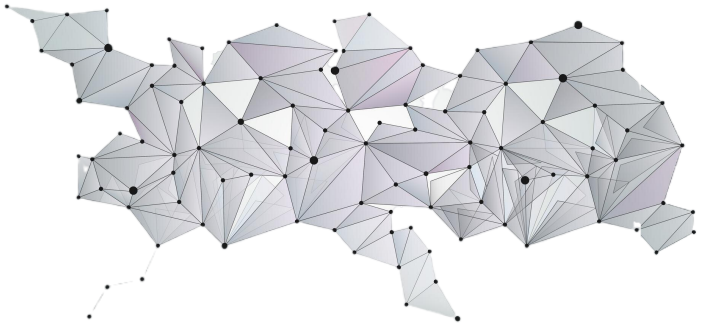

Python Neural Network Implementation + Calculus

Emilio Rodrigo Carreira Villalta

Sunday 4th August, 2024

Table of Contents

List of Figures

1. Introduction

This document explores the process of implementing a neural network from scratch using Python and the NumPy library. The aim is to provide a detailed understanding of the fundamental principles of neural networks through a manual implementation, without relying on advanced machine learning libraries.

The project, available in this GitHub repository [1], is designed to clearly and accessibly illustrate how to build a neural network. It covers everything from defining the network's properties, including its structure and activation functions, to implementing Forward Propagation and Backward Propagation algorithms. These components are essential for the network's operation and training.

The document details the mathematical calculations involved in these processes, offering an in-depth explanation of the formulas and concepts behind each step. Additionally, a comprehensive description of the code is provided so that readers can understand both the "how" and the "why" of each implementation.

This work is inspired by Michael Nielsen's book Neural Networks and Deep Learning [1], which serves as a foundational guide for understanding the principles and techniques discussed herein. The goal is to offer a complete guide that facilitates the understanding of both the theoretical and practical aspects of neural networks, allowing readers to effectively experiment with and learn about the design and training of these artificial intelligence models.

[1] https://github.com/rorro6787/NeuralNetwork

2. What is Machine Learning

Contents

Machine learning is a branch of artificial intelligence that enables systems to learn from and make decisions based on data without explicit programming. Instead of following fixed instructions, machine learning algorithms analyze data to identify patterns and improve their performance over time.

This field encompasses various techniques such as supervised learning, where models are trained on labeled data, and unsupervised learning, where models uncover hidden patterns in unlabeled data. Reinforcement learning, another approach, involves models learning through interaction and feedback.

Applications of machine learning are widespread, including healthcare for predicting diseases and personalizing treatments; finance for detecting fraud and optimizing trading; and everyday technology, such as recommendation systems on streaming platforms. To effectively grasp machine learning, it's important to start with the basics, such as understanding the perceptron, which serves as a foundational concept for more advanced models.

2.1 The perceptron

A perceptron is a fundamental building block in machine learning and neural networks, often used as a basic unit for binary classification tasks. It is the simplest type of artificial neural network. A perceptron is a linear classifier that classifies data into one of two categories. It operates by taking a set of input features, applying weights to these features, and then passing the weighted sum through an activation function to produce an output.

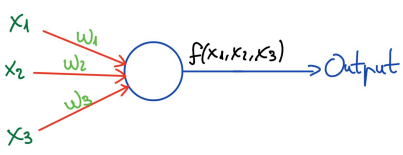

Figure 2.1: The perceptron

$$Output = f(x_1, x_2, x_3) = \begin{cases} 0 & \text{if } (\Sigma_i w_i x_i) + threshold \leq 0 \\ 1 & \text{if } (\Sigma_i w_i x_i) + threshold > 0 \end{cases}$$

We can simplify the mathematical description of perceptrons. Let $\Sigma_i w_i x_i$ be defined as the vector dot product of the input vector and the vector of weights of the perceptron. Additionally, we will replace the threshold notation with b. This b represents how easily the perceptron outputs a 1 (bias).

$$Output = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases}$$

2.2 The sigmoid neuron

A single perceptron is not particularly useful on its own. To perform complex tasks similar to those handled by the human brain, an intriguing approach is to connect multiple perceptrons together, using the outputs of some as inputs to others. However, regardless of how sophisticated the architecture is, perceptrons are limited by their binary outputs, making them vulnerable to small disturbances.

We can address this issue by introducing a new type of artificial neuron known as a sigmoid neuron. These networks are similar to perceptrons, but they use a smoothing activation function (the sigmoid function) to produce their outputs.

$$Output = \sigma(\Sigma_i w_i x_i + b) = \sigma(wx + b)$$

$$\sigma(z) = \frac{1}{1 + e^{-z}} \quad | \quad \sigma(z) \in [0, 1] \ \ \forall z \in \mathcal{R}$$

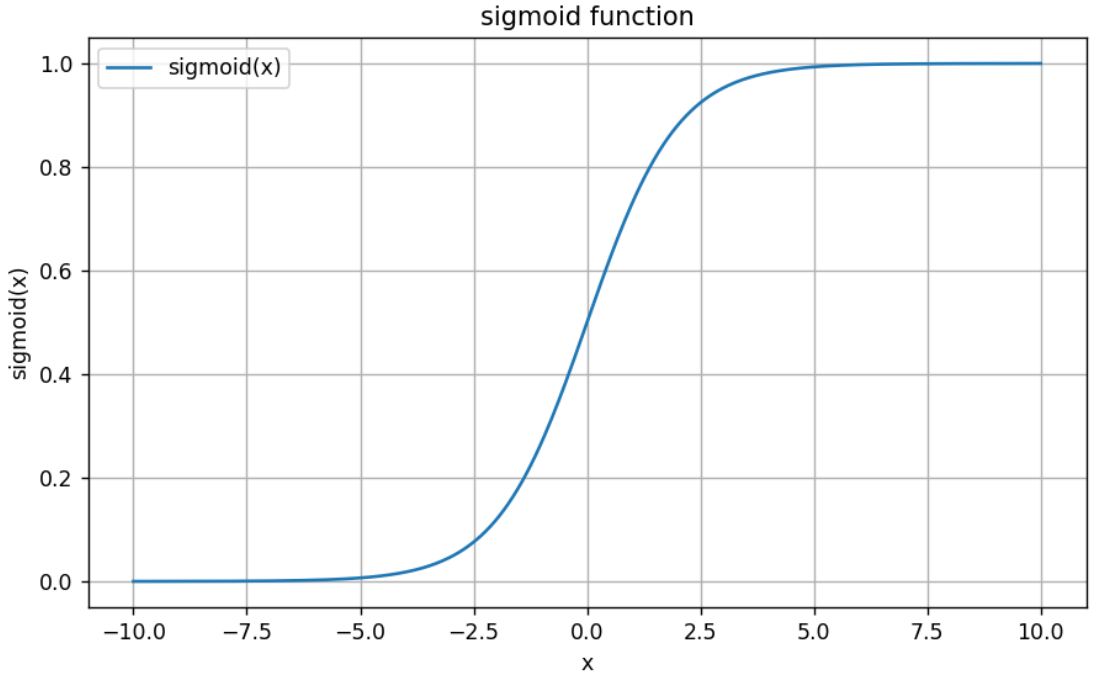

Figure 2.2: The Sigmoid Function

There are many activation functions, each suited for different tasks. However, from now on, when I refer to the activation function, I will be talking about the sigmoid function. If we combine multiple sigmoid neurons, we have a mathematical device capable of performing more complex tasks, but how do these multilayer sigmoid perceptron networks calculate an output?

2.3 Forward Propagation

Now that we have established that the neural networks we will work with, are indeed multilayer sigmoid perceptron networks, we need to formally define their structure and how to evaluate them. The notation is as follows: we will use w^l_{ik} to denote the weight for the connection between the k^{th} neuron in the $(l-1)^{th}$ layer and the i^{th} neuron in the l^{th} layer. Similarly, we will use b^l_i for the bias of the i^{th} neuron in the l^{th} layer and x^l_i for the input of the i^{th} neuron in the l^{th} layer.

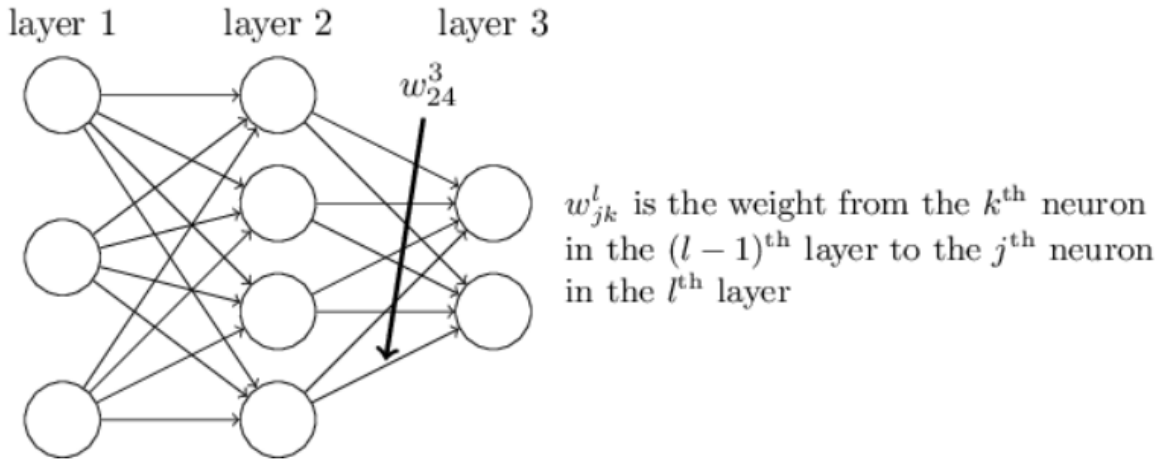

Figure 2.3: Network notation (figure extracted from Michael Nielsen's book)

Having established the nomenclature, we now possess the necessary tools to formally express the output of a layer in the network. Specifically, in the context of a linear network, we can represent the output of the l^{th} layer as a linear transformation of the input, utilizing the respective weights and biases:

$$\begin{pmatrix} x^i_1 \\ x^i_2 \\ \dots \\ x^i_n \end{pmatrix} = \sigma \left(\begin{pmatrix} w^i_{1,1} & w^i_{2,1} & \dots & w^i_{r,1} \\ w^i_{1,2} & w^i_{2,2} & \dots & w^i_{r,2} \\ \dots & \dots & \dots & \dots \\ w^i_{1,n} & w^i_{2,n} & \dots & w^i_{r,n} \end{pmatrix} \cdot \begin{pmatrix} x^{i-1}_1 \\ x^{i-1}_2 \\ \dots \\ x^{i-1}_r \end{pmatrix} + \begin{pmatrix} b^i_1 \\ b^i_2 \\ \dots \\ b^i_n \end{pmatrix} \right)$$

We can also simplify the mathematical notation by using vector notation. In this way, the function that calculates the forward propagation would look like this (where $w^i \cdot x^{i-1}$ represents a matrix multiplication):

$$f(x^i) = \begin{cases} \sigma(w^i \cdot x^{i-1} + b^i) & \text{if} \quad i > 1 \\ x^i & \text{if} \quad i = 1 \end{cases}$$

2.4 Code of the Neural Network

In this section, we showcase and explain the code for the constructor of our Neural Network object, as well as the implementation of the forward propagation function. First, we import the necessary libraries:

```
import numpy as np
import random
```

The next step is to implement at least the sigmoid activation function, which will be used as the default:

```
def sigmoid(z: float) -> float:
    return 1.0/(1.0+np.exp(-z))
```

Since Python is an object-oriented language, we will encapsulate all the information and routines of our neural network within a Python class:

```
class Network(object):
```

The constructor of our Network takes a list specifying its shape (sizes) and constructs the weights and biases matrices. For example, if sizes = [2, 4, 6], it represents a neural network with three layers: the first layer having 2 neurons, the second layer having 4 neurons, and the third layer having 6 neurons:

```
def __init__(self, sizes: List[int]):
    # Dimensions of the neural network
    self.num_layers = len(sizes)
    # Number of layers in the neural network
    self.sizes = sizes

    self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
    self.weights = [np.random.randn(y, x) for x, y in
       ↪ zip(sizes[:-1], sizes[1:])]

    # np.random.rand(a, b) generates a matrix with a rows and
       ↪ b columns with random values
    # sizes[1:] is a list with all the elements in sizes
       ↪ except for the first one
    # sizes[:-1] is a list with all the elements in sizes
       ↪ except for the last one
```

Since the weights and biases matrices are well-defined, implementing the forward propagation function is straightforward, using the mathematical definitions provided earlier:

```
def forward_propagation(self, input, activation=sigmoid):
        for b, w in zip(self.biases, self.weights):
            input = activation(w @ input + b)
            # @ is the matrix multiplication operator
        return input
```

3. How do Neural Networks learn?

Contents

A neural network can be viewed as a complex mathematical function with an arbitrary number of variables. Some of the most powerful generative models have billions of trainable parameters.

When we say we are training a neural network, we mean we are attempting to find the optimal set of weights and biases for the training dataset. Training a neural network involves minimizing a loss function for the training data. However, due to the immense number of variables, finding an analytical solution to this problem is infeasible.

3.1 The loss function

What we aim to create is an algorithm that allows us to find a set of weights and biases that accurately approximate the training data. We denote the desired output of the network for the input x as y(x):

$$C(w,b) = \frac{1}{2n}\sum_{x} \|y(x) - a(x)\|^2$$

For a given training input x, which can have various shapes depending on the problem, y(x) will be a vector filled with 0s and a 1 in the position that represents the desired output. In contrast, a(x) is the output of the network and will also be a vector with the same shape as y(x). The value the network chooses is the one with the highest probability in a(x). The subtraction operation between the two vectors is defined as follows:

$$y(x) = \begin{pmatrix} 0 \\ 1 \\ ... \\ 0 \end{pmatrix} \qquad a(x) = \begin{pmatrix} 0.35 \\ 0.02 \\ ... \\ 0.98 \end{pmatrix}$$

$$y(x) - a(x) = (0 - 0.35) + (1 - 0.02) + ... + (0 - 0.98)$$

This loss function is typically referred to as the MSE, and it indicates how well the network's predictions match the actual outputs. Our goal is to minimize the value of this error function, ideally to 0, by adjusting the values of the weights and biases.

3.2 Gradient Descent

Calculus tells us that when we make small adjustments to the values of the weights, the change in the loss function can be approximated as follows (the same principle applies to biases, but let's focus on weights for the moment):

$$\Delta C \approx \frac{\partial C}{\partial w_1}\Delta w_1 + \frac{\partial C}{\partial w_2}\Delta w_2 + ... + \frac{\partial C}{\partial w_m}\Delta w_m$$

Since we want to minimize C, we need to find values for Δw_i that make ΔC negative. We then define the vector of changes in $\mathbf{w}$ and the vector of the gradient of C as follows:

$$\Delta w = \begin{pmatrix} \Delta w_1 \\ \Delta w_2 \\ ... \\ \Delta w_m \end{pmatrix} \qquad \nabla C = \begin{pmatrix} \frac{\partial C}{\partial w_1} \\ \frac{\partial C}{\partial w_2} \\ ... \\ \frac{\partial C}{\partial w_m} \end{pmatrix}$$

Now, using the vector dot product, we can rewrite the change in the loss function in terms of the gradient and the change in weights:

$$\Delta C \approx \nabla C \cdot \Delta w$$

The amazing part about this equation is that it clearly shows how to ensure ΔC is negative. This introduces our first hyperparameter, the learning rate: η. The learning rate must be greater than 0 and controls the step size of the weight updates during training.

$$\Delta w = -\eta \nabla C$$

$$\Delta C = -\eta ||\nabla C||^2$$

$$w' = w - \eta \nabla C$$

Now we have an expression that we can use to adjust the weights of the network in each iteration. Calculus tells us that if we keep applying this update rule repeatedly, we will continue decreasing C and potentially reach a global minimum. In conclusion, to minimize the loss function, we apply gradient descent to each of the weights and biases as follows:

$$w_k' = w_k - \eta \frac{\partial C}{\partial w_k}$$

$$b_l' = b_l - \eta \frac{\partial C}{\partial b_l}$$

We can also express the gradient descent update rules in matrix form, taking into account that the subtraction operator is the usual element-by-element subtraction:

$$w' = w - \eta \cdot \begin{pmatrix} \frac{\partial C}{\partial w_1} \\ \frac{\partial C}{\partial w_2} \\ ... \\ \frac{\partial C}{\partial w_k} \end{pmatrix} \qquad b' = b - \eta \cdot \begin{pmatrix} \frac{\partial C}{\partial b_1} \\ \frac{\partial C}{\partial b_2} \\ ... \\ \frac{\partial C}{\partial b_l} \end{pmatrix}$$

3.3 Stochastic Gradient Descent

Up to this point, we have all the mathematical tools needed to minimize the loss function and successfully train our neural network. However, the way we have defined the training process requires performing gradient descent for every single training data point. Studies show that as the number of parameters in the network increases, the calculations become significantly slower.

To address this issue, stochastic gradient descent (SGD) was developed. SGD is similar to the original gradient descent, but instead of computing the gradient ∇C for the entire dataset, it estimates the gradient by computing ∇C_x for a small sample of randomly chosen training inputs. By averaging over this small sample, we can quickly obtain a good estimate of the true gradient ∇C, which helps speed up gradient descent and thus accelerates the learning process.

Stochastic Gradient Descent (SGD) works by randomly choosing m examples from the entire dataset of n elements. These m examples constitute what we call the mini-batch. Once the mini-batch is sufficiently large, we use the stochastic approximation of ∇C_{X_j} to ∇C_x:

$$\frac{1}{m}\sum_{j=1}^{m}\nabla C_{X_j} \approx \frac{1}{n}\sum_{x}\nabla C_x = \nabla C$$

$$\nabla C \approx \frac{1}{m}\sum_{j=1}^{m}\nabla C_{X_j}$$

To connect this explicitly to learning in neural networks, suppose w_k and b_l denote the weights and biases in our neural network. Then stochastic gradient descent works by picking out a randomly chosen mini-batch of training inputs and training with those. The equations that describes this are:

$$w_k' \approx w_k - \frac{\eta}{m}\sum_{j=1}^{m}\frac{\partial C_{X_j}}{\partial w_k}$$

$$b_l' \approx b_l - \frac{\eta}{m}\sum_{j=1}^{m}\frac{\partial C_{X_j}}{\partial b_l}$$

3.4 Code of the SGD

Now it's time to move on to the code implementation of everything we have explained so far. Note that we haven't yet discussed how to update the weight and bias matrices. This process involves a complex algorithm called backpropagation, which requires extensive calculus. I prefer to reserve that explanation for later. For now, let's assume that when we call the backpropagation function, it simply works and returns the matrices with updated weights and biases. To begin with, we implement the function that split the training set in multiple minibatches with a fixes size:

```
def SGD(self, training_data, epochs, mini_batch_size, eta):
    if test_data: n_test = len(test_data)
```

```
    n = len(training_data)
    for j in xrange(epochs):
        random.shuffle(training_data)
        mini_batches = [training_data[k:k+mini_batch_size]
            ↪ for k in xrange(0, n, mini_batch_size)]

        # If mini_batch_size = 10, then minibatches will be:
        # mini_batches = [
        # train_data[0:10], train_data[10:20], ...,
            ↪ train_data[n-10:n]
        # ]

        for mini_batch in mini_batches:
            self.update_mini_batch(mini_batch, eta)

        if test_data:
            test_results =
                ↪ [(np.argmax(self.forward_propagation(x)), y)
                ↪ for (x, y) in test_data]
            output = sum(int(x == y) for (x, y) in
                ↪ test_results)
            print(f"Epoch␣{i}:␣{output}␣/␣{n_test}")
        else:
            print(f"Epoch␣{i}␣complete")
```

The second part, is to implement the function that updates the network's weights and biases by applying gradient descent using backpropagation to a single mini batch (we can apply the formulas learnt without paying attention about how the backpropagation function calculates the weights and biases:

```
def update_mini_batch(self, mini_batch, eta):
    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]

    for x, y in mini_batch:
        delta_nabla_b, delta_nabla_w = self.backprop(x, y)
        nabla_b = [nb+dnb for nb, dnb in zip(nabla_b,
            ↪ delta_nabla_b)]
        nabla_w = [nw+dnw for nw, dnw in zip(nabla_w,
            ↪ delta_nabla_w)]

    self.weights = [w-(eta/len(mini_batch))*nw for w, nw in
        ↪ zip(self.weights, nabla_w)]
    self.biases = [b-(eta/len(mini_batch))*nb for b, nb in
        ↪ zip(self.biases, nabla_b)]
```

4. The Backpropagation Algorithm

Contents

The backpropagation algorithm is a fundamental technique used in training artificial neural networks. It is a supervised learning method that adjusts the weights of the network to minimize the error in its predictions. This is achieved by propagating the error backward from the output layer to the input layer. The process involves two main phases: forward propagation, where the input data passes through the network to produce an output, and backward propagation, where the error is calculated and propagated back through the network. During backward propagation, the algorithm uses the chain rule of calculus to compute the gradient of the loss function with respect to each weight, allowing for the precise adjustment of weights to reduce the error.

In practical terms, backpropagation iteratively updates the network's weights by using gradient descent or a variant like stochastic gradient descent (SGD). After each iteration, or epoch, the weights are adjusted in the direction that reduces the error. This process continues until the network's predictions are sufficiently accurate or until another stopping criterion is met, such as a fixed number of epochs. Backpropagation has been instrumental in enabling deep learning, allowing neural networks with many layers (deep neural networks) to be trained effectively. Despite its effectiveness, it requires careful tuning of hyperparameters, such as the learning rate, to ensure convergence and avoid issues like overfitting.

In the context of what we have explained so far, backpropagation is the crucial tool we need to create to complete our working neural network. Before diving into the backpropagation equations, we need to mention two things:

4.1 The Hadamard product

The implementation of backpropagation involves a series of linear algebraic and matrix operations. One particularly useful operation is the Hadamard product, which is a type of matrix multiplication where each element is multiplied element-wise. We denote this operation with the $\odot$ symbol:

$$\begin{pmatrix} a \\ b \\ \dots \\ h \end{pmatrix} \odot \begin{pmatrix} i \\ j \\ \dots \\ t \end{pmatrix} = \begin{pmatrix} a * i \\ b * j \\ \dots \\ h * t \end{pmatrix}$$

4.2 The error of a neuron

As previously discussed, backpropagation is an efficient algorithm that updates the weights and biases of our network based on how the cost function changes. To achieve this, we introduce an intermediate value known as the error of the neuron. This error is defined as the rate of change of the weighted input to the neuron with respect to the cost function:

$$\delta_j^l = \frac{\partial C}{\partial z_j^l}$$

Note that z_j^l is the weighted input of the neuron j in the l^th layer. We use that instead of the output activation to because is algebraically easy to understand:

$$z_j^l = \sum_k w_{j,k}^l \cdot a_k^{l-1} + b_j^l$$

$$a_j^l = \sigma(z_j^l)$$

4.3 The equations of Backpropagation

Backpropagation is based on four fundamental equations. These equations together allow us to compute both the error δ^l and the gradient of the cost function. For each of these four equations, I will first present the equation itself, and you can optionally refer to its proof if needed:

4.3.1 First equation

The initial step in backpropagation is to compute the error in the output layer. This expression provides the error associated with a given input in the output layer, which is essential for propagating the error backward through the network:

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \cdot \sigma'(z_j^L)$$

Here is the element-wise form of the first equation. From this, we can also derive the matrix form of the same equation:

$$\delta^L = \nabla C(a) \odot \sigma'(z^L)$$

There is considerable debate about the best loss function to use, but for the purposes of this explanation, we will use the quadratic cost function, also known as Mean Squared Error (MSE):

$$C(a_j^L) = \frac{1}{2} \cdot \sum_j (y_j - a_j^L)^2$$

$$\frac{\partial C}{\partial a_j^L} = (a_j^L - y_j)$$

With all of this, we can finally derive an explicit formula for the matrix form of the equation using Mean Squared Error (MSE) as the loss function:

$$\delta^L = (a^L - y) \odot \sigma'(z^L)$$

4.3.1.1 Proof of the first equation

Our goal now is to show that we have derived the equation that computes the error of the output layer:

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \cdot \sigma'(z_j^L)$$

In order to prove it, we first recall the definition of the error associated with the weighted input of a neuron:

$$\delta_j^L = \frac{\partial C}{\partial z_j^L}$$

We use the chain rule to express the partial derivative above in terms of partial derivatives with respect to the output activation:

$$\delta_j^L = \sum_k \frac{\partial C}{\partial a_k^L} \frac{\partial a_k^L}{\partial z_j^L}$$

The sum is across all the neurons in the output layer, but since the output activation a_k^L only depends on the weighted input z_j^L for the j^th neuron when $k = j$, the sum disappears, and we can rewrite it as:

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \frac{\partial a_j^L}{\partial z_j^L}$$

We defined above that $a_j^L = \sigma(z_j^L)$, therefore we can rewrite the second term of the expression and obtain the final equation that completes the proof:

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \cdot \sigma'(z_j^L)$$

4.3.2 Second equation

Now that we have an equation that computes the error of the output layer, we need an equation that computes the error of a given layer in terms of the error of the next one (until we reach the output layer):

$$\delta^l = \left((w^{l+1})^T \cdot \delta^{l+1}\right) \odot \sigma'(z^l)$$

This is the matrix form of the equation, but in the same way as in the previous equation, we can obtain the element form of the equation:

$$\delta_j^l = \sum_k w_{k,j}^{l+1} \cdot \delta_k^{l+1} \cdot \sigma'(z_j^l)$$

4.3.2.1 Proof of the second equation

Our goal now is to show that we have derived the equation that computes the error of a given layer in terms of the error of the next one:

$$\delta^l = \left((w^{l+1})^T \cdot \delta^{l+1}\right) \odot \sigma'(z^l)$$

As before, we recall the definition and use the chain rule to rewrite the error in layer l in terms of the error in the next layer:

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} = \sum_k \frac{\partial C}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} = \sum_k \frac{\partial z_k^{l+1}}{\partial z_j^l} \cdot \delta_k^{l+1}$$

$$z_k^{l+1} = \sum_j w_{k,j}^{l+1} \cdot a_j^l + b_k^{l+1} = \sum_j w_{k,j}^{l+1} \cdot \sigma(z_j^l) + b_k^{l+1}$$

$$\frac{\partial z_k^{l+1}}{\partial z_j^l} = w_{k,j}^{l+1} \cdot \sigma'(z_j^l)$$

$$\delta_j^l = \sum_k w_{k,j}^{l+1} \cdot \delta_k^{l+1} \cdot \sigma'(z_j^l)$$

After performing all these transformations, we end up with the element version of the equation that we wanted to prove.

4.3.3 Third equation

Once we have the equations needed to compute the error in all the layers, we need an equation to compute the rate of change of the cost with respect to the biases:

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l$$

In the case of the biases, the error is exactly the rate of change. From this equation, we can obtain its matrix form:

$$\frac{\partial C}{\partial b^l} = \delta^l$$

4.3.3.1 Proof of the third equation

Our goal now is to show that we have derived the equation that computes the the rate of change of the cost with respect to the biases:

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l$$

In this case, to find $\frac{\partial C}{\partial b_j^l}$, we can use the chain rule. The relationship between the biases and the cost function can be expressed through the chain rule as follows:

$$\frac{\partial C}{\partial b_j^l} = \frac{\partial C}{\partial z_j^l} \cdot \frac{\partial z_j^l}{\partial b_j^l}$$

$$z_k^l = \sum_j w_{k,j}^l \cdot a_j^{l-1} + b_k^l$$

$$\frac{\partial z_k^l}{\partial b_j^l} = 1$$

$$\frac{\partial C}{\partial b_j^l} = \frac{\partial C}{\partial z_j^l} = \delta_j^l$$

4.3.4 Fourth equation

Lastly, similar to the biases, we need an equation that computes the rate of change of the cost with respect to the weights:

$$\frac{\partial C}{\partial w_{j,k}^l} = \delta^l \cdot (a^{l-1})^T$$

In the case of the weights, the partial derivatives are computed from quantities of the error and the activation of the previous neuron. From this equation, we can obtain its matrix form:

$$\frac{\partial C}{\partial w^l} = a^{l-1} \cdot \delta^l$$

4.3.4.1 Proof of the fourth equation

Our goal now is to show that we have derived the equation that computes the the rate of change of the cost with respect to the weights:

$$\frac{\partial C}{\partial w^l_{j,k}} = a^{l-1}_k \cdot \delta^l_j$$

In this case, to find $\frac{\partial C}{\partial w^l_{j,k}}$, we can use the chain rule. The relationship between the weights and the cost function can be expressed through the chain rule as follows:

$$\frac{\partial C}{\partial w^l_{j,k}} = \frac{\partial C}{\partial z^l_j} \cdot \frac{\partial z^l_j}{\partial w^l_{j,k}}$$

$$z^l_k = \sum_j w^l_{k,j} \cdot a^{l-1}_j + b^l_k$$

$$\frac{\partial z^l_k}{\partial w^l_{j,k}} = a^{l-1}_j$$

$$\frac{\partial C}{\partial w^l_{j,k}} = a^{l-1}_j \cdot \frac{\partial C}{\partial z^l_j} = a^{l-1}_j \cdot \delta^l_j$$

4.4 The algorithm

We have already shown and proved each of the equations of the backpropagation algorithm. I will now present them together in matrix form, as this is what we will use in our implementation:

$$\delta^L = (a^L - y) \odot \sigma'(z^L)$$

$$\delta^l = \left((w^{l+1})^T \cdot \delta^{l+1}\right) \odot \sigma'(z^l)$$

$$\frac{\partial C}{\partial b^l} = \delta^l$$

$$\frac{\partial C}{\partial w^l} = \delta^l \cdot (a^{l-1})^T$$

With all the equations written, we can now provide a high-level implementation of the backpropagation algorithm:

1. **Forward propagation** : Given a training input x with the desired output y, compute the weighted input of the output layer using:

$$\forall l \in [2, 3, ..., L]$$
$$z^l = w^l \cdot a^{l-1} + b^l$$
$$a^l = \sigma(z^l)$$

2. **Output layer error**: The first step in the backpropagation algorithm is to calculate the error in the output layer associated with the processed training input (remember that we are using MSE as the loss function; if we chose another one, this step would change):

$$\delta^L = \begin{pmatrix} \begin{pmatrix} \sigma(z_1^L) \\ \sigma(z_2^L) \\ ... \\ \sigma(z_n^L) \end{pmatrix} - \begin{pmatrix} y_1 \\ y_2 \\ ... \\ y_n \end{pmatrix} \end{pmatrix} \odot \begin{pmatrix} \sigma'(z_1^L) \\ \sigma'(z_2^L) \\ ... \\ \sigma(z_n^L) \end{pmatrix}$$

3. **Backpropagate the error**: This is the step that gives the algorithm its name. Using the base case of the error calculated in the previous step, we backpropagate the error across all the layers by using the formula that computes the error of an arbitrary layer in terms of the next one:

$$\forall l \in [2, 3, ..., L-1]$$
$$\delta^l = \begin{pmatrix} \begin{pmatrix} w_{1,1}^{l+1} & w_{2,1}^{l+1} & ... & w_{n,1}^{l+1} \\ w_{1,2}^{l+1} & w_{2,2}^{l+1} & ... & w_{n,2}^{l+1} \\ ... & ... & ... & ... \\ w_{1,m}^{l+1} & w_{2,m}^{l+1} & ... & w_{n,m}^{l+1} \end{pmatrix}^T \cdot \begin{pmatrix} \delta_1^{l+1} \\ \delta_2^{l+1} \\ ... \\ \delta_m^{l+1} \end{pmatrix} \end{pmatrix} \odot \begin{pmatrix} \sigma'(z_1^l) \\ \sigma'(z_2^l) \\ ... \\ \sigma'(z_n^l) \end{pmatrix}$$

4. **Gradient of the cost function**: The final step is to compute the gradient of the cost function with respect to the weights and biases:

$$\frac{\partial C}{\partial w^l} = \begin{pmatrix} \frac{\partial C}{\partial w_{1,1}^l} & \frac{\partial C}{\partial w_{2,1}^l} & ... & \frac{\partial C}{\partial w_{n,1}^l} \\ \frac{\partial C}{\partial w_{1,2}^l} & \frac{\partial C}{\partial w_{2,2}^l} & ... & \frac{\partial C}{\partial w_{n,2}^l} \\ ... & ... & ... & ... \\ \frac{\partial C}{\partial w_{1,m}^l} & \frac{\partial C}{\partial w_{2,m}^l} & ... & \frac{\partial C}{\partial w_{n,m}^l} \end{pmatrix} = \begin{pmatrix} \delta_1^l \\ \delta_2^l \\ ... \\ \delta_m^l \end{pmatrix} \cdot \begin{pmatrix} a_1^{l-1} \\ a_2^{l-1} \\ ... \\ a_n^{l-1} \end{pmatrix}^T \qquad \frac{\partial C}{\partial b^l} = \begin{pmatrix} \frac{\partial C}{\partial b_1^l} \\ \frac{\partial C}{\partial b_2^l} \\ ... \\ \frac{\partial C}{\partial b_m^l} \end{pmatrix} = \begin{pmatrix} \delta_1^l \\ \delta_2^l \\ ... \\ \delta_m^l \end{pmatrix}$$

4.5 The code implementation

We have already established all the mathematical tools required to implement our own neural network. It's now time to jump into the code. First, we need to implement the auxiliary functions that backpropagation requires: the derivative of the sigmoid function and the derivative of the loss function (which returns the vector of partial derivatives for the output activation, in this case, the MSE):

```
def sigmoid_prime(z):
    return sigmoid(z)*(1-sigmoid(z))

def cost_derivative(self, output_activations, y):
    return (output_activations-y)
```

Finally, we can move on to the implementation of the backpropagation algorithm. To recap, the algorithm receives a training input with its desired output and returns the gradient of the cost function with respect to both the weights and biases:

```
def backprop(self, x, y):
    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]

    # Forward Propagation
    activation = x
    activations = [x]
    zs = []
    for b, w in zip(self.biases, self.weights):
        z = np.dot(w, activation)+b
        zs.append(z)
        activation = sigmoid(z)
        activations.append(activation)
    # Output layer error
    delta = self.cost_derivative(activations[-1], y) *
       ↪ sigmoid_prime(zs[-1])
    nabla_b[-1] = delta
    nabla_w[-1] = np.dot(delta, activations[-2].transpose())

    # Backpropagate the error
    for l in range(2, self.num_layers):
        z = zs[-l]
        sp = sigmoid_prime(z)
        delta = np.dot(self.weights[-l+1].transpose(), delta)
           ↪ * sp
        # Gradient of the cost function
        nabla_b[-l] = delta
        nabla_w[-l] = np.dot(delta,
           ↪ activations[-l-1].transpose())
    return (nabla_b, nabla_w)
```

5. Testing and Extras

Contents

We have already explored the process of defining a neural network and how it uses calculus to learn classification tasks from data. With this knowledge, we were able to implement our own neural network in raw Python. An important next step is to test how well our neural network performs on a given task.

5.1 Handwritten digits classification

Once you have a trained neural network, you can use it for a multitude of tasks. One very popular and historically significant application is the recognition and classification of handwritten digits. To achieve this, we need to train our neural network using thousands of correctly labeled examples, and then evaluate its performance on a separate set of test data.

5.1.1 The python script

For this purpose, the MNIST dataset provides a comprehensive collection of 60,000 labeled training images of handwritten digits and 10,000 test images. We will create a short Python script to test our neural network and explain how we can adjust different hyperparameters to improve the model's accuracy:

```
import numpy as np
import tensorflow as tf
import network1 as nt
import os

def load_data_wrapper(num_samples=60000):
    # Download the MNIST dataset
    (x_train, y_train), (x_test, y_test) =
       ↪ tf.keras.datasets.mnist.load_data()


    indices = np.random.choice(x_train.shape[0], num_samples,
       ↪ replace=False)
    x_train_sample = x_train[indices]
```

```
    y_train_sample = y_train[indices]

    training_inputs = [np.reshape(x, (784, 1)) for x in
       ↪ x_train_sample]
    training_results = [vectorized_result(y) for y in
       ↪ y_train_sample]
    training_data = list(zip(training_inputs,
       ↪ training_results))

    test_inputs = [np.reshape(x, (784, 1)) for x in x_test]
    test_data = list(zip(test_inputs, y_test))

    return (training_data, test_data)

def vectorized_result(j):
    """Return a 10-dimensional unit vector with a 1.0 in the
       ↪ jth
    position and zeroes elsewhere.  This is used to convert a
       ↪ digit
    (0...9) into a corresponding desired output from the
       ↪ neural
    network."""

    e = np.zeros((10, 1))
    e[j] = 1.0
    return e

def main():
    os.environ['TF_ENABLE_ONEDNN_OPTS'] = '0'

    network = nt.Network([784, 30, 10])
    (training_data, test_data) = load_data_wrapper()

    network.SGD(training_data, 40, 5, 0.1,
       ↪ test_data=test_data)

if __name__ == "__main__":
    main()
```

Once we have the script, there are two important considerations. First, backpropagation is a computationally expensive and slow algorithm due to the NP-completeness of the problem. Training the network on all 60,000 training inputs for multiple epochs involves millions of operations and multiplications, which can take several minutes to compute. To speed up the process, you can reduce the number of training inputs or epochs, but be aware that this may also decrease the accuracy of the model.

Second, there are several hyperparameters of the network that we can adjust to improve its accuracy. These include the mini-batch size, the number of epochs, the learning rate, and the architecture of the network. For example, let's try training

the network with a learning rate of 0.01 (instead of 3, which is typically too high) and keep the other parameters as shown in the code above:

```
Epoch 0: 1754 / 10000
Epoch 1: 1285 / 10000
Epoch 2: 1341 / 10000
Epoch 3: 1275 / 10000
Epoch 4: 1669 / 10000
Epoch 5: 1589 / 10000
...
Epoch 33: 1605 / 10000
Epoch 34: 1598 / 10000
Epoch 35: 1603 / 10000
Epoch 36: 1711 / 10000
Epoch 37: 1679 / 10000
Epoch 38: 1727 / 10000
Epoch 39: 1711 / 10000
```

As you can see, the accuracy was very low compared to what we were hoping for. Let's change the learning rate to 0.1 and see what happens:

```
Epoch 0: 4306 / 10000
Epoch 1: 5477 / 10000
Epoch 2: 5029 / 10000
Epoch 3: 6322 / 10000
Epoch 4: 6606 / 10000
Epoch 5: 7229 / 10000
...
Epoch 33: 7791 / 10000
Epoch 34: 7587 / 10000
Epoch 35: 7616 / 10000
Epoch 36: 7478 / 10000
Epoch 37: 7553 / 10000
Epoch 38: 7482 / 10000
Epoch 39: 7494 / 10000
```

This time, the accuracy improved to nearly 80%. Although there is still considerable room for improvement, the results are promising. I encourage anyone interested to experiment with other hyperparameters to achieve better performance. The key takeaway is that once you have the algorithm, it's a matter of adjusting the parameters until it performs well for your specific use case.

5.2 Extra considerations

In the chapters where I explained the calculus behind neural networks, I consistently used the sigmoid function as the activation function and Mean Squared Error (MSE) as the loss function. However, nowadays, these are not the most commonly used choices for activation and loss functions due to the availability of more effective alternatives.

For loss functions, Cross-Entropy is often preferred over MSE, especially for classification tasks. Cross-Entropy is more suitable for classification because it directly measures the probability of class predictions and penalizes incorrect classifications more effectively. In contrast, MSE is more appropriate for regression tasks, as it does not handle probabilities and class boundaries well, which can lead to slower and less effective training for classification problems. The formula for Cross-Entropy is as follows:

$$C(a^l) = \frac{-1}{n} \cdot \sum_x [y^l \cdot ln(a^l) + (1 - y^l) \cdot ln(1 - a^l)]$$

If we differentiate this function, we can derive the formulas for the error, similar to how we did with the Mean Squared Error (MSE). This process will provide us with the formulas for the partial derivatives of the cost function with respect to the biases and weights. While I won't go through the detailed derivation here, this is how you can obtain the backpropagation formulas for an arbitrary loss function.

Similarly, you can apply this approach to an arbitrary activation function. In practice, you define the activation function, compute its derivative, and then incorporate this expression into the formulas. For example, the ReLU (Rectified Linear Unit) function is often preferred over the sigmoid function. ReLU is advantageous because it mitigates the vanishing gradient problem, leading to faster and more effective training. Unlike sigmoid, which squashes outputs to a small range and can slow down learning, ReLU allows gradients to flow more freely, accelerating the learning process. Here is the formula and plot for the ReLU function:

$$ReLU(z) = max(0, z)$$

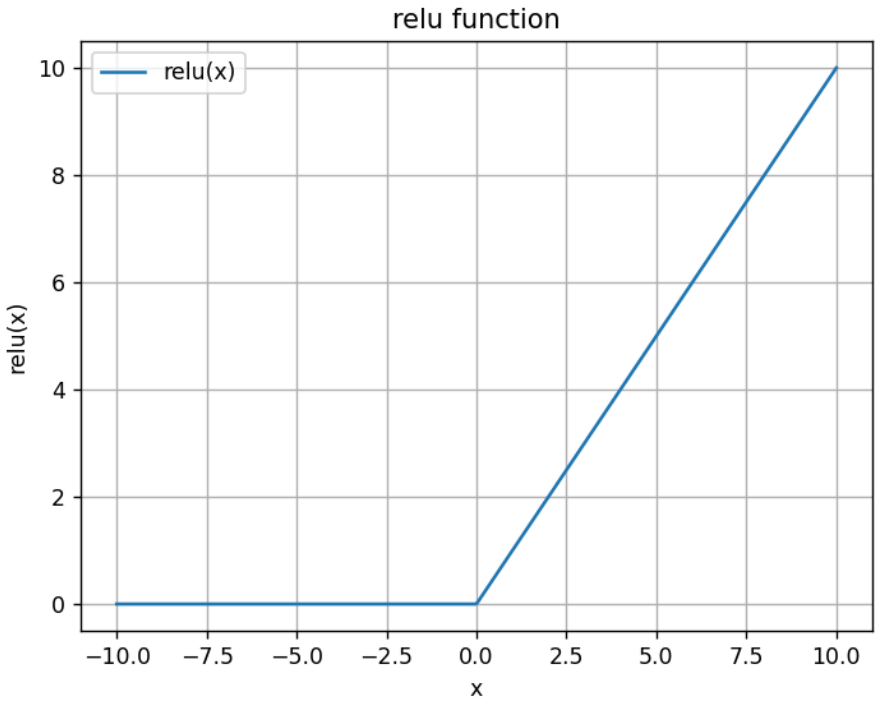

Figure 5.1: The ReLU Function

6. Training a Neural Network is NPC

Contents

In this chapter, after discussing the code implementation and mathematical definitions, we will explain why, even today, training a neural network to function correctly remains an extraordinarily slow and expensive process.

We will demonstrate that training a neural network is an NPC problem by restricting our case to a network with just 3 nodes. This document will follow the demonstration scheme presented in the article by Avrim L. Blum and Ronald L. Rivest [2].

6.1 Description of the 3NN Problem

To begin with, we need to provide an extensive description of the problem we want to prove is NP-complete. To ensure the explanation is both rigorous and easy to understand, we will provide both a formal and an informal definition.

6.1.1 Informal definition of the problem

Given a neural network $\mathcal{N}$, we informally define the training problem on $\mathcal{N}$ as follows:

Given a training set, does there exist a set of weights and threshold values such that $\mathcal{N}$ produces results consistent with the training set?

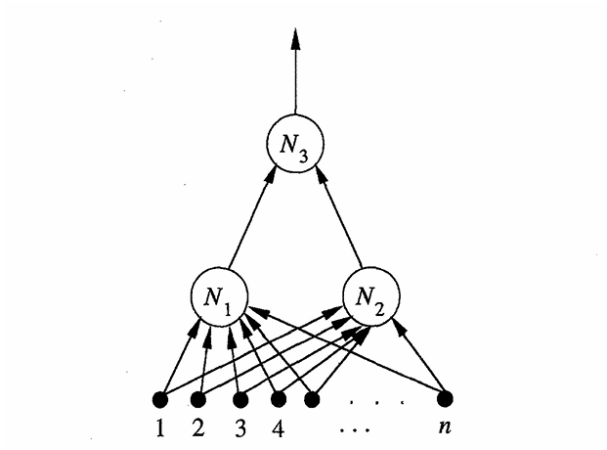

Figure 6.1: Neural network of 3 nodes

6.1.2 Formal Definition of the Problem

From the image of the neural network 6.1 with 3 nodes and n input objects, we can derive the function that calculates the value of each node:

$$N_i(\vec{x}) = \begin{cases} 1 & \text{if } a_1x_1 + a_2x_2 + \cdots + a_mx_m > a_0 \\ 0 & \text{otherwise} \end{cases}$$

- $\mathbf{N}_i$ is the i-th node in my neural network $\mathcal{N}$.
- $\vec{\mathbf{x}}$ is the vector of input values for the node $\mathbf{N}_i$.
- $\mathbf{a}_i$ $(i \neq 0)$ is the set of weights for the node $\mathbf{N}_i$.
- $\mathbf{a}_0$ is the threshold for the node $\mathbf{N}_i$.

Given this definition of a node in a neural network, we can now derive the formal definition of a neural network $\mathcal{N}$. A neural network is a tuple (N, R) where N is the set of nodes in the network and R is the relation between each node and its set of weights and threshold values:

$$\mathcal{N} = (N, R)$$

$$N = \{N_1, N_2, \ldots, N_p\}$$

$$\begin{aligned} R = \{&(N_1, \{a_{1,0}, a_{1,1}, a_{1,2}, \ldots, a_{1,n_1}\}), \\ &(N_2, \{a_{2,0}, a_{2,1}, a_{2,2}, \ldots, a_{2,n_2}\}), \\ &\ldots, \\ &(N_p, \{a_{p,0}, a_{p,1}, a_{p,2}, \ldots, a_{p,n_p}\})\} \end{aligned}$$

Given a neural network with n inputs, a training object would be the set of values that feed into the n inputs. We can formally define the set of training objects as:

$$S = \{x_1, x_2, \ldots, x_n\}$$

$$x_i = (y_i, \mathrm{VR}_i)$$

$$y_i = \{z_{i,1}, z_{i,2}, \ldots, z_{i,k}\}$$

VR_i is the actual value that the network is expected to return when the object x_i is input.

We need to define a function $\mathcal{C}$ that, given a neural network $\mathcal{N}$ and a set of training objects S, determines whether the neural network is consistent with the set of test objects:

$$\mathcal{C}(\langle \mathcal{N}, S \rangle) = \begin{cases} \text{true} & \text{if } \mathcal{N} \textbf{ is } \text{consistent with } S \\ \text{false} & \text{if } \mathcal{N} \textbf{ is not } \text{consistent with } S \end{cases}$$

With all these definitions, we can formally define the problem of training a neural network with 3 nodes, which I will refer to from now on as **3NN**:

$$\textbf{3NN} = \{\langle \mathcal{N}, S \rangle \mid \mathcal{N} = (N, R), |N| = 3, \exists\, \mathcal{N}' = (N, R') \text{ such that } \mathcal{C}(\langle \mathcal{N}', S \rangle) = \text{true}\}$$

6.1.3 Simple Valid Example

Given the complexity of algorithms used in neural networks for learning, let's consider a simple example where each input object has a size of 1. We will have only two input objects:

$$\mathbf{l_1} = (\mathcal{N}, S)$$

$$S = \{x_1, x_2\}, \quad x_1 = (y_1, \text{VR}_1), \quad x_2 = (y_2, \text{VR}_2)$$

$$y_1 = \{1\}, \quad y_2 = \{0\}, \quad \text{VR}_1 = 1, \quad \text{VR}_2 = 0$$

$$\mathcal{N} = (N, R)$$

$$N = \{N_1, N_2, N_3\}$$

$$R = \{(N_1, \{a_{1,0}, a_{1,1}\}), (N_2, \{a_{2,0}, a_{2,1}\}), (N_3, \{a_{3,0}, a_{3,1}, a_{3,2}\})\}$$

The question is whether there exists a combination of weights and thresholds for the network such that it is consistent with this small set of test cases. The algorithm that modifies the weight values to fit the test object set is called "backpropagation." This algorithm uses differential calculus and gradient descent to maximize the learning process.

For this small example, we will not use backpropagation to find the ideal set of weights, as it is unnecessary and can be done somewhat intuitively. The goal is simply to demonstrate that such a set of weights exists. Therefore, we could use the following values for the weights and ***thresholds***:

$$R' = \{(N_1, \{0, 1\}), (N_2, \{0, 1\}), (N_3, \{0, 1, 1\})\}$$

$$N_3 = \text{“output node”}$$

Now, let's see what the neural network returns with this combination of nodes when we input the test cases. To calculate the value of the output node, we first need to calculate the values of the two internal nodes that receive the input:

$$N_1(\{y_1\}) = \begin{cases} 1 & \text{if } \ a_{1,1}y_1 > a_{1,0} \\ 0 & \text{otherwise} \end{cases} \rightarrow \ N_1(\{1\}) = 1$$

$$N_2(\{y_1\}) = \begin{cases} 1 & \text{if } \ a_{2,1}y_1 > a_{2,0} \\ 0 & \text{otherwise} \end{cases} \rightarrow \ N_2(\{1\}) = 1$$

$$N_3(\{N_1(1), N_2(1)\}) = N_3(\{1,1\}) = \begin{cases} 1 & \text{if } \ a_{3,1}(1) + a_{3,2}(1) > a_{3,0} \\ 0 & \text{otherwise} \end{cases} \rightarrow \ N_3(\{1,1\}) = 1$$

$$\mathcal{N}(y_1) = 1 \land \mathrm{VR}_1 = 1$$

The neural network with the selected weights has been able to accurately predict the actual value of the first object. Now let's see what happens with the second object:

$$N_1(\{y_2\}) = \begin{cases} 1 & \text{if } \ a_{1,1}y_2 > a_{1,0} \\ 0 & \text{otherwise} \end{cases} \rightarrow \ N_1(\{1\}) = 0$$

$$N_2(\{y_2\}) = \begin{cases} 1 & \text{if } \ a_{2,1}y_2 > a_{2,0} \\ 0 & \text{otherwise} \end{cases} \rightarrow \ N_2(\{1\}) = 0$$

$$N_3(\{N_1(1), N_2(1)\}) = N_3(\{1,1\}) = \begin{cases} 1 & \text{if } \ a_{3,1}(1) + a_{3,2}(1) > a_{3,0} \\ 0 & \text{otherwise} \end{cases} \rightarrow \ N_3(\{1,1\}) = 0$$

$$\mathcal{N}(y_2) = 0 \land \mathrm{VR}_2 = 0$$

The neural network has also been able to accurately predict the actual value of the second object. This demonstrates that this neural network $\mathcal{N}$ with this assignment of weights and thresholds has been able to produce consistent results on the training set $\mathcal{S}$, therefore:

$$\mathbf{l_1} \in \mathbf{3NN}$$

6.1.4 Simple Invalid Example

As such, given a neural network $\mathcal{N}$ and a training set, with a sufficient number of iterations it usually manages to adapt successfully to the test cases. However, since we are working with a neural network with only 3 nodes, if we create a training set that is quite large and make each value very similar to each other but with different images, it is very likely that the neural network will not be able to adapt to the test cases. For example:

$$\mathbf{l_2} = (\mathcal{N}', S')$$

$$S' = \{\{1, 1\}, \{0'99, 0\}, \{0'98, 1\}, \{0'97, 0\}, \{0'96, 1\}, \{0'95, 0\}\}$$
$$\mathcal{N}' = (N', R')$$
$$N' = \{N_1, N_2, N_3\}$$
$$R' = \{(N_1, \{a_{1,0}, a_{1,1}\}),$$
$$(N_2, \{a_{2,0}, a_{2,1}\}),$$
$$(N_3, \{a_{3,0}, a_{3,1}, a_{3,2}\})\}$$

It is almost impossible for a neural network with only 3 nodes to accurately distinguish the values given the training set S' due to their similarity. Because of this:

$$\mathbf{l}_2 \notin \mathbf{3NN}$$

6.1.5 Application Areas

Artificial neural network models have a multitude of applications and uses in our daily lives. Here are just a few of the many areas where neural networks are transforming the way we interact with technology and tackle complex problems:

- **Voice Recognition**: Used in virtual assistant systems like Siri, Alexa, or Google Assistant to convert speech into text and perform actions.
- **Image Recognition**: Employed in facial recognition applications, object classification in photographs, and security systems such as video surveillance.
- **Natural Language Processing (NLP)**: Applied in machine translation, sentiment analysis on social media, automatic text generation, and chatbots.
- **Autonomous Driving**: In autonomous vehicles to detect pedestrians, vehicles, and traffic signs, and make real-time decisions.
- **Recommendation Systems**: Used by streaming platforms like Netflix and Spotify to recommend movies, music, or products to users.
- **Finance**: In credit risk analysis, fraud detection, and market price prediction.
- **Medicine**: In medical diagnosis, analysis of medical images such as X-rays and MRIs, and drug development.

6.2 3NN $\in$ NP

In the process of proving the NP-completeness of **3NN**, the first step is to demonstrate that it belongs to the class NP. To achieve this, there are two approaches. For this report, we will prove that **3NN** $\in$ **NP** by showing that it has a verifier that operates in polynomial time.

6.2.1 Verifier for 3NN

In the verifier-based proof, the goal is to show that an algorithm $\mathcal{V}$ that operates in polynomial time is capable of verifying whether a certificate or proof $\mathcal{P}$ belongs to the language.

$$\mathbf{3NN} = \{\langle \mathcal{N}, S\rangle \mid \mathcal{N} = (N, R),\ |N| = 3,\ \exists\, \mathcal{N}' = (N, R') : \mathcal{C}(\langle \mathcal{N}', S\rangle) = \text{true}\}$$

$\mathcal{N}'$ is the certificate $\mathcal{P}$

6.2.1.1 Specification of the Algorithm $\mathcal{V}$

$\mathcal{V}$ takes as input $\langle\langle \mathcal{N}, S\rangle, \mathcal{N}'\rangle$

1. Check if $\mathcal{N}'$ is consistent with the training set S. Specifically, verify that $\mathcal{C}(\langle \mathcal{N}', S\rangle) =$ true.
2. Check if $\mathcal{N}$ contains all the nodes N_i in $\mathcal{N}'$.
3. If steps 1 and 2 are satisfied, accept. Otherwise, reject.

6.2.1.2 Complexity of the Algorithm $\mathcal{V}$

The complexity of the algorithm $\mathcal{V}$ can be computed by summing the complexities of steps 1 and 2 separately (since step 3 is a constant-time action). Step 2 also has constant complexity because we are dealing with neural networks of 3 nodes. This means that the complexity of the algorithm is determined by step 1.

The input to this problem is the training set. Each training object has a size k (number of "inputs" to the neural network), so the size of the input for n training objects follows the expression:

$$\text{size} = n \times k$$

For each training object, each of the two internal nodes of our network must perform k operations. Finally, the node that computes the result from the two internal nodes evaluating object i performs two additional operations. Thus, the number of operations to evaluate the neural network with object i follows the expression:

$$\text{image}(n_i) = 2k + 2$$

This means that the number of operations $t(n)$ required for the algorithm to verify step 1 is:

$$t(n, k) = n(2k + 2) = 2kn + 2n$$
$$t(n, k) \in O(n \cdot k)$$

This demonstrates that the algorithm $\mathcal{V}$ can verify in **poly-time** whether the certificate $\mathcal{P}$ belongs to **3NN**. Therefore, it guarantees that:

$$\textbf{3NN} \in \textbf{NP}$$

6.3 3NN $\in$ NPC

To prove that a problem is **NP-complete**, we generally follow the same reasoning framework. We either demonstrate that any problem in **NP** can be reduced to it, or we reduce a problem that has already been proven to be **NP-complete** to it:

$$B \in \textbf{NPC} \iff [(\forall A \in \textbf{NP} \ : \ A \leq_p B) \wedge (B \in \textbf{NP})]$$
$$B \in \textbf{NPC} \iff [(A \in \textbf{NPC}) \wedge (B \in \textbf{NP}) \wedge (A \leq_p B)]$$

For this demonstration, we will choose the first approach. We will introduce a problem that has already been proven to be **NP-complete** and show that a polynomial reduction can be performed to it.

6.3.1 The SS Problem

The problem that is **NP-complete** and that we will use for the reduction is the **Set-Splitting** problem, which from now on I will refer to as **SS**. It was proven **NP-complete** by 'László Lovász' [4].

6.3.1.1 Informal Definition of the Problem

Given a finite set $\mathbf{S}$ and a collection $\mathbf{C}$ of subsets $\mathbf{c_i}$ of $\mathbf{S}$, do there exist two disjoint sets $\mathbf{S_1}$, $\mathbf{S_2}$ such that $\mathbf{S_1} \cup \mathbf{S_2} = \mathbf{S}$ and for every $\mathbf{c_i} \not\subset \mathbf{S_1}$ and $\mathbf{c_i} \not\subset \mathbf{S_2}$?

6.3.1.2 Formal Definition of the Problem

$$\mathbf{SS} = \{\langle \mathbf{S}, \mathbf{C} \rangle \mid \exists\, \mathbf{S_1}, \mathbf{S_2} : (\mathbf{S_1} \cup \mathbf{S_2} = \mathbf{S}) \wedge (\forall \mathbf{c_i} \in \mathbf{C} : \mathbf{c_i} \not\subset \mathbf{S_1} \wedge \mathbf{c_i} \not\subset \mathbf{S_2})\}$$

6.3.1.3 Valid Simple Example

Let's construct an instance of the **SS** problem that **is** valid. To do this, I will construct an arbitrary set S and an arbitrary collection of subsets (each contained in S):

$$\mathbf{l_1} = (S, C)$$

$$S = \{1, 2, 3, 4\} \quad C = \{c_1, c_2, c_3\}$$

$$c_1 = \{1, 2\} \quad c_2 = \{2, 3\} \quad c_3 = \{3, 4\}$$

Now, let's construct two sets S_1 and S_2 such that they satisfy the conditions to belong to the language **SS**:

$$S_1 = \{1, 3\} \quad S_2 = \{2, 4\}$$

$$(S_1 \cup S_2 = S) \wedge (c_1 \not\subset S_1) \wedge (c_1 \not\subset S_2) \wedge (c_2 \not\subset S_1) \wedge (c_2 \not\subset S_2) \wedge (c_3 \not\subset S_1) \wedge (c_3 \not\subset S_2)$$

$$\mathbf{l_1} \in \mathbf{SS}$$

6.3.1.4 Invalid Simple Example

Let's construct an instance of the **SS** problem that **is not** valid. To do this, I will construct an arbitrary set S' and an arbitrary collection of subsets (each contained in S'):

$$\mathbf{l_2} = (S', C')$$

$$S' = \{1, 2, 3\} \quad C' = \{c'_1, c'_2, c'_3\}$$

$$c'_1 = \{1, 2\} \quad c'_2 = \{2, 3\} \quad c'_3 = \{1, 3\}$$

Now, let's show that there is no pair of sets whose union is S' that satisfies the remaining restrictions:

$$S_1 = \{1,2,3\},\ S_2 = \emptyset \rightarrow c'_1 \subset S_1 \rightarrow \mathbf{l_2} \notin \mathbf{SS}$$

$$S_1 = \emptyset,\ S_2 = \{1,2,3\} \rightarrow c'_1 \subset S_2 \rightarrow \mathbf{l_2} \notin \mathbf{SS}$$

$$S_1 = \{1,2\},\ S_2 = \{3\} \rightarrow c'_1 \subset S_1 \rightarrow \mathbf{l_2} \notin \mathbf{SS}$$

$$S_1 = \{3\},\ S_2 = \{1,2\} \rightarrow c'_1 \subset S_2 \rightarrow \mathbf{l_2} \notin \mathbf{SS}$$

$$S_1 = \{2,3\},\ S_2 = \{1\} \rightarrow c_2' \subset S_1 \rightarrow \mathbf{l_2} \notin \mathbf{SS}$$

$$S_1 = \{1\},\ S_2 = \{2,3\} \rightarrow c_2' \subset S_2 \rightarrow \mathbf{l_2} \notin \mathbf{SS}$$

$$S_1 = \{1,3\},\ S_2 = \{2\} \rightarrow c_3' \subset S_1 \rightarrow \mathbf{l_2} \notin \mathbf{SS}$$

$$S_1 = \{2\},\ S_2 = \{1,3\} \rightarrow c_3' \subset S_2 \rightarrow \mathbf{l_2} \notin \mathbf{SS}$$

For each and every possible pair of sets S_1 and S_2, the language membership check for the problem resulted in non-membership. This implies that:

$$\mathbf{l_2} \notin \mathbf{SS}$$

6.3.1.5 Fields of Application

The **SS** problem has applications in various branches of science. Here are some examples:

- **Graph Theory**: In graph theory, **SS** can be used to find partitions in a graph where each subset forms a connected component.
- **Combinatorial Optimization**: In optimization problems, it is often required to divide a set into subsets in a way that satisfies certain constraints, such as minimizing the number of subsets or maximizing some objective function.
- **Network Design**: In network design, it can be useful to partition a set of nodes into disjoint subsets representing different components or segments of the network.
- **Algorithm Analysis**: **SS** can be useful in algorithm analysis to demonstrate the complexity of certain problems or to design efficient algorithms to solve them.
- **Scheduling**: In scheduling, **SS** can be used to allocate resources to different time periods in such a way that no conflicts occur between them.
- **Database Processing**: In database processing, **SS** can be useful for partitioning large datasets into smaller subsets for analysis and processing.
- **Cryptography**: In cryptography, **SS** can be applied to divide secrets into parts that must be combined to recover the original information.

6.3.2 Reduction from SS to 3NN

As we have discussed before, we will demonstrate that **3NN** is an **NP-complete** problem by constructing a polynomial-time reduction to **SS**:

$$3\text{NN} \in \mathbf{NPC} \iff [(\text{SS} \in \mathbf{NPC}) \land (3\text{NN} \in \mathbf{NP}) \land (\text{SS} \leq_p 3\text{NN})]$$

$$\text{SS} \leq_p 3\text{NN}$$

$$\exists f : \forall w \in \text{SS} \iff f(w) \in 3\text{NN}$$

f operates in poly-time

6.3.2.1 3NN from the Geometric Point of View

Before proceeding to the proof of the aforementioned bi-implication, it will be very useful to view the **3NN** problem from a geometric perspective. In this way, a training object can be understood as a point in the **boolean** n-dimensional space:

$$x_i = \{\{0, 1\}^n, \{+' \mid -'\}\}$$

In this manner, the value of each of the points can be +' or -' depending on whether they are an object with value 1 or -1. Due to the presence of the **Threshold** function in each of the two internal nodes $\mathbf{N_1}$ and $\mathbf{N_2}$, these can be interpreted as planes in the defined **boolean** space. $\mathbf{N_1}$ and $\mathbf{N_2}$ divide the space into 4 quadrants due to the 4 possible pairs of values resulting from their calculation:

$$(\mathbf{N_1}, \mathbf{N_2}) = (+1, +1)$$

$$(\mathbf{N_1}, \mathbf{N_2}) = (-1, -1)$$

$$(\mathbf{N_1}, \mathbf{N_2}) = (+1, -1)$$

$$(\mathbf{N_1}, \mathbf{N_2}) = (-1, +1)$$

If the planes are parallel, it means that 1 or 2 of the quadrants do not exist (they are overlapping). Since the outer node $\mathbf{N_3}$ receives the values resulting from the calculation of $\mathbf{N_1}$ and $\mathbf{N_2}$, it is only capable of distinguishing between points in different quadrants. That is, if it receives a pair of values indicating that the points are in the same quadrant, such as (+1, +1), it should return '+'. With all this said, we can assume that the **3NN** problem is equivalent to the one defined below which we will call **CEB**:

Given a collection of points belonging to $\{0, 1\}^n$ and that can have as an image +' or -', does any of the following options exist?

A single plane that separates the points with '+' label from those with '-' label?

or

Two planes that partition the points such that either one quadrant contains all the points with '+' label or one quadrant contains all the points with '-' label?

In the context of proving the bidirectional implication, it will be much simpler to work with a restricted version of **CEB**. We will call it the problem of the quadrant of positive **boolean** examples or simply **CEBP**. It will be defined as follows:

Given O(n) points that belong to ,0,,1,^{n} and can have a '+' or '-' label, do there exist two planes that partition the points such that one quadrant contains all the points with a '+' label and none with a '-' label?

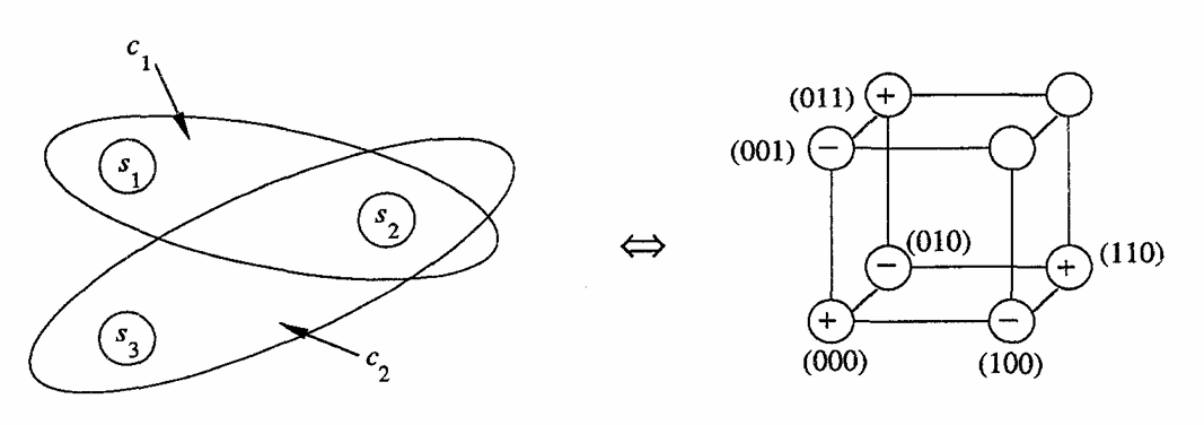

Figure 6.2: Conceptual transformation

Once we have demonstrated that this is **NP-complete**, we will extend the proof to the full problem by adding examples that do not allow the other possibilities at the output node.

6.3.2.2 First Direction of the Implication

The idea in the first direction of the implication is that given an instance of **SS**, we can convert it into an instance of **CEBP** (which will later be extended to **CEB**) such that the constructed instance has a solution if and only if the instance of **SS** had a solution:

$$\exists f : \forall w \in \text{SS} \Longrightarrow f(w) \in \text{CEBP}$$

That is, given an arbitrary instance of the "Set-Splitting" problem such as:

$$S = \{s_1, s_2, ..., s_i\} \quad |S| = n$$

$$C = \{c_1, c_2, ..., c_j\} \ : \ c_k \subset S$$

We will create the following points in the n-dimensional **boolean** space as follows:

- The origin point will be $(0^n, \text{`+'})$
- For each value s_i, a point with '-' label will be created with all 0s and a 1 in position i $\rightarrow (\overset{1}{0}\overset{2}{0} ... \overset{i}{1} ... \overset{n-1}{0}\overset{n}{0}, \text{`}-\text{'})$
- For each c_k, a point with '+' label will be created with 1s in the positions i where $s_i \in c_k$ and 0s in the remaining positions.

Now we demonstrate that the constructed instance of the **CEBP** problem has a solution if the given instance of the **SS** problem had one. For this, given S_1, S_2 (solution of the **SS** instance):

$$P_1 \equiv a_1x_1 + ... + a_nx_n = -\frac{1}{2} \qquad P_1 \equiv b_1x_1 + ... + b_nx_n = -\frac{1}{2}$$

$$a_i = \begin{cases} -1 & \text{if} \quad s_i \in S_1 \\ n & \text{if} \quad s_i \notin S_1 \end{cases} \qquad b_i = \begin{cases} -1 & \text{if} \quad s_i \in S_2 \\ n & \text{if} \quad s_i \notin S_2 \end{cases}$$

$$\vec{a} = (a_1, a_2, ..., a_n)$$

$$\vec{b} = (b_1, b_2, ..., b_n)$$

$$\vec{x} = (x_1, x_2, ..., x_n)$$

Due to the way they have been constructed, for each point $\vec{x}$ with a '+' label, it is guaranteed that $\vec{x} * \vec{a} > -\frac{1}{2}$. Thus, both the quadrant $\vec{x} * \vec{a} > -\frac{1}{2}$ and the quadrant $\vec{x} * \vec{b} > -\frac{1}{2}$ contain all the points with a '+' label and none with a '-' label. P_3 handles distinguishing whether the planes P_1 and P_2 are parallel (returning '+') or not (returning '-') to produce the final result as seen in 6.2.

6.3.2.3 Second Sense of Implication

The idea in the second sense of implication is that given an instance of **CEBP**, we should be able to convert it to an instance of **SS** such that the constructed instance has a solution if and only if the **CEBP** instance had a solution:

$$\forall f(w) \in \text{CEBP} \implies w \in \text{SS}$$

This is equivalent to proving that given an instance that does not belong to **SS**, the transformation returns an instance that does not belong to **CEBP**:

$$\forall w \notin \text{SS} \implies f(w) \notin \text{CEBP}$$

As such, we have already constructed and demonstrated how the "gadget" works. However, I will emphasize that if the given instance of the **SS** problem has no solution, when constructing the list of points, our planes will not be able to separate all points with image '+' from those with image '-'. This is because the positions will overlap, and when we calculate the result by summing the weights, we will obtain values that do not satisfy the established properties.

6.3.2.4 Extension of "CEBP" to "CEB"

It is necessary to extend the transformation a bit because with **CEBP**, we are not making an equivalence with **3NN** because we are restricting the exterior node to perform an "AND" operation with the results of the two interior nodes. However:

$$\textbf{3NN} \equiv \textbf{CEB}$$

$$(\exists f : \forall w \in \text{SS} \iff f(w) \in \text{CEB}) \equiv (\exists f : \forall w \in \text{SS} \iff f(w) \in \text{3NN})$$

Proving the bi-implication for **CEB** involves accepting different modes of operation for the exterior node (not just an "AND"). To achieve this, given an instance of **SS**, we follow the same steps as in **CEBP** but add 3 new dimensions x_{n+1}, x_{n+2}, and x_{n+3} and add the following points to the list of points created:

$$(0 \ldots \ 0101, \text{'+'}), (0 \ldots \ 0011, \text{'+'}), (0 \ldots \ 0100, \text{'-'}), (0 \ldots \ 0010, \text{'-'}),$$

$$(0 \ldots \ 0001, \text{'-'}), (0 \ldots \ 0111, \text{'-'})$$

The points with '+' in these 3 new dimensions can be separated from those with '-' by appropriately calibrating the weights of P_1 and P_2 corresponding to these 3 new dimensions. In this way, the equations of the original planes can be expanded to the following:

$$P_1 \equiv a_1 x_1 + ... + a_n x_n + x_{n+1} + x_{n+2} - x_{n+3} = -\frac{1}{2}$$

$$P_1 \equiv b_1 x_1 + ... + b_n x_n - x_{n+1} - x_{n+2} + x_{n+3} = -\frac{1}{2}$$

$$a_i = \begin{cases} -1 & \text{if} \quad s_i \in S_1 \\ n & \text{if} \quad s_i \notin S_1 \end{cases} \qquad b_i = \begin{cases} -1 & \text{if} \quad s_i \in S_2 \\ n & \text{if} \quad s_i \notin S_2 \end{cases}$$

The rest of the points remain as they were. Only 3 zeros are added to each point to account for the 3 new dimensions included. With this change in the plane equations, P_1 can separate the point (0...0001, '-') from those with '+', while P_2 can separate the other 3 new points with '-' from those with '+'.

With this solution, we do meet the problem constraints of **CEB** since neither of the two planes alone is capable of separating the points with '+' from the new points with '-'. There is also no way for the two planes to confine all the points with '-' into the same quadrant. Because of this, any solution to the **3NN** problem must have all the points with '+' in the same quadrant and also provide a solution to the **SS** problem.

6.3.2.5 Complexity of the Transformation

After explaining the process of creating the transformation function, it is quite evident that it operates in polynomial time, but it is still necessary to calculate its complexity to complete the proof of **NP-completeness**. When the transformation is performed, it creates an "origin" point; as many points with '-' as there are values in the set **S**; and as many points with '+' as there are subsets of **S** in **C**. Six extra points are added at the end, representing the 3 new dimensions of the extended solution. In the input of the **SS** problem:

$$\mathbf{t} = |\, S \,|$$

$$\mathbf{k} = |\, C \,|$$

Each point consists of (t+3) 0s and 1s. The size occupied by all the points created by our transformation can be expressed in terms of the **SS** input as follows:

$$\textbf{size} = (t+3) * (1 + t + k + 6) = t^2 + tk + 10t + 3k + 9$$

$$\textbf{size} \in O(t^2 + tk)$$

The function representing the time needed for our transformation to create the set of points is the transformation time. The time taken to assign the weight values to each of the planes is linear (thus negligible). This means that:

$$\exists f : \forall w \in \text{SS} \iff f(w) \in \text{3NN}$$

f operates in poly-time

$$\text{SS} \leq_p \text{3NN}$$

3NN $\in$ **NPC**

7. Bibliography

[1] **Michael Nielsen**, *Neural Networks and Deep Learning*, Dec 2019, http://neuralnetworksanddeeplearning.com/ pages

[2] **Avrim L. Blum and Ronald L. Rivest**, *Training a 3-Node Neural Network is NP-Complete*, MIT Laboratory for Computer Science Cambridge, Massachusetts 02139. pages

[3] **Michael Sipser**, *Introduction to the Theory of Computation*, 3rd Ed., Boston: Cengage Learning, 2012. pages

[4] **László Lovász**, *On the Shannon capacity of a graph*, IEEE Transactions on Information Theory, vol. 25, no. 1, pp. 1-7, January 1979. pages