Travaux Pratiques — Cours d'Apprentissage Profond

Aurélien Plyer & Kevin Helvig

16 octobre 2025

Table des matières

Session 4 — Attention, Transformers et Vision

Cette session est consacrée à l'architecture Transformer, qui a révolutionné le traitement du langage naturel (NLP) avant de s'étendre à la vision par ordinateur. Vous implémenterez les briques de base de ce modèle, puis vous utiliserez un modèle pré-entraîné pour une tâche de classification d'images.

L'objectif est de coder vous-même les notebooks en suivant les instructions de cet énoncé.

0.1 Partie 1 : Word2Vec, Self-Attention et Mini-Transformer

Objectifs.

Implémenter un modèle Skip-Gram pour obtenir des représentations vectorielles de mots (Word2Vec). Construire une couche de self-attention multi-têtes, le cœur du Transformer. Assembler un mini-Transformer (encodeur seul) pour une tâche de classification de texte. Visualiser les cartes d'attention pour interpréter le comportement du modèle.

0.1.1 Étape 1 : Préparation du corpus de texte

Questions flash.

— Quelle est la taille du vocabulaire après filtrage ? Que fait-on des mots OOV (*out-of-vocabulary*) ?

— La tokenisation est-elle sensible à la casse et à la ponctuation ? Mesurez l'impact sur le nombre de tokens.

— Comment gérez-vous le `padding` : à gauche ou à droite ? Pourquoi est-ce important pour l'attention masquée ?

```
# Vocabulaire (minuscule + ponctuation simple séparée)
def tokenize(txt):
    return [t.strip(",.!?") for t in txt.lower().split()]

def build_synthetic_corpus(n_sentences=20000, seed=SEED):
    rng = random.Random(seed)
    countries = {
        "france": {"capital": "paris", "currency": "euro"},
        "germany": {"capital": "berlin", "currency": "euro"},
        "italy": {"capital": "rome", "currency": "euro"},
        "spain": {"capital": "madrid", "currency": "euro"},
        "japan": {"capital": "tokyo", "currency": "yen"},
```

```
    "usa": {"capital": "washington", "currency": "dollar"},
    "uk": {"capital": "london", "currency": "pound"},
    "canada": {"capital": "ottawa", "currency": "dollar"},
    "brazil": {"capital": "brasilia", "currency": "real"},
    "india": {"capital": "new_delhi", "currency": "rupee"},
}
family = [
    ("king", "queen", "man", "woman"),
    ("actor", "actress", "man", "woman"),
    ("prince", "princess", "boy", "girl"),
    ("duke", "duchess", "man", "woman"),
]
professions = ["scientist","engineer","teacher","doctor","poet"
   ,"painter","chef","musician","lawyer"]
connectors = ["while", "although", "whereas", "however"]
templates = [
    "the capital of {country} is {capital}",
    "in {country} the currency is the {currency}",
    "{A} is the male counterpart of a {B}",
    "a {A} is related to a {B} as {man} to {woman}",
    "many believe that the capital of {country} is {capital}, {
       connector} the currency of {other_country} is the {
       other_currency}",
    "people know that {city} belongs to {country}, but some
       mention {alt_city} incorrectly",
    "despite debates, {city} remains the capital of {country}",
    "experts say the {currency} is used in {country}, {
       connector} tourists spend {other_currency} in {
       other_country}",
]

cities = {v["capital"] for v in countries.values()}
sents = []
for _ in range(n_sentences):
    t = rng.choice(templates)
    c = rng.choice(list(countries.keys()))
    d = countries[c]
    oc = rng.choice([x for x in countries.keys() if x != c])
    od = countries[oc]
    if "{A}" in t:
        A,B,man,woman = rng.choice(family)
```

```
            sent = t.format(A=A, B=B, man=man, woman=woman, country
                =c, capital=d["capital"],
                                currency=d["currency"], connector=rng.
                                    choice(connectors),
                                other_country=oc, other_currency=od["
                                    currency"],
                                city=d["capital"], alt_city=rng.choice(
                                    list(cities - {d['capital']})))
        else:
            sent = t.format(country=c, capital=d["capital"],
                currency=d["currency"],
                                connector=rng.choice(connectors),
                                other_country=oc, other_currency=od["
                                    currency"],
                                city=d["capital"], alt_city=rng.choice(
                                    list(cities - {d['capital']})))
        # Ajouter du bruit lexical léger
        if rng.random() < 0.4:
            p = rng.choice(professions)
            sent = f"{sent}. in unrelated news the {p} gave a talk"
        sents.append(sent.lower())

    return sents

# Mots-clés pour la nouvelle tâche : les pays européens du corpus
EUROPEAN_COUNTRIES = {"france", "germany", "italy", "spain", "uk"}
def label_from_sentence(s):
    # La fonction tokenize est déjà définie dans votre notebook
    toks = tokenize(s)
    return int(any(tok in EUROPEAN_COUNTRIES for tok in toks))


corpus = build_synthetic_corpus(n_sentences=20000)

# Génération de la liste des étiquettes pour l'ensemble du corpus
labels = [label_from_sentence(s) for s in corpus]

# Vérifions la distribution des classes (optionnel mais recommandé)
print(f"Nombre de phrases : {len(labels)}")
print(f"Nombre d'exemples positifs (pays européen) : {sum(labels)}"
    )
```

```
print(f"Proportion d'exemples positifs : {sum(labels) / len(labels)
    :.2%}")
print(corpus[:5], len(corpus))

from collections import Counter
cnt = Counter()
for s in corpus:
    cnt.update(tokenize(s))

# Réservons des identifiants spéciaux
PAD, CLS, UNK = "<pad>", "<cls>", "<unk>"
itos = [PAD, CLS, UNK] + sorted(list(cnt.keys()))
stoi = {w:i for i,w in enumerate(itos)}

vocab_size = len(itos)
```

0.1.2 Étape 2 : Implémentation de Word2Vec (Skip-Gram)

Questions flash.

- Quel est l'effet du *window size* sur la distribution des paires (centre, contexte) ?
- Pourquoi le *negative sampling* stabilise-t-il l'apprentissage ? Quelle est l'influence du nombre de négatifs k sur le signal du gradient ?
- Comparez la norme moyenne des vecteurs d'embedding avant et après entraînement : que remarquez-vous ?

```
class SkipGramNS(nn.Module):
    def __init__(self, vocab, dim=50):
        super().__init__()
        self.in_embed = nn.Embedding(vocab, dim)
        self.out_embed = nn.Embedding(vocab, dim)
        nn.init.uniform_(self.in_embed.weight, -0.5/dim, 0.5/dim)
        nn.init.zeros_(self.out_embed.weight)
    def forward(self, center, pos, neg):
        # center: (B), pos:(B, P), neg:(B, K)
        center_v = ...              # (B, D)
        pos_v = ...                  # (B, P, D)
        neg_v = ...                    # (B, K, D)
        pos_score = ... # (B,P)
        neg_score = ...
        loss =...
```

```
        return -loss
```

0.1.3 Apprentissage

Questions flash.

— La courbe de perte sature-t-elle ? Testez deux *learning rates* différents et comparez le temps de convergence.

— Voyez-vous des signes d'*overfitting* ? Si oui, quelle régularisation simple pourriez-vous activer ?

— Quelle métrique (ex. analogies, voisinage cosinus) utilisez-vous pour évaluer la qualité des embeddings ?

```
# Négatifs selon distribution unigram^0.75
word_freq = np.array([cnt.get(w,1) for w in itos])
p_unigram = word_freq / word_freq.sum()
p_noise = p_unigram ** 0.75; p_noise /= p_noise.sum()

def sample_neg(batch_size, K):
    return torch.tensor(np.random.choice(vocab_size, size=(
       batch_size, K), p=p_noise), dtype=torch.long)

# DataLoader simple
class SGDataset(Dataset):
    def __init__(self, pairs, P=2, K=5):
        self.pairs = pairs; self.P=P; self.K=K
        # Regrouper paires par centre pour tirer P contextes
           positifs si dispo
        from collections import defaultdict
        by_c = defaultdict(list)
        for c,t in pairs:
            by_c[c].append(t)
        self.by_c = by_c
        self.centers = list(by_c.keys())
    def __len__(self): return len(self.centers)
    def __getitem__(self, idx):
        c = self.centers[idx]
        pos_list = self.by_c[c]
        pos = random.choices(pos_list, k=min(self.P, len(pos_list))
           )
        # complétion si nécessaire
```

```
        while len(pos) < self.P: pos.append(stoi[PAD])
        neg = sample_neg(1, self.K).squeeze(0).tolist()
        return c, torch.tensor(pos, dtype=torch.long), torch.tensor
           (neg, dtype=torch.long)

sg_ds = SGDataset(pairs, P=2, K=5)
sg_dl = DataLoader(sg_ds, batch_size=256, shuffle=True, num_workers
   =0, pin_memory=True)
model_sg = SkipGramNS(vocab_size, dim=50).to(device)
opt = torch.optim.Adam(model_sg.parameters(), lr=1e-3)
```

0.1.4 Étape 3 : Construction d'un Mini-Transformer

Questions flash.

- Vérifiez les dimensions : si l'entrée est (B, L, d_{model}), quelles sont les formes de Q, K, V et de la matrice d'attention ?
- Pourquoi normalise-t-on par $\sqrt{d_k}$ dans le produit scalaire ? Que se passe-t-il si on oublie cette normalisation ?
- Quelle est la différence entre *self-attention* masquée (décodage auto-régressif) et non-masquée (encodeur) ?
- À quoi sert l'encodage positionnel ? Testez un encodage appris vs sinusoidal : observez l'impact.

Le cœur du Transformer est le mécanisme de self-attention. Une fois que vous avez implémenté la classe `MultiHeadSelfAttention`, l'étape suivante est de l'intégrer dans un modèle de classification complet.

La tâche de classification. La fonction que nous allons apprendre est une classification binaire de phrases. Pour chaque phrase de notre corpus synthétique, le modèle doit prédire si elle contient ou non un nom de pays européen (défini dans `EUROPEAN_COUNTRIES`). Pour cela, on ajoute un token spécial, `[CLS]`, au début de chaque séquence. Lídée est que le Transformer, via ses couches dáttention, va agréger línformation de toute la phrase dans la représentation finale de ce token `[CLS]`. Il suffira alors de brancher une simple tête de classification (une couche linéaire) sur la sortie de ce token pour obtenir une prédiction pour toute la phrase.

Rappel de cours : Scaled Dot-Product Attention.

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{1}$$

À implémenter. Vous devez dábord implémenter la classe `MultiHeadSelfAttention` comme demandé précédemment.

```
class MultiHeadSelfAttention(nn.Module):
    def __init__(self, d_model=64, n_head=2, ...):
        # ...
    def forward(self, x, mask=None, need_weights=False):
        # TODO: Implémentez la logique de l\'attention multi-têtes.
        ...
        return out
```

Composants du Transformer. Une fois láttention codée, nous pouvons assembler les autres briques :

1. Encodage Positionnel (Positional Encoding) : La self-attention ne tient pas compte de lórdre des mots. On injecte donc une information de position dans les embeddings déntrée. On utilise pour cela des fonctions sinus et cosinus de différentes fréquences.

```
class PositionalEncoding(nn.Module):
    def __init__(self, d_model, max_len=512):
        super().__init__()
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len, dtype=torch.float).
           unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2).float() *
           (-math.log(10000.0) / d_model))
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        self.register_buffer('pe', pe.unsqueeze(0))  # Shape: (1, L
           , D)

    def forward(self, x):
        # x.shape: (B, L, D)
        return x + self.pe[:, :x.size(1), :]
```

2. Bloc Encodeur (Encoder Block) : Cést le module de base, répété N fois. Il est composé dúne couche de self-attention multi-têtes suivie dún petit réseau de neurones (MLP), avec des connexions résiduelles et une normalisation de couche (LayerNorm) après chaque sous-module.

```
class EncoderBlock(nn.Module):
    def __init__(self, d_model=64, n_head=2, mlp_ratio=2.0, drop
       =0.1):
        super().__init__()
```

```
        self.attn = MultiHeadSelfAttention(d_model, n_head,
            attn_drop=drop, proj_drop=drop)
        self.norm1 = nn.LayerNorm(d_model)
        self.mlp = nn.Sequential(
            nn.Linear(d_model, int(d_model * mlp_ratio)),
            nn.GELU(),
            nn.Dropout(drop),
            nn.Linear(int(d_model * mlp_ratio), d_model),
            nn.Dropout(drop),
        )
        self.norm2 = nn.LayerNorm(d_model)

    def forward(self, x, mask=None, need_weights=False):
        # Connexion résiduelle autour de l\'attention
        y, A = self.attn(self.norm1(x), mask=mask, need_weights=
            True)
        x = x + y
        # Connexion résiduelle autour du MLP
        x = x + self.mlp(self.norm2(x))

        if need_weights:
            return x, A
        return x
```

3. Le Classifieur Transformer (TinyTransformerClassifier) : Ce modèle final assemble toutes les pièces : la couche démbedding (figée), léncodage positionnel, une ou plusieurs couches déncodeur, et la tête de classification finale.

```
class TinyTransformerClassifier(nn.Module):
    def __init__(self, vocab_size, w2v_weights, d_model=64, n_head
        =2, n_layers=1, n_cls=2, max_len=128, freeze_embed=True):
        super().__init__()
        # Couche d\'embedding initialisée avec les poids Word2Vec
        self.embed = nn.Embedding(vocab_size, w2v_weights.size(1))
        with torch.no_grad():
            self.embed.weight[:] = w2v_weights
        if freeze_embed:
            for p in self.embed.parameters(): p.requires_grad =
                False

        # Projection vers la dimension du modèle
        self.proj = nn.Linear(w2v_weights.size(1), d_model)
```

```
        self.pos = PositionalEncoding(d_model, max_len=max_len)
        self.layers = nn.ModuleList([EncoderBlock(d_model, n_head)
            for _ in range(n_layers)])
        self.norm = nn.LayerNorm(d_model)
        self.head = nn.Linear(d_model, n_cls)

    def forward(self, x, need_weights=False):
        mask = (x != stoi[PAD]).long()
        h = self.proj(self.embed(x))
        h = self.pos(h)

        attn_maps = []
        for layer in self.layers:
            h, A = layer(h, mask=mask, need_weights=True)
            attn_maps.append(A)

        h = self.norm(h)
        # Pooling : on utilise uniquement la sortie du token [CLS]
        cls_output = h[:, 0, :]
        logits = self.head(cls_output)

        if need_weights:
            return logits, attn_maps, mask
        return logits
```

Instructions pour la boucle dápprentissage. Pour entraîner ce classifieur, vous devez :

1. **Préparer les données** : Créez un `Dataset` qui transforme chaque phrase en une séquence dÍDs de tokens, préfixée par `[CLS]` et paddée à une longueur fixe. Séparez vos données en ensembles déntraînement et de validation.
2. **Instancier le modèle** : Créez une instance de `TinyTransformerClassifier`.
3. **Définir la perte et lóptimiseur** : Utilisez `nn.CrossEntropyLoss` pour la classification et un optimiseur comme `torch.optim.AdamW`.
4. **Écrire la boucle** : Pour chaque époque, itérez sur les batchs du `DataLoader`. Dans la boucle, effectuez les étapes suivantes : passage avant (forward pass), calcul de la perte, mise à zéro des gradients, rétropropagation (`loss.backward()`), et mise à jour des poids (`optimizer.step()`).
5. **Évaluer le modèle** : À la fin de chaque époque, évaluez la performance (perte et précision) sur lénsemble de validation. Assurez-vous de passer le modèle en

mode évaluation (`model.eval()`) et de désactiver le calcul des gradients (`with torch.no_grad():...`).

0.1.5 Étape 4 : Visualisation des cartes dáttention

Questions flash.

— Les têtes se spécialisent-elles sur des motifs linguistiques distincts (négation, accord, entités) ?

— L'attention *explique-t-elle* vraiment la prédiction ? Cherchez un contre-exemple où la carte d'attention est trompeuse.

— Mesurez la stabilité des cartes d'attention lorsque vous perturbez légèrement l'entrée (remplacement d'un mot par un synonyme).

Une fois le modèle entraîné, utilisez la sortie $\texttt{attn}_m aps pour visualiser ce que le modle a appris. Analysez com$

0.2 Partie 2 : Vision Transformer (ViT) et Augmentation de Données

Objectifs.

- Comprendre la structure de sortie d'un Vision Transformer (tokens de patchs + token [CLS]).
- Implémenter vous-même une *tête de classification* (MLP) placée au-dessus du ViT (backbone gelé ou partiellement dégelé).
- Expérimenter l'impact des augmentations de données (faible → forte) sur la convergence et la généralisation.

Jeu de données. Nous utiliserons Hotdog / Not Hotdog (Hugging Face: truepositive/hotd

0.2.1 Étape A : Charger un ViT pré-entraîné comme *backbone*

- Exemple recommandé : vit-base-patch16-224. Le modèle segmente une image $H \times W$ en patchs de taille $P \times P$, aplatis et projetés dans un espace de dimension D (*hidden_size*).
- Formule des tokens : $N = \lfloor H/P \rfloor \times \lfloor W/P \rfloor$ patchs + un token [CLS]. La sortie last_hidden_state a la forme $(B, 1 + N, D)$.
- *Bon réflexe :* ne supposez pas les valeurs; interrogez la configuration du modèle (model.config.hidden_size, model.config.image_size, model.config.patch_

  Questions flash.

  - Pour vit-base-patch16-224, calculez N et vérifiez que la taille de sortie est bien $(B, 197, 768)$. Où apparaît le 197 ?

- Comparez pooler_output (s'il existe) vs le [CLS] brut : lequel utiliser pour la tête ? Justifiez empiriquement.

0.2.2 Étape B : Implémenter la tête MLP

Tâche. Écrivez une classe ClassificationHead qui prend en entrée un vecteur de dimension D et produit C logits.

- Entrée : le vecteur [CLS] (forme (B, D)) ou une agrégation (p. ex. moyenne des tokens patchs).
- Architecture minimale : Linear(D, C). Option : Linear(D, H) $\rightarrow$ ReLU $\rightarrow$ Dropout $\rightarrow$ Linear(H, C).
- Dimensionnement : calculez D *à partir du modèle*. Choisissez $H \in \{D/4, D/2, D\}$ et comparez.
- Apprentissage : commencez par *geler* le ViT et n'entraîner que la tête. Puis dégelez le dernier bloc d'attention pour un léger *fine-tuning*.

Questions flash.

- Que se passe-t-il si vous vous trompez de D (erreur de forme) ? Écrivez une assert qui protège contre les incompatibilités.
- Mesurez l'écart de performance entre tête *linéaire* et tête *à 1 couche cachée*. Sur-apprend-on plus vite ?
- Quelle est l'influence du dropout de la tête lorsque le backbone est gelé ? Et lorsqu'il est partiellement dégelé ?

0.2.3 Étape C : Pipeline d'entraînement et métriques

- Utilisez CrossEntropyLoss et un optimiseur type AdamW. Fixez un batch_size modéré (p. ex. 32) et entraînez 5-10 époques.
- Suivis obligatoires : courbes perte/accuracy train vs val, matrice de confusion, F1 macro (classes potentiellement déséquilibrées).
- Journalisation : consignez les hyperparamètres (seed, LR, scheduler, poids gelés).

Questions flash.

- Observez l'écart train-val. Quand l'overfitting commence-t-il ? *Indice :* regardez la divergence des courbes.
- La tête linéaire converge-t-elle plus vite que la tête MLP ? Expliquez par la capacité et la régularisation implicite.

0.2.4 Étape D : Augmentations de données ⇒ généralisation

Protocole. Comparez au moins 3 configurations, toutes choses égales par ailleurs (mêmes splits, seed, LR) :

a) Baseline : Resize(256) → CenterCrop(224) + normalisation ImageNet.

b) Léger : RandomResizedCrop(224) + RandomHorizontalFlip.

c) Moyen : (léger) + ColorJitter + RandomErasing.

d) Fort : ajoutez RandAugment *ou* AutoAugment. *Option* : Mixup/CutMix (sur les images et labels).

À mesurer. Pour chaque configuration : meilleure accuracy val, F1 macro, temps par époque, et *robustesse* (évaluez sur des recadrages/éclairages non vus). Tracez un barplot des performances et discutez qualitativement des images transformées.

Questions flash.

- Pourquoi les ViT bénéficient-ils particulièrement d'augmentations fortes sur petits datasets ? *Piste :* faible biais inductif vs CNN.
- À partir de quel niveau d'augmentation la performance décroît-elle (sur-apprentissage aux artefacts) ?
- Mixup/CutMix changent la loi des labels. Comment adapter la perte et les métriques à ces labels *mous* ?

0.2.5 Étape E : Interpréter (bonus)

- Attention rollout sur le token [CLS] pour produire une carte de saillance (cf. références). Comparez avant/après augmentation.
- Occlusion par patch : masquez un patch et observez la baisse du logit. Identifiez les régions discriminantes.

Livrables.

- Code de la tête ClassificationHead (avec assert sur la dimension d'entrée).
- Tableau comparatif des configurations d'augmentation (accuracy, F1, temps/ép.).
- 2-3 figures (courbes, matrice de confusion, cartes d'attention/occlusion en bonus) avec commentaires.