Predict diabetes using Multilayer Perceptron

Jordana Izquierdo
Deep Learning Fundamental-Assigment 1
Faculty of Sciences, Engineering and Technology
University of Adelaide
Adelaide, Australia

I. Introduction and Background

The paper describes and provides a detailed account of the implementation of a Multilayer Perceptron (MLP) using Keras to determine whether a patient has diabetes.

MLP is an artificial neural network (ANN) architecture developed as a solution to the limitations of the perceptron, which represents the simplest ANN architecture [1, p.305]. An MLP typically comprises an input layer, one or more hidden layers, and an output layer [1, p.309]. When an MLP incorporates a deep stack of hidden layers, it is referred to as a deep neural network (DNN) [1, p.309].

The task at hand involves experimenting with different combinations of optimizer algorithms (SGD, Adam, Nadam, RMSprop), learning rates (0.1, 0.01, 0.001), and the number of epochs (8, 32, 100, 120), with the goal of identifying the configuration that yields the highest diagnostic accuracy. Through this process, we aim to enhance the MLP's ability to accurately identify diabetic conditions.

II. Method description

The methodology applied is the following:

A. Data Cleaning

The dataset was sourced from https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary/diabetes_scale. It comprises 768 records, 8 features with values scaled between -1 and $+1$, and a target column initially labeled as $+1$ (indicative of diabetes) and -1 (indicative of no diabetes). Some data cleaning steps were necessary to ensure the following:

- The target values were modified to represent 0 (negative for diabetes) in place of -1. This transformation was important because the sigmoid activation function in the output layer, combined with the binary cross-entropy loss function, expects labels to be in the 0 and 1 format.
- The feature values were correctly formatted and of the appropriate type. Additional measures were taken to format the values correctly and convert them to the float type.
- Initially, some records had missing values, which were subsequently removed.

B. Exploratory Data Analysis

- The dataset shows a significant class imbalance between the categories (0 and 1). As shown in Figure 1, the category 0 represents the 34.65% of the dataset with 262 records while the category $+1$ represents the 65.35% of the dataset with 496 records.

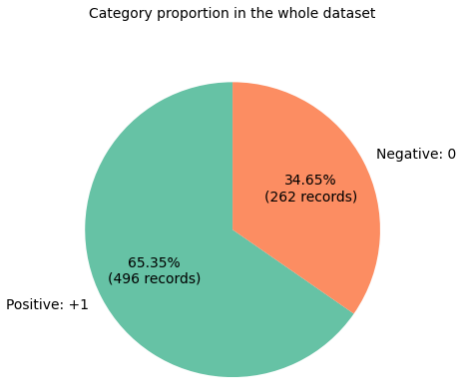

Fig. 1. Category proportion in the whole dataset

- Exploratory data analysis (EDA) plays a pivotal role as it provides insights into the data distribution, potential outliers, and the scales of the data. As illustrated in Figure 2, the dataset is scaled within the range of -1 to $+1$. The scale of the input features holds significance since weight initialization techniques and activation functions in Deep Neural Networks (DNNs) are often tailored based on certain assumptions about the distribution of input features. For this specific dataset, the initialization might align well with the hyperbolic tangent (tanh) function, given its compatibility with the data's range. While it's not mandatory for features to adhere to a normal distribution, a distribution proximate to normal can enhance training stability. Addressing outliers, this dataset does possess a few. These outliers can lead DNNs towards overfitting. As a countermeasure, outliers were identified, eliminated, and subsequently imputed using the median.

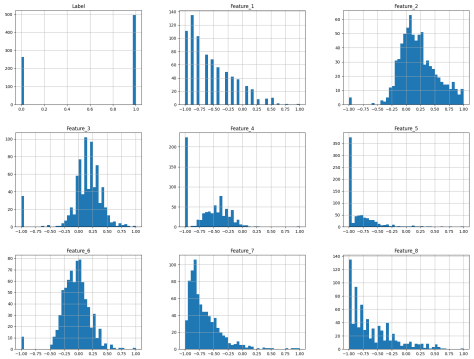

Fig. 2. Feature distribution

- As shown in Figure 3, each histogram represents the distribution of a feature, distinguishing between the different classes. This provides a visual representation of how well-separated the classes are based on each individual feature. Features that have more overlap in the histograms for different classes might be less informative for class separability.

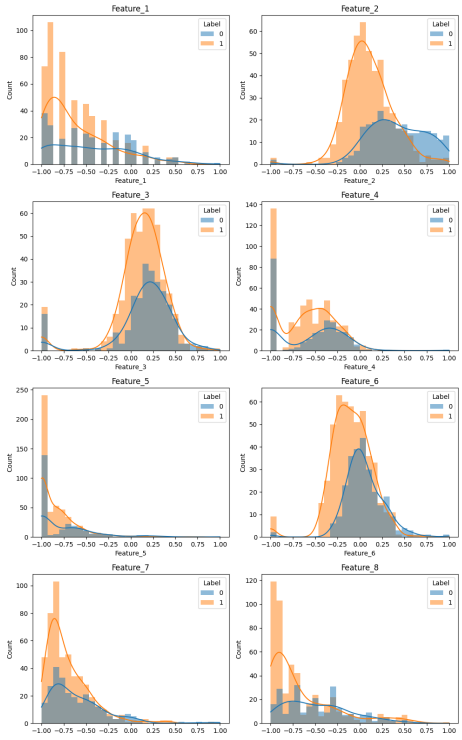

Fig. 3. Separability ability of the features

C. MLP Structure

An MLP consists of an input layer, one or more hidden layers, and an output layer. Every node in a layer is connected to every node in the subsequent layer. See III. METHOD IMPLEMENTATION section for more details.

D. Testing

Once the MLP structure is implemented, the accuracy of the model is tested to measure its generalization ability. See V. EXPERIMENTS AND ANALYSIS for more details.

III. Method Implementation

A Multi-Layer Perceptron (MLP) is an artificial neural network. It consists of layers of nodes. Every node in a layer is connected to every node in the subsequent layer. The structure was created using Keras and applying a sequential API.Creating a sequential API using keras means adding the layers one by one sequentially [1, p.319].

A. Creating the model using sequential API

- Input layer: A random seed is included to assure the results are reproducible. It also considers a vector of 8 columns. Finally, the layer contains 256 neurons.

```
model = keras.models.Sequential(tf.random.set_seed(42))
model.add(keras.layers.Dense(256, activation='tanh',
                             input_shape=(8,)))
```

- Hidden layers:There are three hidden layers with varying node counts, all utilizing the tanh activation function which naturally produces outputs in the range of (-1 to +1). By decreasing the number of nodes in subsequent layers, the network is compelled to learn a compressed representation of the input data. This reduction also decreases the overall number of parameters in the network, potentially helping to prevent overfitting. Additionally, the inclusion of dropout serves as a regularization technique to further prevent overfitting.

```
del.add(keras.layers.Dropout(0.5))
model.add(keras.layers.BatchNormalization())
model.add(keras.layers.Dense(128, activation='tanh'))
model.add(keras.layers.Dropout(0.5))
model.add(keras.layers.BatchNormalization())
model.add(keras.layers.Dense(64, activation='tanh'))
model.add(keras.layers.Dropout(0.5))
model.add(keras.layers.BatchNormalization())
model.add(keras.layers.Dense(32, activation='tanh'))
model.add(keras.layers.BatchNormalization())
model.add(keras.layers.Dense(16, activation='tanh'))
model.add(keras.layers.BatchNormalization())
model.add(keras.layers.Dense(8, activation='tanh'))
model.add(keras.layers.BatchNormalization())
model.add(keras.layers.Dense(1, activation='sigmoid'))
```

- Output layer:A single node using the sigmoid activation function, producing a probability indicating the likelihood of an instance belonging to class +1, suitable for binary classification.

```
model.add(keras.layers.Dense(1, activation='sigmoid'))
```

B. Compiling the model

After the MLP architecture creation, the compile() method is invoked to designate both the loss function and the optimizer.

- Loss Function: We utilize Binary Crossentropy, which is well-suited for binary classification challenges. It quantifies the discrepancy between the predicted probabilities and the true labels.

- Optimizer: The model employs the Adam optimizer, which showed the best performance among Stochastic Gradient Descent (SGD), Nadam, and RMSprop.
- Learning Rate: The model employs 0.01 as learning rate , which performs best between 0.1, and 0.001.

```
model.compile(optimizer=keras.optimizers.SGS(0.01),
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

C. Training and Evaluating the model

The fit() method returns a history object, which contains details about how the model's loss and accuracy changed over epochs, both on training and validation data. This method can set:

- The number of epochs indicates how many times the model will iterate over the entire dataset. Each iteration is an opportunity for the model to adjust its weights based on the errors it made in the previous iteration.
- The batch size means that the model will update its weights after every 32 training examples.

The inclusion of callbacks such as early stopping, which is detailed in the next section.

```
history = model.fit(X, y,
                    epochs=32,
                    batch_size=32,
                    validation_data=(X_val, y_val),
                    callbacks=[early_stopping])
```

After training, the mode is assessing the performance on the validation set with the evaluate() method to understand how well it is performing on unseen data.The function returns the loss and accuracy of the model on the validation data. In this code, these values are captured in the loss and accuracy variables.

```
loss, accuracy = model.evaluate(X_val, y_val)
```

D. Using Callbacks

Early stopping is a technique used during the training of a machine learning model to prevent overfitting and improve generalization performance [1, p.394]. It involves monitoring a validation metric (e.g., validation loss or accuracy) and stopping the training process when the validation metric stops improving or starts deteriorating [1, p.394]. Given that we're training for many epochs (120), adding early stopping can prevent overfitting and save the best model based on validation accuracy.

```
early_stopping=EarlyStopping(monitor='val_accuracy',
    patience=10, restore_best_weights=True)
```

IV. Code Github link

The entire code can be found in the following link: https://github.com/Joizra/Multilayer-Perceptron

V. Experiments and analysis

A. Description of experiments and the aims of the design

An MLP (Multi-Layer Perceptron) has several parameters that can be adjusted to enhance the model's performance. These include the number of layers, the number of neurons in each layer, the choice of optimizer, the learning rate, and the batch size. To achieve better prediction results regarding whether a patient has cancer, the following strategies were employed:

- Various combinations of optimizer algorithms (SGD, Adam, Nadam, RMSprop), learning rates (0.1, 0.01, 0.001), and number of epochs (8, 32, 100, 120) were experimented with, resulting in 48 different model configurations. The model yielding the highest accuracy, 81.96%, utilized SGD as the optimizer algorithm, a learning rate of 0.001, and 32 epochs.
- As illustrated in Figure 4, both the performance of the loss function and the accuracy of the model on the training and validation sets appear stable.

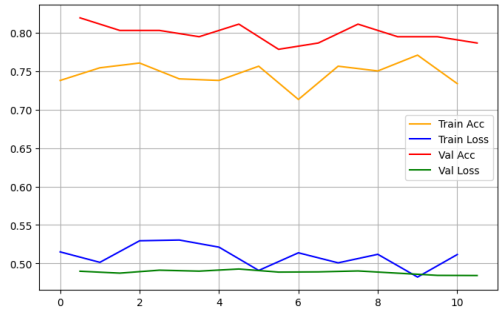

Fig. 4. Model performance

- Increasing the number of layers led to a decrease in the model's performance.
- Decreasing the number of neurons in each successive layer improved the model's performance.
- Initially, the dataset labeled +1 as indicating diabetes and -1 as negative for diabetes. The -1 label was changed to 0. This modification improved the model's accuracy, as the sigmoid activation function in the output layer, when paired with the binary cross-entropy loss function, expects labels in the 0 and 1 format.

B. Results analysis

- As shown in Table 1, The model yielding the highest accuracy, 81.96%, utilized SGD as the optimizer algorithm, a learning rate of 0.001, and 32 epochs.

Parameter	Result
Optimizer	SGD
Learning rates	0.001
Number of epochs	32

TABLE I
FINAL MODEL PARAMETERS

- The model's evaluation was carried out using various metrics, including Accuracy, F1 Score, Recall, and Precision. As shown in Table 2, the model has an accuracy rate of approximately 77.63%, implying that in a set of 100 predictions, the model accurately detects whether a patient has diabetes about 77 times. The F1 Score, while satisfactory, suggests there's potential for enhancement. Moreover, the Recall value of 81.82% demonstrates that the model is able to correctly identify a significant majority of actual positive diabetes cases. On the other hand, the Precision value of 82.65% indicates that out of all the cases the model predicts as positive for diabetes, around 82.65% are true positive cases. This underscores that the model maintains a balance between its sensitivity (Recall) and its ability to avoid false positives (Precision).

Metric	Result
Accuracy	77.63%
F1 Score	82.23%
Recall	81.82%
Precision	82.65%

TABLE II
MODEL EVALUATION METRICS

Further fine-tuning of the neural network's hyperparameters is necessary to attain higher metric values.

VI. REFLEXION ON PROJECT

A. Summary of major project design choices

- Originally, the dataset target considered $+1$ as positive for diabetes and -1 as negative for diabetes. The -1 label was changed to 0. That change results in an increment of the model accuracy as the sigmoid activation function in the output layer paired with the binary crossentropy loss function expects labels in the 0 and 1 format.
- The dataset exhibits a pronounced class imbalance between the categories (0 and 1). This imbalance can influence the model's performance.
- Given the dataset's size, the network can be prone to overfitting. To counteract this, consider implementing regularization techniques like dropout, L1 or L2 regularization, or even simplifying the network architecture.
- The low T-score values suggest that the features might possess limited discriminative power between the classes.

B. Ideas for future work based on experimental evidence

- Other algorithms, such as Support Vector Machines (SVM) or Logistic Regression, might yield better performance for this binary classification task.
- It's advisable to have a more robust dataset since 768 records may be insufficient for training a deep neural network (DNN) effectively.
- It's also crucial to ensure both classes in the dataset (0 and 1) are represented in equal proportions to achieve a balanced training across both classes. Applying techniques to handle imbalance can lead to better overall results.

REFERENCES

[1] A. Géron, Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems. O'Reilly Media, 2019.