Demystifying Deep Learning: Intuitive Comprehension for Building Neural Networks with only Numpy

Sékou Dabo[*†]

March 16, 2024

Abstract

With the rapid advancements in artificial intelligence, particularly in the field of Deep Learning, new technologies are emerging across various sectors. Examples include natural language processing with OpenAI's *ChatGPT*[1] for conversations, molecular generation with Google DeepMind's *AlphaFold*[2] for predicting and understanding the 3D structure of molecules, and medical diagnostics with Google's *Med-PaLM*[3] to address medical queries.

These breakthroughs are sparking a growing interest in artificial intelligence. However, the complexity of underlying architectures and mathematical concepts can be daunting for newcomers. This article aims to demystify this complexity by demonstrating that building AI models is nothing more than assembling intuitive and elementary architectures. With reasonable effort and a level of mathematical proficiency, including algebra, statistics, probabilities, derivatives, and optimization, constructing robust artificial intelligence applications is attainable.

Simultaneously, this article aims to validate acquired skills from my academic training and self-directed learning, notably through freely accessible courses like Andrew Ng's *Deep Learning Specialization*[4] and consulting prominent articles in the field.

To illustrate these concepts, the article leverages the versatility of NumPy, a widely adopted scientific computing library in various fields such as physics, chemistry, biology, and many others. Using NumPy, we build two simple neural networks one for a classification task and another for a regression task. Subsequently, we will train them on the MNIST dataset for handwritten digit recognition and on the Paris Housing dataset from the 2021 Kaggle competition[5] for predicting real estate prices in Paris. Finally, we discuss results. All of this is to emphasize that creating deep neural networks is not as daunting as it may seem, and the fundamentals of artificial intelligence can be accessible to a broader audience.

[*]Correspondence: sekoudabo884@gmail.com

[†]Code: https://github.com/ds4xai/nn-with-numpy-4-intuitive-comprehension

[1]https://help.openai.com/en/articles/6783457-what-is-chatgpt

[2]https://deepmind.google/technologies/alphafold/

[3]https://sites.research.google/med-palm/

[4]https://www.coursera.org/specializations/deep-learning/

[5]https://www.kaggle.com/datasets/mssmartypants/paris-housing-price-prediction

Introduction

In the realm of Deep Learning, the use of frameworks like TensorFlow and PyTorch significantly simplifies the model development process. However, constructing a neural network from scratch enhances our understanding of its actual functioning, providing several meaningful advantages:

- **Reinforced Intuition:** A deep understanding of the internal workings of neural networks.
- **Framework Mastery:** Better expertise with high-level frameworks like PyTorch, Keras, and TensorFlow.
- **Rapid Iteration:** Ease in generating, testing, and quickly interpreting ideas through strong intuition and defining relevant metrics for our problem.
- **Robust Application:** The ability to define appropriate metrics to ensure model consistency when deployed in the real world.
- **Explainability:** A profound understanding of model actions, facilitating the explanation of the adopted approach to others.
- **Effective Debugging:** Increased ability to interpret and quickly rectify errors.
- **Adaptability:** Ease in reusing and adjusting open-source models, whether through transfer learning or fine-tuning.

With this perspective in mind, this article explores the construction and training of a basic neural network, exclusively using the Numpy library. Following this approach, I strongly recommend anyone entering the field of Deep Learning to build their own neural network. When testing this network, carefully examining the flow of information between layers during forward and backward propagation in backpropagation is crucial.

Understanding the effects of weight initialization, modifying activation functions, hyperparameters, and the functioning of the gradient descent optimization algorithm and its variants is essential for a successful experience in the field.

In the body of this article, some of these terms will be developed more extensively than others to maintain conciseness. The main focus lies in the implementation of the neural network. For more in-depth information, I highly recommend Andrew Ng's *Deep Learning Specialization* series.

The remainder of this article will cover:

1. The Basics of Deep Learning: Understanding fundamental concepts such as neurons, layers, a neural network, weights and biases, training, and the gradient descent optimization algorithm.
2. Building and train a Neural Network: Creating a neural network from scratch using Numpy and train our network on real data.

1 Fundamentals of Deep Learning

1.1 Definition: (Deep Learning)

Deep learning, is a branch of artificial intelligence that utilizes various network **topologies (architectures)** of neural networks to enable computers to learn autonomously for a variety of **tasks**.

Let's take a closer look at the highlighted terms.

- **Topology or Architecture:** Depending on the nature and arrangement of neurons and how data is transformed (1D or 2D transformation) from layer to layer, neural architectures can be classified, such as Multilayer Perceptron, MLP [1], Convolutional Neural Networks, CNNs [1, 2], recurrent architectures like Recurrent Neural Network, RNN [1, 3], Gated Recurrent Units, GRU, and Long Short-Term Memory, LSTM [1,3], as well as more recent architectures like Transformers [4], Graph Neural Networks GNNs [5], Generative Adversarial Networks GANs [6], or "hybrid" architectures like RWKV [7] (transformers + RNN).
- **Learning and Training:** Learning algorithms categorize based on the type of data used for model training. Several categories include: *Supervised Learning:* Training examples are pre-labeled by experts, *unsupervised Learning:* Training examples are not labeled, *self-supervised Learning:* Training examples are also used as labels, *reinforcement Learning:* The agent learns from experiences in an environment, *adversarial Learning:* An adversarial agent attempts to reproduce the distribution of training data, *contrastive or Multimodal Learning:* Training data involves different modalities, such as text and images, *transfer Learning:* A model uses knowledge acquired from previous tasks to address new tasks or domains sharing similarities.
- **Task or Problem:** Depending on the problem to solve, it can involve *regression* (predicting a continuous value, for example, the NPK[6] level of agricultural soils), *classification* (predicting membership to a category, for example, the nature of a cancerous pathology, determining whether it is malignant or benign), or dimensionality reduction. *Dimensionality reduction* aims to decrease the number of variables while preserving essential information. For example, in the field of medical imaging, a large set of volumetric images can be reduced to a set of significant features, facilitating analysis while saving computational resources. It can also involve *clustering*. In clustering, groups are defined by the algorithm itself based on a similarity or dissimilarity metric. Depending on the set threshold, groups may evolve, unlike classification where groups are fixed from the start by humans.For example, consider an algorithm clustering café customers based on a metric indicating whether they consume coffee, milk, or both. By defining an inclusive metric, the algorithm will construct two groups: the first with individuals solely drinking coffee, possibly some consuming café au lait, and the second with those solely consuming milk, possibly some drinking café au lait. Conversely, with an exclusive metric, three groups will be created: the first with individuals solely drinking coffee, the second with those solely consuming milk, and the third with those drinking café au lait.

This list is not exhaustive, given the constant evolution of the field where new concepts regularly emerge. The segmentation of certain notions can also vary depending on the sensitivity of engineers and researchers.

[6] https://en.wikipedia.org/wiki/Fertilizer

1.2 Definition: (Neuron)

In the context of deep learning, a neuron, also called an artificial neuron or perceptron, consists of two operations:

1. A linear transformation of its inputs or a weighted sum to which a bias is added.
2. Passing the result through an activation function.

The visual representation of a neuron can vary, but the one we propose is as follows:

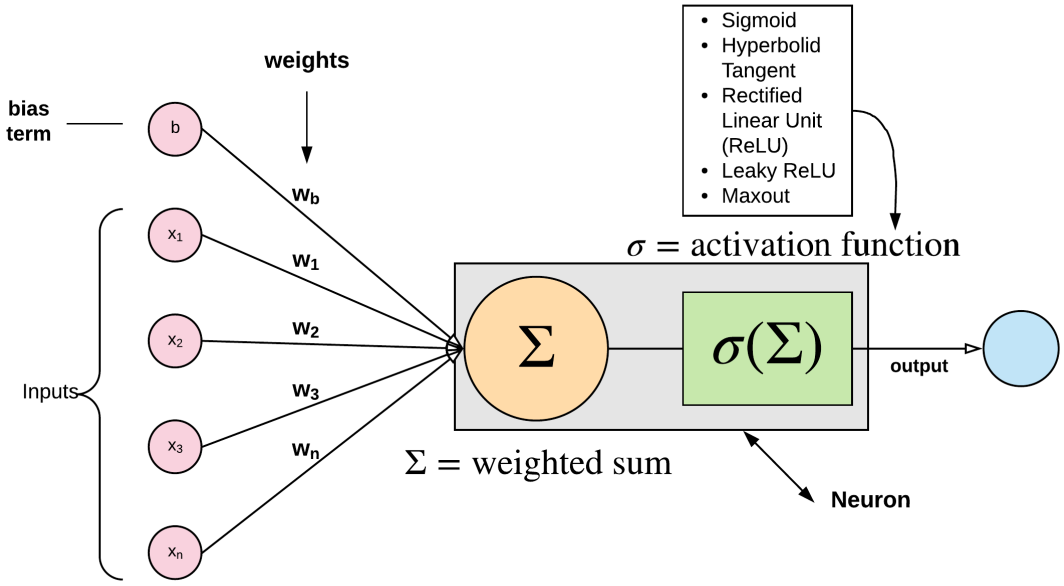


Figure 1: artificial neuron

where x_i are the inputs or activation of previous layer, w_i are the weights, b is the bias and σ is the activation function.
From another perspective, during our high school years, almost all of us have encountered logistic regression. So, to recall, you can think of a perceptron as a logistic regression where the activation function is the sigmoid.

However, a single neuron has limitations in solving relatively complex problems. To overcome these limitations, it is necessary to stack neurons to form layers.

1.3 Definition: (Layer)

A layer is a stack of units (neurons). These units are not always classic neurons. As mentioned earlier, they can be of different types or forms: If the units are traditional neurons, we have a *classic layer*, if the units have memory or are recurrent, we have a *recurrent layer*, if the units are convolution masks, we have either a *convolution layer* or a *pooling layer*, and iff the units are an attention matrix, we have an *attention layer.*,

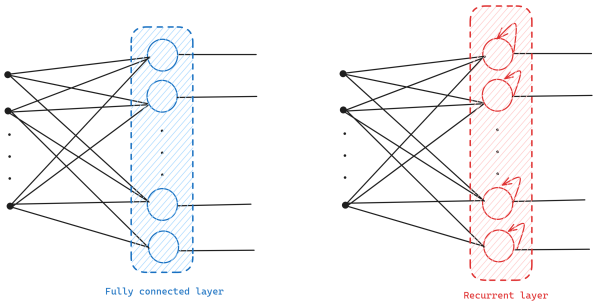


Figure 2: Fully connected and recurrent layer.

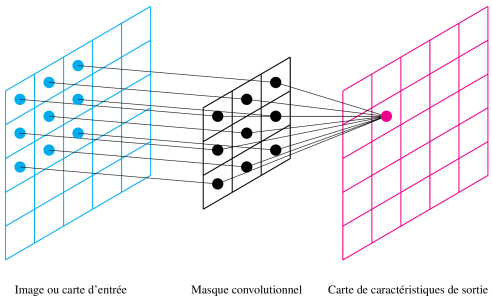


Figure 3: Convolution layer.

source: openclassrooms.com [8]

Layers can also be classified into three categories based on their position in the architecture, if a layer is at the beginning, it is designated as the *input layer*, if it is at the end, it is the *output layer*, and if it is between the two previous layers, it is a *hidden layer*, see figure 4.

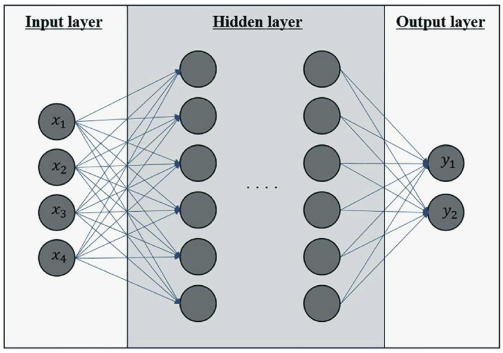


Figure 4: Nature of layers

source: researchgate.net [9]

In the literature, the success of deep learning is frequently associated with big data (a large amount of data available due to our digital footprint or the evolution of sensor technologies, among other factors) and the accessibility of highly advanced computing technologies, a fully justified observation. However, in the background, the introduction of various activation functions also plays a significant role in this success. The following sections of this article will present this aspect through a concrete example.

Firstly, an overview of the main activation functions used in deep learning will be presented.

1.4 Activation Function

An activation function is a mathematical function applied to the output of a linear transformation of an artificial neuron. It plays a crucial role in neural networks by introducing non-linear transformations to learn abstract features[7]. Its key properties include: a) Adding non-linear patterns during network training. b) Maintaining the gradient within a suitable range to avoid gradient explosion or

[7] https://en.wikipedia.org/wiki/Activation_function

vanishing problems. c) Preserving the data distribution despite transformations, facilitating better network training.

Below in Figure 5, is a list of the main activation functions.

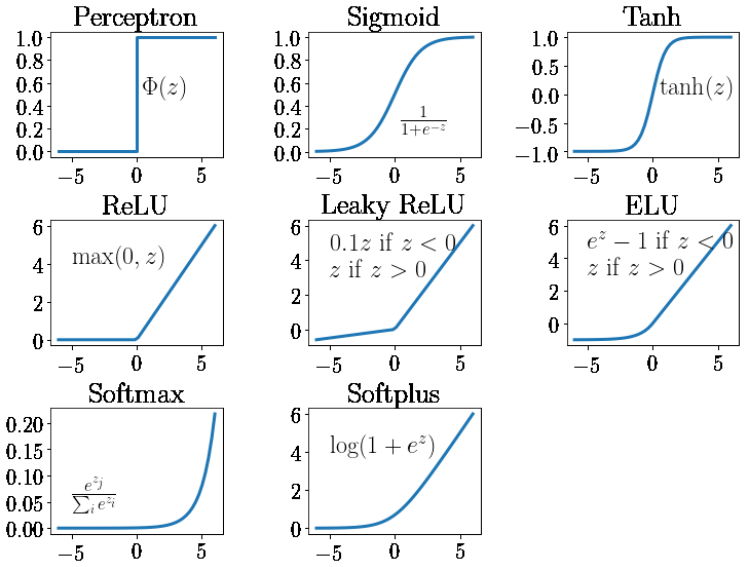


Figure 5: main activation functions

source:researchgate.net [10]

The most important property is the introduction of non-linearity into the function learned during training, as illustrated in this example.

Example: Suppose, for artistic reasons, we want to reproduce an accurate representation of the geographic map of Gabon. Or, for geopolitical reasons, we aim to develop an application capable of instantly detecting if a person has crossed the borders of Gabon.

In this context, we have collected geographical coordinates data (longitude and latitude), which we then labeled in yellow to indicate they are within the Gabonese territory and in green for those located in other states. Each data point in our dataset is represented by $(x_{1,i}, y_{1,i})$, where $x_i = (x_{1,i}, x_{1,i})$ represents latitude $(x_{1,i})$ and longitude $(x_{1,i})$, and y_i is the label of example i.

Thus, we are in a *supervised learning* case, seeking to determine if a given point belongs to the Gabonese territory or not, a *classification task*. We use a fully connected MLP, and in the first case, we use an identity activation function, meaning we apply no activation function, letting the network's output be the direct result of the linear transformation. In the second case, we train another network using an activation function to introduce non-linearity. The results are illustrated in Figure 6 for the first case and Figure 7 for the second case.

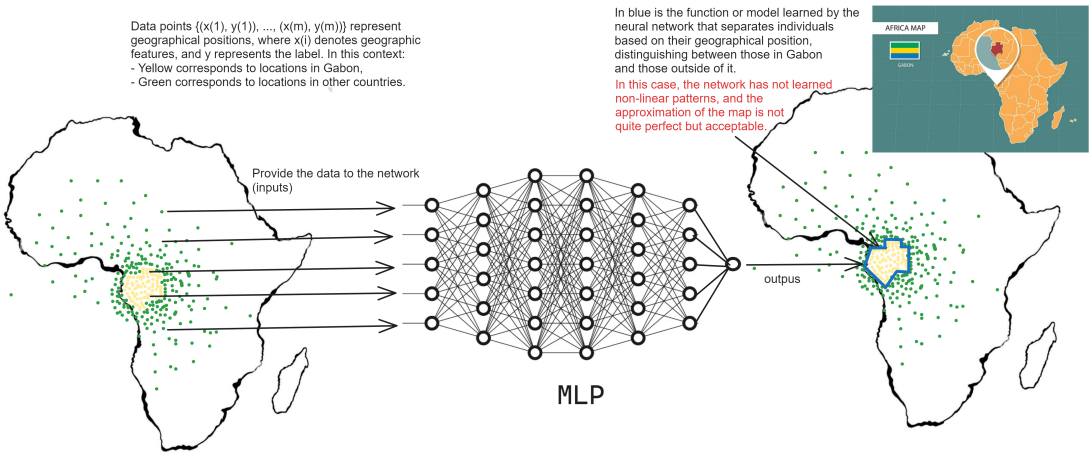


Figure 6: Network with identity activation function (no activation function).

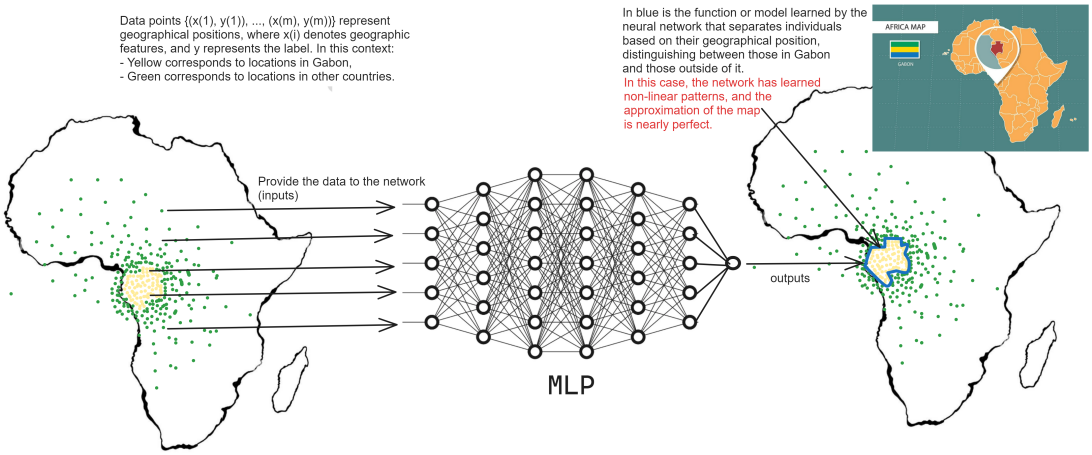


Figure 7: Network with activation function introducing non-linearity.

This is possible thanks to neural networks, which can approximate any function according to the Universal Approximation Theorem [8]***.***

1.5 Neural Network

A neural network is a stack of layers of the same or different types with the goal of transmitting learned information from layer to layer. The complexity of learned features increases with the number of layers. Generally, the more layers, the network is considered deep. Networks can be classified as deep or wide:

- A large or wide network has a single hidden layer but a large number of units.
- A network is considered deep if it has at least two hidden layers.

Large network vs. deep network: In general, a deep network is not necessarily superior to a large network in terms of function approximation. Performance depends on various factors such as the complexity of the function to be approximated, the quality of training data, and the network's ability to learn useful representations. However, it is relevant to note that deep networks often achieve comparable or even better performance with *fewer neurons* than wide networks, due to the efficiency of the hierarchical representations they can learn.

Below in Figure 8, we present the general topology of any type of neural network.

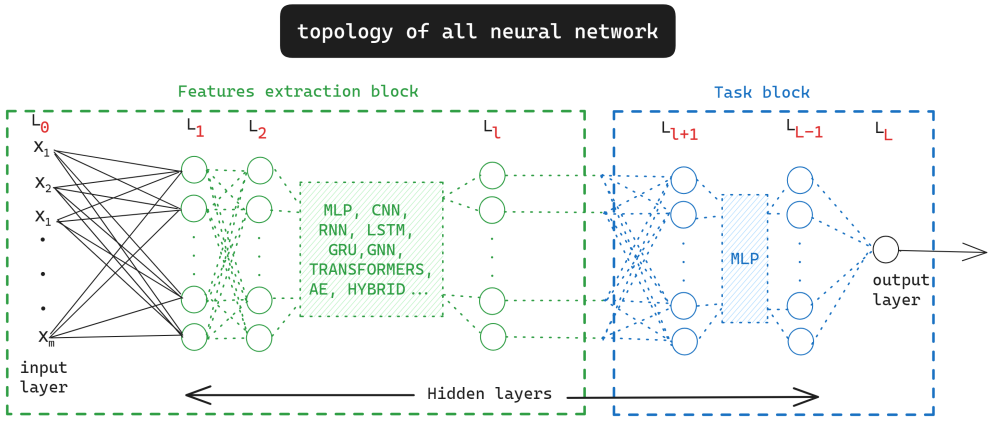


Figure 8: General topology of a neural network.

1.6 Deep Learning vs Machine Learning

The primary distinction between machine learning and deep learning lies in the feature extraction block (in green Figure 8). In machine learning, this block is replaced by humans, typically data engineers and domain experts, who collaboratively manually extract relevant features or information. These elements are then passed to the second block, called the task block (in blue Figure 8).

[8] https://en.wikipedia.org/wiki/Universal_approximation_theorem

The second difference is that, in machine learning, the task block is not a neural network (MLP), but rather machine learning algorithms such as SVM, Random Forest, decision trees, KNN, K-means, regression, and others.

The third difference concerns the nature and quantity of processed data. Machine learning algorithms ingest structured data in tabular form, and beyond a certain threshold, these algorithms become outdated, stagnate in terms of performance, and become very slow. In contrast, deep learning handles a variety of data, whether structured (such as tables) or unstructured (images, audio, text, graphs), and becomes more effective as the data quantity increases, as illustrated in the following Figure 9.

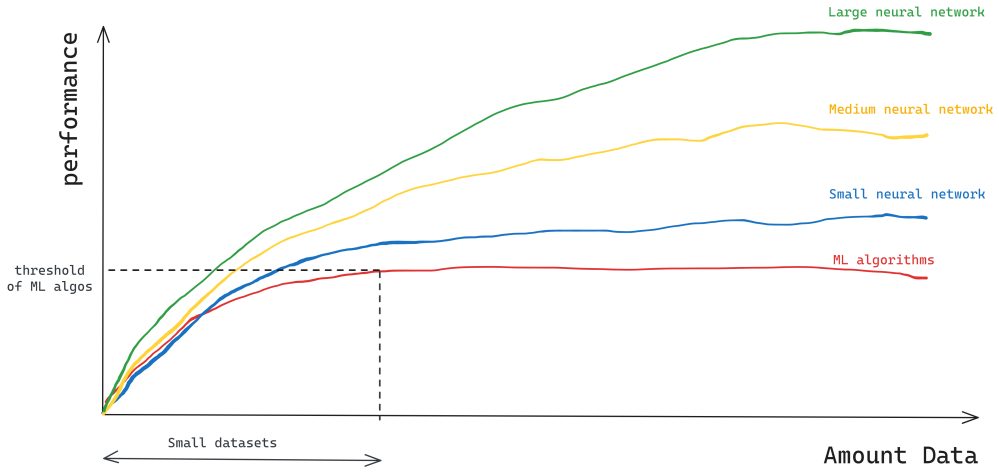


Figure 9: Performance: Neural networks vs. machine learning algorithms depending on the amount of data

In the size part of relatively small datasets the machine learning algorithms can perform better than neural networks

1.7 Training a Neural Network: Objective Function, Forward and Backward Propagation

Training a neural network revolves around four essential steps.

1. It is necessary to define the objective function, also known as the loss function, which measures the gap between the network's predictions and the true labels. Initializing the model parameters and choosing the appropriate optimizer are also crucial at this stage. This objective function is the one that will be optimized during training.

2. Forward propagation involves calculating activation values for each layer of the network using the current parameters. This step involves the successive application of linear transformations followed by activation functions for each layer.

3. The objective function calculation phase measures the gap between the network's predictions and the true labels using the loss function. Minimizing this function is the training objective, achieved by adjusting the network's parameters.

4. Backward propagation is a crucial step in training, where gradients with respect to the parameters are calculated. These gradients express the sensitivity of the loss function to each

parameter of the network. Gradient descent is then used to update the network's parameters in the direction that minimizes the loss function.

An illustration of this process is provided in figure 10.

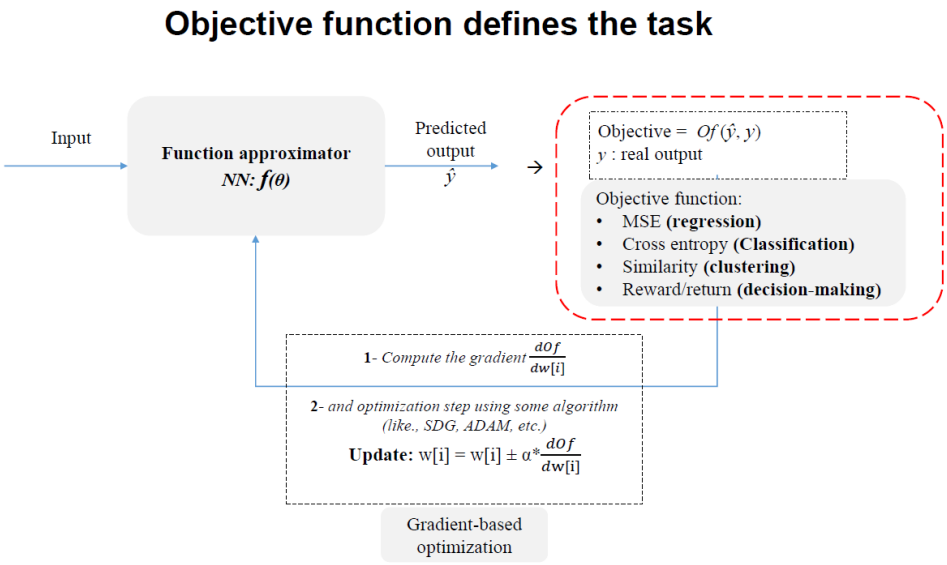


Figure 10: Training loop of a neural network

source: a slide from the neural networks course [11]

In section 2, during the implementation of both architectures, we will implement the various blocks contained in this image.

1.8 Gradient Descent

Gradient descent is a fundamental optimization algorithm used to minimize the loss function of a neural network. The main idea is to iteratively adjust the network parameters in the opposite direction of the gradient of the loss function. This allows the algorithm to converge towards a local minimum of the loss function, corresponding to a neural network model better suited to the training data.

The process of gradient descent can be summarized in a few key steps:

1. Compute the gradients of the loss function with respect to each network parameter.
2. Update the network parameters using the calculated gradients and a learning rate, which controls the size of the steps taken during the update.
3. Repeat these steps for a certain number of iterations until the loss function converges to a minimum.

Algorithm 1 Gradient Descent

Initialize θ, Initialize learning rate α
while Not converged **do**
    Compute predictions: $\hat{y} = f(X; \theta)$
    Compute cost: $J(\theta)$
    Compute gradient: $\nabla_\theta J(\theta)$
    Update parameters: $\theta \leftarrow \theta - \alpha \nabla_\theta J(\theta)$
end while

The choice of the learning rate is crucial, as a too high rate can lead to rapid convergence but with the risk of overshooting the minimum, while a too low rate can result in slow convergence.[9]

1.9 Model Parameters vs Hyperparameters

Model parameters encompass the weights and biases of different layers, adjusted during gradient descent in training. These parameters are optimized to minimize the loss function, and their optimal values are utilized during inference.

On the other hand, hyperparameters such as learning rate, number of iterations, batch size, number of neurons per layer, number of layers, etc., are defined before training and do not evolve during gradient descent. The judicious choice of hyperparameters is crucial to optimize the learning algorithm's performance.
In summary, any parameter updated during learning is a model parameter used during inference; otherwise, it is a hyperparameter.

2 Implementation

In this section, we will implement the architecture of our two neural networks and carry out the various steps of the model training loop, including forward propagation, cost calculation, backward propagation, and parameter updating. Although we will use several equations to perform these tasks, we will not provide their derivation in this article to background the mathematical aspect, as mentioned at the beginning of the article, and also to simplify the content. For those who wish to delve deeper and understand the mathematical foundations behind these equations, we highly recommend consulting the comprehensive derivations provided in Jonas Lalin's three articles: *Feedforward Neural Networks in Depth, Part 1: Forward and Backward Propagations* [10], *Feedforward Neural Networks in Depth, Part 2: Activation Functions* [11], and *Feedforward Neural Networks in Depth, Part 3: Cost Functions* [12], or taking the *Neural Networks and Deep Learning* course from Andrew NG's Deep Learning Specialization on Coursera [13].

Before diving into the implementation, we present a table of notations for easier reading of the code.

[9] https://www.geeksforgeeks.org/gradient-descent-algorithm-and-its-variants/?ref=header_search
[10] https://jonaslalin.com/2021/12/10/feedforward-neural-networks-part-1/
[11] https://jonaslalin.com/2021/12/21/feedforward-neural-networks-part-2/
[12] https://jonaslalin.com/2021/12/22/feedforward-neural-networks-part-3/
[13] https://www.coursera.org/learn/neural-networks-deep-learning/

2.1 Notations

Table 1: Abbreviations and Descriptions

Notation	Description	Dimension
$\cdot$	Matrix product	*operator*
$\odot$	Element-wise matrix product	*operator*
$\mathbf{X}$ or $\mathbf{A}^{[0]}$	Matrix of inputs or features	$\mathbf{n}_X \times \mathbf{m}$
$\mathbf{Y}$ or $\mathbf{A}^{[L]}$	Vector (matrix) of labels	$\mathbf{1} \times \mathbf{m}$
$\mathbf{0} \leq \mathbf{l} \leq \mathbf{L}$	Current layer number	*integer*
$\mathbf{L}$	Number of layers in the neural network or the output layer number	*integer*
$\mathbf{n}_X$ or $\mathbf{n}^{[0]}$	Size of inputs	*integer*
$\mathbf{n}_Y$ or $\mathbf{n}^{[L]}$	Size of outputs	*integer*
$\mathbf{n}^{[l]}$	Number of neurons or units in layer l	*integer*
$\mathbf{W}^{[l]}$	Parameter matrix or weights between layers l and $l-1$	$\mathbf{n}^{[l]} \times \mathbf{n}^{[l-1]}$
$\mathbf{B}^{[l]}$	Vector (matrix) of biases for layer l	$\mathbf{n}^{[l]}$
$\mathbf{Z}^{[l]}$	$\mathbf{Z}^{[l]} = \mathbf{W}^{[l]} \cdot \mathbf{A}^{[l-1]} + \mathbf{B}^{[l]}$	$\mathbf{n}^{[l]} \times \mathbf{m}$
$\mathbf{g}^{[l]}$	Activation function for layer l	*function*
$\mathbf{A}^{[l]}$	$\mathbf{A}^{[l]} = \mathbf{g}(\mathbf{Z}^{[l]})$, matrix of activations for layer l	$\mathbf{n}^{[l]} \times \mathbf{m}$

We illustrate the information propagation in a small neural network with Figure 11. This illustration aims to understand how to retrieve the weight and bias matrices as well as the various equations needed for forward and backward information transmission.

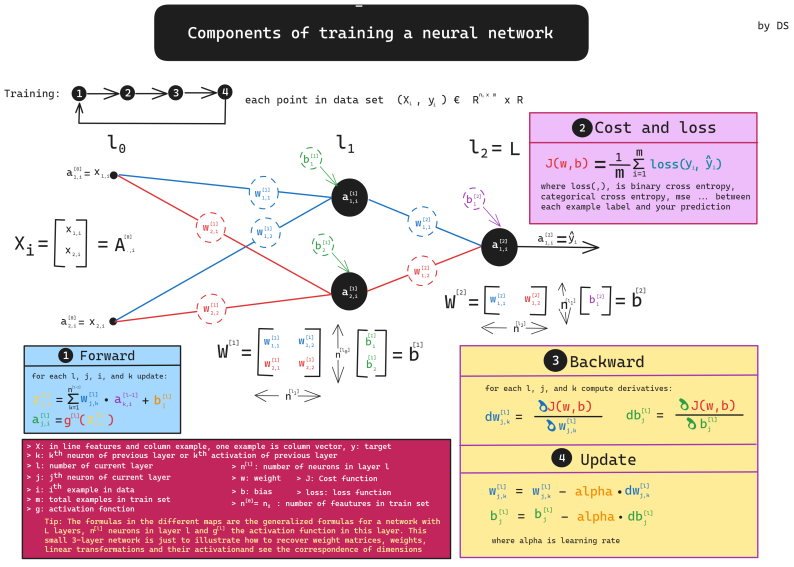


Figure 11: Components of training a neural network.

The equations presented in the cards of Figure 11 are used to perform element-wise calculations. For our implementation, we will use the following equations which represent the vectorized forms of these equations, designed to use the fewest 'for' loops to optimize calculations:

1. **Forward**
   $\forall l = 1, \ldots, L$,

$$\mathbf{Z}^{[l]} = \mathbf{W}^{[l]} \cdot \mathbf{A}^{[l-1]} + \mathbf{B}^{[l]} \tag{1}$$

$$\mathbf{A}^{[l]} = \mathbf{g}^{[l]}(\mathbf{Z}^{[l]}) \tag{2}$$

   For each layer, after computing $\mathbf{Z}^{[l]}$ and $\mathbf{A}^{[l]}$, we will cache them as they will be used during the backward propagation to compute the derivatives.

2. **Backward**
   $\forall l = L-1, \ldots, 1$,

$$\mathbf{dW}^{[l]} = \frac{1}{m}\mathbf{dZ}^{[l]} \cdot \mathbf{A}^{[l]^T} \tag{3}$$

$$\mathbf{dB}^{[l]} = \frac{1}{m} \sum_{\text{columns}} \mathbf{dZ}^{[l]} \tag{4}$$

$$\mathbf{dZ}^{[l-1]} = \mathbf{dW}^{[l]^T} \cdot \mathbf{dZ}^{[l]} \odot \mathbf{g}'^{[l]}(\mathbf{Z}^{[l-1]}) \tag{5}$$

   To compute the derivatives with respect to weights $\mathbf{dW}^{[l]}$ and bias $\mathbf{dB}^{[l]}$, we will need to compute the derivatives $\mathbf{dZ}^{[l]}$. However, we do not have them cached, so we need to compute them using Equation 5. Thus, we need to initialize the backward propagation by computing $\mathbf{dZ}^{[L]}$. Depending on the task we are performing, we compute it accordingly.

   - If we are in a classification task, our loss function is binary cross entropy or categorical cross entropy, and the activation function at the last layer is respectively sigmoid or softmax. Thus, applying the chain rule in two steps, we have:

$$\mathbf{dZ}^{[L]} = \frac{\partial J(W,B)}{\partial Z^{[L]}} = \frac{\partial J(W,B)}{\partial A^{[L]}} \cdot \frac{\partial A^{[L]}}{\partial Z^{[L]}} = \frac{1}{m}(A^{[L]} - Y) \tag{6}$$

   - If we are in a regression task, the loss function is mse, and very often no activation function is used. In this case, we have:

$$\mathbf{dZ}^{[L]} = \frac{\partial J(W,B)}{\partial Z^{[L]}} = \frac{2}{m}(Z^{[L]} - Y) \tag{7}$$

   - We apply the same reasoning for other tasks.

   After computing the derivatives (gradients), we cache them for the update step.

3. **Update**
   $\forall l = 1, \ldots, L$ and α,

$$\mathbf{W}^{[l]} = \mathbf{W}^{[l]} - \alpha \cdot \mathbf{dW}^{[l]} \tag{8}$$

$$\mathbf{B}^{[l]} = \mathbf{B}^{[l]} - \alpha \cdot \mathbf{dB}^{[l]} \tag{9}$$

With all necessary elements in hand, let's move on to implementing our neural networks.

2.2 TASK 1: MNIST (CIFAR 10) handwritten digits classification dataset

Description

Figure 12: mnist: CIFAR 10.

source: https://www.researchgate.net/figure/Some-training-examples-from-a-MNIST-and-b-CIFAR-10-datasets_fig1_300331950

This is a dataset of 60,000 28x28 grayscale images of the 10 digits, along with a test set of 10,000 images. More info can be found at the CIFAR-10 link on Keras. We will follow the workflow described in the following figure **??**.

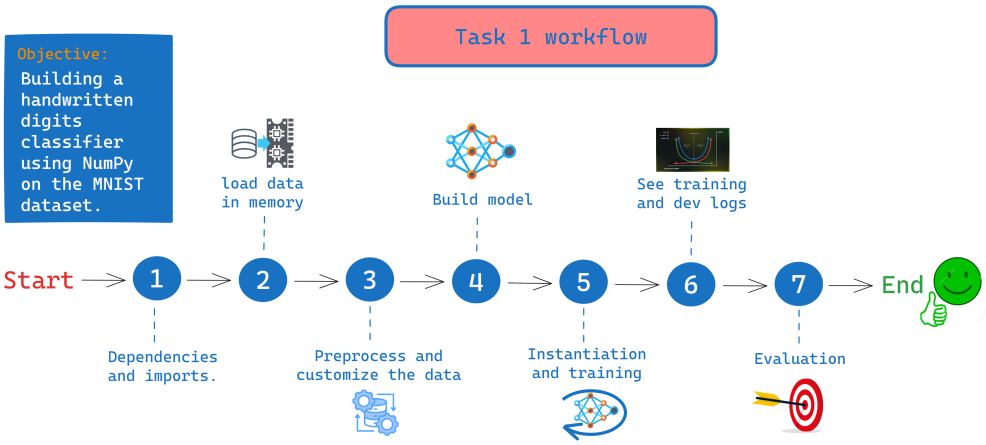


Figure 13: Workflow.

2.2.1 Dependencies and imports.

2.2.1.1 Dependencies

uncomment the cell below and run it to install the project dependencies

[1]:
```
#!pip install numpy pandas seaborn pillow matplotlib scikit-learn keras
```

2.2.1.2 Libraries

[2]:
```
import numpy as np
import pandas as pd
import seaborn as sns
import PIL.Image as Image
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
import keras.datasets.mnist as mnist # use only to retrieve data
```

2.2.2 Retrieve data

We retrieve data from keras hub

[3]:
```
(train_X, train_y), (test_X, test_y) = mnist.load_data()
(train_X.shape, train_y.shape), (test_X.shape, test_y.shape) # check␣
↪dimensions
```

[3]:
```
(((60000, 28, 28), (60000,)), ((10000, 28, 28), (10000,)))
```

2.2.3 Preprocess and customize data dataset

[4]:
```
class PreprocessingData:
    """
        Class for preprocessing data and generating batch iterations.
    """

    def __init__(self, X, y, batch_size, transf=None):
        """
            Initializes the PreprocessingData class with input data, labels,␣
↪and batch size.
        """
        X, y = X.reshape(-1, 28*28).T, y.reshape(-1, 1).T
        # suffle data
        permutation = list(np.random.permutation(X.shape[1]))
        self.X = X[:, permutation]
        self.y = y[:, permutation].reshape(-1, X.shape[1])
        self.batch_size = min(batch_size, X.shape[1])  # if batch_size >␣
↪length
        self.transf = transf
        self.length = X.shape[1]

    def __getitem__(self, index):
        """
            Returns a specific batch based on the index.
```

```
        """
        X_batch = self.X[:, index]
        y_batch = self.y[:, index]
        batch = X_batch, y_batch

        if self.transf:
            batch_transf = self.transf(batch)
            return batch_transf
        return batch

    def __len__(self):
        """
            Returns the total length of the data.
        """
        return self.length

    def batch_iterator(self):
        """
            Generates data batches by iterating over the entire dataset.
        """
        if self.batch_size >= self.length:
            # Handle case where batch_size is greater than total data size
            batch = self.X, self.y
            if self.transf:
                batch_transf = self.transf(batch)
                yield batch_transf
            else:
                yield batch
        else:
            nbr_of_batches = self.length // self.batch_size

            for i in range(nbr_of_batches):
                X_batch = self.X[:, i * self.batch_size: (i + 1) * self.
↪batch_size]
                y_batch = self.y[:, i * self.batch_size: (i + 1) * self.
↪batch_size]
                batch = X_batch, y_batch

                if self.transf:
                    batch_transf = self.transf(batch)
                    yield batch_transf
                else:
                    yield batch

            if self.length % self.batch_size != 0:
                X_last_batch = self.X[:, nbr_of_batches * self.batch_size:]
                y_last_batch = self.y[:, nbr_of_batches * self.batch_size:]
```

```
                last_batch = X_last_batch, y_last_batch

                if self.transf:
                    last_batch_transf = self.transf(last_batch)
                    yield last_batch_transf
                else:
                    yield last_batch


###################### Transformation class ######################
class Transformation:
    """
        Class to define transformations to apply on batches.
    """

    def __init__(self, X_max, nb_classes):
        """
            Initializes the Transformation class with normalization and label␣
 ↪conversion parameters.
        """
        self.X_max = X_max
        self.nb_classes = nb_classes

    def norm_X(self, X):
        """
            Normalizes the input data.
        """
        return X / self.X_max

    def y_to_one_hot(self, y):
        """
            Converts label vector to matrix.
        """
        y = np.reshape(y, (-1, 1))
        Y = np.zeros((self.nb_classes, len(y)))

        for j, label in enumerate(y):
            Y[label, j] = 1

        return Y

    def __call__(self, batch):
        """
            Applies transformations on a given batch.
        """
        X, y = batch[0], batch[1]
        return self.norm_X(X), self.y_to_one_hot(y)
```

2.2.4 Build model

2.2.4.1 Activation functions

[5]:
```
###################### activation functions ######################

def ReLU(z):
    return np.maximum(0, z)

def dReLU(z):
    return np.where(z > 0, 1, 0)

def LeakyReLU(z, alpha=0.01):
    return np.where(z > 0, z, alpha * z)

def dLeakyReLU(z, alpha=0.01):
    return np.where(z > 0, 1, alpha)

def Tanh(z):
    return np.tanh(z)

def dTanh(z):
    return 1 - np.tanh(z)**2

def Sigmoid(z):
    return 1 / (1 + np.exp(-z))

def dSigmoid(z):
    s = Sigmoid(z)
    return s * (1 - s)

def Softmax(z):
    exp_z = np.exp(z - np.max(z, axis=0, keepdims=True))
    return exp_z / np.sum(exp_z, axis=0, keepdims=True)

def dSoftmax(z):
    s = Softmax(z)
    return s * (1 - s)

# dictionary of activation functions
dict_activation = {
    "relu": (ReLU, dReLU),
    "leakyrelu": (LeakyReLU, dLeakyReLU),
    "tanh": (Tanh, dTanh),
    "sigmoid": (Sigmoid, dSigmoid),
    "Softmax": (Softmax, dSoftmax)
}
```

```
###################### loss functions ######################
def binary_cross_entropy_loss(y, y_hat):
    m = y.shape[1]
    loss = -np.sum(y * np.log(y_hat) + (1 - y) * np.log(1 - y_hat)) / m
    return loss

def categorical_cross_entropy_loss(y, y_hat):
    m = y.shape[1]
    epsilon = 1e-10  # numrerical stability
    loss = -np.sum(np.sum(np.multiply(y, np.log(y_hat + epsilon)), axis=0)) / m
    return loss

def mse_loss(y, y_hat):
    return np.mean(np.square(y_hat-y))
```

2.2.4.2 Optimizers

[6]:
```
class Optimizer:
    def __init__(self, learning_rate=0.01, beta1=0.9, beta2=0.999,␣
 ↪epsilon=1e-8):
        self.learning_rate = learning_rate
        self.beta1 = beta1
        self.beta2 = beta2
        self.epsilon = epsilon
        self.momentums = {}  # Store momentums for each parameter
        self.rmsprops = {}  # Store squared gradients for each parameter
        self.t = 0  # Time step counter for Adam

    def update_parameters_SGD(self, model, learning_rate=None):
        learning_rate = learning_rate or self.learning_rate
        for key in model.parameters.keys():
            model.parameters[key] -= learning_rate * model.gradients["d" + key]

    def update_parameters_SGD_with_momentum(self, model, learning_rate=None,␣
 ↪momentum=None):
        learning_rate = learning_rate or self.learning_rate
        momentum = momentum or self.beta1

        for key in model.parameters.keys():
            if key not in self.momentums:
                self.momentums[key] = np.zeros_like(model.parameters[key])

            self.momentums[key] = momentum * self.momentums[key] + (1 -␣
 ↪momentum) * model.gradients["d" + key]
```

```
            model.parameters[key] -= learning_rate * self.momentums[key]

    def update_parameters_RMSProp(self, model, learning_rate=None, beta2=None,␣
↪epsilon=None):
        learning_rate = learning_rate or self.learning_rate
        beta2 = beta2 or self.beta2
        epsilon = epsilon or self.epsilon

        for key in model.parameters.keys():
            if key not in self.rmsprops:
                self.rmsprops[key] = np.zeros_like(model.parameters[key])

            self.rmsprops[key] = beta2 * self.rmsprops[key] + (1 - beta2) * np.
↪square(model.gradients["d" + key])
            model.parameters[key] -= learning_rate * (model.gradients["d" +␣
↪key] / np.sqrt(self.rmsprops[key] + epsilon))

    def update_parameters_Adam(self, model, learning_rate=None, beta1=None,␣
↪beta2=None, epsilon=None):
        self.t += 1
        learning_rate = learning_rate or self.learning_rate
        for key in model.parameters.keys():
            if key not in self.momentums:
                self.momentums[key] = np.zeros_like(model.parameters[key])
            if key not in self.rmsprops:
                self.rmsprops[key] = np.zeros_like(model.parameters[key])

            self.momentums[key] = self.beta1 * self.momentums[key] + (1 - self.
↪beta1) * model.gradients["d" + key]
            self.rmsprops[key] = self.beta2 * self.rmsprops[key] + (1 - self.
↪beta2) * np.square(model.gradients["d" + key])

            mnt_corrected = self.momentums[key] / (1 - self.beta1 ** self.t)
            rms_corrected = self.rmsprops[key] / (1 - self.beta2 ** self.t)
            model.parameters[key] -= learning_rate * mnt_corrected / (np.
↪sqrt(rms_corrected) + self.epsilon)
```

2.2.4.3 Neural network for a classsification task

[7]:
```
class NNClassifier:
    def __init__(self, layers_dims, weight_init="He", bias=True,␣
↪activation_function=None):
        """
            Initializes the neural network with the specified parameters.
        """
```

```
        self.bias = bias
        self.total = 0
        self.memory = {}
        self.gradients = {}
        self.parameters = {}
        self.layers_dims = layers_dims
        self.number_of_layers = len(self.layers_dims)
        self.weight_init = weight_init
        self.initialize_weights() # initialization of parameters with␣
↪"weight_init" method
        self.activation_function, self.activation_derivative = self.
↪get_activation_functions(activation_function)



    def initialize_weights(self):
        """
        Initializes the weights and biases of the neural network.Three cases␣
↪He, Xavier or Randomly with scale
        """
        for l in range(1, self.number_of_layers):
            previous_layer_dim, current_layer_dim = self.layers_dims[l-1],␣
↪self.layers_dims[l]#layer_dim

            if self.weight_init.lower() == "he":
                scale = np.sqrt(2 / previous_layer_dim)
            elif self.weight_init.lower() == "xavier":
                scale = np.sqrt(1 / previous_layer_dim)
            else:
                scale = .001

            self.parameters["W" + str(l)] = np.random.randn(current_layer_dim,␣
↪previous_layer_dim) * scale
            if self.bias:
                self.parameters["B" + str(l)] = np.zeros((current_layer_dim,␣
↪1))




    def get_activation_functions(self, activation):
        """
        get the activation function and its derivative based on the provided␣
↪activation name.
        """
        activation = activation.lower() if activation else "relu"
```

```
        if activation not in dict_activation:
            raise ValueError(f"Invalid activation function: {activation}.␣
↪Supported values: {list(dict_activation.keys())}")

        return dict_activation[activation]



    def forward(self, x):
        """
            Performs the forward pass through the neural network.
            m is number of example
        """
        self.memory["A" + str(0)] = A_previous = x
        # Hidden(s) layer(s)
        for l in range(1, self.number_of_layers - 1):
            W = self.parameters["W" + str(l)]
            if self.bias:
                B = self.parameters["B" + str(l)]
                Z = np.dot(W, A_previous) + B
            else:
                Z = np.dot(W, A_previous)
            self.memory["Z" + str(l)] = Z
            A_current = self.activation_function(Z)
            self.memory["A" + str(l)] = A_current
            A_previous = A_current

        # Output layer
        W = self.parameters["W" + str(self.number_of_layers - 1)]
        if self.bias:
            B = self.parameters["B" + str(self.number_of_layers - 1)]
            Z = np.dot(W, A_previous) + B
        else:
            Z = np.dot(W, A_previous)
        self.memory["Z" + str(self.number_of_layers - 1)] = Z
        x = Z
        return x



    def backward(self, X, Y, Y_proba):
        """
            Performs the backward pass (backpropagation) through the neural␣
↪network to compute gradients.
        """
        m = X.shape[1]
```

```
        A = Y_proba
        dZ = 1/m * (A - Y)
        # For output layer
        self.memory["dZ" + str(self.number_of_layers - 1)] = dZ
        A_previous = self.memory["A" + str(self.number_of_layers - 2)]
        self.gradients["dW" + str(self.number_of_layers - 1)] = np.dot(dZ,␣
↪A_previous.T) / m
        if self.bias:
            self.gradients["dB" + str(self.number_of_layers - 1)] = np.sum(dZ,␣
↪axis=1, keepdims=True) / m

        # For Hidden(s) layer(s)
        for l in reversed(range(1, self.number_of_layers - 1)):
            W_next = self.parameters["W" + str(l + 1)]
            dZ_next = self.memory["dZ" + str(l + 1)]
            Z_current = self.memory["Z" + str(l)]
            A_previous = self.memory["A" + str(l - 1)]
            dZ_current = np.multiply(np.dot(W_next.T, dZ_next), np.
↪vectorize(self.activation_derivative)(Z_current))
            self.gradients["dW" + str(l)] = np.dot(dZ_current, A_previous.T) /␣
↪m
            if self.bias:
                self.gradients["dB" + str(l)] = np.sum(dZ_current, axis=1,␣
↪keepdims=True) / m
            self.memory["dZ" + str(l)] = dZ_current

        return self.gradients



    def total_parameters(self):
        """
        Calculates the total number of parameters in the neural network.
        """
        self.total = 0
        for i in range(1, self.number_of_layers):
            if self.bias:
                self.total += self.layers_dims[i] * (self.layers_dims[i - 1] +␣
↪1)
            else:
                self.total += self.layers_dims[i] * (self.layers_dims[i - 1])
        return self.total


    def predict(self, x):
        """
```

```
        Makes class predictions using the trained neural network.
        """
        return np.argmax(self.proba(x), axis=0)


    def proba(self, x):
        """
        Computes the class probabilities using the trained neural network.
        """
        A = x
        # Hidden layer(s)
        for l in range(1, self.number_of_layers - 1):
            W = self.parameters["W" + str(l)]
            if self.bias:
                B = self.parameters["B" + str(l)]
                x = self.activation_function(np.dot(W, x) + B)
            else:
                 x = self.activation_function(np.dot(W, x))

        # Output layer
        W = self.parameters["W" + str(self.number_of_layers - 1)]
        if self.bias:
            B = self.parameters["B" + str(self.number_of_layers - 1)]
            out = np.dot(W, x) + B
        else:
            out = np.dot(W, x)

        return Softmax(out)



    def check_gradient(self):
        """
        Placeholder for a method that could be used to check gradients.
        """
        pass

    def __call__(self, x):
        """
        Makes the object callable, allowing for convenient use like a function.
        """
        return self.forward(x)

    def __repr__(self):
        """
        Returns a string representation of the neural network object.
        """
```

```
        layer_info = ""
        for i in range(1, self.number_of_layers - 1):
            layer_info += f"\n\tLayer {i}: Linear(in_dim={self.
↪layers_dims[i-1]}, out_dim={self.layers_dims[i]}, bias={self.bias})\
            \n\tActivation function: {self.activation_function.__name__}()"

        return f"NN model with {self.number_of_layers} layers of which {self.
↪number_of_layers - 2} hidden layer\
       \n\n\t(Input layer): (Layer 0 : (dim={self.layers_dims[0]})\
       \n\t(Hidden layers): ({layer_info})\
       \n\t(Output layer): (Layer {self.number_of_layers - 1}␣
↪Linear(in_dim={self.layers_dims[-2]}, out_dim={self.layers_dims[-1]},␣
↪bias={self.bias})"

    def __str__(self):
        """
        Returns a string representation of the neural network object.
        """
        return self.__repr__()
```

2.2.5 Instantiation and training model

2.2.5.1 Hyperparameters and instanciations

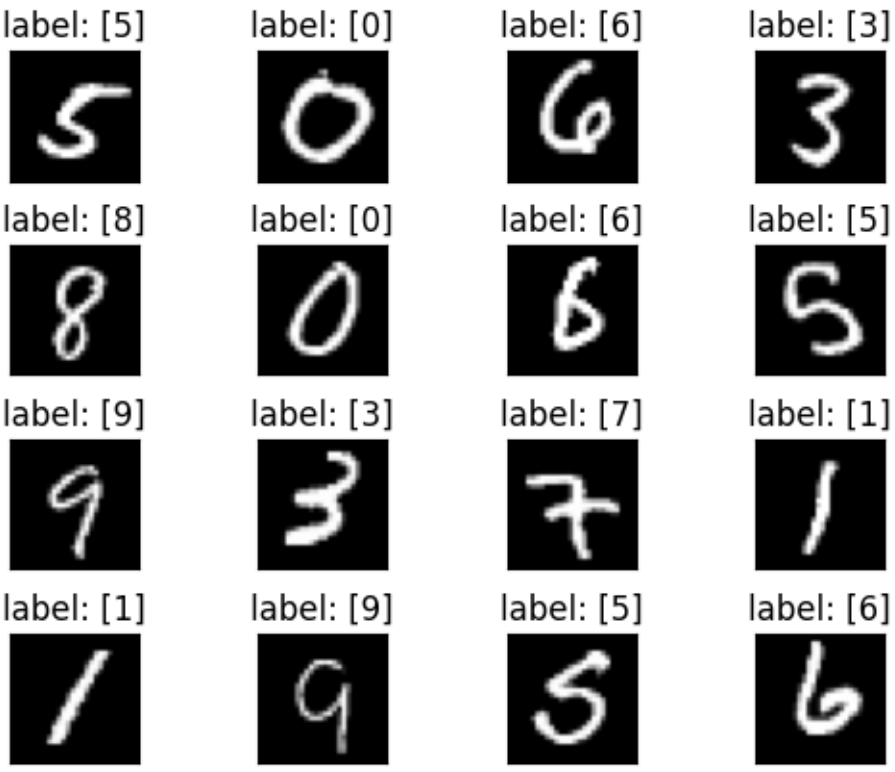
[8]:
```
# hyperparameters
lr = .4
batch_size = 8
nb_iter  =  15
nb_classes = 10
input_dim = 28*28
layers_dims = [input_dim, 64, nb_classes]

# to have the same initial values at runtime
np.random.seed(3)

# data
transf  = Transformation(255., nb_classes)
data_train = PreprocessingData(train_X, train_y, transf=transf, batch_size =␣
 ↪batch_size)
data_dev = PreprocessingData(test_X, test_y, transf=transf, batch_size = 10000)

# criterion  and optimizer
criterion = categorical_cross_entropy_loss
optim = Optimizer()
```

```
# model
model = NNClassifier(layers_dims, weight_init="He", activation_function="relu")
print(f"{model} \n\nNumber of parameters:  {model.total_parameters()/1e6}␣
 ↪Million(s)")

# training and dev logs
train_stuff = {"train_loss": [],
               "train_acc": []
              }
dev_stuff = {"dev_loss": [],
             "dev_acc": []
             }
```

```
NN model with 3 layers of which 1 hidden layer

        (Input layer): (Layer 0 : (dim=784)
        (Hidden layers): (
        Layer 1: Linear(in_dim=784, out_dim=64, bias=True)
        Activation function: ReLU())
        (Output layer): (Layer 2 Linear(in_dim=64, out_dim=10, bias=True)

Number of parameters:  0.05089 Million(s)
```

2.2.5.2 Visualize data

[9]:
```
for i in range(16):
    image, label = data_train[i][0].reshape(28, 28), np.
 ↪argmax(data_train[i][1], axis=0)
    plt.subplot(4, 4, i + 1)
    plt.imshow(image, cmap=plt.get_cmap('gray'))
    plt.title("label: " + str(label))

# vertical spacing
plt.subplots_adjust(hspace=0.5)

# delete axes graduations
for ax in plt.gcf().get_axes():
    ax.set_xticks([])
    ax.set_yticks([])

plt.show()
```



2.2.5.3 Training loop

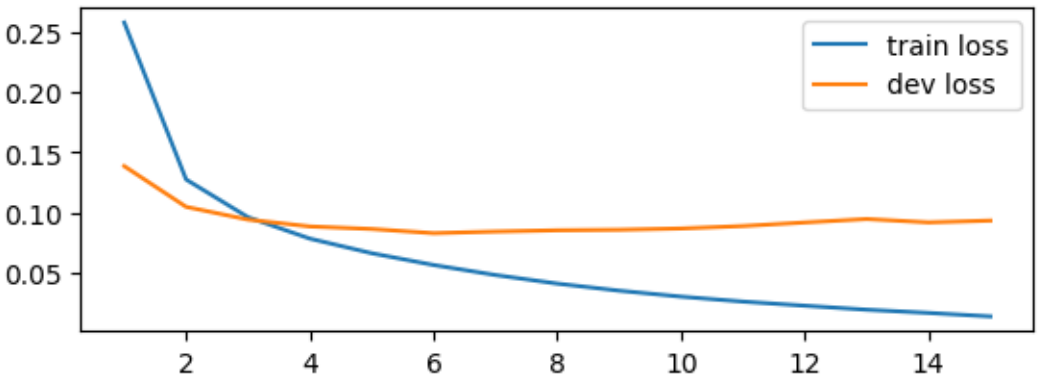
[10]:
```
for i in range(nb_iter):
    # Training stuff
    loss_iter = []
    y_hat_iter = []
    y_iter = []

    for X_batch, y_batch in data_train.batch_iterator():
        # step 1: Forward
        Z = model(X_batch)
        y_proba = Softmax(Z)
        # step 2: Compute loss
        loss = criterion(y_batch, y_proba)
        # step 3: Backward
        grad = model.backward(X_batch, y_batch, y_proba)
        # step 4: update parameters
        optim.update_parameters_SGD(model, learning_rate=lr)
        # Compute accuracy
        y_hat_iter.extend(np.argmax(y_proba, axis=0))
        y_iter.extend(np.argmax(y_batch, axis=0))
        # store loss batch
```

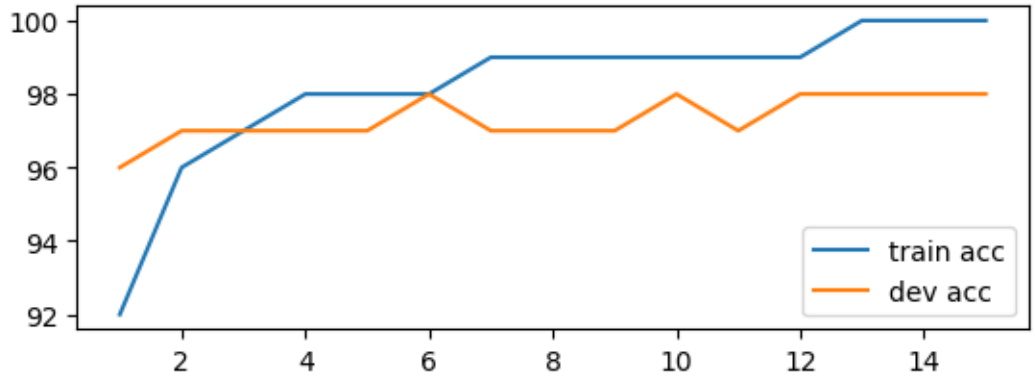
```
        loss_iter.append(loss)


    # Calculate and store training loss and accuracy epoch
    train_loss_mean = np.mean(loss_iter)
    train_accuracy = int(round(np.sum(np.array(y_iter) == np.
↪array(y_hat_iter)) / len(y_iter), 2) * 100)
    train_stuff["train_loss"].append(train_loss_mean)
    train_stuff["train_acc"].append(train_accuracy)

    # Validation mode: use predict or proba method to not update parameters
    X_dev, y_dev = data_dev[:]
    y_proba_dev = model.proba(X_dev)

    # Compute validation loss and accuracy
    dev_loss = criterion(y_dev, y_proba_dev)
    dev_accuracy = int(round(np.sum(np.argmax(y_dev, axis=0) == np.
↪argmax(y_proba_dev, axis=0)) / y_dev.shape[1] , 2) * 100)
    dev_stuff["dev_loss"].append(dev_loss)
    dev_stuff["dev_acc"].append(dev_accuracy)

    # Print progress
    print("Iter: [{}/{}]  Train loss: {:.3f} Dev loss: {:.3f} | Train acc: {}%␣
↪ Dev acc: {}%".format(
        i+1, nb_iter, train_loss_mean, dev_loss, train_accuracy, dev_accuracy))
```

```
Iter: [1/15]  Train loss: 0.258 Dev loss: 0.139 | Train acc: 92%  Dev acc: 96%
Iter: [2/15]  Train loss: 0.127 Dev loss: 0.105 | Train acc: 96%  Dev acc: 97%
Iter: [3/15]  Train loss: 0.096 Dev loss: 0.094 | Train acc: 97%  Dev acc: 97%
Iter: [4/15]  Train loss: 0.078 Dev loss: 0.088 | Train acc: 98%  Dev acc: 97%
Iter: [5/15]  Train loss: 0.066 Dev loss: 0.086 | Train acc: 98%  Dev acc: 97%
Iter: [6/15]  Train loss: 0.056 Dev loss: 0.083 | Train acc: 98%  Dev acc: 98%
Iter: [7/15]  Train loss: 0.048 Dev loss: 0.084 | Train acc: 99%  Dev acc: 97%
Iter: [8/15]  Train loss: 0.040 Dev loss: 0.085 | Train acc: 99%  Dev acc: 97%
Iter: [9/15]  Train loss: 0.035 Dev loss: 0.085 | Train acc: 99%  Dev acc: 97%
Iter: [10/15]  Train loss: 0.030 Dev loss: 0.086 | Train acc: 99%  Dev acc: 98%
Iter: [11/15]  Train loss: 0.025 Dev loss: 0.089 | Train acc: 99%  Dev acc: 97%
Iter: [12/15]  Train loss: 0.022 Dev loss: 0.091 | Train acc: 99%  Dev acc: 98%
Iter: [13/15]  Train loss: 0.019 Dev loss: 0.094 | Train acc: 100%  Dev acc: 98%
Iter: [14/15]  Train loss: 0.016 Dev loss: 0.092 | Train acc: 100%  Dev acc: 98%
Iter: [15/15]  Train loss: 0.013 Dev loss: 0.093 | Train acc: 100%  Dev acc: 98%
```

2.2.6 Visualize training and dev logs

[11]:
```
plt.subplot(211)
plt.plot(range(1,nb_iter+1), train_stuff["train_loss"], label="train loss")
plt.plot(range(1,nb_iter+1), dev_stuff["dev_loss"], label="dev loss")
plt.legend()
plt.show()
plt.subplot(212)
plt.plot(range(1,nb_iter+1), train_stuff["train_acc"], label="train acc")
plt.plot(range(1,nb_iter+1), dev_stuff["dev_acc"], label="dev acc")
plt.legend()
plt.show()
```





2.2.7 Evaluation

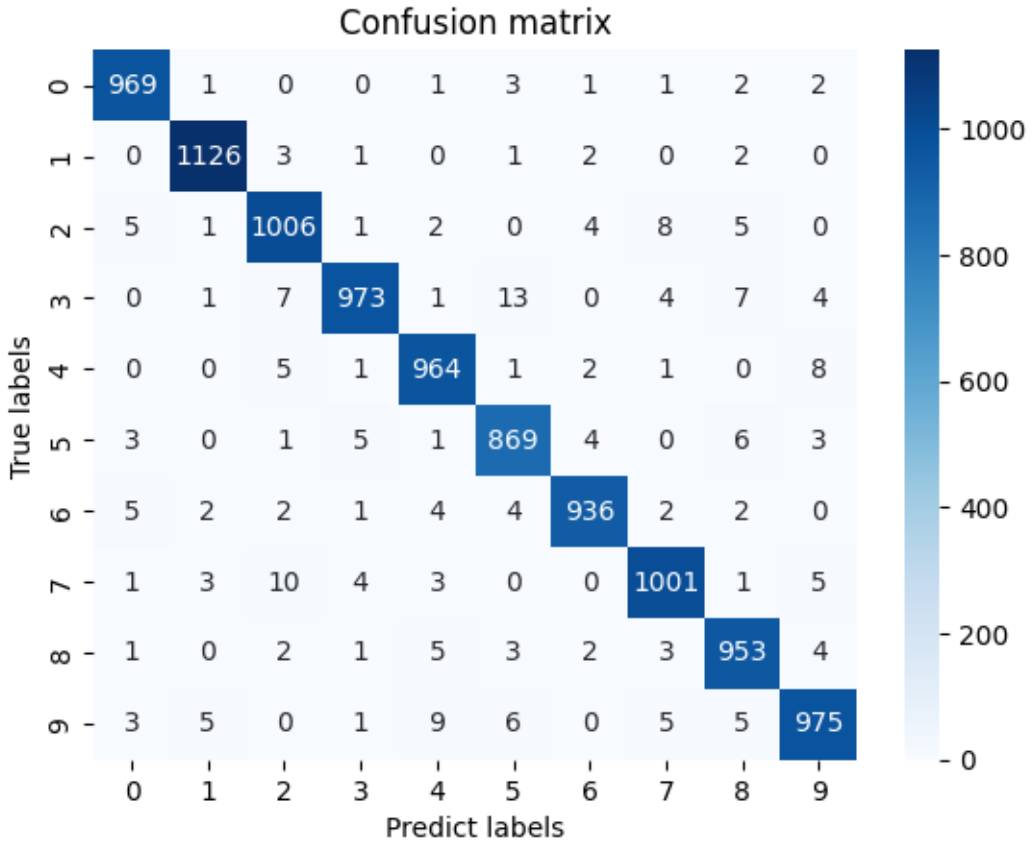
2.2.7.1 Accuracy

[12]:
```
X_dev, y_dev = data_dev[:]
true_labels  = np.argmax(y_dev, axis=0).reshape(-1)
labels_pred  = model.predict(X_dev).reshape(-1)
total_error = (true_labels!=labels_pred).sum()
print(f"\nTotal error: {total_error}/{len(true_labels)}\t ||\tAccuracy: {1 -␣
 ↪(total_error/len(true_labels)):.3f}%\n")
```

```
Total error: 228/10000    ||      Accuracy: 0.977%
```

2.2.7.2 Plotting results

[13]:
```
cfm = confusion_matrix(true_labels, labels_pred)
sns.heatmap(cfm, annot=True, fmt='g', cmap='Blues')
plt.title('Confusion matrix')
plt.xlabel('Predict labels')
plt.ylabel('True labels')
plt.show()
print(f"\nTotal error: {total_error}/{len(true_labels)}\n")
```



```
Total error: 228/10000
```

2.2.7.3 Plotting some prediction

[14]:
```
for i in range(64):
    j = np.random.randint(0, 10000) # for choose image randomly
    # choose and resize for plotting
    image, label = data_dev[j][0].reshape(28, 28), np.argmax(data_dev[j][1],␣
 ↪axis=0)
    pred_label  = model.predict(image.reshape(-1,1)).reshape(-1)
    # plotting
    plt.subplot(8, 8, i + 1)
    plt.imshow(image, cmap=plt.get_cmap('gray'))
    color = "green" if pred_label == label else "red"
    title = "l: " + str(label) + " p: " + str(pred_label)
    plt.title(title, color=color, fontsize=6)

# vertical spacing
plt.subplots_adjust(hspace=0.8)
plt.subplots_adjust(wspace=0.5)
# delete axes graduations
for ax in plt.gcf().get_axes():
    ax.set_xticks([])
    ax.set_yticks([])

plt.show()
```


l: [7] p: [7]
l: [1] p: [1]
l: [3] p: [3]
l: [2] p: [2]
l: [4] p: [4]
l: [2] p: [2]
l: [2] p: [2]
l: [9] p: [9]
l: [0] p: [0]
l: [4] p: [4]
l: [4] p: [4]
l: [7] p: [7]
l: [3] p: [1]
l: [1] p: [1]
l: [6] p: [6]
l: [1] p: [1]
l: [4] p: [4]
l: [5] p: [5]
l: [7] p: [7]
l: [8] p: [8]
l: [1] p: [1]
l: [1] p: [1]
l: [2] p: [2]
l: [4] p: [4]
l: [3] p: [3]
l: [3] p: [3]
l: [9] p: [9]
l: [2] p: [2]
l: [5] p: [5]
l: [5] p: [5]
l: [4] p: [9]
l: [8] p: [8]
l: [9] p: [9]
l: [1] p: [1]
l: [5] p: [5]
l: [1] p: [1]
l: [8] p: [8]
l: [5] p: [5]
l: [1] p: [1]
l: [2] p: [2]
l: [9] p: [9]
l: [6] p: [2]
l: [3] p: [3]
l: [2] p: [2]
l: [3] p: [3]
l: [5] p: [5]
l: [2] p: [2]
l: [4] p: [4]
l: [1] p: [1]
l: [3] p: [3]
l: [1] p: [1]
l: [1] p: [1]
l: [6] p: [6]
l: [8] p: [6]
l: [9] p: [9]
l: [9] p: [9]
l: [0] p: [0]
l: [7] p: [7]
l: [2] p: [2]
l: [7] p: [7]
l: [9] p: [9]
l: [1] p: [1]
l: [1] p: [1]
l: [0] p: [0]

2.3 TASK 2: Paris Housing Price Prediction

Description

Figure 14: Paris.

source: kaggle, dataset-cover

This dataset is an imaginary dataset on real estate prices in Paris, used for educational purposes. More information can be found at the dataset link on Kaggle. We will follow the workflow described in the following figure **??**.

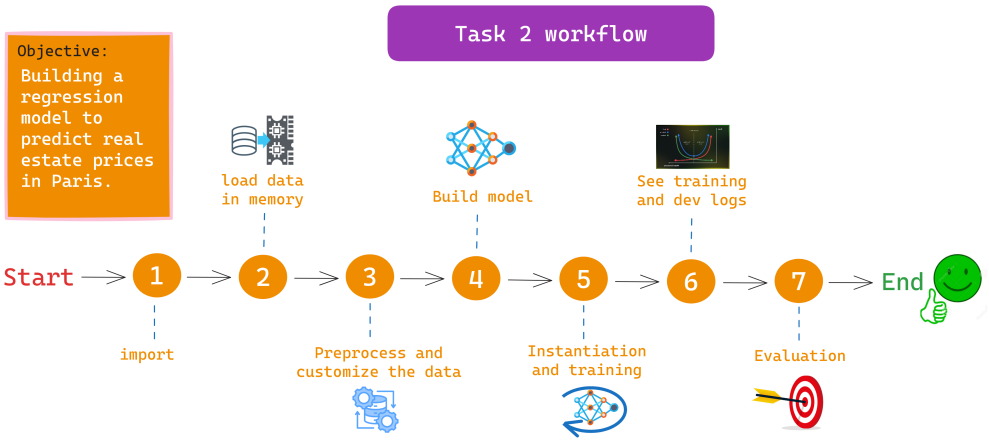


Figure 15: Workflow.

2.3.1 Import libraries

[1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

2.3.2 Load data and print data

[2]:
```
df = pd.read_csv("ParisHousing.csv")
df.head(3)
```

[2]:

	squareMeters	numberOfRooms	hasYard	hasPool	floors	cityCode	\
0	75523	3	0	1	63	9373	
1	80771	39	1	1	98	39381	
2	55712	58	0	1	19	34457	

	cityPartRange	numPrevOwners	made	isNewBuilt	hasStormProtector	\
0	3	8	2005	0	1	
1	8	6	2015	1	0	
2	6	8	2021	0	0	

	basement	attic	garage	hasStorageRoom	hasGuestRoom	price
0	4313	9005	956	0	7	7559081.5
1	3653	2436	128	1	2	8085989.5
2	2937	8852	135	1	9	5574642.1

2.3.3 Preprocess and customize dataset

2.3.3.1 Missing values

[3]:
```
df.isna().sum().sum()
```

[3]: 0

2.3.3.2 Split and standardize data

[4]:
```
np.random.seed(1)
# split
df_train = df.sample(frac=.80)
df_dev   = df.drop(index=df_train.index)
print("Train examples: ", len(df_train), "\tDev examples: ", len(df_dev))
```

```
Train examples:  8000   Dev examples:  2000
```

[5]:
```
X_train, y_train  = df_train.iloc[:, :-1].values.T, df_train.iloc[:, -1].
 ↪values.reshape(1, -1)
X_dev, y_dev      = df_dev.iloc[:, :-1].values.T, df_dev.iloc[:, -1].values.
 ↪reshape(1, -1)

# mean and standard deviation
```

```
mu, sigma =  np.mean(X_train, axis=1, keepdims=True),  np.std(X_train, axis=1,␣
 ↪keepdims=True)
# standardization
X_train =  (X_train - mu) / sigma
X_dev   = (X_dev - mu) / sigma


# dimensionality in trainning
print("\t\tDimentionality in training")
print(f"features dim: {X_train.shape} \tTarget dim: {y_train.shape}")

print("\n\t\tDimentionality in Dev")
print(f"features dim: {X_dev.shape} \tTarget dim: {y_dev.shape}")
```

```
                Dimentionality in training
features dim: (16, 8000)        Target dim: (1, 8000)

                Dimentionality in Dev
features dim: (16, 2000)        Target dim: (1, 2000)
```

2.3.4 Build model

For this task, we will not reimplement the optimizer class and the different activation functions. We will reuse those defined previously. Similarly, for our neural network, we will only modify a few methods to adapt it to this regression task. The modifications are as follows:

- Removal of the *proba* method and rewriting of the *predict* method.
- Adaptation of dZ^L for the task as the loss is no longer cross-entropy but MSE.
- Consequently, we also adapt the *backward* method due to the modification of the previous point.

[8]:
```
class NNRegressor:
    def __init__(self, layers_dims, weight_init="He", bias=True,␣
 ↪activation_function=None):
        """
            Initializes the neural network with the specified parameters.
        """
        self.bias = bias
        self.total = 0
        self.memory = {}
        self.gradients = {}
        self.parameters = {}
        self.layers_dims = layers_dims
        self.number_of_layers = len(self.layers_dims)
        self.weight_init = weight_init
```

```
        self.initialize_weights() # initialization of parameters with␣
↪"weight_init" method
        self.activation_function, self.activation_derivative = self.
↪get_activation_functions(activation_function)



    def initialize_weights(self):
        """
        Initializes the weights and biases of the neural network.Three cases␣
↪He, Xavier or Randomly with scale
        """
        #for layer, layer_dim in enumerate(tuple(zip(self.layers_dims, self.
↪layers_dims[1:])), start=1):
        for l in range(1, self.number_of_layers):
            previous_layer_dim, current_layer_dim = self.layers_dims[l-1],␣
↪self.layers_dims[l]#layer_dim

            if self.weight_init.lower() == "he":
                scale = np.sqrt(2 / previous_layer_dim)
            elif self.weight_init.lower() == "xavier":
                scale = np.sqrt(1 / previous_layer_dim)
            else:
                scale = .001

            self.parameters["W" + str(l)] = np.random.randn(current_layer_dim,␣
↪previous_layer_dim) * scale
            if self.bias:
                self.parameters["B" + str(l)] = np.zeros((current_layer_dim,␣
↪1))



    def get_activation_functions(self, activation):
        """
        get the activation function and its derivative based on the provided␣
↪activation name.
        """
        activation = activation.lower() if activation else "relu"

        if activation not in dict_activation:
            raise ValueError(f"Invalid activation function: {activation}.␣
↪Supported values: {list(dict_activation.keys())}")

        return dict_activation[activation]
```

```
    def forward(self, x):
        """
            Performs the forward pass through the neural network.
            m is number of example
        """
        self.memory["A" + str(0)] = A_previous = x
        # Hidden(s) layer(s)
        for l in range(1, self.number_of_layers - 1):
            W = self.parameters["W" + str(l)]
            if self.bias:
                B = self.parameters["B" + str(l)]
                Z = np.dot(W, A_previous) + B
            else:
                Z = np.dot(W, A_previous)
            self.memory["Z" + str(l)] = Z
            A_current = self.activation_function(Z)
            self.memory["A" + str(l)] = A_current
            A_previous = A_current

        # Output layer
        W = self.parameters["W" + str(self.number_of_layers - 1)]
        if self.bias:
            B = self.parameters["B" + str(self.number_of_layers - 1)]
            Z = np.dot(W, A_previous) + B
        else:
            Z = np.dot(W, A_previous)
        self.memory["Z" + str(self.number_of_layers - 1)] = Z
        x = Z
        return x



    def backward(self, X, Y, Y_hat):
        """
            Performs the backward pass (backpropagation) through the neural␣
↪network to compute gradients.
            Keys represent parameter names (e.g., 'dW1', 'dB2'), and values␣
↪are the corresponding gradients.
        """
        m = X.shape[1]
        dZ = 2/m * (Y_hat - Y) # not activation function

        # For output layer
        self.memory["dZ" + str(self.number_of_layers - 1)] = dZ
```

```
        A_previous = self.memory["A" + str(self.number_of_layers - 2)]
        self.gradients["dW" + str(self.number_of_layers - 1)] = np.dot(dZ,␣
↪A_previous.T) / m
        if self.bias:
            self.gradients["dB" + str(self.number_of_layers - 1)] = np.sum(dZ,␣
↪axis=1, keepdims=True) / m

        # For Hidden(s) layer(s)
        for l in reversed(range(1, self.number_of_layers - 1)):
            W_next = self.parameters["W" + str(l + 1)]
            dZ_next = self.memory["dZ" + str(l + 1)]
            Z_current = self.memory["Z" + str(l)]
            A_previous = self.memory["A" + str(l - 1)]
            dZ_current = np.multiply(np.dot(W_next.T, dZ_next), np.
↪vectorize(self.activation_derivative)(Z_current))
            self.gradients["dW" + str(l)] = np.dot(dZ_current, A_previous.T) /␣
↪m
            if self.bias:
                self.gradients["dB" + str(l)] = np.sum(dZ_current, axis=1,␣
↪keepdims=True) / m
            self.memory["dZ" + str(l)] = dZ_current

        return self.gradients



    def total_parameters(self):
        """
        Calculates the total number of parameters in the neural network.
        """
        self.total = 0
        for i in range(1, self.number_of_layers):
            if self.bias:
                self.total += self.layers_dims[i] * (self.layers_dims[i - 1] +␣
↪1)
            else:
                self.total += self.layers_dims[i] * (self.layers_dims[i - 1])
        return self.total


    def predict(self, x):
        """
        Perform value predictions using the trained neural network.
       """
        A = x
        # Hidden layer(s)
```

```
        for l in range(1, self.number_of_layers - 1):
            W = self.parameters["W" + str(l)]
            if self.bias:
                B = self.parameters["B" + str(l)]
                x = self.activation_function(np.dot(W, x) + B)
            else:
                 x = self.activation_function(np.dot(W, x))

        # Output layer
        W = self.parameters["W" + str(self.number_of_layers - 1)]
        if self.bias:
            B = self.parameters["B" + str(self.number_of_layers - 1)]
            out = np.dot(W, x) + B
        else:
            out = np.dot(W, x)
        return out




    def check_gradient(self):
        """
        Placeholder for a method that could be used to check gradients.
        """
        pass

    def __call__(self, x):
        """
        Makes the object callable, allowing for convenient use like a function.
        """
        return self.forward(x)

    def __repr__(self):
        """
        Returns a string representation of the neural network object.
        """
        layer_info = ""
        for i in range(1, self.number_of_layers - 1):
            layer_info += f"\n\tLayer {i}: Linear(in_dim={self.
↪layers_dims[i-1]}, out_dim={self.layers_dims[i]}, bias={self.bias})\
            \n\tActivation function: {self.activation_function.__name__}()"

        return f"NN model with {self.number_of_layers} layers of which {self.
↪number_of_layers - 2} hidden layer\
      \n\n\t(Input layer): (Layer 0 : (dim={self.layers_dims[0]})\
      \n\t(Hidden layers): ({layer_info})\
```

```
        \n\t(Output layer): (Layer {self.number_of_layers - 1}␣
↪Linear(in_dim={self.layers_dims[-2]}, out_dim={self.layers_dims[-1]},␣
↪bias={self.bias})"

    def __str__(self):
        """
        Returns a string representation of the neural network object.
        """
        return self.__repr__()
```

2.3.5 Instantiation and training model

2.3.5.1 Hyperparameters and instanciations

[9]:
```
# hyperparameters
lr = .4
nb_iter  =  2000
output = 1
input_dim = 16
layers_dims = [input_dim, 512, output]

# to have the same initial values at runtime
np.random.seed(6)

# criterion  and optimizer
criterion = mse_loss
optim = Optimizer()

# model
model = NNRegressor(layers_dims, weight_init="Xavier",␣
 ↪activation_function="tanh")
print(f"{model} \n\nNumber of parameters:  {model.total_parameters()/1e6}␣
 ↪Million(s)")

# training and dev logs
train_loss, dev_loss = [], []
```

```
NN model with 3 layers of which 1 hidden layer

        (Input layer): (Layer 0 : (dim=16)
        (Hidden layers): (
        Layer 1: Linear(in_dim=16, out_dim=512, bias=True)
        Activation function: Tanh())
        (Output layer): (Layer 2 Linear(in_dim=512, out_dim=1, bias=True)

Number of parameters:  0.009217 Million(s)
```

2.3.5.2 Training loop

[10]:
```
for i in range(nb_iter):
    # step 1:  Forward
    y_hat = model(X_train)
    # step 2: Compute loss
    loss = np.sqrt(criterion(y_train, y_hat)) # rmse loss
    # step 3: Backward
    grad = model.backward(X_train, y_train, y_hat)
    # step 4: update parameters
    optim.update_parameters_SGD_with_momentum(model, learning_rate=lr)
    # store loss
    train_loss.append(loss)

    # Validation mode: use predict
    y_hat_dev = model.predict(X_dev).reshape(-1).tolist()
    loss_dev = np.sqrt(criterion(y_dev, y_hat_dev))
    dev_loss.append(loss_dev)
    if i % 200 == 0:
        print(f"iter: [{i}/{nb_iter}]  Train loss: {round(train_loss[-1])} |␣
↪Dev loss: {round(dev_loss[-1])}")
```

```
iter: [0/2000]  Train loss: 5759240 | Dev loss: 5778393
iter: [200/2000]  Train loss: 354616 | Dev loss: 366701
iter: [400/2000]  Train loss: 182331 | Dev loss: 199470
iter: [600/2000]  Train loss: 135302 | Dev loss: 154342
iter: [800/2000]  Train loss: 119819 | Dev loss: 140174
iter: [1000/2000]  Train loss: 110941 | Dev loss: 131009
iter: [1200/2000]  Train loss: 104349 | Dev loss: 125172
iter: [1400/2000]  Train loss: 103513 | Dev loss: 123096
iter: [1600/2000]  Train loss: 99474 | Dev loss: 123352
iter: [1800/2000]  Train loss: 97333 | Dev loss: 119427
```

2.3.6 Visualize training and dev logs

[12]:
```
plt.plot(range(1,nb_iter+1), train_loss, label="train loss")
plt.plot(range(1,nb_iter+1), dev_loss, label="dev loss")
plt.title("Loss in Train vs. Dev")
plt.legend()
plt.show()
```

2.3.7 Evaluation

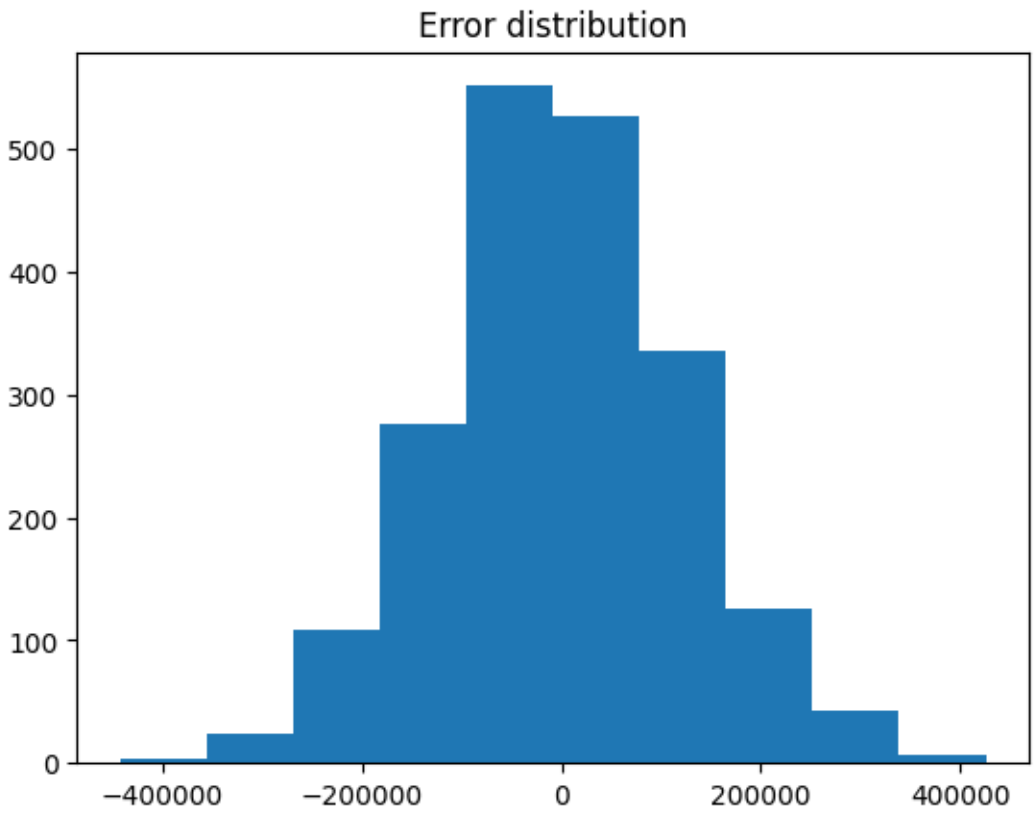
2.3.7.1 RMSE in dev

[15]:
```
print(f"RMSE in dev: {int(np.sqrt(np.mean(np.array(E)**2)))}")
```

```
RMSE in dev: 122684
```

2.3.7.2 Error distribution

[14]:
```
E = (df_dev["price"] - df_dev["pred"]).tolist()

plt.hist(E)
plt.title("Error distribution")
plt.show()
```



2.3.7.3 Prediction

[13]:
```
df_dev["pred"] = model.predict(X_dev).reshape(-1).tolist()
df_dev.loc[:, ["price", "pred"]]
```

[13]:

	price	pred
0	7559081.5	7.641318e+06
2	5574642.1	5.737763e+06
15	7607322.9	7.715967e+06
18	2604486.6	2.620140e+06
20	2888047.9	2.987163e+06
...	...	...
9974	601531.4	5.767606e+05
9983	8766795.5	8.832648e+06
9986	2701055.3	2.612520e+06
9989	9112369.0	9.188656e+06
9995	176425.9	2.769792e+05

[2000 rows x 2 columns]

Conclusion

In summary, this document has provided a thorough overview of neural networks and deep learning, covering essential concepts such as neuron structure, network architecture, activation functions, and the training process. Through practical implementation, we've demonstrated the application of these concepts in solving classification and regression tasks using simple neural network architectures. Additionally, we've highlighted the significance of hyperparameters and their role in model optimization.

By following the systematic workflow outlined in this document, readers can not only grasp the theoretical foundations of neural networks but also gain hands-on experience in building and training models for real-world applications. Whether it's recognizing handwritten digits or predicting housing prices, the principles discussed here lay a solid foundation for diving deeper into the exciting field of deep learning.

References

[1] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton, May 2015. *Deep learning: Review* ResearchGate *DOI:10.1038/nature14539*

[2] Yann Lecun and Y. Bengio, January 1995. *Convolutional Networks for Images, Speech, and Time-Series.* ResearchGate *Conference: The Handbook of Brain Theory and Neural Networks*

[3] Alex Graves, 5 Jun 2014. *Generating Sequences With Recurrent Neural Networks. arXiv:1308.0850v5*

[4] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin, 2 Aug 2023. *Attention Is All You Need. arXiv:1706.03762v7*

[5] Guangming Zhu, Liang Zhang, Youliang Jiang, Yixuan Dang, Haoran Hou, Peiyi Shen, Mingtao Feng, Xia Zhao, Qiguang Miao, Syed Afaq Ali Shah, Mohammed Bennamoun, 22 Jun 2022. *Scene Graph Generation: A Comprehensive Survey. arXiv:2201.00443v2*

[6] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, Yoshua Bengio, 10 Jun 2014. *Generative Adversarial Networks. arXiv:1406.2661v1*

[7] Bo Peng, Eric Alcaide, Quentin Anthony, Alon Albalak, Samuel Arcadinho, Stella Biderman, Huanqi Cao, Xin Cheng, Michael Chung, Matteo Grella, Kranthi Kiran GV, Xuzheng He, Haowen Hou, Jiaju Lin, Przemyslaw Kazienko, Jan Kocon, Jiaming Kong, Bartlomiej Koptyra, Hayden Lau, Krishna Sri Ipsit Mantri, Ferdinand Mom, Atsushi Saito, Guangyu Song, Xiangru Tang, Bolun Wang, Johan S. Wind, Stanislaw Wozniak, Ruichong Zhang, Zhenyuan Zhang, Qihang Zhao, Peng Zhou, Qinghua Zhou, Jian Zhu, Rui-Jie Zhu, 11 Dec 2023. *RWKV: Reinventing RNNs for the Transformer Era. arXiv:2305.13048v2*

[8] *https://openclassrooms.com/fr/courses/5801891-initiez-vous-au-deep-learning/5814626-construisez-des-reseaux-profonds-grace-aux-couches-convolutionnelles*

[9] *https://www.researchgate.net/figure/Artificial-neural-network-There-are-three-layers-an-input-layer-hidden-layers-and-an_fig1_321066887*

[10] *https://www.researchgate.net/figure/Common-activation-functions-in-artificial-neural-networks-NNs-that-introduce_fig7_341310767*

[11] *Slide number 10 from the course on Neural Networks and Deep Learning by Dr. Mohamed Amine Chadi, October 11, 2023 at Mundiapolis University in Casablanca, Morocco.*