RÉPUBLIQUE FRANÇAISE
Liberté
Égalité
Fraternité

METEO FRANCE
À VOS CÔTÉS, DANS UN CLIMAT QUI CHANGE

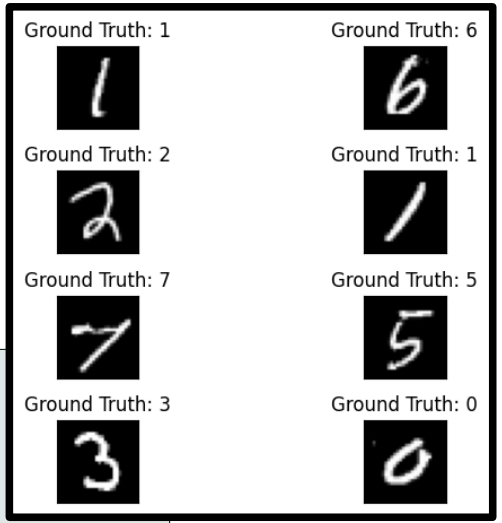
Retour sur le TP n°3 (entraînement d'un CNN)
Présentation du TP n°4 (Transformers)

Pierre Lepetit
ENM, le 7/11/2025

Retour sur le TP 3 :

Nous avons vu comment entraîner un CNN à la classification...

- ...sur un problème de reconnaissance de chiffres manuscrits (I)
  - Construction d'un « Vanilla CNN »
  - Meilleur scores qu'un MLP

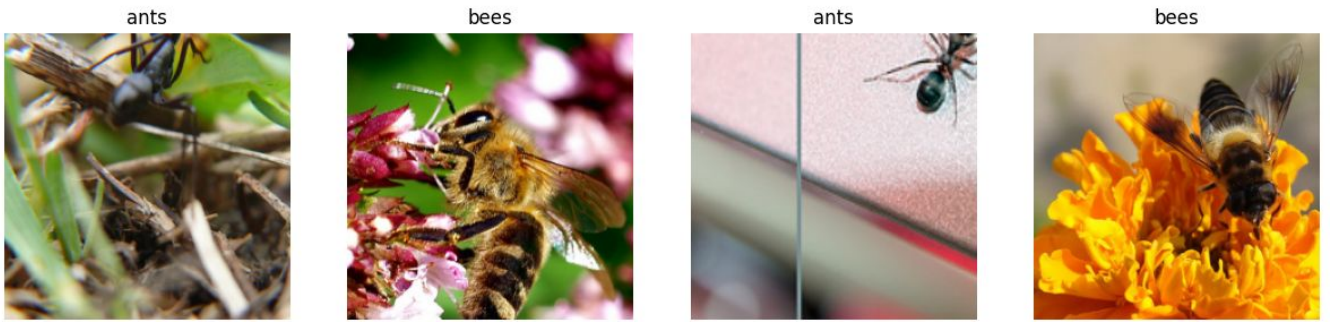



```
class CNN(nn.Module): # Vanilla CNN

    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5, padding=2)
        self.conv2 = nn.Conv2d(10, 10, kernel_size=5, padding=2)
        self.fc1 = nn.Linear(N, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2(x), 2))
        x = x.view(-1, N)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        x = self.fc2(x)
        return F.log_softmax(x, dim=1)
```

Retour sur le TP 3 :

Nous avons vu comment entraîner un CNN à la classification...

- ...sur un problème de reconnaissance de chiffres manuscrits (I)
  - Construction d'un « Vanilla CNN »
  - Meilleur scores qu'un MLP
- ...sur un problème de classification binaire (Hymenoptera dataset)
  - Paramétrage de l'augmentation de données (II.A)



Retour sur le TP 3 :

Nous avons vu comment entraîner un CNN à la classification...

- ...sur un problème de reconnaissance de chiffres manuscrits (I)
  - Construction d'un « Vanilla CNN »
  - Meilleur scores qu'un MLP
- ...sur un problème de classification binaire (Hymenoptera dataset)
  - Paramétrage de l'augmentation de données (II.A)
  - Modification d'un réseau sur étagère (bib pytorch) pré-entraîné(II.B)

```
model = models.resnet18(pretrained=True)
num_ftrs = model.fc.in_features
model.fc = nn.Linear(num_ftrs, 2)
```

Retour sur le TP 3 :

Nous avons vu comment entraîner un CNN à la classification...

- ...sur un problème de reconnaissance de chiffres manuscrits (I)
  - Construction d'un « Vanilla CNN »
  - Meilleur scores qu'un MLP
- ...sur un problème de classification binaire (Hymenoptera dataset)
  - Paramétrage de l'augmentation de données (II.A)
  - Modification d'un réseau sur étagère (bib pytorch) pré-entraîné(II.B)
  - Passage de la descente de gradient sur GPU (II.C) → speed-up 20x

```
model = get_model(pretrained=False).to(device)
optimizer = torch.optim.SGD(model.parameters(), lr=0.001)

for inputs, labels in dataloaders[phase]:
    inputs = inputs.to(device)
    labels = labels.to(device)
    ...
```

Retour sur le TP 3 :

Nous avons vu comment entraîner un CNN à la classification...

- ...sur un problème de reconnaissance de chiffres manuscrits (I)
  - Construction d'un « Vanilla CNN »
  - Meilleur scores qu'un MLP
- ...sur un problème de classification binaire (Hymenoptera dataset)
  - Paramétrage de l'augmentation de données (II.A)
  - Modification d'un réseau sur étagère (bib pytorch) pré-entraîné(II.B)
  - Passage de la descente de gradient sur GPU (II.C)
  - Comparaison entre deux méthodes de Fine-Tuning (II.D, avec ou sans « Freezing »)

```
# resnet_scratch = get_model(pretrained=False)

# Fine-tuning :
resnet_ft = get_model(pretrained=True)

# Gel des poids (freezing):
for module in resnet.modules() :
  if isinstance(module, nn.Conv2d) or isinstance(module, nn.BatchNorm2d):
    for param in module.parameters():
      param.requires_grad = False
```

II) Les Transformers

Rapide historique

- 1950s – 1990s : NLP symbolique, modèles statistiques (e.g. : n-grams)
- 2001 : Neural Language Models (Bengio et al.) → neural embedding
- 2013 - 2015 : Word2Vec, RNNs (eg : seq2seq for translation) + Attention mecanisms
- 2017 : Attention is all you need (transformers)
- 2018 : BERT → bidirectional masking ; breakthrough in how to fine-tune
- 2018 - 2019 : GPT1-2 autoregressive transformer + scaling laws
- 2020 : T5, BART, ELECTRA, ViT (ViT : first Vision Transformer)
- 2021 – 2022 : sparse/efficient Attention (Longformer, Performer, FlashAttention etc)

II) Les Transformers

Ce dont on parlera :

- 1950s – 1990s : NLP symbolique, modèles statistiques (e.g. : n-grams)
- 2001 : Neural Language Models (Bengio et al.) → neural embedding
- 2013 - 2015 : Word2Vec, RNNs (eg : seq2seq for translation) + Attention mecanisms
- 2017 : Attention is all you need (transformers)
- 2018 : BERT → biderectional masking ; breakthrough in how to fine-tune
- 2018 - 2019 : GPT1-2 autoregressive transformer + scaling laws
- 2020 : T5, BART, ELECTRA, ViT (ViT : first Vision Transformer)
- 2021 – 2022 : sparse/efficient Attention (Longformer, Performer, FlashAttention etc)

II) Les Transformers

Tokenisation, embedding et couche attentionnelle

- **Tokenisation** : input → suite finie de L tokens

  Exemples:

  NLP : standardisation du prompt + segmentation (e.g. **Byte Pair Encoding**)

  Computer Vision : découpe en patchs et applatissement

Corpus : {Les}{poules}{du}{couvent}{couvent}{souvent}

Etape 1 : segmentation

les → l e s </w>
poules → p o u l e s </w>
du → d u </w>
couvent → c o u v e n t </w>
couvent → c o u v e n t </w>
souvent → s o u v e n t </w>

II) Les Transformers

Tokenisation, embedding et couche attentionnelle

- **Tokenisation** : input → suite finie de L tokens

  Exemples:

  NLP : standardisation du prompt + segmentation (e.g. **Byte Pair Encoding**)

  Computer Vision : découpe en patchs et applatissement

Corpus : {Les}{poules}{du}{couvent}{couvent}{souvent}

Etape 2 : décompte des digrammes et fusion

(l,e):1, (e,s):1,
(p,o):1, (o,u):4, (u,l):1, (l,e):1, (e,s):1,
(d,u):1, (c,o):2, (u,v):3, (v,e):3, (e,n):3, (n,t):3, (s,o):1

poules → p ou l e s </w>
couvent → c ou v e n t </w>
souvent → s ou v e n t </w>

II) Les Transformers

Tokenisation, embedding et couche attentionnelle

- **Tokenisation** : input → suite finie de L tokens

  Exemples:

  NLP : standardisation du prompt + segmentation (e.g. **Byte Pair Encoding**)

  Computer Vision : découpe en patchs et applatissement

Corpus : {Les}{poules}{du}{couvent}{couvent}{souvent}

Etape 3 : décompte des digrammes et fusion

…, (ou, v): 3, (v, e): 3, (e, n): 3, (n, t): 3

couvent → c ouv e n t </w>

souvent → s ouv e n t </w>

II) Les Transformers

Tokenisation, embedding et couche attentionnelle

- **Tokenisation** : input → suite finie de L tokens

  Exemples:

  NLP : standardisation du prompt + segmentation (e.g. **Byte Pair Encoding**)

  Computer Vision : découpe en patchs et applatissement

Corpus : {Les}{poules}{du}{couvent}{couvent}{souvent}

Etape 3 : décompte des digrammes et fusion
…, (ouv, e): 3, (e, n): 3, (n, t): 3
couvent → c ouve n t </w> souvent → s ouve n t </w>

Etape 4 : couvent → c ouven t </w>, souvent → s ouven t </w>
Etape 5 : couvent → c ouven t </w>, souvent → s ouven t </w>

II) Les Transformers

Tokenisation, embedding et couche attentionnelle

- **Tokenisation** : input → suite finie de L tokens

  Exemples:

  NLP : standardisation du prompt + segmentation (e.g. **Byte Pair Encoding**)

  Computer Vision : découpe en patchs et applatissement

Corpus : {Les}{poules}{du}{couvent}{couvent}{souvent}

Après l'Etape 5 :

Vocabulaire : {l, e, s, p, o, u, d, c, v, n, t, ou, ouv, ouve, ouven, ouvent} → 16 (vs 11)

Longueur de la phrase « L.e.s p.ou.l.e.s d.u c.ouvent c.ouvent s.ouvent» → 16 (vs 32)

NB : BPE de GPT2 : 50 K tokens, 3 à 4 chars/token (Candide : 3.72)

Tokens IDs: [829, 279, 280, 829, 7043, 2284, 1151, 2284, 1151, 24049, 1151]

Tokens: ['les', ' p', 'ou', 'les', ' du', ' cou', 'vent', ' cou', 'vent', ' sou', 'vent']

II) Les Transformers

Tokenisation, embedding et couche attentionnelle

- **Tokenisation** : input → suite finie de L tokens

- **Embedding** : token (et position) → L tenseurs X_i
  Exemples :

  NLP : token IDs → vecteurs entraînables de taille d_model

```
# scratch embedding for NLP :
embed = nn.Embedding(vocab_size, d_model)  # matrix vocab_size x d_model

# pretrained embedding for NLP :
embed  = GPT2Model.from_pretrained("gpt2").wte # 50.257 x 768
embed.weight[1151] # Embedding du token ‘vent’ dans gpt2

# scratch positional embedding : max_length = 1 K (GPT 2)
pos = torch.nn.Parameter(torch.randn(1, max_len, d_model))

# learned positional embedding (GPT2)
pos_vec = GPT2Model.from_pretrained("gpt2").wpe.weight[5] # p.e. de ’cou’
```

II) Les Transformers

Tokenisation, embedding et couche attentionnelle

- **Tokenisation** : input → suite finie de L tokens

- **Embedding** : token (et position) → L tenseurs X_i
  Exemples :

  NLP : token IDs → vecteurs entraînables de taille d_model

```
# scratch embedding for NLP :
embed = nn.Embedding(vocab_size, d_model)  # matrix vocab_size x d_model

# pretrained embedding for NLP :
embed  = GPT2Model.from_pretrained("gpt2").wte # 50.257 x 768
embed.weight[1151] # Embedding du token ‘vent’ dans gpt2

# scratch positional embedding : max_length = 1 K (GPT 2)
pos = torch.nn.Parameter(torch.randn(1, max_len, d_model))

# learned positional embedding (GPT2)
pos_vec = GPT2Model.from_pretrained("gpt2").wpe.weight[5] # p.e. de ’cou’
```

II) Les Transformers

Tokenisation, embedding et couche attentionnelle

- **Tokenisation** : input → suite finie de L tokens

- **Embedding** : token (et position) → L tenseurs X_i
  Exemples :
  NLP : token IDs → vecteurs entraînables de taille D_{model}
  CV : 3 x 16 x 16 patches → conv2d(stride=16)
  → vecteurs de taille D_{model}

```
# scratch embedding for CV :
patch_size = 16           # 16×16 pixels

patch_embed = nn.Conv2d(in_channels=3, out_channels=d_model,
                        kernel_size=patch_size, stride=patch_size)
x = patch_embed(img).flatten(2).transpose(1, 2)  # (batch_size, N_patches, d_model)

# scratch positional embedding : 196 patches in ViT / 197 tokens
pos_embed   = nn.Parameter(torch.zeros(1, num_patches + 1, d_model))
```

II) Les Transformers

Tokenisation, embedding et couche attentionnelle

- **Tokenisation** : input → suite finie de L tokens

- **Embedding** : token (et position) → L tenseurs Xi

- **Couche auto-attentionnelle** (one-headed) :
  Paramètres : W_q , W_k , W_v de tailles d_model x d_kqv
  W_o de taille d_qkv x d_model

  L'attention comporte cinq phases :
  Projection : $Q = X W_q$, $K = X W_k$, $V = X W_v$
  Produits scalaires : $S = Q \cdot {}^tK$
  Score d'attention : A = softmax(S / sqrt(d_qkv)) # (scaling + soft selection + norm.)
  Agrégation : $X' = A V$
  Reprojection : $Y := X' W_o$

II) Les Transformers

Tokenisation, embedding et couche attentionnelle

- **Tokenisation** : input → suite finie de L tokens

- **Embedding** : token (et position) → L tenseurs Xi

- **Couche auto-attentionnelle** (one-headed) :
  Paramètres : W_q , W_k , W_v de tailles d_model x d_kqv
  W_o de taille d_qkv x d_model

  L'attention comporte cinq phases :
  Projection : $Q = X W_q$, $K = X W_k$, $V = X W_v$ # L x d_qkv
  Produits scalaires : $S = Q \cdot {}^tK$ # L x L
  Score d'attention : $A = \text{softmax}(S / \text{sqrt}(d\_qkv))$ # L x L
  Agrégation : $X' = A V$ # L x d_qkv
  Reprojection : $Y := X' W_o$ # L x d_model

II) Les Transformers

Tokenisation, embedding et couche attentionnelle

- **Tokenisation** : input → suite finie de L tokens
- **Embedding** : token (et position) → L tenseurs Xi
- **Couche auto-attentionnelle** (one-headed) :

```
class SelfAttention(nn.Module):
   def __init__(self, d_model):
       self.Wq = nn.Linear(d_model, d_qkv, bias=False)
       ...
       self.Wo = nn.Linear(d_model, d_model, bias=False)

   def forward(self, X):                                        # X: (B, L, d_model)
       Q, K, V = self.Wq(X), self.Wk(X), V = self.Wv(X)         # (B, L, d_qkv)

       scores = Q @ K.transpose(-2, -1)                         # (B, L, L)
       scores = (scores / math.sqrt(Q.size(-1))).softmax(dim=-1) # (B, L, L)

       out = self.Wo(A @ V)                                     # (B, L, d_model)
       return out
```

II) Les Transformers

Tokenisation, embedding et couche attentionnelle

- **Tokenisation** : input → suite finie de L tokens
- **Embedding** : token (et position) → L tenseurs Xi
- **Couche auto-attentionnelle** (one-headed) :

```
class SelfAttention(nn.Module):
   def __init__(self, d_model):
       self.Wq = nn.Linear(d_model, d_qkv, bias=False)
       ...
       self.Wo = nn.Linear(d_model, d_model, bias=False)

   def forward(self, X):                                    # X: (B, L, d_model)
       Q, K, V = self.Wq(X), self.Wk(X), V = self.Wv(X)     # (B, L, d_qkv)

       scores = Q @ K.transpose(-2, -1)                     # (B, L, L)
       scores = (scores / math.sqrt(Q.size(-1))).softmax(dim=-1) # (B, L, L)

       out = self.Wo(A @ V)                                 # (B, L, d_model)
       return out
```

Non linéarités

II) Les Transformers

Tokenisation, embedding et couche attentionnelle

- **Tokenisation** : input → suite finie de L tokens
- **Embedding** : token (et position) → L tenseurs Xi
- **Couche auto-attentionnelle** (one-headed) :

```
class SelfAttention(nn.Module):
   def __init__(self, d_model):
       self.Wq = nn.Linear(d_model, d_qkv, bias=False)
       ...
       self.Wo = nn.Linear(d_model, d_model, bias=False)

   def forward(self, X):                                    # X: (B, L, d_model)
       Q, K, V = self.Wq(X), self.Wk(X), V = self.Wv(X)     # (B, L, d_qkv)

       scores = Q @ K.transpose(-2, -1)                     # (B, L, L)
       scores = (scores / math.sqrt(Q.size(-1))).softmax(dim=-1) # (B, L, L)

       out = self.Wo(A @ V)                                 # (B, L, d_model)
       return out
```

Côut : $O(B.L^2.d\_qkv)$

II) Les Transformers

Tokenisation, embedding et couche attentionnelle

- **Tokenisation** : input → suite finie de L tokens

- **Embedding** : token (et position) → L tenseurs Xi

- **Couche auto-attentionnelle** (multi-headed) :
  Paramètres : 3 x h matrices $W_{q,i}$, $W_{k,i}$, $W_{v,i}$ de tailles d_model x d_kqv
  W_o de taille (h x d_qkv) x d_model

  L'attention comporte cinq phases :

  Projection : $Q_i = X W_{q,i}$, $K_i = X W_{k,i}$, $V_i = X W_{v,i}$ # Q,K,V : h x L x d_qkv
  Produits scalaires : $S_i = Q_i \cdot {}^tK_i$ # S : h x L x L
  Score d'attention : A_i = softmax(S_i / sqrt(d_qkv)) # A : h x L x L
  Agrégation : $X'_i = A_i V_i$ # X' : h x L x d_qkv
  Concaténation : $X' = \text{Concat}(X'_1, \ldots, X'_h)$ # X' : L x (h x d_qkv)
  Reprojection : $Y := X' W_o$ # L x d_model

II) Les Transformers

```
class MultiHeadSelfAttention(nn.Module):
    def __init__(self, d_model, h, d_k):
        super().__init__()
        self.Wq = nn.Linear(d_model, h*d_k, bias=False)
        ...
        self.Wo = nn.Linear(h*d_k, d_model, bias=False)

    def forward(self, X):                                   # X: (B, L, d_model)
        Q, K, V = self.Wq(X), self.Wk(X), self.Wv(X)        # (B, L, h*d_k)
        Q = Q.view(B, L, h, d_k).transpose(1, 2)            # (B, h, L, d_k)
        K = ...
        V = ...
        scores = Q @ K.transpose(-2, -1)                    # (B, h, L, L)
        A = (scores / math.sqrt(d_k)).softmax(dim=-1)       # (B, h, L, L)
        Xh = A @ V                                          # (B, h, L, d_k)
        Xh = Xh.transpose(1, 2).reshape(B, L, h*d_k)        # (B, L, h*d_k)
        out = self.Wo(Xh)                                   # (B, L, d_model)
        return out
```

II) Les Transformers

```
class MultiHeadSelfAttention(nn.Module):
    def __init__(self, d_model, h, d_k):
        super().__init__()
        self.Wq = nn.Linear(d_model, h*d_k, bias=False)
        ...
        self.Wo = nn.Linear(h*d_k, d_model, bias=False)

    def forward(self, X):                                  # X: (B, L, d_model)
        Q, K, V = self.Wq(X), self.Wk(X), self.Wv(X)       # (B, L, h*d_k)
        Q = Q.view(B, L, h, d_k).transpose(1, 2)           # (B, h, L, d_k)
        K = ...
        V = ...
        scores = Q @ K.transpose(-2, -1)                   # (B, h, L, L)
        A = (scores / math.sqrt(d_k)).softmax(dim=-1)      # (B, h, L, L)
        Xh = A @ V                                         # (B, h, L, d_k)
        Xh = Xh.transpose(1, 2).reshape(B, L, h*d_k)       # (B, L, h*d_k)
        out = self.Wo(Xh)                                  # (B, L, d_model)
        return out
```

Côut : $O(B.h.L^2.d\_qkv)$

Pourquoi pas une seule « grosse » tête ? (e.g. h = 1, d_qkv = d_model)

II) Les Transformers

```
class MultiHeadSelfAttention(nn.Module):
    def __init__(self, d_model, h, d_k):
        super().__init__()
        self.Wq = nn.Linear(d_model, h*d_k, bias=False)
        ...
        self.Wo = nn.Linear(h*d_k, d_model, bias=False)

    def forward(self, X):                                  # X: (B, L, d_model)
        Q, K, V = self.Wq(X), self.Wk(X), self.Wv(X)       # (B, L, h*d_k)
        Q = Q.view(B, L, h, d_k).transpose(1, 2)           # (B, h, L, d_k)
        K = ...
        V = ...
        scores = Q @ K.transpose(-2, -1)                   # (B, h, L, L)
        A = (scores / math.sqrt(d_k)).softmax(dim=-1)      # (B, h, L, L)
        Xh = A @ V                                         # (B, h, L, d_k)
        Xh = Xh.transpose(1, 2).reshape(B, L, h*d_k)       # (B, L, h*d_k)
        out = self.Wo(Xh)                                  # (B, L, d_model)
        return out
```

Côut : $O(B.h.L^2.d\_qkv)$

Pourquoi pas une seule « grosse » tête ? (e.g. h = 1, d_qkv = d_model)

→ spécialisation + Q @ ^{t}K, etc, parallélisables

II) Projets : au moins 4 groupes de plus de 3 étudiants

- **Suivi des mouvements d'une caméra** :
  - Input : 500 groupes de 20 images prises par caméra « fixe »
  - Objectif : apprendre à suivre les mouvement de l'axe de visée au cours du temps (dus aux var. de température, à la maintenance, aux changements de paramétrisation)
- **Détection des inondations** :
  - Input : archives de quelques centaines de caméras qui montrent des variations du niveau d'eau visible (bord de rivière, prés inondables, bord de mer)
  - Objectif : parvenir à annoter les séries d'images convenablement, entraîner un modèle à suivre les variations sur :
    1/ les caméras vues à l'entrainement
    2/ de nouvelles caméras
- **Typage des nuage** :
  - Input : codes pour une annotation crowdsourcée, images des caméras ENM + sat.
  - Objectif : prédiction des **types de nuage** + **hauteur** + **nébul** sur les mêmes caméras

II) Les Transformers

Présentation du TP n°4 :

- Visualisation des Q, K, V
- Multihead attentional layers
- Présentation d'exemples jouets :
  - Faciles à simuler
  - Pour lesquels un « vanilla CNN » << Transformer
- Effet de la dilation (?)