Extraction des entités nommées

ÉTAPE 1: Importer les librairies

In [1]:
```
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
import re
import os
from typing import List, Dict, Tuple, Optional
import importlib
import models
import utils
importlib.reload(models)
importlib.reload(utils)
from models.models import *
from utils.datasetloader import NERDataset
from utils.fonctions import (
    load_jnlpba_dataset, load_ncbi_dataset,prepare_ncbi_for_ner,create_embedding
)
from utils.creation_vocabulaire import create_vocab,create_char_vocab,preprocess

import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix, classification_report

import warnings
warnings.filterwarnings("ignore", category=UserWarning)
warnings.filterwarnings("ignore", category=RuntimeWarning)

print("Utilisation du dispositif : ","gpu" if torch.cuda.is_available() else "cp
```

```
Utilisation du dispositif :  gpu
```

ÉTAPE 2: DATA PREPROCESSING

In [2]:
```
jnlpba_sentences, jnlpbaclasses = load_jnlpba_dataset("./datasets/JNLPBA")
print(jnlpba_sentences[1])
```

```
Chargement du dataset JNLPBA depuis: ./datasets/JNLPBA
- sentences: 22402 phrases
- 11 Classes: ['B-DNA', 'I-DNA', 'B-cell_line', 'I-cell_line', 'B-protein', 'I-pr
otein', 'B-cell_type', 'I-cell_type', 'B-RNA', 'I-RNA', 'O']
[('Activation', 'O'), ('of', 'O'), ('the', 'O'), ('CD28', 'B-protein'), ('surfac
e', 'I-protein'), ('receptor', 'I-protein'), ('provides', 'O'), ('a', 'O'), ('maj
or', 'O'), ('costimulatory', 'O'), ('signal', 'O'), ('for', 'O'), ('T', 'O'), ('c
ell', 'O'), ('activation', 'O'), ('resulting', 'O'), ('in', 'O'), ('enhanced',
'O'), ('production', 'O'), ('of', 'O'), ('interleukin-2', 'B-protein'), ('(',
'O'), ('IL-2', 'B-protein'), (')', 'O'), ('and', 'O'), ('cell', 'O'), ('prolifera
tion', 'O'), ('.', 'O')]
```

In [3]:
```python
ncbi_data = load_ncbi_dataset("./datasets/NCBI-Corpus")
ncbi_sentences = prepare_ncbi_for_ner(ncbi_data)
print(ncbi_sentences[1])
```

```
Chargement du dataset NCBI depuis: ./datasets/NCBI-Corpus
Documents chargés: 793
Exemple d'entités dans le premier document: 2
Total de phrases générées (format JNLPBA): 7524

Vérification du format:
Type du premier élément: <class 'tuple'>
Longueur du tuple: 2
Type tokens: <class 'list'> (longueur: 16)
Type labels: <class 'list'> (longueur: 16)
Exemple tokens[:5]: ['Germline', 'mutations', 'in', 'BRCA1', 'are']
Exemple labels[:5]: ['O', 'O', 'O', 'O', 'O']
(['However', ',', 'the', 'function', 'of', 'the', 'BRCA1', 'protein', 'has', 'rem
ained', 'elusive', '.'], ['I-CompositeMention', 'I-CompositeMention', 'I-Composit
eMention', 'I-CompositeMention', 'I-CompositeMention', 'I-CompositeMention', 'I-C
ompositeMention', 'O', 'O', 'O', 'O', 'O'])
Total de phrases générées (format JNLPBA): 7524

Vérification du format:
Type du premier élément: <class 'tuple'>
Longueur du tuple: 2
Type tokens: <class 'list'> (longueur: 16)
Type labels: <class 'list'> (longueur: 16)
Exemple tokens[:5]: ['Germline', 'mutations', 'in', 'BRCA1', 'are']
Exemple labels[:5]: ['O', 'O', 'O', 'O', 'O']
(['However', ',', 'the', 'function', 'of', 'the', 'BRCA1', 'protein', 'has', 'rem
ained', 'elusive', '.'], ['I-CompositeMention', 'I-CompositeMention', 'I-Composit
eMention', 'I-CompositeMention', 'I-CompositeMention', 'I-CompositeMention', 'I-C
ompositeMention', 'O', 'O', 'O', 'O', 'O'])
```

Création de dataset JNLPBA

In [4]:
```python
def create_jnlpba_dataloaders(jnlpba_sentences, batch_size=32, max_seq_len=100,
    """
    Crée les Dataloaders avec Word2Vec
    """
    # 1. Créer les vocabulaires
    vocab = create_vocab(jnlpba_sentences, min_freq=2)
    char_vocab = create_char_vocab(jnlpba_sentences)

    # 2. Créer le mapping des tags
    tag_to_idx, idx_to_tag = create_tag_mapping(jnlpba_sentences)

    # 3. Diviser en train/dev/test
    total = len(jnlpba_sentences)
    train_size = int(0.7 * total)
    dev_size = int(0.15 * total)

    train_sentences = jnlpba_sentences[:train_size]
    dev_sentences = jnlpba_sentences[train_size:train_size + dev_size]
    test_sentences = jnlpba_sentences[train_size + dev_size:]

    print(f"JNLPBA - Train: {len(train_sentences)}, Dev: {len(dev_sentences)}, T

    # 4. Entraîner ou charger Word2Vec
```

```
    word2vec_model = None

    if word2vec_path:
        print(f"Tentative de chargement du modèle Word2Vec depuis: {word2vec_pat
        word2vec_model = load_word2vec_model(word2vec_path)

    if word2vec_model is None:
        print("Entraînement d'un nouveau modèle Word2Vec...")

        # Extraire les phrases pour Word2Vec
        tokenized_sentences = [
        [token.lower() for token, label in sentence]
        for sentence in jnlpba_sentences
    ]


        print(f"Nombre de phrases pour Word2Vec: {len(tokenized_sentences)}")
        print(f"Exemple: {tokenized_sentences[0][:10]}")

        # Entraîner Word2Vec
        from gensim.models import Word2Vec
        word2vec_model = Word2Vec(
            sentences=tokenized_sentences,
            vector_size=200,
            window=5,
            min_count=2,
            workers=4,
            sg=1,  # Skip-gram
            epochs=10
        )

        print(f"Vocabulaire Word2Vec entraîné: {len(word2vec_model.wv)} mots")

        # Sauvegarder le modèle
        if word2vec_path:
            try:
                save_word2vec_model(word2vec_model, word2vec_path)
            except Exception as e:
                print(f"Attention: impossible de sauvegarder le modèle: {e}")
                print("Le modèle sera utilisé en mémoire seulement.")

    # 5. Créer la matrice d'embeddings
    pretrained_embeddings = create_embedding_matrix_from_word2vec(word2vec_model

    # 6. Créer les datasets
    train_dataset = NERDataset(train_sentences, vocab, char_vocab, tag_to_idx, m
    dev_dataset = NERDataset(dev_sentences, vocab, char_vocab, tag_to_idx, max_s
    test_dataset = NERDataset(test_sentences, vocab, char_vocab, tag_to_idx, max

    # 7. Créer les DataLoaders
    train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True
    dev_loader = DataLoader(dev_dataset, batch_size=batch_size, shuffle=False)
    test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

    return {
        'train_loader': train_loader,
        'dev_loader': dev_loader,
        'test_loader': test_loader,
        'vocab': vocab,
        'char_vocab': char_vocab,
```

```python
        'tag_to_idx': tag_to_idx,
        'idx_to_tag': idx_to_tag,
        'pretrained_embeddings': pretrained_embeddings,
        'train_sentences': train_sentences,
        'dev_sentences': dev_sentences,
        'test_sentences': test_sentences,
        'word2vec_model': word2vec_model
    }

BATCH_SIZE = 32
results_jnlpba = create_jnlpba_dataloaders(jnlpba_sentences,batch_size=BATCH_SIZ
```

```
Format vocab: Liste de paires (token, label)
Vocabulaire créé: 12664 mots
Mots uniques: 22678
Mots avec fréquence >= 2: 12661
Format char vocab: Liste de paires (token, label)
Vocabulaire caractères créé: 85 caractères
Caractères uniques: 83
Format tag mapping: Liste de paires (token, label)
Mapping tags créé: 12 tags uniques
Tags: ['B-DNA', 'B-RNA', 'B-cell_line', 'B-cell_type', 'B-protein', 'I-DNA', 'I-R
NA', 'I-cell_line', 'I-cell_type', 'I-protein', 'O']
JNLPBA - Train: 15681, Dev: 3360, Test: 3361
Tentative de chargement du modèle Word2Vec depuis: ./word2Vecembeddings/jnlpba_wo
rd2vec
Modèle Word2Vec chargé: ./word2Vecembeddings/jnlpba_word2vec
Mots trouvés dans Word2Vec: 12661
Mots non trouvés: 0
Couverture: 100.00%
Dataset créé: 15681 phrases valides
Dataset créé: 3360 phrases valides
Dataset créé: 3361 phrases valides
```

In [5]:

```python
train_loader = results_jnlpba['train_loader']

# Récupérer un batch
for batch in train_loader:
    word_ids, char_seqs, tag_ids, lengths = batch
    break  # on prend juste le premier batch pour l'exemple

# Afficher les shapes
print("word_ids shape:", word_ids.shape)   # [batch_size, max_seq_len]
print("char_seqs shape:", char_seqs.shape) # [batch_size, max_seq_len, max_char_
print("tag_ids shape:", tag_ids.shape)     # [batch_size, max_seq_len]
print("lengths:", lengths)

# Afficher un exemple de tokens/labels pour le premier élément du batch
idx_to_word = {v:k for k,v in results_jnlpba['vocab'].items()}  # optionnel si t
idx_to_tag = results_jnlpba['idx_to_tag']

print("\nPremier exemple du batch:")
print("Word IDs :", word_ids[0])
print("Char IDs :", char_seqs[0])
print("Tag IDs  :", tag_ids[0])
print("Longueur réelle:", lengths[0])

# Pour afficher les mots et labels décodés
decoded_words = [idx_to_word.get(w.item(), '<UNK>') for w in word_ids[0][:length
decoded_tags  = [idx_to_tag[t.item()] for t in tag_ids[0][:lengths[0]]]
```

```
print("\nDécodé (mots, tags) :", list(zip(decoded_words, decoded_tags)))
```

```
word_ids shape: torch.Size([32, 100])
char_seqs shape: torch.Size([32, 100, 20])
tag_ids shape: torch.Size([32, 100])
lengths: tensor([38, 36, 27, 19, 44,  9, 38, 34, 25, 29, 27, 34, 24, 37, 28, 34,
36, 30,
        26, 20, 22, 16, 14, 33, 16, 26, 19, 29, 25, 14, 27, 23])

Premier exemple du batch:
Word IDs : tensor([  20,  388,  347,   32,   20,  738,   19,  656,   19,   24,
96,    8,
         309,   30,  612,   68, 4871,   68,   16, 4872,  272,  301,  312,   35,
        1894,   36,  242, 1060,  113, 1091,   32, 4444, 1894,  783, 4871,  656,
         738,   18,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0])
Char IDs : tensor([[39, 24,  7,  ...,  0,  0,  0],
        [37,  7, 21,  ...,  0,  0,  0],
        [13,  8, 23,  ...,  0,  0,  0],
        ...,
        [ 0,  0,  0,  ...,  0,  0,  0],
        [ 0,  0,  0,  ...,  0,  0,  0],
        [ 0,  0,  0,  ...,  0,  0,  0]])
Tag IDs  : tensor([11, 11, 11, 11, 11, 11, 11, 11, 11, 11,  3,  8,  8,  8,  8, 1
1,  3, 11,
        11,  5, 10, 10, 10, 11,  5, 11, 11, 11, 11, 11, 11,  5, 10, 11, 11, 11,
        11, 11,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
         0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
         0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
         0,  0,  0,  0,  0,  0,  0,  0,  0,  0])
Longueur réelle: tensor(38)

Décodé (mots, tags) : [('the', 'O'), ('mechanisms', 'O'), ('involved', 'O'), ('i
n', 'O'), ('the', 'O'), ('inhibition', 'O'), ('of', 'O'), ('growth', 'O'), ('of',
'O'), ('a', 'O'), ('human', 'B-cell_line'), ('b', 'I-cell_line'), ('lymphoma', 'I
-cell_line'), ('cell', 'I-cell_line'), ('line', 'I-cell_line'), (',', 'O'), ('b10
4', 'B-cell_line'), (',', 'O'), ('by', 'O'), ('anti-mhc', 'B-protein'), ('class',
'I-protein'), ('ii', 'I-protein'), ('antibodies', 'I-protein'), ('(', 'O'), ('a
b', 'B-protein'), (')', 'O'), ('were', 'O'), ('compared', 'O'), ('with', 'O'),
('those', 'O'), ('in', 'O'), ('anti-igm', 'B-protein'), ('ab', 'I-protein'), ('-i
nduced', 'O'), ('b104', 'O'), ('growth', 'O'), ('inhibition', 'O'), ('.', 'O')]
```

In [6]:

```
def create_ncbi_dataloaders(ncbi_sentences, batch_size=32, max_seq_len=200, word
    """
    Crée les DataLoaders pour NCBI avec Word2Vec
    """
    # 1. Créer les vocabulaires
    vocab = create_vocab(ncbi_sentences, min_freq=2)
    char_vocab = create_char_vocab(ncbi_sentences)

    # 2. Créer le mapping des tags
    tag_to_idx, idx_to_tag = create_tag_mapping(ncbi_sentences)

    # 3. Diviser en train/dev/test
    total = len(ncbi_sentences)
```

```
    train_size = int(0.7 * total)
    dev_size = int(0.15 * total)

    train_sentences = ncbi_sentences[:train_size]
    dev_sentences = ncbi_sentences[train_size:train_size + dev_size]
    test_sentences = ncbi_sentences[train_size + dev_size:]

    print(f"NCBI - Train: {len(train_sentences)}, Dev: {len(dev_sentences)}, Tes

    # 4. Entraîner ou charger Word2Vec
    if word2vec_path and os.path.exists(word2vec_path):
        print(f"Chargement du modèle Word2Vec depuis: {word2vec_path}")
        word2vec_model = load_word2vec_model(word2vec_path)
    else:
        print("Entraînement d'un nouveau modèle Word2Vec...")
        # Entraîner sur toutes les données
        word2vec_model = train_word2vec_embeddings(ncbi_sentences)

        # Sauvegarder le modèle
        if word2vec_path:
            save_word2vec_model(word2vec_model, word2vec_path)

    # 5. Créer la matrice d'embeddings
    pretrained_embeddings = create_embedding_matrix_from_word2vec(word2vec_model

    # 6. Créer les datasets
    train_dataset = NERDataset(train_sentences, vocab, char_vocab, tag_to_idx, m
    dev_dataset = NERDataset(dev_sentences, vocab, char_vocab, tag_to_idx, max_s
    test_dataset = NERDataset(test_sentences, vocab, char_vocab, tag_to_idx, max

    # 7. Créer les DataLoaders
    train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True
    dev_loader = DataLoader(dev_dataset, batch_size=batch_size, shuffle=False)
    test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

    return {
        'train_loader': train_loader,
        'dev_loader': dev_loader,
        'test_loader': test_loader,
        'vocab': vocab,
        'char_vocab': char_vocab,
        'tag_to_idx': tag_to_idx,
        'idx_to_tag': idx_to_tag,
        'pretrained_embeddings': pretrained_embeddings,
        'train_sentences': train_sentences,
        'dev_sentences': dev_sentences,
        'test_sentences': test_sentences,
        'word2vec_model': word2vec_model
    }

results_ncbi = ncbi_data_loaders = create_ncbi_dataloaders(
    ncbi_sentences,
    batch_size=BATCH_SIZE,
    word2vec_path="./word2Vecembeddings/ncbi"
)
```

```
Format vocab: Tuple (tokens, labels)
Vocabulaire créé: 5925 mots
Mots uniques: 10845
Mots avec fréquence >= 2: 5922
Format char vocab: Tuple (tokens, labels)
Vocabulaire caractères créé: 86 caractères
Caractères uniques: 84
Format tag mapping: Tuple (tokens, labels)
Mapping tags créé: 10 tags uniques
Tags: ['B-CompositeMention', 'B-DiseaseClass', 'B-Modifier', 'B-SpecificDisease',
'I-CompositeMention', 'I-DiseaseClass', 'I-Modifier', 'I-SpecificDisease', 'O']
NCBI - Train: 5266, Dev: 1128, Test: 1130
Entraînement d'un nouveau modèle Word2Vec...
Nombre de phrases pour Word2Vec: 7524
Première phrase: ['germline', 'mutations', 'in', 'brca1', 'are', 'responsible',
'for', 'most', 'cases', 'of']...
Entraînement du modèle Word2Vec...
Vocabulaire Word2Vec: 5922 mots
Taille des vecteurs: 200
Modèle Word2Vec sauvegardé: ./word2Vecembeddings/ncbi.model
Mots trouvés dans Word2Vec: 5922
Mots non trouvés: 0
Couverture: 100.00%
Dataset créé: 5266 phrases valides
Dataset créé: 1128 phrases valides
Dataset créé: 1130 phrases valides
```

In [7]:

```python
train_loader = results_ncbi['train_loader']

# Récupérer un batch
for batch in train_loader:
    word_ids, char_seqs, tag_ids, lengths = batch
    break  # on prend juste le premier batch pour l'exemple

# Afficher les shapes
print("word_ids shape:", word_ids.shape)   # [batch_size, max_seq_len]
print("char_seqs shape:", char_seqs.shape) # [batch_size, max_seq_len, max_char_
print("tag_ids shape:", tag_ids.shape)     # [batch_size, max_seq_len]
print("lengths:", lengths)

# Afficher un exemple de tokens/labels pour le premier élément du batch
idx_to_word = {v:k for k,v in results_ncbi['vocab'].items()}  # optionnel si tu
idx_to_tag = results_ncbi['idx_to_tag']

print("\nPremier exemple du batch:")
print("Word IDs :", word_ids[0])
print("Char IDs :", char_seqs[0])
print("Tag IDs  :", tag_ids[0])
print("Longueur réelle:", lengths[0])

# Pour afficher les mots et labels décodés
decoded_words = [idx_to_word.get(w.item(), '<UNK>') for w in word_ids[0][:length
decoded_tags  = [idx_to_tag[t.item()] for t in tag_ids[0][:lengths[0]]]

print("\nDécodé (mots, tags) :", list(zip(decoded_words, decoded_tags)))
```

```
word_ids shape: torch.Size([32, 200])
char_seqs shape: torch.Size([32, 200, 20])
tag_ids shape: torch.Size([32, 200])
lengths: tensor([26,  5, 25, 17, 23, 24, 47, 53, 40, 38, 28, 20, 28,  1,  2, 33,
6, 22,
         1, 21, 38, 13, 11, 18, 26, 17,  1,  3, 11, 25, 19, 21])

Premier exemple du batch:
Word IDs : tensor([  32,  257,  325, 2253,   24,  260,  227,    5,  343,    1,
12,   21,
        1806,   70,    5,   32, 5492, 1802,  186,   56,  686,  216,   12, 3302,
         783,   18,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0])
Char IDs : tensor([[17,  0,  0,  ...,  0,  0,  0],
        [ 8,  3, 28,  ...,  0,  0,  0],
        [ 5, 12,  6,  ...,  0,  0,  0],
        ...,
        [ 0,  0,  0,  ...,  0,  0,  0],
        [ 0,  0,  0,  ...,  0,  0,  0],
        [ 0,  0,  0,  ...,  0,  0,  0]])
Tag IDs  : tensor([9, 9, 2, 6, 6, 6, 6, 6, 6, 6, 6, 9, 9, 9, 9, 9, 9, 3, 7, 7, 7,
7, 7, 9,
        9, 9, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0])
Longueur réelle: tensor(26)

Décodé (mots, tags) : [('a', 'O'), ('new', 'O'), ('molecular', 'B-DiseaseClass'),
('lesion', 'I-DiseaseClass'), ('has', 'I-DiseaseClass'), ('been', 'I-DiseaseClas
s'), ('identified', 'I-DiseaseClass'), ('in', 'I-DiseaseClass'), ('exon', 'I-Dise
aseClass'), ('<UNK>', 'I-DiseaseClass'), ('of', 'I-DiseaseClass'), ('the', 'O'),
('pah', 'O'), ('gene', 'O'), ('in', 'O'), ('a', 'O'), ('hungarian', 'O'), ('pku',
'B-Modifier'), ('patient', 'I-Modifier'), ('by', 'I-Modifier'), ('direct', 'I-Mod
ifier'), ('sequencing', 'I-Modifier'), ('of', 'I-Modifier'), ('pcr-amplified',
'O'), ('dna', 'O'), ('.', 'O')]
```

Model architecture

In [8]:
```python
from tqdm import tqdm
def train_epoch(model, dataloader, optimizer, device):
    """
    Entraînement pour une epoch avec CRF
```

```python
    """
    model.train()
    total_loss = 0.0

    for batch in tqdm(dataloader,total=len(dataloader),desc="Training"):
        word_ids, char_seqs, tag_ids, lengths = batch

        word_ids = word_ids.to(device)
        char_seqs = char_seqs.to(device)
        tag_ids = tag_ids.to(device)

        mask = (word_ids != 0)
        optimizer.zero_grad()
        # Passer les tags -> forward retourne la loss directement
        loss = model(word_ids, char_seqs, mask, tag_ids)

        loss.backward()
        optimizer.step()

        total_loss += loss.item()

    return total_loss / len(dataloader)


def evaluate(model, dataloader, device, idx_to_tag):
    """
    Evaluate model (CRF)
    """
    model.eval()
    total_loss = 0
    all_predictions = []
    all_targets = []

    with torch.no_grad():
        for batch in dataloader:
            word_ids, char_seqs, tag_ids, lengths = batch

            word_ids = word_ids.to(device)
            char_seqs = char_seqs.to(device)
            tag_ids = tag_ids.to(device)

            mask = (word_ids != 0)

            # Calculer la loss (tags fournis)
            loss = model(word_ids, char_seqs, mask, tag_ids)
            total_loss += loss.item()

            # Décoder les séquences pour obtenir les prédictions
            predictions = model(word_ids, char_seqs, mask, tags=None)  # renvoie

            # Stocker les prédictions et targets
            for preds_seq, tag_seq, seq_len in zip(predictions, tag_ids, lengths
                preds_seq = preds_seq[:seq_len]          # découper au vrai leng
                tag_seq = tag_seq[:seq_len].cpu().numpy()  # tag_ids tensor -> n

                pred_tags = [idx_to_tag.get(idx, 'O') for idx in preds_seq]
                target_tags = [idx_to_tag.get(idx, 'O') for idx in tag_seq]

                all_predictions.extend(pred_tags)
                all_targets.extend(target_tags)
```

```python
    return total_loss / len(dataloader), all_predictions, all_targets



def calculate_metrics(predictions, targets):
    """
    Calculate precision, recall, F1-score
    """
    from sklearn.metrics import precision_recall_fscore_support

    # Filter out 'O' tags for entity-specific metrics
    entity_preds = []
    entity_targets = []

    for pred, target in zip(predictions, targets):
        if target != 'O':
            entity_preds.append(pred)
            entity_targets.append(target)

    precision, recall, f1, _ = precision_recall_fscore_support(
        entity_targets, entity_preds, average='macro', zero_division=0
    )

    return precision, recall, f1
```

JNLPBA Dataset

In [12]:
```python
def run_pipeline(
    dataset_name: str,
    loaders: dict,
    use_char_cnn: bool = False,
    use_char_lstm: bool = False,
    use_attention: bool = False,
    use_fc_fusion: bool = False,
    embedding_url: str = None,
    lstm_hidden_dim: Optional[int] = None,
    epochs: int = 40,
    patience: int = 4,
    checkpoints_dir="./checkpoints",
):
    train_loader = loaders['train_loader']
    dev_loader   = loaders['dev_loader']
    test_loader  = loaders['test_loader']

    vocab        = loaders['vocab']
    char_vocab   = loaders['char_vocab']
    tag_to_idx   = loaders['tag_to_idx']
    idx_to_tag   = loaders['idx_to_tag']
    comb_name = "WE"
    if use_char_cnn and use_char_lstm:
        comb_name += "_char_bilstm_cnn"
    elif use_char_cnn:
        comb_name += "_char_cnn"
    elif use_char_lstm:
        comb_name += "_char_bilstm"

    if use_attention:
        comb_name += "_attention"
```

```python
    if use_fc_fusion:
        comb_name += "_fc"

    checkpoints_path = f"{checkpoints_dir}/{comb_name}/best_model.pt"
    os.makedirs(os.path.dirname(checkpoints_path), exist_ok=True)
    # Chargement des embeddings
    if embedding_url:
        pretrained_embeddings = load_word2vec_model(embedding_url)
    elif 'pretrained_embeddings' in loaders and loaders['pretrained_embeddings']
        pretrained_embeddings = loaders['pretrained_embeddings']
    else:
        print("Embedding nul")
        return
    if lstm_hidden_dim is None:
        lstm_hidden_dim = 256 if dataset_name == 'JNLPBA' else 128
    device = "cuda" if torch.cuda.is_available() else "cpu"
    model = CombinatorialNER(
        vocab_size=len(vocab),
        char_vocab_size=len(char_vocab),
        tag_to_idx=tag_to_idx,
        use_char_cnn=use_char_cnn,
        use_char_lstm=use_char_lstm,
        use_attention=use_attention,
        use_fc_fusion=use_fc_fusion,
        pretrained_embeddings=pretrained_embeddings,
        word_embed_dim=200,
        lstm_hidden_dim=lstm_hidden_dim,
        dropout=0.5
    ).to(device)

    print(f"Model parameters: {sum(p.numel() for p in model.parameters()):,}")

    optimizer = optim.Adam(model.parameters(), lr=0.001, weight_decay=5e-4)

    # Scheduler: ReduceLROnPlateau (réduit le LR quand le plateau est atteint)
    scheduler = optim.lr_scheduler.ReduceLROnPlateau(
        optimizer,
        mode='max',        # On maximise le F1-score
        factor=0.5,        # Réduction de 50%
        patience=2,        # Attendre 2 epochs sans amélioration
        threshold=0.0001,  # Seuil d'amélioration minimum
        min_lr=1e-6        # Learning rate minimum
    )

    best_f1 = 0.0
    patience_counter = 0


    # Pour suivre l'historique
    history = {
        'train_loss': [],
        'dev_loss': [],
        'dev_f1': [],
        'lr': []
    }

    for epoch in range(epochs):
        # Entraînement
        train_loss = train_epoch(model, train_loader, optimizer, device)
```

```python
        # Évaluation
        dev_loss, dev_preds, dev_targets = evaluate(model, dev_loader, device, i
        precision, recall, f1 = calculate_metrics(dev_preds, dev_targets)

        # Mettre à jour le scheduler avec le F1-score
        scheduler.step(f1)

        # Sauvegarder l'historique
        history['train_loss'].append(train_loss)
        history['dev_loss'].append(dev_loss)
        history['dev_f1'].append(f1)
        history['lr'].append(optimizer.param_groups[0]['lr'])

        print(
            f"Epoch {epoch+1:02d}/{epochs} | "
            f"Train loss: {train_loss:.4f} | "
            f"Dev loss: {dev_loss:.4f} | "
            f"P: {precision:.4f} R: {recall:.4f} F1: {f1:.4f} | "
            f"LR: {optimizer.param_groups[0]['lr']:.6f}"
        )

        if f1 > best_f1:
            best_f1 = f1
            patience_counter = 0
            torch.save({
                'epoch': epoch,
                'model_state_dict': model.state_dict(),
                'optimizer_state_dict': optimizer.state_dict(),
                'scheduler_state_dict': scheduler.state_dict(),
                'best_f1': best_f1,
            }, checkpoints_path)
            print(f"Meilleur modèle sauvegardé dans {checkpoints_path} : (F1: {f
        else:
            patience_counter += 1

        if patience_counter >= patience:
            print(f"\nEarly stopping triggered après {epoch+1} epochs.")
            break


    # Charger le meilleur modèle
    checkpoint = torch.load(checkpoints_path)
    model.load_state_dict(checkpoint['model_state_dict'])

    test_loss, test_preds, test_targets = evaluate(
        model,
        test_loader,
        device,
        idx_to_tag
    )

    precision, recall, f1 = calculate_metrics(test_preds, test_targets)

    print("\n" + "="*50)
    print("RÉSULTATS FINAUX SUR LE TEST SET")
    print("="*50)
    print(f"Loss: {test_loss:.4f}")
    print(f"Precision: {precision:.4f}")
    print(f"Recall: {recall:.4f}")
    print(f"F1-score: {f1:.4f}")
```

```python
    print("="*50)

    # Afficher l'historique des learning rates
    print("\nHistorique des learning rates:")
    for i, lr in enumerate(history['lr']):
        if i == 0 or lr != history['lr'][i-1]:
            print(f"  Epoch {i+1}: LR = {lr:.6f}")

    return model, history
```

Cobinaison WE

In [ ]:

```python
model_jnlpba_we, history_jnlpba_we = run_pipeline(
    dataset_name='JNLPBA',
    loaders=results_jnlpba,
    use_char_cnn= None,
    use_char_lstm= None,
    use_attention= None,
    use_fc_fusion= None,
    embedding_url=None,
    epochs=40,
    patience=4,
    checkpoints_dir = "./checkpoints/JNLPBA",
)
```

```
Model parameters: 2,873,972
Training: 100%|██████████| 491/491 [01:20<00:00,  6.07it/s]
Epoch 01/40 | Train loss: 320.0335 | Dev loss: 172.4923 | P: 0.7993 R: 0.6633 F1:
0.7215 | LR: 0.001000
Meilleur modèle sauvegardé dans ./checkpoints/JNLPBA/WE/best_model.pt : (F1: 0.72
15)
Training:  19%|██        | 91/491 [00:12<00:56,  7.05it/s]
```

In [12]:

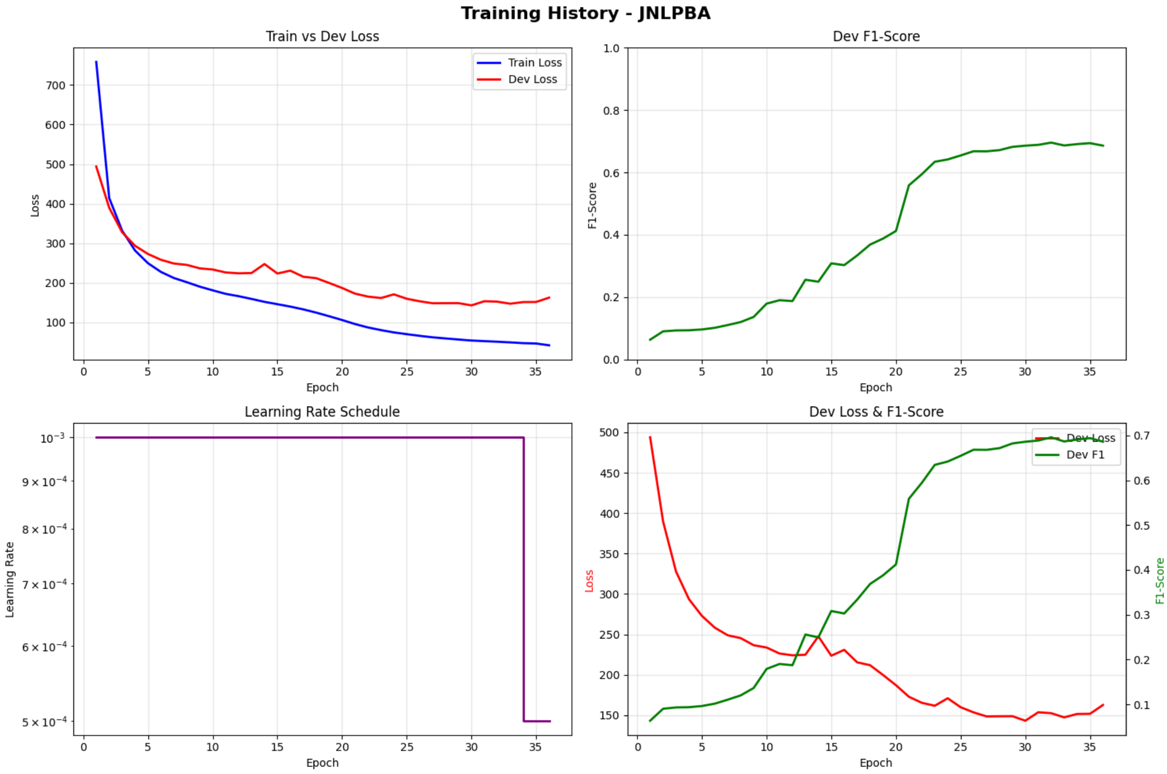
```python
import pickle
import os

def save_vocabularies(data_dict, save_dir):
    """Sauvegarde les vocabulaires et mappings"""
    os.makedirs(save_dir, exist_ok=True)

    with open(os.path.join(save_dir, 'vocab.pkl'), 'wb') as f:
        pickle.dump(data_dict['vocab'], f)

    with open(os.path.join(save_dir, 'char_vocab.pkl'), 'wb') as f:
        pickle.dump(data_dict['char_vocab'], f)

    with open(os.path.join(save_dir, 'tag_to_idx.pkl'), 'wb') as f:
        pickle.dump(data_dict['tag_to_idx'], f)

    with open(os.path.join(save_dir, 'idx_to_tag.pkl'), 'wb') as f:
        pickle.dump(data_dict['idx_to_tag'], f)

    print(f"Vocabulaires sauvegardés dans {save_dir}")

save_vocabularies(results_jnlpba, './vocab/jnlpba')
```

```
Vocabulaires sauvegardés dans ./vocab/jnlpba
```

In [ ]:

```python
def plot_training_history(history, dataset_name):
    """
```

```
    Visualise l'historique d'entraînement
    """
    fig, axes = plt.subplots(2, 2, figsize=(15, 10))
    fig.suptitle(f'Training History - {dataset_name}', fontsize=16, fontweight='

    epochs = range(1, len(history['train_loss']) + 1)

    # Plot 1: Train vs Dev Loss
    axes[0, 0].plot(epochs, history['train_loss'], 'b-', label='Train Loss', lin
    axes[0, 0].plot(epochs, history['dev_loss'], 'r-', label='Dev Loss', linewid
    axes[0, 0].set_xlabel('Epoch')
    axes[0, 0].set_ylabel('Loss')
    axes[0, 0].set_title('Train vs Dev Loss')
    axes[0, 0].legend()
    axes[0, 0].grid(True, alpha=0.3)

    # Plot 2: F1-Score
    axes[0, 1].plot(epochs, history['dev_f1'], 'g-', linewidth=2)
    axes[0, 1].set_xlabel('Epoch')
    axes[0, 1].set_ylabel('F1-Score')
    axes[0, 1].set_title('Dev F1-Score')
    axes[0, 1].set_ylim(0, 1)
    axes[0, 1].grid(True, alpha=0.3)

    # Plot 3: Learning Rate
    axes[1, 0].step(epochs, history['lr'], 'purple', linewidth=2)
    axes[1, 0].set_xlabel('Epoch')
    axes[1, 0].set_ylabel('Learning Rate')
    axes[1, 0].set_title('Learning Rate Schedule')
    axes[1, 0].set_yscale('log')
    axes[1, 0].grid(True, alpha=0.3)

    # Plot 4: Combined (Loss + F1)
    ax2 = axes[1, 1].twinx()
    line1, = axes[1, 1].plot(epochs, history['dev_loss'], 'r-', label='Dev Loss'
    line2, = ax2.plot(epochs, history['dev_f1'], 'g-', label='Dev F1', linewidth
    axes[1, 1].set_xlabel('Epoch')
    axes[1, 1].set_ylabel('Loss', color='r')
    ax2.set_ylabel('F1-Score', color='g')
    axes[1, 1].set_title('Dev Loss & F1-Score')
    lines = [line1, line2]
    labels = [l.get_label() for l in lines]
    axes[1, 1].legend(lines, labels, loc='upper right')
    axes[1, 1].grid(True, alpha=0.3)

    plt.tight_layout()
    plt.show()
    print(f"STATISTIQUES FINALES - {dataset_name}")

    print(f"Meilleur F1-score (dev): {max(history['dev_f1']):.4f}")
    print(f"Final train loss: {history['train_loss'][-1]:.4f}")
    print(f"Final dev loss: {history['dev_loss'][-1]:.4f}")
    print(f"Final learning rate: {history['lr'][-1]:.6f}")

plot_training_history(history_jnlpba, dataset_name="JNLPBA")
```




```
STATISTIQUES FINALES - JNLPBA
Meilleur F1-score (dev): 0.6957
Final train loss: 42.1517
Final dev loss: 162.5667
Final learning rate: 0.000500
```

In [33]:
```python
def load_trained_model(model_path, dataset_name, loaders_dict):
    """
    Charge un modèle entraîné et le prépare pour les prédictions
    """
    device = "cuda" if torch.cuda.is_available() else "cpu"

    # Récupérer les paramètres du loader
    vocab = loaders_dict['vocab']
    char_vocab = loaders_dict['char_vocab']
    tag_to_idx = loaders_dict['tag_to_idx']
    idx_to_tag = loaders_dict['idx_to_tag']

    # Déterminer les paramètres selon le dataset
    if dataset_name == 'JNLPBA':
        lstm_hidden = 256
    else:  # NCBI-Disease
        lstm_hidden = 128

    # Créer le modèle
    model = CombinatorialNER(
        vocab_size=len(vocab),
        char_vocab_size=len(char_vocab),
        tag_to_idx=tag_to_idx,
        pretrained_embeddings=None,  # On ne charge pas les embeddings, le modèl
        word_embed_dim=200,
        lstm_hidden_dim=lstm_hidden,
        dropout=0.5
    ).to(device)

    # Charger les poids sauvegardés
    checkpoint = torch.load(model_path, map_location=device)
```

```
    model.load_state_dict(checkpoint['model_state_dict'])
    model.eval()  # Mode évaluation

    print(f"Modèle chargé depuis: {model_path}")
    print(f"Best F1 enregistré: {checkpoint.get('best_f1', 'N/A'):.4f}")
    print(f"Époque: {checkpoint.get('epoch', 'N/A')}")

    return model, device, idx_to_tag

def plot_confusion_matrix_by_class(predictions, targets, idx_to_tag, dataset_nam
                                   normalize=True, figsize=(12, 10)):
    """
    Affiche la matrice de confusion par classe avec des statistiques détaillées
    """
    # Vérification des types
    print(f"\nVérification des types:")
    print(f"  Type predictions[0]: {type(predictions[0])}, valeur: {predictions[
    print(f"  Type targets[0]: {type(targets[0])}, valeur: {targets[0]}")

    # S'assurer que tout est string
    predictions = [str(p) for p in predictions]
    targets = [str(t) for t in targets]

    # Filtrer les tags 'O' pour se concentrer sur les entités
    entity_predictions = []
    entity_targets = []

    for pred, target in zip(predictions, targets):
        if target != 'O':  # On garde seulement les entités (pas le background)
            entity_predictions.append(pred)
            entity_targets.append(target)

    print(f"\nStatistiques:")
    print(f"  Total tokens: {len(predictions)}")
    print(f"  Entity tokens: {len(entity_targets)}")
    print(f"  Unique entity tags in targets: {set(entity_targets)}")
    print(f"  Unique entity tags in predictions: {set(entity_predictions)}")

    # Obtenir les classes uniques (sauf 'O')
    all_unique_tags = sorted(set(entity_targets + entity_predictions))

    # Filtrer pour garder seulement les tags valides (pas les nombres)
    valid_tags = [tag for tag in all_unique_tags if not tag.isdigit()]

    # Si on a des tags numériques, c'est qu'il y a un problème de conversion
    numeric_tags = [tag for tag in all_unique_tags if tag.isdigit()]
    if numeric_tags:
        print(f"\nATTENTION: Tags numériques trouvés: {numeric_tags}")
        print("Cela indique que les prédictions sont des indices, pas des tags!"
        print(f"Mapping idx_to_tag: {dict(list(idx_to_tag.items())[:10])}")

    print(f"  Tags valides à plotter: {valid_tags}")

    if not valid_tags:
        print("Aucune entité valide trouvée dans les données!")
        return

    cm = confusion_matrix(entity_targets, entity_predictions, labels=valid_tags)
    supports = cm.sum(axis=1)
```

```python
    if normalize:
        # Éviter la division par zéro
        with np.errstate(divide='ignore', invalid='ignore'):
            cm_normalized = cm.astype('float') / supports[:, np.newaxis]
            cm_normalized = np.nan_to_num(cm_normalized, nan=0.0)
        cm_to_plot = cm_normalized
        fmt = '.2f'
        title_suffix = '(Normalisée)'
    else:
        cm_to_plot = cm
        fmt = 'd'
        title_suffix = '(Absolue)'

    # Créer la figure
    plt.figure(figsize=figsize)

    # Si la matrice est trop grande, ajuster la taille
    if len(valid_tags) > 15:
        figsize = (max(12, len(valid_tags)), max(10, len(valid_tags)))
        plt.figure(figsize=figsize)

    sns.heatmap(cm_to_plot, annot=True, fmt=fmt, cmap='Blues',
                xticklabels=valid_tags, yticklabels=valid_tags,
                cbar_kws={'label': 'Proportion' if normalize else 'Count'})

    plt.title(f'Matrice de Confusion - {dataset_name} {title_suffix}',
              fontsize=14, fontweight='bold')
    plt.xlabel('Prédictions', fontsize=12)
    plt.ylabel('Vraies labels', fontsize=12)
    plt.xticks(rotation=45, ha='right')
    plt.yticks(rotation=0)
    plt.tight_layout()
    plt.show()
    filtered_targets = []
    filtered_preds = []

    for t, p in zip(entity_targets, entity_predictions):
        if t in valid_tags and p in valid_tags:
            filtered_targets.append(t)
            filtered_preds.append(p)

    if filtered_targets:
        report = classification_report(filtered_targets, filtered_preds,
                                       labels=valid_tags, zero_division=0)
        print(report)
    else:
        print("Pas assez de données pour générer le rapport")

    for i, tag in enumerate(valid_tags):
        tp = cm[i, i]
        fp = cm[:, i].sum() - tp
        fn = cm[i, :].sum() - tp
        support = supports[i]

        # Éviter les divisions par zéro
        precision = tp / (tp + fp) if (tp + fp) > 0 else 0
        recall = tp / (tp + fn) if (tp + fn) > 0 else 0
        f1 = 2 * precision * recall / (precision + recall) if (precision + recal

        # Support est déjà un entier (sum de la ligne)
```

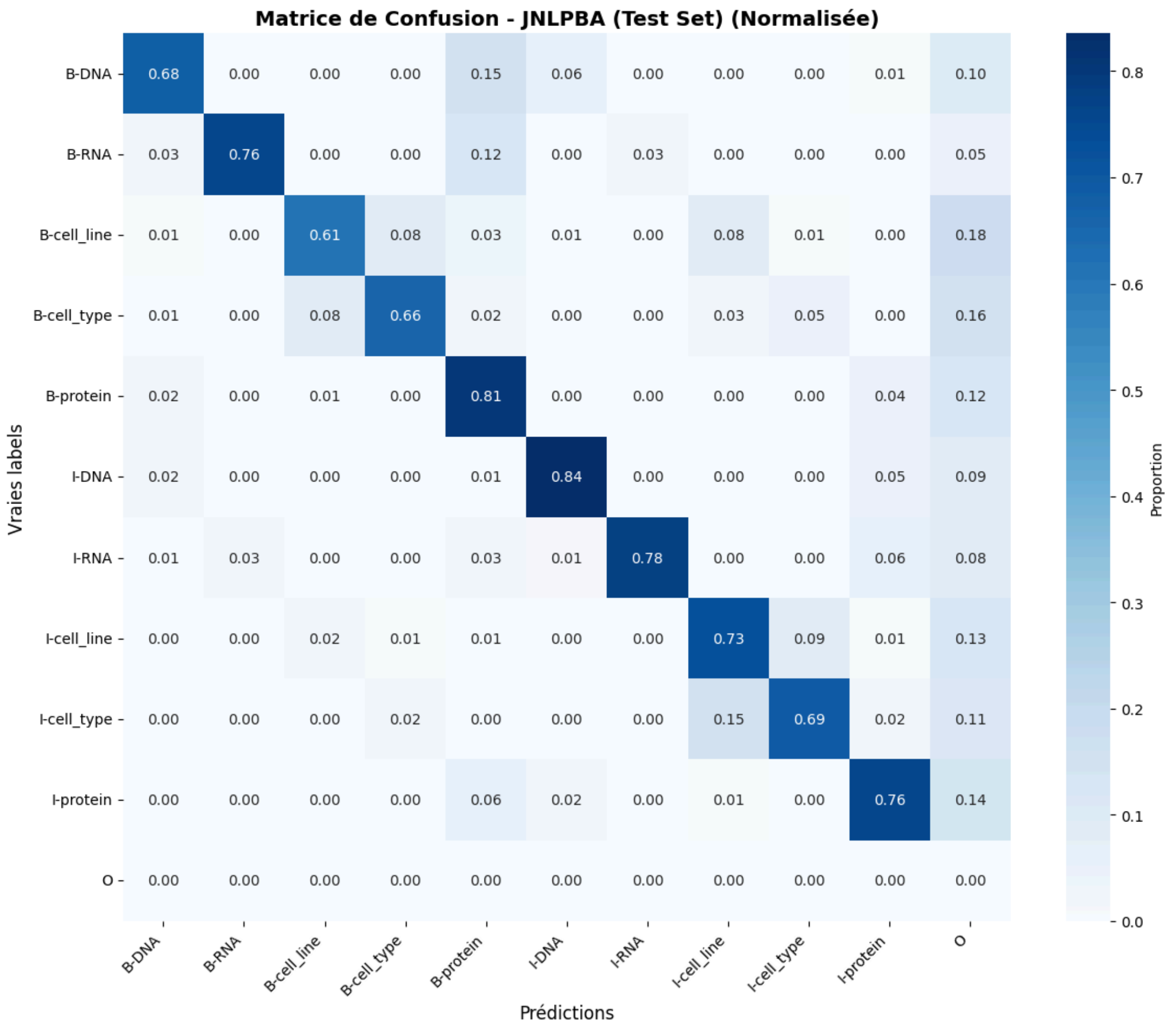
```python
        print(f"{tag:20s}: Precision={precision:.3f}, Recall={recall:.3f}, "
              f"F1={f1:.3f}, Support={int(support)}")

def predict_on_test_set(model, test_loader, device, idx_to_tag):
    """
    Version simplifiée et robuste
    """
    model.eval()
    all_predictions = []
    all_targets = []
    for idx, tag in sorted(idx_to_tag.items()):
        print(f"  {idx} -> {tag}")

    with torch.no_grad():
        for batch_idx, batch in enumerate(test_loader):
            word_ids, char_seqs, tag_ids, lengths = batch

            word_ids = word_ids.to(device)
            char_seqs = char_seqs.to(device)
            tag_ids = tag_ids.to(device)
            mask = (word_ids != 0)
            pred_sequences = model(word_ids, char_seqs, mask, tags=None)

            for i in range(len(word_ids)):
                actual_len = lengths[i]
                if isinstance(pred_sequences, list):
                    pred_seq = pred_sequences[i][:actual_len]
                else:
                    pred_seq = torch.argmax(pred_sequences[i], dim=1)[:actual_le

                # Convertir indices -> tags
                pred_tags = []
                for idx in pred_seq:
                    if isinstance(idx, torch.Tensor):
                        idx = idx.item()
                    tag = idx_to_tag.get(int(idx), 'O')
                    pred_tags.append(tag)

                # Convertir cibles indices -> tags
                target_indices = tag_ids[i][:actual_len].cpu().numpy()
                target_tags = [idx_to_tag.get(int(idx), 'O') for idx in target_i

                all_predictions.extend(pred_tags)
                all_targets.extend(target_tags)

    return all_predictions, all_targets


model_path = "./checkpoints/JNLPBA/best_model_JNLPBA_full.pt"
model, device, idx_to_tag = load_trained_model(
    model_path=model_path,
    dataset_name='JNLPBA',
    loaders_dict=results_jnlpba
)

# 3. Faire des prédictions avec la version corrigée
test_loader = results_jnlpba['test_loader']
predictions, targets = predict_on_test_set(
    model=model,
    test_loader=test_loader,
```

```
    device=device,
    idx_to_tag=idx_to_tag
)
```

```
Modèle chargé depuis: ./checkpoints/JNLPBA/best_model_JNLPBA_full.pt
Best F1 enregistré: 0.6957
Époque: 31
  0 -> <PAD>
  1 -> B-DNA
  2 -> B-RNA
  3 -> B-cell_line
  4 -> B-cell_type
  5 -> B-protein
  6 -> I-DNA
  7 -> I-RNA
  8 -> I-cell_line
  9 -> I-cell_type
  10 -> I-protein
  11 -> O
```

In [32]:
```
from sklearn.metrics import accuracy_score, classification_report

# Pour tous les tokens (y compris 'O')
accuracy = accuracy_score(targets, predictions)
print(f"\nAccuracy globale: {accuracy:.4f}")

entity_preds = [p for p, t in zip(predictions, targets) if t != 'O']
entity_targets = [t for p, t in zip(predictions, targets) if t != 'O']

if entity_targets:
    entity_accuracy = accuracy_score(entity_targets, entity_preds)
    print(f"Accuracy sur les entités: {entity_accuracy:.4f}")
    print(f"Nombre d'entités: {len(entity_targets):,}")
    print("\nRapport de classification (entités seulement):")
    print(classification_report(entity_targets, entity_preds, zero_division=0))
else:
    print("Aucune entité trouvée dans les cibles!")


if set(predictions) == set(targets) or all(isinstance(p, str) for p in predictic
    plot_confusion_matrix_by_class(
        predictions=predictions,
        targets=targets,
        idx_to_tag=idx_to_tag,
        dataset_name='JNLPBA (Test Set)',
        normalize=True,
        figsize=(12, 10)
    )
else:
    print("\nERREUR: Les prédictions et cibles n'ont pas les mêmes types!")
    print(f"Types prédictions: {set(type(p) for p in predictions[:10])}")
    print(f"Types cibles: {set(type(t) for t in targets[:10])}")
```

```
Accuracy globale: 0.9163
Accuracy sur les entités: 0.7491
Nombre d'entités: 16,829

Rapport de classification (entités seulement):
              precision    recall  f1-score   support

       B-DNA       0.82      0.68      0.74       855
       B-RNA       0.84      0.76      0.80        96
 B-cell_line       0.54      0.61      0.57       393
 B-cell_type       0.92      0.66      0.77      1729
   B-protein       0.88      0.81      0.84      4505
       I-DNA       0.88      0.84      0.86      1393
       I-RNA       0.90      0.78      0.84       156
 I-cell_line       0.53      0.73      0.61       792
 I-cell_type       0.92      0.69      0.79      2690
   I-protein       0.91      0.76      0.83      4220
           O       0.00      0.00      0.00         0

    accuracy                           0.75     16829
   macro avg       0.74      0.67      0.70     16829
weighted avg       0.87      0.75      0.80     16829


Vérification des types:
  Type predictions[0]: <class 'str'>, valeur: O
  Type targets[0]: <class 'str'>, valeur: O

Statistiques:
  Total tokens: 87700
  Entity tokens: 16829
  Unique entity tags in targets: {'B-protein', 'I-cell_type', 'B-cell_type', 'B-D
NA', 'I-RNA', 'I-DNA', 'B-RNA', 'I-cell_line', 'B-cell_line', 'I-protein'}
  Unique entity tags in predictions: {'O', 'B-protein', 'I-cell_type', 'B-cell_ty
pe', 'B-DNA', 'I-RNA', 'I-DNA', 'B-RNA', 'I-cell_line', 'B-cell_line', 'I-protei
n'}
  Tags valides à plotter: ['B-DNA', 'B-RNA', 'B-cell_line', 'B-cell_type', 'B-pro
tein', 'I-DNA', 'I-RNA', 'I-cell_line', 'I-cell_type', 'I-protein', 'O']
```



Matrice de Confusion - JNLPBA (Test Set) (Normalisée)
Vraies labels
Prédictions
Proportion
B-DNA
B-RNA
B-cell_line
B-cell_type
B-protein
I-DNA
I-RNA
I-cell_line
I-cell_type
I-protein
O

```
==============================================================
RAPPORT DE CLASSIFICATION DÉTAILLÉ - JNLPBA (Test Set)
==============================================================
              precision    recall  f1-score   support

       B-DNA       0.82      0.68      0.74       855
       B-RNA       0.84      0.76      0.80        96
 B-cell_line       0.54      0.61      0.57       393
 B-cell_type       0.92      0.66      0.77      1729
   B-protein       0.88      0.81      0.84      4505
       I-DNA       0.88      0.84      0.86      1393
       I-RNA       0.90      0.78      0.84       156
 I-cell_line       0.53      0.73      0.61       792
 I-cell_type       0.92      0.69      0.79      2690
   I-protein       0.91      0.76      0.83      4220
           O       0.00      0.00      0.00         0

    accuracy                           0.75     16829
   macro avg       0.74      0.67      0.70     16829
weighted avg       0.87      0.75      0.80     16829


==============================================================
STATISTIQUES PAR CLASSE - JNLPBA (Test Set)
==============================================================
B-DNA               : Precision=0.816, Recall=0.684, F1=0.744, Support=855
B-RNA               : Precision=0.839, Recall=0.760, F1=0.798, Support=96
B-cell_line         : Precision=0.544, Recall=0.608, F1=0.575, Support=393
B-cell_type         : Precision=0.916, Recall=0.660, F1=0.767, Support=1729
B-protein           : Precision=0.884, Recall=0.806, F1=0.843, Support=4505
I-DNA               : Precision=0.878, Recall=0.836, F1=0.857, Support=1393
I-RNA               : Precision=0.904, Recall=0.782, F1=0.838, Support=156
I-cell_line         : Precision=0.526, Recall=0.729, F1=0.611, Support=792
I-cell_type         : Precision=0.918, Recall=0.691, F1=0.788, Support=2690
I-protein           : Precision=0.908, Recall=0.762, F1=0.829, Support=4220
O                   : Precision=0.000, Recall=0.000, F1=0.000, Support=0
```

In [ ]:

```python
model_jnlpba, history_jnlpba = run_complete_pipeline_with_scheduler(
    dataset_name='NCBI',
    loaders=results_ncbi,
    embedding_url=None,
    epochs=40,
    patience=4,
    checkpoints_dir = "./checkpoints/NCBI"
)
```

In [ ]: