

# projet3

October 27, 2025

## 0.1 Import des modules et dépendances

```
[1]: from rnn_model import SimpleRNN
     from data_processing import *
     from data_processing import load_file
```

### 0.1.1 Test rapide du chargement et du tokenizer

On vérifie que : 1. Le fichier est bien chargé. 2. Les phrases sont correctement tokenisées. 3. Les émotions associées sont cohérentes.

```
[2]: text, emotion = load_file("./dataset/train.txt")

     print(text[2])
     print(tokenizer(text[2]))
     print(emotion[2])
```

```
im grabbing a minute to post i feel greedy wrong
['im', 'grabbing', 'a', 'minute', 'to', 'post', 'i', 'feel', 'greedy', 'wrong']
anger
```

### 0.1.2 Vérification du générateur de tokens

On teste la fonction `yield_tokens()` pour s'assurer qu'elle parcourt bien tout le corpus et renvoie une liste plate de tous les mots (tokens).

On limite l'affichage pour ne pas surcharger la sortie.

```
[3]: # Test du générateur de tokens
     lst = list(yield_tokens(text))
     print(lst[:50]) # affiche seulement les 50 premiers tokens
     print("Total tokens:", len(lst))
```

```
['i', 'didnt', 'feel', 'humiliated', 'i', 'can', 'go', 'from', 'feeling', 'so',
'hopeless', 'to', 'so', 'damned', 'hopeful', 'just', 'from', 'being', 'around',
'someone', 'who', 'cares', 'and', 'is', 'awake', 'im', 'grabbing', 'a',
'minute', 'to', 'post', 'i', 'feel', 'greedy', 'wrong', 'i', 'am', 'ever',
'feeling', 'nostalgic', 'about', 'the', 'fireplace', 'i', 'will', 'know',
'that', 'it', 'is', 'still']
Total tokens: 306661
```



### 0.1.3 Construction du vocabulaire

On crée un vocabulaire avec `build_vocab_from_iterator()` :

- chaque mot unique reçoit un identifiant entier,
- les tokens spéciaux <pad> et <unk> sont ajoutés,
- et on vérifie que certains mots courants ont bien une valeur associée.

```
[4]: ## Associer à chaque mot une valeur unique (entier positif) **pas de doublon**
vocab = build_vocab_from_iterator(lst, specials=["<pad>", "<unk>"])
print(len(vocab))
print(vocab["i"])
print(vocab["didn't"])
print(vocab["feel"])
```

15214

2

3

4

### 0.1.4 Vocabulaire des émotions

On construit maintenant le vocabulaire des **classes d'émotions**, de la même manière que pour les mots du texte. Chaque émotion correspond à un identifiant unique.

```
[5]: # Faire pareil avec les émotions, qui représentent les classes
classes = build_vocab_from_iterator(yield_tokens(emotion))
print(len(classes))
print(classes["anger"])
```

6

1

### 0.1.5 Test — Encodage d'une phrase en entiers

```
[6]: # Coder une phrase ## Représenter une phrase comme une suite de valeurs
print(text[1])
codage_entier_phrase = vocab(tokenizer(text[1]))
print(codage_entier_phrase)
```

i can go from feeling so hopeless to so damned hopeful just from being around  
someone who cares and is awake

[2, 6, 7, 8, 9, 10, 11, 12, 10, 13, 14, 15, 8, 16, 17, 18, 19, 20, 21, 22, 23]

### 0.1.6 Représenter chaque phrase comme un tenseur d'entiers

À partir de cette étape, on commence à utiliser **PyTorch** pour manipuler les données. *###*  
Représenter chaque phrase sous forme one-hot

Chaque mot est transformé en un vecteur binaire où une seule position vaut 1.



Charger tout le dataset en one-hot peut être coûteux en mémoire — on préfère souvent le faire **par batch** avec un `DataLoader`.

```
[7]: ## Représenter chaque phrase comme un tensor d'entiers **seulement à partir de
```

```
↪ cette étape, on utilise la librairie pytorch**
```

```
from torch import tensor # Import minimal autorisé
```

```
tensor_entier = tensor(codage_entier_phrase)
```

```
print(tensor_entier)
```

```
tensor([ 2,  6,  7,  8,  9, 10, 11, 12, 10, 13, 14, 15,  8, 16, 17, 18, 19, 20,
        21, 22, 23])
```

```
[8]: ## Représenter chaque phrase comme un tensor one hot **attention, il peut être  

     ↳ difficile de charger en mémoire l'ensemble du dataset sous la forme on-hot,  

     ↳ privilégier si besoin la génération one-hot par batch, en utilisant un  

     ↳ DataLoader**  

     tensor_one_hot = one_hot(tensor_entier, num_classes=len(vocab))  

     print(tensor_one_hot[:10])
```

[illegible]































[illegible]



[illegible]







[illegible]















































































[illegible]















































































































[illegible]























































































































































[illegible]















































































[ ]:



0.2 Entrée (input) : mot par mot, la première dimension correspond à la taille du batch - Dimension : (batch\_size, vocab\_size) ## Sortie (output) : valeurs de classes prédites, la première dimension correspond à la taille du batch - Dimension : (batch\_size, class\_size)

0.2.1 Étape 1 : Tester le réseau avec un mot unique (batch\_size = 1, sans récurrence)

```
[9]: import torch
from rnn_model import SimpleRNN
from data_processing import build_vocab_from_iterator, tokenizer, one_hot
from torch.utils.data import DataLoader
from dataset_nlp import EmotionDataset # ton fichier avec la classe ci-dessus
```

```
[10]: # Exemple de vocabulaire
vocab = build_vocab_from_iterator(["hello", "world", "<pad>", "<unk>"])
vocab_size = len(vocab)

# Initialiser le modèle
emb_size = 128
hidden_size = 256
class_size = 5 # Exemple : 5 classes d'émotions
rnn = SimpleRNN(vocab_size, emb_size, hidden_size, class_size)

# Tester avec un mot unique
word = "hello"
word_idx = vocab([word])[0]
one_hot_word = one_hot([word_idx], num_classes=vocab_size)[0] # [vocab_size]
one_hot_tensor = torch.tensor(one_hot_word, dtype=torch.float32).unsqueeze(0)
# [1, vocab_size]

# Initialiser l'état caché
hidden = rnn.init_hidden(batch_size=1)

# Passer le mot dans le réseau
output, hidden = rnn(one_hot_tensor, hidden)
print("Output:", output)
print("Hidden state:", hidden)
```

Output: tensor([[ -1.3571, -1.6602, -1.8526, -1.5256, -1.7250]],  
grad\_fn=<LogSoftmaxBackward0>)

Hidden state: tensor([[ -1.0035e-01, -1.3253e-01, 2.0601e-02, 1.7054e-01,  
-1.4101e-01,  
-2.3113e-01, -2.2976e-01, -3.1047e-02, -1.5642e-01, 2.3424e-01,  
-1.1943e-01, -2.7248e-01, 1.4614e-01, 1.0315e-01, -1.2543e-01,  
-4.3361e-02, 8.6662e-03, 5.0226e-02, -1.7501e-01, -9.7438e-02,  
7.0754e-02, -1.1957e-01, -1.2396e-01, 9.8346e-02, 7.3166e-02,  
-6.5795e-02, -2.6078e-02, -1.5470e-02, -3.1892e-02, -2.6138e-01,  
-1.1896e-01, 1.1426e-01, 7.6048e-02, 8.2866e-02, 1.1485e-01,



```

-2.7786e-01, 1.5225e-01, 2.1574e-02, -1.1005e-01, 3.0211e-02,
-1.2746e-01, 6.2493e-03, 4.2803e-02, -9.3021e-02, 1.8730e-01,
1.2200e-01, -8.2056e-02, 2.4534e-02, 1.9807e-02, 1.7239e-01,
-6.2639e-02, 8.0631e-02, 1.8794e-01, -2.5721e-02, -4.8843e-02,
6.7863e-02, 5.6038e-02, -2.1934e-01, -1.8591e-01, -1.0159e-02,
-9.9305e-02, 1.2165e-02, 3.8223e-02, 8.9705e-02, 9.9681e-02,
1.6475e-01, 7.3607e-04, -1.5038e-01, 2.7419e-01, -1.9308e-01,
1.4113e-01, 3.8444e-02, -7.5039e-02, -9.9778e-02, -7.2572e-02,
9.0131e-02, -3.8083e-02, 4.6021e-02, -1.6385e-02, 7.5569e-02,
-6.6086e-02, -1.6880e-01, 1.6783e-01, 3.0661e-02, -3.7156e-02,
1.0685e-01, 1.4255e-02, -1.1483e-01, -7.4305e-02, -4.9115e-02,
1.0006e-01, 5.8197e-02, 1.0199e-03, 2.1699e-01, -7.8453e-02,
-1.5973e-01, 1.2058e-01, -3.6325e-02, -1.3857e-01, 3.9181e-02,
1.7146e-02, 2.4709e-02, 7.0732e-02, 7.1236e-03, -1.1617e-01,
-5.2253e-02, 7.3232e-02, 6.8820e-02, -1.1296e-02, -1.1066e-01,
1.7774e-02, 5.4810e-02, -7.0445e-02, -5.0128e-02, 3.1739e-02,
3.7571e-02, -9.3538e-02, 1.3448e-01, 1.3263e-01, 2.1794e-01,
9.8452e-02, 2.8479e-02, 1.4109e-01, -5.5224e-02, 8.0524e-02,
1.3176e-01, 1.0172e-01, 1.3684e-01, -2.0696e-01, -2.6845e-01,
7.3260e-02, 5.9941e-02, -1.1105e-01, 1.7752e-01, 1.1003e-01,
2.0858e-01, -1.9872e-01, -5.1542e-02, -8.6692e-02, -3.4945e-02,
-5.9264e-02, 1.7509e-01, -1.6023e-01, -2.0798e-01, 1.4035e-01,
7.8178e-02, 1.3019e-02, -1.0506e-01, -4.6334e-03, 2.8120e-02,
8.3576e-02, -7.5941e-02, -5.1129e-02, 9.5578e-02, -2.1194e-01,
1.9527e-01, -2.3170e-01, -9.8790e-02, 7.5361e-02, 1.8795e-01,
-5.2247e-02, -9.6844e-02, -2.3368e-02, -1.2786e-01, -1.4081e-01,
-1.3003e-01, 6.6973e-02, -7.3681e-02, 7.0800e-02, -2.7557e-02,
-1.4604e-01, -6.5418e-02, 6.7589e-02, -1.4806e-01, 1.6395e-01,
1.4862e-01, -7.5275e-02, 1.2271e-01, 1.6938e-01, -1.3710e-01,
-3.5635e-02, 1.1402e-01, -9.0846e-03, -4.7396e-02, 7.3813e-04,
-3.7465e-02, 1.2697e-01, -4.3839e-03, 1.8618e-02, -9.2187e-02,
1.2785e-01, 2.1671e-01, -3.6385e-02, -1.1112e-01, 4.0537e-03,
1.3745e-01, -2.0611e-02, -1.8721e-01, -2.3065e-01, 4.0062e-02,
3.7346e-02, 9.1147e-02, 1.1937e-01, -9.5108e-02, -1.5269e-01,
-1.3399e-01, -4.4408e-02, -4.0738e-02, 2.9799e-03, 5.5307e-02,
6.0941e-02, -2.1911e-01, -2.1869e-01, 3.3204e-02, -5.2491e-02,
-1.9787e-02, -2.1044e-01, 3.8503e-02, -5.2044e-02, -2.4632e-01,
1.4593e-01, 2.6422e-02, 7.9448e-02, -1.1152e-01, -1.6377e-01,
4.2179e-02, 1.5102e-01, 2.0483e-02, 4.1200e-02, -1.8713e-01,
-1.3546e-02, -1.0633e-02, 1.9310e-01, -1.1938e-02, -2.6612e-01,
5.8024e-02, -2.3438e-01, 2.5993e-04, 8.6584e-02, 4.4861e-02,
-1.1374e-01, -1.1029e-01, -5.3887e-02, 1.7535e-01, 1.4143e-01,
1.0560e-01, -7.2538e-02, 1.5635e-01, -7.7503e-02, -1.0670e-01,
-6.5537e-02, -1.4012e-01, -3.1854e-03, 2.1980e-02, 6.2597e-02,
-2.0562e-01]], grad_fn=<TanHBackward0>)

```



## 0.2.2 Étape 2 : Tester avec la récurrence (ajout de chaque mot)

```
[14]: # --- 1. Créer un vocabulaire simple ---
vocab = build_vocab_from_iterator(["hello", "world", "<pad>", "<unk>"])
vocab_size = len(vocab)

# --- 2. Initialiser le modèle ---
emb_size = 128
hidden_size = 256
class_size = 5 # par exemple : 5 émotions
rnn = SimpleRNN(vocab_size, emb_size, hidden_size, class_size)

# --- 3. Exemple de phrase ---
sentence = ["hello", "roua"]
indices = vocab(sentence) # -> [index_hello, index_world]

# --- 4. Encoder chaque mot en one-hot (en utilisant ta fonction) ---
one_hot_vectors = one_hot(indices, num_classes=vocab_size)

# --- 5. Initialiser l'état caché ---
hidden = rnn.init_hidden(batch_size=1)

# --- 6. Passer les mots un par un (récurrence) ---
for i, vec in enumerate(one_hot_vectors):
    word_tensor = torch.tensor(vec, dtype=torch.float32).unsqueeze(0) # [1, vocab_size]
    output, hidden = rnn(word_tensor, hidden)
    print(f"Step {i+1} | Word ID: {indices[i]} | Output: {output}")

print("\nFinal hidden state:", hidden)
```

Step 1 | Word ID: 0 | Output: tensor([[ -1.6172, -1.4896, -1.6752, -1.7067, -1.5734]]),

grad\_fn=<LogSoftmaxBackward0>)

Step 2 | Word ID: 3 | Output: tensor([[ -1.6828, -1.4824, -1.6425, -1.5780, -1.6758]]),

grad\_fn=<LogSoftmaxBackward0>)

Final hidden state: tensor([[ 8.2195e-02, -2.3711e-01, -1.4122e-01, 1.1348e-01, -5.0683e-02,

-1.9446e-01, 1.3475e-01, -1.2273e-01, 4.5906e-02, -5.5017e-02,  
-4.6746e-02, -1.9322e-01, 1.0754e-01, -8.7856e-02, -1.8809e-01,  
-4.2591e-02, -4.5787e-02, -6.4169e-02, -1.8305e-01, 6.9305e-03,  
1.9881e-01, -1.3570e-01, -1.4608e-01, -3.6005e-02, -1.4725e-01,  
-2.8959e-02, 2.5060e-01, -4.6009e-02, 6.0508e-02, -3.7236e-02,  
1.4066e-02, -1.9708e-01, 1.3228e-01, -3.2625e-03, 3.9620e-02,  
-1.5147e-01, 1.5441e-01, -5.6318e-02, -4.8049e-02, -6.4424e-02,  
-1.3384e-01, -1.4552e-01, 1.8096e-01, 1.6854e-01, 5.8884e-02,



```

1.1151e-01, -4.9366e-02, 1.0255e-01, 6.7700e-02, 3.1324e-01,
1.6927e-01, -2.1002e-02, 1.9175e-01, 1.9716e-01, 1.3226e-01,
1.6448e-01, -2.6363e-02, 4.1800e-02, -5.3125e-03, -7.9147e-02,
-3.9565e-02, 2.0709e-01, 6.7767e-02, 6.7057e-02, -2.3268e-01,
1.1310e-01, 9.2988e-02, -3.3184e-01, 1.1781e-01, 6.9826e-03,
3.0962e-01, 7.9114e-02, -2.2202e-01, -8.4304e-03, -1.6811e-01,
2.9946e-03, -1.8158e-01, -1.9220e-01, -7.9208e-02, 1.4480e-02,
-2.7294e-01, 7.0503e-02, 8.0594e-02, -1.5268e-01, -9.1313e-03,
4.1393e-02, 2.9102e-01, 2.1818e-01, -5.1811e-02, -1.0788e-02,
-1.7308e-01, 4.0697e-02, 1.0706e-01, 1.9726e-01, -2.3403e-01,
-7.0521e-02, 1.4754e-01, -1.2139e-01, 3.6096e-01, 1.6519e-01,
-7.4117e-02, 1.5878e-03, 1.0951e-01, -1.3556e-01, -2.8668e-01,
1.1417e-01, 6.9243e-02, -7.4227e-02, -6.1268e-02, -6.1870e-02,
-3.5543e-01, -5.9122e-02, -1.1109e-01, -6.4636e-02, -1.2059e-01,
2.9009e-01, 1.6195e-01, 8.6600e-03, 7.4471e-02, -4.0417e-02,
1.3729e-01, 3.7715e-02, 6.6088e-02, 7.9698e-02, 2.2217e-01,
4.6145e-02, 5.3264e-02, 2.8850e-02, -2.7228e-01, 2.5493e-01,
-8.0775e-02, -2.1848e-01, -1.5226e-01, -9.6512e-02, 7.1429e-02,
-1.2123e-01, -1.7072e-02, -1.5918e-01, -1.3743e-01, 3.6923e-01,
2.0207e-01, -1.7557e-01, -2.0467e-01, 1.9471e-01, -5.1946e-03,
1.0098e-01, 9.6954e-03, 1.1310e-01, -1.7527e-01, -9.5537e-02,
2.1855e-01, 9.5689e-02, -5.9635e-02, 6.1443e-02, 1.3595e-01,
-8.0644e-02, -2.1573e-01, 4.6470e-02, -1.3586e-01, 7.4048e-02,
-4.2888e-02, 2.0872e-01, 1.1514e-02, 4.7596e-02, -2.9755e-01,
-6.6916e-02, 1.7111e-01, -1.1378e-01, -8.1982e-02, 1.7245e-01,
1.4705e-02, 1.7429e-01, 1.0439e-01, -1.3874e-01, 3.0701e-01,
-3.8905e-03, 4.2782e-02, -7.2560e-02, -1.5142e-01, 1.7251e-01,
1.0591e-01, 1.1676e-01, 7.6737e-02, -1.9903e-01, 1.6619e-01,
8.0159e-02, 1.1260e-01, -4.1150e-01, -4.2198e-01, 2.2584e-02,
6.0089e-02, 1.0475e-01, -1.1108e-01, 9.7691e-02, -2.2608e-01,
1.4698e-01, 3.7075e-02, -1.9827e-03, 4.0134e-02, 8.2940e-02,
-1.8627e-01, 3.0821e-02, 1.1510e-01, 1.3940e-01, 1.2892e-01,
3.2928e-02, 6.4728e-02, 9.2885e-02, -4.7629e-02, 5.0806e-02,
1.4518e-01, 2.4662e-01, 1.3465e-01, -4.6843e-02, -5.4671e-02,
5.6112e-02, -5.2758e-03, 5.3735e-02, -1.4860e-01, -1.9234e-01,
6.8454e-02, 1.0282e-01, -4.2927e-01, -7.6621e-02, -1.9427e-01,
8.8800e-02, -3.9533e-03, -2.8186e-01, -2.2814e-01, -2.8922e-01,
-7.8973e-02, 1.0183e-01, 1.6025e-01, -2.6884e-04, 6.8160e-02,
1.6013e-01, -1.7194e-02, 1.5115e-02, 1.9099e-01, 2.2653e-01,
-1.1130e-01, 1.8678e-01, -4.1655e-02, -6.5554e-02, 4.2916e-02,
-6.2751e-02, 7.5050e-02, 1.0198e-01, 1.7427e-02, 1.6215e-03,
2.2831e-02, -8.1368e-02, -4.2982e-02, -1.4738e-01, -4.5801e-02,
2.3623e-01]], grad_fn=<TanhBackward0>)
```



### 0.2.3 Étape 3 : Tester avec un batch plus grand (utiliser un DataLoader)

```
[12]: # --- 1. Charger les données ---
text, emotion = load_file("./dataset/train.txt")

# --- 2. Construire les vocabulaires ---
vocab = build_vocab_from_iterator(yield_tokens(text), specials=["<pad>",
↳ "<unk>"])
classes = build_vocab_from_iterator(yield_tokens(emotion))

# --- 3. Créer le dataset et le DataLoader ---
dataset = EmotionDataset(text, emotion, vocab, classes, max_len=20)
loader = DataLoader(dataset, batch_size=8, shuffle=True) # batch de 8 phrases

# --- 4. Initialiser le réseau ---
vocab_size = len(vocab)
emb_size = 64
hidden_size = 128
class_size = len(classes)

rnn = SimpleRNN(vocab_size, emb_size, hidden_size, class_size)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
rnn = rnn.to(device)

# --- 5. Récupérer un batch pour test ---
for X, y in loader:
    X, y = X.to(device), y.to(device)
    batch_size, seq_len, _ = X.shape

    # Initialiser l'état caché pour ce batch
    hidden = rnn.init_hidden(batch_size=batch_size, device=device)

    print(f"Batch shape: {X.shape} (batch_size={batch_size}, seq_len={seq_len},
↳ vocab_size={vocab_size})")

    # Passer la séquence mot par mot
    for t in range(seq_len):
        output, hidden = rnn(X[:, t, :], hidden)

    print("Output shape:", output.shape) # [batch_size, class_size]
    print("Hidden shape:", hidden.shape) # [batch_size, hidden_size]
    break # un seul batch suffit pour le test
```

Batch shape: torch.Size([8, 20, 15214]) (batch\_size=8, seq\_len=20,  
vocab\_size=15214)

Output shape: torch.Size([8, 6])

Hidden shape: torch.Size([8, 128])



```
[13]: from train_rnn import RNNTrainer
      trainer = RNNTrainer(
          train_path="./dataset/train.txt",
          test_path="./dataset/test.txt",
          epochs=20,
          batch_size=16,
          hidden_size=128,
          emb_size=64
      )

      trainer.train()
      trainer.evaluate()
```

Data prepared: 16000 samples, vocab=15214, classes=6

Model initialized on cpu

Epoch 01	Loss: 1.5799	Accuracy: 33.09%
Epoch 02	Loss: 1.5298	Accuracy: 36.24%
Epoch 03	Loss: 1.2457	Accuracy: 47.14%
Epoch 04	Loss: 1.0606	Accuracy: 55.30%
Epoch 05	Loss: 0.8809	Accuracy: 66.69%
Epoch 06	Loss: 0.8072	Accuracy: 71.12%
Epoch 07	Loss: 0.7745	Accuracy: 73.47%
Epoch 08	Loss: 0.7094	Accuracy: 76.62%
Epoch 09	Loss: 0.6356	Accuracy: 79.39%
Epoch 10	Loss: 0.5632	Accuracy: 81.84%
Epoch 11	Loss: 0.5056	Accuracy: 83.88%
Epoch 12	Loss: 0.4794	Accuracy: 84.74%
Epoch 13	Loss: 0.4310	Accuracy: 86.21%
Epoch 14	Loss: 0.4098	Accuracy: 87.00%
Epoch 15	Loss: 0.3836	Accuracy: 87.85%
Epoch 16	Loss: 0.3636	Accuracy: 88.31%
Epoch 17	Loss: 0.4054	Accuracy: 87.14%
Epoch 18	Loss: 0.3513	Accuracy: 88.69%
Epoch 19	Loss: 0.3159	Accuracy: 89.74%
Epoch 20	Loss: 0.3180	Accuracy: 90.18%

Training complete.

Test accuracy: 67.60%

[13]: 67.6

#### 0.2.4 OptimizedRNNTrainer

Cette classe reprend la logique du `RNNTrainer` mais ajoute :

- Un filtrage **TF-IDF adouci** pour ignorer les mots trop rares ou trop fréquents,

- Une **pondération de classes** pour limiter les effets du déséquilibre,
- Un modèle plus **léger et rapide** (hidden=64, embedding=64).



```
[3]: from optimized_rnn_trainer import OptimizedRNNTrainer
```

```
trainer = OptimizedRNNTrainer(  
    train_path="./dataset/train.txt",  
    test_path="./dataset/test.txt",  
    epochs=20,  
    batch_size=16,  
    hidden_size=64,  
    emb_size=64  
)  
  
trainer.train()  
trainer.evaluate()
```

Filtered vocab size: 15212

Training started...

Epoch 01	Loss: 1.7847	Accuracy: 26.10%
Epoch 02	Loss: 1.5026	Accuracy: 45.47%
Epoch 03	Loss: 1.0336	Accuracy: 64.67%
Epoch 04	Loss: 0.8062	Accuracy: 71.39%
Epoch 05	Loss: 0.7247	Accuracy: 74.16%
Epoch 06	Loss: 0.4811	Accuracy: 83.89%
Epoch 07	Loss: 0.3681	Accuracy: 88.29%
Epoch 08	Loss: 0.3034	Accuracy: 90.45%
Epoch 09	Loss: 0.2363	Accuracy: 92.62%
Epoch 10	Loss: 0.3223	Accuracy: 89.81%
Epoch 11	Loss: 0.3273	Accuracy: 89.57%
Epoch 12	Loss: 0.2207	Accuracy: 92.72%
Epoch 13	Loss: 0.1599	Accuracy: 94.76%
Epoch 14	Loss: 0.1986	Accuracy: 93.71%
Epoch 15	Loss: 0.1528	Accuracy: 94.74%
Epoch 16	Loss: 0.1548	Accuracy: 94.88%
Epoch 17	Loss: 0.1763	Accuracy: 94.35%
Epoch 18	Loss: 0.1850	Accuracy: 94.32%
Epoch 19	Loss: 0.1466	Accuracy: 95.26%
Epoch 20	Loss: 0.1358	Accuracy: 95.49%

Training finished.

\ Test accuracy: 63.40%

```
[3]: 63.4
```

```
[8]: import torch  
import matplotlib.pyplot as plt  
from sklearn.decomposition import PCA
```



```

from sklearn.manifold import TSNE

# --- Extraire la matrice d'embeddings ---
embeddings = trainer.rnn.i2e.weight.detach().cpu().numpy() # [vocab_size, ↵
↵emb_size]
words = list(trainer.vocab.mapping.keys())

# --- Réduction PCA ---
pca = PCA(n_components=2)
reduced = pca.fit_transform(embeddings)

max_words = min(len(words), len(reduced), 80)

plt.figure(figsize=(10, 8))
plt.scatter(reduced[:, 0], reduced[:, 1], alpha=0.6, s=12)
for i, word in enumerate(words[:max_words]):
    plt.text(reduced[i, 0], reduced[i, 1], word, fontsize=8)
plt.title("Projection PCA des embeddings de mots")
plt.xlabel("Composante principale 1")
plt.ylabel("Composante principale 2")
plt.show()

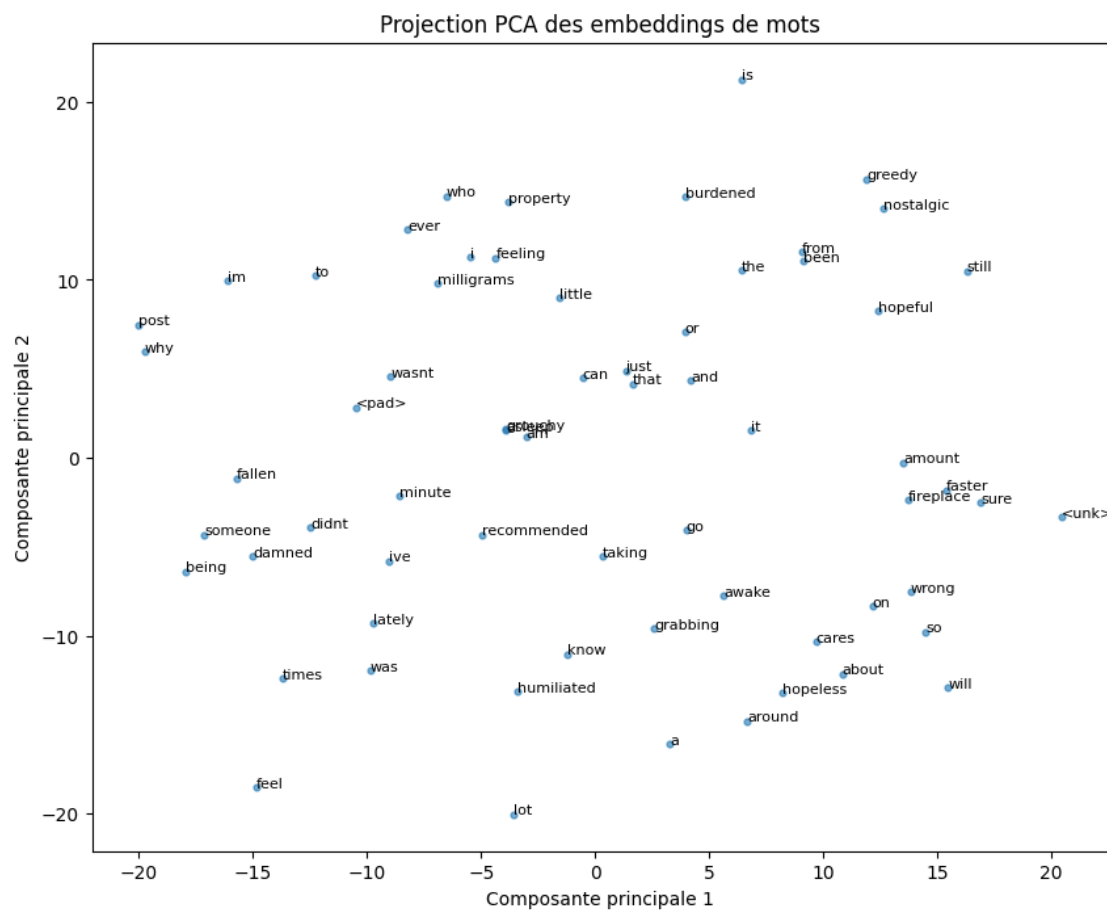
# --- Réduction t-SNE ---
try:
    tsne = TSNE(n_components=2, perplexity=30, max_iter=3000, random_state=42)
except TypeError:
    tsne = TSNE(n_components=2, perplexity=30, n_iter=3000, random_state=42)

reduced_tsne = tsne.fit_transform(embeddings)

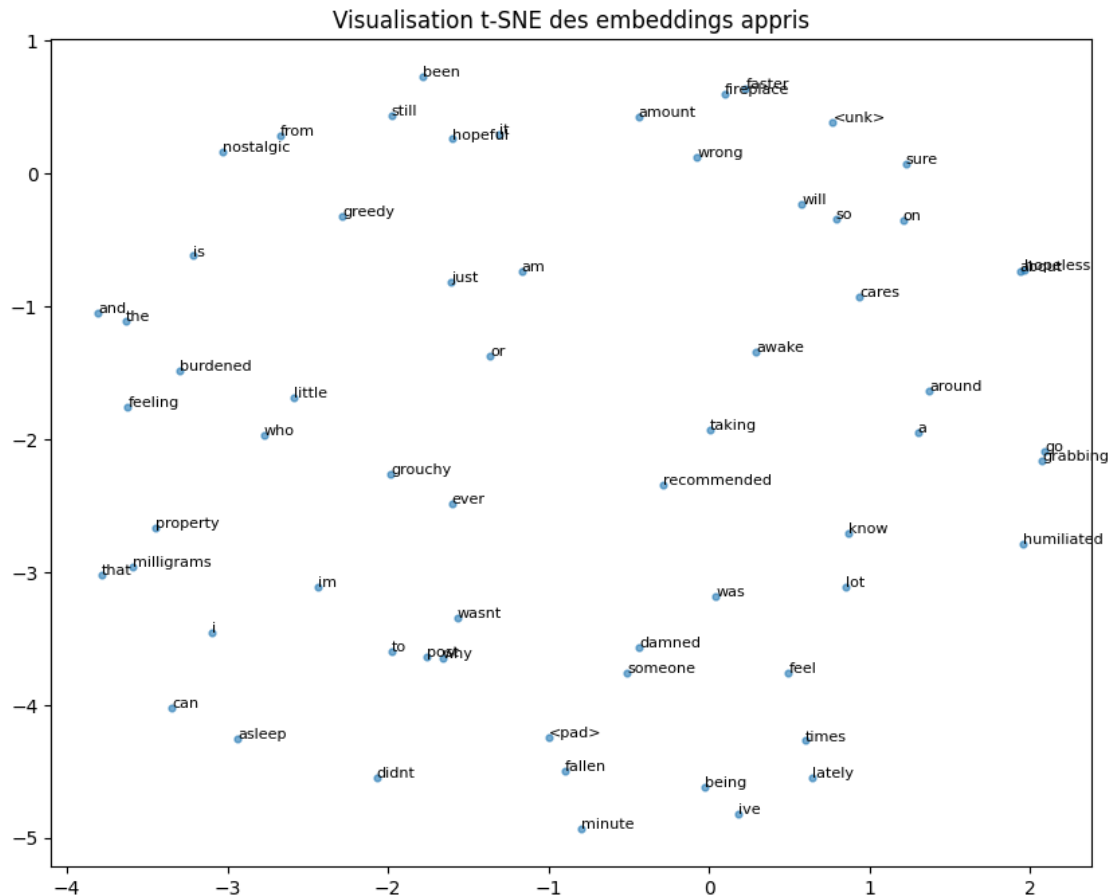
max_words = min(len(words), len(reduced_tsne), 80)
plt.figure(figsize=(10, 8))
plt.scatter(reduced_tsne[:, 0], reduced_tsne[:, 1], alpha=0.6, s=12)
for i, word in enumerate(words[:max_words]):
    plt.text(reduced_tsne[i, 0], reduced_tsne[i, 1], word, fontsize=8)
plt.title("Visualisation t-SNE des embeddings appris")
plt.show()

```









La projection PCA montre une organisation encore diffuse des mots, avec quelques regroupements sémantiques émergents : les termes émotionnels comme *hopeless*, *feel* ou *humiliated* apparaissent proches, tandis que les mots fonctionnels (*the*, *is*, *and*) restent regroupés ailleurs, et les tokens spéciaux comme *et* sont clairement isolés. La visualisation t-SNE affine cette structure en révélant des micro-clusters plus cohérents : les mots exprimant des émotions négatives gravitent ensemble, alors que les mots neutres ou de liaison forment des zones distinctes. Cela indique que le RNN a partiellement capté des relations de sens et de contexte, mais de manière encore superficielle en raison du faible nombre d’époques et de la taille réduite du corpus.

```
[ ]: # =====  
# Use the same vocab for context+word learning  
# =====  
from torch.utils.data import Dataset, DataLoader  
from data_processing import tokenizer, one_hot  
from rnn_model import SimpleRNN  
import torch.nn as nn  
import torch.optim as optim  
  
class ContextDataset(Dataset):
```



```

def __init__(self, texts, vocab, window=3, max_len=5):
    self.vocab = vocab
    self.max_len = max_len
    self.samples = []
    for s in texts:
        tokens = tokenizer(s)
        encoded = vocab(tokens)
        for i in range(1, len(encoded)):
            context = encoded[max(0, i - window):i]
            target = encoded[i]
            self.samples.append((context, target))

def __len__(self):
    return len(self.samples)

def __getitem__(self, idx):
    context, target = self.samples[idx]
    pad_id = self.vocab["<pad>"]
    if len(context) < self.max_len:
        context = [pad_id] * (self.max_len - len(context)) + context
    context_one_hot = one_hot(context, num_classes=len(self.vocab))
    X = torch.tensor(context_one_hot, dtype=torch.float32)
    y = torch.tensor(target, dtype=torch.long)
    return X, y

# --- Build context dataset using same vocab from trainer ---
auto_dataset = ContextDataset(trainer.train_dataset.texts, trainer.vocab)
auto_loader = DataLoader(auto_dataset, batch_size=32, shuffle=True)

# --- Initialize new model for auto-supervised learning ---
device = trainer.device
model_auto = SimpleRNN(len(trainer.vocab), 64, 64, len(trainer.vocab)).
    ↪to(device)
criterion_auto = nn.NLLLoss()
optimizer_auto = optim.Adam(model_auto.parameters(), lr=0.002)

print("\n Auto-supervised training started...\n")
for epoch in range(5):
    model_auto.train()
    total_loss = 0
    for X, y in auto_loader:
        X, y = X.to(device), y.to(device)
        hidden = model_auto.init_hidden(X.size(0), device=device)
        for t in range(X.size(1)):
            output, hidden = model_auto(X[:, t, :], hidden)
            loss = criterion_auto(output, y)

```



```
optimizer_auto.zero_grad()
loss.backward()
optimizer_auto.step()
total_loss += loss.item()
print(f"Epoch {epoch+1} | Loss: {total_loss / len(auto_loader):.4f}")

print("\nAuto-supervised context→word training done.\n")
```

Auto-supervised training started...

[ ]: