Notes on Neural Networks

Paulo S. A. Sousa

2023-05-28

Abstract

Neural networks are introduced as a model for classification and regression problems. Their tendency to overfitting and a possible solution is discussed.
A new remedy for the problem of imbalanced datasets is presented.

1 Introduction

A neural network is a very powerful method of predicting, both classification and regression problems. It mimics the functioning of the human brain, having cells that connect to other cells likewise neurons. The number of cells and the number of connections among them, only bounded by the available computational power, make it possible for a neural network to attain a very high level of complexity, which, in turn, makes it capable of learning very complex relations that may be present in the data.

Neural networks are mathematical models that represent a system of equations and functions. Each artificial neuron receives input from other neurons, processes the input using a mathematical function, and produces an output that can be transmitted to other neurons in the network.

The learning process in neural networks typically involves adjusting the weights (parameters) of the connections between neurons. This is done using optimization algorithms that minimize a predefined loss function. The loss function measures the difference between the network's predicted output and the actual output for a given set of inputs. By minimizing the loss function, the network learns to recognize patterns.

2 Structure and components of a neural network

A neural network is composed of a collection of interconnected nodes, or artificial neurons, which are organized into layers, as the figure below shows (the circles represent cells or neurons):

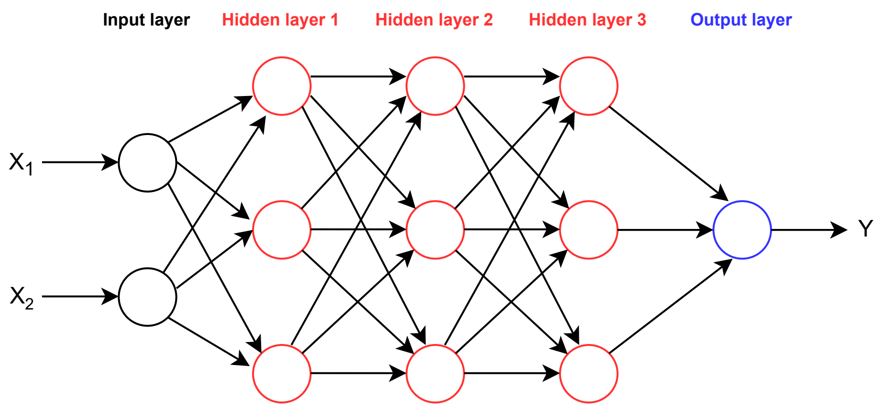


Figure 1: A neural network with 3 hidden layers.

The layers in a neural network can be categorized into three main types:

- **Input layer:** This layer consists of neurons that receive input data, the predictors. Each neuron in the input layer represents a single predictor of the input data.
- **Hidden layer(s):** These layers are situated between the input and output layers and contain neurons that perform intermediate processing. Hidden layers transform the input data into a representation that can be used by the output layer for

recognizing patterns. Neural networks can have multiple hidden layers, and the number of neurons in each hidden layer can vary depending on the complexity of the problem being solved.

- **Output layer:** The output layer consists of neurons that produce the final predictions made by the network. The number of neurons in the output layer depends on the specific task being performed, such as the number of classes in a classification problem or the number of possible outputs in a regression problem.

2.1 Activation functions

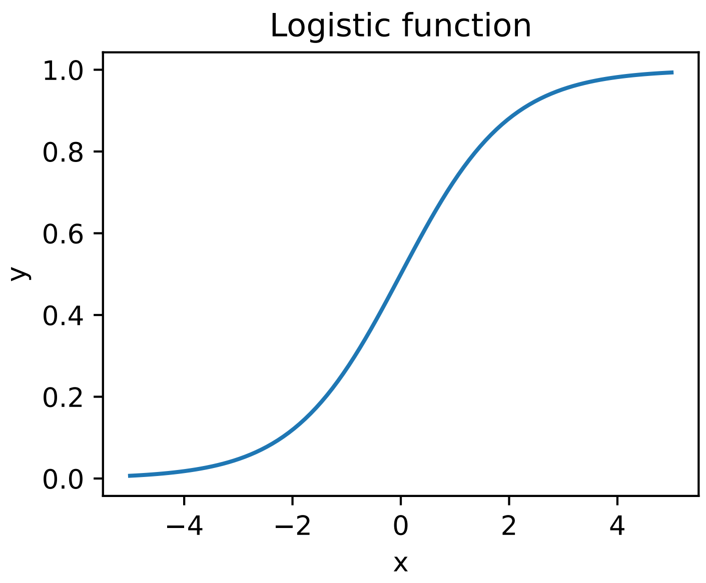
Activation functions are mathematical functions that determine the output of a neuron based on its input. They introduce non-linearity into the network, allowing neural networks to learn and represent complex patterns and relationships in the input data. Activation functions are applied to the weighted sum of the inputs and the bias term within a neuron. Some common activation functions include:

- **Sigmoid or logistic function:** The sigmoid function maps the input to a value between 0 and 1. It is commonly used in output layers for binary classification problems.

  Equation:

$$f(x) = \frac{1}{1 + e^{-x}}.$$

  Graph:

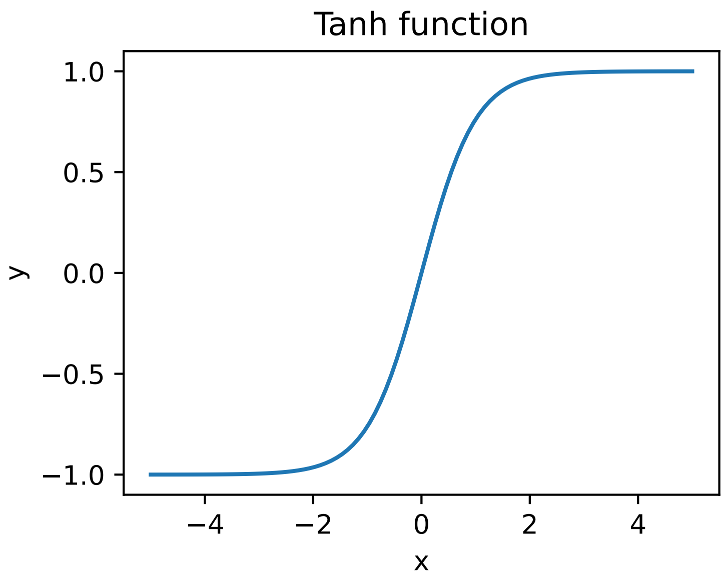


- **Hyperbolic tangent (tanh) function:** The tanh function maps the input to a value between -1 and 1. It is similar to the sigmoid function but has a broader range of output values.

Equation:

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Graph:

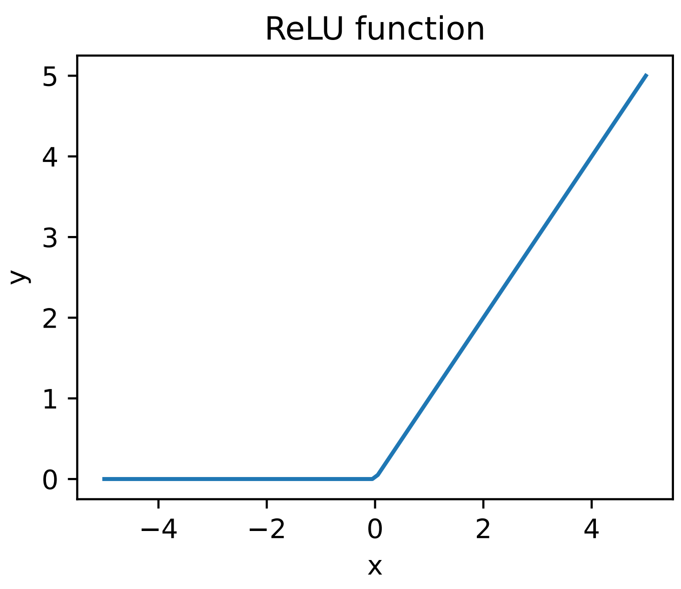


- **Rectified Linear Unit (ReLU) function:** The ReLU function is a piecewise linear function that outputs the input value if it is positive and zero otherwise. ReLU is popular due to its computational efficiency and ability to mitigate the vanishing gradient problem in deep networks (deep learning, in very simplified words, means using neural networks with a large number of hidden layers with a large number of neurons).

  Equation:

$$f(x) = \max(0, x)$$

  Graph:



2.2 Weighted connections between neurons and bias terms

In artificial neural networks, the neurons within different layers are connected through weighted connections. These connections represent the flow of information within each pair of neurons in the network. Each connection has a corresponding weight, which represents the strength of the relationship between the two connected neurons. The weights determine the influence of the input from one neuron on the output of another neuron.

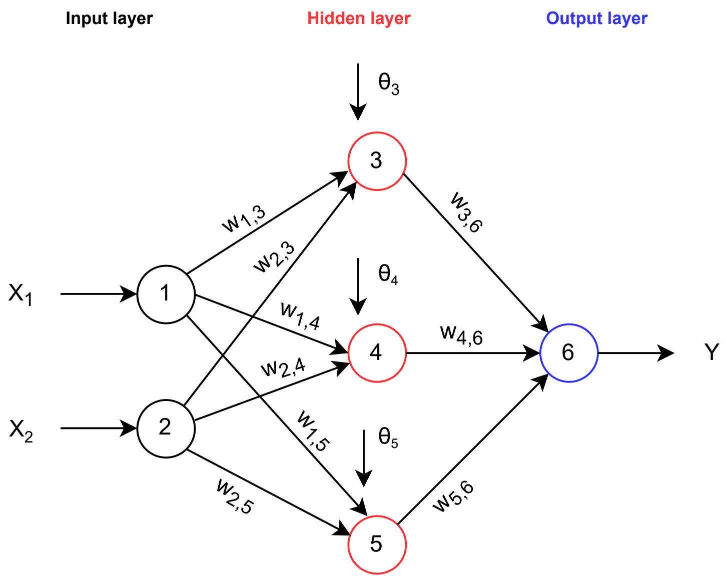


Figure 2: A neural network with a single hidden layer.

In the figure, the weights are w_{ij}, where i and j are the indices of the connecting neurons.

During the forward pass, the input of a neuron is calculated as the sum of the products of the output values from the previous layer's neurons and their corresponding weights. This input value is then passed through an activation function to generate the output of the neuron. The learning process in a neural network involves adjusting these connection weights to minimize a predefined loss function, enabling the network to make better predictions or decisions.

In addition to the weighted connections, each neuron in a neural network (except those in the input layer) also has a bias term. The bias term is a constant value that is added to the weighted sum of the inputs before passing it through the activation function. The bias term allows the network to represent more complex functions by shifting the activation function along the input axis. This helps the network learn patterns and relationships in the input data that may not be linearly separable.

In the figure, the bias terms are represented by θ_i, where i is the index of the neuron.

Similar to the connection weights, the bias terms are also adjusted during the learning process to minimize the loss function. Both weights and biases are considered parameters of the neural network and are learned through optimization algorithms, such as gradient descent.

To illustrate the role of the weights and bias terms in a neural network, let us calculate the output of neuron 3 in the figure, assuming the logistic activation function:

$$\frac{1}{1+e^{-(w_{13}X1+w_{23}X2+\theta_3)}}.$$

The weights and biases in a neural network represent the strength of the relationships between neurons in the network. They determine how the output of one neuron influences the input and, subsequently, the output of another neuron. When a connection has a large weight, it means that the output of the connected neuron has a strong influence on the input of the next neuron. Conversely, when a connection has a small weight, the influence is weak.

3 Learning and Optimization

Loss functions, also known as cost functions or objective functions, are used to quantify the difference between the predicted outputs and the actual output labels in the training dataset. The objective of the learning process in neural networks is to minimize the loss function by adjusting the weights and biases of the network. By minimizing the loss function, the network learns to make more accurate predictions, thus improving its performance on the training data.

3.1 Mean squared error and cross-entropy

There are various loss functions used in machine learning, and the choice of the loss function depends on the specific problem and the nature of the output variable. Some common loss functions include:

- **Mean Squared Error (MSE):** The mean squared error is commonly used in regression tasks, where the output variable is continuous. MSE calculates the average of the squared differences between the predicted values and the actual output values. The goal is to minimize the MSE to improve the model's performance.

  Equation:

  $$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_{\text{actual}} - Y_{\text{predicted}})^2$$

  In the formula, n represents the total number of samples, Y_{actual} represents the actual values, and $Y_{\text{predicted}}$ represents the predicted values.

- **Cross-entropy:** Cross-entropy loss, also known as log loss, is commonly used in classification tasks, where the output variable is categorical. Cross-entropy measures the difference between the predicted probability distribution and the true probability distribution of the class labels. Minimizing cross-entropy loss leads to better classification performance.

  Equation (binary classification):

  $$\text{CE} = -\sum_{i=1}^{n}[Y_{\text{actual}}\log(Y_{\text{predicted}}) + (1 - Y_{\text{actual}})\log(1 - Y_{\text{predicted}})]$$

  In the formula, Y_{actual} represents the actual values (0 or 1, where 0 represents the negative class and 1 represents the positive class), and $Y_{\text{predicted}}$ represents the predicted probability of being of the positive class[1]. The formula calculates the cross-entropy by summing the two terms for each sample, where the first term computes the log likelihood of the actual class, and the second term computes the log likelihood of the opposite class. The summation is then multiplied by -1 to obtain the final cross-entropy value.

  Loss functions quantify the difference between the predicted outputs and actual output labels, and the network's objective is to minimize the loss function by adjusting its weights and biases. Common loss functions include mean squared error for regression tasks and cross-entropy for classification tasks.

3.2 Gradient descent algorithm

Gradient descent is an optimization algorithm that is used to minimize the loss function by iteratively updating the weights and biases of the network. (Other typical optimization algorithms have a gradient descent algorithm as a base.) The algorithm calculates the gradient (partial derivatives) of the loss function concerning each weight and bias and then updates the parameters using a learning rate (step size) multiplied by the negative gradient. This process continues until the loss function converges to its minimum value (ideally a global minimum).

[1] In case this probability is zero, a small positive epsilon value is used instead, to avoid having a logarithm of zero.

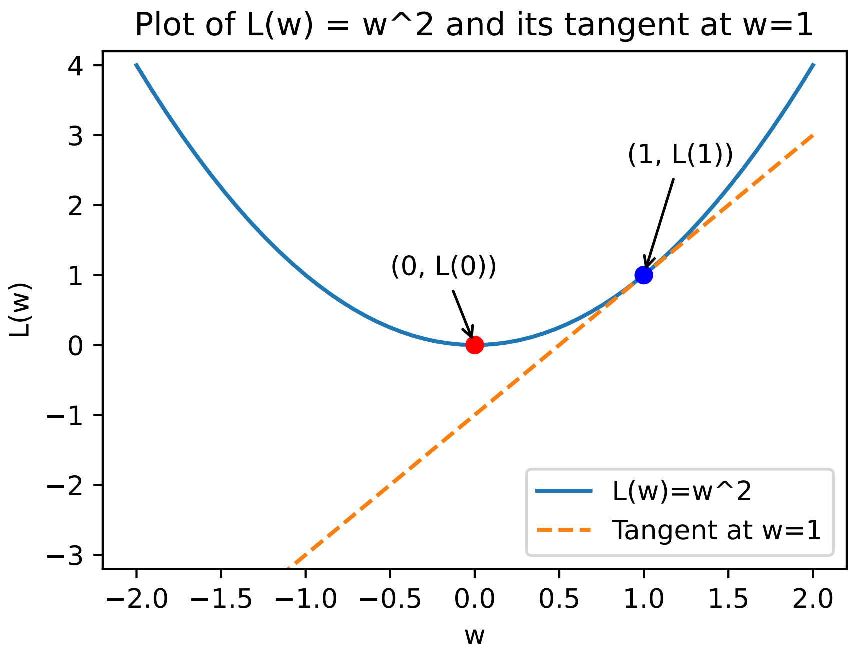
3.2.1 Simple example of application of gradient descent

Suppose that we want to minimize the function

$$L(w) = w^2.$$

This function, whose graph is below, has a single (global) minimum at $w = 0$.

As we have learned from elementary calculus, the derivative of a function at a given point corresponds to the slope of the tangent line to the graph of the function at that point. In the below figure, the tangent line to the graph of the function L is drawn at $w = 1$:



Placed at $w = 1$, in search of the minimum of L, should we go to the right or the left? Well, the chosen direction should be guided by the slope of the tangent line: If positive, we should go to the left; if negative, we should go to the right. And how large should the step be? The step should be proportional to the magnitude of the tangent's slope: The larger the slope, that is, when L is varying the more, the larger can the step can be. Thus, the fundamental formula for the gradient descent algorithm is:

$$w_{k+1} = w_k - \eta L'\left(w_k\right),$$

where η is the proportionality constant, known as the *learning rate.*

By using a learning rate of $\eta = 0.1$ and departing from $w_0 = 1$, we can calculate by hand the successive points given by the gradient descent algorithm, taking into consideration that $L'(w) = 2w$.

Iteration 1

$$\begin{aligned} w_1 &= w_0 - \eta 2 w_0 \\ &= 1 - 0.1 \times 2 \times 1 = 0.8. \end{aligned}$$

Iteration 2

$$\begin{aligned} w_2 &= w_1 - \eta 2 w_1 \\ &= 0.8 - 0.1 \times 2 \times 0.8 = 0.64. \end{aligned}$$

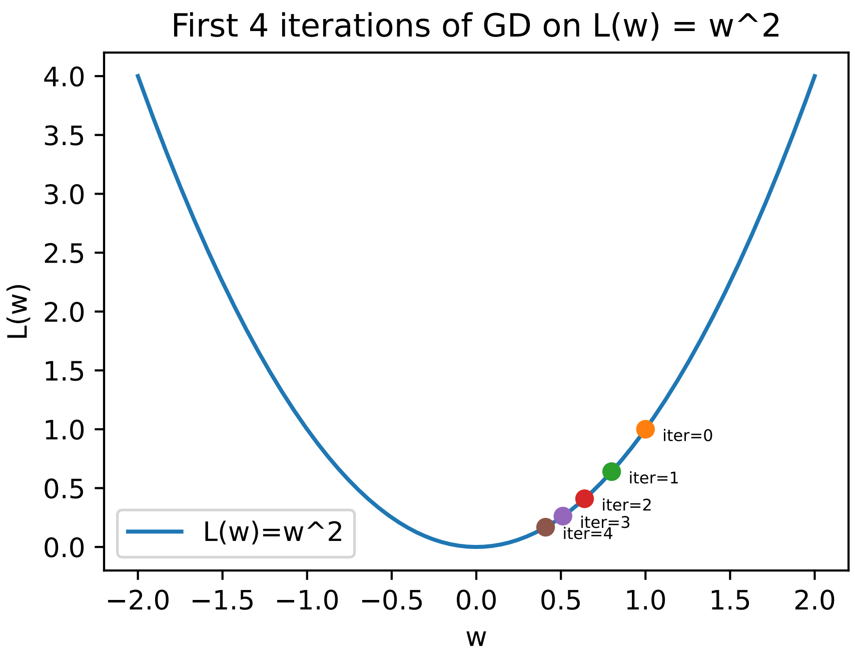
Iteration 3

$$\begin{aligned} w_3 &= w_2 - \eta 2 w_2 \\ &= 0.64 - 0.1 \times 2 \times 0.64 = 0.512. \end{aligned}$$

Iteration 4

$$\begin{aligned} w_3 &= w_2 - \eta 2 w_2 \\ &= 0.512 - 0.1 \times 2 \times 0.512 = 0.4096. \end{aligned}$$

Graphically:



As we can see, the successive values of w get closer and closer to $w = 0$. The iterations could go on and on up to convergence. Python can be used to automatize the algorithm:

```
w = 1
eta = 0.1
n_iter = 100 # number of iterations

for i in range(n_iter):
    w = w - eta * 2 * w

print(w)
```

```
2.0370359763344877e-10
```

After 100 iterations, we have found the minimum of $L(w) = w^2$.

We may be tempted to think that by increasing the learning rate, we can find the minimum of the function $L(w) = w^2$ more efficiently. This may not be true: the minimum may be missed when using a too-large learning rate. In effect, for $\eta = 1.1$, after 100 iterations, we get $w = 82817974.5$, a value too different from zero! (Use the above code to find this result.) Consequently, we should be very judicious about choosing a good value for the learning rate – we can use grid search and cross-validation for that.

4 Prevent overfitting

Regularization techniques are used in artificial neural networks to prevent overfitting, which occurs when the model learns the training data too well, capturing noise or random fluctuations instead of the underlying patterns. Overfitting results in poor generalization performance when the model is presented with new, unseen data. Regularization methods introduce constraints or penalties to the learning process, encouraging the model to learn simpler representations of the input data and improving its generalization capabilities.

4.1 L2 regularization

L2 regularization is a technique that can help prevent overfitting by discouraging complexity in the model. It adds a penalty term to the loss function equal to the sum of the squares of all the weights in the model, scaled by a factor α. This factor α is a hyperparameter, i.e., it is not learned by the model but must be set by the user or determined through cross-validation. When α is larger, the penalty for complexity is more severe, which encourages the model to be simpler and thereby helps to prevent overfitting.

The neural network estimators in `sklearn` (`MLPClassifier` and `MPLRegressor`) have built-in support for L2 regularization: The `alpha` parameter.

Remember that setting an appropriate value for α. If α is too large, the model will be overly simple and may underfit the data. If α is too small, the model may still overfit the data. Therefore, it is usually a good idea to try out several values for α to see which one works best, using, for instance, grid search and cross-validation.

5 Imbalanced data

In classification problems, when the class of interest for us is under-represented in the dataset, the neural network learning will be biased to predict better the classes that do not interest us. For instance, in the case of the bank marketing campaign, the class `yes` is much rarer in the dataset. However, we only care to predict well the class `yes` (the customers that will accept our product) – the cases of class `no` are not relevant to the goal of the bank.

One way we have used to solve this problem has been to manipulate the `class_weight` parameter. Unfortunately, this parameter is not available in the `scikit-learn` neural network estimator (`MLPClassifier`). Therefore, we have to resort to another approach: Random over-sampling.

Focusing on binary classification, random over-sampling resamples, with replacement, the minority class cases of the training set to get more samples of the minority class, making, in this way, both classes balanced. In other words, the random over-sampling procedure duplicates rows of the training set that correspond to the minority class, to increase the number of samples of the minority class that is equal to the number of samples of the majority class.

5.1 Implementation in Python

We need first to load the function that performs the random over-sampling function:

```
from imblearn.over_sampling import RandomOverSampler
```

Then, we can use the function to perform the random over-sampling, which has to be applied *only* to the training set:

```
ros = RandomOverSampler(random_state=45)
x_res, y_res = ros.fit_resample(x_train, y_train)
```

Finally, the final pipeline (the one containing the model) has to be fit to the resampled training set:

```
pipe.fit(x_res, y_res)
```

After the model has been fitted, we can use it to predict the outcome variable of the training (without any resampling) and test sets. (To see the random over-resampling code in action, please see the code that we used in the lecture for the case of the bank marketing campaign.)