POLITECNICO
MILANO 1863

Study and test a machine learning/neural network library in C++

Advanced Programming for Scientific Computing Project
of
Aurelio Marin Aranzana

Supervisor:
Prof. Luca Formaggia

Programme:
Mathematical Engineering

Department of Mathematics
Politecnico di Milano

Contents

1 Introduction

On the latest year with the improvement of the computational hardware, the influence of the machine learning techniques on the world has been increasing expontentially.

The main example of this fact is the introduction of neural network techniques to important problems of prediction or classification.
For this reason, we are interested in study their implementation on the C++ hardware.

We will investigate the MLpack library, Section 2, which implements a huge amount of machine learning techiniques in clustering, regression and classification.

Moreover, we will compute also neural models using the OpenNN library in C++,Section 3.

We have provided different examples that give a overall idea of the way to work with this models, Section 4.

Moreover,we have compared these library with the well-establishe scikit-learn and tensorflow in Python, Section 4.3.

Finally, we explain how to install and compile successfully these libraries, Section 5.

2 MLpack Library

The library mlpack, [Cur+18], is an open-source software that ease the implementation of the widely spread machine learning techniques in C++.

Projects should be expressed in terms of vector and matrix using the linear algebra library Armadillo,[San16].
Moreover, mlpack is also based on the ensmallen library,[Bha+18], and boost C++ library,[Sch14], in order to perform high mathematical algorithm such as numerical optimization.

A Machine Learning project consist in three parts:

- Data Loading and preprocessing
- Algorithm implementation
- Analysis of the results

2.1 Data Loading and preprocessing

As it was already mentioned before, the data is kept in the program in the form of armadillo's vector and matrix.
However, mlpack provides the class "data" where are collected diverse functions to ease the manipulation of the datasets.

The principal function is data::Load() which reads into matrix or vectors the data from different files. This function provides different possible uses, but we will highlight the following one:

```
bool mlpack::data::Load(const std::string & filename,arma::Mat<eT> & matrix,const bool fatal = false,const bool transpose = true)
```

Where the filename refers to the data file which is going to be store in matrix, fatal determines to consider an error an unsuccesfull loading and transpose to directly transpose the data.
The reason of the addition of the tranpose option is owing to the normal format of the data is row-major when the mlpack requires column-major.
Moreover, it is important to say that in order to work with categorical data is recommended to transform it into number or to use the ARFF format.

Additionally, machine learning problems usually require to split the dataset into training and testing. For this reason, mlpack provides some preprocessing command-line programs to do so, but looking to the source we find that they are based on the following function from the class "data":

```
void mlpack::data::Split ( const arma::Mat< T > & input, const arma::Row< U > & inputLabel, arma::Mat< T > & trainData, arma::Mat< T > & testData, arma::Row< U > & trainLabel, arma::Row< U > & testLabel, const double testRatio )
```

2.2 Algorithm implementation

It implements a wide range of machine learning techniques for regression, clustering and classification.To these kind algorithms,it adds data transformation techniques or some more specific Markov models implementation.

In general, all these methods are configured as a class object with variables and methods needed to define and use the models.

These classes could be normally initiated with the training data obtaining directly the final model. Although, it could be use the method Train() or similar if the data is not provided during the object instance.

Moreover,the constructors of these models have default setting related to the specific policies for each algorithm,but they could be change using pre-defined classes by the mlpack library well at the instance moment or using internal methods associated to eh algorithm model.

Once the models are trained, we are ready to use the method Apply() method or similar to obtain new label or new value from the testing data.

2.3 Analysis of the results

Once the training step is finished, we can store them by the serialize() method of the algorithm class to posteriorly reuse them with the data::Load().
Moreover,the results could be save using the save function provided by the armadillo library where it should be specified the type of file according to this library.

In general,regression algorithm contain the computeError() method to calculated the error committed. However, clustering and classification classes do not provide it being necessary to use the function ConfusionMatrix() of the "data" class to obtain a summary of the preditions.

Additionally,mlpack provides a Timer class to compute the time require by these methods.

3 OpenNN Library

The library OpenNN, [Art],is a package designed to work with neural network models. While the previous software was built under armadillo library, this one is constructed over the well-known Eigen library.

3.1 Data Loading and preprocessing

On contrast to mlpack, OpenNN defines its own DataSet class to store the data and work with.
Normally, there are several constructors for DataSet class which consider the last column as target by default. The most common practice is to use the following constructor:

```
DataSet (const string & data_file_name,const char & separator,const bool & new_has_columns_names))
```

which read from a file where the data is separated by separator and could haver or not header row. Other way, you can always set these by internal methods of the DataSet class.

Once that dataset is loaded, we can perform the preprocessing such as the data splitting into training, validation and test set, it has been added the validation set for neural network which is not required by other machine learning techniques.
We have two options: one random and another one sequential:

```
void split_instances_sequential (const double &training_ratio=0.6, const double &selection_ratio=0.2, const double &testing_ratio=0.2)

void split_instances_random (const double &training_ratio=0.6, const double &selection_ratio=0.2, const double &testing_ratio=0.2)
```

Moreover, It is helpful to scale the data during training to achieve a better performance. Therefore, There is defined a scaler layer which receive a vector with Descriptive type element which store the standard deviation and the maximum, minimum and mean values.

3.2 Algorithm implementation

OpenNN provides two different classes which could be associated with model definition step and the training step.

Firstly, we have the NeuralNetwork class to build our model. Every neural model is defined by a vector of pointers to the Layer class.
The library provides the implementation of the most widely-known layers:

- Perceptron, original layer behind the neural network idea.
- Convolutional, it applies the convolution to images separating the images into filters.
- Pooling, it follows the convolutional layer averageing the results reducing the dimensionality.
- Recurrent, it allows the neural model to recall past values
- Long-Short term, most common recurrent layer for forecasting
- Probabilistic,serves as Activation function for classification problem. For regression problem, this is include into Perceptronlayer class.
- Scaling, Unscaling, Bounding and PrincipalComponent layes for other utilities.

Hence, you can initialize the model with a vector of pointer to layers or adding sequentially with the method add_layer(Layer *).

Besides,you can determine the kind of project that you are developing to initialize with a basic model for this goal. We can select between Approximation (perceptron model with linear activation function), Classification (perceptron with softmax activation layer) and Forecast (LSTM Model).

If the model contains images, there is a constructor where the number of convolutional block and their dimension could be specified.

Secondly, we have the TrainingStrategy class which carry out the training of the neural network. It is initialized with the neural model and the dataset. However, The training algorithm is determined by two sub-classes:

- LossMethod Class which refers to the type of error. The choice would depend on the kind of problem. Normally, squared error are associated to regression problems and the cross entropy to classification.
- OptimizationMethod Class specifies how the weights are going to be update, such as gradient descendent algorithm (SGD) or adaptive moment estimation(Adam).

Once we have chosen these parameters we are able to perform the training phase with the method perform_training() which return a OptimizationAlgorithm::Results objects as a result.

Moreover,the library provided a ModelSelection class to select the number of neurons or the subset of the data that optimize the model. It parts from the TrainingStrategy and considers different methods related to the order selection and input selection.

3.3 Analysis of the results

OpenNN provides the TestingAnalysis with references to the NeuralNetwork and DataSet objects.
It reports a wide range of methods that return the error committed by the model.These error function are grouped into four struct: BinaryClassificationRates, KolmogorovSmirnovResults, LinearRegressionAnalysis and RocAnalysisResults.

Moreover, we have functions for obtaining the errors such as:

- testing_analysis.perform_linear_regression_analysis(),Regression problems
- testing_analysis.calculate_binary_classification_tests(), Binary Classification problems
- testing_analysis.calculate_confusion(), Classification problems

Moreover, all the classes mentioned can use a save() function to keep the information into XML file.However, the library seems to be incomplete an classes like Results have defined the save() function, but no implemented.

4 Test cases

We have found some typical dataset to perform machine learning techniques on https://machinelearningmastery.com/standard-machine-learning-datasets/ , https://machinelearningmastery.com/time-series-datasets-for-machine-learning/ and https://www.analyticsvidhya.com/blog/2018/03/comprehensive-collection-deep-learning-datasets/.

4.1 mlpack

MLpack library algorithm is divided into three main categories: Classification, Clustering and Regression, although there are implemented other methods related with Markov chains.
We are going to perform some of these algorithm to show how the library works:

- On the regression frame, we will perform the Linear Regression and the Least Angle regression,4.1.1.
- From classification, we will show the SVM, 4.1.7, the Logistic regression,4.1.6, the perceptron,4.1.4, and AdaBoost,4.1.5.
- Between the clustering method, we will use the k-means,4.1.3.

Apart from these we have perform the PCA transformation as well.

4.1.1 Linear regression and Least-angle regression

We have implemented on C++ the two algorithm provided by MLPack library.
On one hand, we have the linear regression that estimates the coefficient associated to each parameter to express the output as a linear combination of them minimizing the sum of the squares of the residuals made in the results of every single equation.
On the other hand, we have the least-angle regression (LARS) that fit the linear coefficients iteratively by adding the most correlated predictor with the outcome per each step.
We have studied two problem: Firstly, we have the the Swedish Auto Insurance Dataset which involves predicting the total payment for all claims in thousands of Swedish Kronor, given the total number of claims.
It is comprised of 63 observations with 1 input,number of claims, and 1 output, total payment of all claims in thousands of Swedish Konor.

Secondly,we have performed the multi-variable regreseiion based on the Boston House Price dataset, which involves the prediction of a house price in thousands of dollars given details of the house and its neighborhood.
There are 506 observations with 13 input variables and 1 output variable. The variable names are as follows:

- CRIM: per capita crime rate by town.
- ZN: proportion of residential land zoned for lots over 25,000 sq.ft.
- INDUS: proportion of nonretail business acres per town.
- CHAS: Charles River dummy variable (= 1 if tract bounds river; 0 otherwise).
- NOX: nitric oxides concentration (parts per 10 million).
- RM: average number of rooms per dwelling.
- AGE: proportion of owner-occupied units built prior to 1940.
- DIS: weighted distances to five Boston employment centers.
- RAD: index of accessibility to radial highways.
- TAX: full-value property-tax rate per 10,000$.
- PTRATIO: pupil-teacher ratio by town.
- B: $1000 * (Bk–0.63)^2$ where Bk is the proportion of blacks by town.
- LSTAT: % lower status of the population.
- MEDV: Median value of owner-occupied homes in 1000$s.

We will show the code for one of these example since they are virtually the same.For the case of the Boston houses, we have had to erase the headers manually since there is not option provided by MLpack library.

Initially, we introduced the data on the system and separate the dataset in predictor and labels, as it is shown in the next code.

```
data::Load("../../Data/Swedish_auto.txt", predictors);
responses=arma::conv_to<arma::rowvec>::from( predictors.row(predictors.n_rows-1));
predictors=predictors(0,arma::span::all);
```

Figure 4.1: Swedish Auto Insurance dataset loading

Then, we split the data into training and testing using the Data::Split function.

```
double testratio;
  std::cout<<"Introduce the ratio of the Test set ";
  std::cin>>testratio;
  data::Split(predictors,responses,traindata,testdata,trainresponses,testresponses,testratio);
  std::cout<<"Size Train"<<arma::size(traindata)<<" Size Test"<<arma::size(testdata)<<std::endl;
```

Figure 4.2: Swedish Auto Insurance dataset Splitting

Now,we are ready to initialized the machine learning classes. We have directly introduced the data on the object instance. Otherwise, we should made used of the Train() method.

```
LinearRegression lr(traindata, trainresponses,0.2,true );
```

Figure 4.3: Linear regression class

```
LARS lars(traindata, trainresponses);
```

Figure 4.4: Least-Angle regression class

Once the model is trained, we can obtained representative attributes such as the parameters of the linear regression and the beta of the LARS.
We have used the internal method ComputeError(Predictors,Responses) in order to obtain the errors made for each method. However, it is important to notice that the lars version does not average the error per number of observation, so we have divided the result by this number so as to obtain the L2-norm error.
The results obtain for the simple regression by each algorithm, taking as training data the 80% and the rest for testing, are the following:

```
Begin linear regression algorithm
Parameters obtained
   22.2037
    3.3262
Be aware that the models includes intercept
Training Regression L2 square error  :1324.89
Test Regression L2 square error  :957.447
Time required: 1273microseconds
```

Figure 4.5: Single linear regression result

```
Begin least-angle regression algorithm
Beta obtained
   3.9934
Training Regression L2 square error  :1522.84
Test Regression L2 square error  :1277.87
Time required: 435microseconds
```

Figure 4.6: One variable Least-Angle regression Result

We observe that the result are quite similar, but the linear regression obtained a bit better result with a slightly bigger computational time.
The result for the multi-variable regression are the following:

```
Begin linear regression algorithm
Parameters obtained
   13.6664
   -0.1119
    0.0480
   -0.0217
    0.9489
   -6.2955
    5.0400
   -0.0148
   -1.1969
    0.2322
   -0.0123
   -0.5869
    0.0133
   -0.3947

Be aware that the models includes intercept
Training Regression L2 square error  :21.0062
Test Regression L2 square error  :33.6281
Time required: 58834microseconds
```

Figure 4.7: Multi-variable linear regression result

```
Begin least-angle regression algorithm
Beta obtained
  -0.1074
   0.0499
  -0.0341
   0.9554
  -1.0519
   5.8209
  -0.0157
  -1.0028
   0.1792
  -0.0108
  -0.3780
   0.0155
  -0.3538

Training Regression L2 square error  :21.9927
Test Regression L2 square error  :34.8154
Time required: 61118microseconds
```

Figure 4.8: Multi-variable Least-Angle regression Result

We obtain the same conclusion that before, although the time difference is more noticeable.

4.1.2 Kmeans

Among the clustering methos, we have selected to implement the K-means which is on of the most extended on the literature.

One of the most known machine learning problem that we could not miss is the classification of iris flowers based on its characteristics.

There are 150 observations with 4 input variables and 1 output variable. The variable names are as follows:

- Sepal length in cm.
- Sepal width in cm.
- Petal length in cm.
- Petal width in cm.
- Class (Iris Setosa, Iris Versicolour, Iris Virginica).

We have loaded the data,but we have previously prepared it on the format correct, we have separated the iris labels from the predictors.
The initialization of the Kmeans class obtain the centroids and the assigment of each data directly.

```
size_t clusters;
std::cout<<"Introduce the number of cluster to search ";
std::cin>>clusters;

arma::Row<size_t> assignments;
arma::mat centroids;

KMeans<> k;
k.Cluster(TrainData, clusters, assignments, centroids);
```

Figure 4.9: K-means clustering implementation

The algorithm took 34052 microseconds.

We have stored the centroid in a txt file as well as the confusion matrix in a csv, although we have to consider that the confusion matrix is not ordered since is not a proper measure for clustering.

4.1.3 PCA

Sometimes it is usefull to apply reducing dimensionality techniques such as the Principal Component Analysis so as to be able to plot the results or to reduce the computational load.
We have applied the PCA algorithm to the Iris dataset of the previous section,4.1.2.
we have implemented as follows, where we have rewritten the inputData to fit into 2 dimensions according to the second argument of the Apply() method.

```
PCA<> p;
p.Apply(InputData,2);
```

Figure 4.10: PCA implementation

4.1.4 Perceptron

We start with the classification method by applying the perceptron into the iris dataset. The perceptron is a single neural layer designed for classification problems.
We initializ the Perceptron class by introducing the data, the labels associated, the number of classes and the maximum number of iterations.

```
Perceptron<> model(TrainData,TrainLabel.row(0),classes,1000);

model.Classify(TestData, predictedLabels);
```

Figure 4.11: Perceptron implementation

We have computed the confusion matrix, which reflects the good fit of this model.

```
Introduce the ratio of the Test set 0.2
Introduce the number of classes  3
Confusion Matrix:
    7.0000         0         0
         0   11.0000         0
         0         0   11.0000

Time required: 32433microseconds
```

Figure 4.12: Perceptron result

4.1.5 AdaBoost

The Adaptive Boosting, AdaBoost, is a method that rely on a weaker classifier family and average different weaker classifier to obtain a better result.
We have use the Wine Quality Dataset. It involves predicting the quality of white wines on a scale given chemical measures of each wine.There are 4,898 observations with 11 input variables and one output variable. The variable names are as follows:

- Fixed acidity.
- Volatile acidity.
- Citric acid.
- Residual sugar.
- Chlorides.
- Free sulfur dioxide.

- Total sulfur dioxide.
- Density.
- pH.
- Sulphates.
- Alcohol.
- Quality (score between 0 and 10).

The MLpack library proposes two possibilities as weaker learner for this algorithm: Decision tree or perceptron.
We have chosen to follow the perceptron, seen in Section 4.1.4.
Firstly, we have to initialise the weaker learner to the problem of study.Then, we use it as input argument to the AdaBoost algorithm.

```
Perceptron<> p(TrainData, TrainLabel.row(0), 10);
AdaBoost<> adab(TrainData,TrainLabel.row(0),10,p);

adab.Classify(TestData,predictions,probabilities);

data::ConfusionMatrix(predictions, TestLabel, Confusion, 7);
```

Figure 4.13: AdaBoost implementation

We run the algorithm whihc took too many time:

```
Introduce the ratio of the Test set 0.3
Size Train11x4898 Size Label1x4898
Test result
Confusion matrix :
           0           0           0           0           0           0   8.0000e+00
           0           0           0           0           0           0   1.0000e+00
           0           0           0           0           0           0            0
           0           0           0           0           0           0            0
           0           0           0   1.0000e+00   4.0000e+00   3.0000e+00   2.0000e+00
           0           0           0   4.0000e+00   2.8000e+01   2.6200e+02   3.7500e+02
           0           0           0   4.0000e+00   2.3000e+01   1.8500e+02   2.3000e+02
Time required: 129612595microseconds
```

Figure 4.14: AdaBoost Result

4.1.6 Logistic regression

Now, we are going to test the logistic regression with the banknote authentication problem, introduced on Section 4.1.6. by the measures obtained by a picture.
There are 1,372 observations with 4 input variables and 1 output variable. The variable names are as follows:

- Variance of Wavelet Transformed image (continuous).

- Skewness of Wavelet Transformed image (continuous).
- Kurtosis of Wavelet Transformed image (continuous).
- Entropy of image (continuous).
- Class (0 for authentic, 1 for inauthentic).

Its implementation is very similar to the one for Linear Regression changing the LinearRegression class to LogisticRegression.
Nevertheless, It provides other test analysis methods more according to a classification problem. By ComputeError() and ComputeAccuracy() method we obtained the following results:

```
Parameters obtained
   6.8029  -7.1602  -3.9052  -4.8844  -0.6252

Training result
Error : 22.9591
Accuracy : 98.725
Test result
Error : 2.21238
Accuracy : 99.2701
Confusion matrix :
   1.4800e+02            0
   2.0000e+00   1.2400e+02
Time required: 22287microseconds
```

Figure 4.15: Logistic Regression Result

4.1.7 SVM

Finally, the last algorithm that we are going to test is the Support-Vector-Machine (SVM) to the classification problem of the wine quality of Section 4.1.5.

For the implementation, we use the SVM class as it is shown on the next images.

```
LinearSVM svm(TrainData,TrainLabel,7);

parameters=svm.Parameters();
std::cout<<"Parameters obtained"<<std::endl;
std::cout<<parameters<<std::endl;

train_accu=svm.ComputeAccuracy(TrainData,TrainLabel);
test_accu=svm.ComputeAccuracy(TestData,TestLabel);
svm.Classify(TestData,predictions);

data::ConfusionMatrix(predictions, TestLabel, Confusion, 7);
```

Figure 4.16: SVM implementation

We have obtained the following result that compared to the ones obtained with AdaBoost we observe that the SVM algorithm is much faster, but obtain a worse result.

```
Parameters obtained
  -5.3123e-04  -6.5980e-04  -1.1158e-03  -9.3369e-04  -2.1963e-04  -1.5237e-03   1.0067e-03
  -4.3645e-05  -3.2389e-05  -6.8765e-05  -7.6810e-06  -9.2839e-06  -2.7411e-05  -4.1006e-06
   6.4550e-06  -3.7582e-05  -3.7531e-05  -3.0842e-05  -1.0061e-05  -7.4193e-05   4.0721e-05
  -4.0700e-04  -6.8803e-04  -9.9590e-04  -7.4514e-04  -6.6310e-04   3.4092e-04   7.9137e-05
   4.6929e-05  -1.0884e-05   1.0856e-05  -3.4506e-05  -4.1965e-05  -4.4215e-06  -1.8317e-05
  -2.8224e-03  -3.2952e-03  -5.1985e-03  -3.3817e-03  -4.6341e-03  -3.4010e-03   4.1747e-04
  -1.0379e-02  -1.3448e-02  -1.9690e-02  -1.5283e-02  -1.0060e-02  -3.7473e-03  -2.4195e-03
  -1.0839e-04  -1.6883e-04  -2.1284e-04  -2.1935e-04  -1.4651e-04   5.1619e-05   1.3537e-04
  -2.7235e-04  -3.1053e-04  -5.3465e-04  -4.4390e-04  -1.3752e-04  -8.4995e-04   4.4645e-04
  -4.4934e-05  -6.3253e-05  -9.2554e-05  -7.5014e-05  -2.7326e-05  -1.3293e-04   9.0883e-05
  -4.4037e-04  -1.1194e-03  -1.9776e-03  -1.7701e-03  -1.0617e-03  -2.3654e-03   1.6027e-03

Training result
Accuracy : 0.459556
Test result
Accuracy : 0.405516
Confusion matrix :
           0           0           0           0           0           0           0
           0           0           0           0           0           0           0
           0           0           0           0           0           0           0
           0           0           0           0           0           0           0
           0           0           0           0           0           0           0
           0           0           0           0           0           0           0
           0           0           0   7.0000e+00   4.0000e+01   3.0400e+02   3.9700e+02
Time required: 289209microseconds
```

Figure 4.17: SVM Results

4.2 OpenNN

We have performed 5 neural models in order to cover a good part of the library.
Mainly we are going to construct our own models adding layers consequently,but we will try also the predefined models per type problem where we will try ModelSelection class,Sec.4.2.3.
We will focus on the main layers: perceptron for regression,4.2.1, and classification activation function,4.2.2, convolutional for image classification,4.2.5, and lstm for forecasting time series,4.2.4.
We vary the training strategies and test analysis from example to example.

4.2.1 Multi-variable regression

The problem of study based on the Boston House Price dataset already mentioned on Section 4.1.1.

We begin introducing the data into the program and preprocessing,for example, the dataset had the index and we defined it as unusable. Then, we have determined the number of inputs and outputs with the descriptives which serve to the scale layer. Finally, we split the data.

```
DataSet data_set("../../Data/Boston.csv",',',true);
data_set.set_columns_uses({"UnusedVariable","Input","Input","Input","Input","Input","Input","I

const size_t inputs_number = data_set.get_input_variables_number();
const size_t outputs_number = data_set.get_target_variables_number();
const Vector<Descriptives> inputs_descriptives = data_set.scale_inputs_minimum_maximum();
const Vector<Descriptives> targets_descriptives = data_set.scale_targets_minimum_maximum();

data_set.split_instances_random();
```

Figure 4.18: Boston dataset introduction

Consequently, we are ready to define our model based on the once presented in https://predictivemodeler.com/2019/10/19/tensorflow-boston-house-prices/. Using this library the data is normally introduced by the scaler layer which funtionality could be set off. Then, we start defining sequentially the layers ensuring that the input number of one layers corresponds to the values of the previous.
In this case, we have added perceptron layers fixing as activation function the rectilinear function, as shown in the next figure.

```
NeuralNetwork neural_network;

// Scaling layer

ScalingLayer* scaling_layer = new ScalingLayer(inputs_number);
scaling_layer->set_descriptives(inputs_descriptives);
//scaling_layer_pointer->set_scaling_methods(ScalingLayer::NoScaling);

neural_network.add_layer(scaling_layer);

const size_t scaling_layer_outputs_dimensions = scaling_layer->get_neurons_number();
//Perceptron block

PerceptronLayer* perceptron_layer_1 = new PerceptronLayer(scaling_layer_outputs_dimensions,64);
perceptron_layer_1->set_activation_function(PerceptronLayer::RectifiedLinear);
neural_network.add_layer(perceptron_layer_1);

const size_t perceptron_layer_1_outputs = perceptron_layer_1->get_neurons_number();

PerceptronLayer* perceptron_layer_2 = new PerceptronLayer(perceptron_layer_1_outputs,64);
perceptron_layer_2->set_activation_function(PerceptronLayer::RectifiedLinear);
neural_network.add_layer(perceptron_layer_2);

const size_t perceptron_layer_2_outputs = perceptron_layer_2->get_neurons_number();

PerceptronLayer* perceptron_layer_3 = new PerceptronLayer(perceptron_layer_2_outputs,1);
neural_network.add_layer(perceptron_layer_3);

const size_t perceptron_layer_3_outputs = perceptron_layer_3->get_neurons_number();

UnscalingLayer* unscaling_layer_pointer = new UnscalingLayer(perceptron_layer_3_outputs);
neural_network.add_layer(unscaling_layer_pointer);

neural_network.print_summary();
```

Figure 4.19: Regression Neural Model Definition

Once we have defined the model, we can proceed to the training phase where we have set the OptimizationMethos as Stochastic Gradient Descendent (SGD) and the loss error the mean squared error.
Moreover, we have set some characteristic of the SGD,the number of epochs and display features.

```
// Training strategy
TrainingStrategy training_strategy(&neural_network, &data_set);
training_strategy.set_optimization_method(TrainingStrategy::OptimizationMethod::STOCHASTIC_GRADIENT_DESCENT);
training_strategy.set_loss_method(TrainingStrategy::LossMethod::MEAN_SQUARED_ERROR);
training_strategy.get_loss_index_pointer()->set_regularization_method(LossIndex::RegularizationMethod::NoRegularization);

StochasticGradientDescent* sgd_pointer = training_strategy.get_stochastic_gradient_descent_pointer();

sgd_pointer->set_initial_learning_rate(1.0e-3);
sgd_pointer->set_momentum(0.9);
sgd_pointer->set_minimum_loss_increase(1.0e-6);
sgd_pointer->set_maximum_epochs_number(1000);
sgd_pointer->set_display_period(100);
sgd_pointer->set_maximum_time(1800);

const OptimizationAlgorithm::Results training_strategy_results = training_strategy.perform_training();
```

Figure 4.20: Training Strategy Definition and Performance

While training phase, the program will display its evolution showwing the training error, selection error, the elapsed time per epoch and the learning rate value, as shown in next picture.

```
Training with stochastic gradient descent...
Epoch 0;
Parameters norm: 71.7789
Training loss: 1.62446
Batch size: 1000
Gradient norm: 0.410514
Learning rate: 0.001
Elapsed time: 00:00
Selection error: 1.74567
Epoch 100;
Parameters norm: 71.7788
Training loss: 1.58365
Batch size: 1000
Gradient norm: 0.125481
Learning rate: 0.001
Elapsed time: 00:06
Selection error: 1.72079
Epoch 200;
Parameters norm: 71.7787
Training loss: 1.57907
Batch size: 1000
Gradient norm: 0.127909
Learning rate: 0.001
Elapsed time: 00:12
Selection error: 1.72104
```

Figure 4.21: Training Strategy Performance display

Once the training phase is ended, a summary of the training is displayed.

```
Epoch 1000: Maximum number of iterations reached.
Parameters norm: 71.7785
Training loss: 1.57135
Batch size: 1000
Gradient norm: 0.000392563
Learning rate: 0.001
Elapsed time: 01:02
Selection error: 1.73463
```

Figure 4.22: Training Strategy Performance Summary display

Finally, we perform the analysis of the neural model on the test dataset.

```
// Testing analysis

TestingAnalysis testing_analysis(&neural_network, &data_set);
cout<<endl<<"Testing Analysis"<<endl;
Vector< double > TestError=testing_analysis.calculate_testing_errors();
cout<<"Sum Squared error  :"<<TestError[0]<<endl;
cout<<"Mean Squared error   :"<<TestError[1]<<endl;
cout<<"Root Mean Squared error   :"<<TestError[2]<<endl;
cout<<"Normalized Squared error   :"<<TestError[3]<<endl;

const TestingAnalysis::LinearRegressionAnalysis linear_regression_results = testing_analysis.perform_linear_regressi
cout << "Linear Regression analysis"<<endl;
cout<<"Correlation    : " << linear_regression_results.correlation << endl;
```

Figure 4.23: Regression Test analysis

We have obtained the error committed and the linear regression that shows the correlation between the predictions and the real data.

```
Testing Analysis
Sum Squared error  :165.998
Mean Squared error   :1.64354
Root Mean Squared error   :1.28201
Normalized Squared error   :10.7843
Linear Regression analysis
Correlation    : 1
```

Figure 4.24: Regression Test analysis results

4.2.2 Multi-Classification problem

We have chosen the Iris dataset already introduced on Section 4.1.2.

We have introduced the data,by this time we have set the batch size, by default equal to 1000, with the data_set.set_batch_instances_number() method.
We based on the model explained in https://machinelearningmastery.com/multi-class-classification-tutorial-keras-deep-learning-library/.

The main difference respect the regression problem model is that we have add the probabilistic layer with the softmax activation function.

```
ProbabilisticLayer* probabilistic_layer = new ProbabilisticLayer(perceptron_layer_1_outputs, outputs_number);
probabilistic_layer->set_activation_function(ProbabilisticLayer::ActivationFunction::Softmax);
```

Figure 4.25: Probabilistic layer coding

We have change the training strategy to include other optimization method and loss function, quasi newton and normalized mean square error.
Finally, in other to check the well-posedness of the model we have displayed the confusion matrix obtaining the following:

```
Testing Analysis
Confusion:
11 0 0
0 5 4
0 10 0
```

Figure 4.26: Iris confusion matrix

4.2.3 Binary Classification problem

Now, we are going to test the already-given library structures with the banknote authentication problem, introduced on Section 4.1.6.

This time we have used the pre-defined models specifying the input number, hidden number of neurons and the output number to obtained a model compound of a scaler layer, two perceptron layer and a probabilistic layer.
We should set the scaling layer off by changing its scaling method, as shown in the next code.

```
// Neural network
NeuralNetwork neural_network(NeuralNetwork::Classification, {8, 6, 1});

// Scaling layer

ScalingLayer* scaling_layer_pointer = neural_network.get_scaling_layer_pointer();
scaling_layer_pointer->set_descriptives(inputs_descriptives);
scaling_layer_pointer->set_scaling_methods(ScalingLayer::NoScaling);
```

Figure 4.27: Pre-defined Model code

In this example, we have add to the training strategy L2-Regularized commonly used to avoid overfitting on the loss_index pointer.

Moreover, we have proven the model selection class. We have varied the number of neurons since the number of inputs variables is quite short.
After the normal training performance, the model selection class is called. It makes different modifications on the model and repeat the training phase selecting the best model. In the next figure, we observe the output of the model.

```
Algorithm finished.
Iteration: 10
Hidden neurons number: 10
Training loss: 59.2315
Selection error: 20.4083
Elapsed time: 01:18


Optimal order: 6
Optimum selection error: 19.1846
Corresponding training loss: 58.6166
```

Figure 4.28: Pre-defined Model code

Finally, we have applied the function desinged for binary classification errors of the TestingAnalysis class apart of the previous confusion matrix.

```
Test Analysis
Confusion:
65 61
2 146
Binary classification tests:
Classification accuracy         : 0.770073
Error rate                      : 0.229927
Sensitivity                     : 0.515873
Specificity                     : 0.986486
Precision                       : 0.970149
Positive likelihood             : 38.1746
Negative likelihood             : 2.03766
F1 score                        : 0.673575
False positive rate             : 0.0135135
False discovery rate            : 0.0298507
False negative rate             : 0.484127
Negative predictive value       : 0.705314
Matthews correlation coefficient: 0.582516
Informedness                    : 0.50236
Markedness                      : 0.956636
```

Figure 4.29: Binary Classification Test results

4.2.4 Time Series Forecasting

Shampoo-Sales dataset is perfect for time series forecasting. It has 36 observation of the shampoo sales count during three years.
When dealing with time series, it is important to predefined before loading the dataset the number of influential past lags t number of steps ahead that are going to be predicted. Beside, this we have to specify that the fist column correspond to time variable.

It is very important to have into account that the dataset split should be sequential instead of random.

```
DataSet data_set;
    data_set.set_data_file_name("../../Data/shampo_sales.csv");
data_set.set_separator(',');
data_set.set_has_columns_names(true);
size_t lags_number = 4;
size_t steps_ahead = 1;

data_set.set_lags_number(lags_number);
data_set.set_steps_ahead_number(steps_ahead);

data_set.set_time_index(0);

data_set.set_missing_values_method("Mean");

data_set.read_csv();


const size_t inputs_number = data_set.get_input_variables_number();
const size_t outputs_number = data_set.get_target_variables_number();
const Vector<Descriptives> inputs_descriptives = data_set.scale_inputs_minimum_maximum();
const Vector<Descriptives> targets_descriptives = data_set.scale_targets_minimum_maximum();

data_set.split_instances_sequential();
```

Figure 4.30: Shampoo Sales dataset loading

Thereafter,it is very similar to the treatment to the regression problem changing the two perceptron layers to one lstm layers.This time, we have used the adam algorithm for the optimization with the mean squared error.
The results obtained by this implementation is the following:

```
Testing Analysis
Sum Squared error  :1.35931
Mean Squared error   :0.194187
Root Mean Squared error   :0.440667
Normalized Squared error   :1.61311
Linear Regression analysis
Correlation    : 0.449616
```

Figure 4.31: Shampoo Sales test results

4.2.5 Image Classification

One of the most interesting problem dealt with neural model is the Image Classification.

We have decided to use the Fashin MNIST dataset,[XRV17], which is a variation of the most used one MNIST. It is a dataset of Zalando's article images (60000 training images and 10000 for testing) and the aim is to classify one image into the following classes:

- T-shirt/top
- Trouser
- Pullover
- Dress
- Coat
- Sandal
- Shirt
- Sneaker
- Bag
- Ankle boot

Firstly, we have to read the data. We need to set the dimension of the input to (1,28,28) since the dataset give it as one row. Beside this, we need to apply a one-hot encoding to the label variable, so we have to set the target dimension as a vector of the number of labels.
Morever, we have performed the analysis only to 125 images to reduce the computation time required by the program.

```
DataSet data_set("../Data/fashion-mnist_train.csv", ',', true);

data_set.set_input();
data_set.set_column_use(0,DataSet::VariableUse::Target);
data_set.numeric_to_categorical(0);
data_set.set_batch_instances_number(5);

const Vector<size_t> inputs_dimensions({1, 28, 28});
const Vector<size_t> targets_dimensions({10});
data_set.set_input_variables_dimensions(inputs_dimensions);
data_set.set_target_variables_dimensions(targets_dimensions);

//Since it is an example, we are not going to train over the 60000 images
const size_t total_instances = 125;
data_set.set_instances_uses((Vector<string>(total_instances, "Training").assemble(Vector<string>(60000 - total_instanc
data_set.split_instances_random(0.75,0.1,0.1);
```

Figure 4.32: Fashion-MNIST dataset loading

We have constructed our own model by adding convolution blocks that we have defined by adding a convolutional layers followed by pooling layer. We have specify for each convolutional layer ,apart from the input size, the number of filter, their rows and columns.
Once we have finished with the convolutional block, we add the fully connected layer.

Be aware that since the library does not provide a flatten layer, we have to calculate the input layer dimension as the internal product of the output dimension of the convolutional part.
We have made this with the next code.

```
// Convolutional Block

ConvolutionalLayer* convolutional_layer_3 = new ConvolutionalLayer(pooling_layer_2_outputs_dimensions, {2, 3, 3});
neural_network.add_layer(convolutional_layer_3);

const Vector<size_t> convolutional_layer_3_outputs_dimensions = convolutional_layer_3->get_outputs_dimensions();

PoolingLayer* pooling_layer_3 = new PoolingLayer(convolutional_layer_3_outputs_dimensions);
neural_network.add_layer(pooling_layer_3);

const Vector<size_t> pooling_layer_3_outputs_dimensions = pooling_layer_3->get_outputs_dimensions();

// Fully conected layer: Dense layer with number input equal to flatten

PerceptronLayer* perceptron_layer = new PerceptronLayer(pooling_layer_3_outputs_dimensions.calculate_product(), 18);
neural_network.add_layer(perceptron_layer);
```

Figure 4.33: Convolution block and fully conected layer code

Once we have defined the model, the training and test are standard.
We have obtained the following results by training with SGD algorithm and mean squared error.

```
Epoch 11;
Parameters norm: 105.436
Training loss: 1.31657
Batch size: 5
Gradient norm: 0.0465498
Learning rate: 0.001
Elapsed time: 03:31
Selection error: 0.860875
Epoch 12: Maximum number of iterations reached.
Parameters norm: 105.436
Training loss: 1.30501
Batch size: 5
Gradient norm: 0.0285116
Learning rate: 0.001
Elapsed time: 03:49
Selection error: 0.890203


Confusion matrix:

0 0 0 0 0 0 0 2 0 0
0 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 1 0 0
0 0 2 0 0 0 0 0 0 0
0 0 2 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0 1
0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 0 0

Accuracy: 7.69231 %
```

Figure 4.34: Fashion-MNIST Results

4.3 Comparison with Python

We have performed some of these test in Python as well.
We have provided some notebook written using anaconda with jupyter. The machine learning algorithms were implemented using the scikit-learn library,[Ped+11], and the neural networking using keras and tensorflow,[Aba+15].

4.3.1 Scikit-learn

The organization of the scikit-learn and mlpack are very similar. Both consider each technique as an object that keep all the information necessary to apply them to external data along with other utilities.
We have implemented the multivariate regression with the class LinearRegression() obtaining a Test regression L2 square error of 16.57 in 578 microseconds what would suggest a better performance of the algorithm provided by the Python's library.
Additionally, we have tried with the Logistic Regression with the LogisticRegression() class which has obtained almost the same results in 440 microseconds. The mlpack is a bit better, its accuracy was 99.27% on contrast to the 99.18% of the scikit-learn version.

The confusion matrix obtained by the scikit-learn option was:

146 2
2 125

Finally, we have tested the Support-Vector-Machine obtaining better results on 2138 microseconds. Its accuracy on the training set was 0.4511. However, this difference could be due to the fact that the split in both cases have been random , so the variance on these set could explain this difference.

4.3.2 TensorFlow

Both library provide a own data class to ease the computation. However, we have not use the Dataset class from Tensorflow because the dataset used are easily handled with the pandas library.
One big difference between those library is that while OpenNN consider the TrainingStrategy and TestAnalysis as separated class to the NeuralNetwork, tensorflow introduce this inside the model class using the compile() and fit() internal methods for the training. Moreover, It does not provided a specific TestAnalysis class.
Additionally, the tensorflow library has implemented much more utilities than the OpenNN such as the functional model creation that allows non-sequential models or the contribution of standar models for fine-tuning techniques such as VGG16 or InceptioV3.

We have tested three of the previous neural models emulating the same structures. We have recreated the multivariate regression neural model obtaining a Mean Squared Error of 11.85 and a $R^2 score$ of 0.87 on contrast of the 1 obtained with OpenNN alternative.
We have computed also the LSTM model obtaining worse result with the Python model. In this aspect, the OpenNN provide a better solution since they have implemented the transformations needed to apply on the data to express it as a time-series.
Finally, we have perform the image classification model. To emulate the exact same model, we have avoid using the BatchNormalization layer of tensorflow which improve the result because OpenNN does not have it.
In general, the performance of the OpenNN is worse.
Firstly, the different times required to read the data was notorious, while pandas Python's library took 3.47 seconds, OpenNN took several minutes.
The same happened on the training phase, training over the 60000 images with OpenNN was not feaseble, so we have decided to work with a reduced model to show how it works.
Hence, we have obtained better results with the tensorflow approach obtaining an accuracy of 63.11% compared to the 7.69% of the OpenNN.

5 Compilation Instructions

The project has been compile using Ubuntu 18.04.

5.1 Mlpack

The installation and compilation is based on the information provided on https://www.mlpack.org/doc/mlpack-git/doxygen/build.html.

Firstly, it is necessary to install the dependencies:

- Armadillo>=8.400.0
- Boost>=1.58, specifically the modules math_c99, program_options, serialization, unit_test_framework, heap and spirit.
- ensmallen>=2.10.0, will be download if not found.
- CMake>=3.3.1

To install the dependencies, you can use the following commands:

```
$sudo apt-get install libboost-math-dev libboost-program-options-dev libboost-random-dev libboost-test-dev libboost-serialization-dev libboost-all-dev

$sudo apt-get install libxml2-dev libarmadillo-dev binutils-dev doxygen
```

Once the dependencies are ready, we move to the folder with the package to make a build folder where we will execute makefile to install the library.

```
$ cd build
$ cmake -D DEBUG=ON -D PROFILE=ON ../
$ make
$sudo make install
```

When executing cmake, it would be needed to specifies the Boost library directory on the command line (-DBOOST_LIBRARYDIR) since it is stored in /usr/lib/x86_64-linux-gnu. Otherwise,the modules program_options, serialization, unit_test_framework will not be found.

Now, we are ready to run our examples. All the examples include a makefile to ease their compilation. The user will need to change the variable MLPACK_DIR equal to the directory "include" on the previous build directory created.

5.2 OpenNN

The installation and compilation is based on the information provided on
https://www.opennn.net/documentation/building_opennn.html
The only dependency needed is the Eigen library which will be installed automatically while installing OpenNN.
We can download the package with GitHub or SourceForce.However, during the installation we have found that the SourceForce wil not found the information to install the given examples by the library. Therefore, It would be recommended to follow the GitHub via.
We have proceed similarly what we have done with mlpack:

```
$ mkdir opennn-master\Release&&cd opennn-master\Release
$ cmake -DCMAKE_TYPE_BUILD=Release .
$ make
```

This will generate the libopennn.a in Release\opennn.
For each example is associated a makefile where only should be modified the directory of the library content,OPENNN_DIR, and the liopennn.a,LIB_DIR.

Bibliography

[Aba+15] M. Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/ (cit. on p. 32).

[Art] L. Artificial Intelligence Techniques. "opennn neural networks". In: (). URL: https://www.opennn.net/ (cit. on p. 11).

[Bha+18] S. Bhardwaj, R. R. Curtin, M. Edel, Y. Mentekidis, and C. Sanderson. *ensmallen: a flexible C++ library for efficient function optimization*. cite arxiv:1810.09361Comment: Workshop on Systems for ML and Open Source Software at NIPS / NeurIPS, 2018. 2018. DOI: 10.5281/zenodo.2008650. URL: http://arxiv.org/abs/1810.09361 (cit. on p. 7).

[Cur+18] R. R. Curtin, M. Edel, M. Lozhnikov, Y. Mentekidis, S. Ghaisas, and S. Zhang. "mlpack 3: a fast, flexible machine learning library". In: *Journal of Open Source Software* 3 (26 2018), p. 726. DOI: 10.21105/joss.00726. URL: https://doi.org/10.21105/joss.00726 (cit. on p. 7).

[Ped+11] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830 (cit. on p. 32).

[San16] C. Sanderson. "Armadillo C++ Linear Algebra Library". In: (June 2016). DOI: 10.5281/zenodo.55251. URL: https://doi.org/10.5281/zenodo.55251 (cit. on p. 7).

[Sch14] B. Schäling. *The boost C++ libraries*. XML Press, 2014. ISBN: 9781937434366 1937434362 (cit. on p. 7).

[XRV17] H. "Xiao, K. Rasul, and R. Vollgraf. ""Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms"". In: (Aug. 28, 2017). arXiv: cs.LG/1708.07747 [cs.LG] (cit. on p. 29).