UNIVERSITÀ
DI TORINO

Deep Learning

Francesco Marchisotti

`mfnet` – A simple Machine Learning Library

Professor:
Matteo Osella

academic year 2024/2025

Abstract

Modern machine learning libraries such as `PyTorch` and `TensorFlow` are widely used in both academia and industry. They are highly optimized, feature-rich, and provide extensive support for various machine learning tasks. In addition to their extensive features, these libraries are designed to efficiently leverage modern GPUs to significantly speed up training and inference for large-scale machine learning models. To gain a deeper understanding of the workings of such libraries, `mfnet` was developed as a simplified neural network library from scratch. The main objective was to implement a functional framework, focusing on the core concept of backpropagation. The effectiveness of the library was validated by analyzing the learning curves on two benchmark tasks: regression and classification. The results demonstrate that `mfnet` successfully learns and generalizes, confirming the correctness of its implementation.

Contents

Signal Flow in Neural Networks

A Feedforward Fully Connected Neural Network (FFCNN) is a Neural Network (NN) architecture where each neuron in one layer is connected to every neuron in the subsequent layer. A FFCNN learns by iteratively performing two main steps: the forward pass and the backward pass.

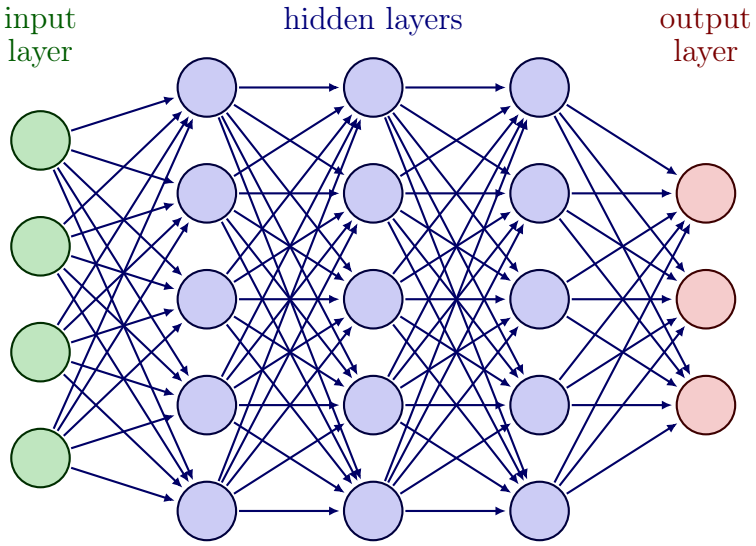

Figure 1: A FFCNN.

Forward pass The input data is propagated through the network, layer by layer, to produce an output prediction. This prediction is then compared to the true target values using a loss function, which quantifies the prediction error.

Backward pass The network uses the computed loss to adjust its internal parameters. This is done by propagating the error backward through the network and updating the weights to minimize the loss. The process of forward and backward passes is repeated for multiple iterations, gradually improving the model's performance. This step is the heart of the learning process, as it allows the network to learn from its mistakes and improve its predictions over time.

Part I
Implementation of `mfnet`

This first part describes the implementation details of the `mfnet` library. All the code for this project is available on GitHub.

1 Basic Data Structure

The fundamental data structure in `mfnet` is the tensor. In this context, a tensor is simply a `numpy` array with a fixed data type of `numpy.float64`. Tensors are used throughout `mfnet` to represent inputs, outputs, intermediate activations, weights, and gradients within the Neural Network.

Conventions and shapes Throughout `mfnet`:

- columns are samples and rows are features/classes;
- tensors are bias-augmented unless stated otherwise: the first row is a constant row of ones that propagates through Linear layers (see Section 3). Activation layers copy this row unchanged, and losses temporarily remove it before computing metrics (see Section 4).

Canonical shapes (with m being the mini-batch size or the dataset size depending on context):

- inputs $X \in \mathbb{R}^{(n_{\text{features}}+1)\times m}$ after bias augmentation;
- linear weights $W^{[l]} \in \mathbb{R}^{(n_{\text{out}}+1)\times(n_{\text{in}}+1)}$ with the first row fixed to $\begin{pmatrix}1 & 0\cdots 0\end{pmatrix}$ (non-learnable);
- pre-activations and activations $Z^{[l]}, A^{[l]} \in \mathbb{R}^{(n_{\text{out}}+1)\times m}$ (see Sections 2 and 3);
- targets:
  - regression: $Y \in \mathbb{R}^{n_{\text{targets}}\times m}$ (the loss removes/ignores any bias row if present);
  - classification: one-hot $Y \in \mathbb{R}^{C\times m}$ with C classes; the loss operates on logits with the bias row removed and restores a zero row in the gradient.

Important note: These conventions apply only to the internal workings of `mfnet`. There might be some functions that violate these conventions; every function is thoroughly documented and clearly states the shape of the input it is designed to work with. End users are expected to provide the data in standard $m \times n_{\text{in}}$ and $m \times n_{\text{targets}}$ or $m \times C$ matrices, and `mfnet` handles the translation via the `DataLoader` class (see Section 5.3).

2 Backpropagation

Before diving into the details of the implementation of `mfnet`, it's necessary to understand the algorithm at the core of the learning process: backpropagation. The goal of the backpropagation algorithm is to compute the derivative of the loss with respect to each weight in the network using the chain rule. The algorithm is more easily understandable using the index notation, but in order to implement it in code, we need to express it in matrix form, so both formulations will be shown.

2.1 Index notation

We start by defining the following quantities:

- L: the total number of layers in the network;
- $n^{[l]}$: the number of neurons in layer l (with $l = 1$ being the input layer, and $l = L$ being the output layer);
- $z_j^{[l]}$: the pre-activation of neuron j in layer l. This is the output of the linear transformation in layer l and the input of the activation function;
- $a_j^{[l]}$: the activation of neuron j in layer l (with $a_j^{[0]} = x_j$, and $a_j^{[L]} = \hat{y}_j$). This is the output of layer l and the input of layer $l + 1$;
- $W_{jk}^{[l]}$: the weight connecting neuron k in layer $l - 1$ to neuron j in layer l (with $W_{j1}^{[l]} = b_j^{[l]}$);
- g: the activation function used in the hidden layers;
- f: the activation function used in the output layer.

The flow of information through layer l is given by:

$$z_j^{[l]} = \sum_{k=1}^{n^{[l-1]}} W_{jk}^{[l]} a_k^{[l-1]}$$

$$a_j^{[l]} = g\left(z_j^{[l]}\right)$$

The derivative of the loss $\mathcal{J}$ with respect to the weight $W_{jk}^{[l]}$ is computed as:

$$\begin{aligned}\frac{\partial \mathcal{J}}{\partial W_{jk}^{[l]}} &= \frac{\partial \mathcal{J}}{\partial z_j^{[l]}} \frac{\partial z_j^{[l]}}{\partial W_{jk}^{[l]}} \\ &= \Delta_j^{[l]} \frac{\partial z_j^{[l]}}{\partial W_{jk}^{[l]}} = \Delta_j^{[l]} a_k^{[l-1]}\end{aligned}$$

Now, $\Delta_j^{[l]}$ can be expressed as a function of $\Delta_j^{[l+1]}$:

$$\begin{aligned}\Delta_j^{[l]} = \frac{\partial \mathcal{J}}{\partial z_j^{[l]}} &= \frac{\partial \mathcal{J}}{\partial a_j^{[l]}} \frac{\partial a_j^{[l]}}{\partial z_j^{[l]}} \\ &= \left(\sum_{i=1}^{n^{[l+1]}} \frac{\partial \mathcal{J}}{\partial z_i^{[l+1]}} \frac{\partial z_i^{[l+1]}}{\partial a_j^{[l]}} \right) \frac{\partial a_j^{[l]}}{\partial z_j^{[l]}} \\ &= \left(\sum_{i=1}^{n^{[l+1]}} \Delta_i^{[l+1]} W_{ij}^{[l+1]} \right) g' \left(z_j^{[l]} \right)\end{aligned}$$

The procedure can be iterated until the last layer L is reached:

$$\Delta_j^{[L]} = \frac{\partial \mathcal{J}}{\partial z_j^{[L]}} = \frac{\partial \mathcal{J}}{\partial a_j^{[L]}} \frac{\partial a_j^{[L]}}{\partial z_j^{[L]}} = \frac{\partial \mathcal{J}}{\partial \hat{y}_j} f' \left(z_j^{[L]} \right)$$

This last term can be computed after the forward pass is completed. By iteration, every $\Delta_j^{[l]}$ can be computed, and thus every $\frac{\partial \mathcal{J}}{\partial W_{jk}^{[l]}}$.

2.2 Matrix notation

Matrix notation can be easily derived from the index notation, being careful with the order of the products and with placing the transposes.

The definitions given in the Section 2.1 are updated as follows:

- m: the number of samples in the training batch;
- $Z^{[l]}$: the $n^{[l]} \times m$ pre-activation of layer l;
- $A^{[l]}$: the $n^{[l]} \times m$ activation of layer l;
- $W^{[l]}$: the $n^{[l]} \times n^{[l-1]}$ weight matrix of layer l.

The flow of information through layer l is given by:

$$Z^{[l]} = W^{[l]} A^{[l-1]} \tag{2.1}$$

$$A^{[l]} = g\left(Z^{[l]}\right) \tag{2.2}$$

The derivative of the loss $\mathcal{J}$ with respect to the weights $W^{[l]}$ is computed as:

$$\frac{\partial \mathcal{J}}{\partial W^{[l]}} = \frac{\partial \mathcal{J}}{\partial Z^{[l]}} \frac{\partial Z^{[l]}}{\partial W^{[l]}} = \Delta^{[l]} A^{[l-1]T} \tag{2.3}$$

Now, $\Delta^{[l]}$ can be expressed as a function of $\Delta^{[l+1]}$:

$$\begin{aligned} \Delta^{[l]} = \frac{\partial \mathcal{J}}{\partial Z^{[l]}} &= \frac{\partial \mathcal{J}}{\partial A^{[l]}} \odot \frac{\partial A^{[l]}}{\partial Z^{[l]}} \\ &= \left(\frac{\partial Z^{[l+1]}}{\partial A^{[l]}} \frac{\partial \mathcal{J}}{\partial Z^{[l+1]}} \right) \odot \frac{\partial A^{[l]}}{\partial Z^{[l]}} \\ &= \left(W^{[l+1]T} \Delta^{[l+1]} \right) \odot g'\left(Z^{[l]}\right) \end{aligned} \tag{2.4}$$

where $\odot$ denotes the element-wise (Hadamard) product.

The procedure can be iterated until the last layer L is reached:

$$\Delta^{[L]} = \frac{\partial \mathcal{J}}{\partial Z^{[L]}} = \frac{\partial \mathcal{J}}{\partial A^{[L]}} \odot \frac{\partial A^{[L]}}{\partial Z^{[L]}} = \frac{\partial \mathcal{J}}{\partial \hat{Y}} \odot f'\left(Z^{[L]}\right)$$

This last term can be computed after the forward pass is completed. By iteration, every $\Delta^{[l]}$ can be computed using Eq. (2.4), and thus every $\frac{\partial \mathcal{J}}{\partial W^{[l]}}$.

Note on bias augmentation In the implementation, tensors always include bias: $A^{[l]}$ includes a leading row of ones and $W^{[l]}$ includes the corresponding bias column (see Section 3), in addition to its bias row. The formulas above map directly by treating those rows/columns as fixed: the activation derivative on the bias row is zeroed in backward passes, and the first row of the gradient for logits/layer inputs is handled accordingly so that biases propagate but are not transformed by nonlinearities. This will be explained in greater detail in Sections 3 and 4, and shown in the practical example in Appendix B.

3 Layers

In `mfnet`, a layer is a fundamental building block of the Neural Network. Each layer performs a specific transformation on the input data. The two main types of layers implemented in `mfnet` are Linear layers and Activation layers.

Layers are stacked together to form a complete Neural Network, with the output of one layer serving as the input to the next layer. Each layer is responsible for maintaining its own parameters and computing gradients during the backpropagation process.

3.1 Linear layer

3.1.1 Forward pass

The Linear layer in `mfnet` applies a linear transformation to the input data, performing a change in dimensionality from n_{in} to n_{out}, where n_{in} is the number of input features and n_{out} is the number of output features. Mathematically, this can be represented as:

$$y = b + Wx$$

where:

- x is the $n_{\text{in}} \times 1$ input vector,
- W is the $n_{\text{out}} \times n_{\text{in}}$ weight matrix,
- b is the $n_{\text{out}} \times 1$ bias vector, and
- y is the $n_{\text{out}} \times 1$ output vector.

Implementation Details The actual implementation is a bit different: the first key difference is that the bias term is absorbed inside the weights matrix, and the input vector is augmented with an additional constant value of 1. The layer expects this "bias feature" to be already present in the input data, and propagates it to the next layer by adding a row of $\begin{pmatrix}1 & 0 \cdots 0\end{pmatrix}$ to the weights matrix. This allows us to rewrite the equation as:

$$\begin{pmatrix}1\\ y\end{pmatrix} = \begin{pmatrix}1 & 0 \cdots 0\\ b & W\end{pmatrix}\begin{pmatrix}1\\ x\end{pmatrix}$$

The second key difference is that, instead of feeding one data point at a time to the network and heavily relying on inefficient for loops, we can feed a batch of m data points at once, and leverage efficient matrix operations. This means that the input x is actually a matrix X where each column represents a different data

point, and the output y is also a matrix Y where each column corresponds to the output for each input data point. The equation then becomes:

$$\begin{pmatrix} 1 \cdots 1 \\ Y \end{pmatrix} = \begin{pmatrix} 1 & 0 \cdots 0 \\ b & W \end{pmatrix} \begin{pmatrix} 1 \cdots 1 \\ X \end{pmatrix}$$

Switching to the backpropagation notation introduced in Section 2, we can summarize the forward pass of a Linear layer with Eq. (2.1):

$$Z^{[l]} = W^{[l]} A^{[l-1]}$$

where:

- $A^{[l-1]}$ is the $(n_{\text{in}} + 1) \times m$ input of the Linear layer l, with $A^{[0]}$ being the input data,
- $W^{[l]}$ is the $(n_{\text{out}} + 1) \times (n_{\text{in}} + 1)$ weights matrix of the Linear layer l, and
- $Z^{[l]}$ is the $(n_{\text{out}} + 1) \times m$ output of the Linear layer l.

All these tensors already include the necessary additions to correctly handle the bias. The Linear layer forward method also stores its input $A^{[l-1]}$ for use in the backward pass.

3.1.2 Backward pass

The Linear layer's responsibility during the backward pass is to compute the gradients of the loss with respect to its weights (using Eq. (2.3)) and pass backward the gradient of the loss with respect to its input. Mathematically, this can be expressed as:

$$\frac{\partial \mathcal{J}}{\partial W^{[l]}} = \frac{\partial \mathcal{J}}{\partial Z^{[l]}} \frac{\partial Z^{[l]}}{\partial W^{[l]}} = \Delta^{[l]} A^{[l-1]T}$$
$$\frac{\partial \mathcal{J}}{\partial A^{[l-1]}} = \frac{\partial Z^{[l]}}{\partial A^{[l-1]}} \frac{\partial \mathcal{J}}{\partial Z^{[l]}} = W^{[l]T} \Delta^{[l]}$$

where $\mathcal{J}$ is the loss and $\Delta^{[l]} = \frac{\partial \mathcal{J}}{\partial Z^{[l]}}$ is the gradient of the loss with respect to the output of the Linear layer l.

Implementation Details The backward method of the Linear layer takes as input $\Delta^{[l]}$ and computes the gradient with respect to the weights $W^{[l]}$. This gradient is then stored in the layer for later use during the optimization step, when it will be used to update the weights.

3.2 Activation layer

3.2.1 Forward pass

The role of the Activation layer is to apply a non-linear activation function g element-wise to the output of the previous Linear layer.

Implementation Details The forward method of the Activation layer takes as input the output of the Linear layer $Z^{[l]}$ and applies the activation function element-wise to produce the activated output $A^{[l]}$ as in Eq. (2.2):

$$A^{[l]} = g(Z^{[l]})$$

This step is rendered more complex by the bias feature, which must be preserved and propagated to the next layer without any modification. Therefore, the activation function is applied only to the rows of $Z^{[l]}$ corresponding to actual features, leaving the first row (the bias feature) unchanged.

The Activation layer forward method also stores its input $Z^{[l]}$ for use in the backward pass. Only two types of activation functions are implemented in `mfnet`, ReLU and Sigmoid, defined as follows:

- ReLU: $g(x) = \max(0, x)$
- Sigmoid: $g(x) = 1/(1 + e^{-x})$

3.2.2 Backward pass

Since the Activation layer does not have any learnable parameters, its backward method is solely responsible for computing the gradient of the loss with respect to its input $Z^{[l]}$:

$$\frac{\partial \mathcal{J}}{\partial Z^{[l]}} = \frac{\partial \mathcal{J}}{\partial A^{[l]}} \odot g'(Z^{[l]})$$

where g' is the derivative of the activation function.

Implementation Details The backward method of the Activation layer takes as input $\frac{\partial \mathcal{J}}{\partial A^{[l]}}$ and computes $\frac{\partial \mathcal{J}}{\partial Z^{[l]}}$. Similar to the forward pass, the bias feature must be preserved during this computation. Therefore, the derivative of the activation function is applied only to the rows corresponding to actual features, setting the bias row to zero. This ensures that the first row of the weight matrix of the previous Linear layer does not get updated during the optimization step, and therefore that the bias feature gets correctly propagated forward through the network.

4 Loss

The final ingredient in the training of a Neural Network is the loss function. This function measures the error between the predicted output of the network and the true target values. The goal of training is to minimize this loss function by adjusting the weights of the network through backpropagation.

Also, as seen in Section 2, the derivative of the loss with respect to the output of the network is needed to start the backpropagation process. `mfnet` implements two of the most important loss functions: Mean Squared Error (MSE), used mainly in regression tasks, and Cross Entropy (CE), used in classification tasks.

4.1 Mean Squared Error

The MSE loss function is defined as:

$$\mathcal{J}(\hat{Y}, Y) = \frac{1}{m} \sum_{i=1}^{m} \left\| \hat{Y}_i - Y_i \right\|_2^2 = \frac{1}{m} \sum_{i=1}^{m} \sum_{j=1}^{n_{\text{feat}}} (\hat{Y}_{ij} - Y_{ij})^2 \tag{4.1}$$

where $\hat{Y}_i$ is the vector of predicted values for the i-th sample, Y_i is the vector of true target values for the i-th sample and m is the number of samples. It represents the square modulus of the error vector, averaged over all samples.

The gradient matrix of the loss with respect to the output of the network is given by:

$$\frac{\partial \mathcal{J}}{\partial \hat{Y}} = \frac{2}{m} (\hat{Y} - Y) \tag{4.2}$$

4.2 Cross Entropy

Cross Entropy is a measure that determines the similarity between two probability distributions. In the context of Neural Networks, it is commonly used as a loss function for classification tasks, where the predicted output of the network represents a probability distribution over different classes.

Definition The CE loss function is defined as:

$$\mathcal{J}(S, Y) = -\frac{1}{m} \sum_{i=1}^{m} \left\| Y_i \odot \log(S_i) \right\|_1 = -\frac{1}{m} \sum_{i=1}^{m} \sum_{j=1}^{C} Y_{ij} \log(S_{ij}) \tag{4.3}$$

where S_{ij} is the predicted probability of the i-th sample belonging to class j, Y_{ij} is the true probability (1 if the sample belongs to class j, 0 otherwise), m is the number of samples, and C is the number of classes.

This formula assumes that the predicted outputs S are a probability distribution (i.e., that $0 \leq S_{ij} \leq 1$ and $\sum_{j=1}^{C} S_{ij} = 1 \; \forall i$).

Such a distribution can be obtained by applying the softmax function to the raw output logits Z_{ij} of the network. The softmax function is defined as:

$$S_{ij} = \frac{e^{Z_{ij}}}{\sum_{k=1}^{C} e^{Z_{ik}}} \tag{4.4}$$

To make sure that the CE loss has the appropriate input, a softmax is automatically applied before it. The user must not manually insert a softmax layer before the loss (nor does `mfnet` define one). The actual formula for the CE loss used in `mfnet` is therefore:

$$\mathcal{J}(Z, Y) = -\frac{1}{m} \sum_{i=1}^{m} \sum_{j=1}^{C} Y_{ij} \log \left(\frac{e^{Z_{ij}}}{\sum_{k=1}^{C} e^{Z_{ik}}} \right) \tag{4.5}$$

Gradient The `CELoss` class is responsible for computing the gradient of the CE loss with respect to the input logits. This can be broken down in two steps.

Softmax Gradient We start by taking the logarithm of the softmax:

$$\log(S_{ij}) = Z_{ij} - \log \left(\sum_{k=1}^{C} e^{Z_{ik}} \right)$$

and then its derivative:

$$\begin{aligned} \frac{\partial \log(S_{ij})}{\partial Z_{lt}} &= \frac{\partial Z_{ij}}{\partial Z_{lt}} - \frac{\partial \log\left(\sum_{k=1}^{C} e^{Z_{ik}}\right)}{\partial Z_{lt}} \\ &= \delta_{ij,lt} - \frac{1}{\sum_{k=1}^{C} e^{Z_{ik}}} \sum_{k=1}^{C} \frac{\partial e^{Z_{ik}}}{\partial Z_{lt}} \\ &= \delta_{ij,lt} - \frac{1}{\sum_{k=1}^{C} e^{Z_{ik}}} \sum_{k=1}^{C} e^{Z_{ik}} \delta_{ik,lt} \\ &= \delta_{ij,lt} - \frac{e^{Z_{it}}}{\sum_{k=1}^{C} e^{Z_{ik}}} \delta_{il} \\ &= \delta_{ij,lt} - S_{it} \delta_{il} \end{aligned} \tag{4.6}$$

The gradient can now be easily obtained by inverting the relation:

$$\frac{\partial \log(S_{ij})}{\partial Z_{lt}} = \frac{1}{S_{ij}} \frac{\partial S_{ij}}{\partial Z_{lt}}$$

$$\frac{\partial S_{ij}}{\partial Z_{lt}} = S_{ij} \frac{\partial \log(S_{ij})}{\partial Z_{lt}} = S_{ij}(\delta_{ij,lt} - S_{it}\delta_{il})$$

CE Gradient We can now differentiate the CE with respect to the input logits Z_{lt}:

$$\begin{aligned}
\frac{\partial \mathcal{J}}{\partial Z_{lt}} &= -\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{C} Y_{ij}\frac{\partial \log(S_{ij})}{\partial Z_{lt}} \\
&\overset{(a)}{=} -\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{C} Y_{ij}\left(\delta_{ij,lt} - S_{it}\delta_{il}\right) \\
&= -\frac{1}{m}\left(Y_{lt} - \sum_{i=1}^{m}\sum_{j=1}^{C} Y_{ij}S_{it}\delta_{il}\right) \\
&= -\frac{1}{m}\left(Y_{lt} - S_{lt}\sum_{j=1}^{C} Y_{lj}\right) \\
&\overset{(b)}{=} -\frac{1}{m}\left(Y_{lt} - S_{lt}\right) \\
&= \frac{S_{lt} - Y_{lt}}{m}
\end{aligned} \tag{4.7}$$

where in (a) we used the result from Eq. (4.6), and in (b) we used the fact that Y_i is a one-hot encoded vector.

Implementation details Since the output logits from the network have the bias row (first row set to all 1s), before doing any calculation in the `CELoss` class we remove this row, both from the logits and from the target labels. For numerical stability, the softmax is actually computed as:

```
e_x = np.exp(logits - np.max(logits, axis=0, keepdims=True))
softmax_pred = e_x / np.sum(e_to_the_x, axis=0, keepdims=True)
```

To prevent numerical problems arising from $\log(0)$, we also clip the softmax output to a minimum value of 10^{-100}.

The `grad` method of the `CELoss` class prepends a row of zeros to the gradient matrix, to match the shape of the input logits.

5 Neural Network, Optimizer and Dataloader

In this section we describe the implementation of more high-level objects that can be built using the building blocks described in the previous sections.

5.1 Neural Network

A Neural Network is made up of a sequence of layers, each one transforming the input tensor into an output tensor, and each one feeding into the next.

As far as coding is concerned, it is a very simple object, and only defines three methods: forward, backward and a method that yields an iterator on all the layers' parameters. This last method is used by the optimizer when performing the update step.

The forward method simply calls the forward method of each layer in sequence, passing the output of one layer as input to the next one.

The backward method does the opposite: it calls the backward method of each layer in reverse order, passing the gradient of the loss with respect to the input of each layer as input to the previous layer.

5.2 Optimizer

The optimizer is responsible for updating the parameters of the Neural Network during training.

The most basic optimizer is Gradient Descent (GD), which updates the parameters according to the rule:

$$W^{[l]} \leftarrow W^{[l]} - \eta \frac{\partial \mathcal{J}}{\partial W^{[l]}} \tag{5.1}$$

where η is the learning rate.

The algorithm in this form suffers from many problems, such as having a big computational overhead and not being able to escape local minima. A more robust version is Stochastic Gradient Descent (SGD), which updates the parameters using a mini-batch (see Section 5.3) instead of the whole dataset. This introduces some stochastic noise in the updates, which can help the optimizer to better explore the space. It also reduces the size of the gradient matrix that needs to be computed, greatly reducing the computational burden.

Implementation details When training a full NN using ReLUs as activation functions, there was often an overflow problem. To help mitigate this, the `Optimizer` class implements gradient clipping, which prevents the gradients from becoming too large by clipping the norm of the gradient tensor to a maximum value provided by the user.

5.3 Dataloader

The dataloader is responsible for loading the data in mini-batches and shuffling it at the beginning of each epoch. This is important for SGD to work properly, as it helps to reduce the correlation between consecutive mini-batches and improves the convergence of the optimizer.

Implementation details In addition to creating the mini-batches, the `DataLoader` class is also responsible for transforming the data, which is usually stored in a $m \times n_{\text{features}}$ matrix (the design matrix), into a format compatible with the `mfnet` library: first, the design matrix is transposed, so that each column represents a data sample and each row represents a feature; then, the bias feature is prepended to the input data as a row of 1s. The same transformations are applied to the target data.

Example For a complete step-by-step example of how the learning process works, please refer to Appendix B.

6 Training utilities

The most useful function provided by the `trainutils.py` module is `train_test_split`. This function takes as input two tensors (input data and target data) and splits them into training and test sets.

Other notable functions include `normalize_features` and `denormalize_features`, which are used to normalize and denormalize the input and target features, respectively; `one_hot_encode` and `one_hot_decode`, which are used to convert categorical labels into one-hot encoded vectors and vice versa; and `accuracy`, which computes the accuracy of predictions for classification tasks, i.e., the number of correct predictions divided by the total number of samples.

7 Training

The `train.py` module provides three convenience functions for training and evaluating neural networks: `train`, `train_test_regression`, and `train_test_classification`.

Overview All functions implement the standard training cycle over epochs:

1. iterate over the training set in mini-batches (see Section 5.3);

2. forward pass to obtain predictions;
3. compute the loss;
4. backward pass to compute gradients;
5. optionally clip gradients if a maximum norm is specified (before the optimizer step; see Section 5);
6. update parameters with the optimizer.

At the end of each epoch, the loss is averaged over all batches and appended to a history that is returned to the caller.

APIs and returned metrics

- `train`: runs training only and returns the per-epoch training loss history.
- `train_test_regression`: trains and evaluates on a test set each epoch; returns training loss and test loss histories.
- `train_test_classification`: as above, plus computes test accuracy each epoch using `accuracy` (see Section 6); returns training loss, test loss, and test accuracy histories.

Evaluation runs a forward pass on the test set without parameter updates.

Part II
Comparison with `PyTorch`

In order to check that all the implementation of `mfnet` is correct, two simple tasks have been performed and compared with `PyTorch`: a regression task on the California Housing dataset and a classification task on the MNIST dataset.

The two tasks have been implemented as closely as possible in both libraries, using the same architecture, loss function and optimizer. Training hyperparameters such as learning rate and batch size were different between the two frameworks, as using the same values for both resulted in a more unstable training for one of the two libraries.

Note that the goal of the project was not to outperform `PyTorch`, but rather to compare the training process of `mfnet` with that of a production machine learning framework and check that the overall behavior is consistent with the expectations of a functioning library.

The code for the comparison can be found on GitHub.

1 Regression on California Housing

After the data was loaded and split in training and test sets, feature normalization was performed on the training set. Using the statistics learned from the training set, the test set was normalized in the same way.

The training hyperparameters used for both libraries are shown in the table below.

Hyperparameter	`mfnet`	`PyTorch`
Number of Epochs	500	500
Learning Rate η	0.001	0.001
Batch Size	1024	1024

For each library, three models were trained and compared:

1. a baseline mean predictor that always predicts the mean value of the training set's targets;
2. a linear regression model;
3. a neural network with one hidden layer of 512 neurons and ReLU activation functions. For `mfnet` only, maximum gradient norm was set to 5 in order to prevent overflow errors.

The learning curves for both libraries are shown in Figs. 2 and 3.

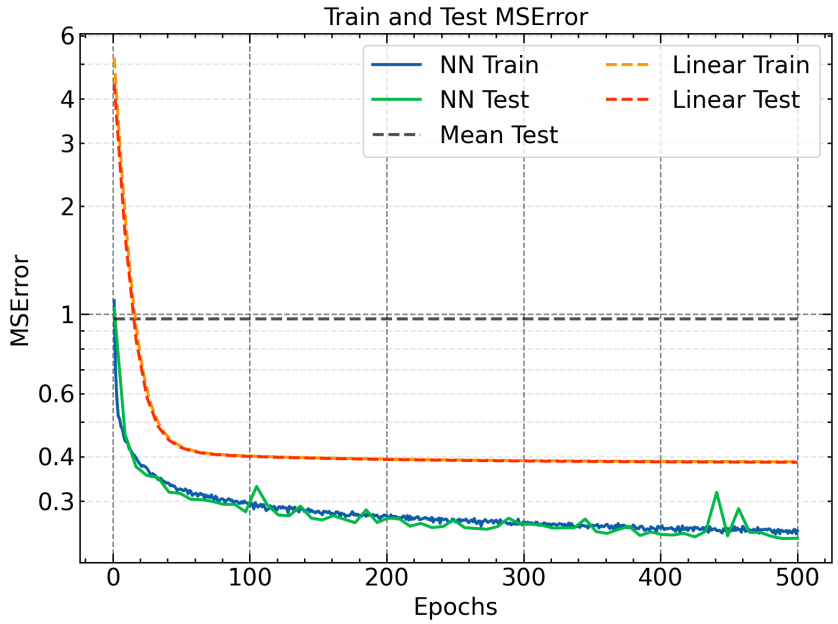

Figure 2: Learning curves for regression task on California Housing dataset using `mfnet`.

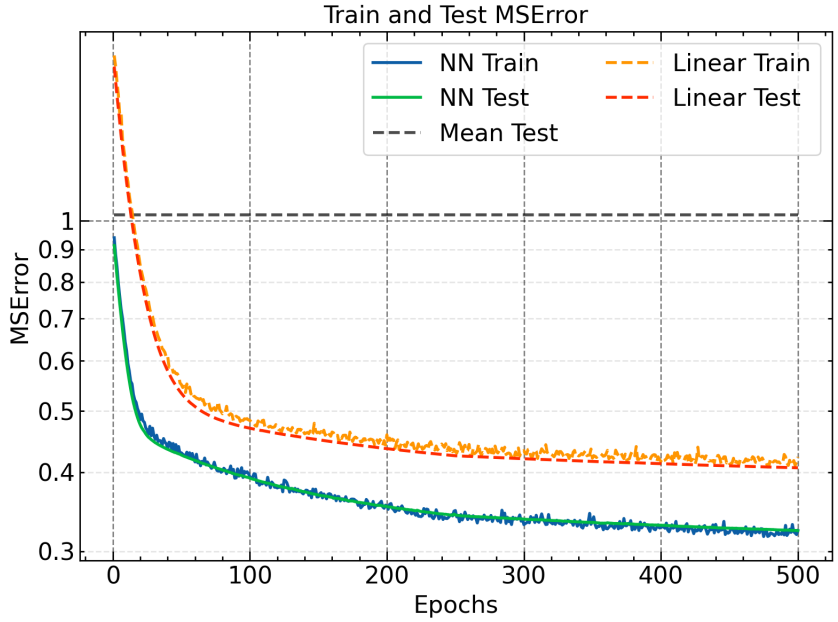

Figure 3: Learning curves for regression task on California Housing dataset using `PyTorch`.

2 Classification on MNIST

After loading the data (already split in training and test sets), the pixel values were normalized to the range [0, 1]. `mfnet` also required the data to be transformed in such a way that the library would be able to understand. This meant converting input data to tensors and one-hot encoding the target labels. For both libraries the 28×28 px images were flattened into 784-dimensional vectors.

The training hyperparameters used for both libraries are shown in the table below.

Hyperparameter	`mfnet`	`PyTorch`
Number of Epochs	100	100
Learning Rate η	0.001	0.01
Batch Size	1024	128

For each library, two models were trained and compared:

1. a linear classification model;
2. a neural network with one hidden layer of 512 neurons and ReLU activation functions. For `mfnet` only, maximum gradient norm was set to 5 in order to prevent overflow errors.

The learning curves and test accuracies for both libraries are shown in Figs. 4 and 5.

The accuracy achieved by `PyTorch` is significantly lower than the one achieved by `mfnet`, so the confusion matrices for both libraries are computed and shown in Figs. 6 and 7. The figures clearly show that, while `mfnet` is able to correctly classify almost all the digits, `PyTorch` struggles with most of them (particularly the digits 0 and 8). This is likely due to insufficient exploration of the hyperparameter space, which wasn't carried out since it was outside the scope of this project.

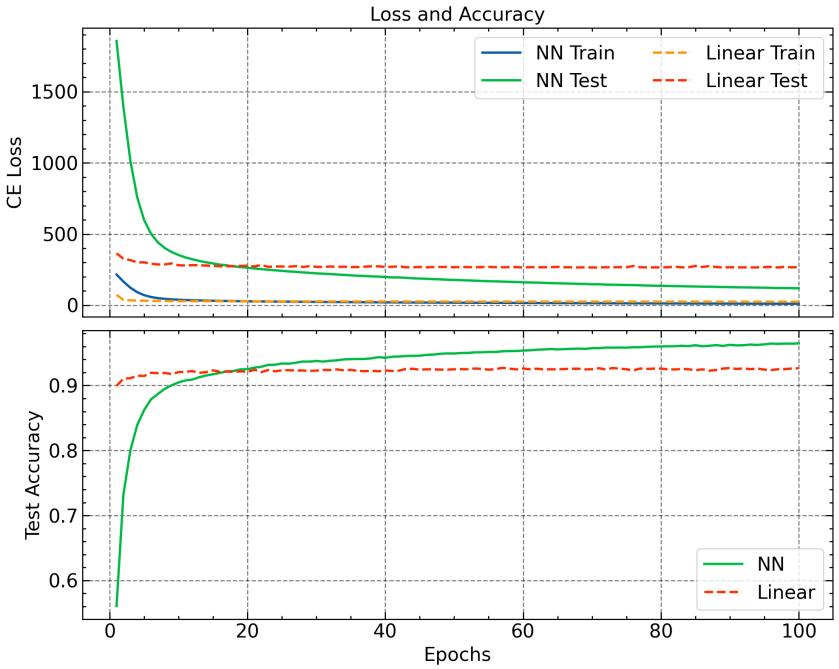

Figure 4: Learning curves for classification task on MNIST dataset using `mfnet`.

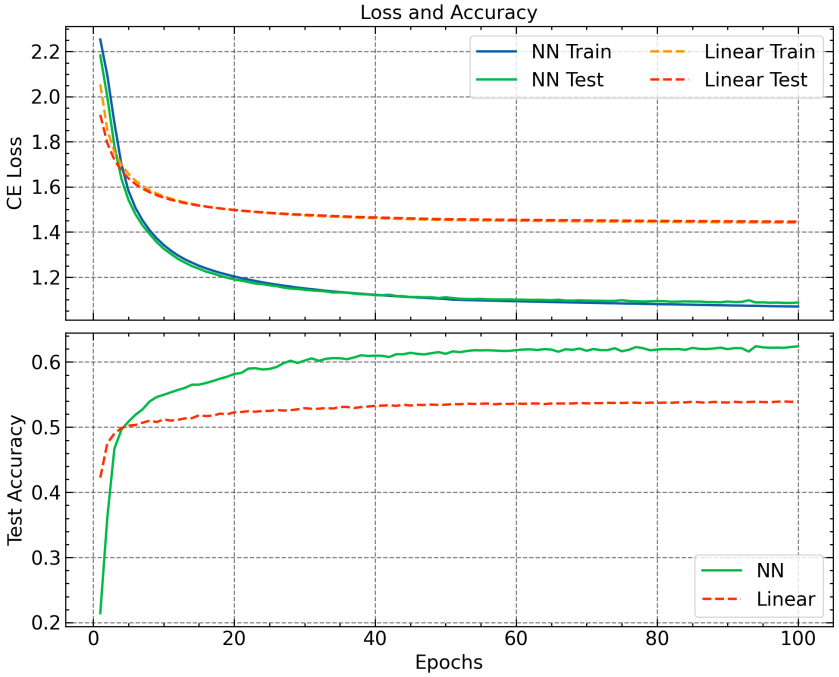

Figure 5: Learning curves for classification task on MNIST dataset using `PyTorch`.

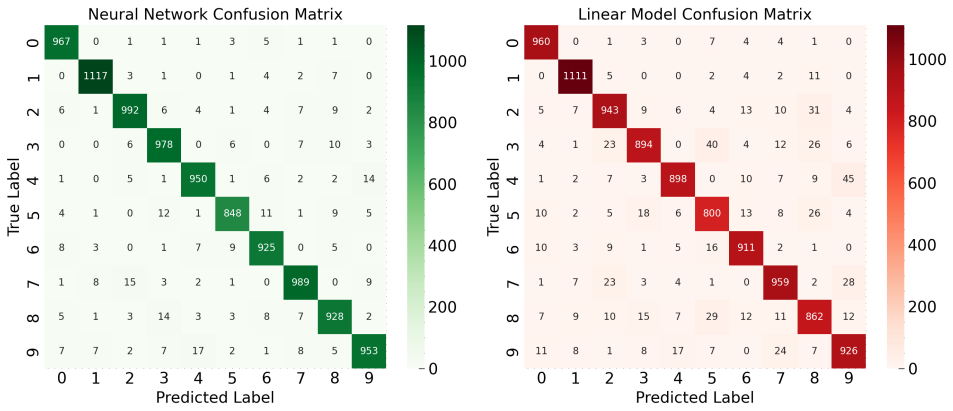

Figure 6: Confusion matrix for classification task on MNIST dataset using `mfnet`.

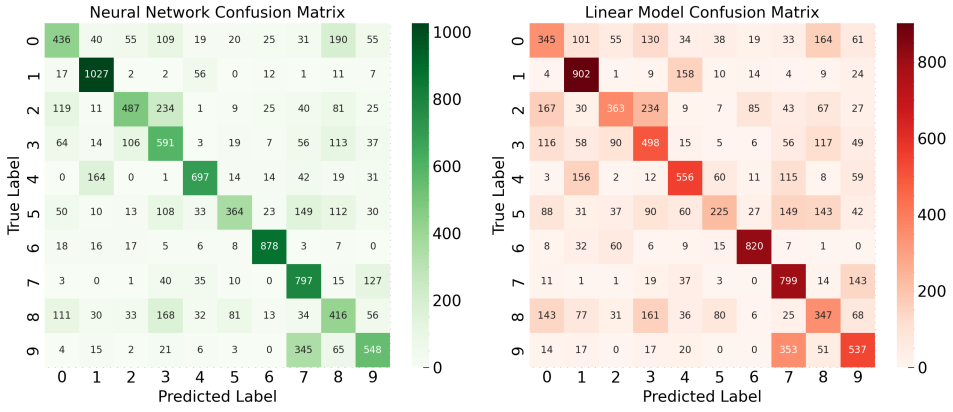

Figure 7: Confusion matrix for classification task on MNIST dataset using `PyTorch`.

Appendices

A Brief note on training times

The training-loop execution times for both libraries are reported in Tables 1 and 2. All runs were executed on a MacBook Pro (2019) with a 2.3 GHz Quad-Core Intel Core i7 CPU. The execution times are taken from the VSCode Jupyter extension and are not the result of an average, but they are the one-shot execution time of the training loop only (without data preprocessing), and as such they represent only an order of magnitude and not precise measurements.

Model	`mfnet`	`PyTorch`
Baseline Mean Predictor	0.0 s	0.0 s
Linear Regression	1.1 s	66.0 s
Neural Network	274.7 s	136.8 s

Table 1: Execution times for regression training loops.

Model	`mfnet`	`PyTorch`
Linear Classification	150.6 s	709.4 s
Neural Network	423.7 s	813.6 s

Table 2: Execution times for classification training loops.

On this CPU and with the implementation used, `mfnet` was faster in three out of four configurations, while `PyTorch` was faster on the Neural Network regression. These results can be influenced by several factors, including implementation details, number of CPU threads, and batch size.

Methodology and fairness Timings refer to the training loops only (model forward/backward and updates), excluding data download and preprocessing. For stricter comparability:

- align batch sizes and learning rates across libraries;
- ensure both libraries use the same number of CPU threads;
- note that gradient clipping was enabled only for `mfnet` in some settings, which may change step time;
- report averages over multiple runs (with standard deviation) when possible.

Differences here should not be generalized to other hardware or larger models: `PyTorch` benefits from highly optimized kernels and vectorized data pipelines, whereas `mfnet` is a simple and instructive CPU-only implementation.

B Step-by-step example of an epoch

To better understand how the bias is handled throughout `mfnet`, it's helpful to consider a practical example: we'll walk through a forward/backward cycle of a small network with two hidden layers and all the activation functions set to the identity.

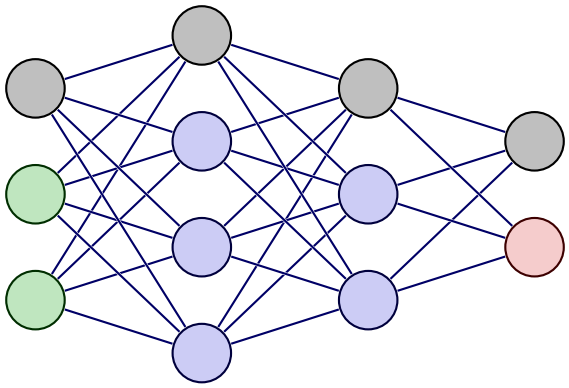

Figure 8: A small Neural Network with two hidden layers. The gray neurons represent the bias features.

Let's pick an input X and target Y:

$$X = \begin{bmatrix} 1 & 2 \\ 1 & 2 \\ 2 & 2 \\ 2 & 3 \end{bmatrix}, \quad Y = \begin{bmatrix} 1 \\ 1 \\ 2 \\ 2 \end{bmatrix}$$

This is a small dataset of four samples with two input features and one target feature.

First, the `DataLoader` transposes the data and prepends the bias feature:

$$A^{[0]} = \tilde{X} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 2 & 2 \\ 2 & 2 & 2 & 3 \end{bmatrix}, \quad \tilde{Y} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 2 & 2 \end{bmatrix}$$

B.1 Forward pass

Now the `forward` method of the `NeuralNetwork` is called with input data $A^{[0]}$.

Layer 1 (Linear) The first layer is a Linear layer with 2 input features (plus bias) and 3 output features (plus bias). Suppose the weights matrix $W^{[1]}$ is:

$$W^{[1]} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

Then, the output of the layer is:

$$Z^{[1]} = W^{[1]} A^{[0]} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 3 & 2 & 3 \\ 3 & 4 & 4 & 5 \\ 2 & 2 & 3 & 3 \end{bmatrix}$$

Layer 1 (Activation) The activation function is applied element-wise, skipping the first row:

$$A^{[1]} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ g(Z^{[1]}[1{:}]) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 3 & 2 & 3 \\ 3 & 4 & 4 & 5 \\ 2 & 2 & 3 & 3 \end{bmatrix}$$

Layer 2 (Linear) The second layer is a Linear layer with 3 input features (plus bias) and 2 output features (plus bias). Suppose the weights matrix $W^{[2]}$ is:

$$W^{[2]} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 \\ -1 & 1 & 2 & -2 \end{bmatrix}$$

Then, the output of the layer is:

$$Z^{[2]} = W^{[2]} A^{[1]} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 5 & 6 & 6 & 7 \\ 3 & 6 & 3 & 6 \end{bmatrix}$$

Layer 2 (Activation) The activation function is applied element-wise, skipping the first row:

$$A^{[2]} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ g(Z^{[2]}[1{:}]) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 5 & 6 & 6 & 7 \\ 3 & 6 & 3 & 6 \end{bmatrix}$$

Layer 3 (Linear) The third layer is a Linear layer with 2 input features (plus bias) and 1 output feature (plus bias). Suppose the weights matrix $W^{[3]}$ is:

$$W^{[3]} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & -1 \end{bmatrix}$$

Then, the output of the layer is:

$$Z^{[3]} = W^{[3]} A^{[2]} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 0 & 3 & 1 \end{bmatrix}$$

Layer 3 (Activation) The activation function is applied element-wise, skipping the first row:

$$\hat{Y} = A^{[3]} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ g(Z^{[3]}[1{:}]) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 0 & 3 & 1 \end{bmatrix}$$

Now we compute the loss, for instance MSE:

$$\begin{aligned} \mathcal{J}(\hat{Y}, \tilde{Y}) &= \frac{1}{m} \sum_{i=1}^{m} \left\| \hat{Y}_i - \tilde{Y}_i \right\|^2 \\ &= \frac{1}{4} \left[(1-1)^2 + (1-1)^2 + (1-1)^2 + (1-1)^2 + \right. \\ &\qquad \left. + (2-1)^2 + (0-1)^2 + (3-2)^2 + (1-2)^2 \right] = 1 \end{aligned}$$

B.2 Backward pass

To start the backward pass, we need to compute the gradient of the loss with respect to the output of the network:

$$\frac{\partial \mathcal{J}}{\partial \hat{Y}} = \frac{2}{m} \left(\hat{Y} - \tilde{Y} \right) = \frac{1}{4} \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & -1 & 1 & -1 \end{bmatrix} = \frac{\partial \mathcal{J}}{\partial A^{[3]}}$$

This is now the input of the `backward` method of the `NeuralNetwork`.

Layer 3 (Activation) The backward method of the Activation layer computes the gradient of the loss with respect to its input $Z^{[3]}$. Since the activation function is the identity, its derivative is 1, and we have:

$$\Delta^{[3]} = \frac{\partial \mathcal{J}}{\partial Z^{[3]}} = \frac{\partial \mathcal{J}}{\partial A^{[3]}} \odot \begin{bmatrix} 0 & 0 & 0 & 0 \\ g'(Z^{[3]}[1{:}]) \end{bmatrix} = \frac{1}{4} \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & -1 & 1 & -1 \end{bmatrix}$$

Layer 3 (Linear) The backward method of the Linear layer computes the gradient of the loss with respect to its weights $W^{[3]}$ and its input $A^{[2]}$:

$$\frac{\partial \mathcal{J}}{\partial W^{[3]}} = \Delta^{[3]} A^{[2]T} = \frac{1}{4}\begin{bmatrix} 0 & 0 & 0 \\ 0 & -2 & -6 \end{bmatrix}$$

$$\frac{\partial \mathcal{J}}{\partial A^{[2]}} = W^{[3]T} \Delta^{[3]} = \frac{1}{4}\begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & -1 & 1 & -1 \\ -1 & 1 & -1 & 1 \end{bmatrix}$$

Layer 2 (Activation) The backward method of the Activation layer computes the gradient of the loss with respect to its input $Z^{[2]}$. Since the activation function is the identity, its derivative is 1, and we have:

$$\Delta^{[2]} = \frac{\partial \mathcal{J}}{\partial Z^{[2]}} = \frac{\partial \mathcal{J}}{\partial A^{[2]}} \odot \begin{bmatrix} 0 & 0 & 0 & 0 \\ g'(Z^{[2]}[1{:}]) \end{bmatrix} = \frac{1}{4}\begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & -1 & 1 & -1 \\ -1 & 1 & -1 & 1 \end{bmatrix}$$

Layer 2 (Linear) The backward method of the Linear layer computes the gradient of the loss with respect to its weights $W^{[2]}$ and its input $A^{[1]}$:

$$\frac{\partial \mathcal{J}}{\partial W^{[2]}} = \Delta^{[2]} A^{[1]T} = \frac{1}{4}\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & -2 & -2 & 0 \\ 0 & 2 & 2 & 0 \end{bmatrix}$$

$$\frac{\partial \mathcal{J}}{\partial A^{[1]}} = W^{[2]T} \Delta^{[2]} = \frac{1}{4}\begin{bmatrix} 2 & -2 & 2 & -2 \\ 0 & 0 & 0 & 0 \\ -2 & 2 & -2 & 2 \\ 3 & -3 & 3 & -3 \end{bmatrix}$$

Layer 1 (Activation) The backward method of the Activation layer computes the gradient of the loss with respect to its input $Z^{[1]}$. Since the activation function is the identity, its derivative is 1, and we have:

$$\Delta^{[1]} = \frac{\partial \mathcal{J}}{\partial Z^{[1]}} = \frac{\partial \mathcal{J}}{\partial A^{[1]}} \odot \begin{bmatrix} 0 & 0 & 0 & 0 \\ g'(Z^{[1]}[1{:}]) \end{bmatrix} = \frac{1}{4}\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ -2 & 2 & -2 & 2 \\ 3 & -3 & 3 & -3 \end{bmatrix}$$

Layer 1 (Linear) The backward method of the Linear layer computes the gradient of the loss with respect to its weights $W^{[1]}$ and its input $A^{[0]}$:

$$\frac{\partial \mathcal{J}}{\partial W^{[1]}} = \Delta^{[1]} A^{[0]T} = \frac{1}{4}\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 4 \\ 0 & 0 & -6 \end{bmatrix}$$

$$\frac{\partial \mathcal{J}}{\partial A^{[0]}} = W^{[1]T} \Delta^{[1]} = \frac{1}{4}\begin{bmatrix} 3 & -3 & 3 & -3 \\ 1 & -1 & 1 & -1 \\ -2 & 2 & -2 & 2 \end{bmatrix} \quad \text{(unused)}$$

B.3 Weight update

The weights are updated using the SGD optimizer:

$$W^{[l]} \leftarrow W^{[l]} - \eta \frac{\partial \mathcal{J}}{\partial W^{[l]}}$$

Setting the learning rate $\eta = 4$ for simplicity[1], we have:

$$W^{[3]} \leftarrow \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & -1 \end{bmatrix} - \begin{bmatrix} 0 & 0 & 0 \\ 0 & -2 & -6 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 3 & 5 \end{bmatrix}$$

$$W^{[2]} \leftarrow \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 \\ -1 & 1 & 2 & -2 \end{bmatrix} - \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & -2 & -2 & 0 \\ 0 & 2 & 2 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 3 & 2 & 1 \\ -1 & -1 & 0 & -2 \end{bmatrix}$$

$$W^{[1]} \leftarrow \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \end{bmatrix} - \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 4 \\ 0 & 0 & -6 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & -3 \\ 1 & 0 & 6 \end{bmatrix}$$

Now a new cycle of forward pass/backward pass/weight update can begin.

[1] This value is way too high to have any chance of yielding an improvement in the predictions in any practical example.