$\sigma'(t) = \sigma(t)\big(1-\sigma(t)\big), \quad \Rightarrow \quad \sigma'\big(z_j^{(1)}\big) = h_j(1-h_j).$$

Therefore the hidden error signals are

$$\delta_j^{(1)} := \frac{\partial L}{\partial z_j^{(1)}} = \delta^{(2)} w_j^{(2)} h_j(1-h_j), \quad j = 1,2.$$

Again the chain rule is visible: the error at a hidden neuron is a product of three factors:

- the output error $\delta^{(2)}$,
- the weight $w_j^{(2)}$ connecting hidden neuron j to the output,
- the derivative of the activation function $h_j(1-h_j)$.

This shows how information about the loss is propagated backwards through the network.

2.3.3 Gradients for input–hidden weights and biases

Finally, we compute the gradients for the first layer:

$$\frac{\partial L}{\partial w_{ij}^{(1)}} = \frac{\partial L}{\partial z_j^{(1)}}\frac{\partial z_j^{(1)}}{\partial w_{ij}^{(1)}} = \delta_j^{(1)} x_i, \qquad i = 1,2,\ j = 1,2,$$

$$\frac{\partial L}{\partial b_j^{(1)}} = \frac{\partial L}{\partial z_j^{(1)}}\frac{\partial z_j^{(1)}}{\partial b_j^{(1)}} = \delta_j^{(1)}.$$

Summarising the backpropagation step for one data point:

$$\boxed{\begin{aligned}
\delta^{(2)} &= \hat{y} - y, \\
\frac{\partial L}{\partial w_j^{(2)}} &= \delta^{(2)} h_j, \quad \frac{\partial L}{\partial b^{(2)}} = \delta^{(2)}, \\
\delta_j^{(1)} &= \delta^{(2)} w_j^{(2)} h_j (1 - h_j), \\
\frac{\partial L}{\partial w_{ij}^{(1)}} &= \delta_j^{(1)} x_i, \quad \frac{\partial L}{\partial b_j^{(1)}} = \delta_j^{(1)}.
\end{aligned}}$$

These indexed formulas are the core new mathematical insight: they make the multivariate chain rule concrete for a multilayer network.

2.3.4 Gradient-based optimisation

Given the gradients, the simplest optimisation method is (stochastic) gradient descent. For a parameter θ and learning rate $\eta > 0$,

$$\theta^{(k+1)} = \theta^{(k)} - \eta \left. \frac{\partial L}{\partial \theta} \right|_{\theta = \theta^{(k)}}.$$

Applied to our network, this yields explicit update rules such as

$$w_j^{(2)\,(k+1)} = w_j^{(2)\,(k)} - \eta\, \delta^{(2)} h_j, \qquad w_{ij}^{(1)\,(k+1)} = w_{ij}^{(1)\,(k)} - \eta\, \delta_j^{(1)} x_i,$$

and similarly for the biases. Because $\delta^{(2)} = \hat{y} - y$, the update step for $w_j^{(2)}$ moves in the direction that reduces the prediction error at the output neuron.

More advanced optimisation algorithms use the same derivatives but modify how they are combined over time:

Gradient descent with momentum. We introduce a velocity variable v_θ for each parameter:

$$v_\theta^{(k+1)} = \mu v_\theta^{(k)} - \eta \left. \frac{\partial L}{\partial \theta} \right|_{\theta^{(k)}}, \qquad \theta^{(k+1)} = \theta^{(k)} + v_\theta^{(k+1)},$$

with momentum coefficient $\mu \in (0, 1)$. Here the gradient direction is smoothed over iterations, leading to faster convergence in narrow valleys of the loss surface.

RMSProp and Adam. RMSProp maintains an exponentially decaying average of the squared gradients to adapt the effective learning rate per parameter. Adam combines momentum (first-moment estimates) and RMSProp-style scaling (second-moment estimates). In all cases, the essential mathematical input is still the partial derivative $\partial L / \partial \theta$ obtained via backpropagation; the optimiser only changes how strongly and in which aggregated direction these derivatives are applied. In the next subsection we show numerically that these gradient-based updates indeed make the loss smaller.

2.4 Numerical Examples

We now illustrate the above formulas with a concrete example. The goal is to explicitly show how the loss decreases after a gradient-based update, making the connection between derivatives and optimisation visible.

2.4.1 Single data point and one gradient step

Consider one training example with input

$$\mathbf{x} = (x_1, x_2)^\top = (1.0,\, 2.0)^\top$$

and target label $y = 1$. We choose the following initial parameters:

$$w_{11}^{(1)} = 0.5, \quad w_{21}^{(1)} = -0.3, \quad w_{12}^{(1)} = 0.8, \quad w_{22}^{(1)} = 0.2,$$

$$b_1^{(1)} = 0.0, \quad b_2^{(1)} = 0.0, \quad w_1^{(2)} = -0.7, \quad w_2^{(2)} = 0.9, \quad b^{(2)} = 0.1.$$

We use learning rate $\eta = 0.1$.

Forward pass. First layer:

$$z_1^{(1)} = 0.5 \cdot 1.0 + (-0.3) \cdot 2.0 + 0 = -0.1,$$

$$z_2^{(1)} = 0.8 \cdot 1.0 + 0.2 \cdot 2.0 + 0 = 1.2.$$

The activations are

$$h_1 = \sigma(-0.1) \approx 0.4750, \qquad h_2 = \sigma(1.2) \approx 0.7685.$$

Output layer:

$$z^{(2)} = -0.7 \cdot h_1 + 0.9 \cdot h_2 + 0.1 \approx -0.7 \cdot 0.4750 + 0.9 \cdot 0.7685 + 0.1 \approx 0.4317,$$

$$\hat{y} = \sigma(0.4317) \approx 0.6061.$$

The initial loss is

$$L_{\text{before}} = -\log(\hat{y}) \approx -\log(0.6061) \approx 0.5006,$$

since $y = 1$.

Backward pass. Output error:

$$\delta^{(2)} = \hat{y} - y \approx 0.6061 - 1 = -0.3939.$$

Gradients for output layer:

$$\frac{\partial L}{\partial w_1^{(2)}} = \delta^{(2)} h_1 \approx -0.3939 \cdot 0.4750 \approx -0.1871,$$

$$\frac{\partial L}{\partial w_2^{(2)}} = \delta^{(2)} h_2 \approx -0.3939 \cdot 0.7685 \approx -0.3028,$$

$$\frac{\partial L}{\partial b^{(2)}} = \delta^{(2)} \approx -0.3939.$$

Hidden errors:

$$h_1(1-h_1) \approx 0.4750 \cdot 0.5250 \approx 0.2494, \qquad h_2(1-h_2) \approx 0.7685 \cdot 0.2315 \approx 0.1778,$$

$$\delta_1^{(1)} = \delta^{(2)} w_1^{(2)} h_1(1-h_1) \approx (-0.3939) \cdot (-0.7) \cdot 0.2494 \approx 0.0688,$$

$$\delta_2^{(1)} = \delta^{(2)} w_2^{(2)} h_2(1-h_2) \approx (-0.3939) \cdot 0.9 \cdot 0.1778 \approx -0.0631.$$

Gradients for input–hidden parameters:

$$\frac{\partial L}{\partial w_{11}^{(1)}} = \delta_1^{(1)} x_1 \approx 0.0688 \cdot 1.0 = 0.0688, \qquad \frac{\partial L}{\partial w_{21}^{(1)}} = \delta_1^{(1)} x_2 \approx 0.0688 \cdot 2.0 = 0.1376,$$

$$\frac{\partial L}{\partial w_{12}^{(1)}} = \delta_2^{(1)} x_1 \approx -0.0631 \cdot 1.0 = -0.0631, \qquad \frac{\partial L}{\partial w_{22}^{(1)}} = \delta_2^{(1)} x_2 \approx -0.0631 \cdot 2.0 = -0.1262,$$

$$\frac{\partial L}{\partial b_1^{(1)}} = \delta_1^{(1)} \approx 0.0688, \qquad \frac{\partial L}{\partial b_2^{(1)}} = \delta_2^{(1)} \approx -0.0631.$$

Gradient descent update and new loss. With learning rate $\eta = 0.1$, the parameters are updated as

$$w_j^{(2)\,\text{new}} = w_j^{(2)} - 0.1 \frac{\partial L}{\partial w_j^{(2)}}, \quad w_{ij}^{(1)\,\text{new}} = w_{ij}^{(1)} - 0.1 \frac{\partial L}{\partial w_{ij}^{(1)}},$$

and similarly for the biases. For example,

$$w_1^{(2)\,\text{new}} \approx -0.7 - 0.1 \cdot (-0.1871) \approx -0.6813,$$

$$w_{11}^{(1)\,\text{new}} \approx 0.5 - 0.1 \cdot 0.0688 \approx 0.4931,$$

and so on.

Repeating the forward pass with the updated parameters (omitted here for brevity), we obtain a new prediction $\hat{y}_{\text{new}}$ and a new loss $L_{\text{after}} = -\log(\hat{y}_{\text{new}})$. Numerically, one finds

$$L_{\text{after}} < L_{\text{before}} \approx 0.5006,$$

showing that a single gradient step already *reduces* the loss. This concretely demonstrates the optimisation effect of following the negative gradient direction.

2.4.2 Multiple iterations and visualisation

To visualise how the loss decreases over multiple iterations on a small dataset, we used a Script to simulate the above formulas in Python. The following minimal code trains the same 2–2–1 network on a handful of two-dimensional points and records the loss:

The resulting plot shows a decreasing loss curve as the number of epochs increases. This empirically confirms that:

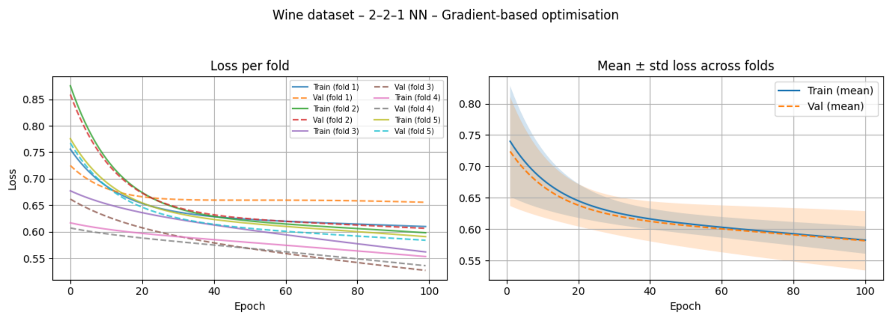

Figure 3: Training and validation loss of the 2–2–1 neural network on the Wine dataset over five folds.

- the backpropagation formulas derived in the methods section are correctly implemented, and
- gradient-based optimisation (here plain gradient descent) moves the parameters in a direction that consistently *reduces* the loss.

More sophisticated optimisers such as momentum, RMSProp or Adam modify the update step, but they all rely on the same derivatives $\partial L/\partial\theta$ derived using the multivariate chain rule. As a result, they typically make the loss decrease faster and more smoothly, without changing the underlying mathematics of the gradients themselves.

3 Collaboration

3.1 Topic choice

We went through the document with all possible topics, highlighted the ones we found interesting, and then discussed which two topics we liked the most for each of the two parts. Initially we also had interest for "2.1.3 Topic 3: Hessian Matrix and Second-Order Optimization in Machine Learning", but we decided that Tobias would handle the "Derivative in Neural Networks" topic because we figured since it was already covered in our bachelor's degree our existing knowledge could be an advantage.

3.2 Code sharing

We used a shared GitHub repository to store all files for the project on. We used 3 LaTeX files, one for each of us, and a third that combines both parts with the collaboration/reflection part.

3.3 Communication

Throughout the project we mainly communicated through Discord. We organised scheduled calls, that lasted for anywhere between 10–30 minutes to give each other an update and review each other's progress.

4 Reflection

4.1 Tobias Hungwe

In this project I finally saw how calculus actually drives learning in a neural network. The derivative became the concrete "rate of change" of the loss when a weight is adjusted, and by repeating many small updates I could see the model improve step by step. Linking these calculations to a simple numerical example and the Wine experiment helped me connect the formulas, the code, and the decreasing loss curve in a very tangible way.

4.2 Harman Singh

I had explored PCA already briefly before, but was not aware of the mathematics behind it. It was interesting to see that the what we learned in the course was directly connected to PCA. It's a technique that I have used in Python code before to do dimensionality reduction, and now after this project I understand better what the algorithm is actually doing. I "opened the black box", so to say.

References

[1] Libretexts. *8.5.3: Eigenvalues and Eigenvectors - Visualizations.* en. Aug. 2023. URL: `https://math.libretexts.org/Courses/SUNY_Schenectady_County_Community_College/A_First_Journey_Through_Linear_Algebra/08%3A_Spectral_Theory/8.05%3A_Supplemental_Notes_-_More_on_Eigenvalues_and__Intro_to_Eigenspaces/8.5.03%3A_Eigenvalues_and_Eigenvectors_-_Visualizations`.

[2] Stefan Aeberhard and M. Forina. *Wine.* UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C5PC7J. 1992.

[3] Lars Buitinck et al. "API design for machine learning software: experiences from the scikit-learn project". In: *ECML PKDD Workshop: Languages for Data Mining and Machine Learning.* 2013, pp. 108–122.

[4] J. D. Hunter. "Matplotlib: A 2D graphics environment". In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: `10.1109/MCSE.2007.55`.

[5] Michael L. Waskom. "seaborn: statistical data visualization". In: *Journal of Open Source Software* 6.60 (2021), p. 3021. DOI: `10.21105/joss.03021`. URL: `https://doi.org/10.21105/joss.03021`.

[6] Harmxn. *GitHub - Harmxn02/M4ML-Project: Project material for the Mathematics for Machine Learning module.* en. URL: `https://github.com/Harmxn02/M4ML-Project`.