UNIVERSITÀ
DI TORINO

Deep Learning

Francesco Marchisotti

`MFNET` – A simple Machine Learning Library

Professor:
Matteo Osella

academic year 2024/2025

Abstract

This document presents `mfnet`, a simple machine learning library developed as a final project for the Deep Learning course. The key feature is the implementation of the backpropagation algorithm, enabling the training of neural networks through gradient descent.

The library is then compared with PyTorch on two simple tasks: a regression (on the California Housing dataset) and a classification (on the MNIST dataset).

Contents

Signal Flow in Neural Networks

A Feedforward Fully Connected Neural Network (FCNN) is a Neural Network architecture where each neuron in one layer is connected to every neuron in the subsequent layer.

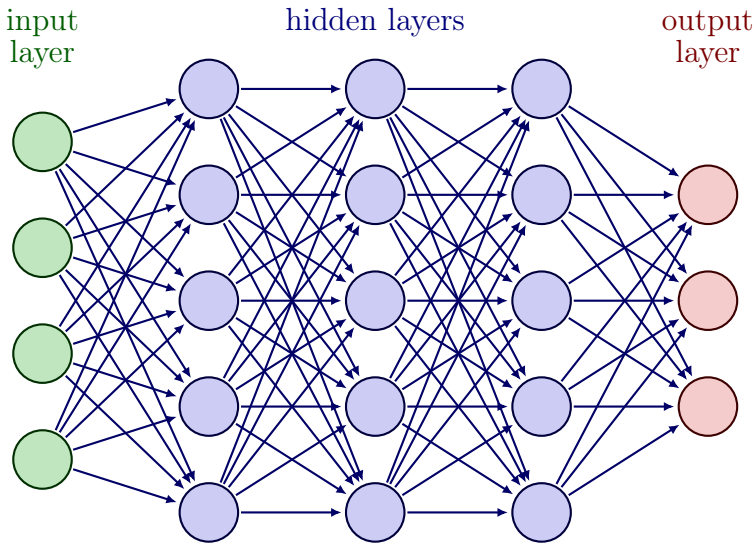

Figure 1: A Feedforward Fully Connected Neural Network (FCNN).

A FCNN learns by iteratively performing two main steps: the forward pass and the backward pass.

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`

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.

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 algorithm is more easily understandable using the indices notation, but in order to implement it in code, we need to express it in matrix form, so both formulations will be shown.

We define m to be the number of samples, $n^{[l]}$ the number of neurons in layer l, $a_j^{[l]}$ the activation of neuron j in layer l (with $a_1^{[l]} = 1$), and $W_{jk}^{[l]}$ the weight connecting neuron k in layer $l-1$ to neuron j in layer l (with $W_{j1}^{[l]} = b_j^{[l]}$ and $W_{1k}^{[l]} = \delta_{1k}$).

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.

Indices notation 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 weights $W_{jk}^{[l]}$ 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 in 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 $\frac{\partial \mathcal{J}}{\partial W_{jk}^{[l]}}$.

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

The flow of information through layer l is given by:

$$Z^{[l]} = W^{[l]} A^{[l-1]}$$
$$A^{[l]} = g\left(Z^{[l]}\right)$$

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

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

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

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

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, and thus every $\frac{\partial \mathcal{J}}{\partial W^{[l]}}$.

3 Layer

In `mfnet`, a layer is a fundamental building block of the neural network. Each layer consists of a set of neurons, and it 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.

Each of the layers in `mfnet` expects the input to be in the form of a matrix with shape $(n_{\text{features}} + 1, n_{\text{samples}})$, where n_{features} is the number of input features and n_{samples} is the number of input samples. The first row of this matrix is reserved for a bias term, which is always set to 1.

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 $n_{\text{in\_features}}$ to $n_{\text{out\_features}}$. Mathematically, this can be represented as:

$$y = b + Wx \tag{1}$$

where:

- x is the $(n_{\text{in\_features}}, 1)$ input vector,
- W is the $(n_{\text{out\_features}}, n_{\text{in\_features}})$ weight matrix,
- b is the $(n_{\text{out\_features}}, 1)$ bias vector, and
- y is the $(n_{\text{out\_features}}, 1)$ 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:

$$\begin{pmatrix}1\\ y\end{pmatrix} = \begin{pmatrix}1 & 0 \cdots 0\\ b & W\end{pmatrix}\begin{pmatrix}1\\ x\end{pmatrix} \tag{2}$$

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 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:

$$\begin{pmatrix} 1 \cdots 1 \\ Y \end{pmatrix} = \begin{pmatrix} 1 & 0 \cdots 0 \\ b & W \end{pmatrix} \begin{pmatrix} 1 \cdots 1 \\ X \end{pmatrix} \tag{3}$$

Switching to standard backpropagation notation, we can summarize the forward pass of a Linear layer as:

$$Z^{[l]} = W^{[l]} A^{[l-1]} \tag{4}$$

where:

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

All these Tensors already include all 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 is to compute the gradients of the loss with respect to its weights during the backward pass. Mathematically, this can be expressed as:

$$\frac{\partial \mathcal{J}}{\partial W^{[l]}} = \frac{\partial \mathcal{J}}{\partial Z^{[l]}} A^{[l-1]T} = \Delta^{[l]} A^{[l-1]T} \tag{5}$$

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.

The Linear layer needs to pass backward its contribution to the gradient of the loss with respect to its input $A^{[l-1]}$. This is computed as:

$$\frac{\partial \mathcal{J}}{\partial A^{[l-1]}} = W^{[l]T} \Delta^{[l]} \tag{6}$$

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. This non-linearity is crucial for the neural network to learn complex patterns in the data.

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 $A^{[l]}$:

$$A^{[l]} = g(Z^{[l]}) \tag{7}$$

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 $Z^{[l]}$:

$$\frac{\partial \mathcal{J}}{\partial Z^{[l]}} = \frac{\partial \mathcal{J}}{\partial A^{[l]}} \odot g'(Z^{[l]}) \tag{8}$$

where g' is the derivative of the activation function and $\odot$ denotes element-wise multiplication (Hadamard product).

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 first row (the bias feature) 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.

3.3 Example

To better understand the workings of the layers, 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.

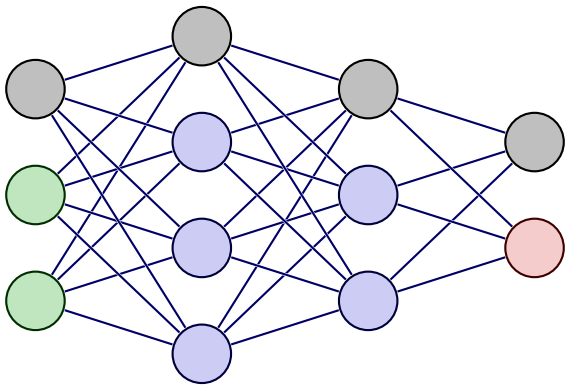

Figure 2: A small neural network with two hidden layers. The first layer is a Linear layer with 2 input features (plus bias) and 3 output features (plus bias). The second layer is a Linear layer with 3 input features (plus bias) and 2 output features (plus bias). The third layer is a Linear layer with 2 input features (plus bias) and 1 output feature (plus bias). All activation functions are set to the identity.

Let's pick an input X and target Y:

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

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

The first step is to prepend the bias feature to the input and to the output:

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

3.3.1 Forward pass

Layer 1 (Linear) The first layer is a Linear layer with 3 input features (plus bias) and 4 output features (plus bias). The weights matrix $W^{[1]}$ is initialized randomly:

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

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

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 4 input features (plus bias) and 3 output features (plus bias). The weights matrix $W^{[2]}$ is initialized randomly:

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

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

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 3 input features (plus bias) and 2 output features (plus bias). The weights matrix $W^{[3]}$ is initialized randomly:

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

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

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:

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

3.3.2 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 last layer.

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 $A^{[0]}$:

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

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

3.3.3 Weight update

The weights are updated using gradient descent:

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

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.

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 (MSE)

The Mean Squared Error loss function is defined as:

$$\mathcal{J}(\hat{y}, y) = \frac{1}{m} \sum_{i=1}^{m} \|\hat{y}_i - y_i\|^2 = \frac{1}{m} \sum_{i=1}^{m} \sum_{j=1}^{n_{\text{feat}}} (\hat{y}_{ij} - y_{ij})^2 \tag{9}$$

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:

$$\frac{\partial \mathcal{J}}{\partial \hat{y}} = \frac{2}{m}(\hat{y} - y) \tag{10}$$