Université Claude Bernard Lyon 1

Intelligence Bio-Inspirée
TP

Classifieur de textes à partir d'un RNN

Élèves :
Nathan Etourneau
Paul Flagel

Enseignants :
Frederic Armetta
Alexandre Devillers
Mathieu Lefort

11 février 2022

Table des matières

Introduction

Le but de ce TP était de se familiariser avec les réseaux récurrents, et au NLP. Au cours de ce TP, nous avons développé un classifieur de textes, tout d'abord avec un réseau de neurone récurrent simple, puis avec un réseau LSTM. Ce classifieur avait la particularité de prendre en entrée directement les mots, et non pas des caractères. Cela fait que les données sont de grande dimension, et rend l'apprentissage un peu plus difficile, mais bien plus intéressant : les modèles réels employés en NLP (notamment les transformeurs), sont des modèles prenant en entrée des mots.

1 Préparation des données

Les données ont déjà été pré-traitées. En effet, on constate que l'ensemble du texte est en minuscule et sans ponctuation. Toutefois, il subsiste certaines fautes de frappes ou bizarreries de twitter : "i feel so dazed a href http twitter".

Les données sont réparties en 3 jeux de données : train, eval et test. Il y a 16 000 données d'entraînement, 2000 données de validation, et 2000 données de test. Elles sont stockées sous le format "texte ;label", nous avons donc écrit un petit script qui lit le fichier, et extrait automatiquement chaque donnée du jeu de donnée, et séparé le texte de son label.

```
def load_file(dataset_type):
    """Reads the file and return the data as lists.
    Data already has been put in lowercase, without punctuation.
    No need to do it again.
    There are typos in the text"""
    text = []
    labels = []

    raw_data = open(f"data/{dataset_type}.txt").read().lower()

    for datapoint in raw_data.split('\n'):
        try:
            text_sample, label = datapoint.split(';')
            text.append(text_sample)
            labels.append(label)
        except ValueError:
            continue
    return text, labels
```

Une fois le texte obtenu sous cette forme, il nous est nécessaire de construire un vocabulaire : cela consiste à extraire tous les N mots présents dans l'ensemble des textes (train, entraînemenent et test), et d'associer un entier unique à chaque mot dans l'intervalle $0, ..., N-1$.

Cette opération s'appelle la tokenization. Nous avons décidé de limiter la taille de ce vocabulaire en supprimant les mots apparaissant trop souvent (plus de 2000 fois), ou pas assez souvent (moins de 2 fois). Nous avons fait ce choix car un mot trop rare ne porte pas une grande information : il est difficile pour un réseau de neurones d'apprendre à partir de peu d'exemples. De même, un mot présent dans chaque document n'est pas très important pour cerner l'émotion

du texte : ce sont des stopwords.

De même, nous limitons la taille des phrases : avoir des phrases trop longues rend les données plus lourdes, pour un gain de performances nul voire négatif.

```
def tokenize_text(text_dataset, sentence_length):
    """Converts a text dataset (list of strings) intro a list
    of short strings (tokens). For instance 'i am Superman'
    is converted in ['I', 'am', 'superman']."""
    processed_text = []
    for sentence in text_dataset:
        words = sentence.split(' ')
        words = [w if w in vocab else 'oov' for w in words]

        # If the sentence is too long, we crop it
        if len(words) > sentence_length:
            words = words[:sentence_length]

        # If the sentence is too short, we pad it with 'oov'
        if len(words) < sentence_length:
            pad_length = sentence_length - len(words)
            pad_with = ['oov' for _ in range(pad_length)]
            words.extend(pad_with)
        processed_text.append(words)
    return processed_text

convert_token_to_vector_dict = {k:v for v,k in enumerate(vocab_with_oov)}
util_matrix = np.eye(len(vocab_with_oov))  # To avoid recomputing the matrix each time.

def project_token(token):
    """Converts a token (a string) into a one hot
    representation."""
    return util_matrix[convert_token_to_vector_dict[token]]


def process_tokenized_text(tokenized_text):
    """Process a tokenized text (a list of list of tokens) into
    a list of lists of one-hot-representations of words (as a numpy array)."""
    result = []
    for sentence in tokenized_text:
        sentence = [project_token(token) for token in sentence]
        result.append(sentence)
    return torch.tensor(np.array(result, dtype=np.float32), device=device)
```

Ensuite, nous traitons les labels : nous verrons plus tard que la fonction de perte que nous utiliserons est la `NLLLoss`, qui requiert que les labels ne soient pas sous forme one-hot, mais sous forme "dense". Ainsi, nous convertissons les labels d'une chaîne de caractère à un entier entre 0 et 5.

```
# Labels are 'fear', 'anger', 'surprise', 'sadness', 'love', 'joy'
# Here, we convert them to values in {0, 1, ..., 5}

label_names = list(set(train_labels))
label_converter = {k:v for (v, k) in enumerate(label_names)}

def get_dense_labels(labels_list):
    """Converts a list of labels into a dense label in {0, ..., 5}."""
    dense_labels = [label_converter[label] for label in labels_list]
    return torch.tensor(np.array(dense_labels, dtype=np.int_), device=device, dtype=torch.lo
```

Nous combinons ensuite l'ensemble de ces fonctions pour écrire une fonction qui extrait et traite les données.

```
def get_dataset(dataset_type, sentence_length):
    text, labels = load_file(dataset_type)
    dataset = process_tokenized_text(tokenize_text(text, sentence_length))
    dense_labels = get_dense_labels(labels)

    return dataset, dense_labels
```

Il suffit d'appeler la fonction `get_dataset` avec la taille de phrase voulue, ainsi qu'avec le type de dataset voulu (parmi 'train', 'val' et 'test').

2 Entraînement avec un simple réseau récurrent

Dans le cadre de ce TP, nous avons utilisé un RNN que nous avons codé à la main : nous n'avons pas utilisé de layer récurrent déjà implémenté dans Pytorch.

Nous nous sommes inspirés du tutoriel RNN de Pytorch disponible ici : `https://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial`
Cependant, nous avons adapté le code car la dimension de notre entrée est trop grande pour utiliser ce réseau tel quel. Nous avons ajouté un layer d'embedding permettant de réduire la dimension de l'entrée.

Nous avons également modifié la fonction initialisant l'état caché du réseau récurrent : le code du tutoriel ne permet pas l'utilisation du réseau sous forme de batchs.

Le code d'initialisation du réseau est le suivant :

```
class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, emb_size, output_size):
        super(RNN, self).__init__()

        self.hidden_size = hidden_size
        self.i2e = nn.Linear(input_size, emb_size)
```

```
        self.e2h = nn.Linear(emb_size + hidden_size, hidden_size)
        self.e2o = nn.Linear(emb_size + hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input, hidden):
        emb = F.relu(self.i2e(input))
        combined = torch.cat((emb, hidden), 1)
        hidden = F.relu(self.e2h(combined))
        output = self.e2o(combined)
        output = self.softmax(output)
        return output, hidden

    def initHidden(self, batch_size):
        return torch.zeros(
            batch_size,
            self.hidden_size,
            dtype=torch.float32,
            requires_grad=True
        )
```

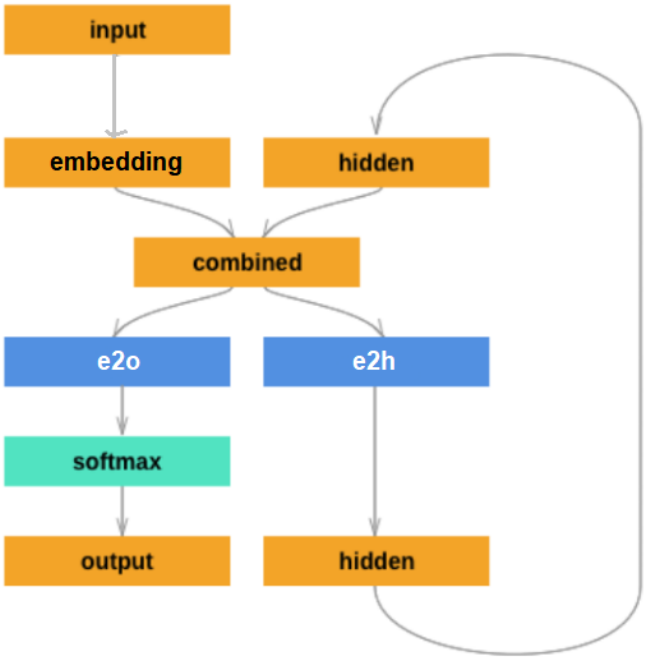

FIGURE 1 – Schéma du modèle implémenté

L'implémentation avec Pytorch est totalement transparente : il suffit de coder la passe avant, et la backpropagation est gérée automatiquement.

Nous avons écrit un script permettant l'entraînement du réseau avec batchs. Notre meilleur hyperparamétrage nous permet d'atteindre **75%** de précision sur le jeu de validation.

3 Entraînement avec un réseau LSTM

Nous avons voulu comparer notre réseau récurrent fait à la main avec un réseau récurrent qui utiliserait les outils de Pytorch. Nous avons implémenté un réseau récurrent constitué d'un embedding, d'un LSTM permettant de traiter une séquence entière en une fois, d'un layer fully connected ainsi que de dropout.

```
class LSTMNet(nn.Module):
    def __init__(self, input_size, hidden_size, emb_size, output_size, num_layers, dropout)
        super(LSTMNet, self).__init__()

        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.i2e = nn.Linear(input_size, emb_size)
        self.lstm = nn.LSTM(emb_size, hidden_size, batch_first=True, num_layers=num_layers)
        self.dropout = nn.Dropout(dropout)
        self.fc = nn.Linear(hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input, hidden):
        emb = F.relu(self.i2e(input))
        output, hidden = self.lstm(emb, hidden)
        output = output[:, -1, :]
        output = self.dropout(F.relu(output))
        output = self.fc(output)
        output = self.softmax(output)
        return output, hidden

    def initHidden(self, batch_size):
        return (
            torch.zeros(self.num_layers,
            batch_size,
            self.hidden_size,
            dtype=torch.float32,
            requires_grad=True
            ),
            torch.zeros(self.num_layers,
            batch_size,
            self.hidden_size,
            dtype=torch.float32,
            requires_grad=True
            )
        )
```

4 Résultats de nos modèles sur les jeux de validation et de test

Nous avons constaté que nos modèles étaient très performants sur le jeu d'entraînement (plus de 90% de précision si nous le souhaitons, mais ce serait de l'overfitting), ainsi que sur le jeu de validation (76% de précision pour le RNN, 81% de précision pour le LSTM). Toutefois, nous

avons constaté que lorsque nous testons notre modèle sur des données nouvelles (jeu de test), les performances ont chuté drastiquement (entre 25% et 35% maximum). Le modèle n'apprend pas et on constate un très grand overfit.

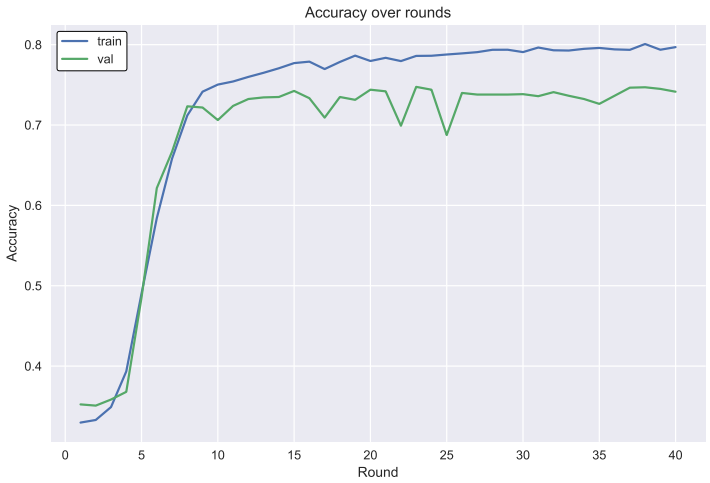

FIGURE 2 – Précision en fonction des rounds pour le RNN sur le jeu d'entraînement et de validation

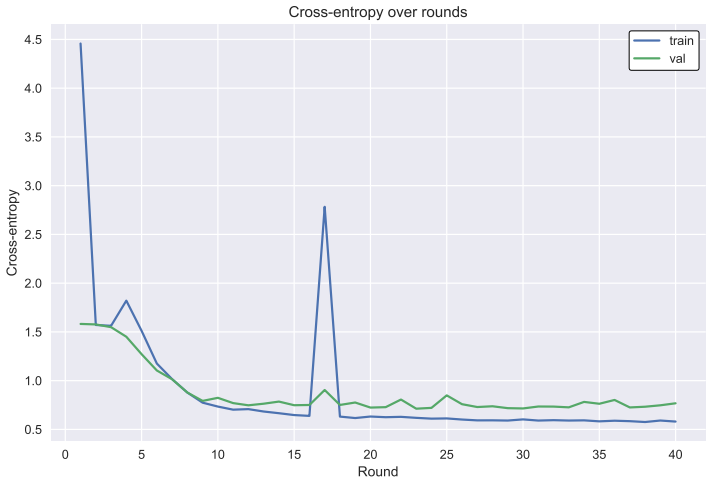

FIGURE 3 – Cross-entropy en fonction des rounds pour le RNN sur le jeu d'entraînement et de validation

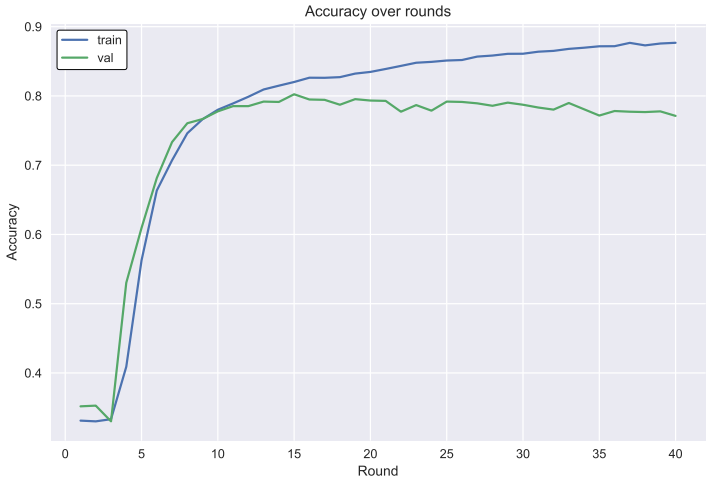

FIGURE 4 – Précision en fonction des rounds pour le LSTM sur le jeu d'entraînement et de validation

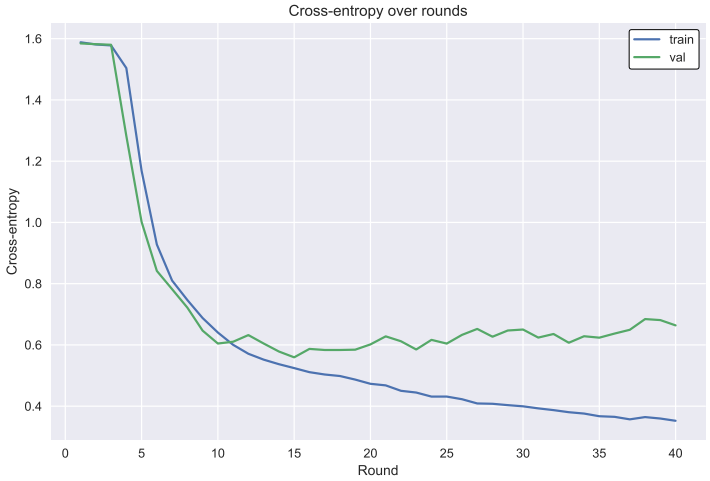

FIGURE 5 – Cross-entropy en fonction des rounds pour le LSTM sur le jeu d'entraînement et de validation

Nous avons essayé d'augmenter le paramètre de régularisation L^2, mais cela a eu pour conséquence d'empêcher le réseau d'apprendre : avec un coefficient de $1 \cdot 10^{-3}$, l'accuracy du réseau de neurones sur le jeu de validation ou de test tombe à 35% maximum, alors que avec $1 \cdot 10^{-4}$, le réseau apprend parfaitement.

Nous avons essayé d'entraîner le modèle sur le jeu d'entraînement, et de l'évaluer sur le jeu de test uniquement, et dans ce cas les performances ne sont pas

Nous avons eu beaucoup de difficultés à résoudre ce problème, nous n'en connaissons pas la cause. Nous nous sommes demandés si nous overfittions pas le jeu de validation, en entraînant suffisamment de modèles jusquà ce qu'un modèle soit miraculeusement performant sur le jeu de validation. Toutefois, cela ne nous semble pas expliquer le fait que le modèle appris n'ait aucun pouvoir prédictif sur le jeu de test.

5 Matrice de confusion

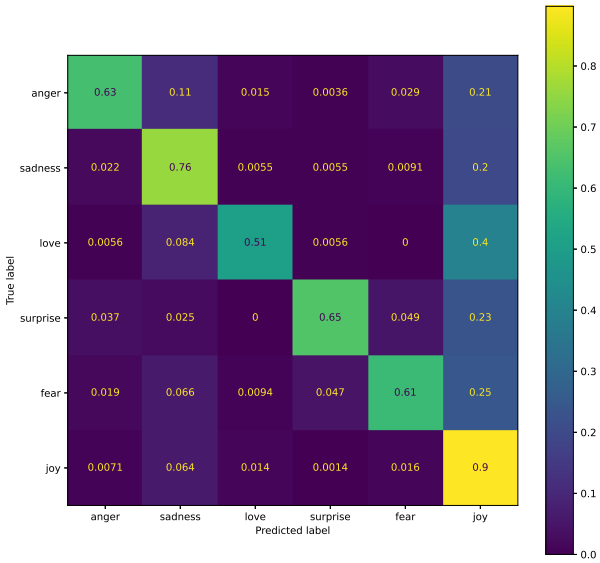

FIGURE 6 – Matrice du confusion du réseau RNN développé, prédictions faites sur le jeu de validation

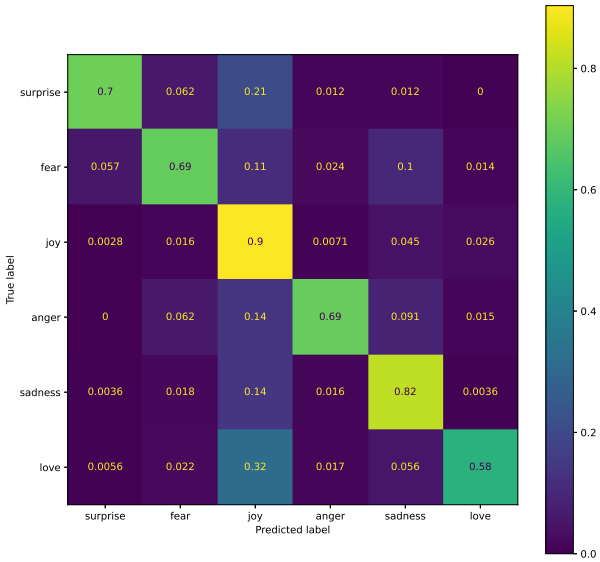

FIGURE 7 – Matrice du confusion du réseau LSTM développé, prédictions faites sur le jeu de validation

6 Optimisation des hyperparamètres

Pour optimiser nos hyperparamètres, nous avons utilisé le package Optuna, qui permet de chercher de manière simple des hyperparamètres performants.

Etant donné que nous avons rencontré des problèmes de performance sur le jeu de test, nous avons principalement cherché à débugger ceci plutôt qu'à trouver des hyperparamètres optimaux. Toutefois, le script est parfaitement fonctionnel.

Conclusion

Au cours de ce TP, nous avons pu nous familiariser avec certains aspects du NLP : l'utilisation de réseaux récurrents, ainsi que certains aspects basiques de préprocessing de texte. Nous avons également pu nous familiariser avec la différentiation automatique de Pytorch, et pu constater qu'elle est très efficace.

Au cours de ce TP, nous avons rencontré de grandes difficultés, notamment certains bugs très frustrants nous ayant empêché d'avancer d'avantage. En particulier, nous ne nous expliquons pas la différence de performances entre le modèle entraîné sur le jeu d'entraînement et évalué sur le jeu de validation, et celles du modèle entraîné sur le jeu d'entraînement (ou bien entraînement + validation) évalué sur le jeu de test.

Nous aurions aimé implémenter une baseline de classification de texte avec une approche type "bag-of-words", pour évaluer la pertinence d'un modèle récurrent pour ce type de problème. Toutefois, nous avons consacré notre temps à essayer de comprendre et débugger le problème cité plus haut.