CMPS 261:Machine Learning Project

Project Report for the Higgs Boson Classification done by The ArtificialMinds

PREPARED FOR

Dr. Rida Assaf

PREPARED BY

Ayla Hmadi

Joelle El Homsi

Bucker Yahfoufi

1. Project Overview

This project aims to use machine learning techniques to classify particle collisions as either HIGGS or background in high-energy physics. The goal is to identify the rare occurrences of HIGGS particles from the vast majority of non-exotic particles produced by high-energy particle colliders. The dataset consists of 600,000 training examples in the form of a CSV file, each with 28 features. We will perform exploratory data analysis, preprocess the data, and use various machine learning models to classify the particle collisions. We will compare the performance of the models using appropriate evaluation metrics and optimise their hyperparameters using cross-validation and grid search.

2. Process

First of all, we created a python notebook to work with it.

Step 1, importing all the required libraries:

1. **Pandas**, for reading, cleaning and transforming data
2. **Numpy**, to make our work with the dataset easier through arrays and matrices
3. **Sklearn**, for data processing, feature selection and modelling
4. **TensorFlow**, for developing and training deep learning models
5. **StandardScaler**, for standardising the data
6. **“train_test_split” “GridSearchCV” “KFols” “cross_val_score”**, for splitting the data into training and testing sets, performing cross-validation, and tuning model hyperparameters
7. **“Sequential” “Model”**, imported from Keras models for building and training deep learning models. Sequential is a linear stack of layers, Model class allows for more complex architectures
8. **“Dense” “LeakyReLu” “Dropout” “BatchNormalization” “ Input”**, different types of layers to be added to the Keras model
9. **“Adam” “Nadam”**, optimization algorithms that can be used to update the model weights during training
10. **KerasClassifier**, a wrapper class ro allow Keras models to be used in scikit-learn pipelines

11. "**L2**", to prevent overfitting by adding a penalty term to the loss function
12. **Drive**, ro mount the the google drive and access the dataset
13. **Load_model**, for loading the saved model

Step 2, Data preparation:

1. We first connect to google drive in order to access the dataset. We used the **drive.mount()** from **google.colab** library to mount the google drive to our Colab notebook. This will give us access to our drive from the notebook.
2. Now we need to load the dataset. We load the dataset into a Pandas dataframe. We save the path of the dataset inside a variable called **path** and we load it using the **pd.read_csv()** function and store it in a variable named **data**.
3. We then named the columns. This is to make the work easier when dealing with the data.
4. Lastly we check for null values. We use the **isna().sum()** function. The output shows that there are null values in the **jet_3_b-tag** column.To deal with this, we first converted the **jet_1_phi** and **jet_4_b-tag** columns to numeric data types using **pd.to_numeric()** function with the **errors=coerce** parameter to convert any non numeric to NaN. We also convert the **class_label** column to integer data using the **astype()** function. We finally drop any rows with null values using the **dropna()** function and remove any duplicates using the **drop_dublicates()** function.

Step 3, data scaling:

1. Firstly, we split the data into features and labels. Features are the input variables or columns that are used to predict the target variable, while labels are the output variable or the columns to be predicted. We use the **iloc[,]** function to do so.
2. Second is feature scaling. After splitting the data, we apply feature scaling to the features to standardise their values. We first create an instance of the **StandardScaler()** class. The **fit_tranform()** method is then used to scale the features. The resulting scaled features are stored in a dataframe with the same column names as before. This step is very important because it helps to prevent the model from giving too much importance to features with large values

Step 4, creating a deep learning model architecture using Keras with TensorFlow backend. Here are the steps of creating this model:

1. First, we define the model input layer. The input layer is defined to have as many neurons as there are features in the data:

```
Inputs = Input(shape=(X.shape[1],))
```

2. We then add the hidden layers. These contain the neurons that perform computations on the input data. The number of hidden layers and neurons in our case are 4 dense layers with 1024, 512, 256, and 128 neurons respectively. Each dense layer applies a linear transformation to the previous layer's output, and an activation function is applied to the output of the dense layer to introduce non-linearity to the model.

```
x = Dense(1024, activation=activation)(inputs)
x = BatchNormalization()(x)
x = Dropout(0.5)(x)
x = Dense(512, activation=activation)(x)
x = BatchNormalization()(x)
x = Dropout(0.4)(x)
x = Dense(256, activation=activation)(x)
x = BatchNormalization()(x)
x = Dropout(0.3)(x)
x = Dense(128, activation=activation)(x)
x = BatchNormalization()(x)
x = Dropout(0.2)(x)
```

3. Thirdly, we define the output layer. This is the final layer, which produces the predicted label. We have one output neuron, and we set the activation function to sigmoid because it is appropriate for binary classification tasks.

```
Output = Dense(1, activation='sigmoid')(x)
```

4. We then specify the optimizer, loss function and evaluation metric, and then we compile the model to train it.

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

5. We then create a Keras wrapper for the model using KerasClassifier, which allows the model to be used with SKlearn's cross validation and grid search functions.

```
improved_model =
KerasClassifier(build_fn=create_improved_model, epochs=50,
batch_size=256, verbose=1)
```

6. Lastly we train and evaluate the model. The model is trained on the training set using the **.fit()** method. The validation split parameter is used to split the training data into a training set and a validation set. The callbacks parameter is used to specify early stopping and learning rate reduction. We finally evaluate the model on the test set using the **score()** method.

```
history = improved_model.fit(X_train, y_train,
validation_split=0.33, callbacks=[early_stopping,
lr_reduction],     verbose=1)
test_accuracy = improved_model.score(X_test, y_test)
```

3. Output

We tried more than tens of models. We performed hyperparameter tuning and other techniques to improve our model's performance. The best one we got so far got us an accuracy of 75.96% on the testing set, and 76.4% on the training set with validation accuracy of 76.6%

4. Other models

We have different types of models, including:

1. **XgBoost**, we tried XgBoost and got accuracies of 74.2%, 74.5% and 74.8%
2. **GBC,** we got 72.2% accuracy

3. **Other neural networks,** we tried different networks with different parameters and got 75.5%, 75.85% and 75.88%

5. Github

We have the ipython notebook, python code, documentation, presentation and report on our github repository : https://github.com/JJoellee/Artificial-Minds.git