Diagnosis, Treatment, and Imaging Analysis Performed by Machine Learning Approaches

Emilio Daza

July 7, 2024

1 Methodology

Neural Network for Diagnosis

The model is composed of **5** layers and **113** neurons in total (post input layer). The number of layers and neurons is totally adjustable by the user to maximize accuracy if the ones already present are insufficient for a different, probably larger, dataset:

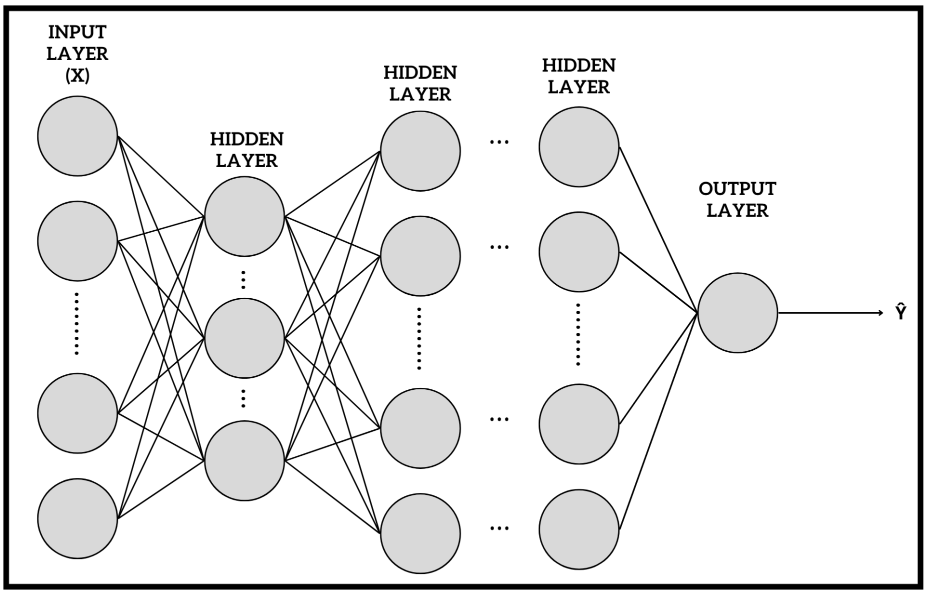

Figure 1: Structure behind Supervised-Learning based Neural Network

Each node in the input layer contains a distinct feature. The total amount of features considered is 13 using the Heart Disease database offered by the University of California, Irvine.

Feature Listing

- Age
- Sex
- cp: Chest pain type
  - Value 1: Typical Angina
  - Value 2: Atypical Angina
  - Value 3: Non-anginal pain
  - Value 4: Asymptomatic
- trestbps: Resting blood pressure in mm Hg on admission to the hospital
- fbs: (Fasting blood sugar $>$ 120 mg/dl) (1 = true; 0 = false)
- restecg: Resting electrocardiographic results:
  - Value 0: Normal
  - Value 1: Having ST-T wave abnormality (T wave inversions and/or ST elevation or depression of $>$ 0.05 mV)
  - Value 2: Showing probable or definite left ventricular hypertrophy by Estes' criteria
- thalach: Maximum heart rate achieved
- exang: Exercise induced angina (1 = yes; 0 = no)
- oldpeak: ST depression induced by exercise relative to rest
- slope: The slope of the peak exercise ST segment
  - Value 1: Upsloping
  - Value 2: Flat
  - Value 3: Downsloping

- ca: Number of major vessels $(0-3)$ colored by fluoroscopy
- thal: 3 = normal; 6 = fixed defect; 7 = reversible defect

And the target considered is the presence $= 1$ or absence $= 0$ of heart disease in the variable "num".

Input Layer Interpretation

To examplify what occurs in the nodes in the input layer, in this dataframe, each row is a sample that contains 13 elements/features. Consider three samples as vectors:

$$\begin{aligned} s_1 &= \langle v_{1,1}, v_{1,2}, \ldots, v_{1,13} \rangle \\ s_2 &= \langle v_{2,1}, v_{2,2}, \ldots, v_{2,13} \rangle \\ s_3 &= \langle v_{3,1}, v_{3,2}, \ldots, v_{3,13} \rangle \end{aligned} \tag{1}$$

One node will be composed of $x_1 = \langle v_{1,1}, v_{2,1}, v_{3,1} \rangle$, the next one of $x_2 = \langle v_{1,2}, v_{2,2}, v_{3,2} \rangle$, and so on for a generalized case with more samples. This means that each node contains one feature collected from the given samples.

Data Processing

All the samples with missing values have been dropped from the analysis, the samples have been shuffled randomly, and 80% of the data (samples) has been assigned to training $(X_{\text{train}}, y_{\text{train}})$ and 20% to testing $(X_{\text{test}}, y_{\text{test}})$, where X is the set of input features and y is the set of target values. After, all the input values were normalized. The process ocurred by dividing all the values from a given feature by the maximum number found on the same feature.

Then, standarization was applied to ensure that all features contribute equally to the model which is defined by

$$t = \frac{x - \mu}{\sigma} \tag{2}$$

where x: original input value, μ: mean of input values, σ: Standard deviation of the feature values, t: Standarized value such that n is the number of values in X_{train} and

$$\mu = \frac{1}{n} \sum_{i=1}^{n} x_i \tag{3}$$

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2} \tag{4}$$

On the other hand, X_{test} is standarized by using the mean and standard deviation of X_{train} to guarantee the model evaluates new data in the same way than when evaluating training data which avoids overfitting.

Forward Propagation

The way to compute the values of the neurons in the first layer (the one after the input layer) is the following:

$$\begin{bmatrix} z_{1,1} & \cdots & z_{1,13} \\ z_{2,1} & \cdots & z_{2,13} \\ \vdots & & \vdots \\ z_{11,1} & \cdots & z_{11,13} \end{bmatrix} = \begin{bmatrix} w_{1,1} & \cdots & w_{1,297} \\ w_{2,1} & \cdots & w_{2,297} \\ \vdots & & \vdots \\ w_{11,1} & \cdots & w_{11,297} \end{bmatrix} \begin{bmatrix} x_{1,1} & \cdots & x_{1,13} \\ x_{2,1} & \cdots & x_{2,13} \\ \vdots & & \vdots \\ x_{297,1} & \cdots & x_{297,13} \end{bmatrix} + \begin{bmatrix} b_{1,1} & \cdots & b_{1,13} \\ b_{2,1} & \cdots & b_{2,13} \\ \vdots & & \vdots \\ b_{11,1} & \cdots & b_{11,13} \end{bmatrix}$$

Which can be re-writed as

$$Z^{[1]} = W^{[1]}X^{[0]} + B^{[1]} := Layer \quad 1 \tag{5}$$

Such that each row in Z represents the information contained in a neuron of the first layer. The new matrix Z becomes the input for an activation function, which in this case is the Rectified Linear Unit (ReLU). The activation function is necessary because it allows us to approximate non-linear functions, which implies that ReLU has to be non-linear, and it is defined in the following manner:

$$ReLU(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$$

Now all the values in Z are input in ReLU which defines a new matrix:

$$A^{[1]} = ReLU(Z^{[1]}) \tag{6}$$

To get the values of the second layer we use $A^{[1]}$ as its input in the same way that it was done before:

$$Z^{[2]} = W^{[2]} A^{[1]} + B^{[2]} := Layer \quad 2 \tag{7}$$

Where, as before, the number of rows in Z is the number of neurons in that layer, and its number of columns is the number of features used in the model which is 13. The shape of $A^{[1]}$ is the same as that of $Z^{[2]}$. The shape of W is determined equally as before: the number of rows is the number of neurons in the new layer and its number of columns is the amount of rows of the matrix it is multiplied to ($A^{[1]}$). Ultimately, the shape of $B^{[1]}$ is one such that the number of rows is equal to the number of neurons in the new layer and its number of columns is equal to the number of columns of the "input matrix" which previously was $X^{[0]}$ but now is $A^{[1]}$.

The mentioned process is iterated for the rest of the layers until the output layer which consists of only one neuron. Which means that the equation for the hidden layer would be

$$Output = W^{[n]} \cdot A^{[n-1]} + B^{[n]} = \langle w_1, w_2, \ldots, w_{13} \rangle \cdot \langle a_1, a_2, \ldots, a_{13} \rangle + b \tag{8}$$

Which implies that

$$Output = w_1 a_1 + \cdots + w_{13} a_{13} + b \tag{9}$$

At that point a different activation function is used which is called Sigmoid. This function is defined in the following manner:

$$Sigmoid\,(z) = \frac{1}{1 + e^{-z}}$$

Which is incredibly useful because $\text{Sigmoid}\,(x) \in (0,1) \quad \forall x \in (-\infty, \infty)$ which means that we can express the scalar obtained in the output layer in a value between 0 and 1. The latter provides a probabilistic interpretation of the results, the value obtained from Sigmoid can be rounded to 1 or 0 meaning that y predicted would be acquired after that process. Remember that y is our target which is presence or absence of heart disease. Therefore,

$$\hat{Y} = \begin{cases} 1 & \text{if Sigmoid}(z) \geq 0.5 \\ 0 & \text{if Sigmoid}(z) < 0.5 \end{cases}$$

Important remark: All the weights (w) and biases (b) are initially picked randomly but are later adjusted properly in a process called Backpropagation to improve predictions

Loss Function

We need to measure the quality of the model so that, afterwards, parameters could be readjusted to find the desired results during the training stage in supervised learning. The function selected to determine accuracy is the Binary Cross Entropy Loss Function (BCE) which is calculated in the following manner

$$BCE = -\frac{1}{N}\sum_{i=0}^{N}[y_i log(\hat{y}_i) + (1-y_i)log(1-\hat{y}_i)] \tag{10}$$

Where y_i is the actual target (0 or 1), $\hat{y}_i$ is the predicted probability of the target (not yet rounded), and N is the number of samples.

It is useful because it is a binary classifier and the model has as targets 0 or 1, and penalizes confident but incorrect predictions.

Since the numeric values inside of the logarithms are between 0 and 1 the logarithms are negative or 0. Therefore, the negative sign that affects the whole sum serves to make BCE positive.

Backpropagation

The idea is to minize the loss function. That is to minimize inaccuracies in the model. To do this we have to remember that we selected random values as weights and biases but these could have been just defined in terms of variables. If we apply all the process and do not select any set of numbers for weights and biases ultimately, we would have the BCE expressed in terms of these unknowns too.

Stochastic Gradient Descend

We need to determine the "level of influence" that the weights and intercepts have on the Loss Function. And we can do it by Partial Derivatives and the Chain Rule. Consider that W and B are matrices of the containing $w_1, w_2, ...$ and $b_1, b_2, ...$ respectively but in this case as variables not numerical values.

$$\frac{\partial BCE}{\partial W} = \frac{\partial BCE}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial W} \tag{11}$$

$$\frac{\partial BCE}{\partial B} = \frac{\partial BCE}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial B} \tag{12}$$

Consider that now $\hat{y}$ is the function of the model without being rounded.

Now, after acquiring

$$\frac{\partial BCE}{\partial W} \tag{13}$$

and

$$\frac{\partial BCE}{\partial B} \tag{14}$$

We can plug in each weight and bias that was initially selected randomly into these functions. We would have

$$\frac{\partial BCE}{\partial W}(W^{[1]}) \quad \wedge \quad \frac{\partial BCE}{\partial B}(B^{[1]}) \tag{15}$$

Now we can finally update our parameters in order to get a better prediction. We can begin minimizing the Loss Function by getting the derivative close to 0. So we can "move" in the loss function to get closer to the global or local minimum by steps (remember that the matrix X has fixed values). This step size is called in Machine Learning lingo as learning rate.

Therefore, we can define step sizes in the following manner:

$$Step\ Size_W = \alpha\ \frac{\partial BCE}{\partial W}(W^{[1]}) \tag{16}$$

$$Step\ Size_B = \alpha\ \frac{\partial BCE}{\partial B}(B^{[1]}) \tag{17}$$

And now we can update the original random parameters selected in $W^{[1]}$ and $B^{[1]}$ in the following way

$$New\ W^{[1]} = W^{[1]} - Step\ Size_W \tag{18}$$

$$New\ B^{[1]} = B^{[1]} - Step\ Size_B \tag{19}$$

And with this we are done with one epoch (interation which begins with input weights and biases and is concluded with the re-update of weights and biases). We model can go through thousands of epochs without problems and have a much more perfect prediction of targets.

But consider that this process (Gradient Descend) is slow for big data. So we use Stochastic Gradient Descend (SGD) which accelerates the process. SGD uses the name algorithm that was described but uses the points that determine the Loss Function differently (the Loss Function can be graphed in a 3D Cartesian Plane with axes "weight", "bias" and BCE). That is, it updates the weights and biases using only one randomly selected data point at a time and introduces noise into the parameters updates which lead to higher variance in the updates.

It is reasonable to ask ourselves: Until when does Stochastic Gradient Descend stop? and the answer is until the number of epochs precised to the model is completed.

Important Remark: In this neural network each layer has its own random weights and biases at the beginning (that's why they are expressed in distinct matrices). So, per epoch not just the first weights and biases get updated but all of them in the different layers.

Code

Listing 1: Cardiovascular Diseases Diagnosis Predictor label

```
import numpy as np
import pandas as pd
import math
from torch import nn
import torch.nn as nn
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
from ucimlrepo import fetch_ucirepo, list_available_datasets
import torch

# fetch dataset
```

```
heart_disease = fetch_ucirepo(id=45)

# data (as pandas dataframes and then numpy arrays)
X = heart_disease.data.features
y = heart_disease.data.targets

X = X.to_numpy()
y = y.to_numpy()

y = y[~np.isnan(X).any(axis = 1)]
X = X[~np.isnan(X).any(axis = 1)]
X = X/X.max(axis = 0)

y[y != 0] = 1

n_samples, n_features = X.shape

# Data Preparation and Data Conversion to Arrays

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2,
                                                     random_state = 1234)

# Standarization
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)

X_train = torch.from_numpy(X_train.astype(np.float32))
X_test = torch.from_numpy(X_test.astype(np.float32))
y_train = torch.from_numpy(y_train.astype(np.float32))
y_test = torch.from_numpy(y_test.astype(np.float32))
```

```
y_train = y_train.view(y_train.shape[0], 1)
y_test = y_test.view(y_test.shape[0], 1)

# Model of 5 Layers
class NeuralNetwork(nn.Module):

    def __init__(self, n_input_features):
        super(NeuralNetwork, self).__init__()
        self.layer1 = nn.Linear(n_input_features, 64)
        self.layer2 = nn.Linear(64,32)
        self.layer3 = nn.Linear(32,16)
        self.output = nn.Linear(16,1)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.relu(self.layer1(x))
        x = self.relu(self.layer2(x))
        x = self.relu(self.layer3(x))
        x = torch.sigmoid(self.output(x))
        return x

model = NeuralNetwork(n_features)

learning_rate = 0.01
criterion = nn.BCELoss()
optimizer = torch.optim.SGD(model.parameters(), lr = learning_rate)

num_epochs = 1000

for epoch in range(num_epochs):
    y_predicted = model(X_train)
    loss = criterion(y_predicted, y_train)
```

```
        loss.backward()

        optimizer.step()

        optimizer.zero_grad()

        if (epoch + 1) % 20 == 0:
            print(f'epoch: {epoch + 1}, loss = {loss.item():.4f}')

with torch.no_grad():
    y_predicted = model(X_test)
    y_predicted_cls = y_predicted.round()
    acc = y_predicted_cls.eq(y_test).sum()/float(y_test.shape[0])
    print(f'accuracy = {acc:.4f}')
    print(y_predicted)
    print(y_predicted_cls)
```

Neural Network for Treatment

K-Nearest Neighbours Algorithm for Imaging Interpretation

To analyze any set of pictures derived from medical imaging consider a 3D graph with axes Red, Green, and Blue (RGB).

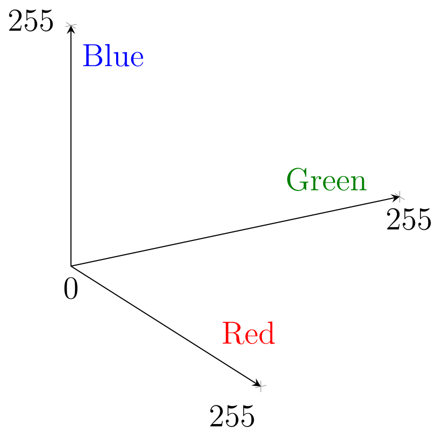

Figure 2: 3D Cartesian Coordinate System with RGB-Labeled Axes

Each pixel in an image is going to contain a 3-tuple that represents it in the form of RGB with numbers ranging from 0 to 255. Therefore, a RGB coordinate can be expressed in the presented graph as (red value, green value, blue value). Each picture is going to have a set of pixels that have their respective coordinates. That will generate many clusters of points such that each cluster represents an image. Let's image, we have three images and each dot in the next graph represents the RGB code of a pixel:

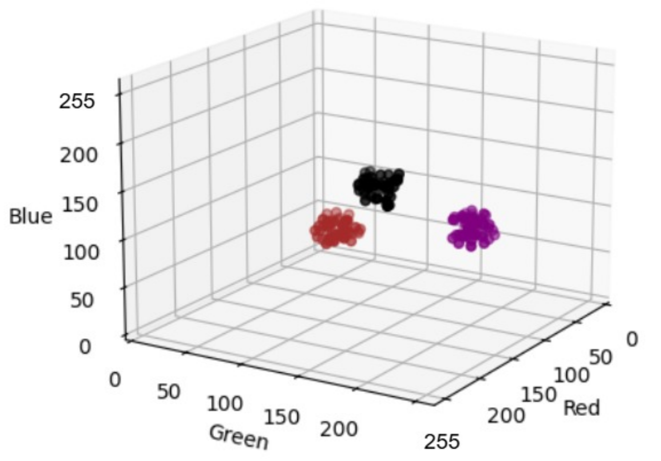

Figure 3: Presence of Clusters to Illustrate Imaging Graphical Interpretation

Important Remark: The colors in the dots (black, purple, and orange) are just a way to emphasize the distinctiveness among clusters. They do not have any additional interpretational value.

Considering Fig. 3, where all the dots already present were part of the training dataset, we would like to provide a new and unseen dot to see how the model predicts to which cluster is belongs.

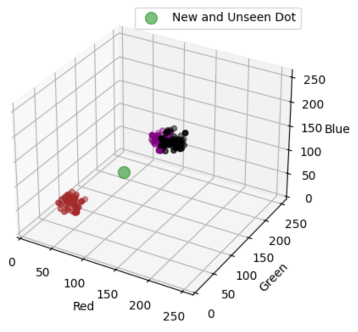

Figure 4: Addition of unseen datapoint

What the K-Nearest Neighbours algorithm does it that it measures the distance of this unseen datapoint with respect to the other points in the training dataset, observes the “k” points (which, by the way, are labelled with respect to the cluster they belong to) that are the the closest to the new datapoint, and performs a majority vote. That is, if for example the majority of the “k” points that are the closest to the new datapoint are labelled as “Cluster 1”, then the new datapoint gets classified as belonging to “Cluster 1”. Note that $k \in \mathbb{N}$ that could be used for instance as “The 4 points that are the closest to the new data point”, where $k = 4$

It is important to note that the distance between points will be calculated using the euclidian distance formula in the 3D Cartesian Plane which is the following:

$$D = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2} \tag{20}$$

Where x stands for the value of red, y for that of green, and finally z for that of blue.

This base model is very useful when we have sets of images that have very distinct colors, however, in the case of fMRI, for example, there are many times where similar colors are expressed but their combination as a whole is what can generate a sudden and important change in diagnosis.

To account for the situation of similar color usage but relevance in differences in overall view, the following approach is proposed.

The new graph to consider would be an n-dimensional one, that means that each coordinate could belong to $\mathbb{R}^n$ such that n is a multiple of 3. Each datapoint will be expressed as a datapoint in the following manner:

$$(Rv_1, Gv_1, Bv1, Rv_2, Gv_2, Bv2, \ldots, Rv_p, Gv_p, Bvp) \tag{21}$$

Where p stands for the number of pixels in the image. The way pixels are numbered is from left to right from top to bottom.

Through this approach, each image in the training dataset would contain all the color information about all their pixeles in ordered manner generating a more precise learning outcome for the model.

The euclidian formula of distance now applied to an n dimensional space would be the following:

$$\sqrt{(u_1 - v_1)^2 + (u_2 - v_2)^2 + \cdots + (u_n - v_n)^2} \tag{22}$$

The mechanism of the K-Nearest Neighbours by considering clusters and majority vote would be the same but now applied to images instead of individual pixels to take into consideration the grand scheme of details.