ESLR laboratory project

Pierre Guillaume

Epita engineer school

Abstract. This report introduces the work done for the ESLR laboratory project. The whole evolution of the work and the acquired knowledge will be demonstrated in this paper. The main goal of the ESLR project is to classify malware by using the ember dataset. This project is divided into three parts. The first part deals with the optimization of K-means classification algorithm. The second part of the project shows how to classify malware using machine learning methods. And the third part is about deep learning methods for malware classification.

1 Kmeans optimizations

In this part we will describe how the k-means algorithm works and also one optimization method. The goal is to make the k-means algorithm faster using OpenMP library.

1.1 K-means algorithm

The K-means is an unsupervised learning algorithm used to classify unlabeled data. The goal of the algorithm is really simple, according to an input k, it has to separate the data into k different clusters.

To achieve this, the k-means follows different steps:

- Generate k random points.
- Assign each data point to its closest cluster (k).
- Compute the mean of each cluster (means of its points), and the mean becomes the new cluster.
- Come back to the second step (stop case: maximum of iteration)

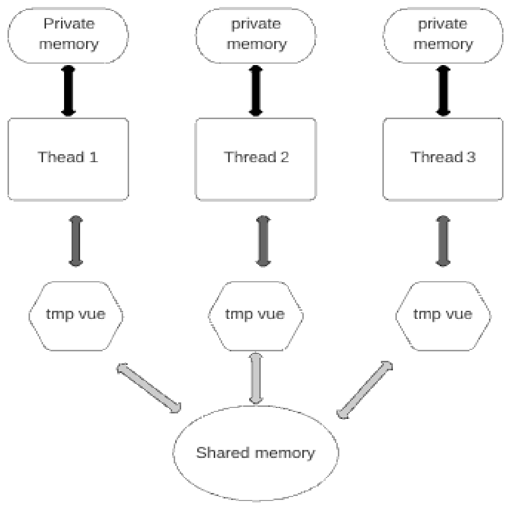
However, this algorithm is not efficient when it comes to time speed. So, a faster version of the algorithm have been created. It's called the mini batch k-means. Moreover, other methods can be used to make it faster like parallelism with OpenMP that we will discribe in the next part.

1.2 Parallelism with OpenMP

OpenMP is the library (C/Fortran) for managing multi-threading. The goal is to indicate to the compiler the instruction to execute in parallel.

The OpenMP library uses a specific model for parallelism. This model is called the fork-join model. The thread master creates other thread (workers) and create a team with them. The other threads work in parallel for the master thread. Each thread has it's own private memory and also each thread has access to a shared memory.

We can see the fork join model on the following picture:



In our case, OpenMP have been used to speed up the k-means algorithm. We instantiated 8 threads because I have 4 processors each composed of 2 cores, one thread for each core. Regarding the speed, to classify ember dataset, before using OpenMP multi-threading, the k-means needed 7 seconds. After the use of OpenMP it took only 1.9 second to classify the ember dataset.

2 Machine Learning in Malware Classification

This part of the report shows two efficient methods of machine learning to achieve the malware classification using Sklearn python library. This part will also deal with partial fit and online learning.

2.1 Logistic Regression

The Logistic Regression model is an algorithm used to solve classification problems that rely on many observational datas. The dataset must contains vectors and labels. The idea between any regression algorithm is to find the best function that will explain the dataset. This function must find the relation between each vector and its associated label. In our case, before using this algorithm we had to pre-process the data to make the algorithm more efficient. Therefore, standard normalization have been used to normalize the data. The main goal of input normalization is to speed-up learning and also to prevent overfitting.

Standard Normalization: Each element is changed (normalized) according to the following formula.

$$x = \frac{x - \mu}{\sigma}$$

where:
μ: is the mean
σ: is the standard deviation

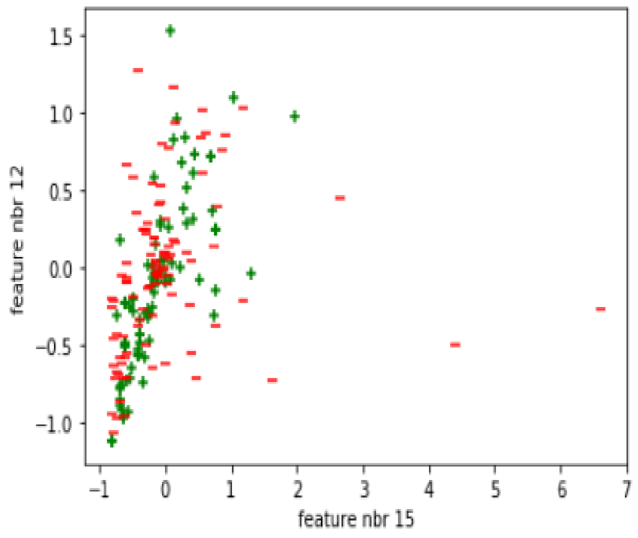
In our case, we have used Numpy arrays to store the data, so for the normalization it was faster to use pre-processing module from sklearn, more precisely the *'StandardScaler()'* function.

Then, the data have been shuffled to be sure, for training, that the distribution is homogeneous. To perform this we have used *'suffle()'* function from sklearn utils module.

After this step, data have been separated: 50000 vectors are used for testing, 50000 for validation and 500000 for training.

For this model, matplotlib have also been used to create a graphical representation of the dataset. Hence, according to specific features, 100 elements have been plotted.

We can see this representation on the following picture.



The green plus signs represents the malicious vectors and the red minus sings shows the benign ones. As we can see, it's difficult to separate them just by knowing only two features.

After the pre-processing of the data, we can use our logistic regression to classify the data and be able predict if a new vector (composed of metadata) is a malicious program or not.

Logistic Regression: The logistic regression algorithm uses a set of data as input and tries, after many learning epochs, to classify new input data. This algorithm is really similar to linear regression, but it uses a logarithmic loss function.

Moreover, the logistic regression uses the 'Sigmoid' activation function, to squash the input between 0 and 1.

The 'Sigmoid' activation function is defined by:

$$f(x) = \frac{1}{1 + e^{-x}}$$

The 'Sigmoid' activation function is useful in neural networks when it comes to output layer, because it ranges the output between 0 and 1. However, in some case in neural network, the use of this function can lead to the vanishing gradient problem during backpropagation and make the learning less efficient.

The logistic model uses parameters (Θ in our case), initialized randomly. These parameters are updated by the training and used for the classification process.

Thus, the output of the logistic regression is computed by:

$$z = \Theta_0 + \Theta_1 x_1 + ... + \Theta_n x_n$$

$$output = sigmoid(z)$$

Then, the cost function is computed:

$$Cost(y, \widehat{y}) = \begin{cases} -\log(\widehat{y}) & : y = 1 \\ -\log(1 - \widehat{y}) & : y = 0 \end{cases}$$

To train our model, it is needed to minimize the cost function by using partial derivative.

However, the first problem we encountered was during training. Actually, the dataset is about 10GB so it didn't fit with the memory. So, the solution that have been found is to use partial fit. It simply consist of using only small parts of the dataset for fitting. And then, repeat it many time to fit the model on the all dataset. It has been done, using the *'partial_fit()'* method from Sklearn. To use this method, we need to instantiate first a Stochastic gradient classifier (from sklearn) because the partial fit method cannot be used on every model. Thereby, we called the *'SGClassifier()'* method with 'log' as loss parameter to use Logistic Regression.

By default, the model is called with L2 penalty.

L1 and L2 regularization: These methods are used to prevent overfitting and allow a better learning. L1 and L2 regularization methods are adding special penalties to the loss function, in order to make the learning different and to prevent overfitting.

For example, in the case of linear regression, the output is computed thanks to the following expression.

$$\widehat{y} = w_1 x_1 + w_2 x_2 + ... + w_N x_N$$

Hence, for the L1 regularization (also called Lasso Regression), the loss is computed like that:

$$loss = ErrorFunction(y, \widehat{y}) + \lambda \sum_{i=1}^{N} |w_i|$$

For the L2 regularization (Ridge Regression), we have:

$$loss = ErrorFunction(y, \widehat{y}) + \lambda \sum_{i=1}^{N} w_i^2$$

The shuffle parameter, really important for avoiding overfitting, is by default activated.

Overfitting appears when the model becomes really good on the training data and really bad or not efficient on the testing data.

Therefore, the final Logistic Regression model have been train on 10 epochs (10 times the whole dataset), between each partial fit, the data have been shuffled. The prediction accuracy on the validation set is also measured between each epoch.

After the 10 epochs, the accuracy during training has reached 96.454%. To see if our model is working we need to check the performances of the testing set. So, the testing accuracy is about 96.342%

2.2 Linear Support Vector Classifier

This part describes the Linear SVM algorithm to classify malware.

As in the previous part, the input data have been normalized using the standard normalization. Moreover, the data have been separated into testing set, training set and validation set using the same distribution as in the previous method.

The goal of support vector machine is to create a frontier between the two classes. And this separator must be as far as possible from both classes. So, the SVM algorithm must find the best separator possible.

To predict the class of a new data, SMV model uses these formulas:

$$h(x) = \sum_{i=1}^{n} (w_i x_i + b) = W^T.X + b$$

where:
X: vector to predict
W: vector of weights (learning parameters)
b: the bias (learning parameter)

Then, we look at the sign of h(x):

$$\begin{cases} h(x) >= 1 & : label1 \\ h(x) <= -1 : label - 1 \end{cases}$$

But before prediction, the model needs to be trained, using 'hinge loss' (in our case). Hence, weights are updated by minimizing the following expression:

$$\min([\frac{1}{n}\sum_{i=1}^{n}(\max(0, 1 - y_i(W.X - b)))] + \lambda||w||^2)$$

Our classification model has been defined using *SGClassifier()* method with the 'hinge' loss and the shuffle boolean activated. Then the model has been fitted during 10 epochs using partial fit because the all the data were too big compared to the memory.

Finally, during training, the model had an accuracy of 0.9654 while during testing the accuracy was about 0.96482.

3 Deep Learning in Malware Classification

In this part we are going to show many different deep learning models for dealing with malware classification using ember dataset. Comparison between the models will be demonstrated.

3.1 DNN model

The first model that have used to classify the data is a simple deep neural network (DNN). This model was composed of one input layer of size 2351 (Dimension of each vector), 6 hidden layers with respectively 700, 650, 550, 400 and 300 neurons. Each neuron of the hidden layer was activated by the 'ReLU' activation function.

ReLU activation function: One of most the famous activation function.

The ReLU activation function is defined by:

$$f(x) = max(0, x)$$

One of the biggest advantage of the ReLU activation function is that it doesn't activate all the neurons at the same time. This property makes the learning more efficient. However, the biggest fault of this activation function is the dead neurons. In some particular cases, especially in deep neural networks, some negative neurons are never updated what makes them never activated (dead neurons). To correct this problem, many other function have been created (Leaky ReLU, SeLU, Swish ...)

Each hidden layer was also constitute of batch normalization and dropout (0.2) to prevent overfitting.

Dropout: The goal of the dropout is to add to each neuron a propability to be activated. In that case, the neuron will not always be activated. The idea is to deactivate some neurons to make the network able to create new learning configurations. This method is mostly use to prevent overfitting and increase the learning accuracy.

Batch normalization: Really powerfull method to reduce overfitting. It also allows to use less dropout and a higher learning rate (speed up learning).

Let's go deeply into the batch normalization method. This method uses two learning parameters γ and β. It takes as input these two parameters and a mini batch of input.

So each output that goes through a batch normalization layer is modified according to the following formulas:

$$\widehat{x_i} = \frac{x_i - \mu}{\sqrt{\sigma^2 + \varepsilon}}$$

$$y_i = \gamma\widehat{x_i} + \beta$$

And then, the model had an output layer of 1 neuron activated by the 'Sigmoid' activation function.

This activation function is useful when it comes to the output layer. In fact, the 'sigmoid' scales any input signal between 0 and 1 making the output more suitable to the resolution of a binary class problem.

The error of the model was computed by the Binary Crossentropy and this error was backpropagated by the Adam optimizer. To perform a better learning, the batch size was set at 10 (update our model every ten elements).

Binary Crossentropy: The binary crossentropy is a loss function used to compute the loss when there are only 2 classes of label (0 or 1). This loss function is defined by:

$$L(y,\widehat{y}) = \frac{1}{N}\sum_{i=0}^{N}(y * \log(\widehat{y_i}) + (1-y) * \log(1-\widehat{y_i}))$$

Then, the loss function must be minimized for learning, using different optimizer.

Adam Optimizer: Adam is an adaptive learning rate algorithm. It means that each parameter has an individual learning rate which is different of SGD that have one learning rate for all parameters.

This learning rate is computated according to first and second moment of the gradient.

The first moment is defined by:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)g_t$$

The second moment is defined by:

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)g_t^2$$

where:
β_1: is the first Adam parameter (usually 0.9)
β_2: is the second Adam parameter (usually 0.999)
g_t: gradient on current minibatch

And then the weights are updated according to the formulas:

$$\widehat{m_t} = \frac{m_t}{1-\beta_1^t}$$

$$\widehat{v_t} = \frac{v_t}{1-\beta_2^t}$$

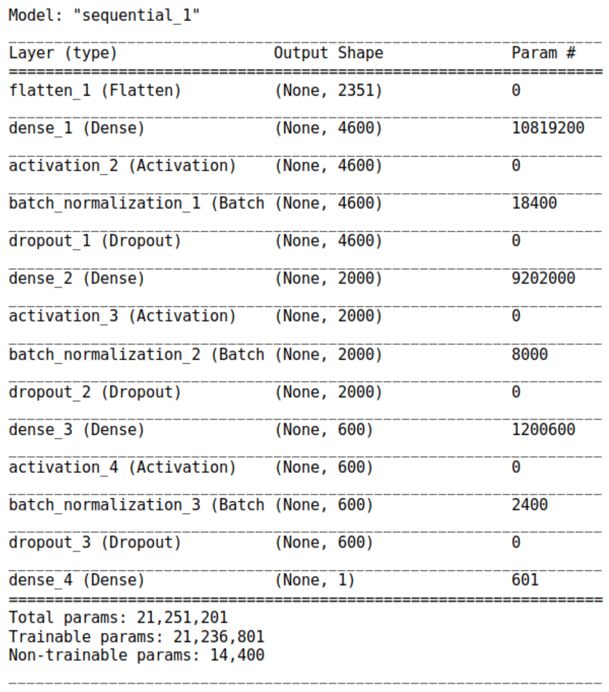
$$w_t = w_{t-1} - \eta\frac{\widehat{m_t}}{\sqrt{\widehat{v_t}}+\epsilon}$$

After one first session of training of 4 epochs (4 training on the whole dataset), the first major problem was encountered. The accuracy during the training was about 0.9554 and the accuracy during the evaluation (on our tests data) was about 0.49646. Which means that our model was good to perform classification on the training data but really bad on new data that the model has never seen. Our model was doing overfitting. After these result, the dataset has been shuffled between each epoch to make learning more efficient and the batch size has been increased. The results remained the same.

After many tries, it has been decided to completely change the model.

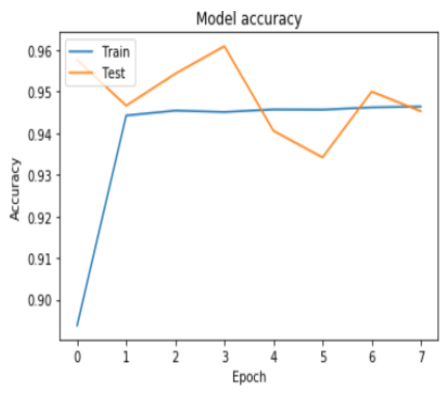
The new model starts first by a normalization of the input data thanks to the standard normalization. As the previous one, the model has and input layer of size 2351. It also has 3 hidden layers with respectively 4600, 2000 and 600 neurons. Here, the goal was to reduce the complexity of our model. Each layer, is activated by the linear activation function. This activation function is not really special, for each x as input it gives the same x as output. Then, each output of this activation function goes trough a regularization L1 (0.001) And then, after the regularization, the output is activated thanks to ReLU. Here, we wanted to use first the regularization and then the ReLU activation to avoid using the L1 regularization on the zero output that ReLU can provide. Each hidden block had also a Batch normalization layer and a dropout (0.01). Finally, the model had an output layer of one neuron activated by the Sigmoid.

The following picture is a summary of the model that we have described before.



```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_1 (Flatten)          (None, 2351)              0
_________________________________________________________________
dense_1 (Dense)              (None, 4600)              10819200
_________________________________________________________________
activation_2 (Activation)    (None, 4600)              0
_________________________________________________________________
batch_normalization_1 (Batch (None, 4600)              18400
_________________________________________________________________
dropout_1 (Dropout)          (None, 4600)              0
_________________________________________________________________
dense_2 (Dense)              (None, 2000)              9202000
_________________________________________________________________
activation_3 (Activation)    (None, 2000)              0
_________________________________________________________________
batch_normalization_2 (Batch (None, 2000)              8000
_________________________________________________________________
dropout_2 (Dropout)          (None, 2000)              0
_________________________________________________________________
dense_3 (Dense)              (None, 600)               1200600
_________________________________________________________________
activation_4 (Activation)    (None, 600)               0
_________________________________________________________________
batch_normalization_3 (Batch (None, 600)               2400
_________________________________________________________________
dropout_3 (Dropout)          (None, 600)               0
_________________________________________________________________
dense_4 (Dense)              (None, 1)                 601
=================================================================
Total params: 21,251,201
Trainable params: 21,236,801
Non-trainable params: 14,400
_________________________________________________________________
```

Hence, the model has been trained during 8 epochs and the validation set have been used during training as we can see on the picture:



With this model, we finally reached 0.9464 (accuracy) during training and 0.91984 during testing.

3.2 CNN model

Because of the low accuracy of the DNN model, Convolution Neural Network (CNN) have been used on the data. As the previous model, the data have been processed before training using Min Max normalization.

Min Max normalization: In the Min Max normalization, a new element is defined using the following formula:

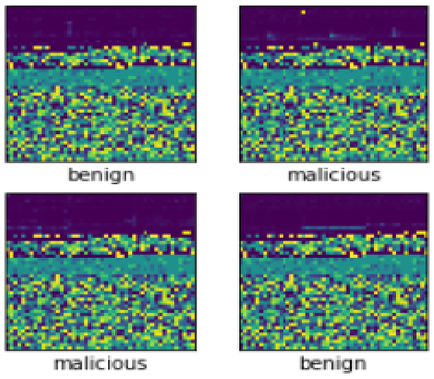
$$x = \frac{x - \min(A)}{\max(A) - \min(A)}$$

where:

A: vector of data

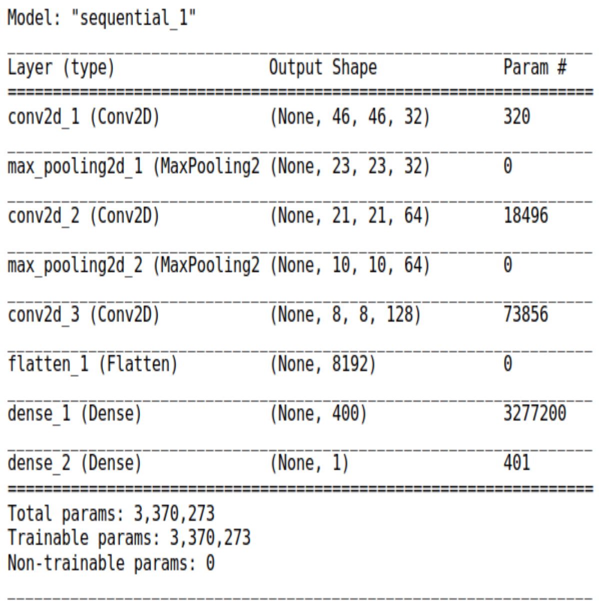
Moreover, CNN models are really efficient on image processing and it needs an image as input. Each element is a vector of 2351 features, so it was needed to reduce the complexity of our vectors to consider them as images (48x48). After dimension reduction, each vector has 2304 features (48x48 image).

Using matplotlib, we can represent our vector as an image:



It is quite difficult to see the difference between a benign and a malicious, but the CNN model will do it for us, and better than human.

For malware classification problem, the best CNN model that have been found is the following:



```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 46, 46, 32)        320
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 23, 23, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 21, 21, 64)        18496
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 10, 10, 64)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 8, 8, 128)         73856
_________________________________________________________________
flatten_1 (Flatten)          (None, 8192)              0
_________________________________________________________________
dense_1 (Dense)              (None, 400)               3277200
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 401
=================================================================
Total params: 3,370,273
Trainable params: 3,370,273
Non-trainable params: 0
_________________________________________________________________
```

The input of our model is a numpy array of size (48, 48, 1). This input array goes through a first 2D Convolution Layer with 32 filters, a kernel of size (3, 3) and 'ReLU activation function'. So, the output of this layer is (46, 46, 32) meaning the size of the image has decreased. Then, the image (numpy array) is processed by a max Pooling layer that reduced the image size by two. In CNNs usually one convolution layer is composed of the 2D Convolution operation and a max pooling.

Then, the CNN model has another Convolution layer with this time 64 filter. Indeed, as the input size is decreasing, more filters are needed. This layer has also a max pooling part. And then, the model has a last Convolution hidden layer with 128 filter, but this time without max pooling.

All this CNN model is stacked on a simple fully connected sequential model. The sequential model is defined by one hidden layer of 400 neurons and 'ReLU' activation function and 1 neuron as output layer (with 'Sigmoid' activation).

In this model, Batch normalization and L1/L2 penalties were not necessary, because no overfitting has been encountered.

Max Pooling: Operation which consist of slicing a kernel (like convolution) on an image. The goal of this operation if to down sample the input representation.

After the definition of the model, the model is compiled with Adam optimizer and Binary Crossentropy.

Then the model is trained on 4 epochs with the shuffle option activated to increase the learning accuracy and reduce overfitting.

After 4 epochs the model reached an accuracy of 0.9765 on the training set and 0.98006 on the testing set.

3.3 LSTM model

The LSTM model is the third model that have been tried. Like the CNN model, it has shown a really good accuracy.

LSTM (Long Short Term Memory) are special type of recurrent neural networks. The big difference between RNN and LSTM is their cells (RNN or LSTM don't have simple neurons like other networks).

Tanh activation function: The tanh activation function is often used in RNN. So, this function is defined as:

$$tanh(x) = \frac{\sinh(x)}{\cosh(c)} = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

RNN cell: In a RNN network, one cell is defined as in the following picture.

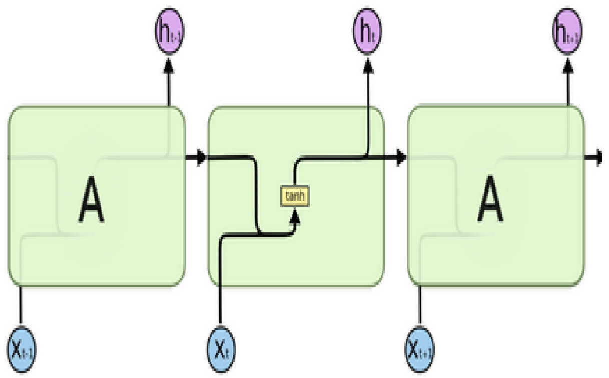


Image From: Here

And the output is computed according to the following formulas:

$$h_t = \tanh\left(W_{hh}h_{t-1} + W_{xh}x_t\right)$$

$$y_y = W_{hy}h_t$$

The major problem with RNN is the vanishing gradient problem. To overcome this problem, the LSTM neural networks have been created.

LSTM Cell: The LSTM cell is composed of 3 part.
Forget Gate: decides what information to forget.
Input Gate: decides according to the input if it is necessary to update our memory state.
Output Gate: decides the output of the cell according to the input and the memory state.

So an LSTM cell can be summary as the following:

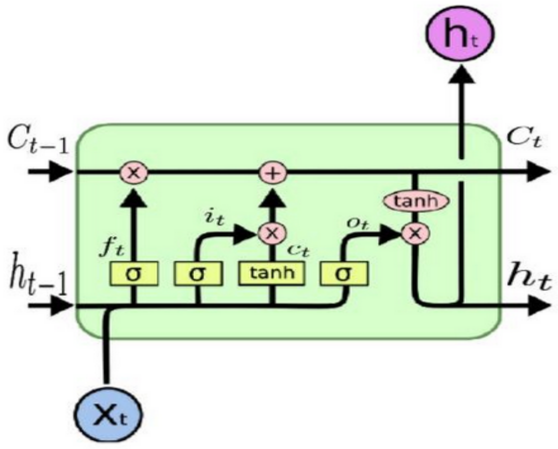


Image from: Here

We can see on the picture that *f*, *i* and *o* correspond respectively to the forget, input and output gates. Hence, The output of the LSTM is defined as the following:

$$f_t = \sigma(W_f.[h_{t-1}, x_t] + b_f)$$

$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i)$$

$$\widetilde{C_t} = \tanh\left(W_c[h_{t-1}, x_t] + b_c\right)$$

$$o_t = \sigma(W_t[h_{t-1}, x_t] + b_o)$$

$$C_t = f_t * C_{t-1} + i_t * \widetilde{C_t}$$

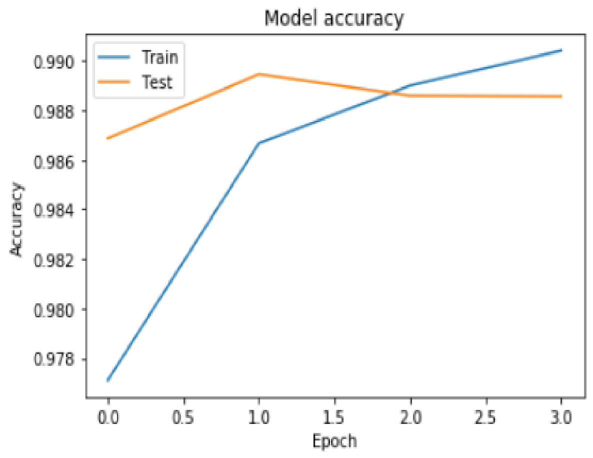
$$h_t = o_t * \tanh(C_t)$$

where:
$W_f.[h_{t-1}, x_t] <=> Wh_{t-1} + Ux_t$
with U the input weights and W the hidden weights

In the case of malware classification, our model is quite simple. Only two LSTM layer are used with both 32 cells and 'ReLU' activation function. Only the output layer uses the 'Sigmoid'. The model is compiled with the binary crossentropy and Adam optimizer.

The model have been trained on 4 epochs and reached and accuracy of 0.9904 during training.

We can see evolution of the training accuracy on the following graph:



After the training, the model has been evaluated on the testing set and showed a really good accuracy: 0.98974.

3.4 CNN-LSTM model

This model is a combination of the CNN model and the LSTM model. The model is composed of first a Convolution Neural Network which is stacked on a LSTM network. This model is quite efficient when it comes to mix images and time learning. In our case, as the LSTM and the CNN were really good to classify our data, it might be interesting to see if the combination can reach a higher accuracy.

Like the CNN, the CNN-LSTM model process the input in the same way. The goal of this processing is to represent each vector as an image.

Then the model is defined with first two Convolution blocks (16 and 32 filters). Each block uses MaxPooling and also swish activation function.

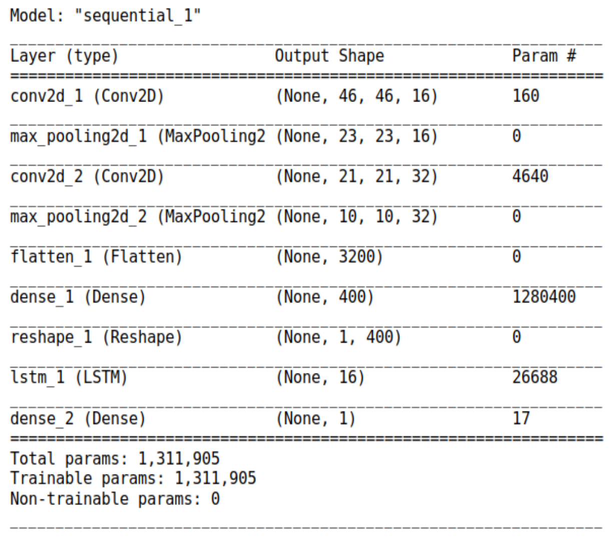
Swish activation function: The Swish activation function is defined as the following:

$$f(x) = \frac{x}{1+e^{-x}} = x * sigmoid(x)$$

This activation function has been created by Google to try to solve the dead neurons problem of the ReLU. This activation function showed better results than the ReLU especially on deep neural networks (between 40 and 50 hidden layers). In our case many tests have been done, and Swish showed better results.

Then, our model flatten the output of the CNN network to represent it as one vector. To make the transition between the CNN and the LSTM, the input pass through another layer (400 neurons with Swish) and then is reshaped. After this transition, there is the LSTM part of the model. Only one LSTM layer is used (16 cells) before the output layer.

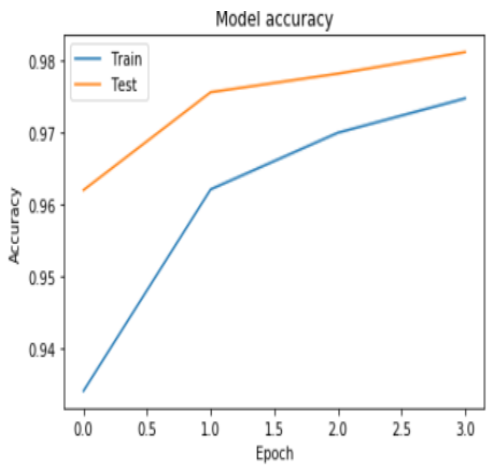
Following you can see the summary of the CNN-LSTM model:



```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 46, 46, 16)        160
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 23, 23, 16)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 21, 21, 32)        4640
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 10, 10, 32)        0
_________________________________________________________________
flatten_1 (Flatten)          (None, 3200)              0
_________________________________________________________________
dense_1 (Dense)              (None, 400)               1280400
_________________________________________________________________
reshape_1 (Reshape)          (None, 1, 400)            0
_________________________________________________________________
lstm_1 (LSTM)                (None, 16)                26688
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 17
=================================================================
Total params: 1,311,905
Trainable params: 1,311,905
Non-trainable params: 0
_________________________________________________________________
```

Following the definition, the model is compiled using Adam and Binary Crossentropy. The model has been trained on 4 epochs and has shown good training results on the data (acc: 0.9747).

We can see the evolution of training on the following graph:



We can see on the graph that the model is better on the validation set than during training which shows a good learning

Finally, after training the model has shown an accuracy of 0.98194 on the test set. This result is less good than the LSTM model accuracy but is still better than the CNN model result.

3.5 Autoencoder model

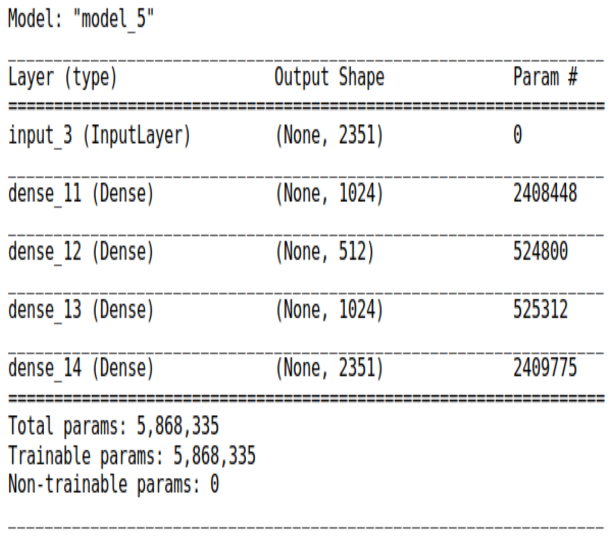
In this part, classification of ember dataset will be shown according to Autoencoder model for classification.

Autoencoders are special neural networks that compress the input data. Then, the data are put in a special state and decompressed in order to reconstruct the input. This type of neural networks are often used for dimension reduction, compression or for removing noise from images. In our case we will use it for classification. More precisely, an Autoencoder model we be trained separately from the final model, and then, the autoencoder will be stacked on one output layer used for classification.

Before defining the model of the autoencoder, the data have of been normalized using standard normalization.

The model of the Autoencoder is defined by 5 layers with respectively 2351, 1024, 512, 1024, 2351 neurons. So, the input data is transformed by compression into a 512 features vector and then decompress and rebuilt. Each layer, uses 'ReLU' activation function, except the last one that uses 'Sigmoid'. Moreover, to avoid overfitting, L1 regularization is used on the first layer of the autoencoder.

Here, we can see the summary of the autoencoder model:



```
Model: "model_5"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_3 (InputLayer)         (None, 2351)              0
_________________________________________________________________
dense_11 (Dense)             (None, 1024)              2408448
_________________________________________________________________
dense_12 (Dense)             (None, 512)               524800
_________________________________________________________________
dense_13 (Dense)             (None, 1024)              525312
_________________________________________________________________
dense_14 (Dense)             (None, 2351)              2409775
=================================================================
Total params: 5,868,335
Trainable params: 5,868,335
Non-trainable params: 0

_________________________________________________________________
```

Then, the autoencoder model is compiled and trained using Adam optimizer and mean squared error (MSE) as a loss function.

MSE: The mean squared error loss function is defined as:

$$loss = \frac{1}{n}\sum_{i=1}^{n}(y_i - \widehat{y_i})^2$$

Binary Crossentropy has also been tested, but MSE was more efficient.

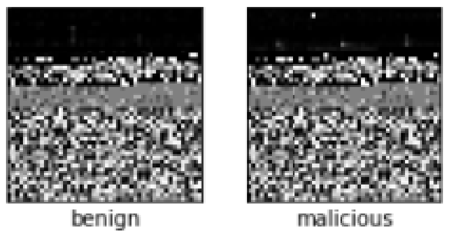
After the training of the autoencoder, the autoencoder model is stacked to an output layer (1 neuron with 'Sigmoid' as activation function) and the model is compiled using Adam optimizer and binary Crossentropy. Hence, the model is trained on 2 epochs only and finally reached an accuracy of 0.9449 during training. On the testing set, the model reached an accuracy of 0.95874.

3.6 CNN-Autoencoder model

The last model that been tested is the CNN autoencoder model. It is a mix between the CNN model and the Autoencoder model. The idea of this model is to increase the efficiency of the autoencoder model as we know that the CNN model is efficient.

In the case of the CNN-autoencoder, the data have pre-processed like for the CNN model, with Min Maz normalization and also the dimension have been reduced.

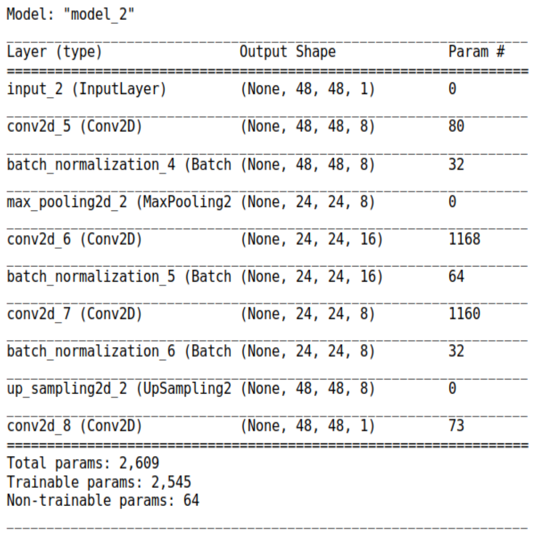
Here, the images have been represented in binary mode:



Like for the CNN model, it is impossible for human to make the difference.

So, this model is quite similar to the autoencoder model. It compresses our input and then rebuild it. The compression part of the network is represented by 2 Convolution block with respectively 8 and 16 filters. Only the first block uses Max Pooling (to reduce the dimension of the image). But each block uses batch normalization to avoid overfitting. The decompression part uses only two block of 8 and 1 filters. Only the first one uses batch normalization and up sampling (rebuild the image by increasing the number of features). Every blocks have the 'ReLU' activation function except the last one that uses 'Sigmoid'.

Here we can see the summary of our CNN autoencoder model:



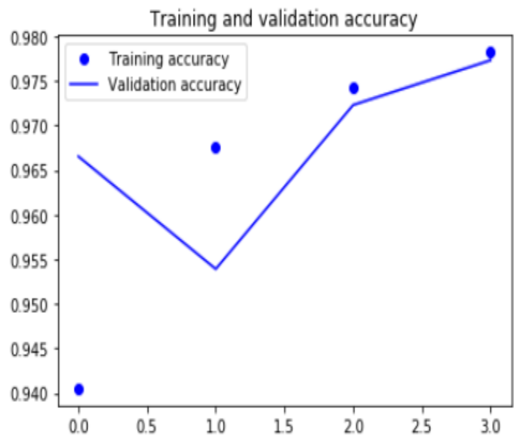
```
Model: "model_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         (None, 48, 48, 1)         0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 48, 48, 8)         80
_________________________________________________________________
batch_normalization_4 (Batch (None, 48, 48, 8)         32
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 24, 24, 8)         0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 24, 24, 16)        1168
_________________________________________________________________
batch_normalization_5 (Batch (None, 24, 24, 16)        64
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 24, 24, 8)         1160
_________________________________________________________________
batch_normalization_6 (Batch (None, 24, 24, 8)         32
_________________________________________________________________
up_sampling2d_2 (UpSampling2 (None, 48, 48, 8)         0
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 48, 48, 1)         73
=================================================================
Total params: 2,609
Trainable params: 2,545
Non-trainable params: 64

_________________________________________________________________
```

Then, the CNN autoencoder is trained during 10 epochs and using a batch size of 256. It also uses MSE loss function and Adam optimizer.

After, the training of the CNN autoencoder, the final classification model has been defined. The autoencoder has been stacked on one dense layer of 400 neurons (using 'ReLU') and one output layer of 1 neuron (using 'Sigmoid'). The model is compiled with the binary crossentropy and Adam optimizer. Regarding the training, the model has been trained on 4 epochs and has reached 0.9783 of accuracy.

We can see on the following picture the results on the training and on the validation set:



Finally, our model is efficient, the accuracy during testing is about 0.98504.

4 Results

After discussing about all these models, let's make a summary of the accuracy of each models in order to see which one was the most efficient to classify our data. In our case, each machine learning model (presented in part 2) have been trained on 10 epochs with partial fit methods and each deep learning model have been trained on 4 epochs, except for autoencoders that have been trained previously and the DNN model (8 epochs).

We can see on the following table, a summary of the different models and of the different results:

Model	Accuracy
Logistic Regression	0.96342
Linear SVClassifier	0.96482
DNN	0.91984
CNN	0.98006
LSTM	0.98974
CNN-LSTM	0.98194
Autoencoder	0.95874
CNN-Autoencoder	0.98504

As we can see, the machine learning models are similar in terms of accuracy about 0.96 both. The worst model, is the simple dense deep neural network. The best model is the LSTM, which has almost reached 99% of accuracy.

5 bibliography

OpenMP:
https://ccub.u-bourgogne.fr/dnum-ccub/IMG/pdf/openMP_Fortran_C.pdf
https://calcul.math.cnrs.fr/attachments/spip/Documents/Ecoles/PF-2011/Cours/openMP.pdf
https://computing.llnl.gov/tutorials/openMP/exercise.html
https://en.wikipedia.org/wiki/OpenMP
https://docs.microsoft.com/fr-fr/cpp/parallel/openmp/reference/openmp-directives?view=vs-2019#barrier

Vectorisation (AVX/AVX2):
http://www.idris.fr/media/formations/simd/idrissimd.pdf
https://software.intel.com/sites/default/files/m/d/4/1/d/8/Intro_to_Intel_AVX.pdf

DNN:
https://github.com/vinayakumarr/dnn-ember/blob/master/DNN-info.pdf
https://machinelearningmastery.com/how-to-reduce-generalization-error-in-deep-neural-net

CNN:
https://www.pyimagesearch.com/2018/12/31/keras-conv2d-and-convolutional-layers/
https://adventuresinmachinelearning.com/keras-tutorial-cnn-11-lines/
https://victorzhou.com/blog/keras-cnn-tutorial/

LSTM:
https://adventuresinmachinelearning.com/keras-lstm-tutorial/
https://gist.github.com/urigoren/b7cd138903fe86ec027e715d493451b4
https://www.tensorflow.org/guide/keras/rnn
http://users.cecs.anu.edu.au/~Tom.Gedeon/conf/ABCs2018/paper/ABCs2018_paper_92.pdf
https://keras.io/layers/recurrent/

CNN-LSTM:
https://machinelearningmastery.com/cnn-long-short-term-memory-networks/
https://machinelearningmastery.com/timedistributed-layer-for-long-short-term-memory-netw

Atoencoder:
https://radicalrafi.github.io/posts/autoencoders-as-classifiers/
https://towardsdatascience.com/extreme-rare-event-classification-using-autoencoders-in-k
https://ramhiser.com/post/2018-05-14-autoencoders-with-keras/

CNN-autoencoder:
https://www.datacamp.com/community/tutorials/autoencoder-classifier-python
https://ramhiser.com/post/2018-05-14-autoencoders-with-keras/

Logistic Regression:
https://towardsdatascience.com/introduction-to-logistic-regression-66248243c148

L1 and L2:
https://towardsdatascience.com/intuitions-on-l1-and-l2-regularisation-235f2db4c261

ML with dask:
https://www.analyticsvidhya.com/blog/2018/08/dask-big-datasets-machine_
learning-python/
https://towardsdatascience.com/speeding-up-your-algorithms-part-4-dask-7c6ed79994ef

Restricted Boltwmann Machines:
https://www.edureka.co/blog/restricted-boltzmann-machine-tutorial/
http://swoh.web.engr.illinois.edu/courses/IE598/handout/fall2016_
slide20.pdf
http://www.cs.toronto.edu/~hinton/absps/ncfast.pdf