Réseau neuronales et Apprentissage Profond

Présenté par: Soufyane MAJDOUB
soufyane.majdoub@usmba.ac.ma

LISAC Laboratory, Computer science department, Faculty of Sciences Dhar El Mahraz, Sidi Mohamed Ben Abdellah University, Fez, Morocco.

Le: May 29, 2025

Réseaux de Neurones Récurrents : Généralités

- Les RNN sont des réseaux conçus pour traiter des **séquences** de données (texte, signal temporel, etc.), en modélisant les dépendants entre éléments successifs.
- Idée clé : **mémoire interne** qui se met à jour à chaque pas de la séquence, permettant de conserver un contexte du passé.
- Contrairement à un réseau classique (MLP) qui traite une entrée de dimension fixe, un RNN peut ingérer des séquences de longueur variable en procesant un élément après l'autre.
- Applications typiques : *modélisation de langage* (prédire le prochain mot), *traduction automatique*, *analyse de sentiments* sur du texte, *prévision de séries temporelles*, etc.

Différents types d'architectures RNN

- **One-to-One** : un seul input fixe $\rightarrow$ un seul output fixe (cas classique, pas récurrent).
- **Many-to-One** : une séquence en entrée $\rightarrow$ une seule sortie (ex: classification de phrase).
- **One-to-Many** : un seul input fixe $\rightarrow$ une séquence en sortie (ex: génération de légende à partir d'une image).
- **Many-to-Many** :
  - *Aligné* : séquence $\rightarrow$ séquence de même longueur (ex: tagger chaque mot d'une phrase).
  - *Non-aligné (Seq2Seq)* : séquence $\rightarrow$ séquence de longueur différente (ex: traduction, avec encodeur récurrent puis décodeur récurrent).

Formulation mathématique d'un RNN

Soit $(x_1, x_2, \ldots, x_T)$ une séquence d'entrée. Le RNN calcule pour chaque pas de temps t :

$$h_t = f(W_{xh}\, x_t + W_{hh}\, h_{t-1} + b_h)$$

où h_t est l'**état caché** au temps t (vecteur de dimension cachée), f est une fonction d'activation (par ex. $\tanh$), et h_0 est initialisé (ex: vecteur nul).

Formulation mathématique d'un RNN

Soit $(x_1, x_2, \ldots, x_T)$ une séquence d'entrée. Le RNN calcule pour chaque pas de temps t :

$$h_t = f(W_{xh}\, x_t + W_{hh}\, h_{t-1} + b_h)$$

où h_t est l'**état caché** au temps t (vecteur de dimension cachée), f est une fonction d'activation (par ex. $\tanh$), et h_0 est initialisé (ex: vecteur nul).

$$y_t = g(W_{hy}\, h_t + b_y)$$

éventuellement, une **sortie** y_t est produite à chaque étape (selon la tâche) via une fonction g (par ex. softmax pour une probabilité sur des classes).

Formulation mathématique d'un RNN

Soit $(x_1, x_2, \ldots, x_T)$ une séquence d'entrée. Le RNN calcule pour chaque pas de temps t :

$$h_t = f(W_{xh}\, x_t + W_{hh}\, h_{t-1} + b_h)$$

où h_t est l'**état caché** au temps t (vecteur de dimension cachée), f est une fonction d'activation (par ex. $\tanh$), et h_0 est initialisé (ex: vecteur nul).

$$y_t = g(W_{hy}\, h_t + b_y)$$

éventuellement, une **sortie** y_t est produite à chaque étape (selon la tâche) via une fonction g (par ex. softmax pour une probabilité sur des classes). Les poids W_{xh}, W_{hh}, W_{hy} et biais b_h, b_y sont partagés *à travers le temps* et appris par **rétropropagation dans le temps (BPTT)**.

Entraînement des RNN : Backpropagation Through Time

- Pour entraîner un RNN, on **“déplie”** le réseau dans le temps : on le considère comme T copies successives (une par pas de temps) partageant les mêmes poids.
- On applique alors l’algorithme de **rétropropagation du gradient** à ce réseau déplié, en accumulant les gradients sur chaque copie temporelle.
- On parle de *Backpropagation Through Time (BPTT)*. Sa complexité est proportionnelle à la longueur de la séquence (plus de pas = réseau déplié plus long).
- Sur de longues séquences, on utilise parfois une **BPTT tronquée** : on déplie le réseau seulement sur un nombre fixe de pas (ex: 20 derniers pas) pour limiter les coûts de calcul et les très longues dépendances.

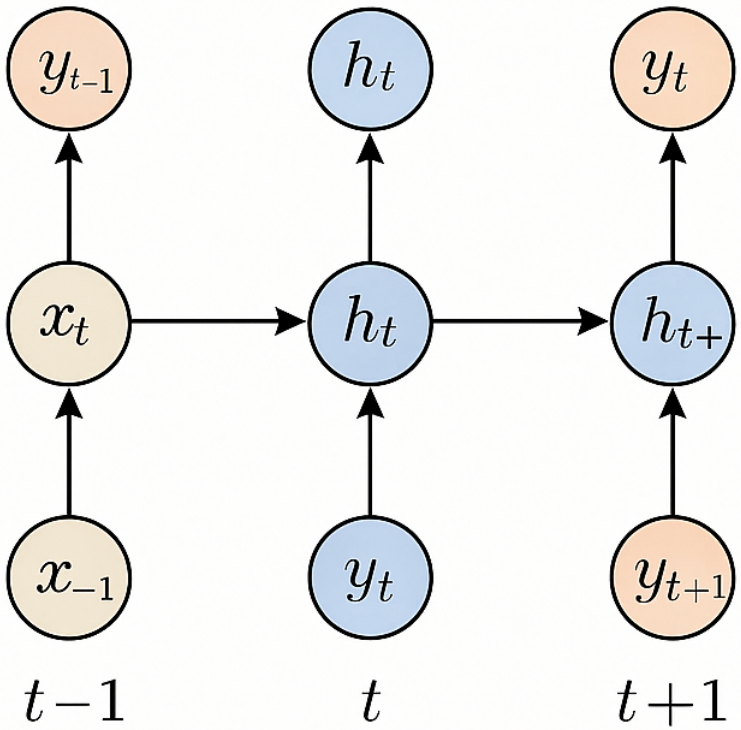
Problèmes de gradient dans les RNN

- **Gradient évanescent** : les gradients s'atténuent de façon exponentielle à mesure qu'on remonte dans le temps, rendant difficile l'apprentissage de dépendances lointaines (le signal d'erreur se dissipe).
- **Gradient explosif** : à l'inverse, les gradients peuvent exploser (devenir très grands), causant des instabilités numériques et des mises à jour de poids gigantesques.
- Ces phénomènes proviennent de la multiplication répétée de dérivées (Jacobiennes) à chaque pas : si leur valeur propre spectrale est < 1, le produit tend vers 0 (évanescence); si > 1, le gradient peut diverger.
- Solutions partielle : *gradient clipping* (limiter la valeur maximale du gradient), initialisations particulières des poids, ou utilisation d'**architectures spéciales (LSTM, GRU)** qui gérent mieux le flux du gradient.

Améliorations simples des RNN

- **RNN bidirectionnel** : on dispose deux RNN en sens inverse (un traite la séquence de gauche à droite, l'autre de droite à gauche) et on combine leurs états. On exploite ainsi le contexte passé et futur (utile quand on désire traiter la séquence entière d'un coup, ex: analyse linguistique).
- **RNN empilés ("Deep")** : on peut empiler plusieurs couches de RNN l'une au-dessus de l'autre (les sorties d'une couche deviennent les entrées de la suivante). Cela permet d'apprendre des représentations plus abstraites à chaque niveau, au prix d'un apprentissage plus complexe (plus de couches = plus difficile à entraîner).

Architecture d'un RNN simple



Un RNN standard déplié sur 3 pas de temps
(chaque cercle est un neurone)

Exemple : analyse de sentiments avec un RNN

- Tâche : classifier une phrase comme *positive* ou *négative* (sentiment).
- On peut utiliser un RNN qui lit la phrase mot par mot. À la fin (dernier mot), l'état caché h_T contient une "représentation" de la phrase.
- On ajoute une couche de sortie (par ex. sigmoïde pour proba positif) prenant h_T en entrée pour prédire la polarité.
- Si la phrase est longue, un RNN simple peut perdre l'information des premiers mots. Des variantes comme LSTM aident à mieux " mémoriser " des informations importantes sur la durée de la phrase.

Exemple : prévision de série temporelle

- Tâche : prévoir la valeur future d'une série (par ex. cours de bourse) à partir des valeurs précédentes.
- On fournit à un RNN une suite de valeurs passées $(x_{t-n}, \ldots, x_{t-1}, x_t)$, il produit une sortie $\hat{y}_{t+1}$ qui est la prédiction de la prochaine valeur.
- L'état caché du RNN agit comme un **réservoir de mémoire** intégrant les informations sur la tendance, les cycles, etc. observés dans la fenêtre de temps précédente.
- Limite : un RNN simple ne peut pas toujours capturer des tendances à très long terme ou saisonnières au-delà de la longueur de séquence fournie. Des méthodes plus sophistiquées (LSTM, ajout de variables exogènes, etc.) peuvent améliorer la précision.

Implémentation TensorFlow d'un RNN

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

# Donn es d'exemple : s quences de longueur 10 (1 feature)
# (X\_train : shape (N, 10, 1), y\_train : shape (N,))
model = keras.Sequential([
layers.SimpleRNN(50, activation='tanh'),     # RNN 50 unit s
layers.Dense(1, activation='sigmoid')       # Couche de sortie
])
model.compile(optimizer='adam', loss='binary\_crossentropy',
metrics=['accuracy'])
# Entra nement (juste illustration, X\_train/y\_train
#doivent  tre  d finis)
model.fit(X\_train, y\_train, epochs=5, batch\_size=32)
```

LSTM : RNN évolué pour dépendances longues

- Le **Long Short-Term Memory (LSTM)** est une variante de RNN introduite par Hochreiter & Schmidhuber (1997) pour pallier le problème du gradient évanescent.
- Idée : **cellule mémoire** avec des **portes** d'entrée, de sortie et d'oubli qui régulent le flux d'information. Permet de conserver des informations sur de plus longues périodes et de décider quoi oublier.
- De nombreux résultats marquants en séquence (traduction, reconnaissance de la parole) ont été obtenus grâce aux LSTM, qui sont devenus un standard avant l'apparition des Transformers.
- Un LSTM calcule comme un RNN un état caché h_t, mais maintient en plus un **état interne** c_t qui accumule les informations importantes sélectionnées.

Architecture interne d'une cellule LSTM

Architecture interne d'une cellule LSTM

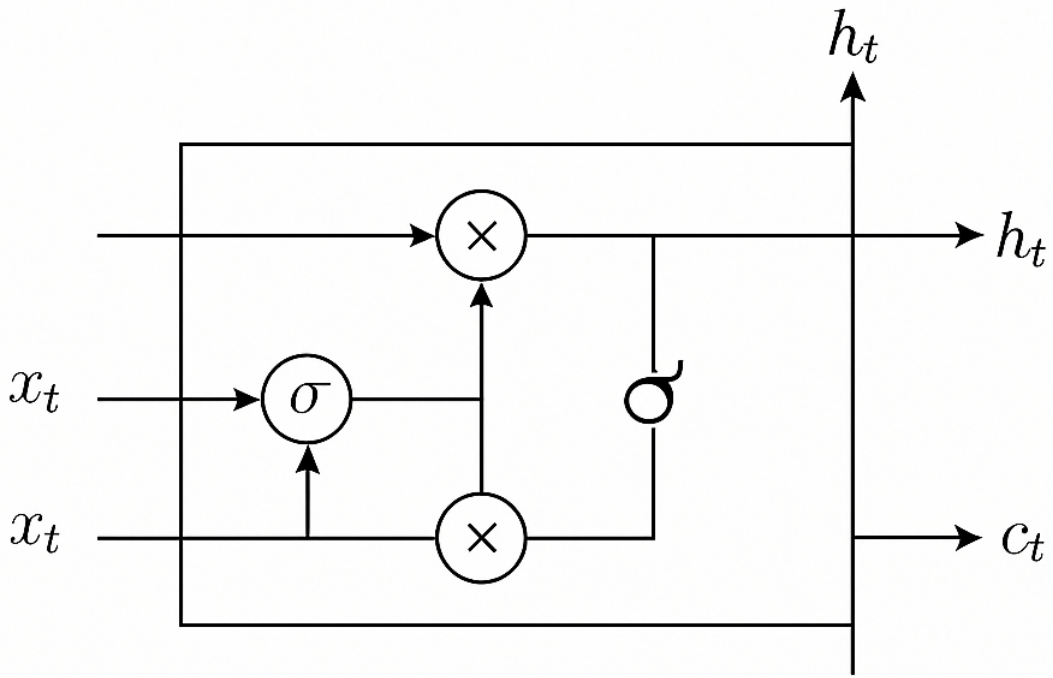


Schéma d'un bloc LSTM avec ses portes (entrée x_t, état caché h_{t-1}) et cell c_t pduisant h_t, c_t)

Formules du LSTM (1/3)

- **Porte d'oubli** f_t : décide quelle part de l'ancien contenu conserver.

$$f_t = \sigma(W_{xf}\, x_t + W_{hf}\, h_{t-1} + W_{cf}\, c_{t-1} + b_f)$$

(sigmoïde entre 0 et 1 : 0 = " tout oublier ", 1 = " tout garder ").

Formules du LSTM (2/3)

- **Porte d'entrée** i_t : décide quelle nouvelle information ajouter dans la cellule.

$$i_t = \sigma(W_{xi}\, x_t + W_{hi}\, h_{t-1} + W_{ci}\, c_{t-1} + b_i)$$

- **État candidat** $\tilde{c}_t$: nouvelle valeur candidate pour la cellule, calculée par une couche neuronale.

$$\tilde{c}_t = \tanh(W_{xc}\, x_t + W_{hc}\, h_{t-1} + b_c)$$

Formules du LSTM (3/3)

- **Mise à jour de la cellule** : combinaison de l'ancien et du nouveau contenu.

$$c_t = f_t c_{t-1} + i_t \tilde{c}_t$$

- **Porte de sortie** o_t : décide quelle part de la cellule affecte l'état caché (et la sortie).

$$o_t = \sigma(W_{xo}\, x_t + W_{ho}\, h_{t-1} + W_{co}\, c_t + b_o)$$

- **État caché mis à jour** : c'est la version " filtrée " de l'état de cellule.

$$h_t = o_t \tanh(c_t)$$

Interprétation des portes du LSTM

- **Porte d'oubli** f_t: contrôle combien de l'ancien *cell state* c_{t-1} on conserve. Ex: $f_t \approx 0$ efface la mémoire précédente (reset si contexte change).
- **Porte d'entrée** i_t: décide quelles nouvelles informations intégrer dans c_t depuis $\tilde{c}_t$. Ex: $i_t \approx 1$ et $\tilde{c}_t$ grand $\Rightarrow$ on écrit fortement une nouvelle info en mémoire.
- **Porte de sortie** o_t: contrôle la proportion de l'état interne c_t qui influence l'état caché h_t (et potentiellement la sortie). Ex: $o_t \approx 0 \Rightarrow$ la cellule garde l'info mais ne la " dit pas " encore.
- Grâce à ces portes, un LSTM peut maintenir une information pertinente dans c_t sur de nombreux pas de temps, et la restituer plus tard, résolvant mieux les dépendances longues.

Exemple : mémoire à long terme d'un LSTM

Considérons la phrase : " Je suis né en **France**... Je parle couramment le **[???]**".

- Pour deviner le mot manquant (" français "), il faut se souvenir plusieurs propositions plus tôt que le pays mentionné est la France.
- Un RNN simple risque d'**oublier** cette information à cause du gradient évanescent s'il y a beaucoup de mots entre les deux.
- Un LSTM peut **retenir** naturellement l'information " France " dans son état de cellule sur toute la durée de la phrase. Ainsi, quand vient le moment de prédire la langue parlée, la porte de sortie laissera passer cette information, et le LSTM proposera " français ".

LSTM vs GRU

- Le **GRU** (*Gated Recurrent Unit*, 2014) est une variante simplifiée du LSTM. Il fusionne la porte d'entrée et la porte d'oubli en une seule **porte de mise à jour**, et combine l'état de cellule et l'état caché en un seul vecteur h_t.
- Avantages du GRU : moins de paramètres qu'un LSTM (plus rapide à entraîner), et performances souvent comparables à LSTM sur de nombreuses applications.
- Dans la pratique, le choix LSTM vs GRU peut se faire par validation : certains jeux de données/tâches préfèrent l'un ou l'autre. GRU peut suffire et être plus efficace, mais LSTM est plus expressif (grâce à son état de cellule distinct).

Implémentation TensorFlow d'un LSTM

```
# Exemple de mod le LSTM pour classification binaire de s quences
model = keras.Sequential([
layers.LSTM(50, return\_sequences=False),   # LSTM 50 unit s
layers.Dense(1, activation='sigmoid')
])
model.compile(optimizer='adam', loss='binary\_crossentropy',
metrics=['accuracy'])
# Entra nement (X\_train : shape (N, timesteps, features))
model.fit(X\_train, Y\_train, epochs=5, batch\_size=32)
```

Gated Recurrent Unit (GRU)

- Formules du **GRU** :

$$z_t = \sigma(W_{xz}x_t + W_{hz}h_{t-1} + b_z)$$

(porte de mise à jour)

$$r_t = \sigma(W_{xr}x_t + W_{hr}h_{t-1} + b_r)$$

(porte de remise à zéro)

$$\tilde{h}_t = \tanh(W_{xh}x_t + W_{hh}(r_t * h_{t-1}) + b_h)$$

(état candidat)

$$h_t = z_t * h_{t-1} + (1 - z_t) * \tilde{h}_t$$

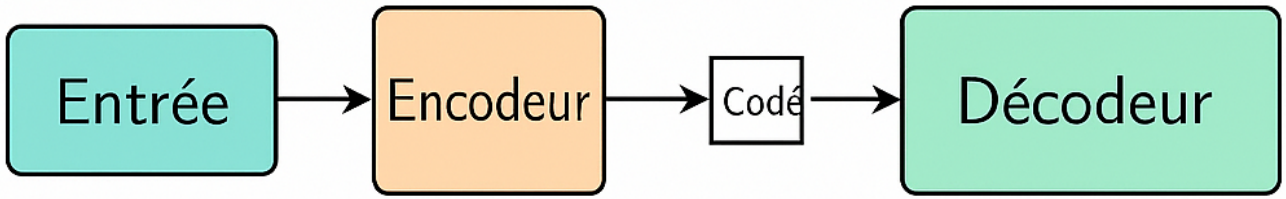
(état final)

- Comme le LSTM, le GRU peut transporter h_{t-1} directement à h_t via $z_t \approx 1$ (si l'info doit persister), ou injecter un nouvel état $\tilde{h}_t$ si z_t est petit.
- Il n'y a pas de sortie o_t explicite ni d'état de cellule séparé : c'est une architecture plus simple.

Autoencodeur : principe

- Un **autoencodeur** est un réseau neuronal entraîné à **reconstruire sa propre entrée** en sortie.
- Il se compose de deux parties : un **encodeur** qui compresse l'entrée x en un vecteur latent z (de dimension plus petite), et un **décodeur** qui reconstruit $\hat{x}$ à partir de z.
- En apprenant à reproduire les données d'entrée en sortie, l'autoencodeur **apprend une représentation latente** utile des données (il doit capturer l'information essentielle).
- Utilisations : *réduction de dimensionnalité* (comme une PCA non-linéaire), *débruitage* (en apprenant à ignorer le bruit), *détection d'anomalies* (un autoencodeur formé sur des données normales reconstruit mal une anomalie), ou encore *génération* de données nouvelles (avec des variantes spéciales).

Architecture d'un autoencodeur



Architecture d'un autoencodeur standard : l'encodeur compresse l'entrée en un code z de plus petite dimension, le décodeur reconstruit l'entrée à partir de z.

Formulation mathématique

- **Encodeur** : fonction f_θ qui projette l'entrée x vers un espace latent de dimension plus faible : $z = f_\theta(x)$.
- **Décodeur** : fonction g_ϕ qui reconstruit $\hat{x} = g_\phi(z)$ dans l'espace original à partir du code latent.
- **Entraînement** : on ajuste les paramètres θ, ϕ pour que $\hat{x} \approx x$ le plus fidèlement possible. On minimise une fonction de perte de reconstruction, par ex:

$$\mathcal{L}(x, \hat{x}) = \|x - \hat{x}\|^2$$

  (somme des carrés) ou une entropie croisée si x est binaire.
- Si la dimension de z est plus petite que x, l'autoencodeur est **sous-complet** et doit apprendre à compresser l'information (sinon la tâche serait triviale en copiant l'entrée).

Variantes d'autoencodeurs

- **Autoencodeur parcimonieux** : la dimension du code peut être égale ou supérieure à l'entrée, mais on ajoute une contrainte de **parcimonie** (beaucoup de coefficients de z proches de 0) via un terme de régularisation. On force le réseau à coder l'info de manière " économique ".
- **Autoencodeur convolutionnel** : pour les images, on utilise des couches de convolution dans l'encodeur et de déconvolution/upsampling dans le décodeur au lieu de couches fully-connected, ce qui préserve la structure spatiale et permet de meilleurs résultats en vision.

Autoencodeur variationnel (VAE)

- Un **autoencodeur variationnel** impose une **distribution probabiliste** sur la variable latente z (typiquement une gaussienne standard).
- L'encodeur produit des paramètres statistiques (moyenne $\mu(x)$ et variance $\sigma^2(x)$) d'une distribution $q_\phi(z|x)$ au lieu d'un vecteur fixe. On **échantillonne** alors z de cette distribution.
- L'entraînement minimise la perte : $\mathbb{E}_{z\sim q_\phi(z|x)}[\|x - g_\theta(z)\|^2] + D_{\mathrm{KL}}(q_\phi(z|x) \parallel p(z))$. Le second terme pousse $q_\phi(z|x)$ à rester proche d'une loi prior $p(z)$ (ex: $\mathcal{N}(0, I)$).
- **Intérêt** : on peut générer de nouvelles données en échantillonnant $z \sim p(z)$ et en calculant $\hat{x} = g_\theta(z)$. Le VAE est une approche générative où l'espace latent est bien organisé.

Applications des autoencodeurs

- **Réduction de dimensionnalité** : compresser des données (par ex. images) en un vecteur de plus petite taille z. Utile pour la visualisation ou l'initialisation de modèles.
- **Dénbruitage (*denoising*)** : en entraînant un autoencodeur à reconstruire l'image d'origine à partir d'une version bruitée, on obtient un **filtre à bruit appris**.
- **Détection d'anomalies** : un autoencodeur entraîné sur des données normales aura du mal à reconstruire une donnée anormale $\Rightarrow$ l'erreur de reconstruction sert d'indicateur d'anomalie (forte erreur = donnée suspecte).
- **Génération de données** : en exploitant des variantes génératives (VAE) ou en combinant avec des techniques adversariales (VAE-GAN), on peut générer de nouvelles images, etc.

Autoencodeur vs Analyse en Composantes Principales (ACP)

- Un autoencodeur **linéaire** (aucune activation non-linéaire) apprendra **exactement** le même sous-espace latent que l'ACP (PCA) : ses poids convergeront vers les vecteurs propres principaux de la covariance des données.
- En revanche, avec des activations non-linéaires (ex: ReLU, sigmoïde), l'autoencodeur peut apprendre des codages non-linéaires bien plus puissants que l'ACP, capturant des structures complexes des données.
- L'autoencodeur n'impose pas que le code soit une combinaison linéaire des entrées : il peut ainsi modéliser des variations qui ne sont pas dans un sous-espace linéaire.
- En pratique, les autoencodeurs profonds surpassent l'ACP pour la compression si les données ont des relations non-linéaires. Cependant, ils sont plus longs à entraîner et moins transparents que l'ACP analytique.

Exemple : autoencodeur débruiteur

- On peut entraîner un autoencodeur **débruiteur** en fournissant une version bruitée de x à l'encodeur tout en utilisant x original comme cible en sortie.
- Ex: on prend des images et on leur ajoute un bruit aléatoire ("sal et poivre", bruit gaussien, etc.) pour obtenir $\tilde{x}$. On entraîne l'autoencodeur sur $\tilde{x} \rightarrow x$.
- Une fois entraîné, le réseau est capable de prendre une image bruitée en entrée et de sortir une image **nettoyée du bruit**.
- Ce type d'autoencodeur apprend une représentation z qui ignore les détails non pertinents (le bruit) et capture l'essentiel de l'image.

Exemple : autoencodeur pour détection d'anomalies

- Prenons des données de capteurs sur une machine en fonctionnement normal. On entraîne un autoencodeur sur ces données **sans anomalie** pour reconstruire les signaux.
- Si un dysfonctionnement survient (anomalie), les données s'écartent du comportement normal appris. L'autoencodeur reconstruit alors très mal ce type de donnée inédit.
- En surveillant l'**erreur de reconstruction** $\|x - \hat{x}\|$, on peut détecter les anomalies : si l'erreur dépasse un certain seuil, on alerte qu'un comportement anormal est présent.
- Cette approche est utilisée en détection de fraudes (cartes de crédit), surveillance d'équipements (capteurs IoT), etc., où les anomalies sont rares et difficiles à modéliser explicitement.

Implémentation TensorFlow d'un autoencodeur

```
from tensorflow import keras
from tensorflow.keras import layers

# Autoencodeur simple pour images 28x28 (entr es aplaties en 784)
input_img = keras.Input(shape=(784,))
encoded = layers.Dense(64, activation='relu')(input_img)
decoded = layers.Dense(784, activation='sigmoid')(encoded)
autoencoder = keras.Model(input_img, decoded)
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')

# Entra nement (X_train : images 28x28 aplaties, valeurs dans [0,1])
autoencoder.fit(X_train, X_train,
epochs=20, batch_size=256,
validation_data=(X_test, X_test))
```

BERT : introduction

- **BERT** (Devlin et al., 2018) est un modèle de langage pré-entraîné qui a défini un nouveau standard en NLP. Acronyme de *Bidirectional Encoder Representations from Transformers*.
- BERT est basé sur l'architecture **Transformer** (Vaswani et al., 2017) mais n'utilise que la partie **encodeur** du Transformer. Il est dit bidirectionnel car il peut exploiter le contexte à gauche et à droite d'un token.
- Il est pré-entraîné de façon auto-supervisée sur un immense corpus textuel (Wikipedia + livres) avec deux tâches :
  - **Modélisation de langage masquée** (MLM)
  - **Prédiction de la phrase suivante** (NSP)
- Après pré-entrainement, BERT peut être **affiné (fine-tuned)** sur des tâches spécifiques (classification, QA, NER, ...) en ajoutant une couche de sortie appropriée.

Tâche 1 : Masked Language Modeling (MLM)

- On cache aléatoirement certains mots de la phrase d'entrée (15% des tokens remplacés par [MASK]) et BERT doit prédire les tokens manquants.
- Ex: " Le `[MASK]` est doux et blanc" $\rightarrow$ BERT doit deviner " chat ".
- Cette tâche force le modèle à apprendre des représentations contextuelles **bidirectionnelles** : le prédicteur du mot manquant doit regarder à la fois à gauche et à droite du masque.
- Contrairement à un modèle de langage classique qui prédit le prochain mot uniquement d'après le passé, BERT intègre le contexte complet de la phrase.

Tâche 2 : Next Sentence Prediction (NSP)

- BERT est entraîné à prédire si une phrase B est la suite logique d'une phrase A. On lui présente des paires de phrases (A, B).
- La moitié du temps B est effectivement la phrase suivante de A dans le corpus, l'autre moitié B est choisie au hasard (phrase sans lien).
- BERT reçoit en entrée la concaténation `[CLS]` A `[SEP]` B `[SEP]`, et produit en sortie une probabilité " Séquence suivante ? Vrai/Faux " à partir du token [CLS].
- Cette tâche incite BERT à comprendre la relation sémantique entre deux phrases (très utile pour des tâches comme les questions-réponses, où il faut savoir si une phrase répond à une autre).

Self-attention : principe

- Le **Transformer** se base sur le mécanisme d'**attention interne** (*self-attention*) pour modéliser les relations entre les mots d'une phrase sans utiliser de récurrence.
- Chaque mot (token) va interagir avec les autres au sein d'une couche d'attention : **requêtes** (Q), **clés** (K) et **valeurs** (V) sont calculés pour chaque token via des transformations linéaires.
- Pour un token donné, on calcule des poids d'attention avec tous les autres tokens en comparant sa requête avec les clés de chaque mot. Ces poids (normalisés par softmax) servent à combiner une somme pondérée des valeurs V de tous les tokens.
- Résultat : chaque token met à jour sa représentation en y incorporant une part de l'information des autres tokens, proportionnellement à leur pertinence contextuelle.

Self-attention : formulation

Le calcul d'une attention (simple) sur une séquence est :

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

où :

- Q, K, V sont les matrices des requêtes, clés et valeurs pour tous les tokens (de dimension $n \times d_k$ si n tokens et vecteurs de dim d_k).
- Le produit QK^T donne une matrice $n \times n$ des **scores d'affinité** entre chaque paire de tokens.
- La division par $\sqrt{d_k}$ **normalise** les scores (évite des valeurs extrêmes lorsque d_k est grand) avant la softmax.
- La softmax transforme chaque ligne en distribution de probabilité sommant à 1 : ce sont les **poids d'attention** que chaque token accorde aux autres.
- En multipliant par V, chaque token accumule les valeurs des autres tokens pondérées par ces poids.

Multi-head attention

- Au lieu d'effectuer une seule attention sur des vecteurs de grande dimension, le Transformer emploie h **têtes d'attention** parallèles.
- Chaque tête utilise des matrices W_i^Q, W_i^K, W_i^V différentes pour projeter les embeddings en requêtes, clés, valeurs de dimension réduite ($d_k = d_{\text{model}}/h$).
- On calcule l'attention pour chaque tête indépendamment, puis on **concatène** les résultats de dimension d_k en un vecteur de dimension d_{model}.
- Une matrice W^O finale (par tête ou partagée) recompose alors ce vecteur. L'idée est que chaque tête peut apprendre à capturer différents types de relations (synonymie, coréférence, dépendance syntaxique, etc.) simultanément.

Encodage positionnel

- Un Transformer n'ayant pas de récurrence, il ne connaît pas l'ordre des mots par défaut. Il faut ajouter une information de **position** des tokens.
- BERT et GPT ajoutent aux embeddings de mot un **embedding de position** (un vecteur unique associé à chaque position 1,2,...,N dans la phrase).
- Dans les implémentations originales, ces vecteurs de position sont fixes et déterministes (fonctions sinusoïdales de différentes fréquences), ou parfois appris comme des paramètres supplémentaires.
- Ainsi, la représentation finale d'un mot = embedding lexical + embedding de position (+ embedding de segment dans BERT pour distinguer phrase A vs B).
- D'autres variantes de Transformers (Transformer-XL, etc.) utilisent des **encodages positionnels relatifs** plus à même de gérer des séquences plus longues de façon efficace.

Architecture de BERT (Transformer encodeur)

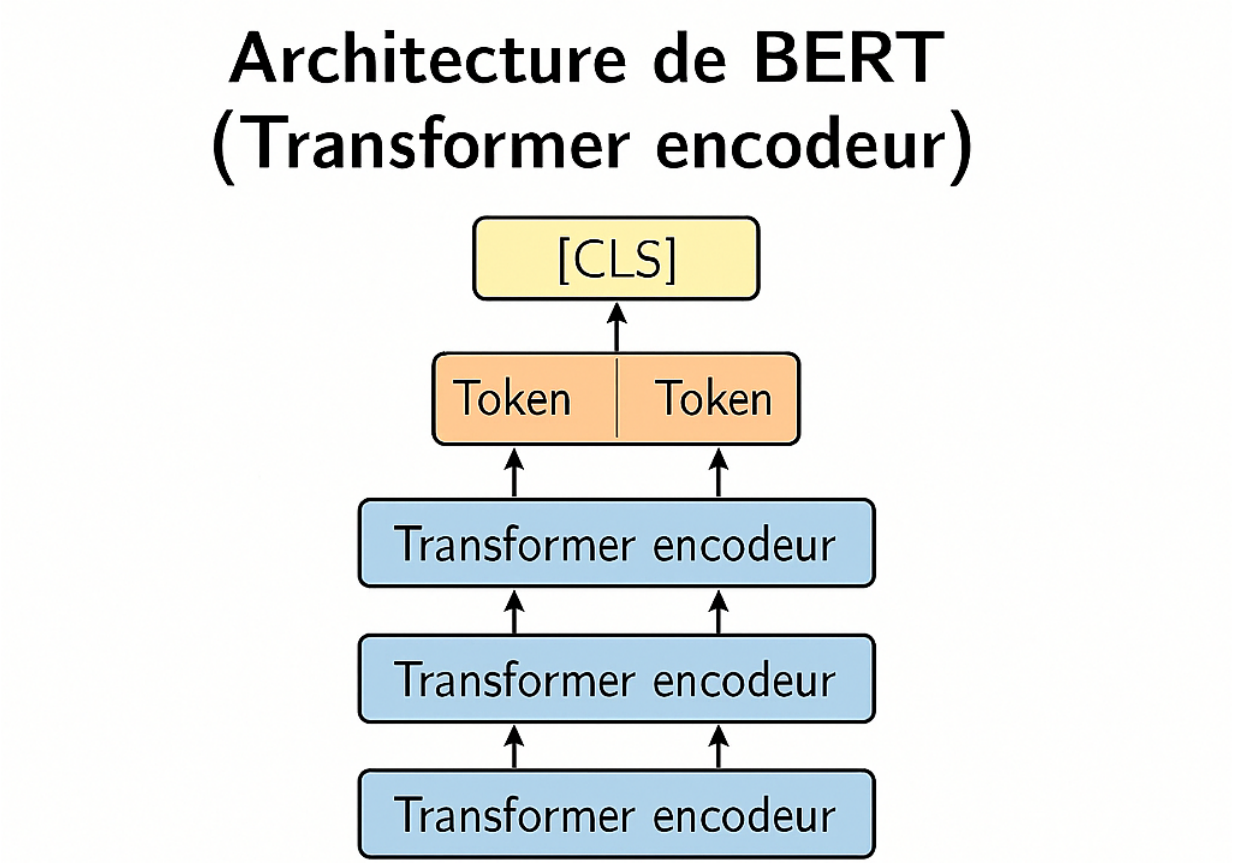


Schéma de BERT : des embeddings de tokens (incluant infos de position et segment) traversent L couches Transformer encodeur.

On utilise l'état [CLS] ou les états des tokens de sortie pour les tâches cibles.

Fine-tuning de BERT sur une tâche

- BERT fournit des **représentations universelles** du langage. Pour une tâche particulière (ex: classification de texte, QA), on ajoute une couche de sortie et on **ré-entraîne** le modèle sur cette tâche (avec un jeu de données labellisé).
- Ex: Classification de sentiments - on prend l'état du token [CLS] à la sortie de BERT et on ajoute une couche Dense sigmoïde pour prédire Positif/Négatif. On entraîne quelques epochs sur un corpus de textes annotés.
- BERT " adapte " alors ses paramètres pour la tâche, tout en gardant la majeure partie de ses connaissances linguistiques acquises.
- Fine-tuner BERT nécessite beaucoup moins de données et de temps que d'entraîner un modèle from scratch. C'est pourquoi les modèles pré-entrainés sont très utiles (parfois quelques centaines d'exemples suffisent pour obtenir de bonnes performances).

Variantes et extensions de BERT

- **RoBERTa** (2019) : variante de BERT par Facebook. Entraîné plus longtemps, sans tâche NSP, sur davantage de données et avec des batchs plus gros. RoBERTa obtient de meilleures perf que BERT original sur plusieurs tâches.
- **DistilBERT** (2019) : version " allégée " de BERT obtenue par **distillation** des connaissances de BERT dans un modèle plus petit (6 couches au lieu de 12). Presque aussi bon que BERT pour moitié moins de poids, plus rapide à exécuter.
- **BERT multilingue** : Google a entraîné une version de BERT sur plus de 100 langues (`bert-base-multilingual`). Cela permet de l'utiliser pour des langues hors anglais (ex: Français) en fine-tuning direct.
- **Spécialisations** : une multitude de variantes adaptent BERT à des domaines ou tâches particulières : BioBERT (biomédical), SciBERT (scientifique), CamemBERT (BERT français), etc., en changeant le corpus d'entrainement.

Limitations de BERT

- **Pas de génération libre** : BERT est un encodeur, on l'utilise pour des tâches de compréhension ou classification. Pour générer du texte, il n'est pas conçu (on préfère GPT).
- **Longueur maximum** : BERT traite des séquences jusqu'à 512 tokens (pour BERT base). Les textes plus longs doivent être segmentés ou traités par des variantes (Longformer, etc.).
- **Tâche NSP discutable** : RoBERTa a montré qu'enlever la tâche Next Sentence Prediction et augmenter la taille de batch peut améliorer les résultats. NSP peut donc ne pas être crucial.
- **Coût et empreinte** : 110M de paramètres (BERT base), 340M (large) - à entraîner c'est coûteux (plusieurs jours sur TPU). Même pour l'inférence, charger 1 ou 3 Go de modèle en mémoire est lourd, ce qui motive des optimisations (distillation, quantification).

Exemple : BERT pour les questions-réponses

On veut utiliser BERT pour extraire la réponse à une question dans un paragraphe :

- Input : **Contexte** (texte) + **Question**. Par ex : Contexte : ” La tour Eiffel mesure 324 m de haut ...”, Question : ” Quelle est la hauteur de la tour Eiffel ?”
- On prépare l’input à la forme [CLS] Contexte [SEP] Question [SEP].
- On ajoute au sommet de BERT deux couches linéaires qui prédisent la position de début et de fin de la réponse dans le texte (indices de tokens).
- En fine-tunant BERT sur un corpus QA (ex: SQuAD), il apprend à repérer le segment du texte répondant à la question. Ici, il identifiera ” 324 m ” comme réponse.
- BERT a obtenu des scores à hauteur humaine sur SQuAD en 2018, marquant une percée significative.

Utilisation de BERT avec HuggingFace

```
from transformers import BertTokenizer, TFBertModel

# Charger le tokenizer et modele BERT pr -entrain  (base, non-case)
tokenizer = BertTokenizer.from\_pretrained('bert-base-uncased')
bert\_model = TFBertModel.from\_pretrained('bert-base-uncased')

text = "Le deep learning est fascinant."
inputs = tokenizer(text, return\_tensors='tf')
outputs = bert\_model(inputs)
embeddings = outputs.last\_hidden\_state
# Tensor shape (1, sequence_length, 768)
```

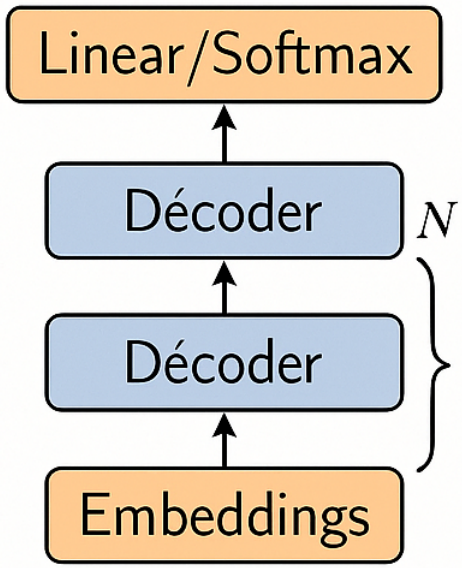
GPT : introduction

- **GPT** (Radford et al., 2018+) est une série de modèles de langage de type Transformer **décodeur** (*auto-régressif*), spécialisés dans la **génération de texte**.
- GPT génère du texte en prédisant le mot suivant d'après le contexte gauche uniquement (comme un modèle de langue classique). Il ne voit pas le futur, uniquement le passé immédiat.
- **GPT-2** (2019) a montré qu'en augmentant drastiquement la taille du modèle et la quantité de données d'entraînement, le modèle pouvait produire des textes très cohérents et difficilement distinguables de textes humains.
- **GPT-3** (2020, 175 milliards de paramètres) a confirmé qu'un modèle génératif de très grande taille pouvait résoudre de nombreuses tâches par simple **"prompting"** (sans même nécessiter de fine-tuning spécifique).

Architecture du GPT (décodeur Transformer)

- GPT utilise uniquement la partie **décodeur** du Transformer. Cela signifie que chaque couche comporte une **attention masquée** : un token ne peut attentivement regarder que les tokens précédents (et lui-même), pas les suivants.
- Le modèle prend une invite (*prompt*) initiale et génère la suite mot par mot.
- Le décodeur du Transformer intègre un **mécanisme auto-régressif** : pour générer le token $t + 1$, on utilise déjà les tokens 1 à t (et leurs représentations internes).
- En sortie du dernier bloc, une couche linéaire suivie d'une softmax donne une distribution de probabilité sur le vocabulaire pour le prochain token. On peut choisir le token le plus probable (greedy) ou **échantillonner** pour obtenir un texte plus varié.

Architecture d'un GPT (schéma)



Architecture générale d'un modèle GPT (Transformeur décodeur) : embeddings + N blocs décodeurs (avec attention masquée) + couche de sortie linéaire/softmax.

Évolution des modèles GPT

- **GPT-1** (2018) : 117M de paramètres. Démontre le principe du pré-entrainement + fine-tuning sur quelques tâches NLP, avec succès modeste.
- **GPT-2** (2019) : jusqu'à 1.5 milliard de paramètres. Génère des textes très cohérents, parfois déroutants. OpenAI a initialement restreint sa diffusion par crainte d'abus (génération de désinformation).
- **GPT-3** (2020) : 175 milliards de paramètres. Capacités impressionnantes : comprend des instructions complexes et résout à peu près toute tâche textuelle via **quelques exemples en prompt** (*few-shot learning*), sans nécessiter de fine-tuning.
- **GPT-3.5 & GPT-4** (2022-2023) : versions affinnées et **multimodales**. GPT-3.5 est aligné pour la conversation (ChatGPT). GPT-4 (non public complet) atteindrait 1 billion de paramètres et accepte entrées texte+image.

ChatGPT et alignement des LLM

- **ChatGPT** (2022) est une application conversationnelle construite sur GPT-3.5. Il a été affiné via l'**apprentissage par renforcement avec feedback humain** (RLHF).
- Principe du RLHF : des humains annotent la qualité des réponses de GPT (préférences), puis un algorithme de renforcement ajuste le modèle pour maximiser ces jugements. Ceci oriente le modèle à " vouloir " répondre de manière utile et non toxique.
- Résultat : ChatGPT suit beaucoup mieux les instructions de l'utilisateur que GPT-3 brut et génère des réponses plus pertinentes et d'un ton adapté. C'est un pas vers des IA linguistiques plus **sûres et alignées** avec les attentes humaines.
- Les techniques d'alignement (dont RLHF) sont désormais courantes pour adapter les grands modèles de langage à une utilisation pratique responsable.

Limites des modèles GPT

- **Coût calculatoire** : GPT-3 a nécessité des milliers de GPU et des semaines d'entraînement. Utiliser GPT-3 ou GPT-4 en production implique des coûts importants (infrastructure cloud, etc.).
- **Connaissances figées** : un modèle GPT pré-entrainé ne sait que ce qu'il y a dans son corpus (arrêté à une certaine date). Il peut fournir des infos périmées si utilisé plus tard sans mise à jour.
- **Hallucinations** : les GPT peuvent inventer des détails faux mais plausibles (halluciner), car ils génèrent par association linguistique, pas par référence à une base de connaissances fiable.
- **Biais et toxicité** : formés sur Internet, ils reflètent aussi les biais ou propos toxiques des données. Malgré les filtres, ils peuvent produire des sorties biaisées, offensantes ou non éthiques dans certains contextes.
- **Contexte limité** : GPT-3 a une fenêtre de contexte de 2048 tokens (GPT-4 jusqu'à 32k). Ils ne peuvent pas considérer un texte trop long en une seule fois, ce qui peut limiter leur capacité à raisonner sur de longs documents.

Utilisation des GPT par *prompting*

- Plutôt que de fine-tuner un GPT sur chaque tâche, on peut **décrire la tâche en langage naturel** et laisser le modèle résoudre le problème. C'est le **"zero-shot/few-shot learning"** des LLM.
- Ex (zero-shot) : demander directement " Traduis en anglais : Bonjour tout le monde." $\rightarrow$ le modèle répond " Hello everyone." sans avoir été spécifiquement entraîné pour la traduction.
- En fournissant un ou deux exemples dans l'invite (few-shot), on améliore la fiabilité : on montre le format de la réponse attendue, le modèle s'aligne.
- L'art du **prompt design** est devenu important pour tirer le meilleur des GPT : choix des instructions, des exemples, etc. pour orienter le modèle à donner la réponse voulue.

Applications des modèles GPT

- **Génération de texte créatif** : à partir d'une invite, GPT peut écrire des histoires, des articles, des poèmes, des résumés, etc.
- **Aide à la programmation** : complétion de code (ex: Github Copilot utilise GPT-3), explication de code, génération de fonctions à partir de descriptions.
- **Chatbots et assistants virtuels** : modèles de la famille GPT (ex: ChatGPT) peuvent converser, répondre à des questions en langage naturel, donner des conseils personnalisés.
- **Éducation et recherche** : répondre à des questions de connaissances générales, expliquer des concepts (avec prudence quant à l'exactitude), aider à la création de quiz ou d'exercices.
- **Outils de rédaction** : GPT peut aider à rédiger des emails, des rapports, corriger la grammaire, reformuler des phrases, ce qui accroît la productivité dans la communication écrite.

Exemple : génération d'histoire par GPT-2

Prompt (début d'histoire) : "Il était une fois, dans un petit village, un vieux forgeron qui..."

Texte généré par GPT-2 :
"...avait passé sa vie à façonner le fer et l'acier. Chaque matin, il se levait avant l'aube, allumait le feu de sa forge et se mettait au travail. Les étincelles jaillissaient tandis qu'il martelait le métal incandescent, et bientôt toute la vallée retentissait du rythme de son labeur. Les villageois disaient que ses créations avaient une âme, car elles étaient forgées avec passion et dévouement. Un jour, un étranger arriva..."

(Exemple de texte généré : cohérent grammaticalement, respectant le style, mais la suite de l'histoire serait à inventer par la suite.)

Génération de texte avec GPT-2

```
from transformers import GPT2Tokenizer, TFGPT2LMHeadModel

tokenizer = GPT2Tokenizer.from\_pretrained('gpt2')
gpt2\_model = TFGPT2LMHeadModel.from\_pretrained('gpt2')

prompt = "Il etait une fois"
input\_ids = tokenizer.encode(prompt, return\_tensors='tf')
# Generer la suite de texte a partir du prompt
output\_ids = gpt2\_model.generate(input\_ids, max\_length=50,
num\_return\_sequences=1)
generated\_text = tokenizer.decode(output\_ids[0], skip\_special\_tokens=T
print(generated\_text)
```

Transformers vs RNN/LSTM

- **Parallélisation** : un RNN traite les tokens séquentiellement (pas de parallèle sur la dimension temps) $\Rightarrow$ difficile à déployer sur GPU pour longues séquences. Un Transformer traite tous les tokens en parallèle par self-attention (complexité $O(n^2)$ en temps, mais très parallélisable).
- **Longue portée** : les RNN/LSTM peinent à conserver les dépendances lointaines (même LSTM a une capacité limitée par c_t et possible oubli). Les Transformers relient directement chaque token à un autre via l'attention, ce qui facilite les dépendances " à longue distance ".
- **Données séquencielles continues** : Les RNN peuvent théoriquement traiter des flux infiniment longs (ex: streaming) en continuant la récurrence. Les Transformers ont une taille de contexte fixe (512, 1024 tokens...). Au-delà, on doit utiliser des astuces (fenêtre glissante, mémoires externes).
- **NLP moderne** : Les Transformers pré-entrainés (BERT, GPT) ont pratiquement remplacé les RNN/LSTM dans la plupart des applications NLP car ils offrent de bien meilleures performances d'ensemble. Néanmoins, les RNN restent utilisés dans certains contextes spécifiques ou pour des données très longues où l'attention totale est trop coûteuse.

BERT vs GPT

- **BERT** : modèle **d'encodeur** bidirectionnel. Tâche de pré-entrainement : deviner des mots masqués + lien entre phrases. BERT excelle dans les **tâches de compréhension** (analyse de sentiments, QA extractif, classification...) où comprendre le contexte global est nécessaire.
- **GPT** : modèle **décodeur** auto-régressif (auto-régressif). Tâche de pré-entrainement : modélisation de langage (prédire le token suivant). GPT excelle dans les **tâches de génération** (rédaction libre, dialogues, complétion de texte) grâce à son mécanisme gauche-droite.
- **Complémentarité** : BERT fournit de bonnes représentations pour extraire du sens, GPT fournit une suite cohérente pour générer du contenu. Les deux approches peuvent se compléter (ex: utiliser BERT pour comprendre une question et GPT pour formuler une réponse).
- En termes de taille de modèle et données, GPT-3/4 sont bien plus grands et coûteux que BERT base/large. Ils sont conçus pour modéliser des connaissances " générales " et faire de l'inférence dans le langage, là où BERT est initialement conçu pour être fine-tuné sur une tâche précise.

Chronologie (RNN $\rightarrow$ Transformer $\rightarrow$ LLM)

- **1997** : Invention du LSTM, résout le problème de la mémoire longue durée dans les RNN.
- **2014** : Succès des seq2seq à base de LSTM en traduction (Sutskever et al.), introduction de l'attention (Bahdanau et al.) dans les RNN encodeur-décodeur.
- **2017** : Publication de " Attention is All You Need " (Vaswani et al.), architecture Transformer sans RNN, surpassant LSTM en traduction tout en étant plus parallélisable.
- **2018** : BERT (premier Transformer bidirectionnel généraliste pré-entrainé) améliore spectaculairement l'état de l'art sur de nombreuses tâches NLP.
- **2019** : GPT-2 montre la puissance générative des Transformers décodeurs. Naissance de nombreuses variantes de BERT.
- **2020** : GPT-3 démontre le " few-shot learning " avec 175B param. et préfigure l'échelle à atteindre pour des IA génératives de qualité.
- **2022-2023** : Démocratisation des **LLM** (Large Language Models) : ChatGPT atteint des centaines de millions d'utilisateurs, GPT-4 intègre l'image, des efforts open-source (BLOOM, LLaMA) voient le jour pour réduire la dépendance aux grands laboratoires.

Coût et ressources d'entraînement

- **BERT base** (110M param) a été entraîné sur 3.3 milliards de mots (Books + Wikipedia). Environ 4 jours sur 16 TPU v3. Coût estimé : plusieurs milliers de dollars.
- **GPT-3** (175B param) a été entraîné sur 500 milliards de tokens (mélange de textes du web, livres). Environ 1024 GPU sur 34 jours. Coût estimé : 4.6 millions USD.
- Ces coûts faramineux expliquent pourquoi ces modèles sont le fait de quelques organisations (Google, OpenAI, Microsoft...).
- Heureusement, la **réutilisation** de ces modèles pré-entraînés (fine-tuning ou API) permet à la communauté au sens large de bénéficier de leurs capacités sans avoir à dépenser ces ressources.
- Tendance actuelle : mutualiser les efforts (modèles open-source), et optimiser (modèles compressés, techniques de pré-entrainement plus efficaces, etc.) pour réduire les barrières d'entrée.

Conclusion

- Les réseaux **RNN** et **LSTM** ont marqué une étape importante pour traiter les séquences, en particulier en traitement de la langue naturelle et séries temporelles.
- Les **autoencodeurs** ont introduit des approches non supervisées pour apprendre des représentations utiles et pour la génération de données.
- L'architecture **Transformer** (BERT, GPT) a révolutionné le domaine du NLP en offrant une meilleure modélisation du contexte et une efficacité accrue pour l'entraînement sur de très grands corpus.
- Ces modèles pré-entrainés ont ouvert la voie à des progrès rapides dans de nombreuses applications d'IA, tout en soulevant de nouveaux défis (taille des modèles, biais, etc.).

Perspectives

- **Modèles multimodaux** : combiner texte, vision, audio (ex: GPT-4 capable de traiter image+texte) pour des IA comprenant plusieurs modalités simultanément.
- **Efficacité** : réduction de la taille des modèles via distillation, pruning, quantization. Exploration de nouvelles architectures (Transformers linéaires, recurrent memory networks) pour gérer de longs contextes avec moins de coût.
- **Interprétabilité** : ouvrir la " boîte noire " des LLM, comprendre ce qu'ils apprennent (études sur les têtes d'attention, neurons, etc.) pour accroître la confiance et diagnostiquer les erreurs.
- **Régulation et déontologie** : encadrer l'usage des modèles de langage (désinformation, plagiat, biais) par des directives, des filtres ou des réglementations, tout en profitant de leurs bénéfices.
- **Nouvelles frontières** : IA plus " générales " (capables de planification, apprentissage continu), intégration avec des bases de connaissances externes, ou interactions plus riches avec l'humain (apprentissage par feedback continu).

Fin

Questions? Comments?