Cours Machine Learning

Chapitre 08

Deep Learning : RNN et Transformers

Objectifs d'apprentissage :

— Comprendre les RNN (Recurrent Neural Networks) pour les séquences
— Maîtriser LSTM et GRU pour résoudre les problèmes de vanishing gradient
— Découvrir le mécanisme d'attention et son importance
— Étudier l'architecture Transformer et son fonctionnement
— Appliquer ces modèles au NLP et aux séries temporelles

Prérequis : Chapitre 06 (Réseaux de Neurones Fondamentaux)

Durée estimée : 8-10 heures

Notebooks : `08_demo_*.ipynb`

Cours ML - Sandbox-ML
Version 1.0 - 2026

Table des matières

1 Introduction aux Données Séquentielles

1.1 Qu'est-ce qu'une séquence ?

Une **séquence** est une suite ordonnée d'éléments où l'ordre a une importance cruciale.

Exemple : Exemples de séquences

- **Texte** : "Le chat mange la souris" (ordre des mots sens)
- **Audio** : Signal sonore échantillonné dans le temps
- **Vidéo** : Séquence d'images (frames)
- **Séries temporelles** : Cours de bourse, température, trafic web
- **ADN** : Séquence de nucléotides (A, C, G, T)

1.2 Limitations des MLP et CNN pour les séquences

MLP :

- Taille d'entrée fixe (impossible pour séquences de longueur variable)
- Pas de mémoire du contexte précédent
- Nombre de paramètres explose avec la longueur

CNN :

- Peut traiter des séquences avec Conv1D
- Champ récepteur limité (contexte local uniquement)
- Pas de mémoire à long terme

1.3 Types de tâches séquentielles

TABLE 1 – Architectures séquence-to-X

Type	Description	Exemple
One-to-One	Entrée fixe Sortie fixe	Classification d'image (CNN)
One-to-Many	Entrée fixe Séquence	Image captioning
Many-to-One	Séquence Sortie fixe	Sentiment analysis
Many-to-Many	Séquence Séquence (même longueur)	Traduction, génération texte Étiquetage de séquences (NER)
	Séquence Séquence (longueur différente)	Traduction automatique

2 Recurrent Neural Networks (RNN)

2.1 Architecture

Définition : RNN

Un RNN traite une séquence $(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_T)$ en maintenant un **état caché** $\mathbf{h}_t$ qui se propage dans le temps :

$$\mathbf{h}_t = \tanh(\mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{W}_{xh}\mathbf{x}_t + \mathbf{b}_h) \tag{1}$$

$$\mathbf{y}_t = \mathbf{W}_{hy}\mathbf{h}_t + \mathbf{b}_y \tag{2}$$

où :
- $\mathbf{h}_t \in \mathbb{R}^h$: état caché au temps t
- $\mathbf{x}_t \in \mathbb{R}^d$: entrée au temps t
- $\mathbf{y}_t \in \mathbb{R}^k$: sortie au temps t
- $\mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$: poids récurrents
- $\mathbf{W}_{xh} \in \mathbb{R}^{h \times d}$: poids d'entrée
- $\mathbf{W}_{hy} \in \mathbb{R}^{k \times h}$: poids de sortie

Caractéristique clé : Les mêmes poids $\mathbf{W}_{hh}, \mathbf{W}_{xh}, \mathbf{W}_{hy}$ sont partagés à chaque pas de temps capacité à traiter des séquences de longueur variable.

2.2 Déroulement dans le temps (Unfolding)

On peut "dérouler" le RNN dans le temps :

$$\begin{aligned}
\mathbf{h}_1 &= \tanh(\mathbf{W}_{hh}\mathbf{h}_0 + \mathbf{W}_{xh}\mathbf{x}_1 + \mathbf{b}_h) \\
\mathbf{h}_2 &= \tanh(\mathbf{W}_{hh}\mathbf{h}_1 + \mathbf{W}_{xh}\mathbf{x}_2 + \mathbf{b}_h) \\
\mathbf{h}_3 &= \tanh(\mathbf{W}_{hh}\mathbf{h}_2 + \mathbf{W}_{xh}\mathbf{x}_3 + \mathbf{b}_h) \\
&\vdots
\end{aligned}$$

Initialisation : $\mathbf{h}_0 = \mathbf{0}$ (ou appris)

2.3 Backpropagation Through Time (BPTT)

Pour entraîner un RNN, on utilise **BPTT** : backpropagation appliquée au réseau déroulé.

Forward pass : Calculer $\mathbf{h}_1, \ldots, \mathbf{h}_T$ et $\mathbf{y}_1, \ldots, \mathbf{y}_T$

Backward pass : Calculer gradients en remontant dans le temps :

$$\frac{\partial L}{\partial \mathbf{h}_t} = \frac{\partial L}{\partial \mathbf{y}_t}\frac{\partial \mathbf{y}_t}{\partial \mathbf{h}_t} + \frac{\partial L}{\partial \mathbf{h}_{t+1}}\frac{\partial \mathbf{h}_{t+1}}{\partial \mathbf{h}_t} \tag{3}$$

2.4 Problème du Vanishing/Exploding Gradient

Attention

Le gradient se propage à travers tous les pas de temps :

$$\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_0} = \prod_{k=1}^{t} \frac{\partial \mathbf{h}_k}{\partial \mathbf{h}_{k-1}} = \prod_{k=1}^{t} \mathbf{W}_{hh}^T \cdot \text{diag}(\tanh'(\mathbf{z}_k)) \tag{4}$$

Vanishing gradient : Si $\|\mathbf{W}_{hh}\| < 1$, le gradient $\to 0$ exponentiellement.
- Le RNN ne peut pas apprendre de dépendances à long terme

Exploding gradient : Si $\|\mathbf{W}_{hh}\| > 1$, le gradient $\to \infty$.
- Solution : **gradient clipping** (limiter la norme du gradient)

2.5 Implémentation simple

```python
import numpy as np

class SimpleRNN:
    def __init__(self, input_dim, hidden_dim, output_dim):
        self.hidden_dim = hidden_dim

        # Initialisation Xavier
        self.Wxh = np.random.randn(hidden_dim, input_dim) * 0.01
        self.Whh = np.random.randn(hidden_dim, hidden_dim) * 0.01
        self.Why = np.random.randn(output_dim, hidden_dim) * 0.01
        self.bh = np.zeros((hidden_dim, 1))
        self.by = np.zeros((output_dim, 1))

    def forward(self, inputs):
        """
        inputs: liste de vecteurs (seq_len, input_dim)
        """
        h = np.zeros((self.hidden_dim, 1))  # h0
        self.last_inputs = inputs
        self.last_hs = {0: h}

        # Forward pass
        for t, x in enumerate(inputs):
            x = x.reshape(-1, 1)
            h = np.tanh(self.Wxh @ x + self.Whh @ h + self.bh)
            self.last_hs[t + 1] = h

        # Output au dernier pas de temps (many-to-one)
        y = self.Why @ h + self.by
        return y, h

    def backward(self, dy, learning_rate=0.001):
        """
        BPTT simplifié (output au dernier pas uniquement)
        """
        n = len(self.last_inputs)

        # Gradients accumulés
        dWxh, dWhh, dWhy = np.zeros_like(self.Wxh), \
                           np.zeros_like(self.Whh), \
                           np.zeros_like(self.Why)
        dbh, dby = np.zeros_like(self.bh), np.zeros_like(self.by)

        # Gradient output
        dWhy += dy @ self.last_hs[n].T
        dby += dy

        # Backprop through time
        dh = self.Why.T @ dy
```

```
50
51          for t in reversed(range(n)):
52              temp = (1 - self.last_hs[t + 1] ** 2) * dh  # tanh'
53              dbh += temp
54              dWxh += temp @ self.last_inputs[t].reshape(1, -1)
55              dWhh += temp @ self.last_hs[t].T
56              dh = self.Whh.T @ temp
57
58          # Gradient clipping
59          for grad in [dWxh, dWhh, dWhy, dbh, dby]:
60              np.clip(grad, -1, 1, out=grad)
61
62          # Update
63          self.Wxh -= learning_rate * dWxh
64          self.Whh -= learning_rate * dWhh
65          self.Why -= learning_rate * dWhy
66          self.bh -= learning_rate * dbh
67          self.by -= learning_rate * dby
```

Listing 1 – RNN vanilla from scratch

3 Long Short-Term Memory (LSTM)

3.1 Motivation

LSTM résout le problème du vanishing gradient en introduisant une **cellule mémoire** $\mathbf{c}_t$ et des **portes (gates)** qui contrôlent le flux d'information.

3.2 Architecture

Définition : LSTM

Un LSTM a 3 portes et 1 cellule mémoire :

$$\mathbf{f}_t = \text{sigmoid}(\mathbf{W}_f[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_f) \quad \text{(forget gate)} \tag{5}$$

$$\mathbf{i}_t = \text{sigmoid}(\mathbf{W}_i[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_i) \quad \text{(input gate)} \tag{6}$$

$$\tilde{\mathbf{c}}_t = \tanh(\mathbf{W}_c[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_c) \quad \text{(candidate cell)} \tag{7}$$

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t \quad \text{(cell state)} \tag{8}$$

$$\mathbf{o}_t = \text{sigmoid}(\mathbf{W}_o[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_o) \quad \text{(output gate)} \tag{9}$$

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t) \quad \text{(hidden state)} \tag{10}$$

où $\odot$ est le produit élément par élément (Hadamard).

3.3 Interprétation des portes

- **Forget gate** ($\mathbf{f}_t$) : Décide quelle information de $\mathbf{c}_{t-1}$ oublier
  - $f_t = 0$: oublier complètement
  - $f_t = 1$: conserver complètement

— **Input gate** ($\mathbf{i}_t$) : Décide quelle nouvelle information stocker dans $\mathbf{c}_t$
  — $i_t = 0$: ignorer la nouvelle information
  — $i_t = 1$: stocker complètement
— **Output gate** ($\mathbf{o}_t$) : Décide quelle partie de $\mathbf{c}_t$ exposer dans $\mathbf{h}_t$

Flux d'information :

$$\mathbf{c}_t = \underbrace{\mathbf{f}_t \odot \mathbf{c}_{t-1}}_{\text{mémoire passée}} + \underbrace{\mathbf{i}_t \odot \tilde{\mathbf{c}}_t}_{\text{nouvelle info}} \tag{11}$$

Astuce

La cellule $\mathbf{c}_t$ agit comme une "autoroute" permettant au gradient de se propager sans atténuation (si $\mathbf{f}_t \approx 1$). Cela résout le vanishing gradient !

3.4 Nombre de paramètres

Pour un LSTM avec h unités cachées et d inputs :

$$\text{Params} = 4 \times (h \times (h + d) + h) = 4h(h + d + 1) \tag{12}$$

Le facteur 4 vient des 4 matrices de poids (forget, input, candidate, output).

Exemple : LSTM(128) avec input(100)

$$\text{Params} = 4 \times 128 \times (128 + 100 + 1) = 117{,}248$$

3.5 Implémentation PyTorch

```python
import torch
import torch.nn as nn

class LSTMModel(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim, num_layers=1):
        super(LSTMModel, self).__init__()

        self.hidden_dim = hidden_dim
        self.num_layers = num_layers

        # LSTM layer
        self.lstm = nn.LSTM(input_dim, hidden_dim, num_layers,
                            batch_first=True)

        # Fully-connected output
        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        """
        x: (batch, seq_len, input_dim)
        """
```

```python
        # Initialiser h0, c0
        h0 = torch.zeros(self.num_layers, x.size(0),
                         self.hidden_dim).to(x.device)
        c0 = torch.zeros(self.num_layers, x.size(0),
                         self.hidden_dim).to(x.device)

        # LSTM forward
        # out: (batch, seq_len, hidden_dim)
        out, (hn, cn) = self.lstm(x, (h0, c0))

        # Prendre le dernier timestep (many-to-one)
        out = self.fc(out[:, -1, :])
        return out

# Exemple d'utilisation
model = LSTMModel(input_dim=10, hidden_dim=128, output_dim=5)

# Séquence de longueur 20
x = torch.randn(32, 20, 10)  # (batch=32, seq_len=20, input_dim=10)
output = model(x)
print(output.shape)  # (32, 5)
```

Listing 2 – LSTM avec PyTorch

4 Gated Recurrent Unit (GRU)

4.1 Architecture

GRU est une variante simplifiée du LSTM avec **2 portes** au lieu de 3.

Définition : GRU

$$\mathbf{z}_t = \text{sigmoid}(\mathbf{W}_z[\mathbf{h}_{t-1}, \mathbf{x}_t]) \quad \text{(update gate)} \tag{13}$$

$$\mathbf{r}_t = \text{sigmoid}(\mathbf{W}_r[\mathbf{h}_{t-1}, \mathbf{x}_t]) \quad \text{(reset gate)} \tag{14}$$

$$\tilde{\mathbf{h}}_t = \tanh(\mathbf{W}[\mathbf{r}_t \odot \mathbf{h}_{t-1}, \mathbf{x}_t]) \quad \text{(candidate)} \tag{15}$$

$$\mathbf{h}_t = (1 - \mathbf{z}_t) \odot \mathbf{h}_{t-1} + \mathbf{z}_t \odot \tilde{\mathbf{h}}_t \tag{16}$$

4.2 Différences avec LSTM

Table 2 – LSTM vs GRU

	LSTM	**GRU**
Nombre de portes	3	2
Cellule mémoire séparée	Oui ($\mathbf{c}_t$)	Non (seulement $\mathbf{h}_t$)
Paramètres	$4h(h+d+1)$	$3h(h+d+1)$
Vitesse	Plus lent	Plus rapide
Performance	Légèrement meilleure	Comparable

Astuce

Règle pratique :

- LSTM : Séquences très longues, dépendances complexes
- GRU : Séquences courtes/moyennes, plus rapide, moins de paramètres
- En pratique, essayer les deux et comparer !

4.3 Implémentation PyTorch

```python
class GRUModel(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim, num_layers=1):
        super(GRUModel, self).__init__()

        self.hidden_dim = hidden_dim
        self.num_layers = num_layers

        self.gru = nn.GRU(input_dim, hidden_dim, num_layers,
                          batch_first=True)
        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        h0 = torch.zeros(self.num_layers, x.size(0),
                         self.hidden_dim).to(x.device)

        out, hn = self.gru(x, h0)
        out = self.fc(out[:, -1, :])
        return out
```

Listing 3 – GRU avec PyTorch

5 Bidirectional RNN/LSTM/GRU

5.1 Motivation

Un RNN classique ne voit que le **contexte passé**. Pour certaines tâches (NER, POS tagging), le contexte **futur** est aussi important.

Exemple : Prédiction de mots

Phrase : "Le _____ mange la souris"

- Contexte passé : "Le"
- Contexte futur : "mange la souris" animal carnivore "chat"

5.2 Architecture

Définition : Bidirectional RNN

Un Bi-RNN a deux RNN :

- **Forward RNN** : lit la séquence de gauche à droite $\overrightarrow{\mathbf{h}}_t$
- **Backward RNN** : lit la séquence de droite à gauche $\overleftarrow{\mathbf{h}}_t$

La sortie finale est la concaténation :

$$\mathbf{h}_t = [\overrightarrow{\mathbf{h}}_t; \overleftarrow{\mathbf{h}}_t] \tag{17}$$

Nombre de paramètres : Doublé (2 RNN indépendants)

5.3 Implémentation

```python
class BiLSTM(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(BiLSTM, self).__init__()

        # bidirectional=True
        self.lstm = nn.LSTM(input_dim, hidden_dim,
                            batch_first=True, bidirectional=True)

        # hidden_dim * 2 car bidirectionnel
        self.fc = nn.Linear(hidden_dim * 2, output_dim)

    def forward(self, x):
        # out: (batch, seq_len, hidden_dim * 2)
        out, _ = self.lstm(x)
        out = self.fc(out[:, -1, :])  # Dernier timestep
        return out
```

Listing 4 – Bidirectional LSTM

6 Mécanisme d'Attention

6.1 Motivation : Problème du goulot d'étranglement

En seq2seq (traduction), le décodeur doit tout comprendre à partir d'un seul vecteur contexte $\mathbf{c}$:

$$\text{Encoder} : (x_1, \ldots, x_n) \rightarrow \mathbf{c} \rightarrow \text{Decoder} : (y_1, \ldots, y_m) \tag{18}$$

Problème : $\mathbf{c}$ est un goulot d'étranglement, surtout pour des séquences longues.

6.2 Attention de Bahdanau

Définition : Attention Mechanism

Au lieu d'un vecteur contexte fixe, on calcule un vecteur contexte **dynamique** $\mathbf{c}_t$ à chaque pas de décodage :

$$e_{t,i} = \text{score}(\mathbf{s}_{t-1}, \mathbf{h}_i) \quad \text{(score d'attention)} \tag{19}$$

$$\alpha_{t,i} = \frac{\exp(e_{t,i})}{\sum_{j=1}^{n} \exp(e_{t,j})} \quad \text{(poids d'attention)} \tag{20}$$

$$\mathbf{c}_t = \sum_{i=1}^{n} \alpha_{t,i} \mathbf{h}_i \quad \text{(contexte pondéré)} \tag{21}$$

où :

- $\mathbf{s}_{t-1}$: état caché du décodeur
- $\mathbf{h}_i$: états cachés de l'encodeur
- $\alpha_{t,i}$: attention sur le mot i de l'entrée

6.3 Fonctions de score

Dot product :

$$\text{score}(\mathbf{s}, \mathbf{h}) = \mathbf{s}^T \mathbf{h} \tag{22}$$

General (bilinear) :

$$\text{score}(\mathbf{s}, \mathbf{h}) = \mathbf{s}^T \mathbf{W}_a \mathbf{h} \tag{23}$$

Additive (Bahdanau) :

$$\text{score}(\mathbf{s}, \mathbf{h}) = \mathbf{v}_a^T \tanh(\mathbf{W}_1 \mathbf{s} + \mathbf{W}_2 \mathbf{h}) \tag{24}$$

6.4 Scaled Dot-Product Attention

Version utilisée dans les Transformers :

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V} \tag{25}$$

où :

- $\mathbf{Q}$ (Queries) : ce qu'on cherche
- $\mathbf{K}$ (Keys) : ce qu'on a
- $\mathbf{V}$ (Values) : ce qu'on renvoie
- d_k : dimension des clés (facteur de normalisation)

Interprétation :

1. Calculer similarité entre $\mathbf{Q}$ et $\mathbf{K}$: $\mathbf{Q}\mathbf{K}^T$
2. Normaliser par $\sqrt{d_k}$ pour éviter valeurs trop grandes
3. Appliquer softmax pour obtenir poids d'attention
4. Pondérer les valeurs $\mathbf{V}$

7 Transformers

7.1 Motivation

Limites des RNN :

- Traitement séquentiel (pas de parallélisation)
- Difficulté avec dépendances à très long terme
- Lent à entraîner

Transformer (Vaswani et al., 2017) :

- Uniquement basé sur l'attention (pas de récurrence)
- Complètement parallélisable
- Capture dépendances à longue distance facilement
- State-of-the-art en NLP (BERT, GPT, T5, etc.)

7.2 Architecture globale

Encoder-Decoder avec 6 couches chacun (original paper) :

- **Encoder** : Traite la séquence d'entrée
- **Decoder** : Génère la séquence de sortie (autorégressif)

7.3 Multi-Head Attention

Définition : Multi-Head Attention

Au lieu d'une seule attention, on calcule h attentions en parallèle avec des projections différentes :

$$\text{head}_i = \text{Attention}(\mathbf{Q}\mathbf{W}_i^Q, \mathbf{K}\mathbf{W}_i^K, \mathbf{V}\mathbf{W}_i^V) \tag{26}$$

$$\text{MultiHead}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)\mathbf{W}^O \tag{27}$$

où $\mathbf{W}_i^Q, \mathbf{W}_i^K, \mathbf{W}_i^V$ sont des matrices de projection apprenables.

Avantages :

- Chaque tête peut se concentrer sur des aspects différents (syntaxe, sémantique, etc.)
- Plus expressif qu'une seule attention

7.4 Positional Encoding

Problème : Sans récurrence, le Transformer ne connaît pas l'ordre des mots !

Solution : Ajouter un **encodage de position** à chaque embedding :

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d}}\right) \tag{28}$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d}}\right) \tag{29}$$

où pos est la position et i la dimension.

$$\mathbf{x}_{\text{input}} = \text{Embedding}(\text{token}) + \text{PositionalEncoding}(pos) \tag{30}$$

7.5 Encoder Layer

Une couche encoder contient :

1. **Multi-Head Self-Attention**
2. **Add & Norm** (Residual connection + Layer Normalization)
3. **Feed-Forward Network** (2 couches FC avec ReLU)
4. **Add & Norm**

$$\mathbf{z} = \text{LayerNorm}(\mathbf{x} + \text{MultiHeadAttention}(\mathbf{x}, \mathbf{x}, \mathbf{x})) \tag{31}$$

$$\text{output} = \text{LayerNorm}(\mathbf{z} + \text{FFN}(\mathbf{z})) \tag{32}$$

7.6 Decoder Layer

Une couche decoder contient :

1. **Masked Multi-Head Self-Attention** (ne voit que le passé)
2. **Add & Norm**
3. **Multi-Head Cross-Attention** (attention sur l'encodeur)
4. **Add & Norm**
5. **Feed-Forward Network**
6. **Add & Norm**

Masking : En génération, le décodeur ne doit voir que les tokens précédents (autorégressif).

7.7 Implémentation simplifiée

```python
import torch
import torch.nn.functional as F

def scaled_dot_product_attention(Q, K, V, mask=None):
    """
    Q, K, V: (batch, seq_len, d_k)
    """
    d_k = Q.size(-1)

    # Scores: (batch, seq_len, seq_len)
    scores = torch.matmul(Q, K.transpose(-2, -1)) / torch.sqrt(torch.tensor(d_k, dtype=torch.float32))

    # Masking (optionnel)
    if mask is not None:
        scores = scores.masked_fill(mask == 0, -1e9)

    # Softmax
    attention_weights = F.softmax(scores, dim=-1)
```

```
19
20      # Weighted sum
21      output = torch.matmul(attention_weights, V)
22
23      return output, attention_weights
```

Listing 5 – Scaled Dot-Product Attention

```
1  class MultiHeadAttention(nn.Module):
2      def __init__(self, d_model, num_heads):
3          super(MultiHeadAttention, self).__init__()
4          assert d_model % num_heads == 0
5
6          self.d_model = d_model
7          self.num_heads = num_heads
8          self.d_k = d_model // num_heads
9
10         # Linear projections
11         self.W_q = nn.Linear(d_model, d_model)
12         self.W_k = nn.Linear(d_model, d_model)
13         self.W_v = nn.Linear(d_model, d_model)
14         self.W_o = nn.Linear(d_model, d_model)
15
16     def split_heads(self, x):
17         """Split into multiple heads"""
18         batch_size, seq_len, d_model = x.size()
19         return x.view(batch_size, seq_len, self.num_heads, self.d_k).
       transpose(1, 2)
20
21     def forward(self, Q, K, V, mask=None):
22         batch_size = Q.size(0)
23
24         # Linear projections
25         Q = self.split_heads(self.W_q(Q))  # (batch, num_heads, seq_len,
       d_k)
26         K = self.split_heads(self.W_k(K))
27         V = self.split_heads(self.W_v(V))
28
29         # Scaled dot-product attention
30         attn_output, _ = scaled_dot_product_attention(Q, K, V, mask)
31
32         # Concatenate heads
33         attn_output = attn_output.transpose(1, 2).contiguous().view(
34             batch_size, -1, self.d_model
35         )
36
37         # Final linear
38         output = self.W_o(attn_output)
39         return output
```

Listing 6 – Multi-Head Attention

```python
class TransformerEncoderLayer(nn.Module):
    def __init__(self, d_model, num_heads, d_ff, dropout=0.1):
        super(TransformerEncoderLayer, self).__init__()

        self.self_attn = MultiHeadAttention(d_model, num_heads)
        self.feed_forward = nn.Sequential(
            nn.Linear(d_model, d_ff),
            nn.ReLU(),
            nn.Linear(d_ff, d_model)
        )

        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, mask=None):
        # Self-attention + residual + norm
        attn_output = self.self_attn(x, x, x, mask)
        x = self.norm1(x + self.dropout(attn_output))

        # Feed-forward + residual + norm
        ff_output = self.feed_forward(x)
        x = self.norm2(x + self.dropout(ff_output))

        return x
```

Listing 7 – Transformer Encoder Layer

8 Applications NLP

8.1 Modèles pré-entraînés

8.1.1 BERT (Bidirectional Encoder Representations from Transformers)

Architecture : Encoder Transformer (12 ou 24 couches)

Pré-entraînement :

- **Masked Language Modeling (MLM)** : Prédire les mots masqués
- **Next Sentence Prediction (NSP)** : Prédire si phrase B suit phrase A

Fine-tuning : Classification, NER, Q&A, etc.

8.1.2 GPT (Generative Pre-trained Transformer)

Architecture : Decoder Transformer (autorégressif)

Pré-entraînement : Language modeling (prédire token suivant)

Modèles :

- GPT-2 (1.5B params)
- GPT-3 (175B params)
- GPT-4 (1.76T params estimé)

8.1.3 T5, BART, RoBERTa, etc.

Nombreuses variantes avec différentes stratégies de pré-entraînement.

8.2 Utilisation avec HuggingFace Transformers

```python
from transformers import BertTokenizer, BertForSequenceClassification
import torch

# Charger modèle pré-entraîné
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
model = BertForSequenceClassification.from_pretrained(
    'bert-base-uncased',
    num_labels=2  # Binary classification
)

# Texte à classifier
text = "This movie is fantastic!"
inputs = tokenizer(text, return_tensors='pt', padding=True, truncation=
    True)

# Forward
outputs = model(**inputs)
logits = outputs.logits
probs = torch.softmax(logits, dim=-1)

print(f"Positive: {probs[0, 1]:.2f}")
print(f"Negative: {probs[0, 0]:.2f}")
```

Listing 8 – BERT pour classification

```python
from transformers import GPT2LMHeadModel, GPT2Tokenizer

tokenizer = GPT2Tokenizer.from_pretrained('gpt2')
model = GPT2LMHeadModel.from_pretrained('gpt2')

# Prompt
prompt = "Once upon a time"
inputs = tokenizer.encode(prompt, return_tensors='pt')

# Génération
outputs = model.generate(
    inputs,
    max_length=50,
    num_return_sequences=1,
    temperature=0.7,
    top_p=0.9,
    do_sample=True
)

generated_text = tokenizer.decode(outputs[0], skip_special_tokens=True)
print(generated_text)
```

Listing 9 – GPT-2 pour génération de texte

9 Séries Temporelles

9.1 Prédiction

```python
import numpy as np
import torch
import torch.nn as nn

# Générer série temporelle synthétique
t = np.linspace(0, 100, 1000)
data = np.sin(t) + 0.1 * np.random.randn(1000)

# Créer séquences (window = 50)
def create_sequences(data, window=50):
    X, y = [], []
    for i in range(len(data) - window):
        X.append(data[i:i+window])
        y.append(data[i+window])
    return np.array(X), np.array(y)

X, y = create_sequences(data, window=50)
X = torch.FloatTensor(X).unsqueeze(-1)  # (N, 50, 1)
y = torch.FloatTensor(y)

# Modèle LSTM
class TimeSeriesLSTM(nn.Module):
    def __init__(self, input_dim=1, hidden_dim=64, num_layers=2):
        super(TimeSeriesLSTM, self).__init__()
        self.lstm = nn.LSTM(input_dim, hidden_dim, num_layers,
    batch_first=True)
        self.fc = nn.Linear(hidden_dim, 1)

    def forward(self, x):
        out, _ = self.lstm(x)
        out = self.fc(out[:, -1, :])  # Dernier timestep
        return out.squeeze()

model = TimeSeriesLSTM()
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

# Training
num_epochs = 100
for epoch in range(num_epochs):
    model.train()
    optimizer.zero_grad()

```

```
    outputs = model(X)
    loss = criterion(outputs, y)

    loss.backward()
    optimizer.step()

    if (epoch + 1) % 10 == 0:
        print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}')
)
```

Listing 10 – LSTM pour prédiction de séries temporelles

10 Bonnes Pratiques

10.1 RNN/LSTM/GRU

- **Gradient clipping** : Limiter norme du gradient (5-10)
- **Layer Normalization** : Stabilise l'entraînement
- **Dropout** : Sur les connexions non-récurrentes uniquement
- **Teacher Forcing** : En seq2seq, utiliser vraies sorties pendant entraînement
- **Beam Search** : Pour génération (au lieu de greedy)

10.2 Transformers

- **Learning Rate Warmup** : Augmenter LR progressivement au début
- **Label Smoothing** : Régularisation pour classification
- **Dropout** : Sur attention et FFN
- **Layer Normalization** : Avant ou après chaque sous-couche
- **Gradient Accumulation** : Pour simuler grands batch sizes

11 Résumé du Chapitre

11.1 Points Clés

- **RNN** : État caché récurrent, vanishing/exploding gradient
- **LSTM** : 3 portes (forget, input, output) + cellule mémoire
- **GRU** : Variante simplifiée avec 2 portes
- **Bidirectional** : Contexte passé + futur
- **Attention** : Pondération dynamique des inputs
- **Transformer** : Uniquement attention, parallélisable, state-of-the-art NLP
- **BERT** : Encoder pré-entraîné (MLM)
- **GPT** : Decoder autorégressif (LM)

11.2 Formules Essentielles

Formules à retenir

RNN :

$$\mathbf{h}_t = \tanh(\mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{W}_{xh}\mathbf{x}_t + \mathbf{b})$$

LSTM (simplifié) :

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t$$
$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t)$$

Scaled Dot-Product Attention :

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V}$$

12 Exercices

12.1 Questions de compréhension

1. Pourquoi le RNN vanilla souffre-t-il du vanishing gradient ?
2. Expliquer le rôle de chaque porte dans un LSTM.
3. Quelle est la différence principale entre LSTM et GRU ?
4. Pourquoi utiliser un Bidirectional RNN pour le NER ?
5. Comment le mécanisme d'attention résout-il le problème du goulot d'étranglement ?
6. Pourquoi le Transformer a-t-il besoin de positional encoding ?

12.2 Exercices pratiques

1. **Prédiction de séries temporelles**
   — Implémenter un LSTM pour prédire une série temporelle
   — Comparer LSTM vs GRU
   — Visualiser les prédictions
2. **Sentiment Analysis**
   — Fine-tuner BERT sur un dataset de reviews (IMDB, Yelp)
   — Comparer avec un LSTM from scratch
3. **Génération de texte**
   — Entraîner un RNN character-level sur Shakespeare
   — Générer du texte avec temperature sampling
4. **Attention Visualization**
   — Implémenter attention de Bahdanau
   — Visualiser les poids d'attention sur une tâche de traduction

Solutions disponibles dans `08_exercices.ipynb` *(solutions intégrées dans le notebook)*

13 Pour Aller Plus Loin

13.1 Lectures Recommandées

— Hochreiter & Schmidhuber (1997) - "Long Short-Term Memory"
— Bahdanau et al. (2014) - "Neural Machine Translation by Jointly Learning to Align and Translate"
— Vaswani et al. (2017) - "Attention Is All You Need" (Transformer)
— Devlin et al. (2018) - "BERT : Pre-training of Deep Bidirectional Transformers"
— Radford et al. (2019) - "Language Models are Unsupervised Multitask Learners" (GPT-2)

13.2 Ressources

— The Illustrated Transformer : https://jalammar.github.io/illustrated-transformer/
— HuggingFace Transformers : https://huggingface.co/docs/transformers/
— Sequence Models (Coursera) : Andrew Ng
— Annotated Transformer : https://nlp.seas.harvard.edu/annotated-transformer/

13.3 Prochaines Étapes

Chapitre suivant recommandé : **Chapitre 09 - Reinforcement Learning**

Le reinforcement learning permet d'entraîner des agents à prendre des décisions séquentielles pour maximiser une récompense.

Références

1. Hochreiter, S., & Schmidhuber, J. (1997). "Long short-term memory". *Neural computation*, 9(8), 1735-1780.
2. Cho, K., et al. (2014). "Learning phrase representations using RNN encoder-decoder for statistical machine translation". *EMNLP*.
3. Bahdanau, D., Cho, K., & Bengio, Y. (2014). "Neural machine translation by jointly learning to align and translate". *arXiv :1409.0473*.
4. Vaswani, A., et al. (2017). "Attention is all you need". *NIPS*.
5. Devlin, J., et al. (2018). "BERT : Pre-training of deep bidirectional transformers for language understanding". *arXiv :1810.04805*.
6. Radford, A., et al. (2019). "Language models are unsupervised multitask learners". *OpenAI blog*.