National University of Computer and Emerging Sciences



# Laboratory Exercise

**Data Structures Lab**

**(CL 2001)**

## Department of Computer Science

**DS Lab Mini Project**

**Objectives**

* Graph Implementation.

### Note: Carefully read the following instructions.

1. You have to do all tasks on Microsoft Visual Studio.
2. Screenshot the solution to each problem and paste it in a word file with the naming convention F20xxxx\_Section\_lab number.

**Project**

**Perceptron**

**Introduction**

Diagram

Description automatically generated**Neural networks from scratch**

Neural networks are usually represented, **vertices(neurons) + oriented edges(dendrites)**, a directed graph.

For complete beginners, you can try playing with neural network on this [webpage.](https://playground.tensorflow.org/#activation=tanh&batchSize=10&dataset=circle&regDataset=reg-plane&learningRate=0.03&regularizationRate=0&noise=0&networkShape=4,2&seed=0.13081&showTestData=false&discretize=false&percTrainData=50&x=true&y=true&xTimesY=false&xSquared=false&ySquared=false&cosX=false&sinX=false&cosY=false&sinY=false&collectStats=false&problem=classification&initZero=false&hideText=false) The easy version of how a neural network does the forward pass is as follows:

1. create a neural network following the above architecture
2. put values in the input neurons (exp: 3 neurons with value 1, 5 and 3)
3. trigger the input layer. The values are multiplied with a weight linking each input neuron to each neuron of the 2nd layer

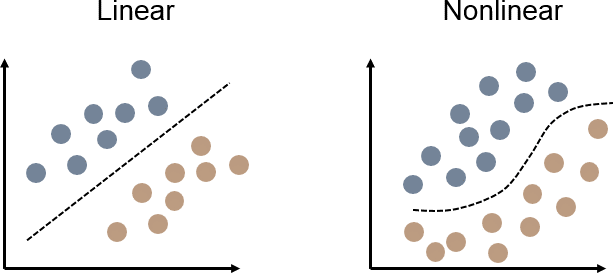
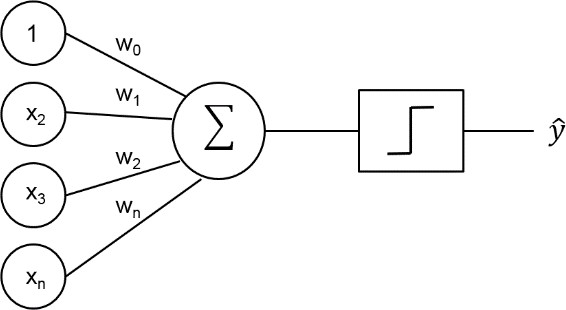
(3neurons\*4neurons=12scalars/weights/edges). Then sum the 3 inputs\*weights for each of the 4 neurons of the 2nd layer

1. Trigger the 2nd layer and do (2) between the 3rd layer and the 2nd layer (the same way it was done between the 2nd and the 1st layer)
2. Read the two values accumulated by the 3rd/output layer

The forward pass is easy. The same computations can be done with matrices. What’s harder to understand and important to re-implement is the backward pass, the method used to learn the right weights. The inputs [1, 5, 3] and outputs of each layer depend on the data we’re processing. The weights [a1, b1, c1, a2..] are here to process all inputs correctly. We’ll focus on finding the good weights.

The Perceptron algorithm is the simplest type of artificial neural network.

It is a model of a single neuron that can be used for two-class classification problems and provides the foundation for later developing much larger networks.



**Perceptron Algorithm**

The Perceptron is inspired by the information processing of a single neural cell called a neuron.

A neuron accepts input signals via its dendrites, which pass the electrical signal down to the cell body. In a similar way, the Perceptron receives input signals from examples of training data that we weight and combined in a linear equation called the activation.

activation = sum (weight\_i \* x\_i) + bias

The activation is then transformed into an output value or prediction using a transfer function, such as the step transfer function.

prediction = 1.0 if activation >= 0.0 else 0.0

The weights of the Perceptron algorithm must be estimated from your training data using stochastic gradient descent.

**Stochastic Gradient Descent**

Gradient Descent is the process of minimizing the error by following the gradients of the cost function. In machine learning, we can use a technique that evaluates and updates the weights every iteration called stochastic gradient descent to minimize the error of a model on our training data.

The way this optimization algorithm works is that each training instance is shown to the model one at a time. The model makes a prediction for a training instance, the error is calculated and the model is updated in order to reduce the error for the next prediction. This procedure can be used to find the set of weights in a model that result in the smallest error for the model on the training data.

For the Perceptron algorithm, each iteration the weights (w) are updated using the equation:

w = w + learning\_rate \* (expected - predicted) \* x

Where w is weight being optimized, learning\_rate is a learning rate that you must configure (e.g. 0.01), (expected – predicted) is the prediction error for the model on the training data attributed to the weight and x is the input value.

**Algorithm**

This tutorial is broken down into 3 parts:

1. Making Predictions.

2. Training Network Weights.

These steps will give you the foundation to implement and apply the Perceptron algorithm to your own classification predictive modeling problems.

**Making Predictions**

Develop a function that can make predictions.

Below is a function named **predict ()** that predicts an output value for a row given a set of weights. The first weight is always the bias as it is standalone and not responsible for a specific input value.

**Dataset**

We can also use previously prepared weights to make predictions for this dataset. Putting this all together we can test **our predict()** function below.

**Training Network Weights**

We can estimate the weight values for our training data using stochastic gradient descent. Stochastic gradient descent requires two parameters:

**Learning Rate:** Used to limit the amount each weight is corrected each time it is updated.

**Epochs:** The number of times to run through the training data while updating the weight. These, along with the training data will be the arguments to the function.

There are 3 loops we need to perform in the function:

1. Loop over each epoch.

2. Loop over each row in the training data for an epoch.

3. Loop over each weight and update it for a row in an epoch.

As you can see, we update each weight for each row in the training data, each epoch.

Weights are updated based on the error the model made. The error is calculated as the difference between the expected output value and the prediction made with the candidate weights.

There is one weight for each input attribute, and these are updated in a consistent way, for example:

w(t+1) = w(t) + learning\_rate \* (expected(t) - predicted(t)) \* x(t)

The bias is updated in a similar way, except without an input as it is not associated with a specific input value:

bias(t+1) = bias(t) + learning\_rate \* (expected(t) - predicted(t))

Now we can put all of this together. Below is a function named train\_weights () that calculates weight values for a training dataset using stochastic gradient descent.

You can see that we also keep track of the sum of the squared error (a positive value) each epoch so that we can print out a nice message each outer loop.

We can test this function on the same small contrived **dataset** from above.

You can see how the problem is learned very quickly by the algorithm. Now, let’s apply this algorithm on a real **dataset.**

**Implementation**

**Read dataset**

Read the dataset. Make sure dataset labeled column is binary classified. Then split the dataset (train test split).

Use the numerical Columns of dataset as features.

Use a function **dataset()** for reading and splitting.

**Building a Neural Network**

1. Use adjacency matrix for graph representation.
2. Instantiate the class. Add an input layer, specify the number of neurons (size).
3. Then add hidden layers (standard), specify the number of neurons (size=5 neurons) and an activation function (sigmoid).
4. Finally, add an output layer, its size (1 output value) and an activation function (sigmoid).

Use a class **NeuralNetwork** that contain model and composed of multiple layers.

Make a function **autogenerate()** so that layers are fully connected, and it also randomizes all the weights. **neuron()** function has a parent layer. A neuron accumulates the output of the edges connected to it (\_accumulated), it outputs that input to its edges after processing it with its \_activation\_function (which you can change on a per-neuron basis). A neuron knows its \_next and \_previous edges. An Edge knows its next neuron (\_n), the neuron its coming from (\_nb), it has a weight (\_w).

**Predicting a value**

The inputs are set up in the \_accumulated variable of the input neurons, then trigger() calls each layer to fire successively, it computes the output of that layer and put it in the input of the neurons in the next layer. Finally the output() just returns the activation\_function(accumulated) use **step function** for each neuron of the last/output layer.

Use

**Task**

1. Download the dataset from classroom->project 2folder.

2. Remove the categorical/textual columns(if any)

4. Apply the Perceptron algorithm from scratch using above code snippets?

5. Apply on above **dataset**

Following are the important term you must understands before starting the project.