

Code pour Protocole d'évaluation - c92e202cc

USER

Je souhaite apprendre en détaille le fonctionnement des transformers et de l'ia générative. Je suis ingénieur en IA depuis 1 ans et je suis en train de faire un thèse en ia générative mais je voudrais revenir a la base des transformeur et notamment de bert pour pouvoir bien comprendre technique les architectures et nouvelle améliorations. Fait moi un cours détailler avec un cas pratique pour redévelopper Bert sans librairie en python

BOT

Thoughts

La personne me demande de lui donner un cours détaillé sur les transformers et l'IA générative, avec un focus particulier sur BERT. La personne est un ingénieur en IA depuis 1 an et fait une thèse en IA générative, mais souhaite revenir aux bases des transformers. Elle souhaite également un cas pratique pour redévelopper BERT sans librairie en Python.

Je vais structurer ma réponse en plusieurs parties:

1. Une introduction aux transformers et leur contexte
2. L'architecture détaillée d'un transformer (encodeur/décodeur)
3. Focus spécifique sur BERT (Bidirectional Encoder Representations from Transformers)
4. Les mécanismes clés (attention, masquage, etc.)
5. Un cas pratique: implémentation simplifiée de BERT en Python pur
6. Analyse des améliorations post-BERT et leur impact sur l'IA générative

Commençons:

Architecture des Transformers et BERT : Cours détaillé

Je comprends votre besoin de revenir aux fondamentaux pour approfondir votre thèse. Voici un cours détaillé sur les transformers avec un focus sur BERT, suivi d'une implémentation pratique.

1. Introduction aux Transformers

Les transformers ont révolutionné le NLP en 2017 avec la publication de "Attention is All You Need" par Vaswani et al. Contrairement aux architectures RNN/LSTM qui traitent les séquences de façon séquentielle, les transformers utilisent des mécanismes d'attention pour traiter les entrées en parallèle, offrant de meilleures performances et la capacité de capturer des dépendances à longue distance.

2. Architecture fondamentale des Transformers

Encodeur-Décodeur

L'architecture originale comporte:

- Un **encodeur** qui traite le texte d'entrée
- Un **décodeur** qui génère la sortie

Chaque bloc contient principalement:

- Des couches d'**attention multi-têtes**
- Des réseaux **feed-forward**
- Des connexions résiduelles et normalisation de couches

Mécanisme d'attention

Le cœur du transformer est l'attention définie par:

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

où Q (requêtes), K (clés), et V (valeurs) sont des transformations linéaires de l'entrée.

L'attention multi-têtes permet d'apprendre différentes relations:

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

3. BERT: Bidirectional Encoder Representations from Transformers

BERT (2018, Devlin et al.) utilise uniquement la partie encodeur du transformer pour créer des représentations contextuelles bidirectionnelles.

Particularités de BERT

- **Bidirectionnalité**: contrairement aux modèles précédents qui lisaient le texte de gauche à droite, BERT considère le contexte dans les deux directions
- **Pré-entraînement en deux étapes**:
 1. Masked Language Modeling (MLM): prédiction de mots masqués
 2. Next Sentence Prediction (NSP): prédiction si deux phrases se suivent

Architecture de BERT

- BERT-base: 12 couches d'encodeurs, 12 têtes d'attention, 768 dimensions cachées (110M paramètres)
- BERT-large: 24 couches, 16 têtes, 1024 dimensions (340M paramètres)

Tokenisation et entrées spéciales

- Tokens spéciaux: [CLS], [SEP], [MASK]
- WordPiece pour la tokenisation

4. Implémentation de BERT en Python

Voici une implémentation simplifiée de BERT sans bibliothèques spécialisées:

```

import numpy as np
import math

class MultiHeadAttention:
    def __init__(self, d_model, num_heads):
        self.d_model = d_model
        self.num_heads = num_heads
        self.d_k = d_model // num_heads

        # Initialisation des matrices de poids
        self.W_q = np.random.randn(d_model, d_model) * 0.01
        self.W_k = np.random.randn(d_model, d_model) * 0.01
        self.W_v = np.random.randn(d_model, d_model) * 0.01
        self.W_o = np.random.randn(d_model, d_model) * 0.01

    def split_heads(self, x):
        # x shape: (batch_size, seq_len, d_model)
        batch_size, seq_len = x.shape[0], x.shape[1]

        # Reshape pour séparer les têtes
        # (batch_size, seq_len, num_heads, d_k)
        x = x.reshape(batch_size, seq_len, self.num_heads, self.d_k)

        # Transpose pour avoir (batch_size, num_heads, seq_len, d_k)
        return x.transpose(0, 2, 1, 3)

    def combine_heads(self, x):
        # x shape: (batch_size, num_heads, seq_len, d_k)
        batch_size, seq_len = x.shape[0], x.shape[2]

        # Transpose pour obtenir (batch_size, seq_len, num_heads, d_k)
        x = x.transpose(0, 2, 1, 3)

        # Combine les têtes
        return x.reshape(batch_size, seq_len, self.d_model)

    def scaled_dot_product_attention(self, Q, K, V, mask=None):
        # Q, K, V shapes: (batch_size, num_heads, seq_len, d_k)
        d_k = Q.shape[-1]

        # Calcul de l'attention:  $QK^T / \sqrt{d_k}$ 
        scores = np.matmul(Q, K.transpose(0, 1, 3, 2)) / math.sqrt(d_k)

        # Application du masque si fourni
        if mask is not None:
            scores = scores + (mask * -1e9)

        # Softmax pour obtenir les poids d'attention
        attention_weights = np.exp(scores) / np.sum(np.exp(scores), axis=-1, keepdims=True)

        # Calcul de la sortie
        output = np.matmul(attention_weights, V)

        return output, attention_weights

    def forward(self, Q, K, V, mask=None):
        batch_size = Q.shape[0]

        # Projections linéaires
        Q = np.matmul(Q, self.W_q) # (batch_size, seq_len, d_model)
        K = np.matmul(K, self.W_k) # (batch_size, seq_len, d_model)
        V = np.matmul(V, self.W_v) # (batch_size, seq_len, d_model)

        # Division en têtes multiples
        Q = self.split_heads(Q) # (batch_size, num_heads, seq_len, d_k)
        K = self.split_heads(K) # (batch_size, num_heads, seq_len, d_k)
        V = self.split_heads(V) # (batch_size, num_heads, seq_len, d_k)

        # Calcul de l'attention
        scaled_attention, attention_weights = self.scaled_dot_product_attention(Q, K, V, mask)

        # Concaténation des têtes
        concat_attention = self.combine_heads(scaled_attention) # (batch_size, seq_len, d_model)

        # Projection finale
        output = np.matmul(concat_attention, self.W_o) # (batch_size, seq_len, d_model)

        return output, attention_weights

class PositionwiseFeedForward:
    def __init__(self, d_model, d_ff):
        self.d_model = d_model
        self.d_ff = d_ff

        # Poids pour les deux couches FF
        self.W1 = np.random.randn(d_model, d_ff) * 0.01
        self.b1 = np.zeros((d_ff,))

        self.W2 = np.random.randn(d_ff, d_model) * 0.01
        self.b2 = np.zeros((d_model,))

    def gelu(self, x):
        # Approximation de GELU (Gaussian Error Linear Unit)

```

```

    return 0.5 * x * (1 + np.tanh(np.sqrt(2 / np.pi) * (x + 0.044715 * x**3)))

def forward(self, x):
    # Première couche avec GELU
    output = np.matmul(x, self.W1) + self.b1
    output = self.gelu(output)

    # Seconde couche linéaire
    output = np.matmul(output, self.W2) + self.b2

    return output

class LayerNormalization:
    def __init__(self, d_model, eps=1e-6):
        self.eps = eps
        self.gamma = np.ones((d_model,)) # facteur d'échelle
        self.beta = np.zeros((d_model,)) # décalage

    def forward(self, x):
        mean = np.mean(x, axis=-1, keepdims=True)
        std = np.std(x, axis=-1, keepdims=True)
        return self.gamma * (x - mean) / (std + self.eps) + self.beta

class EncoderLayer:
    def __init__(self, d_model, num_heads, d_ff, dropout_rate=0.1):
        self.mha = MultiHeadAttention(d_model, num_heads)
        self.ffn = PositionwiseFeedForward(d_model, d_ff)

        self.layer_norm1 = LayerNormalization(d_model)
        self.layer_norm2 = LayerNormalization(d_model)

        self.dropout_rate = dropout_rate

    def forward(self, x, mask=None):
        # Multi-Head Attention avec résidu et normalisation
        attn_output, _ = self.mha.forward(x, x, x, mask)
        # Simuler le dropout (pour simplifier, on le garde minimal)
        attn_output *= (1.0 - self.dropout_rate)
        out1 = self.layer_norm1.forward(x + attn_output)

        # Feed Forward avec résidu et normalisation
        ffn_output = self.ffn.forward(out1)
        # Simuler le dropout
        ffn_output *= (1.0 - self.dropout_rate)
        out2 = self.layer_norm2.forward(out1 + ffn_output)

        return out2

class PositionalEncoding:
    def __init__(self, d_model, max_seq_length=512):
        self.d_model = d_model

        # Création de la matrice d'encodage positionnel
        pe = np.zeros((max_seq_length, d_model))

        # Calcul des positions
        position = np.arange(0, max_seq_length)[:, np.newaxis]

        # Calcul des diviseurs
        div_term = np.exp(np.arange(0, d_model, 2) * -(np.log(10000.0) / d_model))

        # Application des fonctions sinusoïdales
        pe[:, 0::2] = np.sin(position * div_term)
        pe[:, 1::2] = np.cos(position * div_term)

        self.pe = pe[np.newaxis, :, :] # Ajout d'une dimension pour le batch

    def forward(self, x):
        # x shape: (batch_size, seq_len, d_model)
        # Ajoute l'encodage positionnel jusqu'à la longueur de la séquence
        seq_len = x.shape[1]
        return x + self.pe[:, :seq_len, :]

class BERTEmbedding:
    def __init__(self, vocab_size, d_model, max_seq_length=512):
        self.token_embedding = np.random.randn(vocab_size, d_model) * 0.01
        self.position_embedding = PositionalEncoding(d_model, max_seq_length)
        self.segment_embedding = np.random.randn(2, d_model) * 0.01 # Pour distinguer les phrases
        self.layer_norm = LayerNormalization(d_model)
        self.dropout_rate = 0.1

    def forward(self, token_ids, segment_ids):
        # token_ids et segment_ids: (batch_size, seq_len)
        x = np.take(self.token_embedding, token_ids, axis=0)
        x = self.position_embedding.forward(x)

        # Ajouter les embeddings de segment
        s = np.take(self.segment_embedding, segment_ids, axis=0)
        x = x + s

        x = self.layer_norm.forward(x)
        x *= (1.0 - self.dropout_rate) # Simuler dropout

        return x

```

```

class BERT:
    def __init__(self, vocab_size, d_model=768, num_layers=12, num_heads=12, d_ff=3072, max_seq_length=512):
        self.embedding = BERTEmbedding(vocab_size, d_model, max_seq_length)

        # Créer les couches d'encodeur
        self.encoder_layers = [EncoderLayer(d_model, num_heads, d_ff) for _ in range(num_layers)]

        # Couche de sortie pour MLM (Masked Language Modeling)
        self.mlm_dense = np.random.randn(d_model, d_model) * 0.01
        self.mlm_bias = np.zeros((d_model,))
        self.mlm_layer_norm = LayerNormalization(d_model)
        self.mlm_output = np.random.randn(d_model, vocab_size) * 0.01
        self.mlm_output_bias = np.zeros((vocab_size,))

        # Couche de sortie pour NSP (Next Sentence Prediction)
        self.nsp_output = np.random.randn(d_model, 2) * 0.01
        self.nsp_bias = np.zeros((2,))

    def gelu(self, x):
        return 0.5 * x * (1 + np.tanh(np.sqrt(2 / np.pi) * (x + 0.044715 * x**3)))

    def forward(self, token_ids, segment_ids, attention_mask=None):
        # Obtenir les embeddings
        x = self.embedding.forward(token_ids, segment_ids)

        # Passer par toutes les couches d'encodeur
        for encoder_layer in self.encoder_layers:
            x = encoder_layer.forward(x, attention_mask)

        # Sortie pour le MLM
        mlm_hidden = np.matmul(x, self.mlm_dense) + self.mlm_bias
        mlm_hidden = self.gelu(mlm_hidden)
        mlm_hidden = self.mlm_layer_norm.forward(mlm_hidden)
        mlm_logits = np.matmul(mlm_hidden, self.mlm_output) + self.mlm_output_bias

        # Sortie pour le NSP (en utilisant le token [CLS])
        nsp_logits = np.matmul(x[:, 0], self.nsp_output) + self.nsp_bias

        return mlm_logits, nsp_logits

# Exemple d'utilisation
def create_dummy_data(batch_size, seq_length, vocab_size):
    # Créer des données factices pour tester
    token_ids = np.random.randint(0, vocab_size, (batch_size, seq_length))
    segment_ids = np.zeros((batch_size, seq_length), dtype=np.int32)

    # Simuler deux segments (première et deuxième phrase)
    for i in range(batch_size):
        sep_idx = np.random.randint(1, seq_length - 1)
        segment_ids[i, sep_idx:] = 1

    # Créer un masque d'attention (1 pour les positions réelles, 0 pour le padding)
    mask = np.ones((batch_size, 1, 1, seq_length))

    return token_ids, segment_ids, mask

def main():
    # Paramètres de BERT-Base
    vocab_size = 30522 # Taille du vocabulaire WordPiece
    d_model = 768
    num_layers = 12
    num_heads = 12
    d_ff = 3072

    # Créer une instance BERT
    bert = BERT(vocab_size, d_model, num_layers, num_heads, d_ff)

    # Créer des données factices
    batch_size = 2
    seq_length = 32
    token_ids, segment_ids, attention_mask = create_dummy_data(batch_size, seq_length, vocab_size)

    # Passer les données dans le modèle
    mlm_logits, nsp_logits = bert.forward(token_ids, segment_ids, attention_mask)

    print(f"Forme des logits MLM: {mlm_logits.shape}") # (batch_size, seq_length, vocab_size)
    print(f"Forme des logits NSP: {nsp_logits.shape}") # (batch_size, 2)

    # Dans un cas réel, on calculerait la perte et effectuerait une rétropropagation ici

if __name__ == "__main__":
    main()

```

5. Explication du code

Composants clés implémentés:

1. **MultiHeadAttention**: Implémente l'attention multi-têtes avec:
 - Projection des entrées en Q, K, V
 - Division en têtes multiples
 - Calcul de l'attention pondérée
 - Recombinaison des têtes

2. **PositionwiseFeedForward**: Réseaux feed-forward avec GELU comme fonction d'activation (utilisée dans BERT plutôt que ReLU)
3. **LayerNormalization**: Normalisation des couches pour stabiliser l'apprentissage
4. **EncoderLayer**: Combine attention multi-têtes et feed-forward avec connexions résiduelles
5. **PositionalEncoding**: Encodage de la position pour intégrer l'ordre des mots
6. **BERTEmbedding**: Combine embeddings de tokens, position et segments
7. **BERT**: Modèle complet avec:
 - Couches d'encodeur empilées
 - Tête MLM pour prédire les tokens masqués
 - Tête NSP pour la prédiction de phrase suivante

6. Entraînement pratique

Dans une implémentation complète, vous ajouteriez:

```
# Masquage aléatoire pour MLM
def create_masked_input(token_ids, mask_prob=0.15):
    masked_tokens = token_ids.copy()
    mask_positions = []

    for i in range(token_ids.shape[0]):
        for j in range(token_ids.shape[1]):
            # Ignorer les tokens spéciaux
            if token_ids[i, j] in [0, 101, 102]: # [PAD], [CLS], [SEP]
                continue

            if np.random.random() < mask_prob:
                mask_positions.append((i, j))
                prob = np.random.random()

                if prob < 0.8: # 80% des cas: remplacer par [MASK]
                    masked_tokens[i, j] = 103 # ID pour [MASK]
                elif prob < 0.9: # 10% des cas: remplacer par un token aléatoire
                    masked_tokens[i, j] = np.random.randint(1, vocab_size)
                # 10% restants: garder le token original

    return masked_tokens, mask_positions

# Fonction de perte pour MLM
def compute_mlm_loss(mlm_logits, token_ids, mask_positions):
    loss = 0
    for i, j in mask_positions:
        # Calculer la perte en entropie croisée entre la prédiction et le token réel
        probs = np.exp(mlm_logits[i, j]) / np.sum(np.exp(mlm_logits[i, j]))
        target_prob = probs[token_ids[i, j]]
        loss += np.log(target_prob + 1e-10) # Ajouter une petite valeur pour éviter log(0)

    return loss / len(mask_positions) if mask_positions else 0

# Fonction d'optimisation (version simplifiée avec descente de gradient)
def optimize(model, learning_rate, mlm_loss_grad, nsp_loss_grad):
    # Mise à jour des poids basée sur les gradients
    # En pratique, vous utiliseriez un optimiseur comme Adam
    for param, grad in zip(model.parameters(), mlm_loss_grad + nsp_loss_grad):
        param -= learning_rate * grad
```

7. Évolutions post-BERT

Depuis BERT, plusieurs améliorations ont été développées:

1. **RoBERTa**: Optimisation de l'entraînement de BERT (plus de données, batches plus grands, suppression de NSP)
2. **ALBERT**: Architecture allégée avec partage de paramètres entre couches
3. **DistilBERT**: Version distillée et compressée de BERT
4. **ELECTRA**: Entraînement par détection de tokens remplacés plutôt que masqués
5. **T5**: Transformer text-to-text unifiant différentes tâches en un seul format

Pour l'IA générative, les architectures décodeur comme GPT ont pris le dessus, s'appuyant sur l'attention causale et l'auto-régression pour la génération de texte.

8. Pour aller plus loin

Pour approfondir votre compréhension:

- Implémentez la rétropropagation complète
- Ajoutez l'optimisation avec Adam
- Intégrez la tokenisation WordPiece
- Entraînez sur un petit corpus pour valider l'implémentation

Cette implémentation est simplifiée mais contient les éléments essentiels pour comprendre BERT. Pour une utilisation réelle, vous devriez gérer la compilation GPU, l'optimisation de mémoire et utiliser des techniques d'entraînement distribuées.

Avez-vous des questions sur des aspects spécifiques de l'architecture ou de l'implémentation?

Executed 1st Code Block

Execution Result:

...

Forme des logits MLM: (2, 32, 30522)

Forme des logits NSP: (2, 2)

...

USER

je veux que tu m'explique le code etape par etape

BOT

Thoughts

D'accord, je vais expliquer le code étape par étape, en détaillant chaque composant du modèle BERT que j'ai implémenté dans la réponse précédente. Je vais structurer mon explication de manière progressive, en commençant par les composants fondamentaux jusqu'à l'assemblage complet du modèle.

Plan d'explication :

1. MultiHeadAttention - Le mécanisme d'attention qui est au cœur des transformers
2. PositionwiseFeedForward - Les réseaux feed-forward utilisés après l'attention
3. LayerNormalization - La normalisation de couche pour stabiliser l'apprentissage
4. EncoderLayer - L'assemblage de ces composants dans une couche d'encodeur
5. PositionalEncoding - Le mécanisme d'encodage de position
6. BERTEmbedding - La gestion des embeddings dans BERT
7. BERT - Le modèle complet
8. Les fonctions auxiliaires - Création de données et fonction principale

Pour chaque classe/composant, je vais :

- Expliquer son rôle
- Détailler ses méthodes principales
- Expliquer le code ligne par ligne
- Clarifier comment il s'intègre dans l'architecture globale

Explication détaillée du code BERT étape par étape

Je vais vous expliquer chaque partie du code précédent, en détaillant le fonctionnement de chaque classe et méthode.

1. Classe MultiHeadAttention

```
class MultiHeadAttention:
    def __init__(self, d_model, num_heads):
        self.d_model = d_model
        self.num_heads = num_heads
        self.d_k = d_model // num_heads

        # Initialisation des matrices de poids
        self.W_q = np.random.randn(d_model, d_model) * 0.01
        self.W_k = np.random.randn(d_model, d_model) * 0.01
        self.W_v = np.random.randn(d_model, d_model) * 0.01
        self.W_o = np.random.randn(d_model, d_model) * 0.01
```

****Explication**:**

- `d_model` est la dimension du modèle (768 pour BERT-base)
- `num_heads` est le nombre de têtes d'attention (12 pour BERT-base)
- `d_k` est la dimension par tête (d_model divisé par num_heads)
- Les matrices W_q, W_k, W_v sont utilisées pour projeter les entrées en requêtes (Q), clés (K) et valeurs (V)
- W_o est utilisée pour la projection finale après concaténation des têtes

```
def split_heads(self, x):
    # x shape: (batch_size, seq_len, d_model)
    batch_size, seq_len = x.shape[0], x.shape[1]

    # Reshape pour séparer les têtes
    # (batch_size, seq_len, num_heads, d_k)
    x = x.reshape(batch_size, seq_len, self.num_heads, self.d_k)

    # Transpose pour avoir (batch_size, num_heads, seq_len, d_k)
    return x.transpose(0, 2, 1, 3)
```

****Explication**:**

- Cette méthode divise la représentation en plusieurs têtes
- Elle prend un tenseur de forme (batch_size, seq_len, d_model)
- Réorganise le tenseur pour que chaque tête ait sa propre dimension
- Transpose pour faciliter les calculs d'attention par tête

```
def combine_heads(self, x):
    # x shape: (batch_size, num_heads, seq_len, d_k)
    batch_size, seq_len = x.shape[0], x.shape[2]

    # Transpose pour obtenir (batch_size, seq_len, num_heads, d_k)
    x = x.transpose(0, 2, 1, 3)

    # Combine les têtes
    return x.reshape(batch_size, seq_len, self.d_model)
```

****Explication**:**

- Fonction inverse de `split_heads`
- Recombine les têtes d'attention dans un seul tenseur
- Retourne un tenseur de forme (batch_size, seq_len, d_model)

```
def scaled_dot_product_attention(self, Q, K, V, mask=None):
    # Q, K, V shapes: (batch_size, num_heads, seq_len, d_k)
    d_k = Q.shape[-1]

    # Calcul de l'attention: QK^T / sqrt(d_k)
    scores = np.matmul(Q, K.transpose(0, 1, 3, 2)) / math.sqrt(d_k)

    # Application du masque si fourni
    if mask is not None:
        scores = scores + (mask * -1e9)

    # Softmax pour obtenir les poids d'attention
    attention_weights = np.exp(scores) / np.sum(np.exp(scores), axis=-1, keepdims=True)

    # Calcul de la sortie
    output = np.matmul(attention_weights, V)

    return output, attention_weights
```

****Explication**:**

- Implémente le mécanisme d'attention selon la formule: $\text{Attention}(Q, K, V) = \text{softmax}(QK^T / \sqrt{d_k})V$
- Calcule d'abord les scores d'attention entre Q et K
- Divise par $\sqrt{d_k}$ pour stabiliser les gradients
- Applique un masque (optionnel) pour ignorer certaines positions (comme le padding)
- Applique softmax pour obtenir les poids d'attention normalisés
- Multiplie les poids par V pour obtenir la sortie pondérée

```
def forward(self, Q, K, V, mask=None):
    batch_size = Q.shape[0]

    # Projections linéaires
    Q = np.matmul(Q, self.W_q) # (batch_size, seq_len, d_model)
    K = np.matmul(K, self.W_k) # (batch_size, seq_len, d_model)
    V = np.matmul(V, self.W_v) # (batch_size, seq_len, d_model)

    # Division en têtes multiples
    Q = self.split_heads(Q) # (batch_size, num_heads, seq_len, d_k)
    K = self.split_heads(K) # (batch_size, num_heads, seq_len, d_k)
    V = self.split_heads(V) # (batch_size, num_heads, seq_len, d_k)

    # Calcul de l'attention
    scaled_attention, attention_weights = self.scaled_dot_product_attention(Q, K, V, mask)

    # Concaténation des têtes
    concat_attention = self.combine_heads(scaled_attention) # (batch_size, seq_len, d_model)

    # Projection finale
    output = np.matmul(concat_attention, self.W_o) # (batch_size, seq_len, d_model)

    return output, attention_weights
```

****Explication**:**

- Méthode principale qui orchestre le flux complet de l'attention multi-têtes
- Projette d'abord Q, K, V avec leurs matrices respectives
- Divise ensuite en têtes multiples
- Applique l'attention pour chaque tête
- Recombine les sorties des têtes
- Applique une projection finale pour produire la sortie

2. Classe PositionwiseFeedForward

```
class PositionwiseFeedForward:
    def __init__(self, d_model, d_ff):
        self.d_model = d_model
        self.d_ff = d_ff

    # Poids pour les deux couches FF
    self.W1 = np.random.randn(d_model, d_ff) * 0.01
    self.b1 = np.zeros((d_ff,))

    self.W2 = np.random.randn(d_ff, d_model) * 0.01
    self.b2 = np.zeros((d_model,))

    def gelu(self, x):
        # Approximation de GELU (Gaussian Error Linear Unit)
        return 0.5 * x * (1 + np.tanh(np.sqrt(2 / np.pi) * (x + 0.044715 * x**3)))

    def forward(self, x):
        # Première couche avec GELU
        output = np.matmul(x, self.W1) + self.b1
        output = self.gelu(output)
```

```
# Seconde couche linéaire
output = np.matmul(output, self.W2) + self.b2

return output
```

****Explication**:**

- Implémente le réseau feed-forward utilisé après l'attention
- Comprend deux transformations linéaires avec une activation GELU entre elles
- `d_model` est la dimension d'entrée et de sortie (768 pour BERT-base)
- `d_ff` est la dimension intermédiaire (3072 pour BERT-base)
- GELU est l'activation préférée dans BERT (plutôt que ReLU)

3. Classe LayerNormalization

```
class LayerNormalization:
    def __init__(self, d_model, eps=1e-6):
        self.eps = eps
        self.gamma = np.ones((d_model,)) # facteur d'échelle
        self.beta = np.zeros((d_model,)) # décalage

    def forward(self, x):
        mean = np.mean(x, axis=-1, keepdims=True)
        std = np.std(x, axis=-1, keepdims=True)
        return self.gamma * (x - mean) / (std + self.eps) + self.beta
```

****Explication**:**

- Implémente la normalisation de couche (Layer Normalization)
- Normalise chaque élément individuellement sur la dimension des caractéristiques
- Utilise des paramètres apprenables gamma et beta pour l'échelle et le décalage
- Ajoute epsilon pour éviter la division par zéro
- Stabilise l'entraînement et accélère la convergence

4. Classe EncoderLayer

```
class EncoderLayer:
    def __init__(self, d_model, num_heads, d_ff, dropout_rate=0.1):
        self.mha = MultiHeadAttention(d_model, num_heads)
        self.ffn = PositionwiseFeedForward(d_model, d_ff)

        self.layernorm1 = LayerNormalization(d_model)
        self.layernorm2 = LayerNormalization(d_model)

        self.dropout_rate = dropout_rate

    def forward(self, x, mask=None):
        # Multi-Head Attention avec résidu et normalisation
        attn_output, _ = self.mha.forward(x, x, x, mask)
        # Simuler le dropout (pour simplifier, on le garde minimal)
        attn_output *= (1.0 - self.dropout_rate)
        out1 = self.layernorm1.forward(x + attn_output)

        # Feed Forward avec résidu et normalisation
        ffn_output = self.ffn.forward(out1)
        # Simuler le dropout
        ffn_output *= (1.0 - self.dropout_rate)
        out2 = self.layernorm2.forward(out1 + ffn_output)

        return out2
```

****Explication**:**

- Combine tous les composants pour former une couche d'encodeur complète
- Inclut:
 1. Attention multi-têtes (où Q=K=V car c'est de l'auto-attention)
 2. Connexion résiduelle (x + attn_output)
 3. Normalisation de couche
 4. Réseau feed-forward
 5. Autre connexion résiduelle
 6. Seconde normalisation de couche
- Le dropout est simplifié (multiplication par 1-dropout_rate)
- L'ordre "Attention -> Add & Norm -> FFN -> Add & Norm" suit l'architecture originale du Transformer

5. Classe PositionalEncoding

```
class PositionalEncoding:
    def __init__(self, d_model, max_seq_length=512):
        self.d_model = d_model

        # Création de la matrice d'encodage positionnel
        pe = np.zeros((max_seq_length, d_model))

        # Calcul des positions
        position = np.arange(0, max_seq_length)[:, np.newaxis]

        # Calcul des diviseurs
        div_term = np.exp(np.arange(0, d_model, 2) * -(np.log(10000.0) / d_model))
```



```
# Application des fonctions sinusoïdales
pe[:, 0::2] = np.sin(position * div_term)
pe[:, 1::2] = np.cos(position * div_term)

self.pe = pe[np.newaxis, :, :] # Ajout d'une dimension pour le batch

def forward(self, x):
    # x shape: (batch_size, seq_len, d_model)
    # Ajoute l'encodage positionnel jusqu'à la longueur de la séquence
    seq_len = x.shape[1]
    return x + self.pe[:, :seq_len, :]
```

****Explication**:**

- Crée les encodages positionnels pour donner au modèle la notion d'ordre des mots
- Utilise des fonctions sinusoïdales:
 - sin pour les dimensions paires
 - cos pour les dimensions impaires
- Chaque position a un pattern unique sur les dimensions du modèle
- Ces encodages sont ajoutés aux embeddings de tokens
- La formule utilisée garantit que les positions relatives sont encodées de manière cohérente quelle que soit la longueur de la séquence

6. Classe BERTEmbedding

```
class BERTEmbedding:
    def __init__(self, vocab_size, d_model, max_seq_length=512):
        self.token_embedding = np.random.randn(vocab_size, d_model) * 0.01
        self.position_embedding = PositionalEncoding(d_model, max_seq_length)
        self.segment_embedding = np.random.randn(2, d_model) * 0.01 # Pour distinguer les phrases
        self.layer_norm = LayerNormalization(d_model)
        self.dropout_rate = 0.1

    def forward(self, token_ids, segment_ids):
        # token_ids et segment_ids: (batch_size, seq_len)
        x = np.take(self.token_embedding, token_ids, axis=0)
        x = self.position_embedding.forward(x)

        # Ajouter les embeddings de segment
        s = np.take(self.segment_embedding, segment_ids, axis=0)
        x = x + s

        x = self.layer_norm.forward(x)
        x *= (1.0 - self.dropout_rate) # Simuler dropout

    return x
```

****Explication**:**

- Gère les trois types d'embeddings utilisés dans BERT:
 1. Token embeddings: représentation des mots du vocabulaire
 2. Position embeddings: information sur la position des mots dans la séquence
 3. Segment embeddings: distingue les phrases A et B pour la tâche NSP
- Les trois embeddings sont additionnés
- Une normalisation de couche est appliquée
- Un dropout est simulé pour la régularisation

7. Classe BERT

```
class BERT:
    def __init__(self, vocab_size, d_model=768, num_layers=12, num_heads=12, d_ff=3072, max_seq_length=512):
        self.embedding = BERTEmbedding(vocab_size, d_model, max_seq_length)

        # Créer les couches d'encodeur
        self.encoder_layers = [EncoderLayer(d_model, num_heads, d_ff) for _ in range(num_layers)]

        # Couche de sortie pour MLM (Masked Language Modeling)
        self.mlm_dense = np.random.randn(d_model, d_model) * 0.01
        self.mlm_bias = np.zeros((d_model,))
        self.mlm_layer_norm = LayerNormalization(d_model)
        self.mlm_output = np.random.randn(d_model, vocab_size) * 0.01
        self.mlm_output_bias = np.zeros((vocab_size,))

        # Couche de sortie pour NSP (Next Sentence Prediction)
        self.nsp_output = np.random.randn(d_model, 2) * 0.01
        self.nsp_bias = np.zeros((2,))

    def gelu(self, x):
        return 0.5 * x * (1 + np.tanh(np.sqrt(2 / np.pi) * (x + 0.044715 * x**3)))

    def forward(self, token_ids, segment_ids, attention_mask=None):
        # Obtenir les embeddings
        x = self.embedding.forward(token_ids, segment_ids)

        # Passer par toutes les couches d'encodeur
        for encoder_layer in self.encoder_layers:
            x = encoder_layer.forward(x, attention_mask)

        # Sortie pour le MLM
        mlm_hidden = np.matmul(x, self.mlm_dense) + self.mlm_bias
        mlm_hidden = self.gelu(mlm_hidden)
        mlm_hidden = self.mlm_layer_norm.forward(mlm_hidden)
```

```
mlm_logits = np.matmul(mlm_hidden, self.mlm_output) + self.mlm_output_bias
```

```
# Sortie pour le NSP (en utilisant le token [CLS])
nsp_logits = np.matmul(x[:, 0], self.nsp_output) + self.nsp_bias
```

```
return mlm_logits, nsp_logits
```

****Explication**:**

- C'est la classe principale qui assemble tous les composants de BERT
- Définit l'architecture complète avec:
 - La couche d'embedding
 - 12 couches d'encodeur empilées (pour BERT-base)
 - Les têtes de sortie pour MLM et NSP
- Le flux de données:
 1. Les tokens et segments sont transformés en embeddings
 2. Ces embeddings passent par toutes les couches d'encodeur
 3. Pour MLM: projection, GELU, normalisation de couche, projection finale
 4. Pour NSP: utilise la représentation du token [CLS] pour classifier si deux phrases sont consécutives

8. Fonctions auxiliaires

```
def create_dummy_data(batch_size, seq_length, vocab_size):
    # Créer des données factices pour tester
    token_ids = np.random.randint(0, vocab_size, (batch_size, seq_length))
    segment_ids = np.zeros((batch_size, seq_length), dtype=np.int32)

    # Simuler deux segments (première et deuxième phrase)
    for i in range(batch_size):
        sep_idx = np.random.randint(1, seq_length - 1)
        segment_ids[i, sep_idx:] = 1

    # Créer un masque d'attention (1 pour les positions réelles, 0 pour le padding)
    mask = np.ones((batch_size, 1, 1, seq_length))

    return token_ids, segment_ids, mask
```

****Explication**:**

- Crée des données factices pour tester le modèle
- Génère:
 - token_ids: IDs des tokens d'entrée
 - segment_ids: identifie première (0) ou seconde (1) phrase
 - mask: masque d'attention pour ignorer le padding

```
def main():
    # Paramètres de BERT-Base
    vocab_size = 30522 # Taille du vocabulaire WordPiece
    d_model = 768
    num_layers = 12
    num_heads = 12
    d_ff = 3072

    # Créer une instance BERT
    bert = BERT(vocab_size, d_model, num_layers, num_heads, d_ff)

    # Créer des données factices
    batch_size = 2
    seq_length = 32
    token_ids, segment_ids, attention_mask = create_dummy_data(batch_size, seq_length, vocab_size)

    # Passer les données dans le modèle
    mlm_logits, nsp_logits = bert.forward(token_ids, segment_ids, attention_mask)

    print(f"Forme des logits MLM: {mlm_logits.shape}") # (batch_size, seq_length, vocab_size)
    print(f"Forme des logits NSP: {nsp_logits.shape}") # (batch_size, 2)
```

****Explication**:**

- Fonction principale qui:
 1. Définit les hyperparamètres du modèle
 2. Crée une instance de BERT
 3. Génère des données factices
 4. Exécute une passe avant (forward pass)
 5. Affiche les dimensions des sorties pour vérifier

Flux de données complet

Pour résumer le flux de données dans ce code:

1. Les IDs de tokens et segments sont transformés en embeddings
2. Ces embeddings sont augmentés avec l'information positionnelle
3. Les données passent par 12 couches d'encodeur identiques
4. Chaque couche d'encodeur:
 - Applique l'attention multi-têtes
 - Ajoute une connexion résiduelle et normalise
 - Applique le réseau feed-forward
 - Ajoute une autre connexion résiduelle et normalise
5. La sortie finale est utilisée pour:
 - MLM: prédire les tokens masqués

- NSP: classifiez si deux phrases sont consécutives

Cette implémentation capture l'essence de l'architecture BERT tout en simplifiant certains aspects comme le dropout et l'optimisation.

USER

trouve moi un dataset pour entrainer bert et t'il possible d'optimiser l'entrainement sur mac M1

BOT

Searching the web

- trouve moi un dataset pour entrainer bert et t'il possible d'optimiser l'entrainement sur mac M1

Analyzing Results

- [BERT – Enfin un tutoriel simple et rapide - Inside Machine Learning](https://inside-machinelearning.com/bert-enfin-un-tutoriel-simple-et-rapide/) (Web Search Result)
On a bien un modèle à trois entrées qui se rejoignent pour former réseau de couches uniques.. Le modèle peut sembler simple de prime abord mais il faut bien comprendre que dans la couche KerasLayer (la couche de BERT) il y a en fait plusieurs autres couches.. Elle ne sont pas visible ici car elles sont chargées depuis TensorFlow Hub mais le nom du modèle parle de lui même : « bert_en ...
- [A Step-by-Step Guide to Preparing Datasets for BERT ... - Medium](https://ai.plainenglish.io/bert-pytorch-implementation-prepare-dataset-part-1-efd259113e5a) (Web Search Result)
2. Create conversation pairs for NSP. In order to train BERT, we need to generate pairs of conversation. Each pair consists of a line and its follow-up response, with both trimmed to a maximum length defined by SEQ_LEN to ensure consistency for the NLP model. The final output is a list of these dialogue pairs, structured to provide a contextual sequence for the model to learn from
- [Trouver et choisir un dataset de qualité pour l'IA](https://www.innovatiana.com/post/where-to-find-dataset-for-ai) (Web Search Result)
Un dataset équilibré et varié, exempt de redondances ou de sur-représentation d'un groupe spécifique, garantit une meilleure généralisation du modèle. Niveau de bruit dans les données Les données bruitées (informations erronées ou valeurs extrêmes sans explication) peuvent perturber l'apprentissage et affecter la performance du ...
- [BERT : Le "Transformer model" qui s'entraîne et qui représente](https://lesdieuxducode.com/blog/2019/4/bert--le-transformer-model-qui-sentraine-et-qui-represente) (Web Search Result)
BERT c'est pour Bidirectional Encoder Representations from Transformers. Il est sorti des labos Google AI fin 2018, et s'il est ce jour l'objet de notre attention c'est que son modèle est à la fois : Plus performant que ses prédécesseurs en terme de résultats. Plus performant que ses prédécesseurs en terme de rapidité d'apprentissage.
- [GitHub - tkeldenich/BERT_Easy_Implementation](https://github.com/tkeldenich/BERT_Easy_Implementation) (Web Search Result)
Read the post in English. Aujourd'hui, nous allons voir avec ce tutoriel comment implémenter simplement et rapidement le Transformer BERT.. BERT est un modèle de Deep Learning lancé fin 2019 par Google.C'est un Transformer, un type bien spécifique de réseaux de neurones.D'ailleurs BERT signifie "Bidirectional Encoder Representations from Transformers" littéralement "Représentations d ...
- [TOP 5 des sites où trouver des datasets pour le Machine Learning](https://datascientest.com/public-datasets-machine-learning) (Web Search Result)
Cet outil développé par Google est une des manières les plus efficaces pour trouver un dataset en faisant une simple recherche par mots-clés. Exemple: Je veux faire un projet de Machine Learning en rapport avec le tennis et même plus précisément : en rapport avec Roland-Garros.Pourquoi ne pas ajouter les performances de Nadal dans mon projet.
- [GitHub - kountak/bert-classification: Ce répertoire est un exercice d ...](https://github.com/kountak/bert-classification) (Web Search Result)
Modules spécifiques: nn, Dataset, DataLoader, BertModel ... Passe les inputs dans BERT et le classifieur pour obtenir les prédictions. Chargement du modèle: Crée une instance ... Le script démarre une API FastAPI qui permet de poster des textes pour obtenir des prédictions de classe via un modèle BERT et l'endpoint se trouve sur http ...
- [Choisir les bons paramètres pour le pré-entraînement BERT avec TPU](https://ichi.pro/fr/choisir-les-bons-parametres-pour-le-pre-entrainement-bert-avec-tpu-76436016253520) (Web Search Result)
La pré-formation d'un modèle BERT n'est pas facile et de nombreux articles donnent un excellent aperçu de haut niveau sur ce qu'est BERT et les choses étonnantes qu'il peut faire, ou approfondissent un très petit détail d'implémentation. Cela laisse les Data Scientists en herbe, comme moi il y a quelque temps, souvent à la recherche de Notebooks, en se disant: «Cela a fière allure et ...
- [15 meilleures sources de données pour la formation de modèles d'IA](https://squeezegrowth.com/fr/best-data-sources-for-ai-model-training/) (Web Search Result)
Appen est un améliorateur et un innovateur dans le domaine de la formation de modèles d'IA. De la classification d'images à la détection d'objets, les ensembles de données visuelles d'Appen sont un régal pour les yeux de votre IA. Vous voulez apprendre à une voiture autonome à reconnaître un panneau d'arrêt dans le brouillard ?
- [French NLP: entamez le CamemBERT avec les librairies fast-bert et ...](https://medium.com/@vitalshchutski/french-nlp-entamez-le-camembert-avec-les-librairies-fast-bert-et-transformers-14e65f84c148) (Web Search Result)
Il faudra aussi créer un jeu de données pour l'entraînement et pour la validation du modèle, spécifier le path vers le dossier qui contient un fichier .txt avec les noms de toutes les labels:

Thoughts

Le plan est de répondre à deux parties distinctes de la question:

1. Trouver un dataset approprié pour entraîner BERT
2. Expliquer comment optimiser l'entraînement de BERT sur un Mac M1

Analysons les résultats de recherche pour construire une réponse complète et utile:

Partie 1: Datasets pour entraîner BERT

D'après les résultats de recherche, voici quelques options de datasets pour entraîner BERT:

1. Cornell Movie-Dialogs Corpus

D'après le résultat #1, il s'agit d'un "corpus riche en métadonnées de conversations fictives extraites de scripts de films". Ce dataset semble être utilisé dans un tutoriel complet sur la préparation des données pour BERT. Il est particulièrement adapté pour l'entraînement NSP (Next Sentence Prediction) qui est l'une des tâches de pré-entraînement de BERT.

2. Datasets sur Kaggle et Google Dataset Search

Le résultat #5 mentionne que Google Dataset Search est "une des manières les plus efficaces pour trouver un dataset en faisant une simple recherche par mots-clés".

3. Datasets publics pour le français

Pour un entraînement en français, le résultat #9 fait référence à CamemBERT, qui est une version de BERT spécifiquement entraînée pour le français.

Je vais inclure d'autres sources bien connues de datasets comme:

- Hugging Face Datasets
- TensorFlow Datasets
- WikiText/Wikibooks
- BookCorpus
- Common Crawl

Partie 2: Optimisation de l'entraînement sur Mac M1

Les résultats de recherche ne parlent pas spécifiquement de l'optimisation sur Mac M1, mais je peux fournir des conseils basés sur les connaissances générales sur l'architecture M1 et l'entraînement de modèles ML:

1. Utiliser des frameworks optimisés pour M1:

- TensorFlow pour Mac M1 (TensorFlow-metal)
- PyTorch pour Mac avec support MPS (Metal Performance Shaders)

2. Techniques d'optimisation:

- Quantification des modèles
- Parallélisation des données
- Batch size approprié
- Apprentissage mixte (mixed precision)
- Utilisation efficace de la mémoire

3. Paramètres spécifiques à l'entraînement de BERT:

- Taille de modèle réduite (BERT-base plutôt que BERT-large)
- Longueur de séquence optimisée
- Techniques de distillation

Je vais organiser ma réponse en commençant par présenter les options de datasets, puis en donnant des conseils d'optimisation pour le Mac M1.

Datasets pour entraîner BERT et optimisation sur Mac M1

Datasets recommandés pour entraîner BERT

Pour entraîner ou fine-tuner BERT, voici plusieurs datasets adaptés à différents besoins:

1. Datasets généraux pour le pré-entraînement

- **Cornell Movie-Dialogs Corpus**: Collection de conversations extraites de scripts de films, parfait pour l'entraînement de la tâche NSP (Next Sentence Prediction) de BERT. Ce dataset est bien documenté avec un pipeline de préparation détaillé.

- **WikiText**: Corpus extrait de Wikipedia de haute qualité, disponible en plusieurs tailles (2M, 103M tokens). Il est parfaitement adapté au pré-entraînement de modèles linguistiques comme BERT.

- **BookCorpus**: Collection de livres non publiés, souvent utilisée en combinaison avec Wikipedia pour le pré-entraînement de BERT.

2. Datasets pour le fine-tuning en français

- **CamemBERT dataset**: Si vous travaillez en français, ce dataset basé sur le corpus OSCAR (Open Super-large Crawled Aggregated coRpus) est idéal pour adapter BERT au français.

- **Le French Treebank**: Corpus d'articles du journal Le Monde, annotés syntaxiquement.

3. Datasets spécifiques à une tâche

- **Pour la classification de texte**: Le dataset IMDb de critiques de films est excellent pour le fine-tuning de BERT sur une tâche binaire (positif/négatif).

- **Pour les questions-réponses**: SQuAD (Stanford Question Answering Dataset) est une référence pour entraîner BERT à répondre à des questions.

4. Où trouver ces datasets

- **Hugging Face Datasets**: Bibliothèque offrant un accès facile à des centaines de datasets prêts à l'emploi pour le NLP.

- **Google Dataset Search**: Un moteur de recherche spécialisé pour les datasets. Une simple recherche comme "dataset BERT training" peut donner d'excellents résultats.

- **Kaggle**: Contient de nombreux datasets de haute qualité, souvent avec des exemples d'utilisation.

Optimisation de l'entraînement sur Mac M1

L'architecture Apple Silicon M1 offre d'excellentes performances pour l'entraînement de modèles ML quand elle est bien utilisée:

1. Frameworks optimisés pour M1

```
# Pour TensorFlow avec accélération Metal
pip install tensorflow-macos
pip install tensorflow-metal
```

```
# Pour PyTorch avec accélération MPS
pip install torch torchvision torchaudio
```

Exemple de code pour activer l'accélération MPS dans PyTorch:

```
import torch
device = torch.device("mps" if torch.backends.mps.is_available() else "cpu")
model = model.to(device)
```

2. Techniques d'optimisation spécifiques à BERT

- **Réduction de la taille du modèle** : Préférez BERT-base (110M paramètres) à BERT-large (340M paramètres).
- **Réduire la longueur maximale des séquences** : Limitez à 128 ou 256 tokens au lieu de 512 pour économiser de la mémoire.
- **Utiliser des implémentations légères** : DistilBERT (66M paramètres) offre 95% des performances de BERT-base avec 40% moins de paramètres.
- **Mixed precision training** : Avec PyTorch:

```
from torch.cuda.amp import autocast, GradScaler

scaler = GradScaler()

with autocast(device_type='mps' if torch.backends.mps.is_available() else 'cpu'):
    outputs = model(input_ids, attention_mask=attention_mask, labels=labels)
    loss = outputs.loss

scaler.scale(loss).backward()
scaler.step(optimizer)
scaler.update()
```

- **Gradient accumulation** : Pour utiliser des batch sizes plus grands:

```
accumulation_steps = 4 # Accumule les gradients sur 4 batches
for i, batch in enumerate(train_dataloader):
    # Forward pass
    outputs = model(**batch)
    loss = outputs.loss / accumulation_steps
    loss.backward()

    # Update weights après accumulation_steps
    if (i + 1) % accumulation_steps == 0:
        optimizer.step()
        optimizer.zero_grad()
```

3. Optimisations système

- **Désactivez les applications en arrière-plan** pour libérer la mémoire.
- **Augmentez la mémoire d'échange** si nécessaire.
- **Utilisez un SSD externe** rapide pour stocker les datasets volumineux.
- **Refroidissement** : Assurez-vous que votre Mac est bien ventilé pendant l'entraînement.

4. Entraînement distribué sur plusieurs machines

Si un seul Mac M1 n'est pas suffisant, envisagez d'utiliser des instances cloud comme alternative, ou une approche d'entraînement distribuée avec plusieurs machines.

Pour un projet de thèse, je vous recommande également d'explorer des techniques comme l'apprentissage par transfert (partir d'un modèle pré-entraîné et ne fine-tuner que certaines couches), qui peuvent considérablement accélérer l'entraînement tout en maintenant d'excellentes performances.

Avez-vous besoin d'aide pour implémenter une de ces solutions en particulier?

USER

Est ce qu'il est possible de d'augmenter le context windows de bert en réutilisant le technique utiliser dans ce papier

BOT

Thoughts

La question demande s'il est possible d'augmenter la fenêtre de contexte (context window) de BERT en réutilisant les techniques décrites dans le papier "MOBA: Mixture of Block Attention for Long-Context LLMs".

Analysons d'abord le papier pour comprendre l'approche MOBA, puis évaluons si cette technique pourrait être appliquée à BERT.

Le papier MOBA présente:

1. Une nouvelle architecture d'attention appelée "Mixture of Block Attention" pour les LLMs à long contexte
2. MOBA divise le contexte en blocs et utilise un mécanisme de routage/gating pour sélectionner dynamiquement les blocs les plus pertinents pour chaque token de requête
3. Cela permet une attention sparse qui réduit la complexité computationnelle tout en maintenant les performances

Pour déterminer si MOBA peut être utilisé pour augmenter la fenêtre de contexte de BERT, il faut considérer plusieurs éléments:

1. Différences architecturales entre BERT et les modèles de type Transformer plus récents
2. Le mécanisme d'attention de BERT et sa complexité
3. Les défis spécifiques liés à l'extension de la fenêtre de contexte dans BERT
4. La compatibilité de MOBA avec l'architecture BERT

Analyse:

BERT (Bidirectional Encoder Representations from Transformers) est un modèle d'encodeur bidirectionnel qui utilise l'attention standard avec une complexité quadratique $O(n^2)$. Sa fenêtre de contexte standard est de 512 tokens, ce qui est une limitation pour de nombreuses applications.

MOBA propose une solution pour réduire cette complexité quadratique en implémentant une attention par blocs, où chaque token de requête est dirigé vers un sous-ensemble des blocs de clés et de valeurs les plus pertinents, plutôt que d'interagir avec l'ensemble du contexte.

Possibilité d'appliquer MOBA à BERT:

1. Compatibilité architecturale: MOBA peut être intégré dans un transformer standard en remplaçant l'attention complète par l'attention par blocs mixtes, sans changer le reste de l'architecture. Cela suggère qu'il pourrait théoriquement être appliqué à BERT.
2. Bidirectionnalité: BERT est bidirectionnel, ce qui signifie que chaque token peut assister à tous les autres tokens (passés et futurs). MOBA a été conçu principalement pour les modèles auto-régressifs avec masquage causal, mais il pourrait être adapté pour fonctionner dans un contexte bidirectionnel.
3. Modification de l'entraînement: L'application de MOBA à BERT nécessiterait soit un réentraînement complet, soit un fine-tuning pour adapter le modèle à ce nouveau mécanisme d'attention.
4. Préservation des performances: Le papier montre que MOBA maintient des performances comparables à l'attention complète pour les tâches de modélisation linguistique, mais il faudrait vérifier si cela se traduit aussi bien pour les tâches spécifiques de BERT comme MLM (Masked Language Modeling) et NSP (Next Sentence Prediction).

D'après le papier, MOBA pourrait bien s'adapter à BERT:

- Il est conçu pour être un substitut flexible à l'attention complète
- Il permet une transition transparente entre l'attention complète et l'attention sparse
- Il maintient des performances comparables tout en réduisant significativement la complexité computationnelle

Modifications nécessaires pour BERT:

1. Adaptation de la section causale: BERT étant bidirectionnel, la contrainte de causalité dans MOBA devrait être supprimée ou adaptée.
2. Routage bidirectionnel: Le mécanisme de routage devrait prendre en compte les tokens passés et futurs.
3. Apprentissage des blocs pertinents: Le modèle devrait apprendre à sélectionner les blocs pertinents dans un contexte bidirectionnel.

En conclusion, il semble techniquement possible d'appliquer MOBA à BERT pour augmenter sa fenêtre de contexte, mais cela nécessiterait des adaptations et probablement un réentraînement. Les expériences du papier montrant que MOBA maintient des performances comparables à l'attention complète suggèrent que cette approche pourrait être fructueuse pour BERT également.

Adaptation de MOBA pour augmenter la fenêtre de contexte de BERT

Oui, il est tout à fait possible d'adapter la technique MOBA (Mixture of Block Attention) décrite dans ce papier pour augmenter la fenêtre de contexte de BERT. Cette approche présente plusieurs avantages intéressants pour BERT.

Comment MOBA pourrait être appliqué à BERT

1. **Adaptation à la bidirectionnalité**: Contrairement aux modèles autorégressifs mentionnés dans le papier, BERT est un modèle bidirectionnel qui peut regarder les tokens avant et après dans une séquence. Pour adapter MOBA:
 - La contrainte de causalité (qui empêche d'accéder aux tokens futurs) devrait être supprimée
 - Le routage des blocs devrait fonctionner dans les deux directions
 - Tous les blocs pourraient potentiellement être sélectionnés par le mécanisme de gating
2. **Implémentation technique**:
 - Diviser la séquence d'entrée en blocs (comme proposé dans MOBA)
 - Implémenter le mécanisme de gating qui sélectionne dynamiquement les k blocs les plus pertinents pour chaque token
 - Appliquer l'attention uniquement entre chaque token et ses blocs sélectionnés
3. **Avantages pour BERT**:
 - Réduction significative de la complexité computationnelle (de $O(n^2)$ à sous-quadratique)
 - Possibilité d'augmenter la fenêtre de contexte de 512 tokens à plusieurs milliers
 - Maintien des performances du modèle comme démontré dans le papier

Considérations d'implémentation

Pour mettre en œuvre cette approche, vous auriez besoin de:

1. **Modifier l'architecture d'attention** dans BERT pour incorporer le mécanisme de sélection de blocs

```
# Pseudo-code pour une implémentation MOBA dans BERT
def moba_attention(self, query, key, value, block_size=512, top_k=3):
    # Diviser les clés et valeurs en blocs
    blocks_k = split_into_blocks(key, block_size)
    blocks_v = split_into_blocks(value, block_size)

    # Calculer les scores de pertinence entre query et chaque bloc
    block_scores = compute_block_relevance(query, blocks_k)

    # Sélectionner les top-k blocs
    selected_blocks = top_k_blocks(block_scores, k=top_k)

    # Appliquer l'attention seulement aux blocs sélectionnés
    outputs = []
    for q_i in query:
        blocks_i = selected_blocks[i]
        attention_output = self_attention(q_i,
            concat([blocks_k[j] for j in blocks_i]),
            concat([blocks_v[j] for j in blocks_i]))
        outputs.append(attention_output)

    return concat(outputs)
```

2. ****Stratégie d'entraînement****: Le papier suggère deux approches efficaces:
- Entraînement hybride: commencer avec MOBA puis basculer vers l'attention complète pour les derniers 10% des tokens
 - Hybride par couche: utiliser MOBA pour la plupart des couches mais garder l'attention complète pour les dernières couches

Défis potentiels

1. ****Réentraînement nécessaire****: L'adaptation de BERT à MOBA nécessiterait probablement un réentraînement ou au moins un fine-tuning approfondi
2. ****Tâches spécifiques à BERT****: Il faudrait vérifier que les performances sur les tâches de Masked Language Modeling et Next Sentence Prediction restent bonnes
3. ****Granularité des blocs****: Comme mentionné dans le papier, la performance est sensible à la granularité des blocs, ce qui nécessite une optimisation spécifique

L'application de MOBA à BERT constituerait une extension intéressante du travail présenté dans ce papier, et pourrait significativement augmenter l'utilité de BERT pour des applications nécessitant une compréhension de textes longs.

Sources: [2502.13189v1.pdf](#)

USER

nous allons utiliser la nouvelle implémentation de bert comme le papier suivant applique MOBA a ce nouveau bert

BOT

Thoughts

Vous souhaitez appliquer la technique MOBA (Mixture of Block Attention) présentée dans le deuxième papier (2502.13189v1.pdf) au modèle NeoBERT présenté dans le premier papier (2502.19587v1.pdf).

Analysons ce que cela impliquerait:

À propos de NeoBERT

D'après le premier papier:

- NeoBERT est un encodeur bidirectionnel de nouvelle génération qui améliore les capacités des modèles comme BERT et RoBERTa
- Il intègre des améliorations dans l'architecture, les données d'entraînement, et les méthodologies de pré-entraînement
- NeoBERT utilise un rapport profondeur/largeur optimisé (28 couches, 768 dimensions cachées)
- Il a une fenêtre de contexte étendue de 4 096 tokens
- Il utilise 250M de paramètres
- Il surpasse les modèles comme BERTlarge, RoBERTalarge, NomicBERT et ModernBERT

À propos de MOBA

D'après le deuxième papier:

- MOBA (Mixture of Block Attention) est une technique pour améliorer l'efficacité des modèles de langage sur de longs contextes
- Elle divise le contexte en blocs et applique un mécanisme de gating qui sélectionne dynamiquement les blocs les plus pertinents
- MOBA est inspiré par les techniques de Mixture of Experts (MoE)
- Il permet de réduire considérablement la complexité computationnelle tout en maintenant des performances similaires
- MOBA permet une transition fluide entre l'attention complète et l'attention sparse
- Il est particulièrement efficace pour les longues séquences

Application de MOBA à NeoBERT

Pour appliquer MOBA à NeoBERT, voici les étapes conceptuelles:

1. ****Modification du mécanisme d'attention****:
 - Remplacer l'attention standard dans NeoBERT par le mécanisme MOBA
 - Diviser les séquences d'entrée en blocs (par exemple, de taille 512 tokens)
 - Implémenter le mécanisme de routage qui sélectionne les top-k blocs pour chaque token
2. ****Adaptation à l'architecture bidirectionnelle****:
 - Alors que MOBA a été conçu principalement pour des modèles autorégressifs, NeoBERT est bidirectionnel
 - Les contraintes de causalité dans MOBA devraient être adaptées ou supprimées pour permettre une attention bidirectionnelle
3. ****Entraînement/Fine-tuning****:
 - MOBA suggère une stratégie d'entraînement hybride: commencer avec MOBA puis passer à l'attention complète pour les derniers 10% des tokens
 - Alternativement, utiliser une approche hybride par couche: conserver l'attention complète pour certaines couches
4. ****Avantages attendus****:
 - Extension potentielle de la fenêtre de contexte de NeoBERT bien au-delà de 4 096 tokens
 - Réduction significative des coûts computationnels pour l'inférence et l'entraînement
 - Maintien des performances sur les tâches MTEB et GLUE

Je vais maintenant formuler une réponse cohérente qui explique comment appliquer MOBA à NeoBERT et les bénéfices potentiels de cette combinaison.

Application de MOBA au modèle NeoBERT

Vous avez raison de vouloir combiner ces deux innovations! L'intégration de MOBA (Mixture of Block Attention) au modèle NeoBERT représente une approche prometteuse pour améliorer l'efficacité des modèles d'encodeur tout en préservant leurs performances sur les longues séquences. Voici comment vous pourriez procéder:

Architecture combinée NeoBERT-MOBA

1. ****Adaptation du mécanisme d'attention****:
 - Remplacer l'attention standard dans NeoBERT par le mécanisme MOBA
 - Diviser la fenêtre de contexte en blocs (par exemple, blocs de 512 tokens)
 - Implémenter le mécanisme de gating qui sélectionne dynamiquement les top-k blocs les plus pertinents
2. ****Modification pour l'architecture bidirectionnelle****:

```
def moba_attention(self, query, key, value, block_size=512, top_k=3):  
    # Diviser les clés et valeurs en blocs  
    blocks_k = split_into_blocks(key, block_size)
```

```
blocks_v = split_into_blocks(value, block_size)

# Calculer les scores de pertinence entre query et chaque bloc
# Pour NeoBERT, on supprime la contrainte de causalité de MOBA original
block_scores = compute_block_relevance(query, blocks_k)

# Sélectionner les top-k blocs
selected_blocks = top_k_blocks(block_scores, k=top_k)

# Appliquer l'attention seulement aux blocs sélectionnés
outputs = []
for q_i in query:
    blocks_i = selected_blocks[i]
    attention_output = self_attention(q_i,
                                      concat([blocks_k[j] for j in blocks_i]),
                                      concat([blocks_v[j] for j in blocks_i]))
    outputs.append(attention_output)

return concat(outputs)
```

Avantages de cette intégration

- **Extension de la fenêtre de contexte**:**
 - NeoBERT a une fenêtre de contexte de 4 096 tokens
 - Avec MOBA, vous pourriez théoriquement l'étendre à 32K, 64K voire 128K tokens
 - Les résultats de MOBA montrent qu'il fonctionne bien jusqu'à 1M tokens
- **Efficacité computationnelle**:**
 - Réduction significative de la complexité (d' $O(n^2)$ à $O(n \cdot k \cdot b)$ où k est le nombre de blocs sélectionnés et b la taille du bloc)
 - D'après les benchmarks dans le papier MOBA, vous pourriez obtenir un gain de vitesse de 6,5x pour des séquences de 1M tokens
- **Stratégies d'entraînement flexibles**:**
 - Utilisez l'approche hybride MOBA/attention complète: commencez avec MOBA pour 90% de l'entraînement puis passez à l'attention complète
 - Ou utilisez l'approche hybride par couche: gardez les 3 dernières couches en attention complète, les autres en MOBA

Considérations d'implémentation

- **Granularité des blocs**:**
 - Testez différentes tailles de bloc (les expériences de MOBA montrent que la granularité fine est meilleure)
 - Un bon point de départ serait des blocs de 512 tokens avec $\text{top-k}=3$
- **Adaptation à l'architecture bidirectionnelle**:**
 - Contrairement au MOBA original conçu pour les modèles autorégressifs, supprimez les contraintes de causalité
 - Tous les blocs peuvent être considérés, pas seulement les blocs passés
- **Préservation de la précision pour MTEB**:**
 - Pour maintenir les performances de NeoBERT sur MTEB, utilisez MOBA uniquement pendant le prefill et passez à l'attention complète pendant la génération

Cette combinaison pourrait créer un encodeur bidirectionnel exceptionnellement efficace capable de traiter des contextes beaucoup plus longs que les modèles actuels, tout en conservant les performances de pointe de NeoBERT sur les benchmarks MTEB et GLUE.

Souhaitez-vous explorer un aspect particulier de cette intégration ou commencer par implémenter une version simplifiée pour tester ce concept?

Sources: [2502.19587v1.pdf](#) [2502.13189v1.pdf](#)

USER

Écrit moi tous le code qui combine Moba et NeoBert,

BOT

Thoughts

La demande consiste à combiner l'architecture MOBA (Mixture of Block Attention) avec NeoBERT (une nouvelle génération d'architectures BERT). Je vais devoir écrire un code qui intègre ces deux approches.

Basé sur les documents fournis:

- MOBA (Mixture of Block Attention) est une approche qui:
 - Divise le contexte en blocs
 - Utilise un mécanisme de gating pour sélectionner dynamiquement les blocs les plus pertinents pour chaque token
 - Réduit la complexité quadratique de l'attention classique à une complexité sous-quadratique
 - Permet de gérer efficacement de très longues séquences (jusqu'à 1M tokens)
- NeoBERT est une nouvelle architecture BERT qui:
 - Utilise des améliorations architecturales modernes (RoPE, SwiGLU, RMSNorm)
 - Est entraîné sur des données plus récentes et volumineuses
 - Supporte des contextes plus longs (4096 tokens)
 - A une architecture à 28 couches avec 768 de dimension cachée

Pour combiner les deux, je vais:

- Prendre l'architecture NeoBERT comme base
- Remplacer le mécanisme d'attention standard par MOBA
- Conserver les autres améliorations de NeoBERT (activations SwiGLU, RMSNorm, etc.)

Le code devra:

- Implémenter l'architecture NeoBERT
- Implémenter le mécanisme MOBA
- Intégrer MOBA dans NeoBERT

Voyons comment structurer ce code en Python:

Code d'implémentation de NeoBERT avec MOBA (Mixture of Block Attention)

Voici l'implémentation complète qui combine l'architecture NeoBERT avec le mécanisme MOBA pour gérer efficacement les longues séquences:

```
import math
import torch
import torch.nn as nn
import torch.nn.functional as F
from typing import Optional, Tuple, List, Union
import numpy as np

class RMSNorm(nn.Module):
    """Root Mean Square LayerNormalization"""
    def __init__(self, dim: int, eps: float = 1e-6):
        super().__init__()
        self.eps = eps
        self.weight = nn.Parameter(torch.ones(dim))

    def forward(self, x):
        # x: [batch_size, seq_len, dim]
        variance = x.pow(2).mean(-1, keepdim=True)
        x = x * torch.rsqrt(variance + self.eps)
        return x * self.weight

class RotaryEmbedding(nn.Module):
    """Rotary Position Embeddings (RoPE)"""
    def __init__(self, dim: int, base: int = 10000):
        super().__init__()
        self.dim = dim
        self.base = base
        self.inv_freq = 1.0 / (base ** (torch.arange(0, dim, 2).float() / dim))
        self.max_seq_len_cached = None
        self.cos_cached = None
        self.sin_cached = None

    def forward(self, x, seq_len: int):
        # x: [batch_size, seq_len, n_heads, head_dim]
        if seq_len > self.max_seq_len_cached or self.cos_cached is None or self.sin_cached is None:
            self._update_cache(seq_len)

        cos = self.cos_cached[:seq_len].to(x.device)
        sin = self.sin_cached[:seq_len].to(x.device)
        return (x * cos) + (self._rotate_half(x) * sin)

    def _update_cache(self, seq_len: int):
        self.max_seq_len_cached = seq_len
        t = torch.arange(seq_len, device=self.inv_freq.device).type_as(self.inv_freq)
        freqs = torch.einsum('i,j->ij', t, self.inv_freq)
        emb = torch.cat((freqs, freqs), dim=-1)
        self.cos_cached = emb.cos()[None, None, :, :]
        self.sin_cached = emb.sin()[None, None, :, :]

    def _rotate_half(self, x):
        x1, x2 = x[..., : self.dim // 2], x[..., self.dim // 2 :]
        return torch.cat((-x2, x1), dim=-1)

class MoBAAttention(nn.Module):
    """Mixture of Block Attention (MoBA)"""
    def __init__(self, dim: int, num_heads: int, block_size: int = 512, top_k: int = 3):
        super().__init__()
        assert dim % num_heads == 0, "dim must be divisible by num_heads"

        self.dim = dim
        self.num_heads = num_heads
        self.head_dim = dim // num_heads
        self.block_size = block_size
        self.top_k = top_k

        self.q_proj = nn.Linear(dim, dim, bias=False)
        self.k_proj = nn.Linear(dim, dim, bias=False)
        self.v_proj = nn.Linear(dim, dim, bias=False)
        self.o_proj = nn.Linear(dim, dim, bias=False)

        self.rotary = RotaryEmbedding(self.head_dim)

    def _split_heads(self, x, is_key=False):
        # x: [batch_size, seq_len, dim]
        new_shape = x.size()[:-1] + (self.num_heads, self.head_dim)
        x = x.view(*new_shape) # [batch_size, seq_len, num_heads, head_dim]
        if is_key:
            # For keys, we keep the original shape for easier block processing
            return x
        else:
            # For queries and values, we transpose for attention computation
            return x.transpose(1, 2) # [batch_size, num_heads, seq_len, head_dim]

    def _merge_heads(self, x):
        # x: [batch_size, num_heads, seq_len, head_dim]
        x = x.transpose(1, 2) # [batch_size, seq_len, num_heads, head_dim]
        new_shape = x.size()[:-2] + (self.dim,)
        return x.reshape(*new_shape) # [batch_size, seq_len, dim]
```

```

def forward(self, q, k, v, attention_mask=None):
    batch_size, q_len = q.size(0), q.size(1)
    k_len = k.size(1)

    # Apply projections
    q = self.q_proj(q) # [batch_size, q_len, dim]
    k = self.k_proj(k) # [batch_size, k_len, dim]
    v = self.v_proj(v) # [batch_size, k_len, dim]

    # Split heads
    q = self._split_heads(q) # [batch_size, num_heads, q_len, head_dim]
    k_blocks = self._split_heads(k, is_key=True) # [batch_size, k_len, num_heads, head_dim]
    v_blocks = self._split_heads(v, is_key=True) # [batch_size, k_len, num_heads, head_dim]

    # Apply rotary embeddings
    q = self.rotary(q, q_len)
    k_blocks = self.rotary(k_blocks, k_len)

    # Split into blocks
    num_blocks = math.ceil(k_len / self.block_size)
    k_mean_blocks = []

    # Calculate mean representations for each block
    for i in range(num_blocks):
        start_idx = i * self.block_size
        end_idx = min(start_idx + self.block_size, k_len)
        if start_idx < end_idx: # Ensure non-empty block
            # Mean pooling of keys in this block
            k_mean = k_blocks[:, start_idx:end_idx].mean(dim=1) # [batch_size, num_heads, head_dim]
            k_mean_blocks.append(k_mean)

    k_mean_blocks = torch.stack(k_mean_blocks, dim=2) # [batch_size, num_heads, num_blocks, head_dim]

    # Compute block selection scores
    # q: [batch_size, num_heads, q_len, head_dim]
    # k_mean_blocks: [batch_size, num_heads, num_blocks, head_dim]
    block_scores = torch.matmul(q, k_mean_blocks.transpose(-1, -2)) # [batch_size, num_heads, q_len, num_blocks]

    # Apply causal mask to block scores if needed
    if attention_mask is not None:
        # Create block-level causal mask
        # For each query position, identify which blocks it can attend to
        q_positions = torch.arange(q_len, device=q.device)[:, None] # [q_len, 1]
        block_starts = torch.arange(num_blocks, device=q.device) * self.block_size # [num_blocks]
        causal_mask = q_positions >= block_starts # [q_len, num_blocks]
        causal_mask = causal_mask.unsqueeze(0).unsqueeze(1) # [1, 1, q_len, num_blocks]
        block_scores = block_scores.masked_fill(~causal_mask, float('-inf'))

    # Make sure we always attend to the current block
    current_blocks = torch.arange(q_len, device=q.device) // self.block_size
    current_blocks = current_blocks.clamp(max=num_blocks-1) # Ensure we don't exceed the number of blocks

    # Get top-k blocks for each query (excluding current block)
    block_scores_without_current = block_scores.clone()

    # Mask out current blocks to get distinct blocks
    for i in range(batch_size):
        for h in range(self.num_heads):
            for j in range(q_len):
                curr_block = current_blocks[j]
                block_scores_without_current[i, h, j, curr_block] = float('-inf')

    # Get top k-1 blocks (since we'll always include the current block)
    top_k_values, top_k_indices = torch.topk(
        block_scores_without_current, min(self.top_k - 1, num_blocks - 1), dim=-1
    ) # [batch_size, num_heads, q_len, top_k-1]

    # Initialize output tensor
    output = torch.zeros(
        batch_size, self.num_heads, q_len, self.head_dim, device=q.device, dtype=q.dtype
    )

    # Process each query token
    for i in range(batch_size):
        for h in range(self.num_heads):
            for j in range(q_len):
                # Always include current block
                curr_block = current_blocks[j]
                blocks_to_attend = [curr_block]

                # Add top k-1 blocks
                for k_idx in range(top_k_values.size(-1)):
                    if top_k_values[i, h, j, k_idx] > float('-inf'):
                        blocks_to_attend.append(top_k_indices[i, h, j, k_idx].item())

                # Calculate start and end indices for each block
                attn_inputs = []
                for block_idx in blocks_to_attend:
                    start_idx = block_idx * self.block_size
                    end_idx = min(start_idx + self.block_size, k_len)

                    # Apply causal mask within current block
                    if block_idx == curr_block and j >= start_idx and j < end_idx:

```

```

        # If this is the current block and the query is in this block
        # Apply causal masking within the block
        k_block = k_blocks[i, start_idx:end_idx, h] # [block_size, head_dim]
        v_block = v_blocks[i, start_idx:end_idx, h] # [block_size, head_dim]

        # Only attend to positions up to j within the current block
        curr_pos = j - start_idx
        k_block = k_block[:curr_pos+1]
        v_block = v_block[:curr_pos+1]
    else:
        # For other blocks, use the full block
        k_block = k_blocks[i, start_idx:end_idx, h] # [block_size, head_dim]
        v_block = v_blocks[i, start_idx:end_idx, h] # [block_size, head_dim]

    if k_block.size(0) > 0: # Ensure non-empty block
        attn_inputs.append((k_block, v_block))

    if attn_inputs:
        # Concatenate all selected blocks
        k_concat = torch.cat([k for k, _ in attn_inputs], dim=0) # [total_tokens, head_dim]
        v_concat = torch.cat([v for _, v in attn_inputs], dim=0) # [total_tokens, head_dim]

        # Compute attention scores
        q_j = q[i, h, j].unsqueeze(0) # [1, head_dim]
        attn_scores = torch.matmul(q_j, k_concat.transpose(0, 1)) / math.sqrt(self.head_dim) # [1, total_tokens]
        attn_probs = F.softmax(attn_scores, dim=-1)

        # Apply attention
        head_output = torch.matmul(attn_probs, v_concat) # [1, head_dim]
        output[i, h, j] = head_output

    # Merge heads and project back
    output = self._merge_heads(output) # [batch_size, q_len, dim]
    output = self.o_proj(output) # [batch_size, q_len, dim]

    return output

```

```

class SwiGLU(nn.Module):
    """SwiGLU activation as used in NeoBERT"""
    def __init__(self, dim: int, hidden_dim: int = None):
        super().__init__()
        hidden_dim = hidden_dim or int(2/3 * 4 * dim) # 8/3 * dim as in NeoBERT paper

        self.w1 = nn.Linear(dim, hidden_dim, bias=False)
        self.w2 = nn.Linear(dim, hidden_dim, bias=False)
        self.w3 = nn.Linear(hidden_dim, dim, bias=False)

    def forward(self, x):
        gate = F.silu(self.w1(x))
        value = self.w2(x)
        return self.w3(gate * value)

```

```

class NeoBERTLayer(nn.Module):
    """NeoBERT Layer with MOBA attention"""
    def __init__(
        self,
        dim: int,
        num_heads: int,
        moba_block_size: int = 512,
        moba_top_k: int = 3,
        ffn_hidden_dim: int = None
    ):
        super().__init__()
        self.pre_attn_norm = RMSNorm(dim)
        self.attention = MoBAAttention(
            dim=dim,
            num_heads=num_heads,
            block_size=moba_block_size,
            top_k=moba_top_k
        )
        self.pre_ffn_norm = RMSNorm(dim)
        self.ffn = SwiGLU(dim, ffn_hidden_dim)

    def forward(self, x, attention_mask=None):
        # Pre-LN for attention
        attn_input = self.pre_attn_norm(x)
        attn_output = self.attention(attn_input, attn_input, attn_input, attention_mask)
        x = x + attn_output

        # Pre-LN for FFN
        ffn_input = self.pre_ffn_norm(x)
        ffn_output = self.ffn(ffn_input)
        return x + ffn_output

```

```

class NeoBERTEmbedding(nn.Module):
    """Token embeddings for NeoBERT"""
    def __init__(self, vocab_size: int, dim: int, max_position_embeddings: int = 4096):
        super().__init__()
        self.token_embeddings = nn.Embedding(vocab_size, dim)
        self.norm = RMSNorm(dim)
        self.dropout = nn.Dropout(0.1)

```

```

def forward(self, input_ids):
    embeddings = self.token_embeddings(input_ids)
    embeddings = self.norm(embeddings)
    return self.dropout(embeddings)

class NeoBERTMoBA(nn.Module):
    """NeoBERT model with MOBA attention"""
    def __init__(
        self,
        vocab_size: int = 30000,
        hidden_size: int = 768,
        num_hidden_layers: int = 28,
        num_attention_heads: int = 12,
        moba_block_size: int = 512,
        moba_top_k: int = 3,
        max_position_embeddings: int = 4096,
    ):
        super().__init__()
        self.embeddings = NeoBERTEmbedding(
            vocab_size=vocab_size,
            dim=hidden_size,
            max_position_embeddings=max_position_embeddings
        )

        # Create transformer layers with MOBA attention
        self.layers = nn.ModuleList([
            NeoBERTLayer(
                dim=hidden_size,
                num_heads=num_attention_heads,
                moba_block_size=moba_block_size,
                moba_top_k=moba_top_k
            ) for _ in range(num_hidden_layers)
        ])

        self.final_norm = RMSNorm(hidden_size)

        # MLM and NSP heads
        self.mlm_dense = nn.Linear(hidden_size, hidden_size)
        self.mlm_norm = RMSNorm(hidden_size)
        self.mlm_head = nn.Linear(hidden_size, vocab_size)

        self.init_weights()

    def init_weights(self):
        # Initialize weights similar to NeoBERT paper
        for module in self.modules():
            if isinstance(module, nn.Linear):
                # Initialize linear layers with small random values
                module.weight.data.normal_(mean=0.0, std=0.02)
                if module.bias is not None:
                    module.bias.data.zero_()

    def forward(
        self,
        input_ids: torch.LongTensor,
        attention_mask: Optional[torch.Tensor] = None,
        token_type_ids: Optional[torch.LongTensor] = None,
        return_hidden_states: bool = False
    ):
        batch_size, seq_length = input_ids.size()

        # Get embeddings
        hidden_states = self.embeddings(input_ids)

        # Store hidden states if needed
        all_hidden_states = [] if return_hidden_states else None

        # Forward pass through all layers
        for layer in self.layers:
            if return_hidden_states:
                all_hidden_states.append(hidden_states)

            hidden_states = layer(hidden_states, attention_mask)

        # Final layer normalization
        hidden_states = self.final_norm(hidden_states)

        if return_hidden_states:
            all_hidden_states.append(hidden_states)

        # MLM head
        mlm_output = self.mlm_dense(hidden_states)
        mlm_output = F.gelu(mlm_output)
        mlm_output = self.mlm_norm(mlm_output)
        prediction_scores = self.mlm_head(mlm_output)

        outputs = {
            'last_hidden_state': hidden_states,
            'prediction_logits': prediction_scores
        }

        if return_hidden_states:

```

```

        outputs['hidden_states'] = all_hidden_states

    return outputs

def get_extended_attention_mask(self, attention_mask, input_shape):
    # Create attention mask
    if attention_mask is None:
        attention_mask = torch.ones(input_shape, device=self.device)

    # We create a 3D attention mask from a 2D tensor mask.
    # Sizes are [batch_size, 1, 1, to_seq_length]
    # So we can broadcast to [batch_size, num_heads, from_seq_length, to_seq_length]
    extended_attention_mask = attention_mask[:, None, None, :]

    # Convert attention mask to binary:
    extended_attention_mask = (1.0 - extended_attention_mask) * -10000.0

    return extended_attention_mask

# Fonction utilitaire pour tester le modèle
def test_neobert_moba():
    # Créer un petit modèle pour tester
    model = NeoBERTMoBA(
        vocab_size=30000,
        hidden_size=768,
        num_hidden_layers=28,
        num_attention_heads=12,
        moba_block_size=512,
        moba_top_k=3
    )

    # Créer des données d'entrée factices
    batch_size = 2
    seq_length = 1024
    input_ids = torch.randint(0, 30000, (batch_size, seq_length))
    attention_mask = torch.ones((batch_size, seq_length))

    # Forward pass
    outputs = model(input_ids, attention_mask)

    # Vérifier les formes de sortie
    print(f"Last hidden state shape: {outputs['last_hidden_state'].shape}")
    print(f"Prediction logits shape: {outputs['prediction_logits'].shape}")

    print("Test terminé avec succès!")

if __name__ == "__main__":
    test_neobert_moba()

```

Explication du code

Le code ci-dessus combine les architectures NeoBERT et MOBA avec les caractéristiques suivantes:

- Architecture NeoBERT**:
 - Utilise RMSNorm au lieu de LayerNorm
 - Intègre des activations SwiGLU
 - Emploie RoPE (Rotary Position Embeddings) pour encoder les positions
 - Structure avec 28 couches et dimension cachée de 768
 - Pre-LayerNorm pour une meilleure stabilité
- Mécanisme MOBA (Mixture of Block Attention)**:
 - Division du contexte en blocs (taille par défaut: 512 tokens)
 - Sélection dynamique des top-k blocs les plus pertinents (par défaut: 3)
 - Maintien de la causalité (pas d'attention aux blocs futurs)
 - Attention toujours au bloc courant avec masquage causal à l'intérieur du bloc
- Optimisations**:
 - Vectorisation des opérations avec PyTorch
 - Pas de biais dans les couches linéaires (comme dans NeoBERT)
 - Dimensionnement adapté de la couche intermédiaire SwiGLU

Pour utiliser ce modèle avec des contextes plus longs, vous pouvez ajuster les paramètres `moba_block_size` et `moba_top_k` pour optimiser la performance et la précision selon vos besoins.

Sources: [2502.19587v1.pdf](#) [2502.13189v1.pdf](#)

USER

fait moi un protocole d'entrainement et de teste pour cette nouvelle architecture

BOT

Thoughts

La personne me demande de créer un protocole d'entraînement et de test pour l'architecture combinée MOBA-NeoBERT que nous avons discutée dans les échanges précédents.

D'après les documents PDF partagés, j'ai des informations détaillées sur:

- NeoBERT: Un nouvel encodeur bidirectionnel avec des améliorations architecturales (RMSNorm, SwiGLU, etc.)

2. MOBA (Mixture of Block Attention): Une technique d'attention efficace pour les longs contextes qui divise les séquences en blocs et utilise un mécanisme de routage pour sélectionner les blocs les plus pertinents.

Je vais élaborer un protocole complet couvrant:

- La préparation des données
- L'architecture combinée
- Les phases d'entraînement (pré-entraînement, ajustement)
- Les métriques et protocoles d'évaluation
- Les expériences d'ablation pour mesurer l'impact de MOBA sur NeoBERT

Protocole d'entraînement et de test pour l'architecture NeoBERT-MOBA

Aperçu du protocole

Ce document détaille les étapes pour entraîner et évaluer un modèle combinant l'architecture NeoBERT avec le mécanisme d'attention MOBA (Mixture of Block Attention), permettant d'obtenir un encodeur bidirectionnel performant capable de gérer efficacement des contextes longs.

1. Préparation des données

1.1. Datasets de pré-entraînement

- **Corpus principal**: RefinedWeb (600B tokens)
- **Données supplémentaires pour les contextes longs**:
 - RefinedWeb1024+ (séquences > 1024 tokens)
 - RefinedWeb2048+ (séquences > 2048 tokens)
 - Wikipedia pour les évaluations de longue portée

1.2. Prétraitement

- Tokenisation avec le tokenizer WordPiece original (30K tokens)
- Segmentation en séquences de longueurs variables (1024, 4096, jusqu'à 32K tokens)
- Taux de masquage à 20% pour le MLM (Masked Language Modeling)

2. Architecture du modèle

2.1. Configuration NeoBERT-MOBA

```
model_config = {  
    # Configuration NeoBERT  
    "vocab_size": 30000,  
    "hidden_size": 768,  
    "num_hidden_layers": 28,  
    "num_attention_heads": 12,  
    "max_position_embeddings": 32768,  
    "activation_function": "swiglu",  
    "layer_norm_type": "rms_norm",  
  
    # Configuration MOBA  
    "moba_block_size": 512, # Taille des blocs  
    "moba_top_k": 3, # Nombre de blocs à sélectionner  
    "hybrid_layer_count": 3, # Dernières couches en attention complète  
}
```

2.2. Structure hybride

- Les `num_hidden_layers - hybrid_layer_count` premières couches utilisent MOBA
- Les `hybrid_layer_count` dernières couches utilisent l'attention standard
- Chaque token est toujours routé vers son bloc courant avec masquage causal

3. Protocole d'entraînement

3.1. Phase 1: Pré-entraînement initial (1024 tokens)

- **Objectif**: MLM uniquement
- **Durée**: 1M steps (2T tokens)
- **Batch size**: 2M tokens (batch local de 32, accumulation de gradient sur 8 pas)
- **Hyperparamètres**:
 - Optimizer: AdamW ($\beta_1=0.9$, $\beta_2=0.95$, $\epsilon=10^{-8}$)
 - Learning rate: $6e-4$ avec 2000 steps de warmup et décroissance cosinus
 - Weight decay: 0.1
 - Gradient clipping: 1.0

3.2. Phase 2: Extension du contexte (4096 tokens)

- **Durée**: 50K steps (100B tokens)
- **Stratégie de sampling**:
 - RefinedWeb: 20%
 - RefinedWeb1024+: 40%
 - RefinedWeb2048+: 40%
- **MOBA désactivé**: Utiliser l'attention standard pour cette phase

3.3. Phase 3: Activation MOBA (jusqu'à 32K tokens)

- **Durée**: 50K steps (100B tokens)
- **Stratégie MOBA hybride**:
 - Phase A (90% des steps): Utiliser MOBA sur toutes les couches sauf les 3 dernières
 - Phase B (10% des steps): Basculer vers l'attention complète
- **Paramètres MOBA**:
 - Block size: 512
 - Top-k: 3 (sparsité ~81.25% à 8K, ~95.31% à 32K)

3.4. Phase 4: Fine-tuning (optionnel)

- Fine-tuning contrastif sur MTEB pendant 2000 steps
- Stratégie de fine-tuning avec layer-wise hybrid:
 - Conserver MOBA pour le prefill
 - Utiliser l'attention complète pendant la génération

4. Protocole d'évaluation

```
### 4.1. Evaluation des performances générales
- **GLUE**: Évaluation standard avec grid search (batch sizes  $\in \{16, 32\}$ , learning rates  $\in \{1e-5, 2e-5, 3e-5\}$ )
- **MTEB**: Évaluation sur les 56 tâches de l'anglais (classification, clustering, retrieval, etc.)

### 4.2. Évaluation des capacités de contexte long
- **Position-wise LM loss**: Mesurer la perte à différentes positions (0-2K, 2K-4K, ..., 30K-32K)
- **Trailing token loss**: Perte sur les 2K derniers tokens des séquences longues
- **"Needle in a Haystack"**: Évaluation de la capacité à retrouver de l'information dans un long contexte
- **RULER @128K**: Évaluation de la compréhension du raisonnement à longue distance

### 4.3. Évaluation d'efficacité
- **Mémoire GPU**: Consommation maximale durant l'entraînement et l'inférence
- **Temps de calcul**: Comparaison des temps de forward pass pour différentes longueurs (8K, 32K, 128K, 1M)
- **Speedup ratio**: Gain de vitesse MOBA vs attention complète
```

5. Expériences d'ablation

```
### 5.1. Impact de la granularité des blocs
Entraîner des versions avec différentes configurations de blocs en maintenant une sparsité constante:
- 8 blocs, top-k=2
- 16 blocs, top-k=4
- 32 blocs, top-k=8
- 64 blocs, top-k=16
- 128 blocs, top-k=32

### 5.2. Impact de la stratégie hybride
Comparer les performances de:
- MOBA complet (toutes les couches)
- Hybride avec 1, 3, 5, 10 couches d'attention complète
- Attention complète

### 5.3. Impact du choix des métriques pour le routage
Comparer différentes méthodes de calcul des scores d'affinité:
- Mean pooling (standard)
- Max pooling
- Combinaison min/max pooling
- Attention basée sur un token représentatif
```

```
## 6. Suite du développement

Suite aux résultats des évaluations, prévoir:
- Ajustement des hyperparamètres MOBA
- Optimisation de l'implémentation pour différents matériels
- Extension à des contextes encore plus longs (>1M tokens)
- Adaptation à d'autres langues
```

Ce protocole permettra d'évaluer rigoureusement l'efficacité et les performances de l'architecture NeoBERT-MOBA, en particulier sa capacité à traiter efficacement des contextes très longs tout en maintenant d'excellentes performances.

Sources: [2502.19587v1.pdf](#) [2502.13189v1.pdf](#)

USER

Ecrit le code pour Préparation des données

BOT

Thoughts

Je vais écrire un code complet pour la préparation des données pour entraîner le modèle NeoBERT-MOBA. Je vais couvrir:

- 1. Téléchargement et préparation des corpus RefinedWeb (simulation, car le corpus complet est très grand)
- 2. Prétraitement des données avec tokenisation WordPiece
- 3. Filtrage des données pour créer des sous-ensembles spécifiques (RefinedWeb1024+, RefinedWeb2048+)
- 4. Construction de DataLoaders pour l'entraînement avec masquage MLM
- 5. Échantillonnage stratégique pour les différentes phases d'entraînement

Je vais utiliser les bibliothèques standard comme Hugging Face Datasets, Tokenizers, et PyTorch pour cette implémentation.

Préparation des données pour l'entraînement de NeoBERT-MOBA

Voici un code complet pour la préparation des données, couvrant le téléchargement, le prétraitement, la tokenisation et la création des dataloaders pour l'entraînement du modèle NeoBERT-MOBA.

```
import os
import json
import random
import logging
import numpy as np
import torch
from torch.utils.data import Dataset, DataLoader, DistributedSampler
from datasets import load_dataset, concatenate_datasets
from tokenizers import Tokenizer, models, normalizers, pre_tokenizers, decoders, trainers, processors
from transformers import PreTrainedTokenizerFast
from typing import Dict, List, Optional, Tuple, Union
import multiprocessing
from tqdm import tqdm

# Configuration du logging
logging.basicConfig(
    format="%(asctime)s - %(levelname)s - %(message)s",
    datefmt="%m/%d/%Y %H:%M:%S",
```

```

    level=logging.INFO,
)
logger = logging.getLogger(__name__)

class DataPreparation:
    def __init__(
        self,
        base_dir: str = "./data",
        vocab_size: int = 30000,
        max_length: int = 32768,
        mlm_probability: float = 0.15,
        seed: int = 42
    ):
        self.base_dir = base_dir
        self.vocab_size = vocab_size
        self.max_length = max_length
        self.mlm_probability = mlm_probability
        self.seed = seed
        self.tokenizer = None

        # Créer le répertoire de base s'il n'existe pas
        os.makedirs(base_dir, exist_ok=True)

        # Fixer la seed pour la reproductibilité
        random.seed(seed)
        np.random.seed(seed)
        torch.manual_seed(seed)

    def download_data(self):
        """
        Télécharge les corpus principaux pour l'entraînement.
        Pour un usage réel, vous auriez besoin d'accéder à RefinedWeb.
        Cet exemple utilise des datasets publics plus petits.
        """
        logger.info("Téléchargement des données...")

        # Pour simuler RefinedWeb, nous utilisons quelques datasets publics
        datasets = {}

        # Bookcorpus + Wikipedia (pour simuler un corpus général)
        datasets["main"] = load_dataset("bookcorpus", split="train")
        wiki_dataset = load_dataset("wikipedia", "20220301.en", split="train")

        # Filtrer wiki pour ne garder que le texte
        wiki_dataset = wiki_dataset.map(
            lambda x: {"text": x["text"]},
            remove_columns=[col for col in wiki_dataset.column_names if col != "text"]
        )

        # Concatenate datasets
        datasets["main"] = concatenate_datasets([datasets["main"], wiki_dataset])

        # Pour les contextes plus longs, nous pouvons utiliser PG19
        pg19 = load_dataset("pg19", split="train")

        # Sauvegarder les données
        os.makedirs(os.path.join(self.base_dir, "raw"), exist_ok=True)

        logger.info(f"Sauvegarde du corpus principal... ({len(datasets['main'])} exemples)")
        datasets["main"].save_to_disk(os.path.join(self.base_dir, "raw", "main"))

        logger.info(f"Sauvegarde du corpus PG19... ({len(pg19)} exemples)")
        pg19.save_to_disk(os.path.join(self.base_dir, "raw", "pg19"))

        logger.info("Téléchargement des données terminé!")

        return datasets

    def train_tokenizer(self, dataset, vocab_size: int = None):
        """
        Entraîne un tokenizer WordPiece sur le dataset fourni
        """
        vocab_size = vocab_size or self.vocab_size
        logger.info(f"Entraînement du tokenizer avec une taille de vocabulaire de {vocab_size}...")

        # Créer le dossier pour le tokenizer
        tokenizer_dir = os.path.join(self.base_dir, "tokenizer")
        os.makedirs(tokenizer_dir, exist_ok=True)

        # Initialiser le tokenizer (WordPiece comme BERT original)
        tokenizer = Tokenizer(models.WordPiece(unk_token="[UNK]"))

        # Normaliser le texte
        tokenizer.normalizer = normalizers.Sequence([
            normalizers.NFD(),
            normalizers.Lowercase(),
            normalizers.StripAccents(),
        ])

        # Pre-tokenisation
        tokenizer.pre_tokenizer = pre_tokenizers.Whitespace()

        # Configurer l'entraîneur
        trainer = trainers.WordPieceTrainer(

```



```

vocab_size=vocab_size,
special_tokens=["[UNK]", "[CLS]", "[SEP]", "[PAD]", "[MASK]"],
)

# Fonction pour générer les textes pour l'entraînement du tokenizer
def batch_iterator(batch_size=1000):
    for i in range(0, len(dataset), batch_size):
        yield dataset[i:i+batch_size]["text"]

# Entraîner le tokenizer
tokenizer.train_from_iterator(batch_iterator(), trainer=trainer, length=len(dataset))

# Ajouter le post-processing pour gérer les paires de séquences (comme BERT)
tokenizer.post_processor = processors.TemplateProcessing(
    single="[CLS] $A [SEP]",
    pair="[CLS] $A [SEP] $B [SEP]",
    special_tokens=[
        ("[CLS]", tokenizer.token_to_id("[CLS]")),
        ("[SEP]", tokenizer.token_to_id("[SEP]")),
    ],
)

# Ajouter le décodeur
tokenizer.decoder = decoders.WordPiece()

# Sauvegarder le tokenizer
tokenizer_path = os.path.join(tokenizer_dir, "wordpiece_tokenizer.json")
tokenizer.save(tokenizer_path)
logger.info(f"Tokenizer sauvegardé: {tokenizer_path}")

# Créer un tokenizer compatible avec transformers
self.tokenizer = PreTrainedTokenizerFast(
    tokenizer_object=tokenizer,
    unk_token="[UNK]",
    cls_token="[CLS]",
    sep_token="[SEP]",
    pad_token="[PAD]",
    mask_token="[MASK]",
)

# Sauvegarder le tokenizer sous format transformers
self.tokenizer.save_pretrained(tokenizer_dir)
logger.info(f"Tokenizer HF sauvegardé: {tokenizer_dir}")

return self.tokenizer

def tokenize_dataset(self, dataset, max_length=None, tokenizer=None):
    """
    Tokenise le dataset pour l'entraînement
    """
    max_length = max_length or self.max_length
    tokenizer = tokenizer or self.tokenizer

    if tokenizer is None:
        raise ValueError("Le tokenizer n'a pas été initialisé")

    logger.info(f"Tokenisation du dataset ({len(dataset)} exemples)...")

    # Fonction de tokenisation pour map
    def tokenize_function(examples):
        return tokenizer(
            examples["text"],
            truncation=True,
            max_length=max_length,
            return_special_tokens_mask=True,
        )

    # Tokeniser le dataset
    tokenized_dataset = dataset.map(
        tokenize_function,
        batched=True,
        num_proc=multiprocessing.cpu_count(),
        remove_columns=["text"],
        desc="Tokenisation du dataset",
    )

    return tokenized_dataset

def create_sequence_datasets(self, tokenized_dataset, lengths=[1024, 2048, 4096, 8192]):
    """
    Crée des datasets filtrés par longueur de séquence
    """
    logger.info("Création des datasets filtrés par longueur...")

    sequence_datasets = {}

    for min_length in lengths:
        # Filtrer par longueur
        filtered = tokenized_dataset.filter(
            lambda x: len(x["input_ids"]) >= min_length,
            num_proc=multiprocessing.cpu_count(),
            desc=f"Filtrage {min_length}+",
        )

```

```

# Sauvegarder le dataset filtré
output_dir = os.path.join(self.base_dir, "processed", f"seq_{min_length}")
os.makedirs(output_dir, exist_ok=True)
filtered.save_to_disk(output_dir)

sequence_datasets[f"seq_{min_length}"] = filtered

logger.info(f"Dataset seq_{min_length}: {len(filtered)} exemples")

return sequence_datasets

def create_mlm_dataset(self, tokenized_dataset):
    """
    Préparation finale des données pour l'entraînement MLM
    """
    class MLMDataset(Dataset):
        def __init__(self, tokenized_dataset, tokenizer, mlm_probability=0.15, max_length=512):
            self.tokenized_dataset = tokenized_dataset
            self.tokenizer = tokenizer
            self.mlm_probability = mlm_probability
            self.max_length = max_length

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

        def __getitem__(self, idx):
            # Récupérer un exemple
            item = self.tokenized_dataset[idx]

            # Tronquer si nécessaire
            input_ids = item["input_ids"]
            if len(input_ids) > self.max_length:
                start = random.randint(0, len(input_ids) - self.max_length)
                input_ids = input_ids[start:start + self.max_length]

            # Créer le tenseur d'attention
            attention_mask = [1] * len(input_ids)

            # Gérer le padding
            padding_length = self.max_length - len(input_ids)
            if padding_length > 0:
                input_ids = input_ids + [self.tokenizer.pad_token_id] * padding_length
                attention_mask = attention_mask + [0] * padding_length

            # Créer des labels pour MLM
            labels = input_ids.copy()

            # Appliquer le masquage MLM
            # On ne masque pas [CLS], [SEP], [PAD]
            special_tokens_mask = item.get("special_tokens_mask", [0] * len(input_ids))
            probability_matrix = torch.full((len(input_ids),), self.mlm_probability)
            probability_matrix.masked_fill_(torch.tensor(special_tokens_mask, dtype=torch.bool), value=0.0)

            masked_indices = torch.bernoulli(probability_matrix).bool()
            labels = torch.tensor(labels)
            labels[~masked_indices] = -100 # We only compute loss on masked tokens

            # 80% des tokens masqués sont remplacés par [MASK]
            indices_replaced = torch.bernoulli(torch.full(labels.shape, 0.8)).bool() & masked_indices
            input_ids = torch.tensor(input_ids)
            input_ids[indices_replaced] = self.tokenizer.convert_tokens_to_ids(self.tokenizer.mask_token)

            # 10% des tokens masqués sont remplacés par un token aléatoire
            indices_random = torch.bernoulli(torch.full(labels.shape, 0.5)).bool() & masked_indices & ~indices_replaced
            random_words = torch.randint(len(self.tokenizer), labels.shape, dtype=torch.long)
            input_ids[indices_random] = random_words[indices_random]

            return {
                "input_ids": input_ids.long(),
                "attention_mask": torch.tensor(attention_mask).long(),
                "labels": labels.long(),
            }

    return MLMDataset(tokenized_dataset, self.tokenizer, self.mlm_probability, self.max_length)

def create_data loaders(self, datasets, batch_sizes, max_lengths, ddp=False):
    """
    Crée les data loaders pour l'entraînement avec différentes configurations
    """
    data loaders = {}

    for name, dataset in datasets.items():
        for max_length in max_lengths:
            for batch_size in batch_sizes:
                # Créer le dataset MLM
                self.max_length = max_length
                mlm_dataset = self.create_mlm_dataset(dataset)

                # Créer le sampler pour DDP si nécessaire
                sampler = None
                if ddp:
                    sampler = DistributedSampler(mlm_dataset)

                # Créer le data loader

```

```

        dataloader = DataLoader(
            mlm_dataset,
            batch_size=batch_size,
            shuffle=(sampler is None),
            sampler=sampler,
            num_workers=4,
            pin_memory=True,
        )

        key = f"{name}_len{max_length}_bs{batch_size}"
        dataloaders[key] = dataloader

        logger.info(f"Dataloader créé: {key} ({len(dataloader)} batches)")

    return dataloaders

def mixing_strategy_dataloader(self, datasets, max_length, batch_size,
                               mixing_weights=None, ddp=False):
    """
    Crée un dataloader qui mélange plusieurs datasets selon une stratégie spécifique
    """
    if mixing_weights is None:
        mixing_weights = {name: 1.0 / len(datasets) for name in datasets.keys()}

    # Normaliser les poids
    total = sum(mixing_weights.values())
    mixing_weights = {k: v / total for k, v in mixing_weights.items()}

    # Créer les datasets MLM
    mlm_datasets = {}
    for name, dataset in datasets.items():
        self.max_length = max_length
        mlm_datasets[name] = self.create_mlm_dataset(dataset)

    # Créer un dataloader mixte
    class MixedDataLoader:
        def __init__(self, dataloaders, mixing_weights):
            self.dataloaders = dataloaders
            self.mixing_weights = mixing_weights
            self.iterators = {name: iter(dl) for name, dl in self.dataloaders.items()}
            self.lengths = {name: len(dl) for name, dl in self.dataloaders.items()}
            self.total_steps = sum(
                int(length * weight) for name, (length, weight) in
                zip(self.dataloaders.keys(), zip(self.lengths.values(), self.mixing_weights.values()))
            )

        def __len__(self):
            return self.total_steps

        def __iter__(self):
            self.iterators = {name: iter(dl) for name, dl in self.dataloaders.items()}
            for _ in range(self.total_steps):
                # Choisir un dataset selon les poids
                dataset_name = random.choices(
                    list(self.mixing_weights.keys()),
                    weights=list(self.mixing_weights.values()),
                    k=1
                )[0]

                # Récupérer un batch
                try:
                    batch = next(self.iterators[dataset_name])
                except StopIteration:
                    # Réinitialiser l'iterator si nécessaire
                    self.iterators[dataset_name] = iter(self.dataloaders[dataset_name])
                    batch = next(self.iterators[dataset_name])

                yield batch

    # Créer les dataloaders individuels
    dataloaders = {}
    for name, mlm_dataset in mlm_datasets.items():
        # Créer le sampler pour DDP si nécessaire
        sampler = None
        if ddp:
            sampler = DistributedSampler(mlm_dataset)

        # Créer le dataloader
        dataloaders[name] = DataLoader(
            mlm_dataset,
            batch_size=batch_size,
            shuffle=(sampler is None),
            sampler=sampler,
            num_workers=4,
            pin_memory=True,
        )

    # Créer et retourner le dataloader mixte
    mixed_dataloader = MixedDataLoader(dataloaders, mixing_weights)

    return mixed_dataloader

def get_dataloaders_for_phase(self, phase, ddp=False):
    """

```

Retourne les dataloaders spécifiques pour chaque phase d'entraînement

```
"""
if phase == 1:
    # Phase 1: Pré-entraînement initial (1024 tokens)
    main_dataset = load_dataset("json", data_files=os.path.join(self.base_dir, "processed", "main"))
    return self.create_dataloaders(
        {"main": main_dataset},
        batch_sizes=[32],
        max_lengths=[1024],
        ddp=ddp
    )

elif phase == 2:
    # Phase 2: Extension du contexte (4096 tokens)
    datasets = {
        "main": load_dataset("json", data_files=os.path.join(self.base_dir, "processed", "main")),
        "seq_1024": load_dataset("json", data_files=os.path.join(self.base_dir, "processed", "seq_1024")),
        "seq_2048": load_dataset("json", data_files=os.path.join(self.base_dir, "processed", "seq_2048")),
    }

    return self.mixing_strategy_dataloader(
        datasets,
        max_length=4096,
        batch_size=4,
        mixing_weights={"main": 0.2, "seq_1024": 0.4, "seq_2048": 0.4},
        ddp=ddp
    )

elif phase == 3:
    # Phase 3: Activation MOBA (jusqu'à 32K tokens)
    datasets = {
        "main": load_dataset("json", data_files=os.path.join(self.base_dir, "processed", "main")),
        "seq_1024": load_dataset("json", data_files=os.path.join(self.base_dir, "processed", "seq_1024")),
        "seq_2048": load_dataset("json", data_files=os.path.join(self.base_dir, "processed", "seq_2048")),
        "seq_4096": load_dataset("json", data_files=os.path.join(self.base_dir, "processed", "seq_4096")),
        "seq_8192": load_dataset("json", data_files=os.path.join(self.base_dir, "processed", "seq_8192")),
    }

    return self.mixing_strategy_dataloader(
        datasets,
        max_length=32768,
        batch_size=1,
        mixing_weights={"main": 0.1, "seq_1024": 0.2, "seq_2048": 0.2, "seq_4096": 0.25, "seq_8192": 0.25},
        ddp=ddp
    )

else:
    raise ValueError(f"Phase d'entraînement non reconnue: {phase}")

def process_complete_pipeline():
    """
    Exécute le pipeline complet de préparation des données
    """
    # Initialisation
    data_prep = DataPreparation(
        base_dir="./data",
        vocab_size=30000,
        max_length=32768,
        mlm_probability=0.15
    )

    # Téléchargement des données
    datasets = data_prep.download_data()

    # Entraînement du tokenizer
    tokenizer = data_prep.train_tokenizer(datasets["main"])

    # Tokenisation du dataset principal
    tokenized_main = data_prep.tokenize_dataset(datasets["main"])

    # Création des datasets filtrés par longueur
    sequence_datasets = data_prep.create_sequence_datasets(
        tokenized_main,
        lengths=[1024, 2048, 4096, 8192]
    )

    # Sauvegarde
    os.makedirs(os.path.join(data_prep.base_dir, "processed"), exist_ok=True)
    tokenized_main.save_to_disk(os.path.join(data_prep.base_dir, "processed", "main"))

    # Création des dataloaders pour chaque phase
    phase1_loaders = data_prep.get_dataloaders_for_phase(1)

    logger.info("Pipeline de préparation des données terminé!")
    logger.info(f"Datasets disponibles: {list(sequence_datasets.keys())}")
    logger.info(f"Dataloaders pour la phase 1: {list(phase1_loaders.keys())}")

    return {
        "tokenizer": tokenizer,
        "datasets": sequence_datasets,
        "dataloaders": {
            "phase1": phase1_loaders,
            "phase2": data_prep.get_dataloaders_for_phase(2),
        }
    }
```

```
        "phase3": data_prep.get_dataloaders_for_phase(3)
    }
}

if __name__ == "__main__":
    results = process_complete_pipeline()
    print("Préparation des données terminée!")
```

Explication du code

Le code ci-dessus fournit un pipeline complet pour la préparation des données pour l'entraînement du modèle NeoBERT-MOBA, avec les fonctionnalités suivantes:

1. Téléchargement et préparation des données

- Utilise des datasets publics (BookCorpus, Wikipedia, PG19) comme substituts pour RefinedWeb
- Structure les données en formats appropriés pour l'entraînement

2. Entraînement du tokenizer WordPiece

- Implémente un tokenizer WordPiece avec 30K tokens, similaire à BERT
- Inclut les tokens spéciaux comme [CLS], [SEP], [MASK], etc.
- Sauvegarde le tokenizer au format Hugging Face

3. Prétraitement et tokenisation des données

- Tokenise l'ensemble du corpus
- Crée des sous-ensembles filtrés par longueur (1024+, 2048+, etc.)
- Gère efficacement les séquences longues

4. Création de datasets MLM

- Implémente un masquage MLM avec 15% de tokens masqués
- Respecte la stratégie 80/10/10 (80% [MASK], 10% random, 10% unchanged)
- Gère le padding et les masques d'attention

5. Stratégie d'échantillonnage mixte

- Crée un dataloader mixte qui combine plusieurs datasets
- Permet un échantillonnage pondéré entre différentes sources
- Adapté à chaque phase d'entraînement

6. Configuration spécifique par phase

- Phase 1: Pré-entraînement initial avec séquences courtes
- Phase 2: Extension du contexte avec un mix de longueurs
- Phase 3: Entraînement longue séquence avec MOBA

7. Support DDP (Distributed Data Parallel)

- Compatible avec l'entraînement distribué
- Utilise DistributedSampler pour la parallélisation

Ce code constitue une base solide pour préparer les données nécessaires à l'entraînement du modèle NeoBERT-MOBA, spécialement optimisé pour gérer efficacement les longues séquences.

USER

Ecrit le code pour l'étape 2 Architecture du modèle

BOT

Thoughts

Dans cette partie, je dois créer le code pour l'étape 2 du protocole d'entraînement et de test pour l'architecture NeoBERT-MOBA. Cette section correspond à l'implémentation du modèle NeoBERT-MOBA en utilisant le mécanisme d'attention MOBA pour gérer efficacement les contextes longs.

Selon le protocole fourni, l'étape 2 concerne "Architecture du modèle" avec les configurations suivantes:

2.1. Configuration NeoBERT-MOBA

```
model_config = {
    # Configuration NeoBERT
    "vocab_size": 30000,
    "hidden_size": 768,
    "num_hidden_layers": 28,
    "num_attention_heads": 12,
    "max_position_embeddings": 32768,
    "activation_function": "swiglu",
    "layer_norm_type": "rms_norm",

    # Configuration MOBA
    "moba_block_size": 512, # Taille des blocs
    "moba_top_k": 3,        # Nombre de blocs à sélectionner
    "hybrid_layer_count": 3, # Dernières couches en attention complète
}
```

2.2. Structure hybride

- Les `num_hidden_layers` - `hybrid_layer_count` premières couches utilisent MOBA
- Les `hybrid_layer_count` dernières couches utilisent l'attention standard
- Chaque token est toujours routé vers son bloc courant avec masquage causal

Je vais implémenter cette architecture en Python en combinant les éléments du modèle NeoBERT que nous avons précédemment implémenté avec le mécanisme MOBA décrit dans l'article que l'utilisateur a partagé.

Le code devra inclure:

1. L'implémentation de l'architecture de base NeoBERT (couches RMSNorm, activation SwiGLU, etc.)
2. L'implémentation du mécanisme d'attention MOBA (division en blocs, sélection des top-k blocs)
3. L'implémentation de la structure hybride (MOBA pour certaines couches, attention complète pour d'autres)
4. Des fonctions pour créer et configurer le modèle selon les paramètres fournis

Je vais me baser sur le code précédent de NeoBERT et y intégrer le mécanisme MOBA, en m'inspirant de la description fournie dans le PDF sur MOBA.

Code d'implémentation de l'architecture NeoBERT-MOBA

Voici l'implémentation complète de l'architecture NeoBERT-MOBA, qui intègre le mécanisme d'attention MOBA pour gérer efficacement les contextes longs:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import math
from typing import Optional, Tuple, List, Dict, Any, Union
import numpy as np

class RMSNorm(nn.Module):
    """Root Mean Square Layer Normalization (RMSNorm)"""
    def __init__(self, dim: int, eps: float = 1e-6):
        super().__init__()
        self.eps = eps
        self.weight = nn.Parameter(torch.ones(dim))

    def forward(self, x):
        # x: [batch_size, seq_len, dim]
        variance = x.pow(2).mean(-1, keepdim=True)
        x = x * torch.rsqrt(variance + self.eps)
        return x * self.weight

class RotaryEmbedding(nn.Module):
    """Rotary Position Embeddings (RoPE)"""
    def __init__(self, dim: int, base: int = 10000):
        super().__init__()
        self.dim = dim
        self.base = base
        self.inv_freq = 1.0 / (base ** (torch.arange(0, dim, 2).float() / dim))
        self.max_seq_len_cached = None
        self.cos_cached = None
        self.sin_cached = None

    def forward(self, x, seq_len: int):
        # x: [batch_size, n_heads, seq_len, head_dim]
        if seq_len > self.max_seq_len_cached or self.cos_cached is None or self.sin_cached is None:
            self._update_cache(seq_len)

        cos = self.cos_cached[:seq_len].to(x.device)
        sin = self.sin_cached[:seq_len].to(x.device)
        return (x * cos) + (self._rotate_half(x) * sin)

    def _update_cache(self, seq_len: int):
        self.max_seq_len_cached = seq_len
        t = torch.arange(seq_len, device=self.inv_freq.device).type_as(self.inv_freq)
        freqs = torch.einsum('i,j->ij', t, self.inv_freq)
        emb = torch.cat((freqs, freqs), dim=-1)
        self.cos_cached = emb.cos()[None, None, :, :]
        self.sin_cached = emb.sin()[None, None, :, :]

    def _rotate_half(self, x):
        x1, x2 = x[..., : self.dim // 2], x[..., self.dim // 2 :]
        return torch.cat((-x2, x1), dim=-1)

class SwiGLU(nn.Module):
    """SwiGLU activation function"""
    def __init__(self, dim: int, hidden_dim: int = None):
        super().__init__()
        hidden_dim = hidden_dim or int(2/3 * 4 * dim) # 8/3 * dim as in NeoBERT paper

        self.w1 = nn.Linear(dim, hidden_dim, bias=False)
        self.w2 = nn.Linear(dim, hidden_dim, bias=False)
        self.w3 = nn.Linear(hidden_dim, dim, bias=False)

    def forward(self, x):
        gate = F.silu(self.w1(x))
        value = self.w2(x)
        return self.w3(gate * value)

class MoBAAttention(nn.Module):
    """Mixture of Block Attention (MoBA)"""
    def __init__(
        self,
        hidden_size: int,
        num_heads: int,
        block_size: int = 512,
        top_k: int = 3,
        rotary_dim: int = None,
    ):
        super().__init__()
        self.hidden_size = hidden_size
```

```

self.num_heads = num_heads
self.head_dim = hidden_size // num_heads
self.block_size = block_size
self.top_k = top_k

self.q_proj = nn.Linear(hidden_size, hidden_size, bias=False)
self.k_proj = nn.Linear(hidden_size, hidden_size, bias=False)
self.v_proj = nn.Linear(hidden_size, hidden_size, bias=False)
self.o_proj = nn.Linear(hidden_size, hidden_size, bias=False)

self.rotary = RotaryEmbedding(self.head_dim)

self.scale = 1.0 / math.sqrt(self.head_dim)

def _split_heads(self, x):
    # x: [batch_size, seq_len, hidden_size]
    new_shape = x.size()[:-1] + (self.num_heads, self.head_dim)
    x = x.view(*new_shape) # [batch_size, seq_len, num_heads, head_dim]
    return x.permute(0, 2, 1, 3) # [batch_size, num_heads, seq_len, head_dim]

def _merge_heads(self, x):
    # x: [batch_size, num_heads, seq_len, head_dim]
    x = x.permute(0, 2, 1, 3) # [batch_size, seq_len, num_heads, head_dim]
    new_shape = x.size()[:-2] + (self.hidden_size,)
    return x.reshape(*new_shape) # [batch_size, seq_len, hidden_size]

def forward(self, hidden_states, attention_mask=None, past_key_value=None):
    batch_size, seq_len = hidden_states.shape[:2]

    # Linear projections
    q = self.q_proj(hidden_states)
    k = self.k_proj(hidden_states)
    v = self.v_proj(hidden_states)

    # Split into heads
    q = self._split_heads(q) # [batch_size, num_heads, seq_len, head_dim]
    k = self._split_heads(k) # [batch_size, num_heads, seq_len, head_dim]
    v = self._split_heads(v) # [batch_size, num_heads, seq_len, head_dim]

    # Apply RoPE to q and k
    q = self.rotary(q, seq_len)
    k = self.rotary(k, seq_len)

    # Split sequence into blocks
    num_blocks = math.ceil(seq_len / self.block_size)

    # Compute mean representation for each block
    block_means = []
    for i in range(num_blocks):
        start_idx = i * self.block_size
        end_idx = min(start_idx + self.block_size, seq_len)
        if end_idx > start_idx: # Ensure non-empty block
            # Mean pooling of keys in this block for all heads
            # Shape: [batch_size, num_heads, 1, head_dim]
            block_mean = k[:, :, start_idx:end_idx].mean(dim=2, keepdim=True)
            block_means.append(block_mean)

    # Stack block means
    # Shape: [batch_size, num_heads, num_blocks, head_dim]
    block_means = torch.cat(block_means, dim=2)

    # Initialize output tensor
    attn_output = torch.zeros_like(q)

    # Process each query token
    for i in range(seq_len):
        # Get the current query
        # Shape: [batch_size, num_heads, 1, head_dim]
        query = q[:, :, i:i+1]

        # Compute relevance scores between query and block means
        # Shape: [batch_size, num_heads, 1, num_blocks]
        block_scores = torch.matmul(query, block_means.transpose(-1, -2)) * self.scale

        # Apply causal mask - query can only attend to blocks up to its position
        current_block = i // self.block_size
        causal_mask = torch.arange(num_blocks, device=query.device) > current_block
        block_scores = block_scores.masked_fill(
            causal_mask.view(1, 1, -1), float('-inf'))
    )

    # Always include the current block
    # Set a large positive value to ensure selection
    block_scores[:, :, :, current_block] = torch.finfo(block_scores.dtype).max

    # Get top-k blocks (always including current block)
    # Shape: [batch_size, num_heads, 1, top_k]
    _, top_k_indices = torch.topk(block_scores, min(self.top_k, num_blocks), dim=-1)

    # Get keys and values for selected blocks
    selected_k = []
    selected_v = []

    for b in range(batch_size):

```

```

for h in range(self.num_heads):
    k_for_query = []
    v_for_query = []

    for block_idx in top_k_indices[b, h, 0]:
        block_idx = block_idx.item()
        start_idx = block_idx * self.block_size
        end_idx = min(start_idx + self.block_size, seq_len)

        # For the current block, apply causal masking
        if block_idx == current_block:
            k_block = k[b, h:h+1, start_idx:i+1] # Up to current position
            v_block = v[b, h:h+1, start_idx:i+1]
        else:
            k_block = k[b, h:h+1, start_idx:end_idx]
            v_block = v[b, h:h+1, start_idx:end_idx]

        if k_block.size(2) > 0: # Ensure non-empty block
            k_for_query.append(k_block)
            v_for_query.append(v_block)

    if k_for_query:
        # Concatenate selected keys and values
        k_concat = torch.cat(k_for_query, dim=2) # [1, seq_len_selected, head_dim]
        v_concat = torch.cat(v_for_query, dim=2) # [1, seq_len_selected, head_dim]

        # Compute attention scores
        attn_scores = torch.matmul(query[b:b+1, h:h+1], k_concat.transpose(-1, -2)) * self.scale
        attn_probs = F.softmax(attn_scores, dim=-1)

        # Apply attention
        attn_output[b:b+1, h:h+1, i:i+1] = torch.matmul(attn_probs, v_concat)

# Merge heads and project back
attn_output = self._merge_heads(attn_output)
attn_output = self.o_proj(attn_output)

return attn_output

```

```

class FullAttention(nn.Module):
    """Standard Self-Attention with RoPE"""
    def __init__(self, hidden_size: int, num_heads: int):
        super().__init__()
        self.hidden_size = hidden_size
        self.num_heads = num_heads
        self.head_dim = hidden_size // num_heads

        self.q_proj = nn.Linear(hidden_size, hidden_size, bias=False)
        self.k_proj = nn.Linear(hidden_size, hidden_size, bias=False)
        self.v_proj = nn.Linear(hidden_size, hidden_size, bias=False)
        self.o_proj = nn.Linear(hidden_size, hidden_size, bias=False)

        self.rotary = RotaryEmbedding(self.head_dim)

        self.scale = 1.0 / math.sqrt(self.head_dim)

    def _split_heads(self, x):
        # x: [batch_size, seq_len, hidden_size]
        new_shape = x.size()[:-1] + (self.num_heads, self.head_dim)
        x = x.view(*new_shape) # [batch_size, seq_len, num_heads, head_dim]
        return x.permute(0, 2, 1, 3) # [batch_size, num_heads, seq_len, head_dim]

    def _merge_heads(self, x):
        # x: [batch_size, num_heads, seq_len, head_dim]
        x = x.permute(0, 2, 1, 3) # [batch_size, seq_len, num_heads, head_dim]
        new_shape = x.size()[:-2] + (self.hidden_size,)
        return x.reshape(*new_shape) # [batch_size, seq_len, hidden_size]

    def forward(self, hidden_states, attention_mask=None, past_key_value=None):
        batch_size, seq_len = hidden_states.shape[:2]

        # Linear projections
        q = self.q_proj(hidden_states)
        k = self.k_proj(hidden_states)
        v = self.v_proj(hidden_states)

        # Split into heads
        q = self._split_heads(q) # [batch_size, num_heads, seq_len, head_dim]
        k = self._split_heads(k) # [batch_size, num_heads, seq_len, head_dim]
        v = self._split_heads(v) # [batch_size, num_heads, seq_len, head_dim]

        # Apply RoPE to q and k
        q = self.rotary(q, seq_len)
        k = self.rotary(k, seq_len)

        # Compute attention scores
        attention_scores = torch.matmul(q, k.transpose(-1, -2)) * self.scale

        # Apply causal mask
        causal_mask = torch.triu(
            torch.ones((seq_len, seq_len), dtype=torch.bool, device=q.device),
            diagonal=1
        )

```



```

        attention_scores = attention_scores.masked_fill(
            causal_mask.unsqueeze(0).unsqueeze(0), float('-inf'))
    )

    # Apply attention mask if provided
    if attention_mask is not None:
        attention_scores = attention_scores + attention_mask

    # Calculate attention probabilities
    attention_probs = F.softmax(attention_scores, dim=-1)

    # Apply attention
    context = torch.matmul(attention_probs, v)

    # Merge heads and project back
    context = self._merge_heads(context)
    attn_output = self.o_proj(context)

    return attn_output

class NeoBERTMoBALayer(nn.Module):
    """NeoBERT Layer with MoBA or Full Attention"""
    def __init__(
        self,
        hidden_size: int,
        num_heads: int,
        use_moba: bool = True,
        moba_block_size: int = 512,
        moba_top_k: int = 3
    ):
        super().__init__()
        self.use_moba = use_moba

        self.norm1 = RMSNorm(hidden_size)

        if use_moba:
            self.attention = MoBAAttention(
                hidden_size=hidden_size,
                num_heads=num_heads,
                block_size=moba_block_size,
                top_k=moba_top_k
            )
        else:
            self.attention = FullAttention(
                hidden_size=hidden_size,
                num_heads=num_heads
            )

        self.norm2 = RMSNorm(hidden_size)
        self.mlp = SwiGLU(hidden_size)

    def forward(self, hidden_states, attention_mask=None):
        # Pre-LN for attention
        attn_input = self.norm1(hidden_states)
        attn_output = self.attention(attn_input, attention_mask)
        hidden_states = hidden_states + attn_output

        # Pre-LN for FFN
        mlp_input = self.norm2(hidden_states)
        mlp_output = self.mlp(mlp_input)
        hidden_states = hidden_states + mlp_output

        return hidden_states

class NeoBERTEmbeddings(nn.Module):
    """Embeddings for NeoBERT"""
    def __init__(self, vocab_size: int, hidden_size: int):
        super().__init__()
        self.word_embeddings = nn.Embedding(vocab_size, hidden_size)
        self.norm = RMSNorm(hidden_size)
        self.dropout = nn.Dropout(0.1)

    def forward(self, input_ids):
        embeddings = self.word_embeddings(input_ids)
        embeddings = self.norm(embeddings)
        return self.dropout(embeddings)

class NeoBERTMoBA(nn.Module):
    """NeoBERT model with MOBA attention"""
    def __init__(
        self,
        vocab_size: int = 30000,
        hidden_size: int = 768,
        num_hidden_layers: int = 28,
        num_attention_heads: int = 12,
        max_position_embeddings: int = 32768,
        moba_block_size: int = 512,
        moba_top_k: int = 3,
        hybrid_layer_count: int = 3,
    ):
        super().__init__()

```

```

self.config = {
    "vocab_size": vocab_size,
    "hidden_size": hidden_size,
    "num_hidden_layers": num_hidden_layers,
    "num_attention_heads": num_attention_heads,
    "max_position_embeddings": max_position_embeddings,
    "moba_block_size": moba_block_size,
    "moba_top_k": moba_top_k,
    "hybrid_layer_count": hybrid_layer_count,
}

self.embeddings = NeoBERTEmbeddings(vocab_size, hidden_size)

# Create layers with hybrid architecture
# Last few layers use full attention, the rest use MoBA
self.layers = nn.ModuleList()
for i in range(num_hidden_layers):
    # Use MoBA for early layers, full attention for last hybrid_layer_count layers
    use_moba = i < (num_hidden_layers - hybrid_layer_count)
    self.layers.append(
        NeoBERTMoBALayer(
            hidden_size=hidden_size,
            num_heads=num_attention_heads,
            use_moba=use_moba,
            moba_block_size=moba_block_size,
            moba_top_k=moba_top_k
        )
    )

self.final_norm = RMSNorm(hidden_size)

# MLM head
self.mlm_dense = nn.Linear(hidden_size, hidden_size)
self.mlm_norm = RMSNorm(hidden_size)
self.mlm_head = nn.Linear(hidden_size, vocab_size)

self.init_weights()

def init_weights(self):
    for module in self.modules():
        if isinstance(module, nn.Linear):
            module.weight.data.normal_(mean=0.0, std=0.02)
            if module.bias is not None:
                module.bias.data.zero_()
        elif isinstance(module, nn.Embedding):
            module.weight.data.normal_(mean=0.0, std=0.02)

def forward(self, input_ids, attention_mask=None, token_type_ids=None, return_hidden_states=False):
    batch_size, seq_len = input_ids.size()

    # Create attention mask if not provided
    if attention_mask is None:
        attention_mask = torch.ones(batch_size, seq_len, device=input_ids.device)

    # Convert to extended attention mask (1.0 for tokens to attend, 0.0 for masked tokens)
    extended_attention_mask = attention_mask.unsqueeze(1).unsqueeze(2)
    extended_attention_mask = (1.0 - extended_attention_mask) * -10000.0

    # Process embeddings
    hidden_states = self.embeddings(input_ids)

    # Store all hidden states if requested
    all_hidden_states = [] if return_hidden_states else None

    # Forward through all layers
    for layer in self.layers:
        if return_hidden_states:
            all_hidden_states.append(hidden_states)

        hidden_states = layer(hidden_states, extended_attention_mask)

    # Final layer norm
    hidden_states = self.final_norm(hidden_states)

    if return_hidden_states:
        all_hidden_states.append(hidden_states)

    # MLM head
    mlm_output = self.mlm_dense(hidden_states)
    mlm_output = F.gelu(mlm_output)
    mlm_output = self.mlm_norm(mlm_output)
    prediction_scores = self.mlm_head(mlm_output)

    outputs = {
        'last_hidden_state': hidden_states,
        'prediction_logits': prediction_scores,
    }

    if return_hidden_states:
        outputs['hidden_states'] = all_hidden_states

    return outputs

def switch_attention_mode(self, layer_idx, use_moba):

```

```

"""
Switches a specific layer between MoBA and Full Attention
Useful for training strategies that transition between attention types
"""
if layer_idx >= len(self.layers):
    raise ValueError("Layer index {layer_idx} is out of range (0-{len(self.layers)-1})")

hidden_size = self.config["hidden_size"]
num_heads = self.config["num_attention_heads"]
moba_block_size = self.config["moba_block_size"]
moba_top_k = self.config["moba_top_k"]

# Create new attention module
if use_moba:
    new_attention = MoBAAttention(
        hidden_size=hidden_size,
        num_heads=num_heads,
        block_size=moba_block_size,
        top_k=moba_top_k
    )
else:
    new_attention = FullAttention(
        hidden_size=hidden_size,
        num_heads=num_heads
    )

# Transfer weights if dimensions match
old_attention = self.layers[layer_idx].attention
new_attention.q_proj.weight.data.copy_(old_attention.q_proj.weight.data)
new_attention.k_proj.weight.data.copy_(old_attention.k_proj.weight.data)
new_attention.v_proj.weight.data.copy_(old_attention.v_proj.weight.data)
new_attention.o_proj.weight.data.copy_(old_attention.o_proj.weight.data)

# Replace attention module
self.layers[layer_idx].attention = new_attention
self.layers[layer_idx].use_moba = use_moba

def switch_to_hybrid_mode(self, hybrid_layer_count):
    """
    Switches the model to hybrid mode with the last 'hybrid_layer_count' layers
    using Full Attention and the rest using MoBA
    """
    for i in range(len(self.layers)):
        use_moba = i < (len(self.layers) - hybrid_layer_count)
        if use_moba != self.layers[i].use_moba:
            self.switch_attention_mode(i, use_moba)

# Update config
self.config["hybrid_layer_count"] = hybrid_layer_count

def switch_to_full_attention(self):
    """Switches all layers to Full Attention"""
    for i in range(len(self.layers)):
        if self.layers[i].use_moba:
            self.switch_attention_mode(i, False)

# Update config
self.config["hybrid_layer_count"] = len(self.layers)

def switch_to_full_moba(self):
    """Switches all layers to MoBA Attention"""
    for i in range(len(self.layers)):
        if not self.layers[i].use_moba:
            self.switch_attention_mode(i, True)

# Update config
self.config["hybrid_layer_count"] = 0

def create_neobert_moba_model(config):
    """
    Helper function to create a NeoBERT-MOBA model from a configuration
    """
    model = NeoBERTMoBA(
        vocab_size=config.get("vocab_size", 30000),
        hidden_size=config.get("hidden_size", 768),
        num_hidden_layers=config.get("num_hidden_layers", 28),
        num_attention_heads=config.get("num_attention_heads", 12),
        max_position_embeddings=config.get("max_position_embeddings", 32768),
        moba_block_size=config.get("moba_block_size", 512),
        moba_top_k=config.get("moba_top_k", 3),
        hybrid_layer_count=config.get("hybrid_layer_count", 3),
    )
    return model

# Exemple d'utilisation
if __name__ == "__main__":
    # Configuration du modèle
    model_config = {
        # Configuration NeoBERT
        "vocab_size": 30000,
        "hidden_size": 768,
        "num_hidden_layers": 28,

```

```

"num_attention_heads": 12,
"max_position_embeddings": 32768,

# Configuration MOBA
"moba_block_size": 512, # Taille des blocs
"moba_top_k": 3, # Nombre de blocs à sélectionner
"hybrid_layer_count": 3, # Dernières couches en attention complète
}

# Création du modèle
model = create_neobert_moba_model(model_config)

# Test avec un input factice
batch_size = 2
seq_len = 1024
input_ids = torch.randint(0, 30000, (batch_size, seq_len))

# Forward pass
with torch.no_grad():
    outputs = model(input_ids)

print(f"Last hidden state shape: {outputs['last_hidden_state'].shape}")
print(f"Prediction logits shape: {outputs['prediction_logits'].shape}")

# Test de changement de mode d'attention
print(f"Avant: {sum(int(layer.use_moba) for layer in model.layers)} couches MOBA")
model.switch_to_hybrid_mode(5)
print(f"Après: {sum(int(layer.use_moba) for layer in model.layers)} couches MOBA")

# Test avec le nouveau mode d'attention
with torch.no_grad():
    outputs = model(input_ids)

print(f"Last hidden state shape: {outputs['last_hidden_state'].shape}")

```

Explication de l'architecture

Cette implémentation comprend plusieurs composants clés:

1. Composants de base

- **RMSNorm**: Normalisation par racine carrée de moyenne, plus légère que LayerNorm
- **RotaryEmbedding**: Encodage positionnel rotatif (RoPE) pour un meilleur traitement des séquences longues
- **SwiGLU**: Activation avancée pour améliorer les performances du réseau feed-forward

2. Mécanismes d'attention

- **MoBAAttention**: Implémentation du "Mixture of Block Attention"
 - Divise la séquence d'entrée en blocs de taille fixe
 - Calcule des scores de pertinence entre chaque requête et les blocs
 - Sélectionne dynamiquement les top-k blocs pour chaque requête
 - Garantit que chaque requête attende toujours à son bloc courant
 - Applique le masquage causal pour préserver l'autorégressivité
- **FullAttention**: Attention standard complète avec RoPE

3. Architecture hybride

- **NeoBERTMoBALayer**: Couche transformer avec le choix entre MoBA ou attention complète
- **NeoBERTMoBA**: Modèle complet intégrant les deux types d'attention
 - Structure hybride: les premières couches utilisent MoBA, les dernières utilisent l'attention complète
 - Méthodes pour basculer dynamiquement entre les modes d'attention

4. Fonctionnalités de transition

- Transition flexible entre MoBA et attention complète durant l'entraînement
- Support pour les stratégies d'entraînement hybrides (MoBA pour l'efficacité/attention complète pour la précision)
- Possibilité d'ajuster le nombre de couches utilisant chaque type d'attention

Cette architecture est conçue pour offrir un excellent compromis entre efficacité computationnelle et performance sur des contextes longs, tout en restant flexible pour diverses stratégies d'entraînement et d'inférence.

Sources: [2502.19587v1.pdf](#) [2502.13189v1.pdf](#)

USER

écrit le code pour Protocole d'entraînement

BOT

Thoughts

Le protocole d'entraînement pour l'architecture NeoBERT-MOBA devrait inclure:

1. La configuration du processus d'entraînement pour les différentes phases
2. L'implémentation des optimizers, learning rate schedulers, et autres hyperparamètres
3. Des fonctions pour gérer l'entraînement distribué
4. La logique d'entraînement pour les phases 1, 2 et 3 décrites dans le protocole
5. Des mécanismes d'évaluation et de checkpointing

Je vais créer un code complet qui implémente ce protocole d'entraînement en PyTorch avec support pour le distributed training. Le code s'appuiera sur les classes de modèle et de préparation des données que nous avons déjà implémentées.

Voici la structure que je vais suivre:

1. Définition des hyperparamètres d'entraînement pour chaque phase

2. Implémentation d'un trainer avec:
 - Support pour l'entraînement distribué
 - Monitoring des métriques et logging
 - Checkpointing et reprise d'entraînement
3. Implémentation des fonctions d'entraînement spécifiques pour chaque phase
4. Script principal d'exécution

Le code devra être complet et prêt à être exécuté, en supposant que les modules de modèle et de préparation des données sont disponibles.

Code du protocole d'entraînement pour NeoBERT-MOBA

Voici l'implémentation complète du protocole d'entraînement pour l'architecture NeoBERT-MOBA, couvrant les trois phases d'entraînement décrites:

```
import os
import time
import math
import json
import logging
import argparse
from datetime import datetime
from dataclasses import dataclass, field
from typing import Dict, List, Optional, Any, Union, Tuple

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.distributed as dist
from torch.nn.parallel import DistributedDataParallel as DDP
from torch.utils.data import DataLoader
from torch.cuda.amp import GradScaler, autocast
from torch.utils.tensorboard import SummaryWriter

# Supposez que ces imports proviennent de vos modules précédents
from model_architecture import NeoBERTMoBA, create_neobert_moba_model
from data_preparation import DataPreparation

# Configuration logging
logging.basicConfig(
    format="%(asctime)s - %(levelname)s - %(name)s - %(message)s",
    datefmt="%m/%d/%Y %H:%M:%S",
    level=logging.INFO,
)
logger = logging.getLogger(__name__)

@dataclass
class TrainingArguments:
    """Arguments d'entraînement pour NeoBERT-MOBA"""

    # Chemins et organisation
    output_dir: str = field(default="./outputs")
    logging_dir: str = field(default=None)
    checkpoint_dir: str = field(default=None)

    # Hyperparamètres généraux
    learning_rate: float = field(default=6e-4)
    weight_decay: float = field(default=0.1)
    adam_beta1: float = field(default=0.9)
    adam_beta2: float = field(default=0.95)
    adam_epsilon: float = field(default=1e-8)
    max_grad_norm: float = field(default=1.0)

    # Paramètres d'entraînement
    num_train_epochs: int = field(default=1)
    max_steps: int = field(default=-1)
    warmup_steps: int = field(default=2000)
    warmup_ratio: float = field(default=0.0)

    # Taille de lot et accumulation de gradient
    per_device_train_batch_size: int = field(default=8)
    gradient_accumulation_steps: int = field(default=1)

    # Efficacité
    fp16: bool = field(default=True)
    fp16_opt_level: str = field(default="O1")

    # Logging et évaluation
    logging_steps: int = field(default=100)
    eval_steps: int = field(default=1000)
    save_steps: int = field(default=5000)
    save_total_limit: Optional[int] = field(default=3)

    # Distributed training
    local_rank: int = field(default=-1)
    world_size: int = field(default=1)

    # MOBA spécifique
    moba_activation_step: Optional[int] = field(default=None)
    hybrid_layer_transition_steps: List[int] = field(default_factory=list)

    # Phase spécifique
    training_phase: int = field(default=1)

    def __post_init__(self):
```

```

if self.logging_dir is None:
    self.logging_dir = os.path.join(self.output_dir, "logs")
if self.checkpoint_dir is None:
    self.checkpoint_dir = os.path.join(self.output_dir, "checkpoints")

os.makedirs(self.output_dir, exist_ok=True)
os.makedirs(self.logging_dir, exist_ok=True)
os.makedirs(self.checkpoint_dir, exist_ok=True)

class NeoBERTMoBATrainer:
    """Trainer pour l'entraînement du modèle NeoBERT-MOBA"""

    def __init__(
        self,
        model: NeoBERTMoBA,
        args: TrainingArguments,
        train_dataloader: DataLoader,
        eval_dataloader: Optional[DataLoader] = None,
        lr_scheduler=None,
    ):
        self.model = model
        self.args = args
        self.train_dataloader = train_dataloader
        self.eval_dataloader = eval_dataloader

        # Distributed training setup
        self.is_distributed = args.local_rank != -1
        self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

        if self.is_distributed:
            if args.local_rank == 0:
                logger.info(f"Distributed training enabled, world size: {args.world_size}")
            self.device = torch.device(f"cuda:{args.local_rank}")
            torch.cuda.set_device(args.local_rank)
            if not dist.is_initialized():
                dist.init_process_group(backend="nccl")

        self.model = self.model.to(self.device)

        # Wrap model in DDP
        if self.is_distributed:
            self.model = DDP(
                self.model,
                device_ids=[args.local_rank],
                output_device=args.local_rank,
                find_unused_parameters=True
            )

        # Prepare optimizer
        self.optimizer = self._create_optimizer()

        # LR Scheduler
        self.lr_scheduler = lr_scheduler
        if self.lr_scheduler is None:
            self.lr_scheduler = self._create_scheduler()

        # Mixed precision training
        self.scaler = GradScaler() if args.fp16 else None

        # Tensorboard logging
        self.tb_writer = None
        if args.local_rank in [-1, 0]:
            self.tb_writer = SummaryWriter(log_dir=args.logging_dir)

        # Training state
        self.global_step = 0
        self.epoch = 0
        self.best_eval_loss = float("inf")

        # Loss tracking
        self.tr_loss = 0.0
        self.logging_loss = 0.0

    def _create_optimizer(self):
        """Crée l'optimiseur pour l'entraînement"""
        no_decay = ["bias", "norm", "LayerNorm.weight", "RMSNorm.weight"]
        optimizer_grouped_parameters = [
            {
                "params": [p for n, p in self.model.named_parameters()
                           if not any(nd in n for nd in no_decay)],
                "weight_decay": self.args.weight_decay,
            },
            {
                "params": [p for n, p in self.model.named_parameters()
                           if any(nd in n for nd in no_decay)],
                "weight_decay": 0.0,
            },
        ]

    return torch.optim.AdamW(
        optimizer_grouped_parameters,
        lr=self.args.learning_rate,
        betas=(self.args.adam_beta1, self.args.adam_beta2),

```

```

        eps=self.args.adam_epsilon
    )

def _create_scheduler(self):
    """Crée le scheduler de learning rate avec warmup"""
    num_training_steps = self._get_total_training_steps()

    # Calculer les steps de warmup
    if self.args.warmup_ratio > 0:
        warmup_steps = int(num_training_steps * self.args.warmup_ratio)
    else:
        warmup_steps = self.args.warmup_steps

    logger.info(f"Warmup steps: {warmup_steps}")
    logger.info(f"Total training steps: {num_training_steps}")

    def lr_lambda(current_step: int):
        # Warmup phase
        if current_step < warmup_steps:
            return float(current_step) / float(max(1, warmup_steps))
        # Decay phase (cosine decay)
        progress = float(current_step - warmup_steps) / float(max(1, num_training_steps - warmup_steps))
        return 0.5 * (1.0 + math.cos(math.pi * progress))

    return torch.optim.lr_scheduler.LambdaLR(self.optimizer, lr_lambda)

def _get_total_training_steps(self):
    """Calcule le nombre total de steps d'entraînement"""
    if self.args.max_steps > 0:
        return self.args.max_steps

    # Calcul basé sur le nombre d'époques
    num_update_steps_per_epoch = math.ceil(
        len(self.train_dataloader) / self.args.gradient_accumulation_steps
    )

    return num_update_steps_per_epoch * self.args.num_train_epochs

def train(self):
    """Entraîne le modèle selon les paramètres spécifiés"""
    num_training_steps = self._get_total_training_steps()

    logger.info("***** Démarrage de l'entraînement *****")
    logger.info(f" Phase d'entraînement = {self.args.training_phase}")
    logger.info(f" Nombre d'époques = {self.args.num_train_epochs}")
    logger.info(f" Steps d'entraînement = {num_training_steps}")
    logger.info(f" Gradient accumulation steps = {self.args.gradient_accumulation_steps}")
    logger.info(f" Batch size effectif = {self.args.per_device_train_batch_size * self.args.gradient_accumulation_steps * (self.args.world_size if self.is_distributed else 1)}")

    self.model.train()
    self.model.zero_grad()

    train_iterator = range(self.args.num_train_epochs)

    for epoch in train_iterator:
        self.epoch = epoch
        epoch_iterator = self.train_dataloader

        for step, batch in enumerate(epoch_iterator):
            loss = self._training_step(batch)

            # Accumulate gradients if needed
            if (step + 1) % self.args.gradient_accumulation_steps == 0 or (
                step + 1 == len(epoch_iterator)
            ):
                # Clip gradients
                if self.args.fp16:
                    self.scaler.unscale_(self.optimizer)

                torch.nn.utils.clip_grad_norm_(
                    self.model.parameters(), self.args.max_grad_norm
                )

                # Update parameters
                if self.args.fp16:
                    self.scaler.step(self.optimizer)
                    self.scaler.update()
                else:
                    self.optimizer.step()

                self.lr_scheduler.step()
                self.model.zero_grad()
                self.global_step += 1

            # Logging
            if self.args.logging_steps > 0 and self.global_step % self.args.logging_steps == 0:
                self._log_metrics(loss)

            # Evaluation
            if self.eval_dataloader is not None and self.args.eval_steps > 0 and self.global_step % self.args.eval_steps == 0:
                self._evaluate()

            # Checkpointing
            if self.args.save_steps > 0 and self.global_step % self.args.save_steps == 0:

```

```

        self._save_checkpoint()

    # Apply MOBA activation at specific step if configured
    if self.args.moba_activation_step is not None and self.global_step == self.args.moba_activation_step:
        self._activate_moba()

    # Apply hybrid layer transitions at specific steps if configured
    if self.args.hybrid_layer_transition_steps:
        for step_idx, hybrid_count in enumerate(self.args.hybrid_layer_transition_steps):
            if self.global_step == step_idx:
                self._set_hybrid_layers(hybrid_count)

    # Check if we've reached max steps
    if 0 < self.args.max_steps <= self.global_step:
        epoch_iterator.close()
        break

    # Save checkpoint at the end of each epoch
    self._save_checkpoint(epoch=epoch)

    if 0 < self.args.max_steps <= self.global_step:
        train_iterator.close()
        break

    # Final evaluation
    if self.eval_dataloader is not None:
        self._evaluate()

    # Final checkpoint
    self._save_checkpoint(final=True)

    # Close tensorboard writer
    if self.tb_writer:
        self.tb_writer.close()

    logger.info("***** Entraînement terminé *****")

    return self.global_step, self.tr_loss / self.global_step

def _training_step(self, batch):
    """Exécute un pas d'entraînement"""
    # Move batch to device
    batch = {k: v.to(self.device) for k, v in batch.items()}

    # Mixed precision forward pass
    if self.args.fp16:
        with autocast():
            outputs = self.model(**batch)
            loss = outputs['loss'] if 'loss' in outputs else outputs['prediction_logits'].view(-1, outputs['prediction_logits'].size(-1)).gather(1, batch['labels'].view(-1, 1)).mean()
    else:
        outputs = self.model(**batch)
        loss = outputs['loss'] if 'loss' in outputs else outputs['prediction_logits'].view(-1, outputs['prediction_logits'].size(-1)).gather(1, batch['labels'].view(-1, 1)).mean()

    # Scale loss for gradient accumulation
    loss = loss / self.args.gradient_accumulation_steps

    # Backward pass
    if self.args.fp16:
        self.scaler.scale(loss).backward()
    else:
        loss.backward()

    # Update tr_loss for logging
    self.tr_loss += loss.item() * self.args.gradient_accumulation_steps

    return loss.item()

def _log_metrics(self, loss):
    """Log les métriques sur Tensorboard"""
    if self.args.local_rank in [-1, 0]:
        # Calculer la perte moyennée depuis le dernier log
        avg_loss = (self.tr_loss - self.logging_loss) / self.args.logging_steps
        self.logging_loss = self.tr_loss

        # Log sur Tensorboard
        if self.tb_writer:
            self.tb_writer.add_scalar('lr', self.lr_scheduler.get_last_lr()[0], self.global_step)
            self.tb_writer.add_scalar('loss', avg_loss, self.global_step)

        # Log dans la console
        logger.info(f"Step {self.global_step}: loss = {avg_loss:.4f}, lr = {self.lr_scheduler.get_last_lr()[0]:.8f}")

def _evaluate(self):
    """Évalue le modèle sur l'ensemble d'évaluation"""
    if self.eval_dataloader is None:
        return

    logger.info("***** Évaluation *****")
    self.model.eval()
    eval_loss = 0.0
    nb_eval_steps = 0

    for batch in self.eval_dataloader:
        batch = {k: v.to(self.device) for k, v in batch.items()}

```



```

with torch.no_grad():
    outputs = self.model(**batch)
    loss = outputs['loss'] if 'loss' in outputs else outputs['prediction_logits'].view(-1, outputs['prediction_logits'].size(-1)).gather(1, batch['labels'].view(-1, 1)).mean()

    eval_loss += loss.item()
    nb_eval_steps += 1

eval_loss = eval_loss / nb_eval_steps

# Log les résultats
if self.args.local_rank in [-1, 0]:
    logger.info(f"Eval loss: {eval_loss:.4f}")

if self.tb_writer:
    self.tb_writer.add_scalar('eval_loss', eval_loss, self.global_step)

# Sauvegarder le meilleur modèle
if eval_loss < self.best_eval_loss:
    self.best_eval_loss = eval_loss
    self._save_checkpoint(best=True)

self.model.train()

def _save_checkpoint(self, epoch=None, best=False, final=False):
    """Sauverde un checkpoint du modèle"""
    if self.args.local_rank in [-1, 0]:
        # Créer le nom du checkpoint
        prefix = "best" if best else "final" if final else f"checkpoint-{self.global_step}"
        if epoch is not None:
            prefix = f"{prefix}-epoch-{epoch}"

        # Créer le répertoire de sauvegarde
        checkpoint_dir = os.path.join(self.args.checkpoint_dir, prefix)
        os.makedirs(checkpoint_dir, exist_ok=True)

        # Sauvegarder le modèle
        logger.info(f"Sauverde du modèle dans {checkpoint_dir}")

        # Sauverde du modèle (gestion pour DDP)
        model_to_save = self.model.module if hasattr(self.model, "module") else self.model
        torch.save(model_to_save.state_dict(), os.path.join(checkpoint_dir, "pytorch_model.bin"))

        # Sauverde la configuration
        with open(os.path.join(checkpoint_dir, "config.json"), "w") as f:
            json.dump(model_to_save.config, f)

        # Sauverde l'état d'entraînement
        torch.save(
            {
                "optimizer": self.optimizer.state_dict(),
                "lr_scheduler": self.lr_scheduler.state_dict(),
                "scaler": self.scaler.state_dict() if self.scaler else None,
                "epoch": self.epoch,
                "global_step": self.global_step,
                "best_eval_loss": self.best_eval_loss,
            },
            os.path.join(checkpoint_dir, "optimizer.pt"),
        )

        # Nettoyer les checkpoints anciens
        if self.args.save_total_limit > 0:
            self._rotate_checkpoints()

def _rotate_checkpoints(self):
    """Supprime les checkpoints les plus anciens si la limite est atteinte"""
    # Chercher tous les checkpoints
    glob_checkpoints = [
        str(x) for x in os.listdir(self.args.checkpoint_dir)
        if x.startswith("checkpoint-") and os.path.isdir(os.path.join(self.args.checkpoint_dir, x))
    ]

    if len(glob_checkpoints) <= self.args.save_total_limit:
        return

    # Trier par étape (plus récent en dernier)
    ordering_and_checkpoint = []
    for checkpoint in glob_checkpoints:
        try:
            step = int(checkpoint.split("-")[1])
            ordering_and_checkpoint.append((step, checkpoint))
        except:
            continue

    # Garder uniquement les N plus récents
    checkpoints_sorted = sorted(ordering_and_checkpoint)
    checkpoints_sorted = checkpoints_sorted[:len(checkpoints_sorted) - self.args.save_total_limit]

    # Supprimer les plus anciens
    for _, checkpoint in checkpoints_sorted:
        checkpoint_dir = os.path.join(self.args.checkpoint_dir, checkpoint)
        logger.info(f"Suppression du checkpoint obsolète {checkpoint_dir}")
        for filename in os.listdir(checkpoint_dir):
            os.remove(os.path.join(checkpoint_dir, filename))

```

```

os.rmdir(checkpoint_dir)

def _activate_moba(self):
    """Active le mode MoBA pour les couches configurées"""
    logger.info("Activation du mécanisme MOBA")

    # Accéder au modèle (gérer DDP)
    model = self.model.module if hasattr(self.model, "module") else self.model

    # Configurer les couches selon le nombre de couches hybrides défini
    hybrid_count = model.config.get("hybrid_layer_count", 3)
    model.switch_to_hybrid_mode(hybrid_count)

def _set_hybrid_layers(self, hybrid_count):
    """Définit le nombre de couches hybrides (utilisant l'attention complète)"""
    logger.info(f"Configuration des couches hybrides: {hybrid_count} couches en attention complète")

    # Accéder au modèle (gérer DDP)
    model = self.model.module if hasattr(self.model, "module") else self.model

    # Configurer les couches
    model.switch_to_hybrid_mode(hybrid_count)

def train_phase1(data_prep, model_config, training_args=None):
    """Phase 1: Pré-entraînement initial (1024 tokens)"""

    # Configuration pour Phase 1
    if training_args is None:
        training_args = TrainingArguments(
            output_dir="./outputs/phase1",
            logging_dir="./logs/phase1",
            training_phase=1,

            # Hyperparamètres spécifiques à Phase 1
            learning_rate=6e-4,
            warmup_steps=2000,
            weight_decay=0.1,
            max_grad_norm=1.0,

            # Taille de lot et accumulation
            per_device_train_batch_size=32,
            gradient_accumulation_steps=8,

            # Durée d'entraînement
            max_steps=1000000, # 1M steps comme indiqué dans le protocole

            # Logging et checkpointing
            logging_steps=100,
            save_steps=5000,
            eval_steps=5000,

            # Pas de MOBA dans la Phase 1
            moba_activation_step=None,

            # Efficacité
            fp16=True,
        )

    # Créer/charger le modèle
    model = create_neobert_moba_model(model_config)

    # Préparation des données
    train_dataloader = data_prep.get_dataloaders_for_phase(1, ddp=training_args.local_rank != -1)["main_len1024_bs32"]

    # Créer un dataloader d'évaluation si nécessaire
    eval_dataloader = None

    # Créer le trainer
    trainer = NeoBERTMoBATrainer(
        model=model,
        args=training_args,
        train_dataloader=train_dataloader,
        eval_dataloader=eval_dataloader
    )

    # Entraîner le modèle
    global_step, avg_loss = trainer.train()

    logger.info(f"Phase 1 terminée. Steps: {global_step}, Perte moyenne: {avg_loss}")

    return model

def train_phase2(data_prep, model_config, model=None, training_args=None):
    """Phase 2: Extension du contexte (4096 tokens)"""

    # Configuration pour Phase 2
    if training_args is None:
        training_args = TrainingArguments(
            output_dir="./outputs/phase2",
            logging_dir="./logs/phase2",
            training_phase=2,

```

```

# Hyperparamètres spécifiques à Phase 2
learning_rate=3e-4, # Learning rate plus faible pour fine-tuning
warmup_steps=1000,
weight_decay=0.1,
max_grad_norm=1.0,

# Taille de lot et accumulation
per_device_train_batch_size=4,
gradient_accumulation_steps=4,

# Durée d'entraînement
max_steps=50000, # 50K steps comme indiqué dans le protocole

# Logging et checkpointing
logging_steps=100,
save_steps=1000,
eval_steps=1000,

# Pas de MOBA dans la Phase 2
moba_activation_step=None,

# Efficacité
fp16=True,
)

# Charger le modèle de la Phase 1 si non fourni
if model is None:
    # Charger le dernier checkpoint de la Phase 1
    checkpoint_path = "/outputs/phase1/checkpoints/final"
    model = create_neobert_moba_model(model_config)
    model.load_state_dict(torch.load(os.path.join(checkpoint_path, "pytorch_model.bin")))
    logger.info(f"Modèle chargé depuis le checkpoint: {checkpoint_path}")

# Préparation des données pour Phase 2
train_dataloader = data_prep.get_dataloaders_for_phase(2, ddp=training_args.local_rank != -1)

# Créer un dataloader d'évaluation si nécessaire
eval_dataloader = None

# Créer le trainer
trainer = NeoBERTMoBATrainer(
    model=model,
    args=training_args,
    train_dataloader=train_dataloader,
    eval_dataloader=eval_dataloader
)

# Entraîner le modèle
global_step, avg_loss = trainer.train()

logger.info(f"Phase 2 terminée. Steps: {global_step}, Perte moyenne: {avg_loss}")

return model

def train_phase3(data_prep, model_config, model=None, training_args=None):
    """Phase 3: Activation MOBA (jusqu'à 32K tokens)"""

    # Configuration pour Phase 3
    if training_args is None:
        training_args = TrainingArguments(
            output_dir="/outputs/phase3",
            logging_dir="/logs/phase3",
            training_phase=3,

            # Hyperparamètres spécifiques à Phase 3
            learning_rate=2e-4, # Learning rate encore plus faible
            warmup_steps=500,
            weight_decay=0.1,
            max_grad_norm=1.0,

            # Taille de lot et accumulation
            per_device_train_batch_size=1, # Séquences très longues
            gradient_accumulation_steps=8,

            # Durée d'entraînement
            max_steps=50000, # 50K steps comme indiqué dans le protocole

            # Activer MOBA après le premier step
            moba_activation_step=1,

            # Transitions hybrides
            # À 90% des steps (45000), passer à l'attention complète sur toutes les couches
            hybrid_layer_transition_steps=[model_config["num_hidden_layers"]] * 45000 + [0] * 5000,

            # Logging et checkpointing
            logging_steps=50,
            save_steps=1000,
            eval_steps=1000,

            # Efficacité
            fp16=True,
        )

```

```

# Charger le modèle de la Phase 2 si non fourni
if model is None:
    # Charger le dernier checkpoint de la Phase 2
    checkpoint_path = "../outputs/phase2/checkpoints/final"
    model = create_neobert_moba_model(model_config)
    model.load_state_dict(torch.load(os.path.join(checkpoint_path, "pytorch_model.bin")))
    logger.info(f"Modèle chargé depuis le checkpoint: {checkpoint_path}")

# Préparation des données pour Phase 3
train_dataloader = data_prep.get_dataloaders_for_phase(3, ddp=training_args.local_rank != -1)

# Créer un dataloader d'évaluation si nécessaire
eval_dataloader = None

# Créer le trainer
trainer = NeoBERTMoBATrainer(
    model=model,
    args=training_args,
    train_dataloader=train_dataloader,
    eval_dataloader=eval_dataloader
)

# Entraîner le modèle
global_step, avg_loss = trainer.train()

logger.info(f"Phase 3 terminée. Steps: {global_step}, Perte moyenne: {avg_loss}")

return model

def main():
    """Fonction principale qui exécute l'ensemble du protocole d'entraînement"""

    parser = argparse.ArgumentParser(description="Entraînement de NeoBERT-MOBA")
    parser.add_argument("--phase", type=int, default=1, choices=[1, 2, 3], help="Phase d'entraînement à exécuter")
    parser.add_argument("--continue_from_checkpoint", type=str, default=None, help="Chemin vers un checkpoint pour continuer l'entraînement")
    parser.add_argument("--local_rank", type=int, default=-1, help="Rang local pour l'entraînement distribué")
    parser.add_argument("--data_dir", type=str, default="../data", help="Répertoire des données")

    args = parser.parse_args()

    # Initialisation pour distributed training
    if args.local_rank != -1:
        torch.cuda.set_device(args.local_rank)
        dist.init_process_group(backend="nccl")

    # Configuration du modèle
    model_config = {
        # Configuration NeoBERT
        "vocab_size": 30000,
        "hidden_size": 768,
        "num_hidden_layers": 28,
        "num_attention_heads": 12,
        "max_position_embeddings": 32768,

        # Configuration MOBA
        "moba_block_size": 512,
        "moba_top_k": 3,
        "hybrid_layer_count": 3,
    }

    # Préparation des données
    data_prep = DataPreparation(
        base_dir=args.data_dir,
        vocab_size=model_config["vocab_size"],
        max_length=model_config["max_position_embeddings"],
        mlm_probability=0.15
    )

    # Initialiser le modèle si on ne continue pas depuis un checkpoint
    model = None
    if args.continue_from_checkpoint:
        model = create_neobert_moba_model(model_config)
        model.load_state_dict(torch.load(os.path.join(args.continue_from_checkpoint, "pytorch_model.bin")))
        logger.info(f"Modèle chargé depuis le checkpoint: {args.continue_from_checkpoint}")

    # Exécuter la phase d'entraînement spécifiée
    if args.phase == 1:
        model = train_phase1(data_prep, model_config, model)
    elif args.phase == 2:
        model = train_phase2(data_prep, model_config, model)
    elif args.phase == 3:
        model = train_phase3(data_prep, model_config, model)

    logger.info("Entraînement terminé avec succès!")

if __name__ == "__main__":
    main()

```

Explication du protocole d'entraînement

Ce code implémente un protocole d'entraînement complet pour l'architecture NeoBERT-MOBA, qui se déroule en trois phases distinctes, chacune avec des objectifs et

configurations spécifiques:

1. Structure du code

- **TrainingArguments**: Classe de configuration qui définit tous les hyperparamètres d'entraînement
- **NeoBERTMoBATrainer**: Classe principale gérant le processus d'entraînement avec:
 - Support pour l'entraînement distribué (DDP)
 - Mixed precision training (FP16)
 - Logging et checkpointing
 - Optimisation avec AdamW et scheduler à décroissance cosinus avec warmup
 - Gestion des transitions entre modes d'attention (MOBA et attention complète)

2. Phases d'entraînement

- #### Phase 1: Pré-entraînement initial (1024 tokens)
- **Objectif**: Entraînement MLM de base avec séquences de longueur standard
 - **Configuration**:
 - Learning rate: 6e-4
 - Batch size: 32 (× 8 accumulation)
 - 1M steps (2T tokens)
 - Sans MOBA (attention standard uniquement)
 - Warmup: 2000 steps

- #### Phase 2: Extension du contexte (4096 tokens)
- **Objectif**: Adapter le modèle à des séquences plus longues
 - **Configuration**:
 - Learning rate: 3e-4
 - Batch size: 4 (× 4 accumulation)
 - 50K steps (100B tokens)
 - Mélange de données de différentes longueurs
 - Sans MOBA (attention standard uniquement)

- #### Phase 3: Activation MOBA (jusqu'à 32K tokens)
- **Objectif**: Entraîner le modèle sur des séquences très longues avec MOBA
 - **Configuration**:
 - Learning rate: 2e-4
 - Batch size: 1 (× 8 accumulation)
 - 50K steps
 - Activation de MOBA dès le début
 - Stratégie hybride:
 - 90% des steps: MOBA sur premières couches + attention complète sur dernières couches
 - 10% derniers steps: basculement vers attention complète partout

3. Fonctionnalités clés

- **Stratégie d'activation MOBA**: Permet d'activer MOBA à une étape spécifique de l'entraînement
- **Transition hybride**: Bascule dynamique entre différentes configurations de couches MOBA/attention complète
- **Monitoring**: Logging détaillé des métriques d'entraînement et d'évaluation
- **Checkpointing**: Sauvegarde régulière de l'état du modèle et de l'entraînement
- **Reprise d'entraînement**: Possibilité de continuer depuis un checkpoint existant
- **Support distribué**: Entraînement sur plusieurs GPUs avec Distributed Data Parallel

4. Utilisation

- Le script peut être exécuté avec différentes options:
- `--phase`: Sélectionne la phase d'entraînement (1, 2 ou 3)
 - `--continue_from_checkpoint`: Reprend l'entraînement depuis un checkpoint existant
 - `--local_rank`: Rang local pour l'entraînement distribué
 - `--data_dir`: Répertoire des données prétraitées

Ce protocole d'entraînement complet permet d'obtenir un modèle NeoBERT-MOBA performant avec une capacité de traitement de contextes longs significativement améliorée grâce au mécanisme d'attention MOBA.

USER

ecrit le code pour le Protocole d'évaluation

BOT

Thoughts

Le protocole d'évaluation doit permettre de tester les performances de notre modèle NeoBERT-MOBA sur différentes dimensions:

1. Évaluation des performances générales (GLUE, MTEB)
2. Évaluation des capacités de contexte long:
 - Position-wise LM loss
 - Trailing token loss
 - "Needle in a Haystack"
 - RULER @128K
3. Évaluation d'efficacité:
 - Mémoire GPU
 - Temps de calcul
 - Speedup ratio
4. Expériences d'ablation

Voici les principales fonctionnalités que le code d'évaluation devrait inclure:

1. Un système modulaire d'évaluation qui permet de sélectionner différentes tâches
2. Des métriques spécifiques pour évaluer les modèles de contexte long
3. Des métriques d'efficacité computationnelle
4. Des visualisations pour comparer différentes configurations
5. Support pour les expériences d'ablation (tester différentes configurations)

Je vais créer un framework d'évaluation complet qui inclut toutes ces fonctionnalités.

Code du protocole d'évaluation pour NeoBERT-MOBA

Voici l'implémentation complète du protocole d'évaluation pour mesurer les performances de l'architecture NeoBERT-MOBA:

```
import os
import json
import time
import torch
import random
import logging
import argparse
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm
from dataclasses import dataclass, field
from typing import Dict, List, Optional, Union, Any, Tuple
from collections import defaultdict
import pandas as pd
import seaborn as sns
from torch.utils.data import Dataset, DataLoader
import torch.nn.functional as F
from datasets import load_dataset, concatenate_datasets

# Pour l'intégration aux benchmarks
from transformers import AutoTokenizer, AutoModelForMaskedLM
from evaluate import load
from sklearn.metrics import accuracy_score, precision_recall_fscore_support

# Import de nos modules pour NeoBERT-MOBA
from model_architecture import NeoBERTMoBA, create_neobert_moba_model

# Configuration du logging
logging.basicConfig(
    format="%(asctime)s - %(levelname)s - %(name)s - %(message)s",
    datefmt="%m/%d/%Y %H:%M:%S",
    level=logging.INFO,
)
logger = logging.getLogger(__name__)

@dataclass
class EvaluationArguments:
    """Arguments pour l'évaluation du modèle NeoBERT-MOBA"""
    model_path: str = field(
        default=None,
        metadata={"help": "Chemin vers le checkpoint du modèle à évaluer"}
    )
    output_dir: str = field(
        default="./evaluation_results",
        metadata={"help": "Répertoire de sortie pour les résultats d'évaluation"}
    )
    config_path: str = field(
        default=None,
        metadata={"help": "Chemin vers le fichier de configuration du modèle"}
    )
    tokenizer_path: str = field(
        default=None,
        metadata={"help": "Chemin vers le tokenizer"}
    )

    # Évaluations à effectuer
    run_glue: bool = field(
        default=False,
        metadata={"help": "Exécuter l'évaluation GLUE"}
    )
    run_mteb: bool = field(
        default=False,
        metadata={"help": "Exécuter l'évaluation MTEB"}
    )
    run_long_context: bool = field(
        default=True,
        metadata={"help": "Exécuter les évaluations de contexte long"}
    )
    run_efficiency: bool = field(
        default=True,
        metadata={"help": "Exécuter les évaluations d'efficacité"}
    )
    run_ablation: bool = field(
        default=False,
        metadata={"help": "Exécuter les études d'ablation"}
    )

    # Configurations spécifiques
    context_lengths: List[int] = field(
        default_factory=lambda: [2048, 4096, 8192, 16384, 32768],
        metadata={"help": "Longueurs de contexte à évaluer"}
    )
    batch_size: int = field(
        default=1,
        metadata={"help": "Taille du batch pour l'évaluation"}
    )
```

```

max_samples: int = field(
    default=100,
    metadata={"help": "Nombre maximum d'échantillons à évaluer"}
)

# Configurations MOBA pour les ablations
moba_configs: List[Dict[str, Any]] = field(
    default_factory=lambda: [
        {"block_size": 512, "top_k": 2},
        {"block_size": 512, "top_k": 3},
        {"block_size": 512, "top_k": 4},
        {"block_size": 256, "top_k": 3},
        {"block_size": 1024, "top_k": 3}
    ],
    metadata={"help": "Configurations MOBA pour les études d'ablation"}
)

device: str = field(
    default="cuda" if torch.cuda.is_available() else "cpu",
    metadata={"help": "Device à utiliser pour l'évaluation"}
)

class LongContextDataset(Dataset):
    """Dataset pour l'évaluation des capacités de contexte long"""

    def __init__(self, tokenizer, max_length=32768, dataset_name="pg19", split="test", max_samples=100, seed=42):
        self.tokenizer = tokenizer
        self.max_length = max_length
        self.max_samples = max_samples

        np.random.seed(seed)
        random.seed(seed)

        logger.info(f"Chargement du dataset {dataset_name} pour l'évaluation de contexte long")
        self.dataset = load_dataset(dataset_name, split=split)

        # Limiter le nombre d'échantillons
        if max_samples and max_samples < len(self.dataset):
            indices = np.random.choice(len(self.dataset), max_samples, replace=False)
            self.dataset = self.dataset.select(indices)

        logger.info(f"Dataset chargé: {len(self.dataset)} échantillons")

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

    def __getitem__(self, idx):
        text = self.dataset[idx]["text"]

        # Tokeniser le texte
        tokens = self.tokenizer.encode(text, add_special_tokens=True, truncation=True, max_length=self.max_length)

        return {
            "input_ids": tokens,
            "text": text
        }

    def collate_fn(self, batch):
        """Fonction de collation personnalisée pour le dataloader"""
        input_ids = [item["input_ids"] for item in batch]
        texts = [item["text"] for item in batch]

        # Padding
        max_len = max(len(ids) for ids in input_ids)
        padded_input_ids = []
        attention_masks = []

        for ids in input_ids:
            padding_length = max_len - len(ids)
            padded_ids = ids + [self.tokenizer.pad_token_id] * padding_length
            mask = [1] * len(ids) + [0] * padding_length

            padded_input_ids.append(padded_ids)
            attention_masks.append(mask)

        return {
            "input_ids": torch.tensor(padded_input_ids),
            "attention_mask": torch.tensor(attention_masks),
            "texts": texts
        }

class NeedleInHaystackDataset(Dataset):
    """Dataset pour l'évaluation 'Needle in a Haystack'"""

    def __init__(self, tokenizer, max_length=32768, num_samples=100, seed=42):
        self.tokenizer = tokenizer
        self.max_length = max_length
        self.num_samples = num_samples

        np.random.seed(seed)
        random.seed(seed)

```

```

# Charger un dataset de phrases pour créer les contextes
logger.info("Chargement des données pour Needle in a Haystack")
self.wiki_dataset = load_dataset("wikipedia", "20220301.en", split="train")

# Générer les exemples
self.examples = self._generate_examples()

def _generate_examples(self):
    examples = []

    logger.info(f"Génération de {self.num_samples} exemples Needle in a Haystack")
    for _ in tqdm(range(self.num_samples)):
        # Sélectionner une phrase "aiguille"
        needle_idx = random.randint(0, len(self.wiki_dataset) - 1)
        wiki_text = self.wiki_dataset[needle_idx]["text"]

        # Extraire une phrase de longueur moyenne
        sentences = wiki_text.split(". ")
        if len(sentences) < 2:
            continue

        needle_sentence_idx = random.randint(0, len(sentences) - 1)
        needle = sentences[needle_sentence_idx].strip() + "."

        # Vérifier que l'aiguille n'est pas trop courte ni trop longue
        if len(needle.split()) < 5 or len(needle.split()) > 25:
            continue

        # Créer un champ de foin (texte aléatoire)
        haystack_parts = []

        # Position où insérer l'aiguille (aléatoire)
        position_percentile = random.uniform(0.1, 0.9) # Entre 10% et 90% de la longueur totale

        # Construire le champ de foin en tokenisant pour contrôler la longueur
        haystack_tokens = 0
        target_tokens = int(self.max_length * 0.95) # Utiliser 95% de la longueur max

        # Déterminer où insérer l'aiguille en tokens
        needle_position = int(target_tokens * position_percentile)

        # Construire le début du champ de foin
        while haystack_tokens < needle_position:
            # Sélectionner un texte aléatoire différent de celui contenant l'aiguille
            filler_idx = random.randint(0, len(self.wiki_dataset) - 1)
            if filler_idx == needle_idx:
                continue

            filler_text = self.wiki_dataset[filler_idx]["text"]

            # Prendre une portion du texte
            filler_tokens = self.tokenizer.encode(filler_text, add_special_tokens=False)
            tokens_to_take = min(needle_position - haystack_tokens, len(filler_tokens))

            if tokens_to_take <= 0:
                continue

            text_to_add = self.tokenizer.decode(filler_tokens[:tokens_to_take])
            haystack_parts.append(text_to_add)
            haystack_tokens += tokens_to_take

        # Ajouter l'aiguille
        haystack_parts.append(needle)
        needle_tokens = self.tokenizer.encode(needle, add_special_tokens=False)
        haystack_tokens += len(needle_tokens)

        # Continuer à ajouter du texte jusqu'à atteindre la longueur cible
        while haystack_tokens < target_tokens:
            filler_idx = random.randint(0, len(self.wiki_dataset) - 1)
            if filler_idx == needle_idx:
                continue

            filler_text = self.wiki_dataset[filler_idx]["text"]

            filler_tokens = self.tokenizer.encode(filler_text, add_special_tokens=False)
            tokens_to_take = min(target_tokens - haystack_tokens, len(filler_tokens))

            if tokens_to_take <= 0:
                continue

            text_to_add = self.tokenizer.decode(filler_tokens[:tokens_to_take])
            haystack_parts.append(text_to_add)
            haystack_tokens += tokens_to_take

        # Construire le texte complet
        haystack = " ".join(haystack_parts)

        # Générer 5 questions sur l'aiguille
        question = f"Quelle information se trouve dans la phrase: '{needle}'?"

        examples.append({
            "haystack": haystack,
            "needle": needle,
            "question": question,

```



```

        "position_percentile": position_percentile
    })

    logger.info(f"Généré {len(examples)} exemples valides")
    return examples

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

def __getitem__(self, idx):
    example = self.examples[idx]

    # Tokeniser le texte complet (haystack + question)
    full_text = f'{example["haystack"]}\n\nQuestion: {example["question"]}\nRéponse:'

    # Tokenisation
    tokens = self.tokenizer.encode(full_text, add_special_tokens=True, truncation=True, max_length=self.max_length)

    return {
        "input_ids": tokens,
        "needle": example["needle"],
        "position_percentile": example["position_percentile"],
        "text": full_text
    }

def collate_fn(self, batch):
    """Fonction de collation personnalisée pour le dataloader"""
    input_ids = [item["input_ids"] for item in batch]
    needles = [item["needle"] for item in batch]
    positions = [item["position_percentile"] for item in batch]
    texts = [item["text"] for item in batch]

    # Padding
    max_len = max(len(ids) for ids in input_ids)
    padded_input_ids = []
    attention_masks = []

    for ids in input_ids:
        padding_length = max_len - len(ids)
        padded_ids = ids + [self.tokenizer.pad_token_id] * padding_length
        mask = [1] * len(ids) + [0] * padding_length

        padded_input_ids.append(padded_ids)
        attention_masks.append(mask)

    return {
        "input_ids": torch.tensor(padded_input_ids),
        "attention_mask": torch.tensor(attention_masks),
        "needles": needles,
        "positions": positions,
        "texts": texts
    }

class RULERDataset(Dataset):
    """Dataset pour l'évaluation RULER (Reasoning Understanding Long & Extended Reasoning)"""

    def __init__(self, tokenizer, max_length=32768, num_samples=50, sequence_length=128000, seed=42):
        self.tokenizer = tokenizer
        self.max_length = max_length
        self.num_samples = num_samples
        self.sequence_length = sequence_length

        np.random.seed(seed)
        random.seed(seed)

        logger.info("Création des exemples RULER")
        self.examples = self._generate_examples()

    def _generate_examples(self):
        """Génère des exemples de raisonnement à longue distance"""
        examples = []

        for i in range(self.num_samples):
            # Créer un problème de raisonnement à longue distance
            example = self._create_long_dependency_problem()
            examples.append(example)

        return examples

    def _create_long_dependency_problem(self):
        """Crée un problème avec des dépendances à longue distance"""
        # Générer un ensemble de variables avec des valeurs
        variables = {}
        for var in "ABCDEFGHIJKLMNOPQRSTUVWXYZ":
            variables[var] = random.randint(1, 100)

        # Créer une série d'étapes de calcul
        steps = []
        dependencies = []
        final_variable = None

        # Choisir des variables à utiliser dans ce problème
        selected_vars = random.sample(list(variables.keys()), min(10, len(variables)))

```

```

# Créer le problème avec des dépendances à longue distance
operations = ["+", "-", "*", "/"]

for i in range(len(selected_vars) - 1):
    # Choisir deux variables pour l'opération
    if i == 0:
        # Première étape: utiliser deux variables initiales
        var1, var2 = selected_vars[0], selected_vars[1]
        dependencies.extend([var1, var2])
    else:
        # Utiliser le résultat précédent et une nouvelle variable
        var1 = f"result_{i-1}"
        var2 = selected_vars[i+1]
        dependencies.append(var2)

    # Choisir une opération
    op = random.choice(operations)

    # Créer l'étape
    result_var = f"result_{i}"
    if op == "+":
        step = f"{result_var} = {var1} + {var2}"
    elif op == "-":
        step = f"{result_var} = {var1} - {var2}"
    elif op == "*":
        step = f"{result_var} = {var1} * {var2}"
    else: # division entière
        # Éviter la division par zéro
        if var2 == "0" or (var2 in variables and variables[var2] == 0):
            op = "+"
            step = f"{result_var} = {var1} + {var2}"
        else:
            step = f"{result_var} = {var1} // {var2}"

    steps.append(step)
    final_variable = result_var

# Créer le texte du problème
problem_text = "Vous devez suivre les étapes de calcul suivantes:\n\n"

# Ajouter les définitions de variables
problem_text += "Définitions des variables:\n"
for var in selected_vars:
    problem_text += f"{var} = {variables[var]}\n"

problem_text += "\nCalculs à effectuer:\n"

# Ajouter beaucoup de texte entre les définitions et les étapes
filler_text = "Ceci est un texte de remplissage pour créer une longue séquence. " * 5000
problem_text += f"\n{filler_text}\n\n"

# Ajouter les étapes
for step in steps:
    problem_text += f"{step}\n"

# Ajouter la question
question = f"Quelle est la valeur finale de {final_variable}?"
problem_text += f"\n{question}"

# Calculer la réponse
namespace = variables.copy()
for step in steps:
    exec(step, namespace)

answer = namespace[final_variable]

return {
    "problem_text": problem_text,
    "question": question,
    "answer": answer,
    "dependencies": dependencies
}

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

def __getitem__(self, idx):
    example = self.examples[idx]

    # Tokeniser le texte complet
    tokens = self.tokenizer.encode(example["problem_text"], add_special_tokens=True, truncation=True, max_length=self.max_length)

    return {
        "input_ids": tokens,
        "question": example["question"],
        "answer": example["answer"],
        "dependencies": example["dependencies"],
        "text": example["problem_text"]
    }

def collate_fn(self, batch):
    """Fonction de collation personnalisée pour le dataloader"""
    input_ids = [item["input_ids"] for item in batch]

```

```

questions = [item["question"] for item in batch]
answers = [item["answer"] for item in batch]
dependencies = [item["dependencies"] for item in batch]
texts = [item["text"] for item in batch]

# Padding
max_len = max(len(ids) for ids in input_ids)
padded_input_ids = []
attention_masks = []

for ids in input_ids:
    padding_length = max_len - len(ids)
    padded_ids = ids + [self.tokenizer.pad_token_id] * padding_length
    mask = [1] * len(ids) + [0] * padding_length

    padded_input_ids.append(padded_ids)
    attention_masks.append(mask)

return {
    "input_ids": torch.tensor(padded_input_ids),
    "attention_mask": torch.tensor(attention_masks),
    "questions": questions,
    "answers": answers,
    "dependencies": dependencies,
    "texts": texts
}

```

class NeoBERTMoBAEvaluator:

"""Classe principale pour l'évaluation de NeoBERT-MOBA"""

```

def __init__(self, args):
    self.args = args

```

```

# Créer le répertoire de sortie
os.makedirs(args.output_dir, exist_ok=True)

```

```

# Charger le modèle
self.model = self._load_model()

```

```

# Charger le tokenizer
self.tokenizer = self._load_tokenizer()

```

```

self.device = torch.device(args.device)
self.model.to(self.device)

```

```

logger.info(f"Modèle et tokenizer chargés. Dispositif: {self.device}")

```

```

def _load_model(self):
    """Charge le modèle NeoBERT-MOBA à partir du checkpoint"""
    if self.args.model_path is None:
        raise ValueError("Le chemin du modèle doit être spécifié")

    logger.info(f"Chargement du modèle depuis {self.args.model_path}")

    # Charger la configuration
    config_path = self.args.config_path or os.path.join(self.args.model_path, "config.json")
    with open(config_path, "r") as f:
        config = json.load(f)

    # Créer le modèle
    model = create_neobert_moba_model(config)

    # Charger les poids
    weights_path = self.args.model_path
    if os.path.isdir(weights_path):
        weights_path = os.path.join(weights_path, "pytorch_model.bin")

    model.load_state_dict(torch.load(weights_path, map_location="cpu"))

    return model

```

```

def _load_tokenizer(self):
    """Charge le tokenizer pour le modèle"""
    tokenizer_path = self.args.tokenizer_path or self.args.model_path

    logger.info(f"Chargement du tokenizer depuis {tokenizer_path}")
    tokenizer = AutoTokenizer.from_pretrained(tokenizer_path)

    # S'assurer que les tokens spéciaux sont définis
    if tokenizer.pad_token is None:
        tokenizer.pad_token = tokenizer.eos_token or "[PAD]"

    return tokenizer

```

```

def evaluate_all(self):
    """Exécute toutes les évaluations demandées"""
    results = {}

    # Évaluation GLUE
    if self.args.run_glue:
        logger.info("Démarrage de l'évaluation GLUE")
        glue_results = self.evaluate_glue()
        results["glue"] = glue_results

```

```

# Sauvegarde des résultats intermédiaires
with open(os.path.join(self.args.output_dir, "glue_results.json"), "w") as f:
    json.dump(glue_results, f, indent=2)

# Évaluation MTEB
if self.args.run_mteb:
    logger.info("Démarrage de l'évaluation MTEB")
    mteb_results = self.evaluate_mteb()
    results["mteb"] = mteb_results

# Sauvegarde des résultats intermédiaires
with open(os.path.join(self.args.output_dir, "mteb_results.json"), "w") as f:
    json.dump(mteb_results, f, indent=2)

# Évaluation de contexte long
if self.args.run_long_context:
    logger.info("Démarrage de l'évaluation de contexte long")
    long_context_results = self.evaluate_long_context()
    results["long_context"] = long_context_results

# Sauvegarde des résultats intermédiaires
with open(os.path.join(self.args.output_dir, "long_context_results.json"), "w") as f:
    json.dump(long_context_results, f, indent=2)

# Évaluation d'efficacité
if self.args.run_efficiency:
    logger.info("Démarrage de l'évaluation d'efficacité")
    efficiency_results = self.evaluate_efficiency()
    results["efficiency"] = efficiency_results

# Sauvegarde des résultats intermédiaires
with open(os.path.join(self.args.output_dir, "efficiency_results.json"), "w") as f:
    json.dump(efficiency_results, f, indent=2)

# Études d'ablation
if self.args.run_ablation:
    logger.info("Démarrage des études d'ablation")
    ablation_results = self.run_ablation_studies()
    results["ablation"] = ablation_results

# Sauvegarde des résultats intermédiaires
with open(os.path.join(self.args.output_dir, "ablation_results.json"), "w") as f:
    json.dump(ablation_results, f, indent=2)

# Sauvegarde des résultats complets
logger.info("Sauvegarde des résultats d'évaluation")
with open(os.path.join(self.args.output_dir, "evaluation_results.json"), "w") as f:
    json.dump(results, f, indent=2)

# Générer des visualisations
self.generate_visualizations(results)

return results

def evaluate_glue(self):
    """Évalue le modèle sur les tâches GLUE"""
    logger.info("Évaluation sur GLUE non implémentée dans cette version")
    return {"status": "not_implemented"}

def evaluate_mteb(self):
    """Évalue le modèle sur les tâches MTEB"""
    logger.info("Évaluation sur MTEB non implémentée dans cette version")
    return {"status": "not_implemented"}

def evaluate_long_context(self):
    """Évalue les capacités de contexte long du modèle"""
    results = {}

    # 1. Position-wise LM loss
    logger.info("Évaluation de la perte de langage par position")
    position_loss_results = self.evaluate_position_wise_loss()
    results["position_wise_loss"] = position_loss_results

    # 2. Trailing token loss
    logger.info("Évaluation de la perte sur les tokens finaux")
    trailing_loss_results = self.evaluate_trailing_token_loss()
    results["trailing_token_loss"] = trailing_loss_results

    # 3. Needle in a Haystack
    logger.info("Évaluation Needle in a Haystack")
    needle_results = self.evaluate_needle_in_haystack()
    results["needle_in_haystack"] = needle_results

    # 4. RULER (Raisonnement à longue distance)
    logger.info("Évaluation RULER")
    ruler_results = self.evaluate_ruler()
    results["ruler"] = ruler_results

    return results

def evaluate_position_wise_loss(self):
    """Évalue la perte par position dans le contexte"""
    results = {}

```

```

# Pour chaque longueur de contexte
for context_length in self.args.context_lengths:
    if context_length > self.model.config["max_position_embeddings"]:
        logger.info(f"Skipping context length {context_length} (exceeds model's max position embeddings)")
        continue

    logger.info(f"Évaluation de la perte position par position pour longueur {context_length}")

    # Créer un dataset pour cette longueur
    dataset = LongContextDataset(
        self.tokenizer,
        max_length=context_length,
        max_samples=min(20, self.args.max_samples) # Moins d'échantillons pour les contextes longs
    )

    dataloader = DataLoader(
        dataset,
        batch_size=self.args.batch_size,
        collate_fn=dataset.collate_fn
    )

    # Mesurer la perte par position
    position_losses = defaultdict(list)

    self.model.eval()
    with torch.no_grad():
        for batch in tqdm(dataloader, desc=f"Évaluation contexte {context_length}"):
            input_ids = batch["input_ids"].to(self.device)
            attention_mask = batch["attention_mask"].to(self.device)

            # Créer des labels (décalés d'une position)
            labels = input_ids.clone()
            labels[:, :-1] = input_ids[:, 1:]
            labels[:, -1] = -100 # Ignorer le dernier token

            # Forward pass
            outputs = self.model(input_ids, attention_mask)
            logits = outputs["prediction_logits"]

            # Calculer la perte par position
            for pos in range(min(context_length, input_ids.size(1))):
                # Extraire les logits et labels pour cette position
                pos_logits = logits[:, pos, :]
                pos_labels = labels[:, pos]

                # Ignorer les positions masquées
                if (pos_labels != -100).sum() == 0:
                    continue

                # Calculer la perte
                loss = F.cross_entropy(pos_logits, pos_labels, ignore_index=-100, reduction="mean")

                # Déterminer le bin de position
                bin_idx = pos // 2000 # Regrouper par blocs de 2K tokens
                bin_name = f"{bin_idx*2000}-{(bin_idx+1)*2000}"

                position_losses[bin_name].append(loss.item())

            # Calculer la moyenne des pertes par bin
            avg_losses = {bin_name: np.mean(losses) for bin_name, losses in position_losses.items()}
            perplexities = {bin_name: np.exp(avg_loss) for bin_name, avg_loss in avg_losses.items()}

            results[str(context_length)] = {
                "losses": avg_losses,
                "perplexities": perplexities
            }

    return results

def evaluate_trailing_token_loss(self):
    """Évalue la perte sur les derniers tokens des longues séquences"""
    results = {}

    # Pour chaque longueur de contexte
    for context_length in self.args.context_lengths:
        if context_length > self.model.config["max_position_embeddings"]:
            continue

        logger.info(f"Évaluation de la perte sur les tokens finaux pour longueur {context_length}")

        # Créer un dataset pour cette longueur
        dataset = LongContextDataset(
            self.tokenizer,
            max_length=context_length,
            max_samples=min(20, self.args.max_samples)
        )

        dataloader = DataLoader(
            dataset,
            batch_size=self.args.batch_size,
            collate_fn=dataset.collate_fn
        )

```

```

# Mesurer la perte sur les derniers 2K tokens
trailing_losses = []

self.model.eval()
with torch.no_grad():
    for batch in tqdm(dataloader, desc="Évaluation trailing {context_length}"):
        input_ids = batch["input_ids"].to(self.device)
        attention_mask = batch["attention_mask"].to(self.device)

        # S'assurer que nous avons des séquences assez longues
        if input_ids.size(1) < 2048:
            continue

        # Créer des labels (décalés d'une position)
        labels = input_ids.clone()
        labels[:, :-1] = input_ids[:, 1:]
        labels[:, -1] = -100 # Ignorer le dernier token

        # Forward pass
        outputs = self.model(input_ids, attention_mask)
        logits = outputs["prediction_logits"]

        # Isoler les 2K derniers tokens
        trailing_start = max(0, input_ids.size(1) - 2048)
        trailing_logits = logits[:, trailing_start:, :]
        trailing_labels = labels[:, trailing_start:]

        # Calculer la perte
        loss = F.cross_entropy(
            trailing_logits.reshape(-1, trailing_logits.size(-1)),
            trailing_labels.reshape(-1),
            ignore_index=-100,
            reduction="mean"
        )

        trailing_losses.append(loss.item())

# Calculer la moyenne des pertes
if trailing_losses:
    avg_loss = np.mean(trailing_losses)
    perplexity = np.exp(avg_loss)

    results[str(context_length)] = {
        "loss": avg_loss,
        "perplexity": perplexity
    }
else:
    results[str(context_length)] = {
        "loss": None,
        "perplexity": None,
        "note": "Pas assez de données pour calculer la perte"
    }

return results

def evaluate_needle_in_haystack(self):
    """Évalue la capacité à trouver de l'information dans un long contexte"""
    results = {}

    # Définir les longueurs de contexte pour ce test
    test_lengths = [8192, 16384, 32768]
    test_lengths = [l for l in test_lengths if l <= self.model.config["max_position_embeddings"]]

    if not test_lengths:
        logger.warning("Aucune longueur de contexte valide pour le test Needle in a Haystack")
        return {"status": "skipped"}

    # Pour chaque longueur de contexte
    for context_length in test_lengths:
        logger.info(f"Évaluation Needle in a Haystack pour longueur {context_length}")

        # Créer le dataset
        dataset = NeedleInHaystackDataset(
            self.tokenizer,
            max_length=context_length,
            num_samples=min(30, self.args.max_samples)
        )

        dataloader = DataLoader(
            dataset,
            batch_size=self.args.batch_size,
            collate_fn=dataset.collate_fn
        )

        # Collecter les résultats par quartile de position
        position_results = {
            "q1": [], # 0-25%
            "q2": [], # 25-50%
            "q3": [], # 50-75%
            "q4": [] # 75-100%
        }

    self.model.eval()
    with torch.no_grad():

```

```

for batch in tqdm(data_loader, desc="Needle in Haystack {context_length}"):
    input_ids = batch["input_ids"].to(self.device)
    attention_mask = batch["attention_mask"].to(self.device)
    needles = batch["needles"]
    positions = batch["positions"]

    # Forward pass
    outputs = self.model(input_ids, attention_mask)
    logits = outputs["prediction_logits"]

    # Évaluer les résultats
    for i, (pos, needle) in enumerate(zip(positions, needles)):
        # Déterminer le quartile
        quartile = "q1" if pos < 0.25 else "q2" if pos < 0.5 else "q3" if pos < 0.75 else "q4"

        # Trouver où la question commence
        text = batch["texts"][i]
        question_start = text.find("Question: ")

        if question_start == -1:
            continue

        # Isoler la partie après "Réponse:"
        response_start = text.find("Réponse:", question_start)
        if response_start == -1:
            response_start = len(text) - 1

        # Convertir en tokens
        response_token_idx = len(self.tokenizer.encode(text[:response_start], add_special_tokens=True)) - 1

        # Obtenir les prédictions pour les prochains tokens
        next_token_logits = logits[i, response_token_idx:response_token_idx+5]

        # Vérifier si les prédictions évoquent le contenu de l'aiguille
        keywords = needle.split()
        keywords = [k.lower() for k in keywords if len(k) > 3] # Mots significatifs

        if not keywords:
            continue

        # Prendre les top-5 tokens prédits
        top_tokens = []
        for token_logits in next_token_logits:
            top5_values, top5_indices = torch.topk(token_logits, 5)
            top_tokens.extend(self.tokenizer.convert_ids_to_tokens(top5_indices.tolist()))

        # Vérifier si un des mots clés de l'aiguille apparaît dans les prédictions
        found = False
        for keyword in keywords:
            if any(keyword.lower() in token.lower() for token in top_tokens):
                found = True
                break

        position_results[quartile].append(int(found))

# Calculer les taux de succès par quartile
success_rates = {}
for quartile, results_list in position_results.items():
    if results_list:
        success_rates[quartile] = {
            "success_rate": np.mean(results_list),
            "count": len(results_list)
        }
    else:
        success_rates[quartile] = {
            "success_rate": None,
            "count": 0,
            "note": "Pas d'exemples dans ce quartile"
        }

results[str(context_length)] = success_rates

return results

def evaluate_ruler(self):
    """Évalue la compréhension du raisonnement à longue distance"""
    results = {}

    # Définir les longueurs de contexte pour ce test
    test_lengths = [32768]
    test_lengths = [l for l in test_lengths if l <= self.model.config["max_position_embeddings"]]

    if not test_lengths:
        logger.warning("Aucune longueur de contexte valide pour le test RULER")
        return {"status": "skipped"}

    # Pour chaque longueur de contexte
    for context_length in test_lengths:
        logger.info(f"Évaluation RULER pour longueur {context_length}")

        # Créer le dataset
        dataset = RULERDataset(
            self.tokenizer,
            max_length=context_length,

```

```

        num_samples=min(20, self.args.max_samples)
    )

    dataloader = DataLoader(
        dataset,
        batch_size=self.args.batch_size,
        collate_fn=dataset.collate_fn
    )

    # Collecter les résultats
    correct_answers = 0
    total_examples = 0

    self.model.eval()
    with torch.no_grad():
        for batch in tqdm(dataloader, desc=f"RULER {context_length}"):
            input_ids = batch["input_ids"].to(self.device)
            attention_mask = batch["attention_mask"].to(self.device)
            answers = batch["answers"]

            # Forward pass
            outputs = self.model(input_ids, attention_mask)
            logits = outputs["prediction_logits"]

            # Évaluer les résultats
            for i, answer in enumerate(answers):
                # Trouver où la question finit
                text = batch["texts"][i]
                question_end = text.find("?") + 1

                if question_end <= 0:
                    continue

                # Convertir en tokens
                response_token_idx = len(self.tokenizer.encode(text[:question_end], add_special_tokens=True)) - 1

                # Obtenir les prédictions pour les prochains tokens
                next_token_logits = logits[i, response_token_idx:response_token_idx+10]

                # Convertir la réponse attendue en texte
                expected_answer_str = str(answer)

                # Décoder les tokens prédits
                predicted_tokens = []
                for token_logits in next_token_logits:
                    top_idx = torch.argmax(token_logits).item()
                    predicted_tokens.append(top_idx)

                predicted_text = self.tokenizer.decode(predicted_tokens)

                # Vérifier si la réponse attendue apparaît dans la prédiction
                if expected_answer_str in predicted_text:
                    correct_answers += 1

            total_examples += 1

    # Calculer le taux de succès
    if total_examples > 0:
        success_rate = correct_answers / total_examples
    else:
        success_rate = None

    results[str(context_length)] = {
        "success_rate": success_rate,
        "correct": correct_answers,
        "total": total_examples
    }

    return results

def evaluate_efficiency(self):
    """Évalue l'efficacité computationnelle du modèle"""
    results = {}

    # 1. Mesurer la consommation mémoire
    logger.info("Mesure de la consommation mémoire")
    memory_usage = self.measure_memory_usage()
    results["memory_usage"] = memory_usage

    # 2. Mesurer les temps de calcul
    logger.info("Mesure des temps de calcul")
    computation_times = self.measure_computation_times()
    results["computation_times"] = computation_times

    # 3. Calculer le ratio d'accélération
    logger.info("Calcul du ratio d'accélération MOBA vs attention complète")
    speedup_ratio = self.calculate_speedup_ratio()
    results["speedup_ratio"] = speedup_ratio

    return results

def measure_memory_usage(self):
    """Mesure la consommation mémoire pour différentes longueurs"""
    results = {}

```



```

# Récupérer l'accès à la mémoire CUDA si disponible
torch.cuda.empty_cache()

for context_length in self.args.context_lengths:
    if context_length > self.model.config["max_position_embeddings"]:
        continue

    logger.info(f"Mesure de la consommation mémoire pour longueur {context_length}")

    # Créer un input de la taille souhaitée
    dummy_input = torch.randint(
        0, self.model.config["vocab_size"],
        (1, context_length),
        device=self.device
    )

    # Mesurer la mémoire avant
    if torch.cuda.is_available():
        torch.cuda.reset_peak_memory_stats()
        torch.cuda.empty_cache()
        start_mem = torch.cuda.memory_allocated()

    # Forward pass
    self.model.eval()
    with torch.no_grad():
        # Mode MOBA
        model_to_test = self.model
        if hasattr(model_to_test, "module"):
            model_to_test = model_to_test.module

        # Activer le mode MOBA
        if hasattr(model_to_test, "switch_to_hybrid_mode"):
            model_to_test.switch_to_hybrid_mode(model_to_test.config["hybrid_layer_count"])

        _ = self.model(dummy_input)

    # Mesurer la mémoire après
    if torch.cuda.is_available():
        peak_mem_moba = torch.cuda.max_memory_allocated() - start_mem
        torch.cuda.empty_cache()
    else:
        peak_mem_moba = None

    # Réinitialiser les statistiques de mémoire
    if torch.cuda.is_available():
        torch.cuda.reset_peak_memory_stats()
        torch.cuda.empty_cache()
        start_mem = torch.cuda.memory_allocated()

    # Mode attention complète
    if hasattr(model_to_test, "switch_to_full_attention"):
        model_to_test.switch_to_full_attention()

    _ = self.model(dummy_input)

    # Mesurer la mémoire après
    if torch.cuda.is_available():
        peak_mem_full = torch.cuda.max_memory_allocated() - start_mem
        torch.cuda.empty_cache()
    else:
        peak_mem_full = None

    # Restaurer le mode MOBA
    if hasattr(model_to_test, "switch_to_hybrid_mode"):
        model_to_test.switch_to_hybrid_mode(model_to_test.config["hybrid_layer_count"])

    # Enregistrer les résultats
    results[str(context_length)] = {
        "moba_memory": peak_mem_moba / (1024 * 1024) if peak_mem_moba is not None else None, # MB
        "full_memory": peak_mem_full / (1024 * 1024) if peak_mem_full is not None else None, # MB
        "memory_reduction": (
            (1 - peak_mem_moba / peak_mem_full) * 100
            if peak_mem_moba is not None and peak_mem_full is not None else None
        ), # Pourcentage
    }

return results

def measure_computation_times(self):
    """Mesure les temps de calcul pour différentes longueurs"""
    results = {}

    for context_length in self.args.context_lengths:
        if context_length > self.model.config["max_position_embeddings"]:
            continue

        logger.info(f"Mesure des temps de calcul pour longueur {context_length}")

        # Créer un input de la taille souhaitée
        dummy_input = torch.randint(
            0, self.model.config["vocab_size"],
            (1, context_length),
            device=self.device

```

```

)

# Mesurer le temps pour MOBA
self.model.eval()

# Mode MOBA
model_to_test = self.model
if hasattr(model_to_test, "module"):
    model_to_test = model_to_test.module

# Activer le mode MOBA
if hasattr(model_to_test, "switch_to_hybrid_mode"):
    model_to_test.switch_to_hybrid_mode(model_to_test.config["hybrid_layer_count"])

# Warm-up
with torch.no_grad():
    for _ in range(3):
        _ = self.model(dummy_input)

# Mesurer le temps
torch.cuda.synchronize()
start_time = time.time()

with torch.no_grad():
    for _ in range(5):
        _ = self.model(dummy_input)

torch.cuda.synchronize()
moba_time = (time.time() - start_time) / 5

# Mode attention complète
if hasattr(model_to_test, "switch_to_full_attention"):
    model_to_test.switch_to_full_attention()

# Warm-up
with torch.no_grad():
    for _ in range(3):
        _ = self.model(dummy_input)

# Mesurer le temps
torch.cuda.synchronize()
start_time = time.time()

with torch.no_grad():
    for _ in range(5):
        _ = self.model(dummy_input)

torch.cuda.synchronize()
full_time = (time.time() - start_time) / 5

# Restaurer le mode MOBA
if hasattr(model_to_test, "switch_to_hybrid_mode"):
    model_to_test.switch_to_hybrid_mode(model_to_test.config["hybrid_layer_count"])

# Enregistrer les résultats
results[str(context_length)] = {
    "moba_time": moba_time,
    "full_time": full_time,
    "speedup": full_time / moba_time if moba_time > 0 else None
}

return results

def calculate_speedup_ratio(self):
    """Calcule le ratio d'accélération entre MOBA et attention complète"""
    # Utiliser les résultats déjà calculés des temps de calcul
    computation_times = self.measure_computation_times()

    # Extraire les speedups
    speedups = {
        length: data["speedup"]
        for length, data in computation_times.items()
        if "speedup" in data and data["speedup"] is not None
    }

    # Calculer des statistiques
    if speedups:
        avg_speedup = np.mean(list(speedups.values()))
        max_speedup = max(speedups.values())
        min_speedup = min(speedups.values())
    else:
        avg_speedup = None
        max_speedup = None
        min_speedup = None

    return {
        "per_length": speedups,
        "average": avg_speedup,
        "maximum": max_speedup,
        "minimum": min_speedup
    }

def run_ablation_studies(self):
    """Exécute les études d'ablation sur différentes configurations MOBA"""

```

```

results = {}

# Définir une longueur de contexte de test pour les ablations
test_length = 8192
if test_length > self.model.config["max_position_embeddings"]:
    test_length = self.model.config["max_position_embeddings"]

logger.info(f"Exécution des études d'ablation avec contexte de {test_length} tokens")

# 1. Impact de la granularité des blocs
logger.info("Ablation: Impact de la granularité des blocs")
block_granularity_results = self.ablation_block_granularity(test_length)
results["block_granularity"] = block_granularity_results

# 2. Impact de la stratégie hybride
logger.info("Ablation: Impact de la stratégie hybride")
hybrid_strategy_results = self.ablation_hybrid_strategy(test_length)
results["hybrid_strategy"] = hybrid_strategy_results

# 3. Impact du choix des métriques pour le routage
# Note: Cette ablation nécessiterait de modifier le code du modèle
# pour supporter différentes métriques de routage
logger.info("Ablation: Impact des métriques de routage non implémentée")
results["routing_metrics"] = {"status": "not_implemented"}

return results

def ablation_block_granularity(self, context_length):
    """Étudie l'impact de la granularité des blocs"""
    results = {}

    # Créer un input de la taille souhaitée
    dummy_input = torch.randint(
        0, self.model.config["vocab_size"],
        (1, context_length),
        device=self.device
    )

    model_to_test = self.model
    if hasattr(model_to_test, "module"):
        model_to_test = model_to_test.module

    # Tester différentes configurations
    for config in self.args.moba_configs:
        block_size = config["block_size"]
        top_k = config["top_k"]

        logger.info(f"Test de granularité: block_size={block_size}, top_k={top_k}")

        # Mettre à jour la configuration
        if not hasattr(model_to_test, "config"):
            logger.warning("Le modèle n'a pas d'attribut config, impossible de modifier la configuration MOBA")
            continue

        # Sauvegarde de la configuration originale
        original_block_size = model_to_test.config.get("moba_block_size", 512)
        original_top_k = model_to_test.config.get("moba_top_k", 3)

        # Appliquer la nouvelle configuration
        model_to_test.config["moba_block_size"] = block_size
        model_to_test.config["moba_top_k"] = top_k

        # Réinitialiser les couches avec la nouvelle configuration
        if hasattr(model_to_test, "switch_to_hybrid_mode"):
            hybrid_count = model_to_test.config.get("hybrid_layer_count", 3)

        # Recréer les couches MoBA avec les nouveaux paramètres
        for i in range(len(model_to_test.layers)):
            if i < (len(model_to_test.layers) - hybrid_count):
                # Cette couche devrait utiliser MoBA
                model_to_test.layers[i].use_moba = True
                model_to_test.layers[i].attention = model_to_test.MoBAAttention(
                    hidden_size=model_to_test.config["hidden_size"],
                    num_heads=model_to_test.config["num_attention_heads"],
                    block_size=block_size,
                    top_k=top_k
                )

        # Mesurer les performances
        self.model.eval()
        with torch.no_grad():
            # Mesurer le temps et la mémoire
            if torch.cuda.is_available():
                torch.cuda.empty_cache()
                torch.cuda.reset_peak_memory_stats()
                start_mem = torch.cuda.memory_allocated()

            torch.cuda.synchronize()
            start_time = time.time()

            _ = self.model(dummy_input)

            torch.cuda.synchronize()
            end_time = time.time()

```

```

if torch.cuda.is_available():
    peak_mem = torch.cuda.max_memory_allocated() - start_mem
    torch.cuda.empty_cache()
else:
    peak_mem = None

# Mesurer la qualité (perte MLM)
# Créer un dataset pour cette longueur
dataset = LongContextDataset(
    self.tokenizer,
    max_length=context_length,
    max_samples=5 # Petit échantillon pour l'ablation
)

dataloader = DataLoader(
    dataset,
    batch_size=1,
    collate_fn=dataset.collate_fn
)

# Mesurer la perte
losses = []

for batch in dataloader:
    input_ids = batch["input_ids"].to(self.device)
    attention_mask = batch["attention_mask"].to(self.device)

    # Créer des labels (décalés d'une position)
    labels = input_ids.clone()
    labels[:, :-1] = input_ids[:, 1:]
    labels[:, -1] = -100 # Ignorer le dernier token

    # Forward pass
    outputs = self.model(input_ids, attention_mask)
    logits = outputs["prediction_logits"]

    # Calculer la perte
    loss = F.cross_entropy(
        logits.reshape(-1, logits.size(-1)),
        labels.reshape(-1),
        ignore_index=-100,
        reduction="mean"
    )

    losses.append(loss.item())

avg_loss = np.mean(losses) if losses else None

# Enregistrer les résultats
config_name = f"block{block_size}_top{top_k}"
results[config_name] = {
    "block_size": block_size,
    "top_k": top_k,
    "time": end_time - start_time,
    "memory": peak_mem / (1024 * 1024) if peak_mem is not None else None, # MB
    "loss": avg_loss,
    "perplexity": np.exp(avg_loss) if avg_loss is not None else None
}

# Restaurer la configuration originale
model_to_test.config["moba_block_size"] = original_block_size
model_to_test.config["moba_top_k"] = original_top_k

# Restaurer les couches originales
if hasattr(model_to_test, "switch_to_hybrid_mode"):
    hybrid_count = model_to_test.config.get("hybrid_layer_count", 3)

# Recréer les couches MoBA avec les paramètres originaux
for i in range(len(model_to_test.layers)):
    if i < (len(model_to_test.layers) - hybrid_count):
        # Cette couche devrait utiliser MoBA
        model_to_test.layers[i].use_moba = True
        model_to_test.layers[i].attention = model_to_test.MoBAAttention(
            hidden_size=model_to_test.config["hidden_size"],
            num_heads=model_to_test.config["num_attention_heads"],
            block_size=original_block_size,
            top_k=original_top_k
        )

return results

def ablation_hybrid_strategy(self, context_length):
    """Étudie l'impact de la stratégie hybride"""
    results = {}

    # Créer un input de la taille souhaitée
    dummy_input = torch.randint(
        0, self.model.config["vocab_size"],
        (1, context_length),
        device=self.device
    )

    model_to_test = self.model

```

```

if hasattr(model_to_test, "module"):
    model_to_test = model_to_test.module

# Vérifier que le modèle supporte le changement de mode
if not hasattr(model_to_test, "switch_to_hybrid_mode"):
    logger.warning("Le modèle ne supporte pas le changement de mode hybride")
    return {"status": "not_supported"}

# Tester différentes configurations hybrides
total_layers = len(model_to_test.layers)
hybrid_counts = [0, 1, 3, 5, 10, total_layers] # Nombre de couches d'attention complète

for hybrid_count in hybrid_counts:
    if hybrid_count > total_layers:
        continue

    logger.info(f"Test de stratégie hybride: {hybrid_count} couches d'attention complète")

    # Appliquer la configuration
    model_to_test.switch_to_hybrid_mode(hybrid_count)

    # Mesurer les performances
    self.model.eval()
    with torch.no_grad():
        # Mesurer le temps et la mémoire
        if torch.cuda.is_available():
            torch.cuda.empty_cache()
            torch.cuda.reset_peak_memory_stats()
            start_mem = torch.cuda.memory_allocated()

        torch.cuda.synchronize()
        start_time = time.time()

        _ = self.model(dummy_input)

        torch.cuda.synchronize()
        end_time = time.time()

        if torch.cuda.is_available():
            peak_mem = torch.cuda.max_memory_allocated() - start_mem
            torch.cuda.empty_cache()
        else:
            peak_mem = None

    # Mesurer la qualité (perte MLM)
    # Créer un dataset pour cette longueur
    dataset = LongContextDataset(
        self.tokenizer,
        max_length=context_length,
        max_samples=5 # Petit échantillon pour l'ablation
    )

    dataloader = DataLoader(
        dataset,
        batch_size=1,
        collate_fn=dataset.collate_fn
    )

    # Mesurer la perte
    losses = []

    for batch in dataloader:
        input_ids = batch["input_ids"].to(self.device)
        attention_mask = batch["attention_mask"].to(self.device)

        # Créer des labels (décalés d'une position)
        labels = input_ids.clone()
        labels[:, :-1] = input_ids[:, 1:]
        labels[:, -1] = -100 # Ignorer le dernier token

        # Forward pass
        outputs = self.model(input_ids, attention_mask)
        logits = outputs["prediction_logits"]

        # Calculer la perte
        loss = F.cross_entropy(
            logits.reshape(-1, logits.size(-1)),
            labels.reshape(-1),
            ignore_index=-100,
            reduction="mean"
        )

        losses.append(loss.item())

    avg_loss = np.mean(losses) if losses else None

# Enregistrer les résultats
config_name = f"hybrid{hybrid_count}"
results[config_name] = {
    "hybrid_count": hybrid_count,
    "moba_count": total_layers - hybrid_count,
    "time": end_time - start_time,
    "memory": peak_mem / (1024 * 1024) if peak_mem is not None else None, # MB
    "loss": avg_loss,

```

```

    "perplexity": np.exp(avg_loss) if avg_loss is not None else None
}

# Restaurer la configuration originale
original_hybrid_count = model_to_test.config.get("hybrid_layer_count", 3)
model_to_test.switch_to_hybrid_mode(original_hybrid_count)

return results

def generate_visualizations(self, results):
    """Génère des visualisations des résultats d'évaluation"""
    logger.info("Génération des visualisations des résultats")

    # Créer le répertoire pour les visualisations
    vis_dir = os.path.join(self.args.output_dir, "visualizations")
    os.makedirs(vis_dir, exist_ok=True)

    # 1. Position-wise Loss
    if "long_context" in results and "position_wise_loss" in results["long_context"]:
        self._visualize_position_loss(results["long_context"]["position_wise_loss"], vis_dir)

    # 2. Trailing Token Loss
    if "long_context" in results and "trailing_token_loss" in results["long_context"]:
        self._visualize_trailing_loss(results["long_context"]["trailing_token_loss"], vis_dir)

    # 3. Needle in a Haystack
    if "long_context" in results and "needle_in_haystack" in results["long_context"]:
        self._visualize_needle_haystack(results["long_context"]["needle_in_haystack"], vis_dir)

    # 4. Efficiency metrics
    if "efficiency" in results:
        self._visualize_efficiency(results["efficiency"], vis_dir)

    # 5. Ablation results
    if "ablation" in results:
        self._visualize_ablation(results["ablation"], vis_dir)

def _visualize_position_loss(self, position_loss_results, vis_dir):
    """Visualise les résultats de perte par position"""
    plt.figure(figsize=(12, 8))

    for context_length, data in position_loss_results.items():
        if "perplexities" not in data:
            continue

        perplexities = data["perplexities"]
        positions = [int(pos.split("-")[0]) for pos in perplexities.keys()]
        ppl_values = list(perplexities.values())

        # Trier par position
        sorted_pairs = sorted(zip(positions, ppl_values))
        sorted_positions, sorted_ppls = zip(*sorted_pairs) if sorted_pairs else ([], [])

        plt.plot(sorted_positions, sorted_ppls, marker='o', label=f"Context {context_length}")

    plt.xlabel("Position (tokens)")
    plt.ylabel("Perplexity")
    plt.title("Perplexity by Position")
    plt.legend()
    plt.grid(True, linestyle='--', alpha=0.7)
    plt.savefig(os.path.join(vis_dir, "position_perplexity.png"), dpi=300, bbox_inches='tight')
    plt.close()

def _visualize_trailing_loss(self, trailing_loss_results, vis_dir):
    """Visualise les résultats de perte sur les tokens finaux"""
    plt.figure(figsize=(10, 6))

    context_lengths = []
    perplexities = []

    for context_length, data in trailing_loss_results.items():
        if "perplexity" not in data or data["perplexity"] is None:
            continue

        context_lengths.append(int(context_length))
        perplexities.append(data["perplexity"])

    # Trier par longueur de contexte
    sorted_pairs = sorted(zip(context_lengths, perplexities))
    sorted_lengths, sorted_ppls = zip(*sorted_pairs) if sorted_pairs else ([], [])

    plt.bar(range(len(sorted_lengths)), sorted_ppls, color='skyblue')
    plt.xticks(range(len(sorted_lengths)), [str(l) for l in sorted_lengths], rotation=45)
    plt.xlabel("Context Length")
    plt.ylabel("Trailing 2K Tokens Perplexity")
    plt.title("Perplexity of Trailing 2K Tokens by Context Length")
    plt.grid(True, axis='y', linestyle='--', alpha=0.7)
    plt.tight_layout()
    plt.savefig(os.path.join(vis_dir, "trailing_perplexity.png"), dpi=300)
    plt.close()

def _visualize_needle_haystack(self, needle_results, vis_dir):
    """Visualise les résultats de Needle in a Haystack"""
    plt.figure(figsize=(12, 8))

```

```

quartiles = ["q1", "q2", "q3", "q4"]
quartile_labels = ["0-25%", "25-50%", "50-75%", "75-100%"]

for context_length, data in needle_results.items():
    success_rates = []
    for q in quartiles:
        if q in data and "success_rate" in data[q] and data[q]["success_rate"] is not None:
            success_rates.append(data[q]["success_rate"] * 100) # En pourcentage
        else:
            success_rates.append(0)

    x = range(len(quartiles))
    plt.plot(x, success_rates, marker='o', label=f"Context {context_length}")

plt.xticks(range(len(quartiles)), quartile_labels)
plt.xlabel("Position Percentile")
plt.ylabel("Success Rate (%)")
plt.title("Needle in a Haystack Success Rate by Position")
plt.legend()
plt.grid(True, linestyle='--', alpha=0.7)
plt.ylim(0, 100)
plt.savefig(os.path.join(vis_dir, "needle_haystack.png"), dpi=300, bbox_inches='tight')
plt.close()

```

```

def _visualize_efficiency(self, efficiency_results, vis_dir):
    """Visualise les métriques d'efficacité"""
    # 1. Visualiser la consommation mémoire
    if "memory_usage" in efficiency_results:
        plt.figure(figsize=(10, 6))

        memory_data = efficiency_results["memory_usage"]
        context_lengths = []
        moba_memory = []
        full_memory = []

        for context_length, data in memory_data.items():
            if "moba_memory" in data and "full_memory" in data:
                context_lengths.append(int(context_length))
                moba_memory.append(data["moba_memory"])
                full_memory.append(data["full_memory"])

        # Trier par longueur de contexte
        sorted_data = sorted(zip(context_lengths, moba_memory, full_memory))
        if sorted_data:
            context_lengths, moba_memory, full_memory = zip(*sorted_data)

            x = range(len(context_lengths))
            width = 0.35

            plt.bar([i - width/2 for i in x], moba_memory, width, label='MOBA')
            plt.bar([i + width/2 for i in x], full_memory, width, label='Full Attention')

            plt.xticks(x, [str(l) for l in context_lengths])
            plt.xlabel("Context Length")
            plt.ylabel("Memory Usage (MB)")
            plt.title("Memory Usage: MOBA vs Full Attention")
            plt.legend()
            plt.grid(True, axis='y', linestyle='--', alpha=0.7)
            plt.tight_layout()
            plt.savefig(os.path.join(vis_dir, "memory_usage.png"), dpi=300)

        plt.close()

    # 2. Visualiser les temps de calcul
    if "computation_times" in efficiency_results:
        plt.figure(figsize=(10, 6))

        time_data = efficiency_results["computation_times"]
        context_lengths = []
        moba_times = []
        full_times = []

        for context_length, data in time_data.items():
            if "moba_time" in data and "full_time" in data:
                context_lengths.append(int(context_length))
                moba_times.append(data["moba_time"])
                full_times.append(data["full_time"])

        # Trier par longueur de contexte
        sorted_data = sorted(zip(context_lengths, moba_times, full_times))
        if sorted_data:
            context_lengths, moba_times, full_times = zip(*sorted_data)

            x = range(len(context_lengths))
            width = 0.35

            plt.bar([i - width/2 for i in x], moba_times, width, label='MOBA')
            plt.bar([i + width/2 for i in x], full_times, width, label='Full Attention')

            plt.xticks(x, [str(l) for l in context_lengths])
            plt.xlabel("Context Length")
            plt.ylabel("Computation Time (s)")
            plt.title("Computation Time: MOBA vs Full Attention")

```

```

plt.legend()
plt.grid(True, axis='y', linestyle='--', alpha=0.7)
plt.tight_layout()
plt.savefig(os.path.join(vis_dir, "computation_time.png"), dpi=300)

plt.close()

# 3. Visualiser le speedup
if "speedup_ratio" in efficiency_results:
    plt.figure(figsize=(10, 6))

    speedup_data = efficiency_results["speedup_ratio"]
    if "per_length" in speedup_data:
        context_lengths = []
        speedups = []

        for context_length, speedup in speedup_data["per_length"].items():
            context_lengths.append(int(context_length))
            speedups.append(speedup)

        # Trier par longueur de contexte
        sorted_data = sorted(zip(context_lengths, speedups))
        if sorted_data:
            context_lengths, speedups = zip(*sorted_data)

        plt.plot(context_lengths, speedups, marker='o', linestyle='-', color='blue')

        plt.xlabel("Context Length")
        plt.ylabel("Speedup Ratio")
        plt.title("MOBA Speedup Ratio vs Full Attention")
        plt.grid(True, linestyle='--', alpha=0.7)
        plt.savefig(os.path.join(vis_dir, "speedup_ratio.png"), dpi=300, bbox_inches='tight')

    plt.close()

def _visualize_ablation(self, ablation_results, vis_dir):
    """Visualise les résultats des études d'ablation"""
    # 1. Visualiser l'impact de la granularité des blocs
    if "block_granularity" in ablation_results:
        plt.figure(figsize=(12, 10))

        block_data = ablation_results["block_granularity"]
        configs = []
        times = []
        memories = []
        perplexities = []

        for config_name, data in block_data.items():
            if "time" in data and "memory" in data and "perplexity" in data:
                configs.append(config_name)
                times.append(data["time"])
                memories.append(data["memory"])
                perplexities.append(data["perplexity"])

        if configs:
            # Sous-figure pour le temps
            plt.subplot(3, 1, 1)
            plt.bar(configs, times, color='skyblue')
            plt.ylabel("Time (s)")
            plt.title("Time by Block Configuration")
            plt.xticks(rotation=45)
            plt.grid(True, axis='y', linestyle='--', alpha=0.7)

            # Sous-figure pour la mémoire
            plt.subplot(3, 1, 2)
            plt.bar(configs, memories, color='lightgreen')
            plt.ylabel("Memory (MB)")
            plt.title("Memory by Block Configuration")
            plt.xticks(rotation=45)
            plt.grid(True, axis='y', linestyle='--', alpha=0.7)

            # Sous-figure pour la perplexité
            plt.subplot(3, 1, 3)
            plt.bar(configs, perplexities, color='salmon')
            plt.ylabel("Perplexity")
            plt.title("Perplexity by Block Configuration")
            plt.xticks(rotation=45)
            plt.grid(True, axis='y', linestyle='--', alpha=0.7)

        plt.tight_layout()
        plt.savefig(os.path.join(vis_dir, "block_granularity.png"), dpi=300)

    plt.close()

# 2. Visualiser l'impact de la stratégie hybride
if "hybrid_strategy" in ablation_results:
    plt.figure(figsize=(12, 10))

    hybrid_data = ablation_results["hybrid_strategy"]
    configs = []
    times = []
    memories = []
    perplexities = []

```



```

for config_name, data in hybrid_data.items():
    if "time" in data and "memory" in data and "perplexity" in data:
        configs.append(config_name)
        times.append(data["time"])
        memories.append(data["memory"])
        perplexities.append(data["perplexity"])

if configs:
    # Sous-figure pour le temps
    plt.subplot(3, 1, 1)
    plt.bar(configs, times, color='skyblue')
    plt.ylabel("Time (s)")
    plt.title("Time by Hybrid Configuration")
    plt.xticks(rotation=45)
    plt.grid(True, axis='y', linestyle='--', alpha=0.7)

    # Sous-figure pour la mémoire
    plt.subplot(3, 1, 2)
    plt.bar(configs, memories, color='lightgreen')
    plt.ylabel("Memory (MB)")
    plt.title("Memory by Hybrid Configuration")
    plt.xticks(rotation=45)
    plt.grid(True, axis='y', linestyle='--', alpha=0.7)

    # Sous-figure pour la perplexité
    plt.subplot(3, 1, 3)
    plt.bar(configs, perplexities, color='salmon')
    plt.ylabel("Perplexity")
    plt.title("Perplexity by Hybrid Configuration")
    plt.xticks(rotation=45)
    plt.grid(True, axis='y', linestyle='--', alpha=0.7)

    plt.tight_layout()
    plt.savefig(os.path.join(vis_dir, "hybrid_strategy.png"), dpi=300)

plt.close()

def main():
    """Fonction principale pour exécuter l'évaluation"""
    parser = argparse.ArgumentParser(description="Évaluation de NeoBERT-MOBA")
    parser.add_argument("--model_path", type=str, required=True, help="Chemin vers le checkpoint du modèle")
    parser.add_argument("--config_path", type=str, default=None, help="Chemin vers la configuration du modèle")
    parser.add_argument("--tokenizer_path", type=str, default=None, help="Chemin vers le tokenizer")
    parser.add_argument("--output_dir", type=str, default="./evaluation_results", help="Répertoire de sortie")

    parser.add_argument("--run_glue", action="store_true", help="Exécuter l'évaluation GLUE")
    parser.add_argument("--run_mteb", action="store_true", help="Exécuter l'évaluation MTEB")
    parser.add_argument("--run_long_context", action="store_true", help="Exécuter les évaluations de contexte long")
    parser.add_argument("--run_efficiency", action="store_true", help="Exécuter les évaluations d'efficacité")
    parser.add_argument("--run_ablation", action="store_true", help="Exécuter les études d'ablation")

    parser.add_argument("--context_lengths", type=int, nargs="+", default=[2048, 4096, 8192, 16384, 32768], help="Longueurs de contexte à évaluer")
    parser.add_argument("--max_samples", type=int, default=100, help="Nombre maximum d'échantillons")
    parser.add_argument("--batch_size", type=int, default=1, help="Taille du batch")

    parser.add_argument("--device", type=str, default="cuda" if torch.cuda.is_available() else "cpu", help="Device à utiliser")

    args = parser.parse_args()

    # Convertir les arguments en EvaluationArguments
    eval_args = EvaluationArguments(
        model_path=args.model_path,
        config_path=args.config_path,
        tokenizer_path=args.tokenizer_path,
        output_dir=args.output_dir,

        run_glue=args.run_glue,
        run_mteb=args.run_mteb,
        run_long_context=args.run_long_context,
        run_efficiency=args.run_efficiency,
        run_ablation=args.run_ablation,

        context_lengths=args.context_lengths,
        max_samples=args.max_samples,
        batch_size=args.batch_size,

        device=args.device,
    )

    # Créer l'évaluateur
    evaluator = NeoBERTMoBAEvaluator(eval_args)

    # Exécuter l'évaluation
    results = evaluator.evaluate_all()

    logger.info(f"Évaluation terminée. Résultats sauvegardés dans {args.output_dir}")

if __name__ == "__main__":
    main()

```

Ce code fournit un framework complet pour évaluer les performances du modèle NeoBERT-MOBA, en se concentrant particulièrement sur ses capacités à traiter des contextes longs. Voici les principales fonctionnalités:

1. Datasets d'évaluation spécialisés

Le code implémente trois datasets personnalisés pour l'évaluation des contextes longs:

- **LongContextDataset**: Évalue la perte du modèle de langage à différentes positions dans de longs contextes
- **NeedleInHaystackDataset**: Teste la capacité du modèle à localiser et utiliser des informations spécifiques dans un long contexte
- **RULERDataset**: Évalue la capacité de raisonnement à longue distance sur des problèmes structurés

2. Métriques d'évaluation

Le protocole évalue plusieurs dimensions clés:

2.1. Évaluation des performances générales

- Support pour GLUE et MTEB (à implémenter selon les besoins)

2.2. Évaluation des capacités de contexte long

- **Position-wise LM loss**: Mesure la perte à différentes positions dans le contexte
- **Trailing token loss**: Évalue la perte sur les 2K derniers tokens des séquences longues
- **Needle in a Haystack**: Teste la capacité à retrouver des informations dans un long contexte
- **RULER**: Évalue la compréhension et le raisonnement à longue distance

2.3. Évaluation d'efficacité computationnelle

- **Consommation mémoire**: Compare MOBA vs attention complète
- **Temps de calcul**: Mesure les différences de performance
- **Ratio d'accélération**: Quantifie le gain de MOBA par rapport à l'attention complète

3. Études d'ablation

Le code permet de comparer différentes configurations du modèle:

- **Granularité des blocs**: Teste différentes tailles de blocs et nombres de blocs sélectionnés
- **Stratégie hybride**: Évalue l'impact de différentes combinaisons de couches MOBA et d'attention complète

4. Visualisations des résultats

Le code génère automatiquement des visualisations claires:

- Graphiques de perplexité par position
- Comparaisons de performance par longueur de contexte
- Visualisations pour les études d'ablation

5. Architecture flexible

L'architecture du code est conçue pour être modulaire et extensible:

- Paramétrable via ligne de commande
- Exécution sélective des évaluations
- Sauvegarde structurée des résultats

Utilisation du protocole

Pour exécuter l'évaluation complète:

```
```bash
python evaluation.py --model_path ./checkpoints/neobert_moba_final \
 --run_long_context --run_efficiency --run_ablation \
 --context_lengths 2048 4096 8192 16384 32768 \
 --output_dir ./evaluation_results
```
```

Ce protocole d'évaluation complet permet d'analyser en profondeur les performances de l'architecture NeoBERT-MOBA, particulièrement sa capacité à gérer efficacement les longs contextes grâce au mécanisme MOBA, et de comparer différentes configurations pour déterminer le meilleur compromis entre efficacité et qualité.

USER

écrit le code pour :

5. Expériences d'ablation

5.1. Impact de la granularité des blocs

Entraîner des versions avec différentes configurations de blocs en maintenant une sparsité constante:

- 8 blocs, top-k=2
- 16 blocs, top-k=4
- 32 blocs, top-k=8
- 64 blocs, top-k=16
- 128 blocs, top-k=32

5.2. Impact de la stratégie hybride

Comparer les performances de:

- MOBA complet (toutes les couches)
- Hybride avec 1, 3, 5, 10 couches d'attention complète
- Attention complète

5.3. Impact du choix des métriques pour le routage

Comparer différentes méthodes de calcul des scores d'affinité:

- Mean pooling (standard)
- Max pooling
- Combinaison min/max pooling
- Attention basée sur un token représentatif

6. Suite du développement
Suite aux résultats des évaluations, prévoir:

Ajustement des hyperparamètres MOBA
Optimisation de l'implémentation pour différents matériels
Extension à des contextes encore plus longs (>1M tokens)
Adaptation à d'autres langues

BOT

Thoughts

Vous me demandez de créer le code pour effectuer des expériences d'ablation sur le modèle NeoBERT-MOBA et pour planifier la suite du développement. Ces expériences visent à étudier l'impact de différents aspects du modèle: la granularité des blocs, la stratégie hybride, et les métriques de routage.

Je vais diviser mon approche en plusieurs parties:

1. Code pour l'étude de l'impact de la granularité des blocs
2. Code pour l'étude de l'impact de la stratégie hybride
3. Code pour l'étude de l'impact des métriques de routage
4. Plan pour la suite du développement

Commençons par le code pour les expériences d'ablation.

1. Impact de la granularité des blocs

Cette expérience consiste à tester différentes configurations de blocs tout en maintenant une sparsité constante:

- 8 blocs, top-k=2
- 16 blocs, top-k=4
- 32 blocs, top-k=8
- 64 blocs, top-k=16
- 128 blocs, top-k=32

Ces configurations maintiennent toutes un ratio de sparsité de 75% (en sélectionnant 25% des blocs disponibles).

```
import os
import json
import torch
import logging
import argparse
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm
from datetime import datetime
from torch.utils.data import DataLoader
from model_architecture import NeoBERTMoBA, create_neobert_moba_model
from data_preparation import DataPreparation

# Configuration du logging
logging.basicConfig(
    format="%(asctime)s - %(levelname)s - %(name)s - %(message)s",
    datefmt="%m/%d/%Y %H:%M:%S",
    level=logging.INFO,
)
logger = logging.getLogger(__name__)

def run_block_granularity_ablation(
    base_model_path,
    output_dir,
    context_length=32768,
    batch_size=1,
    max_samples=5,
    device="cuda",
):
    """
    Exécute l'étude d'ablation sur la granularité des blocs MOBA.
    Teste différentes configurations avec une sparsité constante.
    """
    os.makedirs(output_dir, exist_ok=True)

    # Configurations des expériences
    configurations = [
        {"block_size": context_length // 8, "top_k": 2, "name": "block8_top2"},
        {"block_size": context_length // 16, "top_k": 4, "name": "block16_top4"},
        {"block_size": context_length // 32, "top_k": 8, "name": "block32_top8"},
        {"block_size": context_length // 64, "top_k": 16, "name": "block64_top16"},
        {"block_size": context_length // 128, "top_k": 32, "name": "block128_top32"},
    ]

    # Charger la configuration du modèle
    with open(os.path.join(base_model_path, "config.json"), "r") as f:
        model_config = json.load(f)

    # Résultats
    results = {}

    # Préparer les données d'évaluation
    data_prep = DataPreparation(max_length=context_length)
    eval_dataset = data_prep.create_evaluation_dataset(
        dataset_name="pg19",
        split="test",
        max_samples=max_samples
    )
```

```

eval_dataloader = DataLoader(
    eval_dataset,
    batch_size=batch_size,
    collate_fn=eval_dataset.collate_fn
)

# Tester chaque configuration
for config in configurations:
    logger.info(f"Évaluation de la configuration: {config['name']}")

    # Mettre à jour la configuration du modèle
    model_config["moba_block_size"] = config["block_size"]
    model_config["moba_top_k"] = config["top_k"]

    # Charger le modèle avec la configuration modifiée
    model = create_neobert_moba_model(model_config)
    model.load_state_dict(torch.load(os.path.join(base_model_path, "pytorch_model.bin")))
    model = model.to(device)
    model.eval()

    # Mesurer les performances (temps, mémoire, perte)
    time_measurements = []
    memory_usage = []
    losses = []

    for batch in tqdm(eval_dataloader, desc=f"Évaluation {config['name']}"):
        input_ids = batch["input_ids"].to(device)
        attention_mask = batch["attention_mask"].to(device)

        # Créer des labels (décalés d'une position)
        labels = input_ids.clone()
        labels[:, :-1] = input_ids[:, 1:]
        labels[:, -1] = -100 # Ignorer le dernier token

        # Mesurer le temps et la mémoire
        if torch.cuda.is_available():
            torch.cuda.empty_cache()
            torch.cuda.reset_peak_memory_stats()
            start_mem = torch.cuda.memory_allocated()

        torch.cuda.synchronize()
        start_time = time.time()

        with torch.no_grad():
            outputs = model(input_ids, attention_mask)
            logits = outputs["prediction_logits"]

            # Calculer la perte
            loss = torch.nn.functional.cross_entropy(
                logits.reshape(-1, logits.size(-1)),
                labels.reshape(-1),
                ignore_index=-100,
                reduction="mean"
            )

        torch.cuda.synchronize()
        end_time = time.time()

        if torch.cuda.is_available():
            peak_mem = torch.cuda.max_memory_allocated() - start_mem
            memory_usage.append(peak_mem / (1024 * 1024)) # MB

        time_measurements.append(end_time - start_time)
        losses.append(loss.item())

    # Calculer les moyennes
    avg_time = np.mean(time_measurements)
    avg_memory = np.mean(memory_usage) if memory_usage else None
    avg_loss = np.mean(losses)
    perplexity = np.exp(avg_loss)

    # Stocker les résultats
    results[config['name']] = {
        "block_size": config["block_size"],
        "top_k": config["top_k"],
        "avg_time": avg_time,
        "avg_memory": avg_memory,
        "avg_loss": avg_loss,
        "perplexity": perplexity
    }

    logger.info(f"Résultats pour {config['name']}:")
    logger.info(f" Temps moyen: {avg_time:.4f} s")
    logger.info(f" Mémoire moyenne: {avg_memory:.2f} MB" if avg_memory else " Mémoire moyenne: N/A")
    logger.info(f" Perte moyenne: {avg_loss:.4f}")
    logger.info(f" Perplexité: {perplexity:.4f}")

# Sauvegarder les résultats
with open(os.path.join(output_dir, "block_granularity_results.json"), "w") as f:
    json.dump(results, f, indent=2)

# Créer des visualisations
create_block_granularity_visualizations(results, output_dir)

```

```

return results

def create_block_granularity_visualizations(results, output_dir):
    """Crée des visualisations pour les résultats de l'ablation sur la granularité des blocs"""
    plt.figure(figsize=(15, 10))

    # Extraire les données
    configs = list(results.keys())
    times = [results[c]["avg_time"] for c in configs]
    memories = [results[c]["avg_memory"] for c in configs if results[c]["avg_memory"] is not None]
    losses = [results[c]["avg_loss"] for c in configs]
    perplexities = [results[c]["perplexity"] for c in configs]

    # Tableau pour les étiquettes de l'axe X plus lisibles
    x_labels = [f"{results[c]['block_size']}/{results[c]['top_k']}" for c in configs]

    # Sous-graphique pour le temps
    plt.subplot(2, 2, 1)
    plt.bar(configs, times, color='skyblue')
    plt.ylabel("Temps (s)")
    plt.title("Temps d'inférence par configuration de bloc")
    plt.xticks(rotation=45)
    plt.grid(True, axis='y', linestyle='--', alpha=0.7)

    # Sous-graphique pour la mémoire
    if memories:
        plt.subplot(2, 2, 2)
        plt.bar(configs[:len(memories)], memories, color='lightgreen')
        plt.ylabel("Mémoire (MB)")
        plt.title("Utilisation mémoire par configuration de bloc")
        plt.xticks(rotation=45)
        plt.grid(True, axis='y', linestyle='--', alpha=0.7)

    # Sous-graphique pour la perte
    plt.subplot(2, 2, 3)
    plt.bar(configs, losses, color='salmon')
    plt.ylabel("Perte")
    plt.title("Perte par configuration de bloc")
    plt.xticks(rotation=45)
    plt.grid(True, axis='y', linestyle='--', alpha=0.7)

    # Sous-graphique pour la perplexité
    plt.subplot(2, 2, 4)
    plt.bar(configs, perplexities, color='plum')
    plt.ylabel("Perplexité")
    plt.title("Perplexité par configuration de bloc")
    plt.xticks(rotation=45)
    plt.grid(True, axis='y', linestyle='--', alpha=0.7)

    plt.tight_layout()
    plt.savefig(os.path.join(output_dir, "block_granularity_comparison.png"), dpi=300)

```

2. Impact de la stratégie hybride

Cette expérience compare les performances avec différentes stratégies hybrides:

- MOBA complet (toutes les couches)
- Hybride avec 1, 3, 5, 10 couches d'attention complète
- Attention complète

```

def run_hybrid_strategy_ablation(
    base_model_path,
    output_dir,
    context_length=32768,
    batch_size=1,
    max_samples=5,
    device="cuda",
):
    """
    Exécute l'étude d'ablation sur la stratégie hybride.
    Compare différents nombres de couches d'attention complète.
    """
    os.makedirs(output_dir, exist_ok=True)

    # Charger la configuration du modèle
    with open(os.path.join(base_model_path, "config.json"), "r") as f:
        model_config = json.load(f)

    # Nombre total de couches
    total_layers = model_config["num_hidden_layers"]

    # Configurations des expériences (nombre de couches d'attention complète)
    hybrid_configurations = [
        {"hybrid_count": 0, "name": "full_moba"},
        {"hybrid_count": 1, "name": "hybrid_1"},
        {"hybrid_count": 3, "name": "hybrid_3"},
        {"hybrid_count": 5, "name": "hybrid_5"},
        {"hybrid_count": 10, "name": "hybrid_10"},
        {"hybrid_count": total_layers, "name": "full_attention"}
    ]

    # Résultats
    results = {}

```

```

# Préparer les données d'évaluation
data_prep = DataPreparation(max_length=context_length)
eval_dataset = data_prep.create_evaluation_dataset(
    dataset_name="pg19",
    split="test",
    max_samples=max_samples
)
eval_dataloader = DataLoader(
    eval_dataset,
    batch_size=batch_size,
    collate_fn=eval_dataset.collate_fn
)

# Tester chaque configuration
for config in hybrid_configurations:
    logger.info(f"Évaluation de la configuration: {config['name']}")

    # Mettre à jour la configuration du modèle
    model_config["hybrid_layer_count"] = config["hybrid_count"]

    # Charger le modèle avec la configuration modifiée
    model = create_neobert_moba_model(model_config)
    model.load_state_dict(torch.load(os.path.join(base_model_path, "pytorch_model.bin")))
    model = model.to(device)
    model.eval()

    # Appliquer la stratégie hybride
    model.switch_to_hybrid_mode(config["hybrid_count"])

    # Mesurer les performances (temps, mémoire, perte)
    time_measurements = []
    memory_usage = []
    losses = []

    for batch in tqdm(eval_dataloader, desc=f"Évaluation {config['name']}"):
        input_ids = batch["input_ids"].to(device)
        attention_mask = batch["attention_mask"].to(device)

        # Créer des labels (décalés d'une position)
        labels = input_ids.clone()
        labels[:, :-1] = input_ids[:, 1:]
        labels[:, -1] = -100 # Ignorer le dernier token

        # Mesurer le temps et la mémoire
        if torch.cuda.is_available():
            torch.cuda.empty_cache()
            torch.cuda.reset_peak_memory_stats()
            start_mem = torch.cuda.memory_allocated()

        torch.cuda.synchronize()
        start_time = time.time()

        with torch.no_grad():
            outputs = model(input_ids, attention_mask)
            logits = outputs["prediction_logits"]

            # Calculer la perte
            loss = torch.nn.functional.cross_entropy(
                logits.reshape(-1, logits.size(-1)),
                labels.reshape(-1),
                ignore_index=-100,
                reduction="mean"
            )

        torch.cuda.synchronize()
        end_time = time.time()

        if torch.cuda.is_available():
            peak_mem = torch.cuda.max_memory_allocated() - start_mem
            memory_usage.append(peak_mem / (1024 * 1024)) # MB

        time_measurements.append(end_time - start_time)
        losses.append(loss.item())

    # Calculer les moyennes
    avg_time = np.mean(time_measurements)
    avg_memory = np.mean(memory_usage) if memory_usage else None
    avg_loss = np.mean(losses)
    perplexity = np.exp(avg_loss)

    # Stocker les résultats
    results[config['name']] = {
        "hybrid_count": config["hybrid_count"],
        "moba_count": total_layers - config["hybrid_count"],
        "avg_time": avg_time,
        "avg_memory": avg_memory,
        "avg_loss": avg_loss,
        "perplexity": perplexity
    }

    logger.info(f"Résultats pour {config['name']}:")
    logger.info(f" Temps moyen: {avg_time:.4f} s")
    logger.info(f" Mémoire moyenne: {avg_memory:.2f} MB" if avg_memory else " Mémoire moyenne: N/A")

```

```

logger.info(f" Perte moyenne: {avg_loss:.4f}")
logger.info(f" Perplexité: {perplexity:.4f}")

# Sauvegarder les résultats
with open(os.path.join(output_dir, "hybrid_strategy_results.json"), "w") as f:
    json.dump(results, f, indent=2)

# Créer des visualisations
create_hybrid_strategy_visualizations(results, output_dir, total_layers)

return results

def create_hybrid_strategy_visualizations(results, output_dir, total_layers):
    """Crée des visualisations pour les résultats de l'ablation sur la stratégie hybride"""
    plt.figure(figsize=(15, 10))

    # Extraire les données
    configs = list(results.keys())
    hybrid_counts = [results[c]["hybrid_count"] for c in configs]
    times = [results[c]["avg_time"] for c in configs]
    memories = [results[c]["avg_memory"] for c in configs if results[c]["avg_memory"] is not None]
    losses = [results[c]["avg_loss"] for c in configs]
    perplexities = [results[c]["perplexity"] for c in configs]

    # Calculer le % de couches MOBA
    moba_percentages = [(total_layers - hc) / total_layers * 100 for hc in hybrid_counts]

    # Sous-graphique pour le temps par rapport au % de MOBA
    plt.subplot(2, 2, 1)
    plt.plot(moba_percentages, times, marker='o', linestyle='-', color='blue')
    plt.xlabel("% de couches MOBA")
    plt.ylabel("Temps (s)")
    plt.title("Temps d'inférence vs % de couches MOBA")
    plt.grid(True, linestyle='--', alpha=0.7)

    # Sous-graphique pour la mémoire par rapport au % de MOBA
    if memories:
        plt.subplot(2, 2, 2)
        plt.plot(moba_percentages[:len(memories)], memories, marker='o', linestyle='-', color='green')
        plt.xlabel("% de couches MOBA")
        plt.ylabel("Mémoire (MB)")
        plt.title("Utilisation mémoire vs % de couches MOBA")
        plt.grid(True, linestyle='--', alpha=0.7)

    # Sous-graphique pour la perte par rapport au % de MOBA
    plt.subplot(2, 2, 3)
    plt.plot(moba_percentages, losses, marker='o', linestyle='-', color='red')
    plt.xlabel("% de couches MOBA")
    plt.ylabel("Perte")
    plt.title("Perte vs % de couches MOBA")
    plt.grid(True, linestyle='--', alpha=0.7)

    # Sous-graphique pour la perplexité par rapport au % de MOBA
    plt.subplot(2, 2, 4)
    plt.plot(moba_percentages, perplexities, marker='o', linestyle='-', color='purple')
    plt.xlabel("% de couches MOBA")
    plt.ylabel("Perplexité")
    plt.title("Perplexité vs % de couches MOBA")
    plt.grid(True, linestyle='--', alpha=0.7)

    plt.tight_layout()
    plt.savefig(os.path.join(output_dir, "hybrid_strategy_comparison.png"), dpi=300)

```

3. Impact du choix des métriques pour le routage

Cette expérience compare différentes méthodes de calcul des scores d'affinité pour le routage:

- Mean pooling (standard)
- Max pooling
- Combinaison min/max pooling
- Attention basée sur un token représentatif

Pour cela, nous devons modifier le code de la classe MoBAAttention pour supporter différentes métriques de routage. Puis, nous testons chaque variante.

D'abord, modifions la classe MoBAAttention pour supporter différentes métriques de routage

```

class MoBAAttention(nn.Module):
    """Mixture of Block Attention (MoBA) avec différentes métriques de routage"""
    def __init__(
        self,
        hidden_size: int,
        num_heads: int,
        block_size: int = 512,
        top_k: int = 3,
        routing_metric: str = "mean_pooling", # Nouvelle option
    ):
        super().__init__()
        self.hidden_size = hidden_size
        self.num_heads = num_heads
        self.head_dim = hidden_size // num_heads
        self.block_size = block_size
        self.top_k = top_k
        self.routing_metric = routing_metric # Nouvelle option

```

```

self.q_proj = nn.Linear(hidden_size, hidden_size, bias=False)
self.k_proj = nn.Linear(hidden_size, hidden_size, bias=False)
self.v_proj = nn.Linear(hidden_size, hidden_size, bias=False)
self.o_proj = nn.Linear(hidden_size, hidden_size, bias=False)

self.rotary = RotaryEmbedding(self.head_dim)

self.scale = 1.0 / math.sqrt(self.head_dim)

def _compute_block_representation(self, keys, block_indices):
    """
    Calcule la représentation d'un bloc selon la métrique de routage choisie.

    Args:
        keys: Tenseur des clés [batch_size, num_heads, seq_len, head_dim]
        block_indices: Indices des tokens dans ce bloc

    Returns:
        Représentation du bloc [batch_size, num_heads, 1, head_dim]
    """
    if self.routing_metric == "mean_pooling":
        # Moyenne standard des clés dans le bloc
        return keys[:, :, block_indices].mean(dim=2, keepdim=True)

    elif self.routing_metric == "max_pooling":
        # Maximum des valeurs des clés dans le bloc
        return keys[:, :, block_indices].max(dim=2, keepdim=True)[0]

    elif self.routing_metric == "min_max_pooling":
        # Combinaison du min et max pooling
        max_values = keys[:, :, block_indices].max(dim=2, keepdim=True)[0]
        min_values = keys[:, :, block_indices].min(dim=2, keepdim=True)[0]
        return (max_values + min_values) / 2

    elif self.routing_metric == "representative_token":
        # Utiliser le premier token du bloc comme représentatif
        # On pourrait aussi utiliser le token du milieu ou d'autres stratégies
        if len(block_indices) > 0:
            return keys[:, :, block_indices[0]:block_indices[0]+1]
        else:
            # Bloc vide, retourner des zéros
            batch_size = keys.size(0)
            return torch.zeros(batch_size, self.num_heads, 1, self.head_dim, device=keys.device)

    else:
        raise ValueError(f"Metric de routage non supportée: {self.routing_metric}")

def forward(self, hidden_states, attention_mask=None, past_key_value=None):
    # Le reste du code de la méthode forward reste identique, mais nous utilisons
    # _compute_block_representation au lieu de mean_pool directement

    # [Code existant pour la projection et la préparation des tenseurs]

    # Lors du calcul des scores d'affinité entre la requête et les blocs:
    block_means = []
    for i in range(num_blocks):
        start_idx = i * self.block_size
        end_idx = min(start_idx + self.block_size, seq_len)
        if end_idx > start_idx: # Bloc non vide
            block_indices = list(range(start_idx, end_idx))
            block_rep = self._compute_block_representation(k, block_indices)
            block_means.append(block_rep)

    # [Reste du code existant]

# Maintenant, créons le code pour l'expérience d'ablation sur les métriques de routage

def run_routing_metrics_ablation(
    base_model_path,
    output_dir,
    context_length=32768,
    batch_size=1,
    max_samples=5,
    device="cuda",
):
    """
    Exécute l'étude d'ablation sur les métriques de routage pour MoBA.
    Compare différentes méthodes de calcul des scores d'affinité.
    """
    os.makedirs(output_dir, exist_ok=True)

    # Charger la configuration du modèle
    with open(os.path.join(base_model_path, "config.json"), "r") as f:
        model_config = json.load(f)

    # Configurations des expériences (métriques de routage)
    routing_metrics = [
        "mean_pooling",
        "max_pooling",
        "min_max_pooling",
        "representative_token"
    ]

```



```

# Résultats
results = {}

# Préparer les données d'évaluation
data_prep = DataPreparation(max_length=context_length)
eval_dataset = data_prep.create_evaluation_dataset(
    dataset_name="pg19",
    split="test",
    max_samples=max_samples
)
eval_dataloader = DataLoader(
    eval_dataset,
    batch_size=batch_size,
    collate_fn=eval_dataset.collate_fn
)

# Classe pour créer un modèle avec une métrique de routage spécifique
class NeoBERTMoBAWithCustomRouting(NeoBERTMoBA):
    def __init__(self, *args, routing_metric="mean_pooling", **kwargs):
        super().__init__(*args, **kwargs)
        self.routing_metric = routing_metric

    # Recréer les couches d'attention avec la métrique de routage personnalisée
    hybrid_count = self.config.get("hybrid_layer_count", 3)

    for i in range(len(self.layers)):
        if i < (len(self.layers) - hybrid_count):
            # Cette couche devrait utiliser MoBA
            self.layers[i].use_moba = True
            self.layers[i].attention = MoBAAttention(
                hidden_size=self.config["hidden_size"],
                num_heads=self.config["num_attention_heads"],
                block_size=self.config["moba_block_size"],
                top_k=self.config["moba_top_k"],
                routing_metric=routing_metric
            )

# Tester chaque métrique de routage
for metric in routing_metrics:
    logger.info(f"Évaluation de la métrique de routage: {metric}")

    # Créer le modèle avec la métrique de routage personnalisée
    model = NeoBERTMoBAWithCustomRouting(
        vocab_size=model_config["vocab_size"],
        hidden_size=model_config["hidden_size"],
        num_hidden_layers=model_config["num_hidden_layers"],
        num_attention_heads=model_config["num_attention_heads"],
        max_position_embeddings=model_config["max_position_embeddings"],
        moba_block_size=model_config["moba_block_size"],
        moba_top_k=model_config["moba_top_k"],
        hybrid_layer_count=model_config["hybrid_layer_count"],
        routing_metric=metric
    )

    # Charger les poids du modèle
    model.load_state_dict(torch.load(os.path.join(base_model_path, "pytorch_model.bin")))
    model = model.to(device)
    model.eval()

    # Mesurer les performances (temps, perte)
    time_measurements = []
    losses = []

    for batch in tqdm(eval_dataloader, desc=f"Évaluation {metric}"):
        input_ids = batch["input_ids"].to(device)
        attention_mask = batch["attention_mask"].to(device)

        # Créer des labels (décalés d'une position)
        labels = input_ids.clone()
        labels[:, :-1] = input_ids[:, 1:]
        labels[:, -1] = -100 # Ignorer le dernier token

        torch.cuda.synchronize()
        start_time = time.time()

        with torch.no_grad():
            outputs = model(input_ids, attention_mask)
            logits = outputs["prediction_logits"]

            # Calculer la perte
            loss = torch.nn.functional.cross_entropy(
                logits.reshape(-1, logits.size(-1)),
                labels.reshape(-1),
                ignore_index=-100,
                reduction="mean"
            )

        torch.cuda.synchronize()
        end_time = time.time()

        time_measurements.append(end_time - start_time)
        losses.append(loss.item())

```

```

# Calculer les moyennes
avg_time = np.mean(time_measurements)
avg_loss = np.mean(losses)
perplexity = np.exp(avg_loss)

# Stocker les résultats
results[metric] = {
    "avg_time": avg_time,
    "avg_loss": avg_loss,
    "perplexity": perplexity
}

logger.info(f"Résultats pour {metric}:")
logger.info(f" Temps moyen: {avg_time:.4f} s")
logger.info(f" Perte moyenne: {avg_loss:.4f}")
logger.info(f" Perplexité: {perplexity:.4f}")

# Sauvegarder les résultats
with open(os.path.join(output_dir, "routing_metrics_results.json"), "w") as f:
    json.dump(results, f, indent=2)

# Créer des visualisations
create_routing_metrics_visualizations(results, output_dir)

return results

def create_routing_metrics_visualizations(results, output_dir):
    """Crée des visualisations pour les résultats de l'ablation sur les métriques de routage"""
    plt.figure(figsize=(12, 6))

    # Extraire les données
    metrics = list(results.keys())
    times = [results[m]["avg_time"] for m in metrics]
    losses = [results[m]["avg_loss"] for m in metrics]
    perplexities = [results[m]["perplexity"] for m in metrics]

    # Graphique pour le temps
    plt.subplot(1, 2, 1)
    plt.bar(metrics, times, color='skyblue')
    plt.ylabel("Temps (s)")
    plt.title("Temps d'inférence par métrique de routage")
    plt.xticks(rotation=45)
    plt.grid(True, axis='y', linestyle='--', alpha=0.7)

    # Graphique pour la perplexité
    plt.subplot(1, 2, 2)
    plt.bar(metrics, perplexities, color='salmon')
    plt.ylabel("Perplexité")
    plt.title("Perplexité par métrique de routage")
    plt.xticks(rotation=45)
    plt.grid(True, axis='y', linestyle='--', alpha=0.7)

    plt.tight_layout()
    plt.savefig(os.path.join(output_dir, "routing_metrics_comparison.png"), dpi=300)

```

4. Fonction principale pour exécuter toutes les expériences

Maintenant, créons une fonction principale qui exécute toutes ces expériences d'ablation :

```

def main():
    """Fonction principale pour exécuter toutes les expériences d'ablation"""
    parser = argparse.ArgumentParser(description="Expériences d'ablation pour NeoBERT-MOBA")
    parser.add_argument("--model_path", type=str, required=True, help="Chemin vers le checkpoint du modèle")
    parser.add_argument("--output_dir", type=str, default="./ablation_results", help="Répertoire de sortie")
    parser.add_argument("--context_length", type=int, default=32768, help="Longueur du contexte pour l'évaluation")
    parser.add_argument("--max_samples", type=int, default=5, help="Nombre maximum d'échantillons")
    parser.add_argument("--batch_size", type=int, default=1, help="Taille du batch")
    parser.add_argument("--device", type=str, default="cuda" if torch.cuda.is_available() else "cpu", help="Device à utiliser")

    parser.add_argument("--run_granularity", action="store_true", help="Exécuter l'ablation sur la granularité des blocs")
    parser.add_argument("--run_hybrid", action="store_true", help="Exécuter l'ablation sur la stratégie hybride")
    parser.add_argument("--run_routing", action="store_true", help="Exécuter l'ablation sur les métriques de routage")
    parser.add_argument("--run_all", action="store_true", help="Exécuter toutes les ablations")

    args = parser.parse_args()

    # Créer le répertoire de sortie principal
    os.makedirs(args.output_dir, exist_ok=True)

    # Exécuter les expériences sélectionnées
    if args.run_all or args.run_granularity:
        granularity_dir = os.path.join(args.output_dir, "block_granularity")
        run_block_granularity_ablation(
            base_model_path=args.model_path,
            output_dir=granularity_dir,
            context_length=args.context_length,
            batch_size=args.batch_size,
            max_samples=args.max_samples,
            device=args.device
        )

    if args.run_all or args.run_hybrid:

```

```

hybrid_dir = os.path.join(args.output_dir, "hybrid_strategy")
run_hybrid_strategy_ablation(
    base_model_path=args.model_path,
    output_dir=hybrid_dir,
    context_length=args.context_length,
    batch_size=args.batch_size,
    max_samples=args.max_samples,
    device=args.device
)

if args.run_all or args.run_routing:
    routing_dir = os.path.join(args.output_dir, "routing_metrics")
    run_routing_metrics_ablation(
        base_model_path=args.model_path,
        output_dir=routing_dir,
        context_length=args.context_length,
        batch_size=args.batch_size,
        max_samples=args.max_samples,
        device=args.device
    )

logger.info("Expériences d'ablation terminées. Résultats sauvegardés dans " + args.output_dir)

if __name__ == "__main__":
    main()

```

5. Plan pour la suite du développement

Maintenant, préparons une fonction qui génère un rapport pour la suite du développement basé sur les résultats des expériences d'ablation:

```

def generate_development_roadmap(ablation_results_dir, output_file):
    """
    Génère un rapport sur la suite du développement basé sur les résultats d'ablation.

    Args:
        ablation_results_dir: Répertoire contenant les résultats des ablations
        output_file: Chemin du fichier où sauvegarder le rapport
    """
    # Charger les résultats des ablations
    results = {}

    # Charger les résultats de granularité des blocs
    granularity_file = os.path.join(ablation_results_dir, "block_granularity", "block_granularity_results.json")
    if os.path.exists(granularity_file):
        with open(granularity_file, "r") as f:
            results["granularity"] = json.load(f)

    # Charger les résultats de stratégie hybride
    hybrid_file = os.path.join(ablation_results_dir, "hybrid_strategy", "hybrid_strategy_results.json")
    if os.path.exists(hybrid_file):
        with open(hybrid_file, "r") as f:
            results["hybrid"] = json.load(f)

    # Charger les résultats des métriques de routage
    routing_file = os.path.join(ablation_results_dir, "routing_metrics", "routing_metrics_results.json")
    if os.path.exists(routing_file):
        with open(routing_file, "r") as f:
            results["routing"] = json.load(f)

    # Analyser les résultats pour des recommandations
    recommendations = {}

    # Analyser la granularité des blocs
    if "granularity" in results:
        granularity_data = results["granularity"]

        # Trouver la configuration avec la meilleure perplexité
        best_perplexity = float('inf')
        best_config = None

        for config, data in granularity_data.items():
            if data["perplexity"] < best_perplexity:
                best_perplexity = data["perplexity"]
                best_config = config

        if best_config:
            recommendations["block_size"] = granularity_data[best_config]["block_size"]
            recommendations["top_k"] = granularity_data[best_config]["top_k"]

    # Analyser la stratégie hybride
    if "hybrid" in results:
        hybrid_data = results["hybrid"]

        # Trouver le meilleur compromis entre perplexité et vitesse
        configs = []
        for config, data in hybrid_data.items():
            configs.append({
                "name": config,
                "hybrid_count": data["hybrid_count"],
                "perplexity": data["perplexity"],
                "time": data["avg_time"]
            })

```

```

# Trier par perplexité
configs.sort(key=lambda x: x["perplexity"])

# Prendre la configuration avec la meilleure perplexité dans les 3 premières
# qui a aussi un bon temps d'exécution
best_hybrid = configs[0]
for i in range(1, min(3, len(configs))):
    if configs[i]["time"] < best_hybrid["time"] * 0.8: # 20% plus rapide
        best_hybrid = configs[i]

recommendations["hybrid_count"] = best_hybrid["hybrid_count"]

# Analyser les métriques de routage
if "routing" in results:
    routing_data = results["routing"]

    # Trouver la métrique avec la meilleure perplexité
    best_perplexity = float('inf')
    best_metric = None

    for metric, data in routing_data.items():
        if data["perplexity"] < best_perplexity:
            best_perplexity = data["perplexity"]
            best_metric = metric

    if best_metric:
        recommendations["routing_metric"] = best_metric

# Générer le rapport
report = f""# Feuille de route pour le développement futur de NeoBERT-MOBA

## Recommandations basées sur les expériences d'ablation

### 1. Ajustement des hyperparamètres MOBA

Sur la base des expériences d'ablation, les paramètres optimaux recommandés sont:

"""

if "block_size" in recommendations and "top_k" in recommendations:
    report += f"- **Taille de bloc**: {recommendations['block_size']}\n"
    report += f"- **Paramètre top-k**: {recommendations['top_k']}\n"
else:
    report += "- Des tests supplémentaires sont nécessaires pour déterminer la taille de bloc et le paramètre top-k optimaux.\n"

if "hybrid_count" in recommendations:
    report += f"- **Nombre de couches d'attention complète**: {recommendations['hybrid_count']}\n"
else:
    report += "- Des tests supplémentaires sont nécessaires pour déterminer le nombre optimal de couches d'attention complète.\n"

if "routing_metric" in recommendations:
    report += f"- **Métrique de routage recommandée**: {recommendations['routing_metric']}\n"
else:
    report += "- Des tests supplémentaires sont nécessaires pour déterminer la métrique de routage optimale.\n"

report += """
### 2. Optimisation de l'implémentation

Pour améliorer les performances sur différents matériels:

- Optimiser l'implémentation CUDA pour améliorer l'efficacité sur GPU
- Explorer les optimisations pour CPU (AVX, OpenMP)
- Adapter l'implémentation pour hardware spécialisé (TPU, NPU)
- Optimiser la gestion mémoire pour réduire l'empreinte mémoire lors de l'inférence
- Intégrer avec les frameworks d'inférence comme ONNX Runtime, TensorRT

### 3. Extension à des contextes plus longs

Pour atteindre et dépasser 1M tokens:

- Tester des stratégies de mise à l'échelle pour adapter la taille des blocs et le paramètre top-k
- Explorer des techniques de compression mémoire pour stocker l'historique
- Développer une version hiérarchique de MOBA avec plusieurs niveaux de granularité
- Expérimenter avec des techniques d'attention récurrente pour les très longs contextes
- Intégrer avec des méthodes d'indexation ou de recherche pour les contextes extrêmement longs (>10M tokens)

### 4. Adaptation à d'autres langues

Pour améliorer les performances multilingues:

- Tester MOBA avec des tokenizers spécifiques aux langues
- Ajuster la granularité des blocs pour les langues aux caractéristiques différentes
- Évaluer l'impact des différentes longueurs moyennes de mots selon les langues
- Développer des ensembles de données d'évaluation spécifiques aux langues pour les longs contextes
- Explorer des techniques d'adaptation par langue pour les métriques de routage

### 5. Prochaines étapes immédiates

Actions prioritaires à court terme:

1. Implémenter les hyperparamètres optimaux identifiés dans les expériences d'ablation
2. Réaliser des tests d'efficacité et de perplexité sur un ensemble plus large de benchmarks
3. Développer une version optimisée de l'implémentation CUDA

```

4. Tester la scalabilité avec des séquences jusqu'à 2M tokens
5. Intégrer les retours d'utilisateurs sur les cas d'utilisation réels

Calendrier suggéré

- **T1**: Optimisation des hyperparamètres et de l'implémentation
- **T2**: Extension à des contextes plus longs (>1M tokens) et tests approfondis
- **T3**: Adaptation multilingue et tests spécifiques aux langues
- **T4**: Intégration avec d'autres architectures de modèles et frameworks d'inférence

"""

```
# Écrire le rapport dans un fichier
with open(output_file, "w") as f:
    f.write(report)
```

```
logger.info(f"Rapport de feuille de route généré: {output_file}")
```

Ajouter à la fonction main

```
def main():
```

```
    # ... [code existant]
```

```
    # Générer la feuille de route pour le développement futur
    roadmap_file = os.path.join(args.output_dir, "development_roadmap.md")
    generate_development_roadmap(args.output_dir, roadmap_file)
```

```
    logger.info("Expériences d'ablation terminées. Résultats sauvegardés dans " + args.output_dir)
    logger.info("Feuille de route pour le développement futur générée: " + roadmap_file)
```

```
if __name__ == "__main__":
    main()
```

Finalement, assemblons tout ce code dans un seul script Python pour réaliser les expériences d'ablation et générer la feuille de route pour le développement futur. Ce script permet d'évaluer diverses configurations du modèle NeoBERT-MOBA et de fournir des recommandations basées sur les résultats.

Je vais maintenant finaliser le code en ajoutant les importations nécessaires et en vérifiant la cohérence globale.

Code pour les Expériences d'Ablation de NeoBERT-MOBA

Voici le code complet pour réaliser les expériences d'ablation demandées et prévoir la suite du développement pour NeoBERT-MOBA.

```
import os
import json
import time
import torch
import logging
import argparse
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm
from datetime import datetime
from torch.utils.data import DataLoader
import torch.nn as nn
import torch.nn.functional as F
import math

# Supposez que ces imports proviennent de vos modules précédents
from model_architecture import NeoBERTMoBA, create_neobert_moba_model, MoBAAttention, RotaryEmbedding
from data_preparation import DataPreparation

# Configuration du logging
logging.basicConfig(
    format="%(asctime)s - %(levelname)s - %(name)s - %(message)s",
    datefmt="%m/%d/%Y %H:%M:%S",
    level=logging.INFO,
)
logger = logging.getLogger(__name__)

class CustomMoBAAttention(nn.Module):
    """MoBA avec différentes métriques de routage"""
    def __init__(
        self,
        hidden_size: int,
        num_heads: int,
        block_size: int = 512,
        top_k: int = 3,
        routing_metric: str = "mean_pooling",
    ):
        super().__init__()
        self.hidden_size = hidden_size
        self.num_heads = num_heads
        self.head_dim = hidden_size // num_heads
        self.block_size = block_size
        self.top_k = top_k
        self.routing_metric = routing_metric

        self.q_proj = nn.Linear(hidden_size, hidden_size, bias=False)
        self.k_proj = nn.Linear(hidden_size, hidden_size, bias=False)
        self.v_proj = nn.Linear(hidden_size, hidden_size, bias=False)
        self.o_proj = nn.Linear(hidden_size, hidden_size, bias=False)
```

```

self.rotary = RotaryEmbedding(self.head_dim)

self.scale = 1.0 / math.sqrt(self.head_dim)

def _split_heads(self, x):
    new_shape = x.size()[:-1] + (self.num_heads, self.head_dim)
    x = x.view(*new_shape)
    return x.permute(0, 2, 1, 3)

def _merge_heads(self, x):
    x = x.permute(0, 2, 1, 3)
    new_shape = x.size()[:-2] + (self.hidden_size,)
    return x.reshape(*new_shape)

def _compute_block_representation(self, keys, block_indices):
    """
    Calcule la représentation d'un bloc selon la métrique choisie
    """
    if len(block_indices) == 0:
        return torch.zeros(keys.size(0), keys.size(1), 1, keys.size(3), device=keys.device)

    keys_in_block = keys[:, :, block_indices]

    if self.routing_metric == "mean_pooling":
        return keys_in_block.mean(dim=2, keepdim=True)

    elif self.routing_metric == "max_pooling":
        return keys_in_block.max(dim=2, keepdim=True)[0]

    elif self.routing_metric == "min_max_pooling":
        max_values = keys_in_block.max(dim=2, keepdim=True)[0]
        min_values = keys_in_block.min(dim=2, keepdim=True)[0]
        return (max_values + min_values) / 2

    elif self.routing_metric == "representative_token":
        # Utiliser le milieu du bloc comme token représentatif
        middle_idx = len(block_indices) // 2
        return keys[:, :, block_indices[middle_idx]:block_indices[middle_idx]+1]

    else:
        raise ValueError(f"Métrique de routage non supportée: {self.routing_metric}")

def forward(self, hidden_states, attention_mask=None, past_key_value=None):
    batch_size, seq_len = hidden_states.shape[:2]

    # Linear projections
    q = self.q_proj(hidden_states)
    k = self.k_proj(hidden_states)
    v = self.v_proj(hidden_states)

    # Split into heads
    q = self._split_heads(q)
    k = self._split_heads(k)
    v = self._split_heads(v)

    # Apply RoPE to q and k
    q = self.rotary(q, seq_len)
    k = self.rotary(k, seq_len)

    # Split sequence into blocks
    num_blocks = math.ceil(seq_len / self.block_size)

    # Compute block representations for gating
    block_reps = []
    for i in range(num_blocks):
        start_idx = i * self.block_size
        end_idx = min(start_idx + self.block_size, seq_len)
        if end_idx > start_idx:
            block_indices = list(range(start_idx, end_idx))
            block_rep = self._compute_block_representation(k, block_indices)
            block_reps.append(block_rep)

    # Stack block representations
    block_reps = torch.cat(block_reps, dim=2)

    # Initialize output tensor
    attn_output = torch.zeros_like(q)

    # Process each query token
    for i in range(seq_len):
        # Get the current query
        query = q[:, :, i:i+1]

        # Compute relevance scores between query and blocks
        block_scores = torch.matmul(query, block_reps.transpose(-1, -2)) * self.scale

        # Apply causal mask - query can only attend to blocks up to its position
        current_block = i // self.block_size
        causal_mask = torch.arange(num_blocks, device=query.device) > current_block
        block_scores = block_scores.masked_fill(
            causal_mask.view(1, 1, 1, -1), float("-inf"))
    )

    # Always include the current block

```

```

block_scores[:, :, current_block] = torch.finfo(block_scores.dtype).max

# Get top-k blocks
_, top_k_indices = torch.topk(block_scores, min(self.top_k, num_blocks), dim=-1)

# Get keys and values for selected blocks
selected_k = []
selected_v = []

for b in range(batch_size):
    for h in range(self.num_heads):
        k_for_query = []
        v_for_query = []

        for block_idx in top_k_indices[b, h, 0]:
            block_idx = block_idx.item()
            start_idx = block_idx * self.block_size
            end_idx = min(start_idx + self.block_size, seq_len)

            # For the current block, apply causal masking
            if block_idx == current_block:
                k_block = k[b, h:h+1, start_idx:i+1]
                v_block = v[b, h:h+1, start_idx:i+1]
            else:
                k_block = k[b, h:h+1, start_idx:end_idx]
                v_block = v[b, h:h+1, start_idx:end_idx]

            if k_block.size(2) > 0:
                k_for_query.append(k_block)
                v_for_query.append(v_block)

        if k_for_query:
            # Concatenate selected keys and values
            k_concat = torch.cat(k_for_query, dim=2)
            v_concat = torch.cat(v_for_query, dim=2)

            # Compute attention scores
            attn_scores = torch.matmul(query[b:b+1, h:h+1], k_concat.transpose(-1, -2)) * self.scale
            attn_probs = F.softmax(attn_scores, dim=-1)

            # Apply attention
            attn_output[b:b+1, h:h+1, i:i+1] = torch.matmul(attn_probs, v_concat)

# Merge heads and project back
attn_output = self._merge_heads(attn_output)
attn_output = self.o_proj(attn_output)

return attn_output

class NeoBERTMoBAWithCustomRouting(NeoBERTMoBA):
    """NeoBERT-MOBA avec métrique de routage personnalisée"""
    def __init__(self, *args, routing_metric="mean_pooling", **kwargs):
        super().__init__(*args, **kwargs)
        self.routing_metric = routing_metric

        # Recréer les couches d'attention avec la métrique personnalisée
        hybrid_count = self.config.get("hybrid_layer_count", 3)

        for i in range(len(self.layers)):
            if i < (len(self.layers) - hybrid_count):
                # Cette couche utilise MoBA
                self.layers[i].attention = CustomMoBAAttention(
                    hidden_size=self.config["hidden_size"],
                    num_heads=self.config["num_attention_heads"],
                    block_size=self.config["moba_block_size"],
                    top_k=self.config["moba_top_k"],
                    routing_metric=self.routing_metric
                )

def run_block_granularity_ablation(
    base_model_path,
    output_dir,
    context_length=32768,
    batch_size=1,
    max_samples=5,
    device="cuda"
):
    """
    Étude d'ablation sur la granularité des blocs MOBA
    """
    os.makedirs(output_dir, exist_ok=True)

    # Configurations à tester
    configurations = [
        {"blocks": 8, "top_k": 2, "name": "8_blocks_top2"},
        {"blocks": 16, "top_k": 4, "name": "16_blocks_top4"},
        {"blocks": 32, "top_k": 8, "name": "32_blocks_top8"},
        {"blocks": 64, "top_k": 16, "name": "64_blocks_top16"},
        {"blocks": 128, "top_k": 32, "name": "128_blocks_top32"},
    ]

    # Charger la configuration du modèle

```

```

with open(os.path.join(base_model_path, "config.json"), "r") as f:
    model_config = json.load(f)

# Préparer les données d'évaluation
data_prep = DataPreparation(max_length=context_length)
eval_dataset = data_prep.create_evaluation_dataset(
    dataset_name="pg19",
    split="test",
    max_samples=max_samples
)
eval_dataloader = DataLoader(
    eval_dataset,
    batch_size=batch_size,
    collate_fn=eval_dataset.collate_fn
)

# Résultats
results = {}

# Tester chaque configuration
for config in configurations:
    logger.info(f"Évaluation de la configuration: {config['name']}")

    # Calculer la taille du bloc pour cette configuration
    block_size = context_length // config["blocks"]

    # Mettre à jour la configuration
    test_config = model_config.copy()
    test_config["moba_block_size"] = block_size
    test_config["moba_top_k"] = config["top_k"]

    # Charger le modèle
    model = create_neobert_moba_model(test_config)
    model.load_state_dict(torch.load(os.path.join(base_model_path, "pytorch_model.bin")))
    model = model.to(device)
    model.eval()

    # Mesurer performance
    time_measurements = []
    memory_usage = []
    losses = []

    for batch in tqdm(eval_dataloader, desc=f"Évaluation {config['name']}"):
        input_ids = batch["input_ids"].to(device)
        attention_mask = batch["attention_mask"].to(device)

        # Créer labels (décalés d'une position)
        labels = input_ids.clone()
        labels[:, :-1] = input_ids[:, 1:]
        labels[:, -1] = -100 # Ignorer le dernier token

        # Mesurer mémoire et temps
        if torch.cuda.is_available():
            torch.cuda.empty_cache()
            torch.cuda.reset_peak_memory_stats()
            start_mem = torch.cuda.memory_allocated()

        torch.cuda.synchronize()
        start_time = time.time()

        with torch.no_grad():
            outputs = model(input_ids, attention_mask)
            logits = outputs["prediction_logits"]

            # Calculer perte
            loss = F.cross_entropy(
                logits.reshape(-1, logits.size(-1)),
                labels.reshape(-1),
                ignore_index=-100,
                reduction="mean"
            )

        torch.cuda.synchronize()
        end_time = time.time()

        if torch.cuda.is_available():
            peak_mem = torch.cuda.max_memory_allocated() - start_mem
            memory_usage.append(peak_mem / (1024 * 1024)) # MB

        time_measurements.append(end_time - start_time)
        losses.append(loss.item())

    # Calculer moyennes
    avg_time = np.mean(time_measurements)
    avg_memory = np.mean(memory_usage) if memory_usage else None
    avg_loss = np.mean(losses)
    perplexity = np.exp(avg_loss)

    # Stocker résultats
    results[config['name']] = {
        "blocks": config["blocks"],
        "top_k": config["top_k"],
        "block_size": block_size,
        "sparsity": 1 - (config["top_k"] / config["blocks"]),

```



```

        "avg_time": avg_time,
        "avg_memory": avg_memory,
        "avg_loss": avg_loss,
        "perplexity": perplexity
    }

    logger.info(f"Résultats pour {config['name']}:")
    logger.info(f" Temps moyen: {avg_time:.4f} s")
    logger.info(f" Mémoire moyenne: {avg_memory:.2f} MB" if avg_memory else " Mémoire: N/A")
    logger.info(f" Perte moyenne: {avg_loss:.4f}")
    logger.info(f" Perplexité: {perplexity:.4f}")

# Sauvegarder résultats
with open(os.path.join(output_dir, "block_granularity_results.json"), "w") as f:
    json.dump(results, f, indent=2)

# Créer visualisations
create_block_granularity_visualizations(results, output_dir)

return results

def create_block_granularity_visualizations(results, output_dir):
    """Crée des visualisations pour les résultats d'ablation de granularité"""
    plt.figure(figsize=(15, 10))

    # Extraire données
    configs = list(results.keys())
    blocks = [results[c]["blocks"] for c in configs]
    times = [results[c]["avg_time"] for c in configs]
    memories = [results[c]["avg_memory"] for c in configs if results[c]["avg_memory"] is not None]
    perplexities = [results[c]["perplexity"] for c in configs]

    # Sous-graphique pour le temps
    plt.subplot(2, 2, 1)
    plt.bar(configs, times, color='skyblue')
    plt.ylabel("Temps (s)")
    plt.title("Temps d'inférence par configuration")
    plt.xticks(rotation=45)
    plt.grid(True, axis='y', linestyle='--', alpha=0.7)

    # Sous-graphique pour la mémoire
    if memories:
        plt.subplot(2, 2, 2)
        plt.bar(configs[:len(memories)], memories, color='lightgreen')
        plt.ylabel("Mémoire (MB)")
        plt.title("Utilisation mémoire par configuration")
        plt.xticks(rotation=45)
        plt.grid(True, axis='y', linestyle='--', alpha=0.7)

    # Sous-graphique pour la perplexité
    plt.subplot(2, 2, 3)
    plt.bar(configs, perplexities, color='salmon')
    plt.ylabel("Perplexité")
    plt.title("Perplexité par configuration")
    plt.xticks(rotation=45)
    plt.grid(True, axis='y', linestyle='--', alpha=0.7)

    # Sous-graphique pour la relation blocks vs perplexité
    plt.subplot(2, 2, 4)
    plt.plot(blocks, perplexities, marker='o', linestyle='-', color='purple')
    plt.xlabel("Nombre de blocs")
    plt.ylabel("Perplexité")
    plt.title("Relation entre nombre de blocs et perplexité")
    plt.grid(True, linestyle='--', alpha=0.7)

    plt.tight_layout()
    plt.savefig(os.path.join(output_dir, "block_granularity_comparison.png"), dpi=300)

def run_hybrid_strategy_ablation(
    base_model_path,
    output_dir,
    context_length=32768,
    batch_size=1,
    max_samples=5,
    device="cuda"
):
    """
    Étude d'ablation sur la stratégie hybride
    """
    os.makedirs(output_dir, exist_ok=True)

    # Charger configuration modèle
    with open(os.path.join(base_model_path, "config.json"), "r") as f:
        model_config = json.load(f)

    # Nombre total de couches
    total_layers = model_config["num_hidden_layers"]

    # Configurations à tester
    hybrid_configs = [
        {"hybrid_count": 0, "name": "full_moba"},
        {"hybrid_count": 1, "name": "hybrid_1"},
    ]

```

```

{"hybrid_count": 3, "name": "hybrid_3"},
{"hybrid_count": 5, "name": "hybrid_5"},
{"hybrid_count": 10, "name": "hybrid_10"},
{"hybrid_count": total_layers, "name": "full_attention"}
]

# Préparer données évaluation
data_prep = DataPreparation(max_length=context_length)
eval_dataset = data_prep.create_evaluation_dataset(
    dataset_name="pg19",
    split="test",
    max_samples=max_samples
)
eval_dataloader = DataLoader(
    eval_dataset,
    batch_size=batch_size,
    collate_fn=eval_dataset.collate_fn
)

# Résultats
results = {}

# Tester chaque configuration
for config in hybrid_configs:
    logger.info(f"Évaluation de la configuration: {config['name']}")

    # Charger modèle
    model = create_neobert_moba_model(model_config)
    model.load_state_dict(torch.load(os.path.join(base_model_path, "pytorch_model.bin")))
    model = model.to(device)
    model.eval()

    # Appliquer stratégie hybride
    model.switch_to_hybrid_mode(config["hybrid_count"])

    # Mesurer performance
    time_measurements = []
    memory_usage = []
    losses = []

    for batch in tqdm(eval_dataloader, desc=f"Évaluation {config['name']}"):
        input_ids = batch["input_ids"].to(device)
        attention_mask = batch["attention_mask"].to(device)

        # Créer labels
        labels = input_ids.clone()
        labels[:, :-1] = input_ids[:, 1:]
        labels[:, -1] = -100

        # Mesurer mémoire et temps
        if torch.cuda.is_available():
            torch.cuda.empty_cache()
            torch.cuda.reset_peak_memory_stats()
            start_mem = torch.cuda.memory_allocated()

        torch.cuda.synchronize()
        start_time = time.time()

        with torch.no_grad():
            outputs = model(input_ids, attention_mask)
            logits = outputs["prediction_logits"]

            # Calculer perte
            loss = F.cross_entropy(
                logits.reshape(-1, logits.size(-1)),
                labels.reshape(-1),
                ignore_index=-100,
                reduction="mean"
            )

        torch.cuda.synchronize()
        end_time = time.time()

        if torch.cuda.is_available():
            peak_mem = torch.cuda.max_memory_allocated() - start_mem
            memory_usage.append(peak_mem / (1024 * 1024))

        time_measurements.append(end_time - start_time)
        losses.append(loss.item())

    # Calculer moyennes
    avg_time = np.mean(time_measurements)
    avg_memory = np.mean(memory_usage) if memory_usage else None
    avg_loss = np.mean(losses)
    perplexity = np.exp(avg_loss)

    # Stocker résultats
    results[config['name']] = {
        "hybrid_count": config["hybrid_count"],
        "moba_count": total_layers - config["hybrid_count"],
        "moba_percentage": (total_layers - config["hybrid_count"]) / total_layers * 100,
        "avg_time": avg_time,
        "avg_memory": avg_memory,
        "avg_loss": avg_loss,

```

```

    "perplexity": perplexity
}

logger.info(f"Résultats pour {config['name']}:")
logger.info(f" Temps moyen: {avg_time:.4f} s")
logger.info(f" Mémoire moyenne: {avg_memory:.2f} MB" if avg_memory else " Mémoire: N/A")
logger.info(f" Perte moyenne: {avg_loss:.4f}")
logger.info(f" Perplexité: {perplexity:.4f}")

# Sauvegarder résultats
with open(os.path.join(output_dir, "hybrid_strategy_results.json"), "w") as f:
    json.dump(results, f, indent=2)

# Créer visualisations
create_hybrid_strategy_visualizations(results, output_dir, total_layers)

return results

def create_hybrid_strategy_visualizations(results, output_dir, total_layers):
    """Crée des visualisations pour les résultats d'ablation de stratégie hybride"""
    plt.figure(figsize=(15, 10))

    # Extraire données
    configs = list(results.keys())
    hybrid_counts = [results[c]["hybrid_count"] for c in configs]
    moba_percentages = [results[c]["moba_percentage"] for c in configs]
    times = [results[c]["avg_time"] for c in configs]
    memories = [results[c]["avg_memory"] for c in configs if results[c]["avg_memory"] is not None]
    perplexities = [results[c]["perplexity"] for c in configs]

    # Sous-graphique pour le temps
    plt.subplot(2, 2, 1)
    plt.plot(moba_percentages, times, marker='o', linestyle='-', color='blue')
    plt.xlabel("% de couches MOBA")
    plt.ylabel("Temps (s)")
    plt.title("Temps d'inférence vs % de couches MOBA")
    plt.grid(True, linestyle='--', alpha=0.7)

    # Sous-graphique pour la mémoire
    if memories:
        plt.subplot(2, 2, 2)
        plt.plot(moba_percentages[:len(memories)], memories, marker='o', linestyle='-', color='green')
        plt.xlabel("% de couches MOBA")
        plt.ylabel("Mémoire (MB)")
        plt.title("Utilisation mémoire vs % de couches MOBA")
        plt.grid(True, linestyle='--', alpha=0.7)

    # Sous-graphique pour la perplexité
    plt.subplot(2, 2, 3)
    plt.plot(moba_percentages, perplexities, marker='o', linestyle='-', color='red')
    plt.xlabel("% de couches MOBA")
    plt.ylabel("Perplexité")
    plt.title("Perplexité vs % de couches MOBA")
    plt.grid(True, linestyle='--', alpha=0.7)

    # Sous-graphique pour le compromis perplexité/temps
    plt.subplot(2, 2, 4)
    plt.scatter(times, perplexities, c=moba_percentages, cmap='viridis', s=100)

    # Ajouter étiquettes pour chaque point
    for i, config in enumerate(configs):
        plt.annotate(config, (times[i], perplexities[i]),
                     xytext=(5, 5), textcoords='offset points')

    plt.colorbar(label="% de couches MOBA")
    plt.xlabel("Temps (s)")
    plt.ylabel("Perplexité")
    plt.title("Compromis perplexité/temps")
    plt.grid(True, linestyle='--', alpha=0.7)

    plt.tight_layout()
    plt.savefig(os.path.join(output_dir, "hybrid_strategy_comparison.png"), dpi=300)

def run_routing_metrics_ablation(
    base_model_path,
    output_dir,
    context_length=32768,
    batch_size=1,
    max_samples=5,
    device="cuda"
):
    """
    Étude d'ablation sur les métriques de routage
    """
    os.makedirs(output_dir, exist_ok=True)

    # Charger configuration modèle
    with open(os.path.join(base_model_path, "config.json"), "r") as f:
        model_config = json.load(f)

    # Métriques de routage à tester
    routing_metrics = [

```

```

"mean_pooling",
"max_pooling",
"min_max_pooling",
"representative_token"
]

# Préparer données évaluation
data_prep = DataPreparation(max_length=context_length)
eval_dataset = data_prep.create_evaluation_dataset(
    dataset_name="pg19",
    split="test",
    max_samples=max_samples
)
eval_dataloader = DataLoader(
    eval_dataset,
    batch_size=batch_size,
    collate_fn=eval_dataset.collate_fn
)

# Résultats
results = {}

# Tester chaque métrique
for metric in routing_metrics:
    logger.info(f"Évaluation de la métrique de routage: {metric}")

    # Créer modèle avec métrique personnalisée
    model = NeoBERTMoBAWithCustomRouting(
        vocab_size=model_config["vocab_size"],
        hidden_size=model_config["hidden_size"],
        num_hidden_layers=model_config["num_hidden_layers"],
        num_attention_heads=model_config["num_attention_heads"],
        max_position_embeddings=model_config["max_position_embeddings"],
        moba_block_size=model_config["moba_block_size"],
        moba_top_k=model_config["moba_top_k"],
        hybrid_layer_count=model_config["hybrid_layer_count"],
        routing_metric=metric
    )

    # Charger poids
    model.load_state_dict(torch.load(os.path.join(base_model_path, "pytorch_model.bin")))
    model = model.to(device)
    model.eval()

    # Mesurer performance
    time_measurements = []
    losses = []

    for batch in tqdm(eval_dataloader, desc=f"Évaluation {metric}"):
        input_ids = batch["input_ids"].to(device)
        attention_mask = batch["attention_mask"].to(device)

        # Créer labels
        labels = input_ids.clone()
        labels[:, :-1] = input_ids[:, 1:]
        labels[:, -1] = -100

        torch.cuda.synchronize()
        start_time = time.time()

        with torch.no_grad():
            outputs = model(input_ids, attention_mask)
            logits = outputs["prediction_logits"]

            # Calculer perte
            loss = F.cross_entropy(
                logits.reshape(-1, logits.size(-1)),
                labels.reshape(-1),
                ignore_index=-100,
                reduction="mean"
            )

            torch.cuda.synchronize()
            end_time = time.time()

            time_measurements.append(end_time - start_time)
            losses.append(loss.item())

    # Calculer moyennes
    avg_time = np.mean(time_measurements)
    avg_loss = np.mean(losses)
    perplexity = np.exp(avg_loss)

    # Stocker résultats
    results[metric] = {
        "avg_time": avg_time,
        "avg_loss": avg_loss,
        "perplexity": perplexity
    }

    logger.info(f"Résultats pour {metric}:")
    logger.info(f" Temps moyen: {avg_time:.4f} s")
    logger.info(f" Perte moyenne: {avg_loss:.4f}")
    logger.info(f" Perplexité: {perplexity:.4f}")

```

```

# Sauvegarder résultats
with open(os.path.join(output_dir, "routing_metrics_results.json"), "w") as f:
    json.dump(results, f, indent=2)

# Créer visualisations
create_routing_metrics_visualizations(results, output_dir)

return results

def create_routing_metrics_visualizations(results, output_dir):
    """Crée des visualisations pour les résultats d'ablation de métriques de routage"""
    plt.figure(figsize=(15, 5))

    # Extraire données
    metrics = list(results.keys())
    times = [results[m]["avg_time"] for m in metrics]
    perplexities = [results[m]["perplexity"] for m in metrics]

    # Sous-graphique pour le temps
    plt.subplot(1, 2, 1)
    bars = plt.bar(metrics, times, color='skyblue')
    plt.ylabel("Temps (s)")
    plt.title("Temps d'inférence par métrique de routage")
    plt.xticks(rotation=45)
    plt.grid(True, axis='y', linestyle='--', alpha=0.7)

    # Ajouter valeurs sur les barres
    for bar in bars:
        height = bar.get_height()
        plt.text(bar.get_x() + bar.get_width()/2., height,
                 f'{height:.3f}',
                 ha='center', va='bottom')

    # Sous-graphique pour la perplexité
    plt.subplot(1, 2, 2)
    bars = plt.bar(metrics, perplexities, color='salmon')
    plt.ylabel("Perplexité")
    plt.title("Perplexité par métrique de routage")
    plt.xticks(rotation=45)
    plt.grid(True, axis='y', linestyle='--', alpha=0.7)

    # Ajouter valeurs sur les barres
    for bar in bars:
        height = bar.get_height()
        plt.text(bar.get_x() + bar.get_width()/2., height,
                 f'{height:.3f}',
                 ha='center', va='bottom')

    plt.tight_layout()
    plt.savefig(os.path.join(output_dir, "routing_metrics_comparison.png"), dpi=300)

def generate_development_roadmap(ablation_results_dir, output_file):
    """
    Génère un rapport sur la suite du développement basé sur les résultats d'ablation
    """
    # Charger résultats des ablations
    results = {}

    # Charger résultats de granularité
    granularity_file = os.path.join(ablation_results_dir, "block_granularity", "block_granularity_results.json")
    if os.path.exists(granularity_file):
        with open(granularity_file, "r") as f:
            results["granularity"] = json.load(f)

    # Charger résultats de stratégie hybride
    hybrid_file = os.path.join(ablation_results_dir, "hybrid_strategy", "hybrid_strategy_results.json")
    if os.path.exists(hybrid_file):
        with open(hybrid_file, "r") as f:
            results["hybrid"] = json.load(f)

    # Charger résultats des métriques de routage
    routing_file = os.path.join(ablation_results_dir, "routing_metrics", "routing_metrics_results.json")
    if os.path.exists(routing_file):
        with open(routing_file, "r") as f:
            results["routing"] = json.load(f)

    # Analyser résultats pour recommandations
    recommendations = {}

    # Analyser granularité des blocs
    if "granularity" in results:
        granularity_data = results["granularity"]

        # Trouver meilleure configuration (perplexité)
        best_perplexity = float('inf')
        best_config = None

        for config, data in granularity_data.items():
            if data["perplexity"] < best_perplexity:
                best_perplexity = data["perplexity"]
                best_config = config

```

```

if best_config:
    recommendations["block_size"] = granularity_data[best_config]["block_size"]
    recommendations["blocks"] = granularity_data[best_config]["blocks"]
    recommendations["top_k"] = granularity_data[best_config]["top_k"]

# Analyser stratégie hybride
if "hybrid" in results:
    hybrid_data = results["hybrid"]

    # Trouver meilleur compromis perplexité/vitesse
    configs = []
    for config, data in hybrid_data.items():
        configs.append({
            "name": config,
            "hybrid_count": data["hybrid_count"],
            "perplexity": data["perplexity"],
            "time": data["avg_time"]
        })

    # Trier par perplexité
    configs.sort(key=lambda x: x["perplexity"])

    # Prendre configuration avec meilleure perplexité parmi les 3 premières
    # qui a aussi un bon temps d'exécution
    best_hybrid = configs[0]
    for i in range(1, min(3, len(configs))):
        if configs[i]["time"] < best_hybrid["time"] * 0.8: # 20% plus rapide
            best_hybrid = configs[i]

    recommendations["hybrid_count"] = best_hybrid["hybrid_count"]

# Analyser métriques de routage
if "routing" in results:
    routing_data = results["routing"]

    # Trouver métrique avec meilleure perplexité
    best_perplexity = float('inf')
    best_metric = None

    for metric, data in routing_data.items():
        if data["perplexity"] < best_perplexity:
            best_perplexity = data["perplexity"]
            best_metric = metric

    if best_metric:
        recommendations["routing_metric"] = best_metric

# Générer rapport
report = f""""# Feuille de route pour le développement futur de NeoBERT-MOBA

## Recommandations basées sur les expériences d'ablation

### 1. Ajustement des hyperparamètres MOBA

Sur la base des expériences d'ablation, les paramètres optimaux recommandés sont:

""""

    if "blocks" in recommendations and "top_k" in recommendations:
        report += f""- **Nombre de blocs**: {recommendations["blocks"]}\n"
        report += f""- **Taille de bloc**: {recommendations["block_size"]} tokens\n"
        report += f""- **Paramètre top-k**: {recommendations["top_k"]}\n"
    else:
        report += "- Des tests supplémentaires sont nécessaires pour déterminer la granularité optimale des blocs.\n"

    if "hybrid_count" in recommendations:
        report += f""- **Nombre de couches d'attention complète**: {recommendations["hybrid_count"]}\n"
    else:
        report += "- Des tests supplémentaires sont nécessaires pour déterminer la stratégie hybride optimale.\n"

    if "routing_metric" in recommendations:
        report += f""- **Métrique de routage recommandée**: {recommendations["routing_metric"]}\n"
    else:
        report += "- Des tests supplémentaires sont nécessaires pour déterminer la métrique de routage optimale.\n"

    report += """"

### 2. Optimisation de l'implémentation

Pour améliorer les performances sur différents matériels:

- Optimiser l'implémentation CUDA pour améliorer l'efficacité sur GPU
- Explorer les optimisations pour CPU (AVX, OpenMP)
- Adapter l'implémentation pour hardware spécialisé (TPU, NPU)
- Optimiser la gestion mémoire pour réduire l'empreinte mémoire lors de l'inférence
- Intégrer avec les frameworks d'inférence comme ONNX Runtime, TensorRT

### 3. Extension à des contextes plus longs

Pour atteindre et dépasser 1M tokens:

- Tester des stratégies de mise à l'échelle pour adapter la taille des blocs et le paramètre top-k
- Explorer des techniques de compression mémoire pour stocker l'historique
- Développer une version hiérarchique de MOBA avec plusieurs niveaux de granularité

```

- Expérimenter avec des techniques d'attention récurrente pour les très longs contextes
- Intégrer avec des méthodes d'indexation ou de recherche pour les contextes extrêmement longs (>10M tokens)

4. Adaptation à d'autres langues

Pour améliorer les performances multilingues:

- Tester MOBA avec des tokenizers spécifiques aux langues
- Ajuster la granularité des blocs pour les langues aux caractéristiques différentes
- Évaluer l'impact des différentes longueurs moyennes de mots selon les langues
- Développer des ensembles de données d'évaluation spécifiques aux langues pour les longs contextes
- Explorer des techniques d'adaptation par langue pour les métriques de routage

5. Prochaines étapes immédiates

Actions prioritaires à court terme:

1. Implémenter les hyperparamètres optimaux identifiés dans les expériences d'ablation
2. Réaliser des tests d'efficacité et de perplexité sur un ensemble plus large de benchmarks
3. Développer une version optimisée de l'implémentation CUDA
4. Tester la scalabilité avec des séquences jusqu'à 2M tokens
5. Intégrer les retours d'utilisateurs sur les cas d'utilisation réels

Calendrier suggéré

- **T1** : Optimisation des hyperparamètres et de l'implémentation
- **T2** : Extension à des contextes plus longs (>1M tokens) et tests approfondis
- **T3** : Adaptation multilingue et tests spécifiques aux langues
- **T4** : Intégration avec d'autres architectures de modèles et frameworks d'inférence

"""

```
# Écrire rapport dans fichier
with open(output_file, "w") as f:
    f.write(report)
```

```
logger.info(f"Rapport de feuille de route généré: {output_file}")
```

def main():

"""Fonction principale pour exécuter toutes les expériences d'ablation"""

```
parser = argparse.ArgumentParser(description="Expériences d'ablation pour NeoBERT-MOBA")
parser.add_argument("--model_path", type=str, required=True, help="Chemin vers le checkpoint du modèle")
parser.add_argument("--output_dir", type=str, default="/ablation_results", help="Répertoire de sortie")
parser.add_argument("--context_length", type=int, default=32768, help="Longueur du contexte pour l'évaluation")
parser.add_argument("--max_samples", type=int, default=5, help="Nombre maximum d'échantillons")
parser.add_argument("--batch_size", type=int, default=1, help="Taille du batch")
parser.add_argument("--device", type=str, default="cuda" if torch.cuda.is_available() else "cpu", help="Device à utiliser")
```

```
parser.add_argument("--run_granularity", action="store_true", help="Exécuter l'ablation sur la granularité des blocs")
parser.add_argument("--run_hybrid", action="store_true", help="Exécuter l'ablation sur la stratégie hybride")
parser.add_argument("--run_routing", action="store_true", help="Exécuter l'ablation sur les métriques de routage")
parser.add_argument("--run_all", action="store_true", help="Exécuter toutes les ablations")
```

```
args = parser.parse_args()
```

```
# Créer répertoire de sortie principal
os.makedirs(args.output_dir, exist_ok=True)
```

```
# Exécuter expériences sélectionnées
```

```
if args.run_all or args.run_granularity:
    granularity_dir = os.path.join(args.output_dir, "block_granularity")
    run_block_granularity_ablation(
        base_model_path=args.model_path,
        output_dir=granularity_dir,
        context_length=args.context_length,
        batch_size=args.batch_size,
        max_samples=args.max_samples,
        device=args.device
    )
```

```
if args.run_all or args.run_hybrid:
    hybrid_dir = os.path.join(args.output_dir, "hybrid_strategy")
    run_hybrid_strategy_ablation(
        base_model_path=args.model_path,
        output_dir=hybrid_dir,
        context_length=args.context_length,
        batch_size=args.batch_size,
        max_samples=args.max_samples,
        device=args.device
    )
```

```
if args.run_all or args.run_routing:
    routing_dir = os.path.join(args.output_dir, "routing_metrics")
    run_routing_metrics_ablation(
        base_model_path=args.model_path,
        output_dir=routing_dir,
        context_length=args.context_length,
        batch_size=args.batch_size,
        max_samples=args.max_samples,
        device=args.device
    )
```

```
# Générer feuille de route pour développement futur
```

```
roadmap_file = os.path.join(args.output_dir, "development_roadmap.md")
generate_development_roadmap(args.output_dir, roadmap_file)

logger.info("Expériences d'ablation terminées. Résultats sauvegardés dans " + args.output_dir)
logger.info("Feuille de route pour le développement futur générée: " + roadmap_file)

if __name__ == "__main__":
    main()
```

Utilisation du script

Pour exécuter toutes les expériences d'ablation:

```
```bash
python ablation_experiments.py --model_path ./checkpoints/neobert_moba --output_dir ./ablation_results --run_all
```
```

Pour exécuter une expérience spécifique:

```
```bash
python ablation_experiments.py --model_path ./checkpoints/neobert_moba --output_dir ./ablation_results --run_granularity
```
```

Fonctionnalités du code

1. **Étude de la granularité des blocs**: Teste 5 configurations de granularité différentes (8 à 128 blocs) avec un taux de sparsité constant.
2. **Étude de la stratégie hybride**: Compare différentes combinaisons de couches MOBA et d'attention complète.
3. **Étude des métriques de routage**: Évalue 4 méthodes de calcul des scores d'affinité:
 - Mean pooling (standard)
 - Max pooling
 - Combinaison min/max pooling
 - Token représentatif
4. **Visualisations automatiques**: Génère des graphiques pour chaque expérience permettant d'analyser les compromis entre efficacité et performance.
5. **Feuille de route générée automatiquement**: Produit un rapport détaillé avec des recommandations pour la suite du développement basées sur les résultats d'ablation.

Ce code permet d'optimiser rigoureusement les paramètres de NeoBERT-MOBA et d'explorer systématiquement les compromis entre performance et efficacité computationnelle.

Sources: [2502.13189v1.pdf](#)

USER

combien de puissance de calcul et heure de gpu estime tu pour faire les 3 phase d'entrainement ? Et est ce qu'avec des 4 RTX 4090 avec 24gbb de vram cela peut fonctionnner pour tester le code dans un premier temps ?

BOT

Thoughts

Pour estimer la puissance de calcul et les heures GPU nécessaires pour les 3 phases d'entraînement du modèle NeoBERT-MOBA, je dois analyser plusieurs facteurs:

1. Taille du modèle et paramètres
2. Quantité de données et nombre d'étapes
3. Consommation de mémoire et performance par GPU

D'après le code, voici les configurations principales du modèle:

- Taille du vocabulaire: 30,000
- Dimension cachée: 768
- Nombre de couches: 28
- Nombre de têtes d'attention: 12
- Longueur de position maximale: 32,768

C'est un modèle de taille moyenne, similaire à un BERT-base mais avec plus de couches.

Maintenant, analysons chaque phase d'entraînement:

Phase 1: Pré-entraînement initial (1024 tokens)

- Nombre d'étapes: 1,000,000
- Batch size par GPU: 32
- Gradient accumulation: 8
- Batch size effectif: $32 \times 8 \times \text{nombre_GPUs}$

Phase 2: Extension du contexte (4096 tokens)

- Nombre d'étapes: 50,000
- Batch size par GPU: 4
- Gradient accumulation: 4
- Batch size effectif: $4 \times 4 \times \text{nombre_GPUs}$

Phase 3: Activation MOBA (jusqu'à 32K tokens)

- Nombre d'étapes: 50,000
- Batch size par GPU: 1
- Gradient accumulation: 8
- Batch size effectif: $1 \times 8 \times \text{nombre_GPUs}$

Pour une RTX 4090 (24 GB VRAM):

1. La consommation de mémoire dépendra principalement de la longueur des séquences et du batch size.
2. Une RTX 4090 peut traiter des séquences d'environ 32K tokens, mais avec un petit batch size.
3. Le débit moyen d'une RTX 4090 pour l'entraînement de modèles de langage est approximativement de 80-120 TFLOPS.

Estimation du temps avec 4 RTX 4090:

Phase 1:

- La séquence est courte (1024 tokens), ce qui est moins exigeant en mémoire
- Batch size 32 par GPU devrait être possible
- Environ 400-600 exemples par seconde avec 4 GPUs
- Pour 1M steps: ~1,700-2,500 heures GPU (425-625 heures avec 4 GPUs)

Phase 2:

- Séquences plus longues (4096 tokens)
- Batch size 4 par GPU semble raisonnable
- Environ 40-60 exemples par seconde avec 4 GPUs
- Pour 50K steps: ~210-320 heures GPU (52-80 heures avec 4 GPUs)

Phase 3:

- Très longues séquences (jusqu'à 32K tokens)
- Batch size 1 par GPU est nécessaire
- Environ 2-4 exemples par seconde avec 4 GPUs
- Pour 50K steps: ~350-700 heures GPU (87-175 heures avec 4 GPUs)

Total estimé: ~2,260-3,520 heures GPU, soit environ 565-880 heures (24-37 jours) avec 4 RTX 4090 en utilisation continue.

Faisabilité avec 4 RTX 4090:

Pour le modèle complet et l'entraînement comme décrit:

- Phase 1: Faisable avec 4 RTX 4090, avec le batch size spécifié
- Phase 2: Faisable avec 4 RTX 4090, mais pourrait nécessiter une réduction du batch size
- Phase 3: C'est la phase la plus exigeante en mémoire. Avec des séquences de 32K tokens, chaque GPU devra probablement utiliser un batch size de 1, et même ainsi, cela pourrait être serré avec une RTX 4090 à 24GB VRAM.

Pour tester le code dans un premier temps:

1. Oui, vous pouvez utiliser 4 RTX 4090 pour commencer à tester le code, surtout pour les phases 1 et 2.
2. Pour la phase 3, vous pourriez avoir besoin de:
 - Réduire temporairement la longueur maximale des séquences (par exemple à 16K au lieu de 32K)
 - Utiliser des techniques d'économie de mémoire comme la remise à zéro des activations
 - Utiliser le gradient checkpointing
 - Envisager une précision mixte (FP16) qui est déjà configurée dans le code

Une approche progressive serait:

1. Tester d'abord avec un petit sous-ensemble de données pour vérifier que tout fonctionne
2. Exécuter des tests de mémoire pour déterminer les limites exactes pour votre matériel
3. Ajuster les hyperparamètres en conséquence
4. Commencer par la phase 1, qui est la moins exigeante en mémoire mais la plus longue en temps

Une estimation plus fine nécessiterait des benchmarks spécifiques sur votre matériel, car les performances peuvent varier considérablement en fonction de nombreux facteurs.

Estimation des ressources pour l'entraînement de NeoBERT-MOBA

Puissance de calcul et heures GPU estimées

En analysant les trois phases d'entraînement pour NeoBERT-MOBA, voici une estimation des ressources nécessaires:

Phase 1: Pré-entraînement initial (1024 tokens)

- **1 million d'étapes** avec 2 billions de tokens
- **Batch size effectif**: $32 \times 8 \times 4 \text{ GPUs} = 1024$
- **Estimation**: ~500-600 heures avec 4 GPUs (environ 2000-2400 heures GPU)
- **Débit approximatif**: 400-500 exemples/seconde avec 4 RTX 4090

Phase 2: Extension du contexte (4096 tokens)

- **50,000 étapes** avec 100 milliards de tokens
- **Batch size effectif**: $4 \times 4 \times 4 \text{ GPUs} = 64$
- **Estimation**: ~60-80 heures avec 4 GPUs (environ 240-320 heures GPU)
- **Débit approximatif**: 50-70 exemples/seconde avec 4 RTX 4090

Phase 3: Activation MOBA (32K tokens)

- **50,000 étapes** avec séquences très longues
- **Batch size effectif**: $1 \times 8 \times 4 \text{ GPUs} = 32$
- **Estimation**: ~100-150 heures avec 4 GPUs (environ 400-600 heures GPU)
- **Débit approximatif**: 3-5 exemples/seconde avec 4 RTX 4090

Total estimé: ~660-830 heures avec 4 GPUs (27-35 jours d'entraînement continu)

Faisabilité avec 4 RTX 4090 (24GB VRAM)

Oui, vous pouvez utiliser 4 RTX 4090 pour tester le code, mais avec quelques considérations importantes:

Phase 1

- **Parfaitement viable** - Les séquences courtes (1024 tokens) et le batch size de 32 sont bien adaptés à la VRAM disponible
- Cette phase fonctionnera sans problème sur les RTX 4090

Phase 2

- **Généralement viable** - Les séquences de 4096 tokens avec batch size 4 devraient tenir dans 24GB
- Vous pourriez avoir besoin d'activer le gradient checkpointing pour économiser de la mémoire

Phase 3

- **Potentiellement limitée** - Les séquences de 32K tokens sont très exigeantes en mémoire
- Recommandations pour cette phase:
 - Utiliser un batch size de 1 par GPU (comme déjà configuré)
 - Activer la précision mixte (FP16) qui est déjà configurée dans le code
 - Utiliser gradient checkpointing

- Vous pourriez avoir besoin de réduire temporairement la longueur maximale à 16K-24K tokens pour les premiers tests

Stratégie de test recommandée

1. ****Commencer petit**** - Exécutez d'abord des tests courts (100-1000 étapes) pour vérifier la stabilité et la consommation mémoire
2. ****Monitoring des ressources**** - Surveillez la consommation VRAM et ajustez les paramètres si nécessaire
3. ****Approche progressive**** - Commencez par la phase 1, puis passez aux phases suivantes
4. ****Optimisation de mémoire**** - Implémentez des techniques d'économie de mémoire comme:
 - Gradient checkpointing
 - Offloading des optimiseurs
 - Flash Attention (si disponible pour votre configuration)

Pour une vérification rapide de votre setup, vous pourriez d'abord exécuter l'entraînement sur un petit sous-ensemble de données pendant quelques jours avant de vous engager dans l'entraînement complet.

USER

utilisent un dataset plus petit que refined web pour tester le code fait un code modulaire.

```
import os
import json
import random
import logging
import numpy as np
import torch
from torch.utils.data import Dataset, DataLoader, DistributedSampler
from datasets import load_dataset, concatenate_datasets
from tokenizers import Tokenizer, models, normalizers, pre_tokenizers, decoders, trainers, processors
from transformers import PreTrainedTokenizerFast
from typing import Dict, List, Optional, Tuple, Union
import multiprocessing
from tqdm import tqdm
```

Configuration du logging

```
logging.basicConfig(
    format="%(asctime)s - %(levelname)s - %(message)s",
    datefmt="%m/%d/%Y %H:%M:%S",
    level=logging.INFO,
)
logger = logging.getLogger(__name__)
```

class DataPreparation:

```
    def __init__(
        self,
        base_dir: str = "./data",
        vocab_size: int = 30000,
        max_length: int = 32768,
        mlm_probability: float = 0.15,
        seed: int = 42
    ):
        self.base_dir = base_dir
        self.vocab_size = vocab_size
        self.max_length = max_length
        self.mlm_probability = mlm_probability
        self.seed = seed
        self.tokenizer = None
```

```
        # Créer le répertoire de base s'il n'existe pas
        os.makedirs(base_dir, exist_ok=True)
```

```
        # Fixer la seed pour la reproductibilité
        random.seed(seed)
        np.random.seed(seed)
        torch.manual_seed(seed)
```

def download_data(self):

```
    """
    Télécharge les données de RefinedWeb pour l'entraînement.
    """
    logger.info("Téléchargement des données...")
```

```
    # Utiliser RefinedWeb comme source principale
    from datasets import load_dataset
    rw = load_dataset("tiiuae/falcon-refinedweb")
```

```
    # Stocker dans notre dictionnaire de datasets
    datasets = {}
    datasets["main"] = rw
```

```
    # Pour les contextes plus longs, nous pouvons utiliser PG19
    pg19 = load_dataset("pg19", split="train")
```

```
    # Sauvegarder les données
    os.makedirs(os.path.join(self.base_dir, "raw"), exist_ok=True)
```

```
    logger.info(f"Sauvegarde du corpus RefinedWeb... ({len(datasets['main'])} exemples)")
    datasets["main"].save_to_disk(os.path.join(self.base_dir, "raw", "main"))
```

```
    logger.info(f"Sauvegarde du corpus PG19... ({len(pg19)} exemples)")
    pg19.save_to_disk(os.path.join(self.base_dir, "raw", "pg19"))
```

```
    logger.info("Téléchargement des données terminé!")
```

```
    return datasets
```

```

def train_tokenizer(self, dataset, vocab_size: int = None):
    """
    Entraîne un tokenizer WordPiece sur le dataset fourni
    """
    vocab_size = vocab_size or self.vocab_size
    logger.info(f"Entraînement du tokenizer avec une taille de vocabulaire de {vocab_size}...")

    # Créer le dossier pour le tokenizer
    tokenizer_dir = os.path.join(self.base_dir, "tokenizer")
    os.makedirs(tokenizer_dir, exist_ok=True)

    # Initialiser le tokenizer (WordPiece comme BERT original)
    tokenizer = Tokenizer(models.WordPiece(unk_token="[UNK]"))

    # Normaliser le texte
    tokenizer.normalizer = normalizers.Sequence([
        normalizers.NFD(),
        normalizers.Lowercase(),
        normalizers.StripAccents(),
    ])

    # Pre-tokenisation
    tokenizer.pre_tokenizer = pre_tokenizers.Whitespace()

    # Configurer l'entraîneur
    trainer = trainers.WordPieceTrainer(
        vocab_size=vocab_size,
        special_tokens=["[UNK]", "[CLS]", "[SEP]", "[PAD]", "[MASK]"],
    )

    # Fonction pour générer les textes pour l'entraînement du tokenizer
    def batch_iterator(batch_size=1000):
        for i in range(0, len(dataset), batch_size):
            yield dataset[i:i+batch_size]["text"]

    # Entraîner le tokenizer
    tokenizer.train_from_iterator(batch_iterator(), trainer=trainer, length=len(dataset))

    # Ajouter le post-processing pour gérer les paires de séquences (comme BERT)
    tokenizer.post_processor = processors.TemplateProcessing(
        single="[CLS] $A [SEP]",
        pair="[CLS] $A [SEP] $B [SEP]",
        special_tokens=[
            ("[CLS]", tokenizer.token_to_id("[CLS]")),
            ("[SEP]", tokenizer.token_to_id("[SEP]")),
        ],
    )

    # Ajouter le décodeur
    tokenizer.decoder = decoders.WordPiece()

    # Sauvegarder le tokenizer
    tokenizer_path = os.path.join(tokenizer_dir, "wordpiece_tokenizer.json")
    tokenizer.save(tokenizer_path)
    logger.info(f"Tokenizer sauvegardé: {tokenizer_path}")

    # Créer un tokenizer compatible avec transformers
    self.tokenizer = PreTrainedTokenizerFast(
        tokenizer_object=tokenizer,
        unk_token="[UNK]",
        cls_token="[CLS]",
        sep_token="[SEP]",
        pad_token="[PAD]",
        mask_token="[MASK]",
    )

    # Sauvegarder le tokenizer sous format transformers
    self.tokenizer.save_pretrained(tokenizer_dir)
    logger.info(f"Tokenizer HF sauvegardé: {tokenizer_dir}")

    return self.tokenizer

def tokenize_dataset(self, dataset, max_length=None, tokenizer=None):
    """
    Tokenise le dataset pour l'entraînement
    """
    max_length = max_length or self.max_length
    tokenizer = tokenizer or self.tokenizer

    if tokenizer is None:
        raise ValueError("Le tokenizer n'a pas été initialisé")

    logger.info(f"Tokenisation du dataset ({len(dataset)} exemples)...")

    # Fonction de tokenisation pour map
    def tokenize_function(examples):
        return tokenizer(
            examples["text"],
            truncation=True,
            max_length=max_length,
            return_special_tokens_mask=True,
        )

```

```

# Tokeniser le dataset
tokenized_dataset = dataset.map(
    tokenize_function,
    batched=True,
    num_proc=multiprocessing.cpu_count(),
    remove_columns=["text"],
    desc="Tokenisation du dataset",
)

return tokenized_dataset

def create_sequence_datasets(self, tokenized_dataset, lengths=[1024, 2048, 4096, 8192]):
    """
    Crée des datasets filtrés par longueur de séquence
    """
    logger.info("Création des datasets filtrés par longueur...")

    sequence_datasets = {}

    for min_length in lengths:
        # Filtrer par longueur
        filtered = tokenized_dataset.filter(
            lambda x: len(x["input_ids"]) >= min_length,
            num_proc=multiprocessing.cpu_count(),
            desc=f"Filtrage {min_length}+"
        )

        # Sauvegarder le dataset filtré
        output_dir = os.path.join(self.base_dir, "processed", f"seq_{min_length}")
        os.makedirs(output_dir, exist_ok=True)
        filtered.save_to_disk(output_dir)

        sequence_datasets[f"seq_{min_length}"] = filtered

        logger.info(f"Dataset seq_{min_length}: {len(filtered)} exemples")

    return sequence_datasets

def create_mlm_dataset(self, tokenized_dataset):
    """
    Préparation finale des données pour l'entraînement MLM
    """
    class MLMDataset(Dataset):
        def __init__(self, tokenized_dataset, tokenizer, mlm_probability=0.15, max_length=512):
            self.tokenized_dataset = tokenized_dataset
            self.tokenizer = tokenizer
            self.mlm_probability = mlm_probability
            self.max_length = max_length

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

        def __getitem__(self, idx):
            # Récupérer un exemple
            item = self.tokenized_dataset[idx]

            # Tronquer si nécessaire
            input_ids = item["input_ids"]
            if len(input_ids) > self.max_length:
                start = random.randint(0, len(input_ids) - self.max_length)
                input_ids = input_ids[start:start + self.max_length]

            # Créer le tenseur d'attention
            attention_mask = [1] * len(input_ids)

            # Gérer le padding
            padding_length = self.max_length - len(input_ids)
            if padding_length > 0:
                input_ids = input_ids + [self.tokenizer.pad_token_id] * padding_length
                attention_mask = attention_mask + [0] * padding_length

            # Créer des labels pour MLM
            labels = input_ids.copy()

            # Appliquer le masquage MLM
            # On ne masque pas [CLS], [SEP], [PAD]
            special_tokens_mask = item.get("special_tokens_mask", [0] * len(input_ids))
            probability_matrix = torch.full((len(input_ids),), self.mlm_probability)
            probability_matrix.masked_fill_(torch.tensor(special_tokens_mask, dtype=torch.bool), value=0.0)

            masked_indices = torch.bernoulli(probability_matrix).bool()
            labels = torch.tensor(labels)
            labels[~masked_indices] = -100 # We only compute loss on masked tokens

            # 80% des tokens masqués sont remplacés par [MASK]
            indices_replaced = torch.bernoulli(torch.full(labels.shape, 0.8)).bool() & masked_indices
            input_ids = torch.tensor(input_ids)
            input_ids[indices_replaced] = self.tokenizer.convert_tokens_to_ids(self.tokenizer.mask_token)

            # 10% des tokens masqués sont remplacés par un token aléatoire
            indices_random = torch.bernoulli(torch.full(labels.shape, 0.5)).bool() & masked_indices & ~indices_replaced
            random_words = torch.randint(len(self.tokenizer), labels.shape, dtype=torch.long)
            input_ids[indices_random] = random_words[indices_random]

```

```

        return {
            "input_ids": input_ids.long(),
            "attention_mask": torch.tensor(attention_mask).long(),
            "labels": labels.long(),
        }

    return MLMDataset(tokenized_dataset, self.tokenizer, self.mlm_probability, self.max_length)

def create_dataloaders(self, datasets, batch_sizes, max_lengths, ddp=False):
    """
    Crée les dataloaders pour l'entraînement avec différentes configurations
    """
    dataloaders = {}

    for name, dataset in datasets.items():
        for max_length in max_lengths:
            for batch_size in batch_sizes:
                # Créer le dataset MLM
                self.max_length = max_length
                mlm_dataset = self.create_mlm_dataset(dataset)

                # Créer le sampler pour DDP si nécessaire
                sampler = None
                if ddp:
                    sampler = DistributedSampler(mlm_dataset)

                # Créer le dataloader
                dataloader = DataLoader(
                    mlm_dataset,
                    batch_size=batch_size,
                    shuffle=(sampler is None),
                    sampler=sampler,
                    num_workers=4,
                    pin_memory=True,
                )

                key = f"{name}_len{max_length}_bs{batch_size}"
                dataloaders[key] = dataloader

            logger.info(f"Dataloader créé: {key} ({len(dataloader)} batches)")

    return dataloaders

def mixing_strategy_dataloader(self, datasets, max_length, batch_size,
                               mixing_weights=None, ddp=False):
    """
    Crée un dataloader qui mélange plusieurs datasets selon une stratégie spécifique
    """
    if mixing_weights is None:
        mixing_weights = {name: 1.0 / len(datasets) for name in datasets.keys()}

    # Normaliser les poids
    total = sum(mixing_weights.values())
    mixing_weights = {k: v / total for k, v in mixing_weights.items()}

    # Créer les datasets MLM
    mlm_datasets = {}
    for name, dataset in datasets.items():
        self.max_length = max_length
        mlm_datasets[name] = self.create_mlm_dataset(dataset)

    # Créer un dataloader mixte
    class MixedDataLoader:
        def __init__(self, dataloaders, mixing_weights):
            self.dataloaders = dataloaders
            self.mixing_weights = mixing_weights
            self.iterators = {name: iter(dl) for name, dl in self.dataloaders.items()}
            self.lengths = {name: len(dl) for name, dl in self.dataloaders.items()}
            self.total_steps = sum(
                int(length * weight) for name, (length, weight) in
                zip(self.dataloaders.keys(), zip(self.lengths.values(), self.mixing_weights.values()))
            )

        def __len__(self):
            return self.total_steps

        def __iter__(self):
            self.iterators = {name: iter(dl) for name, dl in self.dataloaders.items()}
            for _ in range(self.total_steps):
                # Choisir un dataset selon les poids
                dataset_name = random.choices(
                    list(self.mixing_weights.keys()),
                    weights=list(self.mixing_weights.values()),
                    k=1
                )[0]

                # Récupérer un batch
                try:
                    batch = next(self.iterators[dataset_name])
                except StopIteration:
                    # Réinitialiser l'iterator si nécessaire
                    self.iterators[dataset_name] = iter(self.dataloaders[dataset_name])
                    batch = next(self.iterators[dataset_name])

```

```

        yield batch

# Créer les dataloaders individuels
dataloaders = {}
for name, mlm_dataset in mlm_datasets.items():
    # Créer le sampler pour DDP si nécessaire
    sampler = None
    if ddp:
        sampler = DistributedSampler(mlm_dataset)

    # Créer le dataloader
    dataloaders[name] = DataLoader(
        mlm_dataset,
        batch_size=batch_size,
        shuffle=(sampler is None),
        sampler=sampler,
        num_workers=4,
        pin_memory=True,
    )

# Créer et retourner le dataloader mixte
mixed_dataloader = MixedDataLoader(dataloaders, mixing_weights)

return mixed_dataloader

def get_dataloaders_for_phase(self, phase, ddp=False):
    """
    Retourne les dataloaders spécifiques pour chaque phase d'entraînement
    """
    if phase == 1:
        # Phase 1: Pré-entraînement initial (1024 tokens)
        main_dataset = load_dataset("json", data_files=os.path.join(self.base_dir, "processed", "main"))
        return self.create_dataloaders(
            {"main": main_dataset},
            batch_sizes=[32],
            max_lengths=[1024],
            ddp=ddp
        )

    elif phase == 2:
        # Phase 2: Extension du contexte (4096 tokens)
        datasets = {
            "main": load_dataset("json", data_files=os.path.join(self.base_dir, "processed", "main")),
            "seq_1024": load_dataset("json", data_files=os.path.join(self.base_dir, "processed", "seq_1024")),
            "seq_2048": load_dataset("json", data_files=os.path.join(self.base_dir, "processed", "seq_2048")),
        }

        return self.mixing_strategy_dataloader(
            datasets,
            max_length=4096,
            batch_size=4,
            mixing_weights={"main": 0.2, "seq_1024": 0.4, "seq_2048": 0.4},
            ddp=ddp
        )

    elif phase == 3:
        # Phase 3: Activation MOBA (jusqu'à 32K tokens)
        datasets = {
            "main": load_dataset("json", data_files=os.path.join(self.base_dir, "processed", "main")),
            "seq_1024": load_dataset("json", data_files=os.path.join(self.base_dir, "processed", "seq_1024")),
            "seq_2048": load_dataset("json", data_files=os.path.join(self.base_dir, "processed", "seq_2048")),
            "seq_4096": load_dataset("json", data_files=os.path.join(self.base_dir, "processed", "seq_4096")),
            "seq_8192": load_dataset("json", data_files=os.path.join(self.base_dir, "processed", "seq_8192")),
        }

        return self.mixing_strategy_dataloader(
            datasets,
            max_length=32768,
            batch_size=1,
            mixing_weights={"main": 0.1, "seq_1024": 0.2, "seq_2048": 0.2, "seq_4096": 0.25, "seq_8192": 0.25},
            ddp=ddp
        )

    else:
        raise ValueError(f"Phase d'entraînement non reconnue: {phase}")

def process_complete_pipeline():
    """
    Exécute le pipeline complet de préparation des données
    """
    # Initialisation
    data_prep = DataPreparation(
        base_dir="./data",
        vocab_size=30000,
        max_length=32768,
        mlm_probability=0.15
    )

    # Téléchargement des données
    datasets = data_prep.download_data()

    # Entraînement du tokenizer
    tokenizer = data_prep.train_tokenizer(datasets["main"])

```

```
# Tokenisation du dataset principal
tokenized_main = data_prep.tokenize_dataset(datasets["main"])

# Création des datasets filtrés par longueur
sequence_datasets = data_prep.create_sequence_datasets(
    tokenized_main,
    lengths=[1024, 2048, 4096, 8192]
)

# Sauvegarde
os.makedirs(os.path.join(data_prep.base_dir, "processed"), exist_ok=True)
tokenized_main.save_to_disk(os.path.join(data_prep.base_dir, "processed", "main"))

# Création des dataloaders pour chaque phase
phase1_loaders = data_prep.get_dataloaders_for_phase(1)

logger.info("Pipeline de préparation des données terminé!")
logger.info(f"Datasets disponibles: {list(sequence_datasets.keys())}")
logger.info(f"Dataloaders pour la phase 1: {list(phase1_loaders.keys())}")

return {
    "tokenizer": tokenizer,
    "datasets": sequence_datasets,
    "dataloaders": {
        "phase1": phase1_loaders,
        "phase2": data_prep.get_dataloaders_for_phase(2),
        "phase3": data_prep.get_dataloaders_for_phase(3)
    }
}

if __name__ == "__main__":
    results = process_complete_pipeline()
    print("Préparation des données terminée!")
```