UNIVERSITE PARIS-SUD

Master Informatique 1ere Année

Année 2017-2018

Rapport de TER Réseau de Neurones

Par

Gao Jiaxin

Xue Di

Classification de tweets

Enseignant: ALEXANDRE ALLAUZEN, Université Paris-Sud

I-Introduction

Dans le cadre de ce travail, nous avons choisi de baser notre étude sur le traitement et l'analyse de séquences de symboles. Ces données sur lesquelles nous allons travaillé sont des tweets provenant directement de vrais exemples trouvés sur le site « www.twitter.com ». Cet ensemble de données est disponible à l'adresse suivante : http://thinknook.com/wp-content/uploads/2012/09/Sentiment-Analysis-Dataset.zip.

Le développement de ce projet à été réalisé avec le langage Python et les réseaux de neurones ont été effectué à l'aide de l'API « Torch » et de son interface « PyTorch ».

La reste de rapport est organisée comme suit : dans la deuxième partie, nous vous présentons les demarchés concrètes: nous vous expliquons chaque étape de traitement de données et à quoi cela sert, nous vous présentons des modèles différentes que nous avons réalisées dans ce projet, puis, analyser les avantages et les inconvénients de chaque modèle. Enfin, nous vous donnons une conclusion, des difficultés que nous avons rencontrées et ce que nous avons appris grâce à ce projet.

II-Demarchés concrètes

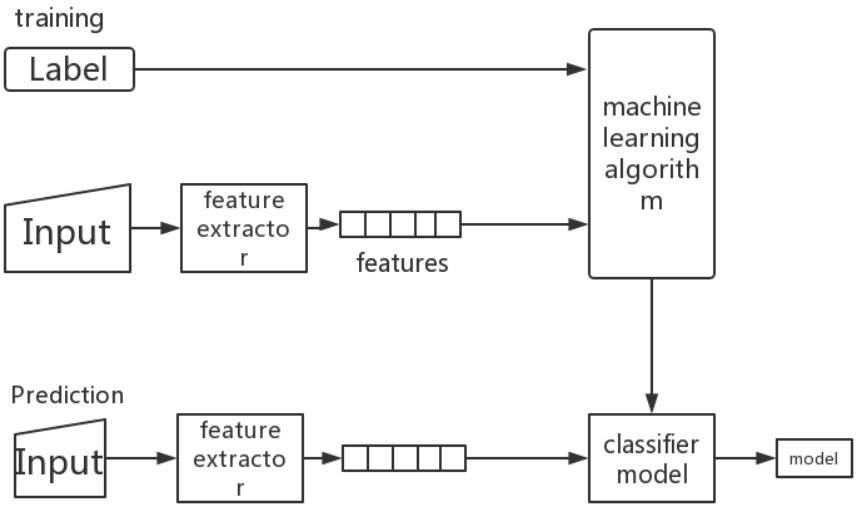

Figure 1: Processus d'apprentissage automatique et structure
Reference:《Natural Language Processing with Python》

1. La tâche

Notre tâche consiste à construire des classificateurs qui utilisent différents modèles pour former les données, dans l'espoir d'obtenir une plus grande précision.

L'idée de la méthode d'apprentissage automatique est de sélectionner d'abord une partie du texte exprimant des émotions positives et une partie du texte exprimant des émotions négatives, et d'utiliser des méthodes d'apprentissage automatique pour s'entraîner à l'obtention d'un classificateur d'émotions. À travers ce classificateur émotionnel, tous les textes sont classés positivement et négativement. La classification finale peut donner au texte une catégorie de 0 ou 1, et elle peut également donner une valeur de probabilité.

Nous allons par conséquent réaliser de l'apprentissage supervisé afin de répondre au mieux à cette tâche. Les modèles qui seront utilisés durant toute cette analyse sont, comme le spécifie cette unité d'enseignement, des réseaux de neurones.

Avant d'extraire des "features" des données, nous devons d'abord effectuer un prétraitement des données.

2. Traitement de données

En effet, la base de données sur laquelle nous avons travaillé regroupe 1 578 627 tweets différents. Chacun étant bien évidemment étiqueté avec le sentiment qui lui correspond.

Un problème complexe à prendre en compte, concerne la «propreté» des ***tweets*** sur lesquelles on souhaite travailler. Par exemple, les fautes d'orthographe, une ponctuation excessive, la présence de smileys, de divers caractères, ainsi que la récurrence abusive de même lettres en font des séquences qui ne sont pas exploitables directement par un modèle d'apprentissage.
Il est donc importante d'effectuer un pré-traitement sur ces données si l'on souhaite de bons résultats. Voici quelques exemples pour nous bien comprendre les Tweets:

1. 64,1,Sentiment140, i get off work sooooon! i miss cody booo. haven't seen him i n foreverr!
2. 61,0,Sentiment140, hate u ... leysh t9ar5 ... =(((((((..
3. 82,0,Sentiment140, I wish I could go to T4 On The Beach :'(Would be great t o see @Shontelle_Layne & @DanMerriweather

Nous téléchargeons l'ensemble de données.
Nous chargé un partie de l'ensemble de données. Pendant le chargement nous devons d'abord prétraiter les données et supprimer certains symboles étranges, par exemple, filtrez certains caractères non ASCII, convertissez les majuscules en minuscules, etc.
Ce que nous devons encore faire est de supprimer les mots courants, ici, "stopwords" est basé sur de la bibliothèque "nltk.corpus";

```
from nltk.corpus import stopwords
stop_words = set(stopwords.words('english'))
data_clean=[]
for i in data_set:
    t = list(filter(lambda l: l not in stop_words, i))
    data_clean.append(t)
print(len(data_clean))
```

```
99999
```

Figure 2: supprimer les “stopwords” en utilisant le bibliothèque NLTK.

```
In [5]: # compter les mots

def wordCount(listdephrases):
    c = {}
    for phrase in listdephrases:
        for mot in phrase:
            if mot in c:
                c[mot]+=1
            else:
                c[mot]=1
    return c

comptes = wordCount(data_clean)
print (len(comptes))
```

```
103749
```

Figure 3: les mots utiles sont 103749 (la taille vocabulaire)

Après cela, nous filtrons les données, et nous filtrons en fonction du nombre d'occurrences de chaque mot. D'abord sélectionner le nombre de mots dont le nombre d'occurrences est plus de (0, 1, 2, 5 et 10) respectivement. Notre vocabulaire original est les mots qui apparaît plus de 0 fois. Nous comparons ces chiffres et trouvons que le nombre de mots avec une fréquence plus de 2 est relativement raisonnable pour le modélisation. Si l'ensemble de données est trop grand, la quantité de calcul est trop grande; L'ensemble de données est trop petit et le résultat n'est pas représentatif.

En plus, nous devons toujours nous débarrasser de certaines phrases vides.
C'est tout le prétraitement des données.

En effectuant une série de pré-traitement sur les données d'origine, nous avons obtenu des données utiles;

Ensuite, établir les trois modèles et appliquer les à l'ensemble “train data” et à l'ensemble de “test data” séparément pour obtenir le taux correct de leur ensembles respectifs.
Nous pouvons également filtrer certains ensembles de données en définissant le taille de paramètre “mini-batch”, bien sûr, si l'ensemble de données est relativement petit, nous pouvons utiliser directement l'ensemble de données.

3. Application et analyse de modèle

Dans tous les trois modèles, le nombre de training data est 30000, le nombre de test data est 10000.

a. Modèle 1

```
In [9]: class CBOW_classifier(nn.Module): # un layer

            def __init__(self, vocab_size, embedding_dim):
                super(CBOW_classifier, self).__init__()
                self.embeddings = nn.Embedding(vocab_size, embedding_dim)
                self.linear1 = nn.Linear(embedding_dim, 1)
            def forward(self, inputs):
                out = self.linear1(self.makeCBOW(inputs))
                return F.sigmoid(out)


            def makeCBOW(self,input_idx):
                input_var =ag.Variable(th.LongTensor(input_idx))
                embeds =self.embeddings(input_var).sum(dim=0)
                return embeds
```

Figure 4: le modèle qui eu deux layer(embedding, layer)

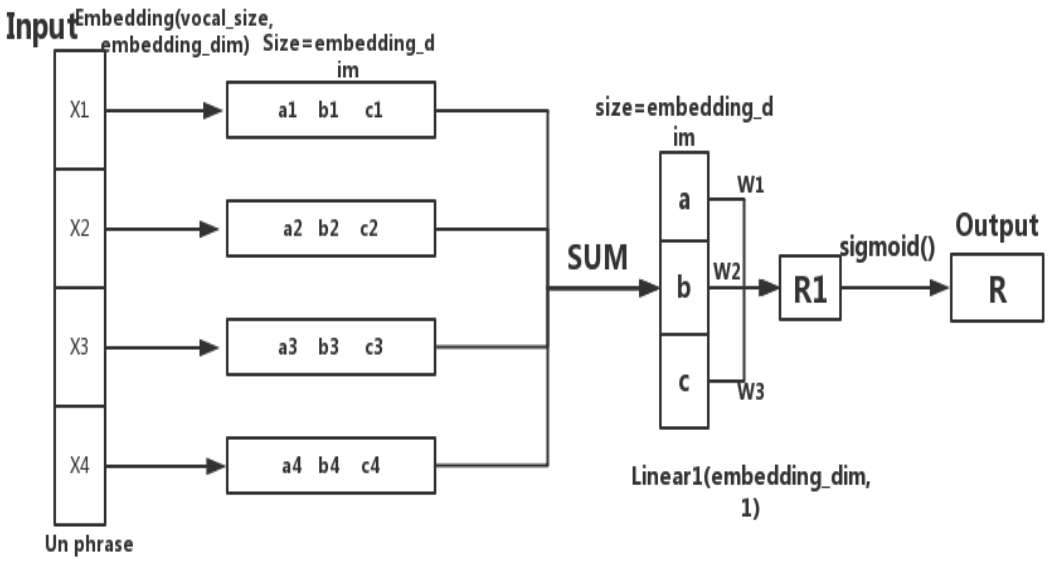

Figure 5: le figure de la prèmiere modèle(normale)

```
optimizer = th.optim.Adam(model.parameters(), lr=learning_rate)
```

Figure 6: la fonction pour l'optimization

1). D‘abord, on fait **l'initialisation du module**, exécutez (__init__ ()). Il définit la structure du réseau neuronal.
2). **nn.Embedding (vocab_size, embedding_dim)**, où vocab_size représente la taille du vocabulaire, (embedding_dim) est un chiffre qui représente le nombre de dimensions pour chaque mots utiles.
3). nn.Linear(): Type linéaire dont l'entrée est des noeuds (le nombre de noeuds est embedding_dim) et la sortie est 1 noeud. Comme son nom, il représente une transformation linéaire.
4). sigmoid(): Le layer suivant est le layer sigmoïde. C'est parce que la fonction sigmoid peut mapper le champ de nombre réel à l'espace [0,1]. La valeur de la fonction peut juste être interprétée comme la probabilité d'être une classe positive (la plage de probabilité est 0 ~ 1). et il est préférable de la comparer à la valeur des Labels.
5). nn.BCELoss(): La fonction objectif.
6). torch.optim(): Pour utiliser torch.optim, nous devons construire un objet optimiseur(le nom de cette objet est Optimizer) pour enregistrer l'état actuel et mettre à jour les paramètres en fonction du gradient calculé. tous les optimiseur use la méthode step () pour mettre à jour tous les paramètres. Il y a deux façons de l'appeler:

optimizer.step() Ceci est une version simplifiée supportée par la plupart des optimiseurs.

Lorsque le gradient est calculé en utilisant la méthode backward (), nous vont utiliser la méthode suivante:

```
prediction=model1.forward(txtidx[t])
loss=loss_fn(prediction,ag.Variable(l[t]))
optimizer.zero_grad()
loss.backward()
optimizer.step()
```

7). Version mini_batch:

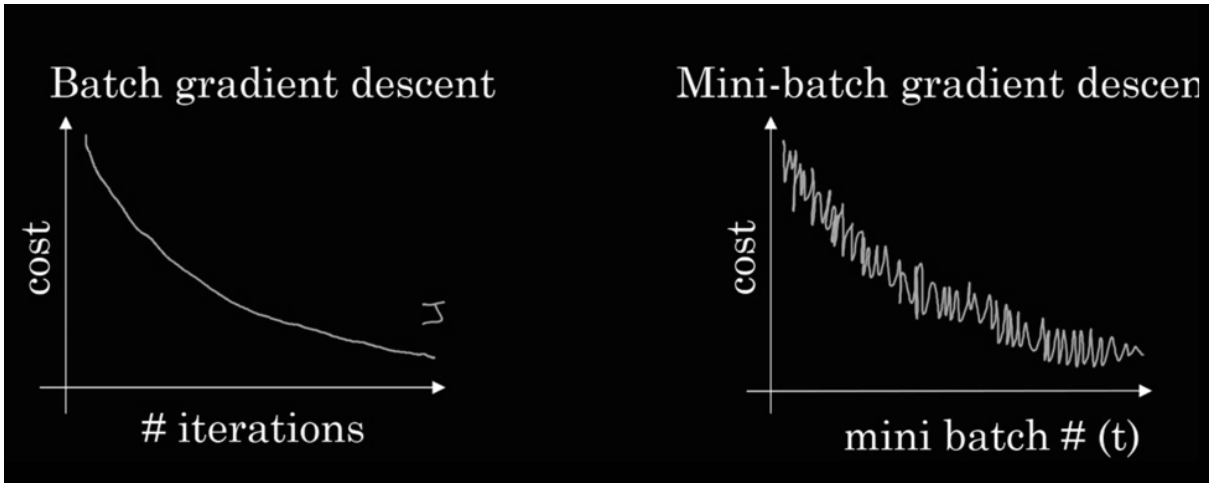

Figure 7: la différence entre “Batch” et “Mini-batch”

Comment implémenter-nous mini_batch?

D'abord nous générons un nombre aléatoire de 0 à 100, utilisons ce nombre aléatoire comme valeur initiale pour la première itération, puis ajoutons 10 à ce nombre aléatoire, effectuons la deuxième itération, et ainsi de suite.

Voici le résultat de la modèle 1:

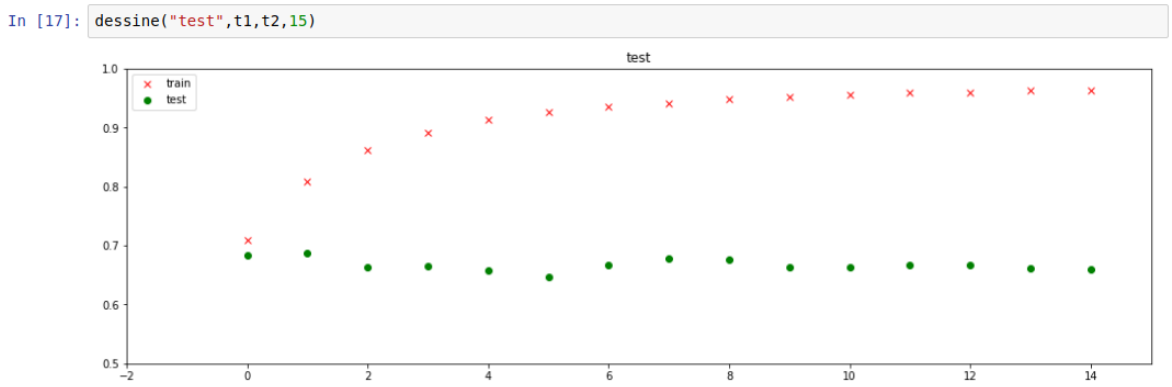

Figure 8: le résulat de test (tous les données)

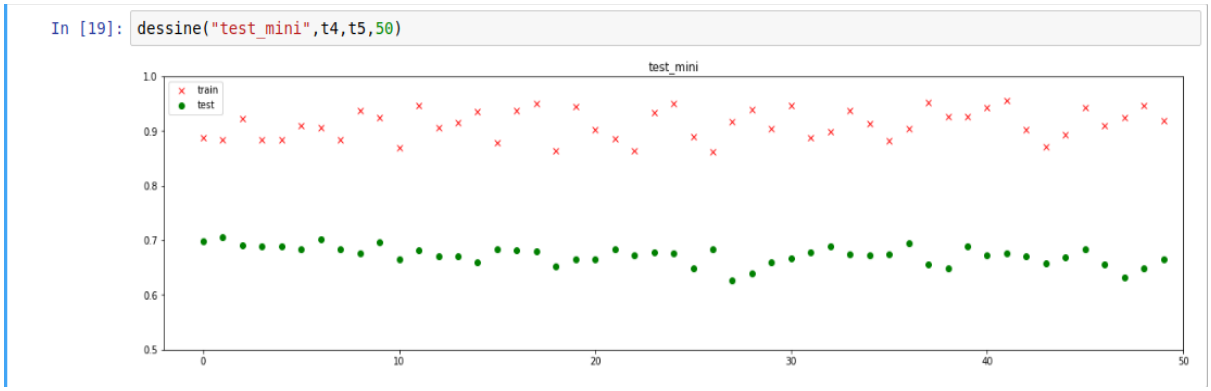

Figure 9: le résulat de test_mini(les données qui sont choisi par mini_batch)

Les résultats sont similaires aux deux graphiques: si nous choisissons d'entraîner tous les ensembles de données, la précision de l'ensemble d'apprentissage continuera d'augmenter et la précision de l'ensemble de tests fluctuera.

Par les deux images, nous pouvons voir que la précision est la plus élevée entre le 8 et le 9, avec une précision d'environ 68%.

Bien sûr, nous pouvons voir que lorsque nous utilisons la méthode d'entraînement **(mini-batch),** le résultat va fluctuer.

b. Modèle 2 qui ont unt “connexion complète”.

```
class CBOW_classifier_plus(nn.Module): # deux layers

    def __init__(self, vocab_size, embedding_dim):
        super(CBOW_classifier_plus, self).__init__()
        self.embeddings = nn.Embedding(vocab_size, embedding_dim)
        self.linear1 = nn.Linear(embedding_dim, 10)
        self.linear2 = nn.Linear(10, 1)
    def forward(self, inputs):
        out = self.linear1(self.makeCBOW(inputs))
        out= F.relu(out)
        out=self.linear2(out)
        return F.sigmoid(out)


    def makeCBOW(self,input_idx):
        input_var =ag.Variable(th.LongTensor(input_idx))
        embeds =self.embeddings(input_var).sum(dim=0)
        return embeds
```

Figure 10:: le modèle complexe

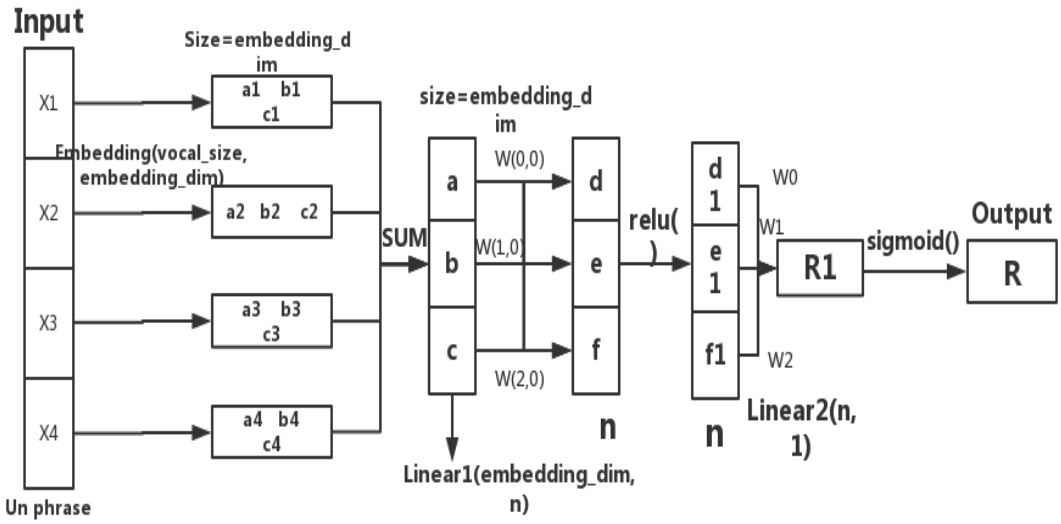

Figure 11: le figure de la modèle qui ont une “Connexion complète”

1). relu(): la fonction d'activation.

(En fait, tant qu'il satisfait une certaine spécification, la fonction d'activation peut avoir une myriade de formes, pas nécessairement celle-ci relu()); il peut aider de normaliser les données, car le résultat du réseau de Layer précédent n'est probablement pas compris entre 0, mais la plage de données doit être unifiée. on utilise la fonction d‘activation pour limiter la plage de données.

2). “Layer” entièrement connectée:

Plus il y a de couches dans la couche intermédiaire, plus le taux de précision est élevé.
Nous avons donc décidé d'ajouter une couche cachée. La couche cachée est une forme complètement connectée.

3). Version mini_batch

Nous avons expliqué dans le modèle 1.

Voici le résultat de la modèle 2:

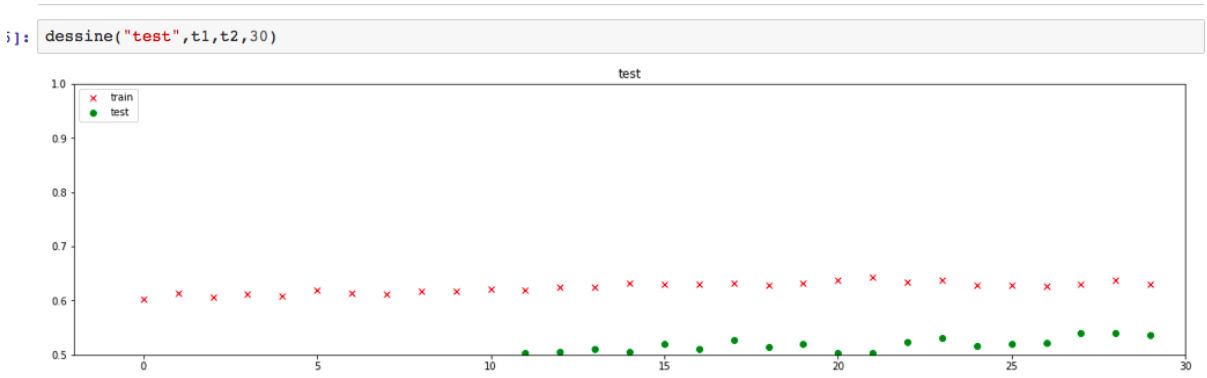

Figure 12: le résulat de test(tous les données)

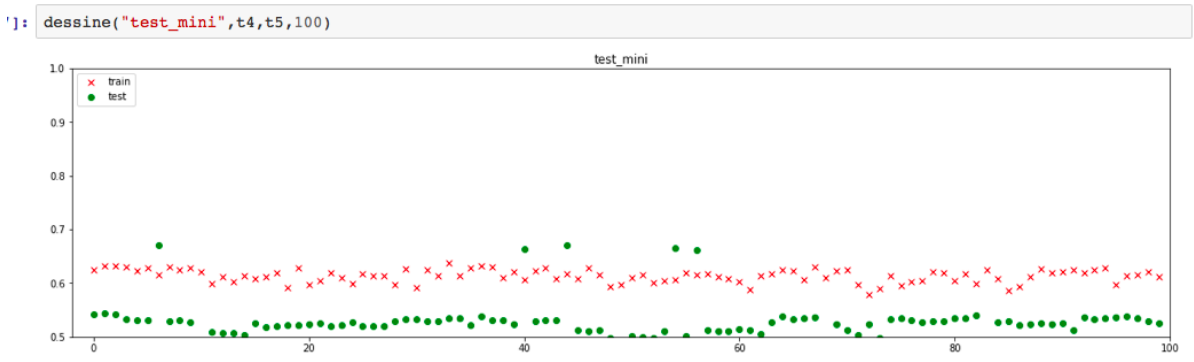

Figure 13: le résulat de test_mini(les données qui sont choisi par mini_batch)

Dans des circonstances normales, nous pensons que plus les couches de réseaux neuronaux sont intermédiaires, plus le taux de précision sera élevé.
Ici, nous ajoutons une couche entièrement connectée basée sur le premier modèle, mais après l'avoir ajoutée, la vitesse de traitement de l'ensemble des données ralentit beaucoup. Étonnamment, le taux de précision a non seulement augmenté mais a beaucoup baissé. Et dans le second modèle, "Total erreur" ne descend pas toujours, mais il augmente d'abord puis fluctue.
Nous ne savons pas quel est le problème, mais nous pensons que le second modèle ne s'applique pas à cette tâche.

Devinez: Lorsqu'il s'agit de problèmes plus complexes, plus il y a de couches d'un réseau neuronal, plus sa précision sera précise. Mais pour des problèmes simples, ce n'est pas forcément.

c. Modèle 3

```
In [10]: class ConvModel(nn.Module):

             def __init__(self, vocab_size, embedding_size, sent_length):
                 super(ConvModel, self).__init__()
                 self.embeds_dim = embedding_size
                 self.sent_length = sent_length
                 self.embeddings = nn.Embedding(vocab_size, embedding_size)
                 self.conv1D_1 = nn.Conv1d(embedding_size, embedding_size, 7, stride=1)
                 self.maxPool1D = nn.MaxPool1d(self.sent_length - 6)
                 self.linear1 = nn.Linear(embedding_size, 1)

             def forward(self, inputs):
                 out = self.makeCBOW(inputs)
                 out = out.view((-1, self.embeds_dim, self.sent_length))
                 out = self.conv1D_1(out)
                 out = self.maxPool1D(out).view(-1, self.embeds_dim)
                 out = self.linear1(out)
                 return F.sigmoid(out)

             def makeCBOW(self,input_idx):
                 input_var =ag.Variable(th.LongTensor(input_idx))
                 embeds =self.embeddings(input_var)
                 return embeds
```

Figure 14: le modele 3

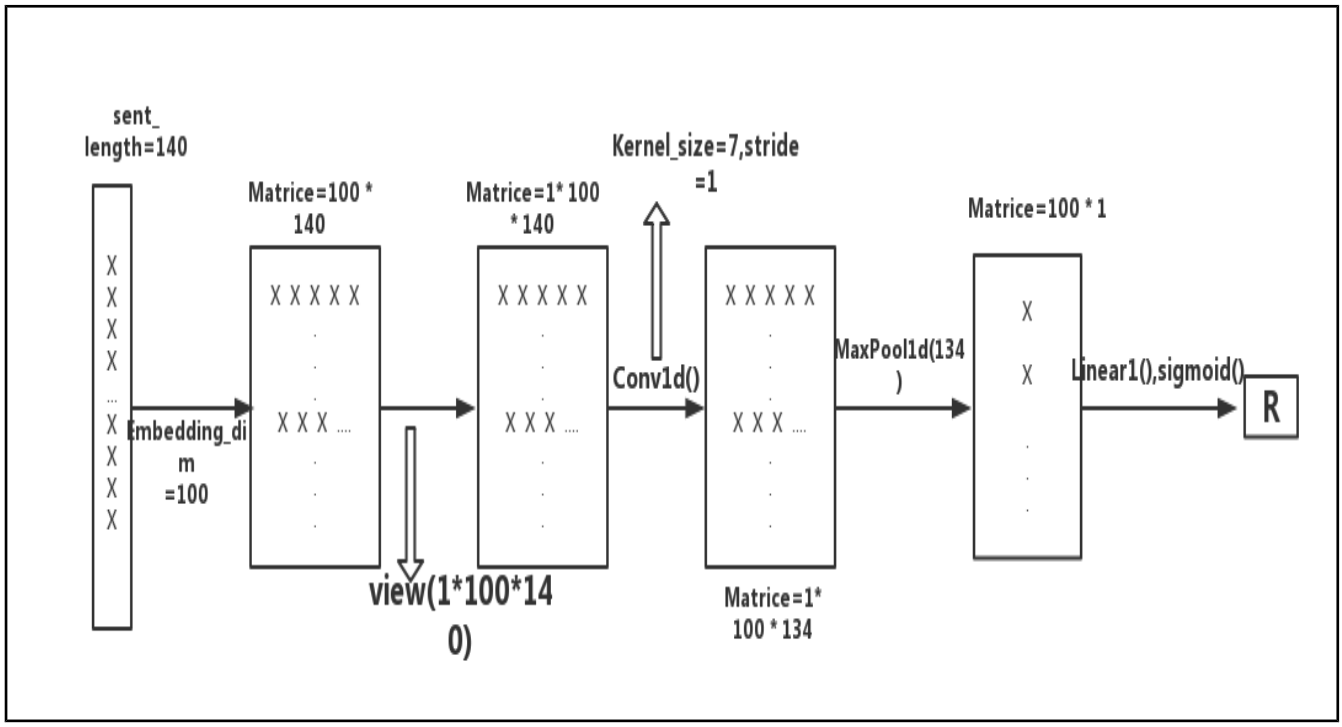

Figure 15: le figure de la modèle 3

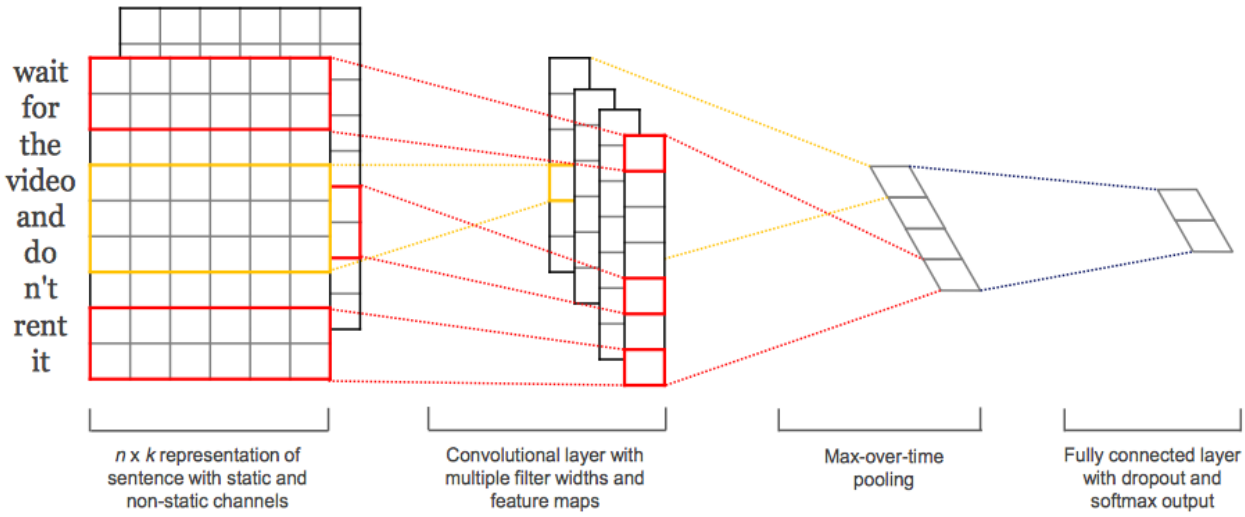

Figure 16: réference de (Convolutional Neural Networks for Sentence Classification)

D'abord, nous changeons la longueur de toutes les phrases au valeur de paramètre (*sent_length*) qui equal à 140 (on ajoute les 0 avant les phases) . Après, on utilise la fonction ***out.view((-1, self.embeds_dim, self.sent_length)),*** pour changer la forme de tensor.
Conv1d(in_channels, out_channels, kernel_size, stride=1): Dans notre code, (in_channels) equal à embedding_size, (out_channels) aussi equal à embedding_size, (kernel_size) est la taille de Noyau de convolution qui equal à 7; et on met stride = 1;

MaxPool1d(): "Pooling Layer", qui sont utilisé pour réduire la dimension des données. Comprime chaque sous-matrice, réduire les dimensions de la matrice d'entrée.
Linear(): comme le modèle 1 et modèle 2.
Sigmoid(): comme le modèle 1er modèle 2.

Voici le résultat de la modèle 3:

Quand le learning rate est 1e-2:

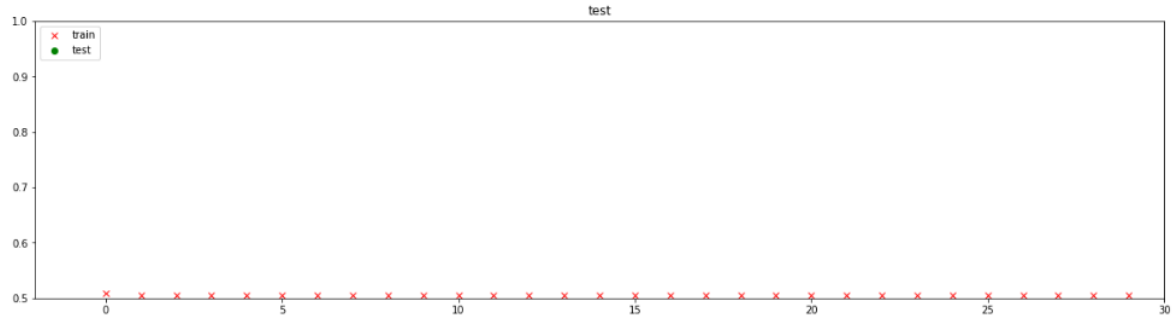

Figure 17: voici le figure quand le learn_rate=1e-2;

Quand le learning rate est 1e-2, le taux de précision est toujours autour de 50% (comme rien fait), je pense que la raison en est que le learning rate est trop élevé, donc on change le learning rate à 1e-3. Le résultat est bien meilleur, mais il y a aussi un problème, le programme lance beaucoup plus lentement.

Quand le learning rate est 1e-3:

```
('epoch :', 0, 'train :', 0.6385333333333333)
('epoch :', 0, 'error :', 19320.469742212306)
('epoch :', 0, 'test :', 67.98)
('epoch :', 1, 'train :', 0.7365333333333334)
('epoch :', 1, 'error :', 15807.227760817508)
('epoch :', 1, 'test :', 70.72)
('epoch :', 2, 'train :', 0.7938666666666667)
('epoch :', 2, 'error :', 13223.954747923883)
('epoch :', 2, 'test :', 71.03)
('epoch :', 3, 'train :', 0.8380333333333333)
('epoch :', 3, 'error :', 10971.533507717846)
('epoch :', 3, 'test :', 69.98)
('epoch :', 4, 'train :', 0.8770666666666667)
('epoch :', 4, 'error :', 8922.37372009449)
('epoch :', 4, 'test :', 69.48)
('epoch :', 5, 'train :', 0.9093333333333333)
('epoch :', 5, 'error :', 6877.376435621098)
('epoch :', 5, 'test :', 68.03)
```

Figure 18: le résulat de test(tous les données)

Nous avons passé une nuit à lancer cinq fois le programme, on peut voir le taux correct sur l'ensemble de test est d'environ 70%, nous utilisons seulement une couche de convolution, le résultat est déjà pas mal, je pense que si nous ajoutons plus couches de convolution, nous obtiendrons de meilleurs résultats. Peut-être que nous pouvons avoir une chance de le vérifier dans le futur.

V- Conclusion

Ce TER nous permet de familiariser et d'appliquer des connaissances de mathématique(linear algebra, opérations matricielles, etc) et Python (numpy, pytorch, matplotlib, etc) et algorithme d'apprentissage que nous avons appris en cours. Nous avons rencontré des difficultés, mais sous la direction de l'enseignant, les difficultés ont été résolues.
Grâce à ce TER, nous comprenons mieux le concept d'apprentissage automatique et son application pratique.

VI-REFERENCES

[1].https://www-cs.stanford.edu/people/alecmgo/papers/TwitterDistantSupervision09.pdf

[2].http://www.numpy.org/
[3].https://pytorch.org/docs/master/nn.html
[4].https://github.com/yoonkim/CNN_sentence
[5].https://arxiv.org/abs/1408.5882