Tikeng Notsawo Pascal Junior
pascalnotsawo@gmail.com
National Advanced School of Engineering of Yaounde Cameroun, department of computer engineering.

Sentiment analysis on the IMBD Large Movie Review Dataset

Abstract

In this assignment, we will build machine learning models to detect sentiments (i.e. detect whether a sentence is positive or negative) using IMBD Large Movie Review Dataset. In order to do so, we will first situate the context of our work, then we will construct and compare with a well-defined performance metric different approaches to solve the problem, while detailing the strengths and weaknesses of each of them.

I. Introduction

Sentiment analysis is a text analysis method that detects polarity (e.g. a positive or negative opinion) within text, whether a whole document, paragraph, sentence, or clause. The challenge of sentiment analysis is that, contrary to simple text classification, using an intuitive lexical-based classification doesn't work well. The reason is that among the overwhelming number of reviews, there are reviews which don't contain any intuitively subjective words and however express a strong opinion. Other reviews contain highly pejorative words and express a positive opinion (and reciprocally). The existing work on sentiment analysis can be classified from different points of views: technique used, view of the text, level of detail of text analysis, rating level, etc.

From a technical point of view, we identified machine learning, lexicon-based, statistical and rule-based approaches. In the context of machine learning, the last few years have seen different approaches to solve this problem, including statistical approaches (Shelke et al. [1], Farhadloo et al [2]), approaches based on **recurrent neural networks and their variants such as LSTM** (Subarno et al, [3]), approaches based on the **convolutional neural network** (Jan et al [4]) and those **based on the attention mechanism** (Delvin et al. [5]).

In this paper, we will exploit these different approaches and evaluate them with a well-defined performance metric, in order to select two to three best approaches among those tested. In addition to this, we will make our code available.

II. Related Work

Sentiment analysis is a traditional research hotspot in the NLP field (Wang and Manning 2012) [6]. Representative solutions for the categories listed in the introduction include the following: Sentiment Classification from Online Customer reviews Using Lexical Contextual Sentence Structure (Khan et al.) [7], Combining Lexicon and Learning based Approaches for Concept-Level Sentiment Analysis (Mudinas et al.)[8], Interdependent Latent Dirichlet Allocation (ILDA) introduced in 2011 by Moghaddam and Ester [9], A Joint Model of Feature Mining and Sentiment Analysis for Product Review Rating, introduced in 2011 by de Albornoz et al.[10], Opinion Digger, introduced in 2010 by Moghaddam and Ester [11], Latent Aspect Rating Analysis with a Model-based method, where the model is called the Latent Rating Regression (LRR) model [12] and was created by Wang et al. in 2010 [12].

With the release of online completions, abundant methods were proposed to explore the limits of previous models. Tang et al. (Tang et al., 2016a) [13] proposed to make use of bidirectional Long Short Term Memory (LSTM) (Hochreiter and Schmid huber, 1997) [14] to encode the sentence from the left and right to the aspect-term. This model primarily verifies the effectiveness of deep models for ABSA Tang et al. (Tang et al., 2016b) [15] then put forward a neural reasoning model in analogy to the memory network to perform the reasoning in many steps. There are also many other works dedicating to solve this task (Pan and Wang, 2018 [16]; Liu et al., 2018 [17]; Zhang and Liu, 2017 [18]).

Another related topic is semi-supervised learning for the text classification. Recently, Data augmentation methods (Xie et al., 2019 [19]; Berthelot et al., 2019 [20]) achieve a great success on low resource datasets. Moreover, A simple but efficient method is to use pre-trained modules, e.g., initializing the word embedding or bottom layers with pre-training. Word embedding technique has been widely used in NLP models, e.g., Glove (Pennington et al., 2014) [21] and ELMo (Peters et al., 2018)[22]. Recently, Bidirectional Encoder Representations from Transformer (BERT) (Devlin et al., 2018) [5] replaces the embedding layer to context dependent layer with the pre-trained bidirectional language model to capture the contextual representation.

VAE-based semi-supervised methods, on the other hand, are able to cooperate with various kinds of classifiers. VAE has been applied in many semi-supervised NLP tasks, ranging from text classification (Xu et al., 2017) [23] , relation extraction (Marcheggiani and Titov, 2016) [24] to sequence tagging (Chen et al., 2018). [25]

III. Background

III.1) Sentiment Analysis

Sentiment analysis and opinion mining is the field of computational study of people's opinion expressed in written language or text. Sentiment analysis brings together various research areas such as natural language processing, data mining and text mining, and is fast becoming of major importance to organizations as they integrate online commerce into their operations.
The input of the problem is a collection of written reviews about an object. The object could be a product or service and the goal is to discover people's opinions expressed in those written reviews. The input reviews are in the form of free text and do not have any structure (people can write whatever they like however they want). Dealing with unstructured data is a challenging problem.

Sentiment analysis can be done in different levels. In aspect-level sentiment analysis there are two tasks that need to be addressed. The first task is aspect identification which is the process of discovering those attributes of the object that people are commenting on. These attributes of the object are called aspects. The second task is sentiment identification which is the process of discovering people's opinions expressed about each one of the aspects. Aforementioned tasks can be solved in 2 separate steps or can be solved simultaneously.

III.2) RNN-LSTM

An RNN (Graves, 2013) [28] takes in sequence of words, X=$\{x_1, ..., x_T\}$, one at a time, and produces a hidden state, h, for each word. We use the RNN recurrently by feeding in the current word x_t as well as the hidden state from the previous word, h_{t-1}, to produce the next hidden state, h_t.

$$h_t = \mathbf{RNN}(x_t, h_{t-1}) \quad \textbf{Eq.1}$$

Once we have our final hidden state, h_t, (from feeding in the last word in the sequence, x_t) we feed it through a linear layer, f, (also known as a fully connected layer), to receive our predicted sentiment,

$$\hat{y} = f(h_t) \qquad \textbf{Eq.2.}$$

Below (Fig 1) shows an example sentence, with the RNN predicting zero, which indicates a negative sentiment. The RNN is shown in orange and the linear layer shown in silver. Note that we use the same RNN for every word, i.e. it has the same parameters. The initial hidden state, h_0, is a tensor initialized to all zeros.

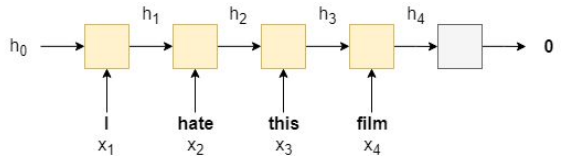

Fig 1 : Illustration of the functioning of a recurrent neural network.

Long Short-Term Memory (LSTM) (Hochreiter and Schmidhuber, 1997) [29] is a variant of RNN. Why is an LSTM better than a standard RNN? Standard RNNs suffer from the vanishing gradient problem. LSTMs overcome this by having an extra recurrent state called a cell, c - which can be thought of as the "memory" of the LSTM - and the use use multiple gates which control the flow of information into and out of the memory. We can simply think of the LSTM as a function of x_t, h_t and c_t, instead of just x_t and h_t.

$$h_t, c_t = LSTM(x_t, h_{t-1}, c_{t-1}) \qquad \textbf{Eq.3}$$

Thus, the model using an LSTM looks something like (with the embedding layers omitted):

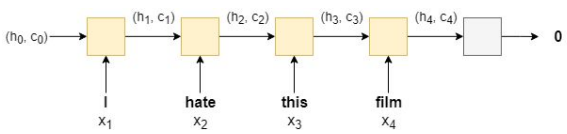

Fig 2 : Illustration of the functioning of LSTM.

The initial cell state, c_0, like the initial hidden state is initialized to a tensor of all zeros. The sentiment prediction is still, however, only made using the final hidden state, not the final cell state, i.e. $\hat{y} = f(h_t)$ (Eq.2)

III.3) CNN

Traditionally, CNNs (Fukushima, 1980) [26] are used to analyse images and are made up of one or more convolutional layers, followed by one or more linear layers. The convolutional layers use filters (also called kernels or receptive fields) which scan across an image and produce a processed version of the image. This processed version of the image can be fed into another convolutional layer or a linear layer. Each filter has a shape, e.g. a 3x3 filter covers a 3 pixel wide and 3 pixel high area of the image, and each element of the filter has a weight associated with it, the 3x3 filter would have 9 weights.
In traditional image processing these weights were specified by hand by engineers, however the main advantage of the convolutional layers in neural networks is that these weights are learned via backpropagation.
The intuitive idea behind learning the weights is that your convolutional layers act like feature extractors, extracting parts of the image that are most important for your CNN's goal, e.g. if using a CNN to detect faces in an image, the CNN may be looking for features such as the existence of a nose, mouth or a pair of eyes in the image.
So why use CNNs on text? In the same way that a 3x3 filter can look over a patch of an image, a 1x2 filter can look over a 2 sequential words in a piece of text, i.e. a bi-gram. The intuition here is that the appearance of certain bi-grams, tri-grams and n-grams within the review will be a good indication of the final sentiment.

III.4) Transformer-BERT

Previous work in sequence modeling used the common framework sequence-to-sequence (seq2seq) (Sutskever et al., 2014) [27], with techniques such as recurrent neural networks (RNNs) (Graves, 2013) [28] and long short-term memory (LSTM) (Hochreiter and Schmidhuber, 1997) [29]. The architecture of transformers is not based on RNNs but on attention mechanics (Vaswani et al., 2017) [30], which decides what sequences are important in each computational step. The encoder does not only map the input to a higher dimensional space vector, but also uses the important keywords as additional input to the decoder. This in turn improves the decoder because it has additional information such which sequences are important and which keywords give context to the sentence.

Pre-trained language models are providing a context to words, that have previously been learning the occurrence and representations of words from unannotated training data. Bidirectional encoder representations from transformers (BERT) is a pre-trained language model that is designed to consider the context of a word from both left and right side simultaneously (Devlin et al., 2019)[5]. While the concept is simple, it improves results at several NLP tasks such as sentiment analysis and question and answering systems. BERT can extract more context features from a sequence compared to training left and right separately, as other models such as ELMo do (Peters et al., 2018)[22].

The left and right pre-training of BERT is achieved using modified language model masks, called masked language model (MLM). The purpose of MLM is to mask a random word in a sentence with a small probability. When the model masks a word it replaces the word with a token [MASK]. The model later tries to predict the masked word by using the context from both left and right of the masked word with the help of transformers. In addition to left and right context extraction using MLM, BERT has an additional key objective which differs from previous works, namely next-sentence prediction.

III.5) Classification Performance Metrics

III.5.1) Confusion Matrix

In the field of machine learning and specifically the problem of statistical classification, a confusion matrix, also known as an error matrix, is a specific table layout that allows visualization of the performance of an algorithm, typically a supervised learning one (in unsupervised learning it is usually called a matching matrix). Each row of the matrix represents the instances in a predicted class while each column represents the instances in an actual class (or vice versa). The name stems from the fact that it makes it easy to see if the system is confusing two classes (i.e. commonly mislabeling one as another).(see Table 1)

		Actual class	
		Positive (P)	Negative (N)
Predicted class	Positive (P)	True Positive (TP)	False Positive (FP)
	Negative (N)	False Negative (FN)	True Negative (TN)

Table 1 : confusion matrix in the case of binary classification

III.5.2) Classification Accuracy

Classification accuracy is the number of correct predictions made divided by the total number of

predictions made (multiplied by 100 to turn it into a percentage) (Eq. 4).

$$ACC = \frac{TP+TN}{P+N} = \frac{TP+TN}{TP+TN+FP+FN} \quad \text{Eq.4}$$

When we use accuracy, we assign equal cost to false positives and false negatives. When that data set is imbalanced - say it has 99% of instances in one class and only 1 % in the other - there is a great way to lower the cost. Predict that every instance belongs to the majority class, get accuracy of 99% and go home early.
The problem starts when the actual costs that we assign to every error are not equal. If we deal with a rare but fatal disease, the cost of failing to diagnose the disease of a sick person is much higher than the cost of sending a healthy person to more tests.

III.5.3) Precision

Precision is the number of True Positives divided by the number of True Positives and False Positives. Put another way, it is the number of positive predictions divided by the total number of positive class values predicted. It is also called the Positive Predictive Value (PPV). Precision can be thought of as a measure of a classifiers exactness. A low precision can also indicate a large number of False Positives.

$$PPV = \frac{TP}{TP+FP} \quad \text{Eq.5}$$

III.5.4) Recall

Recall is the number of True Positives divided by the number of True Positives and the number of False Negatives. Put another way it is the number of positive predictions divided by the number of positive class values in the test data. It is also called Sensitivity or the True Positive Rate. Recall can be thought of as a measure of a classifiers completeness. A low recall indicates many False Negatives.

$$R = \frac{TP}{TP+FN} \quad \text{Eq.6}$$

III.5.5) F1 score

F1 Score (Eq.7) is needed when you want to seek a balance between Precision and Recall. So what is the difference between F1 Score and Accuracy then? We have previously seen that accuracy can be largely contributed by a large number of True Negatives which in most business circumstances, we do not focus on much whereas False Negative and False Positive usually has business costs (tangible & intangible) thus F1 Score might be a better measure to use if we need to seek a balance between Precision and Recall AND there is an uneven class distribution (large number of Actual Negatives).

$$F1 - score = 2 \times \frac{PPV \times R}{PPV + R} = \frac{2 \times TP}{2 \times TP + FN + FP} \quad \text{Eq.7}$$

IV. Methodology

IV.1) Data

We used Pytorch and TorchText.
With TorchText, the ratio for splitting the IMDB dataset originates from the data itself, as 25,000 reviews are provided for training and 25,000 for testing. From the dataset website: TorchText provide a set of 25,000 highly polar movie reviews for training, and 25,000 for testing.
We then split the training data : 5000 for validation and 20000 for training (ratio of 0.8). By splitting our data, we also set the **random seed** for reproducibility, making sure that we get the same train/validation/test split every time.

One of the main concepts of TorchText is the `Field`. These define how your data should be processed. In our sentiment classification task the data consists of both the raw string of the review and the sentiment, either "pos" or "neg". The parameters of a `Field` specify how the data should be processed.

We use the `TEXT` field to define how the review should be processed, and the `LABEL` field to process the sentiment.

IV.2) Models

We experimented during our work with three main models: **recurrent models, convolutional models and models based completely on the attention mechanism**. All our classes are implemented in the model.py scripts of the github repository and inherits from the torch.nn.Module class.

IV.2.1) RNN-LSTM

a) RNN

In the RNN class of model.py, we define the layers of the module. Our three layers are an embedding layer, the RNN, and a linear layer. All layers have their parameters initialized to random values, unless explicitly specified.

- The embedding layer is used to transform our sparse one-hot vector (sparse as most of the elements are 0) into a dense embedding vector (dense as the dimensionality is a lot smaller and all the elements are real numbers). This embedding layer is simply a single fully connected layer. As well as reducing the dimensionality of the input to the RNN, there is the theory that words which have similar impact on the sentiment of the review are mapped close together in this dense vector space.
- The RNN layer is our RNN which takes in our dense vector and the previous hidden state h_{t-1}, which it uses to calculate the next hidden state, h_t.
- Finally, the linear layer takes the final hidden state and feeds it through a fully connected layer, $f(h_t)$, transforming it to the correct output dimension.

The forward method is called when we feed examples into our model.
Each batch, text, is a tensor of size `[sentence length, batch size]`. That is a batch of sentences, each having each word converted into a one-hot vector.
You may notice that this tensor should have another dimension due to the one-hot vectors, however PyTorch conveniently stores a one-hot vector as it's index value, i.e. the tensor representing a sentence is just a tensor of the indexes for each token in that sentence. The act of converting a list of tokens into a list of indexes is commonly called numericalizing.
The input batch is then passed through the embedding layer to get embedded, which gives us a dense vector representation of our sentences. embedded is a tensor of size `[sentence length, batch size, embedding dim]`. embedded is then fed into the RNN. In some frameworks you must feed the initial hidden state, h_0, into the RNN, however in PyTorch, if no initial hidden state is passed as an argument it defaults to a tensor of all zeros.
The RNN returns 2 tensors, output of size `[sentence length, batch size, hidden dim]` and hidden of size `[1, batch size, hidden dim]`. output is the concatenation of the hidden state from every time step, whereas hidden is simply the final hidden state. We verify this using the assert statement. Note the squeeze method, which is used to remove a dimension of size 1.
Finally, we feed the last hidden state, hidden, through the linear layer, fc, to produce a prediction.

The instance of our RNN class have this parameters :
- The input dimension is the dimension of the one-hot vectors, which is equal to the vocabulary size.

- The embedding dimension is the size of the dense word vectors : We have chosen 100 depends on the size of our vocabulary.
- The hidden dimension is the size of the hidden states: We have chosen 256 depends on the vocabulary size, the size of the dense vectors and the complexity of the task.
- The output dimension is usually the number of classes, however in the case of only 2 classes the output value is between 0 and 1 and thus can be 1-dimensional, i.e. a single scalar real number.

- Forces

Easy to train.

- Weaknesses

Vanishing gradient problem
This model produces the lowest results compared to the other models studied.
That's why we proposed the LSTM as our first model, as instructed in the exercise.

b) LSTM

The implementation details are identical to those of RNN, except that we replace the RNN layer with LSTM. Minor changes made during data processing, training, validation or testing are better explained in the github repository.

- We used packed padded sequences, which will make our LSTM only process the non-padded elements of our sequence, and for any padded element the output will be a zero tensor. To use packed padded sequences, we have to tell the LSTM how long the actual sequences are. We do this by setting `include_lengths = True` for our TEXT field. This will cause batch.text to now be a tuple with the first element being our sentence (a numericalized tensor that has been padded) and the second element being the actual lengths of our sentences.
- Instead of having our word embeddings initialized randomly, they are initialized with these pre-trained vectors. We get these vectors simply by specifying which vectors we want and passing it as an argument to build_vocab. TorchText handles downloading the vectors and associating them with the correct words in our vocabulary. We use the `glove.6B.100d` vectors. glove is the algorithm used to calculate the vectors. 6B indicates these vectors were trained on 6 billion tokens and 100d indicates these vectors are 100-dimensional.

Our LSTM instance have this parameters :

- vocab_size = len(dataset["TEXT"].vocab)
- embedding_dim = 100
- hidden_dim = 256
- output_dim = 1
- n_layers = 2
- bidirectional = True
- dropout = 0.5
- pad_idx = dataset["TEXT"].vocab.stoi[dataset["TEXT"].pad_token]

- Forces :

overcomes the vanishing gradient problem.

- Weaknesses :

Consumes too much memory during training, training takes longer.

IV.2.2) CNN

We implement the convolutional layers with nn.Conv2d. The in_channels argument is the number of channels in your image going into the convolutional layer. In actual images this is usually 3 (one channel for each of the red, blue and green channels), however when using text we only have a single channel, the text itself. The out_channels is the number of filters and the kernel_size is the size of the filters. Each of our kernel_size is going to be `[n x emb_dim]` where n is the size of the n-grams.

In PyTorch, RNNs want the input with the batch dimension second, whereas CNNs want the batch dimension first - we do not have to permute the data here as we have already set `batch_first = True` in our `TEXT` field. We then pass the sentence through an embedding layer to get our embeddings. The second dimension of the input into a nn.Conv2d layer must be the channel dimension. As text technically does not have a channel dimension, we unsqueeze our tensor to create one. This matches with our `in_channels=1` in the initialization of our convolutional layers.

We then pass the tensors through the convolutional and pooling layers, using the ReLU activation function after the convolutional layers. Another nice feature of the pooling layers is that they handle sentences of different lengths. The size of the output of the convolutional layer is dependent on the size of the input to it, and different batches contain sentences of different lengths. Without the max pooling layer the input to our linear layer would depend on the size of the input sentence (not what we want). One option to rectify this would be to trim/pad all sentences to the same length, however with the max pooling layer we always know the input to the linear layer will be the total number of filters.
There an exception to this if your sentence(s) are shorter than the largest filter used. You will then have to pad your sentences to the length of the largest filter. In the IMDb data there are no reviews only 5 words long so we don't have to worry about that, but you will if you are using your own data.

Finally, we perform dropout on the concatenated filter outputs and then pass them through a linear layer to make our predictions.

Our CNN instance have this parameters :

- vocab_size = len(TEXT.vocab)
- embedding_dim = 100
- n_filters = 100
- filter_sizes = [3,4,5]
- output_dim = 1,
- dropout = 0.5
- pad_idx = TEXT.vocab.stoi[TEXT.pad_token]

- Forces :

Easy to train as it converges quickly

- Weaknesses :

Its major inconvenience is that it is not adapted to the text, using it on text is a bit like using a hammer to kill a fly when one could be satisfied with a clap of the hands.

IV.2.2) Transformer/BERT

Transformer models are considerably larger than anything else covered above. As such we are going to use the transformers library to get pre-trained transformers and use them as our embedding layers. We froze (not train) the transformer and only train the remainder of the model which learns from the representations produced by the transformer. In this case we used a multi-layer bi-directional GRU, however any model can learn from these representations.

Our multi-layer bi-directional GRU have this parameters :

- hidden_dim = 256
- output_dim = 1
- n_layers = 2
- bidirectional = True
- dropout = 0.25

- Forces :

Capture long-range dependencies and converges quickly. Produces better results than other models.

- Weaknesses :

Training takes more time than with previous models.

IV.3) Experiment settings

a) Evaluation metrics

When we use accuracy, we assign equal cost to false positives and false negatives. When that data set is imbalanced - say it has 99% of instances in one class and only 1 % in the other - there is a great way to lower the cost. Predict that every instance belongs to the majority class, get accuracy of 99% and go home early.
The problem starts when the actual costs that we assign to every error are not equal. If we deal with a rare but fatal disease, the cost of failing to diagnose the disease of a sick person is much higher than the cost of sending a healthy person to more tests.
But in our case we do not face this problem. **So we used the accuracy as our performance metric**. However, we will also present the F1-score (to combine recall and precision) and the confusion matrix to better understand our model.

b) optimizer

We used SGD for the RNN model and Adam for the other models (i.e. the three we chose as requested in the exercise): LSTM, CNN and BERT-GRU.
SGD updates all parameters with the same learning rate and choosing this learning rate can be tricky.
Adam adapts the learning rate for each parameter, giving parameters that are updated more frequently lower learning rates and parameters that are updated infrequently higher learning rates.
Also note how we do not have to provide an initial learning rate for Adam as PyTorch specifies a sensibile default initial learning rate.

c) Loss function

In PyTorch the loss function is commonly called a criterion. The loss function here is binary cross entropy with logits. Our model currently outputs an unbound real number. As our labels are either 0 or 1, we want to restrict the predictions to a number between 0 and 1. We do this using the sigmoid or logit functions.
We then use this this bound scalar to calculate the loss using binary cross entropy.
The BCEWithLogitsLoss criterion carries out both the sigmoid and the binary cross entropy steps.

d) Regularization

Although we've added improvements to our models (LSTM, CNN and BERT-GRU), each one adds additional parameters. Without going into overfitting into too much detail, the more parameters we have in our model, the higher the probability that our model will overfit (memorize the training data, causing a low training error but high validation/testing error, i.e. poor generalization to new, unseen examples). To combat this, we use regularization. More specifically, we use a method of regularization called dropout.
Dropout works by randomly dropping out (setting to 0) neurons in a layer during a forward pass. The probability that each neuron is dropped out is set by a hyperparameter and each neuron with dropout applied is considered independently. One theory about why dropout works is that a model with parameters dropped out can be seen as a "weaker" (less parameters) model. The predictions from all these "weaker" models (one for each forward pass) get averaged together within in the parameters of the model. Thus, our one model can be thought of as an ensemble of weaker models, none of which are over-parameterized and thus should not overfit.

e) Hardware and Schedule

The experiments were conducted on a google cloud virtual machine instance:
NVIDIA Tesla T4 × 1, 1 GPU, 8 vCPUs, 52 GB RAM.
Below are the training times of our models, and the corresponding number of epochs.

Models	Training times	Numbers of epochs
RNN	01 min 57 s	09
LSTM	06 min 30 s	10
CNN	03 min 50 s	10
CNN1d	02 min 00 s	10
BERT-GRU	18 min 18 s	01

f) Training Data and Batching

The training data has been described above. We used a batch size of 128 during our experiments.

V. Results

V.1) Quantitatives Results

We present here the results obtained on the test set.

Models	Accuracy (%)	F1-Score (%)
SGD or Linear support vector machines	50.69	67.00
Multinomial Naive Bayes	75.10	75.00
Logistic Regression	75.12	75.00
CNN	85.62	85.61
LSTM	86.86	86.85
BERT-GRU	**89.81**	**89.92**

Table 2 : performance comparisons of our models with basic models

In Fig. 3, we present the short evolution of loss and accuracy during our trainings.

V.2) Qualitatives Results

Here we have taken two examples of reviews: one positive and one negative. And we can see in the following table the different values produced by our models.

- example negative review (NR)

This film is too scary, too much gunfire and blood spilled inside. I can't watch bad movies like this anymore.

- example positive review (PR)

Among these actors, I prefer the most romantic one, he likes what he does, is positive about chess and knows how to celebrate victories.

Models	sentiments	
	NR	PR
LSTM	0.01	0.11
CNN	0.18	0.85
BERT-GRU	0.15	0.95

Table 3 : some predictions made by our models

V.3) Ablation Studies

a) Effect of training data size.

With a training size of 10,000 examples, we obtained the results not very far from those obtained with 20,000. This proves not only the robustness of our models, but also and above all

the fact that the training data is overall very representative of the data set.

b) Results obtained with the variants of our models

CNN1d = 1-dimensional convolutional layers, where the embedding dimension is the depth of the filter and the number of tokens in the sentence is the width.

Models	Accuracy (%)	F1-Score (%)
RNN	**47.86**	**34.36**
CNN1d	**85.95**	**85.96**

Table 4 : performance of RNN and CNN1d models

So we see two things:
- The effect of the vanishing gradient problem on our results: the LSTM outperforms the RNN in accuracy by 39.00% and F1-score of 52.49%.
- We notice that the CNN1d produces better results than our previous CNN, which proves the importance of always adapting the parameters of our models to the problem we want to solve.
These differences can also be seen in the predictions made.

Models	sentiments	
	NR	PR
RNN	0.52	0.53
CNN1d	0.07	0.70

Table 5 : some predictions made by our variant models

VI. Conclusion

In this assignment, we were looking at sentiment analysis on the IMBD Large Movie Review Dataset. We began by presenting previous research that has addressed this topic while setting the context for our work, then we presented our models and the results they produced. We found that neural models produce better results than statistical models.

Références

[1] Nilesh Shelke, Shriniwas Deshpande, Vilas Thakare. Statistical Approach for Sentiment Analysis of Product Reviews. International Journal of Computer Science and Network, Volume 5, Issue 3, June 2016.

[2] Farhadloo, Mohsen & Rolland, Erik. (2013). Multi-Class Sentiment Analysis with Clustering and Score Representation. Proceedings - IEEE 13th International Conference on Data Mining Workshops, ICDMW 2013. 904-912. 10.1109/ICDMW.2013.63.

[3] Pal, Subarno & Ghosh, Soumadip & Nag, Amitava. (2018). Sentiment Analysis in the Light of LSTM Recurrent Neural Networks. International Journal of Synthetic Emotions. 9. 33-39. 10.4018/IJSE.2018010103.

[4] Jan Deriu & Mark Cieliebak. Sentiment Analysis using Convolutional Neural Networks with Multi-Task Training and Distant Supervision on Italian Tweets.

[5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. https://arxiv.org/abs/1810.04805.

[6] Wang, Sida and Manning, Christopher. Baselines and Bigrams: Simple, Good Sentiment

and Topic Classification. Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)

[7] Aurangzeb Khan, Baharum Baharudin, and Khairullah Khan. Sentiment classification from online customer reviews using lexical contextual sentence structure. In Software Engineering and Computer Systems, pages 317-331. Springer, 2011.

[8] Andrius Mudinas, Dell Zhang, and Mark Levene. Combining lexicon and learning based approaches for concept-level sentiment analysis. In Proceedings of the First International Workshop on Issues of Sentiment Discovery and Opinion Mining, page 5. ACM, 2012.

[9] Samaneh Moghaddam and Martin Ester. Ilda: interdependent lda model for learning latent aspects and their ratings from online product reviews. In Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval, pages 665-674. ACM, 2011.

[10] Jorge Carrillo de Albornoz, Laura Plaza, Pablo Gerv´as, and Alberto D´ıaz. A joint model of feature mining and sentiment analysis for product review rating. In Advances in Information Retrieval, pages 55-66. Springer, 2011.

[11] Samaneh Moghaddam and Martin Ester. Opinion digger: an unsupervised opinion miner from unstructured product reviews. In Proceedings of the 19th ACM international conference on Information and knowledge management, pages 1825-1828. ACM, 2010.

[12] Hongning Wang, Yue Lu, and Chengxiang Zhai. Latent aspect rating analysis on review text data: a rating regression approach. In Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining, pages 783-792. ACM, 2010.

[13] Duyu Tang, Bing Qin, Xiaocheng Feng, and Ting Liu. 2016a. Effective lstms for target-dependent sentiment classification. In COLING 2016, 26th International Conference on Computational Linguistics, Proceedings of the Conference: Technical Papers, December 11-16, 2016, Osaka, Japan, pages 3298–3307.

[14] Sepp Hochreiter and Jurgen Schmidhuber. 1997. Long short-term memory. Neural Computation, 9(8):1735–1780.

[15] Duyu Tang, Bing Qin, and Ting Liu. 2016b. Aspect level sentiment classification with deep memory network. In Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, EMNLP 2016, Austin, Texas, USA, November 1-4, 2016, pages 214–224.

[16] Sinno Jialin Pan and Wenya Wang. 2018. Recursive neural structural correspondence network for crossdomain aspect and opinion co-extraction. pages 2171–2181.

[17] Fei Liu, Trevor Cohn, and Timothy Baldwin. 2018. Recurrent entity networks with delayed memory update for targeted aspect-based sentiment analysis. pages 278–283.

[18] Yue Zhang and Jiang Ming Liu. 2017. Attention modeling for targeted sentiment. In Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics, EACL 2017, Valencia, Spain, April 3-7, 2017, Volume 2: Short Papers, pages 572–577.

[19] Qizhe Xie, Zihang Dai, Eduard Hovy, Minh-Thang Luong, and Quoc V Le. 2019. Unsupervised data augmentation. arXiv preprint arXiv:1904.12848.

[20] David Berthelot, Nicholas Carlini, Ian Goodfellow, Nicolas Papernot, Avital Oliver, and Colin Raffel. 2019. Mixmatch: A holistic approach to semi-supervised learning. arXiv preprint arXiv:1905.02249.

[21] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. Glove: Global vectors for word representation. In Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL, pages 1532–1543.

[22] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. 2018. Deep contextualized word representations. In Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2018, New Orleans, Louisiana, USA, June 1-6, 2018, Volume 1 (Long Papers), pages 2227–2237.

[23] Weidi Xu, Haoze Sun, Chao Deng, and Ying Tan. 2017. Variational autoencoder for semi-supervised text classification. In Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA., pages 3358–3364

[24] Diego Marcheggiani and Ivan Titov. 2016. Discretestate variational autoencoders for joint discovery and factorization of relations. TACL, 4:231–244.

[25] Mingda Chen, Qingming Tang, Karen Livescu, and Kevin Gimpel. 2018. Variational sequential labelers for semi-supervised learning. In Proceedings of the2018 Conference on Empirical Methods in Natural Language Processing, pages 215–226.

[26] Dr. Kunihiko Fukushima, Kansai University, Japan. Neocognitron. http://www.scholarpedia.org/article/Neocognitron.

[27] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to sequence learning with neural networks. In Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2, NIPS'14, pages 3104–3112, Cambridge, MA, USA. MIT Press.

[28] Alex Graves. 2013. Generating sequences with recurrent neural networks. CoRR, abs/1308.0850.

[29] Sepp Hochreiter and Jurgen Schmidhuber. 1997. Long short-term memory. Neural Comput., 9(8):1735–1780.

[30] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N.

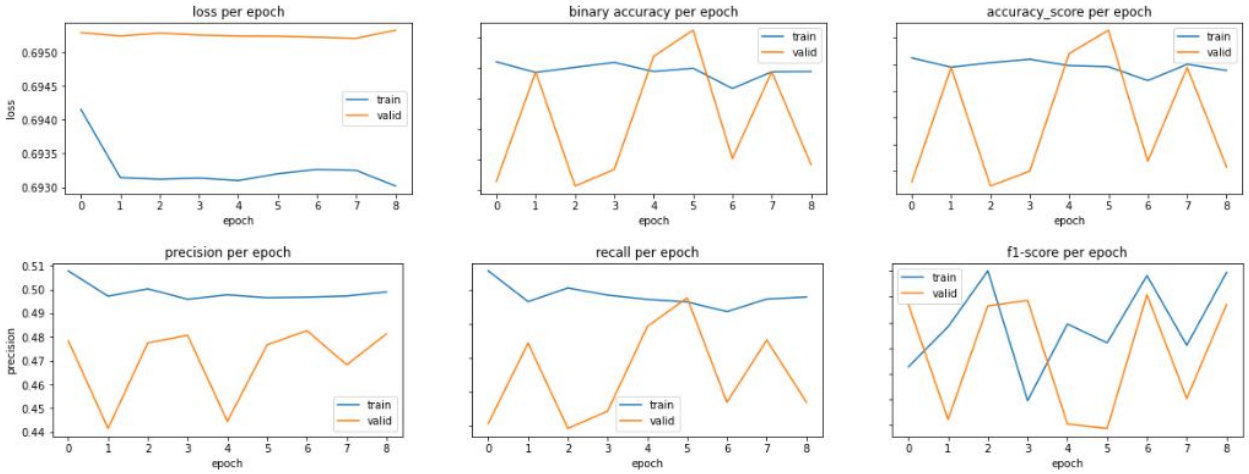

Fig 3.1 : RNN

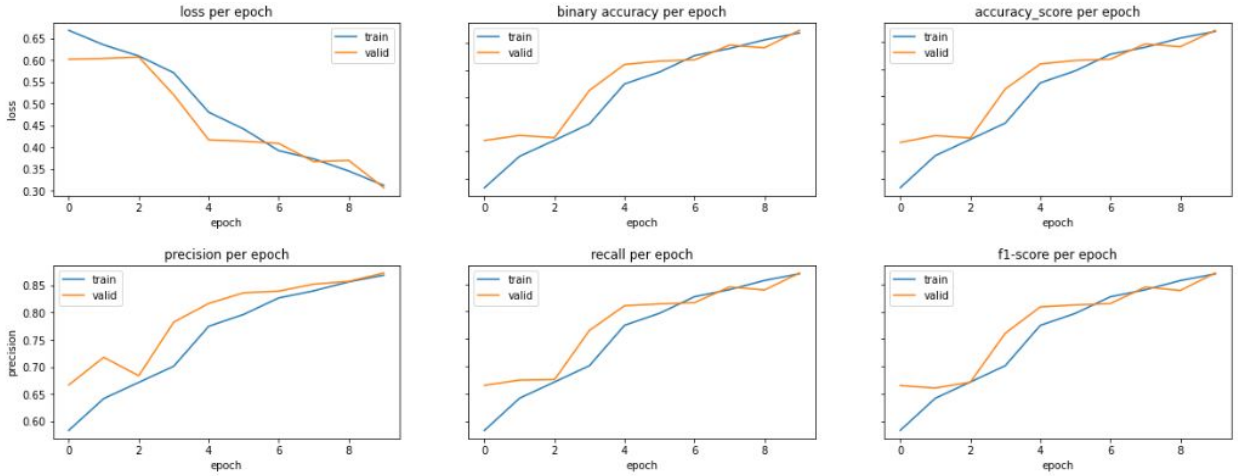

Fig 3.2 : LSTM

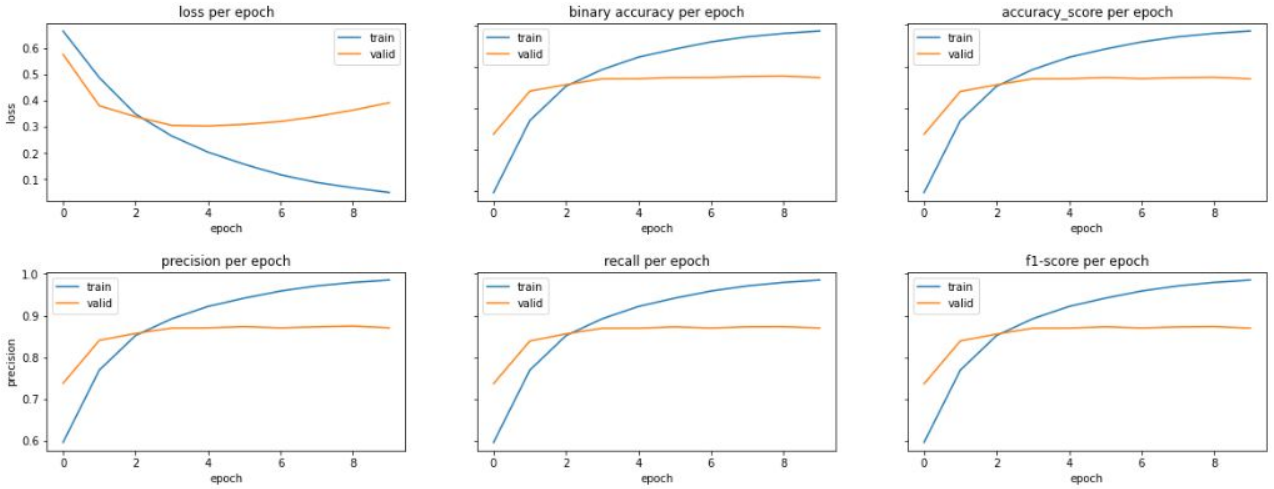

Fig 3.4 : CNN

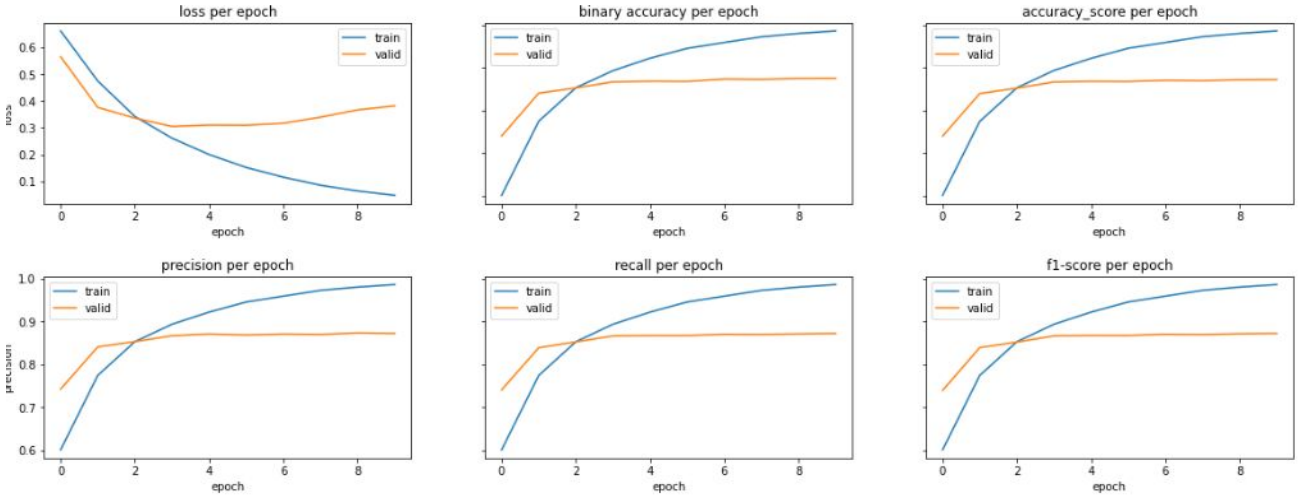

Fig 3.5 : CNN1d

Fig3 : evolution of loss, accuracy, precision, recall and f1-score during our trainings.