

Large language models before transformers

RNN, GRU, LSTM, SeqtoSeq

Mohamed Hédi Riahi

February 24, 2025

Réseaux de Neurones Récurrents (RNN)

- **Problématique** : Les réseaux feed-forward ne peuvent pas gérer des données séquentielles de longueur variable.
- **Solution** : Les RNN introduisent une mémoire interne pour capturer les dépendances temporelles.
- **Applications** :
 - Traitement du langage naturel (NLP).
 - Reconnaissance vocale.
 - Prédiction de séries temporelles.

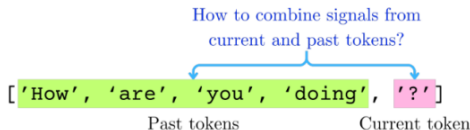


Figure: Représentation d'un RNN

Pourquoi les RNN ?

- **Données séquentielles** : Les RNN sont conçus pour traiter des séquences (mots, frames vidéo, données temporelles).
- **Memory à court terme** : Ils maintiennent une mémoire des états précédents.
- **Flexibilité** : Peuvent gérer des entrées et sorties de longueurs variables.

Combinaison des signaux

- **Tokens passés | Token actuel**
- Les unités récurrentes parcourent séquentiellement les éléments d'une séquence.
- Elles mélangent l'information d'un élément avec les éléments précédents.



Figure: Unité récurrente combinant les signaux

Représentation déployée

- **Forme déployée** : Une manière de visualiser les RNN sur plusieurs pas de temps.
- Chaque pas de temps partage les mêmes paramètres (poids et biais).

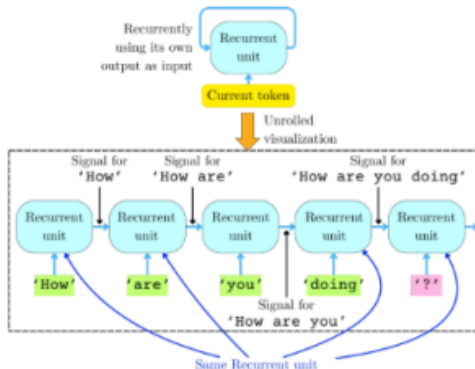


Figure: Représentation déployée d'une unité récurrente

Réseau de Jordan (1986)

- **Introduction** : Michael I. Jordan pour modéliser des données séquentielles.
- **Caractéristiques** :
 - Intègre des dépendances temporelles via des mécanismes de rétroaction.
 - Utilise la sortie précédente comme entrée pour l'état caché.

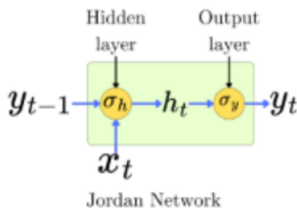


Figure: Architecture du réseau de Jordan

Équations du réseau de Jordan

- **État caché :**

$$h_t = \sigma(Wx_t + Uy_{t-1} + b)$$

- **Sortie :**

$$y_i = \sigma_y(W_y h_i + b_y)$$

- **Fonctions d'activation :**

- σ : Tangente hyperbolique (tanh).
- σ_y : Identité (régression) ou Softmax (classification).

Initialisation et itérations

- **Initialisation** : y_0 est un vecteur nul.
- **Première itération** : x_1 est le premier mot de la séquence.
- **Exemple** : Pour une phrase, chaque mot est traité séquentiellement.

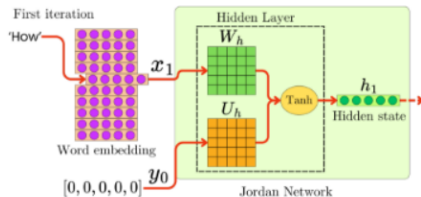


Figure: Première itération du réseau de Jordan

Couche de sortie

- **Rôle** : Mapper l'état caché h_i à la sortie y_i .
- **Applications** :
 - Classification : Softmax pour prédire des classes.
 - Régression : Fonction identité pour des valeurs continues.

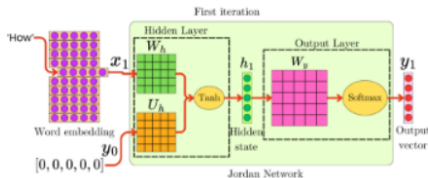


Figure: Couche de sortie du réseau de Jordan

Deuxième itération

- **Deuxième itération** : x_2 est le deuxième mot de la séquence.
- **Rétroaction** : La sortie y_1 de l'itération précédente est réutilisée.

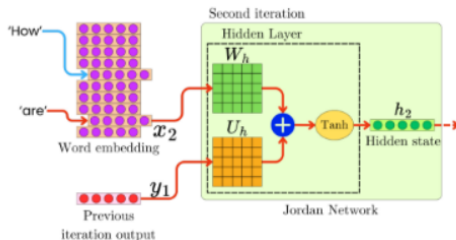


Figure: Deuxième itération du réseau de Jordan

Réseau d'Elman (1990)

- **Objectif** : Capturer des représentations intermédiaires plus abstraites.
- **Innovation** : Utilise l'état caché h_{t-1} comme signal de rétroaction (au lieu de y_t dans le réseau de Jordan).

- **Équations** :

- État caché :

$$h_t = \sigma_h(W_x x_t + U_h h_{t-1} + b_h)$$

- Sortie :

$$y_t = \sigma_y(W_h h_t + b_y)$$

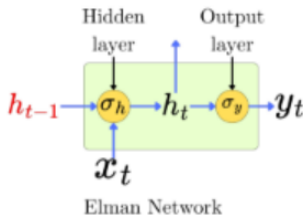


Figure: Architecture du réseau d'Elman

Avantages du réseau d'Elman

- **Récurrance d'état caché** : Permet d'apprendre des dépendances temporelles arbitraires.
- **Flexibilité** : La dimensionnalité de l'état caché peut être choisie librement.
- **Applications** : Modélisation de séquences complexes (langage, séries temporelles).

Problème des gradients disparus et explosifs

- **Contexte** : Les RNN simples ont du mal à apprendre des dépendances à long terme.
- **Problème** : Lors de l'entraînement sur des séquences longues, les gradients peuvent :
 - **Disparaître** : Devenir trop petits pour mettre à jour les poids.
 - **Exploser** : Devenir trop grands, causant une instabilité.

- Équation de l'état caché :

$$\mathbf{h}_t = \sigma_h(W_h \mathbf{x}_t + U_h \mathbf{h}_{t-1} + \mathbf{b}_h)$$

- Chaîne de dépendance :

$$\mathbf{h}_1 \rightarrow \mathbf{h}_2 \rightarrow \cdots \rightarrow \mathbf{h}_T$$

- **Dépendance de \mathbf{h}_T sur \mathbf{h}_1 :**

$$\frac{\partial \mathbf{h}_T}{\partial \mathbf{h}_1} = \frac{\partial \mathbf{h}_T}{\partial \mathbf{h}_{T-1}} \frac{\partial \mathbf{h}_{T-1}}{\partial \mathbf{h}_{T-2}} \cdots \frac{\partial \mathbf{h}_2}{\partial \mathbf{h}_1}$$

- **Simplification :** Si σ_h est la fonction identité :

$$\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}} = U_h$$

- **Chaîne complète :**

$$\frac{\partial \mathbf{h}_T}{\partial \mathbf{h}_1} = U_h^{T-1}$$

- **Condition** : Si les valeurs propres de U_h ont des magnitudes < 1 .
- **Effet** :

$$\left\| U_h^{T-1} \right\| \rightarrow 0 \quad \text{quand} \quad T \rightarrow \infty$$

- **Conséquence** : Les gradients deviennent trop petits, et le réseau "oublie" les informations à long terme.

- **Condition** : Si les valeurs propres de U_h ont des magnitudes > 1 .
- **Effet** :

$$\left\| U_h^{T-1} \right\| \rightarrow \infty \quad \text{quand} \quad T \rightarrow \infty$$

- **Conséquence** : Les gradients deviennent trop grands, causant des mises à jour instables (voire des **NaN**).

- **Gradients disparus :**

- Le réseau ne peut pas apprendre des dépendances à long terme.
- Exemple : Difficulté à retenir des informations au début d'une séquence longue.

- **Gradients explosifs :**

- Instabilité pendant l'entraînement.
- Risque de divergence des poids.

- **Initialisation soignée des poids** : Par exemple, initialisation de Xavier ou He.
- **Utilisation de fonctions d'activation adaptées** : Comme ReLU ou des variantes.
- **Architectures avancées** : LSTM, GRU, Transformers.
- **Troncature des gradients** : Pour éviter l'explosion des gradients.

- **Initialisation soignée des poids** : Par exemple, initialisation de Xavier ou He.
- **Utilisation de fonctions d'activation adaptées** : Comme ReLU ou des variantes.
- **Architectures avancées** : LSTM, GRU, Transformers.
- **Troncature des gradients** : Pour éviter l'explosion des gradients.
- Les gradients disparus et explosifs sont des problèmes majeurs dans les RNN simples.
- Ces limitations ont motivé le développement d'architectures plus robustes comme les LSTM et les Transformers.

- **Problématique** : Les RNN simples ont du mal à apprendre des dépendances à long terme (gradients disparus/explosifs).
- **Solution** : Les LSTM introduisent des portes pour contrôler le flux d'information.
- **Historique** : Proposé par Sepp Hochreiter et Jürgen Schmidhuber en 1997.
- **Impact** : Les LSTM ont révolutionné le traitement du langage naturel, la reconnaissance vocale, et d'autres tâches séquentielles.
- **Applications modernes** : Traduction automatique (Google Translate), génération de texte, prédiction de séries temporelles.

Architecture des LSTM

- **Composants clés :**

- **État caché** (h_t) : Représente la mémoire à court terme.
- **État de cellule** (c_t) : Représente la mémoire à long terme.
- **Portes** : Contrôlent le flux d'information.

- **Entrées :**

- x_t : Entrée actuelle au temps t .
- h_{t-1} : État caché précédent.
- c_{t-1} : État de la cellule précédent.

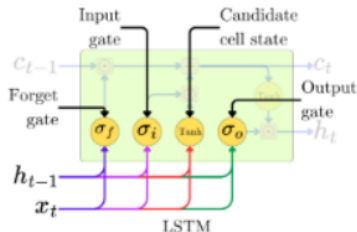


Figure: Architecture d'une cellule LSTM

- **Porte d'oubli** (f_t) : Décide quelle partie de l'ancien état de cellule conserver.

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f)$$

- **Porte d'entrée** (i_t) : Décide quelle partie du nouvel état de cellule candidat ajouter.

$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i)$$

- **Porte de sortie** (o_t) : Décide quelle partie de l'état de cellule exposer comme état caché.

$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o)$$

- **Fonction logistique** (σ) : Varie entre 0 et 1, agit comme un interrupteur.

Fonctionnement des portes

- **Porte d'oubli** : Si $f_t \approx 0$, l'information est oubliée. Si $f_t \approx 1$, l'information est conservée.
- **Porte d'entrée** : Si $i_t \approx 0$, le nouvel état est ignoré. Si $i_t \approx 1$, le nouvel état est ajouté.
- **Porte de sortie** : Si $o_t \approx 0$, l'état de cellule est masqué. Si $o_t \approx 1$, l'état de cellule est exposé.

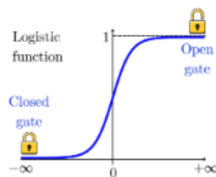


Figure: Fonction logistique utilisée dans les portes LSTM

Les entrées dans une cellule LSTM

- **Entrées principales :**
 - \mathbf{x}_t : Entrée actuelle au temps t .
 - \mathbf{h}_{t-1} : État caché précédent.
- **Portes :** Les entrées sont mélangées à travers quatre composants :
 - Porte d'oubli.
 - Porte d'entrée.
 - Porte de sortie.
 - Couche de génération de l'état de cellule candidat.

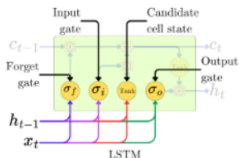


Figure: Les entrées \mathbf{x}_t et \mathbf{h}_{t-1} sont mélangées à travers quatre composants dans une cellule LSTM.

Multiplication élément par élément

- **Opération** : La multiplication élément par élément (\odot) combine deux vecteurs de même taille.
- **Exemple** :

$$\begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \odot \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} a_1 \cdot b_1 \\ a_2 \cdot b_2 \\ a_3 \cdot b_3 \end{bmatrix}$$

- **Utilisation dans les LSTM** :
 - Combinaison de l'ancien état de cellule et du nouvel état candidat.
 - Contrôle du flux d'information via les portes.

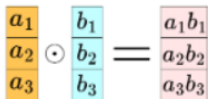

$$\begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \odot \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} a_1 b_1 \\ a_2 b_2 \\ a_3 b_3 \end{bmatrix}$$

Figure: La multiplication élément par élément de deux vecteurs conduit à un nouveau vecteur de même taille.

- Les entrées dans une cellule LSTM sont traitées à travers des portes pour contrôler le flux d'information.
- La multiplication élément par élément est une opération clé pour combiner les états.
- Ces mécanismes permettent aux LSTM de gérer des dépendances à long terme efficacement.

- **Génération** : Un nouvel état de cellule candidat est généré.

$$\tilde{c}_t = \tanh(W_c x_t + U_c h_{t-1} + b_c)$$

- **Fonction tanh** : Varie entre -1 et +1, permet de capturer des informations positives et négatives.
- **Rôle** : Représente une nouvelle information potentielle à ajouter à l'état de cellule.

Mise à jour de l'état de cellule

- **Combinaison** : L'état de cellule est mis à jour en combinant l'ancien état et le nouvel état candidat.

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

- **Multiplication élément par élément (\odot)** : Permet de mélanger les informations de manière contrôlée.
- **Exemple** :
 - Si $f_t = 0$ et $i_t = 1$, l'état de cellule est entièrement mis à jour.
 - Si $f_t = 1$ et $i_t = 0$, l'état de cellule reste inchangé.



Figure: Mise à jour de l'état de cellule

- **État caché** : Expose une partie de l'état de cellule via la porte de sortie.

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

- **Rôle** : L'état caché h_t est utilisé pour la prédiction et la rétropropagation.
- **Exemple** : Dans une tâche de traduction, h_t représente la compréhension contextuelle de la phrase jusqu'au temps t .

- Équation de mise à jour :

$$\mathbf{c}_i = \mathbf{f}_i \odot \mathbf{c}_{i-1} + \mathbf{i}_i \odot \tilde{\mathbf{c}}_i$$

- Dépendance des gradients :

$$\frac{\partial \mathbf{c}_i}{\partial \mathbf{c}_{i-1}} = \mathbf{f}_i$$

- Chaîne de dépendance :

$$\mathbf{c}_1 \rightarrow \mathbf{c}_2 \rightarrow \cdots \rightarrow \mathbf{c}_T$$

- Accumulation des gradients :

$$\frac{\partial \mathbf{c}_T}{\partial \mathbf{c}_1} = \prod_{i=2}^T \mathbf{f}_i$$

- **Porte d'oubli (f_i)** : Résultat d'une fonction logistique, donc $0 < f_i < 1$.
- **Stabilité des gradients** :
 - Si $f_i \approx 1$, les gradients restent stables.
 - Si $f_i \approx 0$, les gradients décroissent.
- **Exemple** : Si $f_i = 0.99$ sur 100 étapes, le produit reste proche de 1.
- **Impact** : L'état de cellule \mathbf{c}_T est bien adapté pour la mémoire à long terme.

Visualisation de la mémoire à long terme

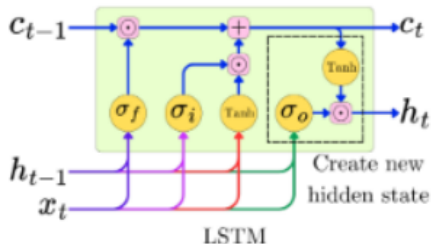


Figure: La couche LSTM produit deux états : \mathbf{c}_i (mémoire à long terme) et \mathbf{h}_i (mémoire à court terme).

- **Dérivée de la porte de sortie :**

$$\frac{\partial \mathbf{h}_t}{\partial \mathbf{o}_t}$$

- Peut être petite ou grande selon la pente de la sigmoïde.
- Si la sigmoïde est saturée (proche de 0 ou 1), la dérivée est petite.

- **Dérivée de $\tanh(\mathbf{c}_t)$:**

$$\frac{\partial \tanh(\mathbf{c}_t)}{\partial \mathbf{c}_t} = 1 - \tanh^2(\mathbf{c}_t)$$

- Proche de 0 si \mathbf{c}_t est saturé (proche de -1 ou 1).

- **Dérivée partielle :**

$$\frac{\partial \mathbf{c}_t}{\partial \mathbf{h}_{t-1}}$$

- Dépend des portes d'oubli, d'entrée et de l'état de cellule candidat.
- Chacune de ces composantes peut saturer ou diminuer les signaux.

- **Accumulation des gradients :**

- Les gradients de \mathbf{h}_T à \mathbf{h}_1 accumulent plusieurs facteurs de portes.
- Ces facteurs sont bornés dans $[0, 1]$, ce qui entraîne une décroissance exponentielle sur de longues séquences.

- **Décroissance des gradients :**

- Les gradients peuvent devenir trop petits pour mettre à jour les poids efficacement.
- Cela rend difficile l'apprentissage de dépendances à long terme.

- **Explosion partielle des gradients :**

- Si les portes poussent fortement dans certaines directions, les gradients peuvent partiellement exploser.
- Cependant, les signaux à long terme sont plus susceptibles de se dégrader.

Pourquoi \mathbf{h}_t est adapté à la mémoire à court terme

- **Comportement de \mathbf{h}_t :**

- L'état caché \mathbf{h}_t dépend de la porte de sortie o_t et de $\tanh(\mathbf{c}_t)$.
- Ces composantes peuvent fluctuer rapidement, reflétant des changements immédiats.

- **Gradients sensibles :**

- Les gradients dans \mathbf{h}_t sont plus sensibles aux changements à court terme.
- Cela rend \mathbf{h}_t mieux adapté pour la mémoire à court terme.

- Les gradients dans l'état caché \mathbf{h}_t sont influencés par les portes et $\tanh(\mathbf{c}_t)$.
- La décroissance des gradients rend \mathbf{h}_t plus adapté à la mémoire à court terme.
- Ces mécanismes expliquent pourquoi les LSTM sont efficaces pour gérer à la fois la mémoire à court et à long terme.

- **Contexte** : Introduites en 2014 par Cho et al. comme une alternative aux LSTM.
- **Objectif** : Résoudre le problème des gradients disparus tout en simplifiant l'architecture.
- **Avantages** :
 - Moins de portes que les LSTM (2 au lieu de 3).
 - Performances similaires aux LSTM pour de nombreuses tâches.

Architecture des GRU

- **Entrées :**

- \mathbf{x}_t : Entrée actuelle au temps t .
- \mathbf{h}_{t-1} : État caché précédent.

- **Portes :**

- Porte de mise à jour (\mathbf{z}_t).
- Porte de réinitialisation (\mathbf{r}_t).

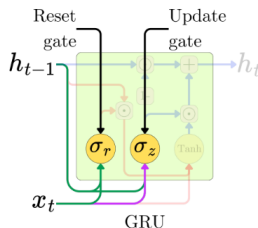


Figure: Les entrées \mathbf{x}_t et \mathbf{h}_{t-1} sont mélangées à travers deux portes : la porte de mise à jour et la porte de réinitialisation.

- **Rôle** : Décide quelle partie de l'état caché précédent ignorer.
- **Équation** :

$$\mathbf{r}_t = \sigma(W_r \mathbf{x}_t + U_r \mathbf{h}_{t-1} + \mathbf{b}_r)$$

- **Comportement** :
 - Si $\mathbf{r}_t \approx 0$: L'état caché précédent est ignoré (réinitialisation).
 - Si $\mathbf{r}_t \approx 1$: L'état caché précédent est conservé.

État caché candidat

- **Génération** : Un nouvel état caché candidat est généré.
- **Équation** :

$$\tilde{\mathbf{h}}_t = \tanh(W_h \mathbf{x}_t + U_h(\mathbf{r}_t \odot \mathbf{h}_{t-1}) + \mathbf{b}_h)$$

- **Rôle** : Injecte de nouvelles informations à partir de l'entrée actuelle.

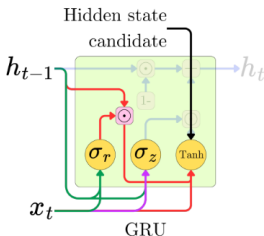


Figure: L'état caché candidat permet à la cellule d'injecter de nouvelles informations.

- **Rôle** : Décide combien d'informations anciennes et nouvelles conserver.
- **Équation** :

$$\mathbf{z}_t = \sigma(W_z \mathbf{x}_t + U_z \mathbf{h}_{t-1} + \mathbf{b}_z)$$

- **Comportement** :
 - Si $\mathbf{z}_t \approx 1$: L'état caché est remplacé par le candidat.
 - Si $\mathbf{z}_t \approx 0$: L'état caché précédent est conservé.

Mise à jour de l'état caché

- Équation :

$$\mathbf{h}_t = (1 - \mathbf{z}_t) \odot \mathbf{h}_{t-1} + \mathbf{z}_t \odot \tilde{\mathbf{h}}_t$$

- Interprétation :

- \mathbf{z}_t agit comme un curseur entre mémoire à court terme (nouvelle information) et mémoire à long terme (ancienne information).
- Si $\mathbf{z}_t \approx 1$: Privilégie la nouvelle information.
- Si $\mathbf{z}_t \approx 0$: Privilégie l'ancienne information.

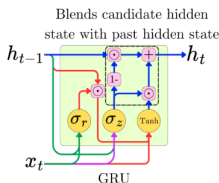


Figure: La porte de mise à jour permet d'équilibrer l'ancienne et la nouvelle information.

- **Simplicité** : Moins de paramètres que les LSTM (2 portes au lieu de 3).
- **Efficacité** : Performances similaires aux LSTM pour de nombreuses tâches.
- **Applications** :
 - Traduction automatique.
 - Génération de texte.
 - Prédiction de séries temporelles.

- Les GRU sont une alternative simplifiée et efficace aux LSTM.
- Elles utilisent deux portes pour gérer les dépendances à court et à long terme.
- **Perspectives** : Les GRU continuent d'être largement utilisées dans les tâches de traitement de séquences.

Long-Term Memory vs Short-Term Memory

- **Objectif** : Comprendre comment les GRU gèrent les gradients pour éviter leur disparition ou explosion.
- **Porte de mise à jour (z_t)** : Contrôle le mélange entre mémoire à long terme et à court terme.
- **Comportement asymptotique** :
 - Si $z_t \approx 0$: Le modèle conserve la mémoire à long terme.
 - Si $z_t \approx 1$: Le modèle privilégie la mémoire à court terme.

Comportement des gradients

- **Cas extrême** : $z_t \approx 0$ (mémoire à long terme).

- **Dérivée partielle** :

$$\frac{\partial z_t}{\partial h_{t-1}} \approx 0$$

- **Conséquence** :

$$\frac{\partial h_t}{\partial h_{t-1}} \approx 1 - z_t$$

- **Chaîne de dépendance** :

$$h_1 \rightarrow h_2 \rightarrow \dots \rightarrow h_T$$

- **Accumulation des gradients** :

$$\frac{\partial h_T}{\partial h_1} = \prod_{t=2}^T (1 - z_t)$$

Stabilité des gradients

- **Propriété** : $1 - z_t \lesssim 1$ pour chaque itération.
- **Impact** : Le produit des gradients décroît lentement, ce qui permet de capturer des dépendances à long terme.
- **Avantage** : Les gradients ne disparaissent pas rapidement, même sur de longues séquences.

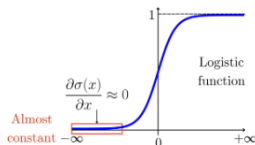


Figure: La fonction logistique est presque constante dans la région asymptotique.

- **Cas extrême** : $z_t \approx 1$ (mémoire à court terme).
- **Dérivée partielle** :

$$\frac{\partial h_t}{\partial h_{t-1}} \approx z_t \frac{\partial h_t}{\partial h_{t-1}}$$

- **Accumulation des gradients** :
 - Les gradients sont influencés par plusieurs facteurs de portes.
 - Ces facteurs sont bornés dans $[0, 1]$, ce qui entraîne une décroissance rapide des gradients.
- **Impact** : Le réseau est configuré pour capturer des dépendances à court terme.

- Les GRU utilisent la porte de mise à jour z_t pour équilibrer mémoire à long terme et à court terme.
- **Mémoire à long terme** : Les gradients décroissent lentement, permettant de capturer des dépendances à long terme.
- **Mémoire à court terme** : Les gradients décroissent rapidement, adapté pour des changements immédiats.
- **Perspectives** : Ces mécanismes expliquent pourquoi les GRU sont efficaces pour les tâches séquentielles.

- Équation de l'état caché :

$$\mathbf{h}_k = \mathbf{o}_k \odot \tanh(\mathbf{c}_k)$$

- Dépendance des gradients :

$$\frac{\partial \mathbf{h}_k}{\partial \mathbf{h}_{k-1}} = \frac{\partial \mathbf{h}_k}{\partial \mathbf{c}_k} \cdot \frac{\partial \mathbf{c}_k}{\partial \mathbf{h}_{k-1}} + \frac{\partial \mathbf{h}_k}{\partial \mathbf{o}_k} \cdot \frac{\partial \mathbf{o}_k}{\partial \mathbf{h}_{k-1}}$$

- Comportement :

- Les gradients peuvent fluctuer rapidement.
- L'état caché \mathbf{h}_k est mieux adapté pour la mémoire à court terme.

Propriétés des gradients dans l'état caché

- **Porte de sortie (\mathbf{o}_k)** : Contrôle la quantité d'information exposée.
- **Fonction \tanh** : Peut saturer, réduisant les gradients.
- **Impact** : Les gradients dans \mathbf{h}_k sont plus sensibles aux changements immédiats.

Comparaison Mémoire Long Terme vs Court Terme

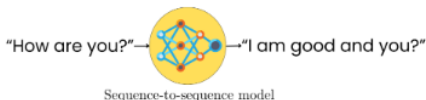
- **Mémoire à long terme (c_i) :**
 - Gradients stables sur de longues séquences.
 - Adapté pour retenir des informations sur de longues périodes.
- **Mémoire à court terme (h_i) :**
 - Gradients fluctuants.
 - Adapté pour des changements immédiats ou contextuels.

Comparaison Mémoire Long Terme vs Court Terme

- Les LSTM utilisent deux types de mémoire : à long terme (\mathbf{c}_i) et à court terme (\mathbf{h}_i).
- La porte d'oubli stabilise les gradients pour la mémoire à long terme.
- La porte de sortie permet des ajustements rapides pour la mémoire à court terme.
- Ces mécanismes font des LSTM un outil puissant pour les tâches séquentielles complexes.

Modèles Sequence-to-Sequence (Seq2Seq)

Les modèles de langage modernes (LLMs) et l'architecture des transformers sont apparus comme une évolution dans la quête de construction de modèles de séquence à séquence (seq2seq). Un modèle de séquence à séquence est un modèle qui prend en entrée une séquence de texte et génère une autre séquence de longueur potentiellement différente en sortie.



- **Traduction automatique** : traduire un texte d'une langue à une autre.
- **Résumé de texte** : condenser un texte long en un résumé concis.
- **Réponse aux questions** : générer des réponses à des questions basées sur un contexte.
- **IA conversationnelle (chatbots)** : réponses conversationnelles multi-tours aux requêtes des utilisateurs.

Évolution des Modèles Seq2Seq

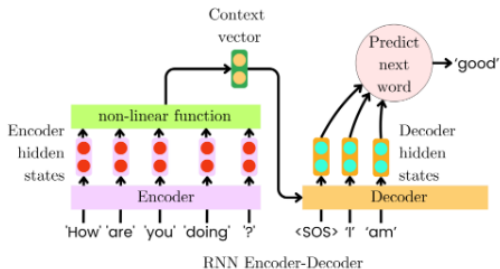
- **2014** : Introduction des modèles seq2seq avec l'architecture encodeur-décodeur RNN (Sutskever et al.).
- **2014** : Intégration des mécanismes d'attention (Bahdanau).
- **2015** : Raffinement des mécanismes d'attention (Luong).
- **2016-2017** : Exploration des réseaux de neurones convolutifs (CNN) pour le traitement parallèle.
- **2017** : Introduction du *Transformer* dans l'article *Attention Is All You Need* (Vaswani et al.).



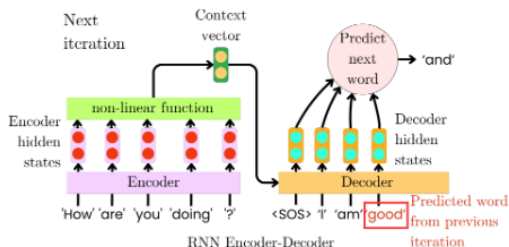
L'architecture encodeur-décodeur RNN a été développée comme un modèle autoregressif pour prédire une nouvelle séquence de texte à partir d'une séquence de texte en entrée. Il y a trois composants principaux dans le modèle :

- **L'encodeur** : il compresse la séquence d'entrée en un vecteur de contexte de dimension fixe.
- **Le décodeur** : il génère la séquence de sortie étape par étape.
- **La tête de prédiction** : elle convertit l'état caché du décodeur en une distribution de probabilité sur le vocabulaire cible.

Architecture Encodeur-Décodeur RNN



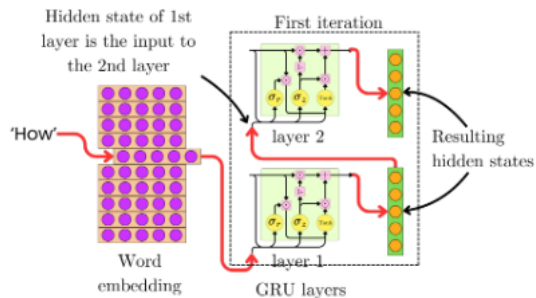
Architecture Encodeur-Décodeur RNN



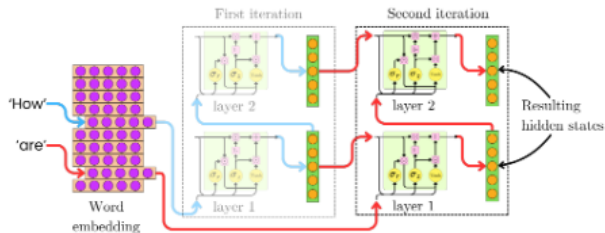
L'encodeur est composé d'une couche d'embedding et de plusieurs couches d'unités récurrentes (GRU). Pour chaque mot de la séquence d'entrée, sa représentation vectorielle x_t est utilisée comme entrée au réseau GRU, qui produit les états cachés correspondants.

$$h_t^i = \begin{cases} \text{GRU}_{\text{enc}}(x_t, h_{t-1}^i) & \text{si } i = 1, \\ \text{GRU}_{\text{enc}}(h_t^{i-1}, h_{t-1}^i) & \text{si } i > 1. \end{cases} \quad (1)$$

Architecture Encodeur-Décodeur RNN



Architecture Encodeur-Décodeur RNN



À chaque itération, les états cachés résultants encodent la séquence d'entrée complète jusqu'au mot associé à l'itération actuelle :

$$\mathbf{h}_t^i = \text{RNN}_{\text{suc}}^i(\mathbf{x}_1, \dots, \mathbf{x}_t)$$

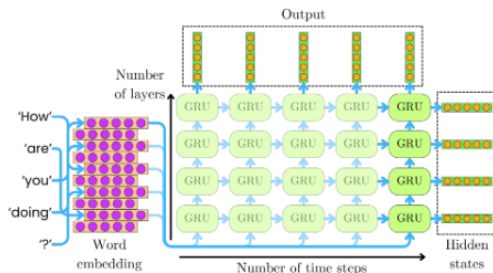
Une fois que nous avons itéré à travers tous les mots de la séquence d'entrée, nous obtenons deux ensembles de vecteurs :

- Les états cachés $[\mathbf{h}_1^L, \dots, \mathbf{h}_T^L]$ pour chaque itération t de la dernière couche L .
- Les états cachés $[\mathbf{h}_T^1, \dots, \mathbf{h}_T^L]$ de la dernière itération T pour toutes les couches L .

Nous faisons souvent référence au premier ensemble de vecteurs comme les "états de sortie" et au second comme les "états cachés".

Dans le contexte de l'architecture encodeur-décodeur RNN, nous allons utiliser ceux que l'on appelle les "états cachés". Ces états cachés encodent des informations cruciales pour la génération de la séquence de sortie.

Extraction des États Cachés



Nous pouvons extraire deux ensembles de vecteurs d'états cachés de l'encodeur : les états cachés $[\mathbf{h}_1^1, \dots, \mathbf{h}_T^M]$ pour chaque itération t de la dernière couche L , et les états cachés $[\mathbf{h}_T^1, \dots, \mathbf{h}_T^L]$ de la dernière itération T pour toutes les couches t .

La séquence d'entrée complète est encodée dans ces états cachés et est utilisée comme vecteur de contexte pour le décodeur afin de générer une réponse.

Le décodeur est similaire à l'encodeur mais est initialisé avec le token de début de séquence $\langle \text{SOS} \rangle$ et les états cachés de l'encodeur.

$$s_t^i = \begin{cases} \text{GRU}'_{\text{dec}}(\mathbf{y}_t, \mathbf{s}_{t-1}^i) & \text{si } i = 1, \\ \text{GRU}'_{\text{dec}}(s_{t-1}^{tr}, s_{t-1}^i) & \text{si } i > 1. \end{cases} \quad (2)$$

Architecture Encodeur-Décodeur RNN

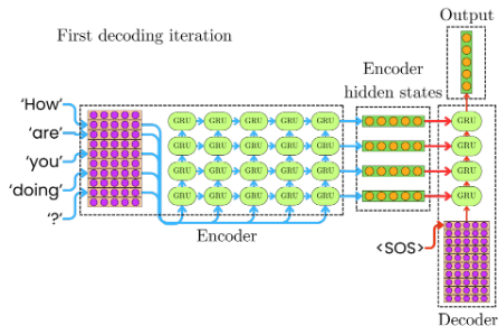


Figure: Lors de la première itération du processus de décodage, le décodeur prend comme entrée les états cachés de l'encodeur \mathbf{h}_t^b et la représentation vectorielle du token **<SOS>** (début de séquence).

Architecture Encodeur-Décodeur RNN

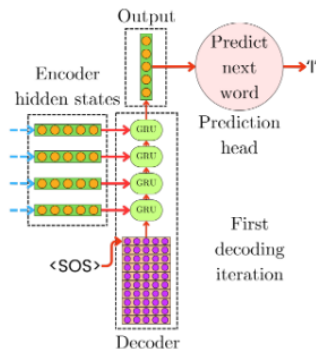


Figure: La tête de prédiction utilise la sortie du décodeur pour prédire le mot suivant.

Architecture Encodeur-Décodeur RNN

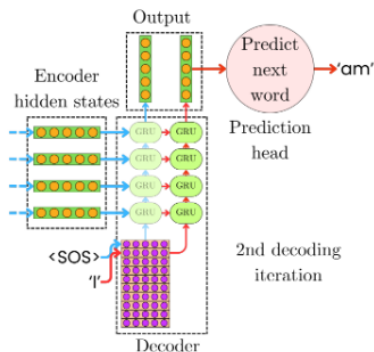


Figure: Lors de la deuxième itération, nous utilisons le mot prédit de l'itération précédente pour continuer le processus de décodage.

Architecture Encodeur-Décodeur RNN

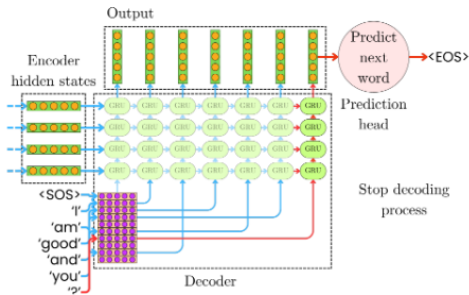


Figure: Le processus de décodage s'arrête lorsque le modèle prédit que le mot suivant est le token <EOS> (fin de séquence).

À chaque itération, l'interaction avec la séquence d'entrée se fait à travers les états cachés de l'encodeur :

$$\mathbf{s}_t^L = \text{RNN}_{\text{dec}}^L(\mathbf{y}_1, \dots, \mathbf{y}_t, \mathbf{h}_1^L, \dots, \mathbf{h}_T^L)$$

(5)

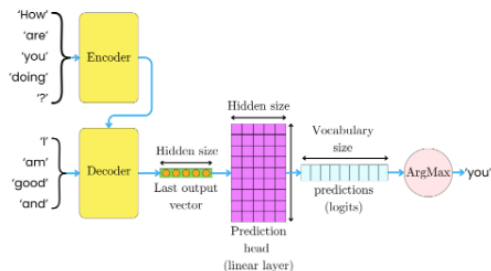
Fin du Processus de Décodage

Nous itérons ce processus de décodage jusqu'à ce que le modèle émette un token de fin de séquence $\langle \text{EIS} \rangle$. Dans ce cas, nous interrompons le processus de décodage et retournons la séquence de sortie.

La tête de prédiction est un classifieur qui produit une probabilité pour chaque mot du vocabulaire. Le vecteur de sortie du décodeur est transformé en logits via une couche linéaire.

$$\text{Prochain mot} = \arg \max_{\text{word}} W_p \mathbf{s}_t^L \quad (3)$$

La Tête de Prédiction



La perte est calculée sur la séquence cible étant donnée la séquence source :

$$\mathcal{L} = - \sum_{i=1}^{T'} P(y_i^* | y_1^*, \dots, y_{L-1}^*, \mathbf{b}_T^*, \dots, \mathbf{b}_{T'}^*) \quad (4)$$

La transformation Softmax est utilisée pour estimer les probabilités des mots.

$$\mathbf{p}_t = \text{Softmax}(\mathbf{l}_t) = \frac{e^{\mathbf{l}_t}}{\sum_{i=1}^{|\mathcal{V}|} e^{U_t}} \quad (5)$$

Problèmes de l'encodeur-décodeur RNN

- Compression de la séquence en un seul vecteur de taille fixe.
- Perte d'informations pour les longues séquences.
- Difficulté à gérer les dépendances distantes.

Solution : Mécanisme d'attention de Bahdanau

- Vecteurs de contexte dynamiques.
- Permet au décodeur de se concentrer sur des parties spécifiques de l'entrée.

La couche d'attention

- Combine les vecteurs de sortie de l'encodeur et les états cachés du décodeur.
- Trois couches linéaires : W_h , W_s , et v .
- Utilisation de \tanh pour combiner les vecteurs.

La couche d'attention

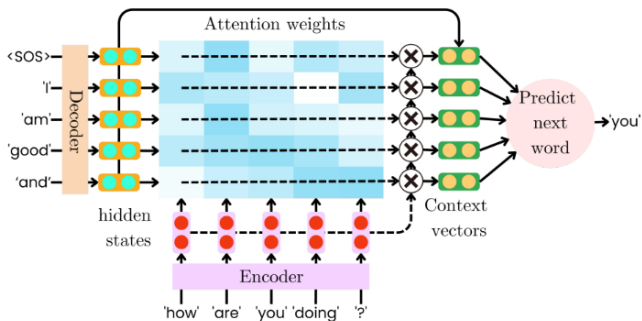


Figure: Calcul des poids d'attention entre les mots de l'entrée et de la sortie.

La couche d'attention

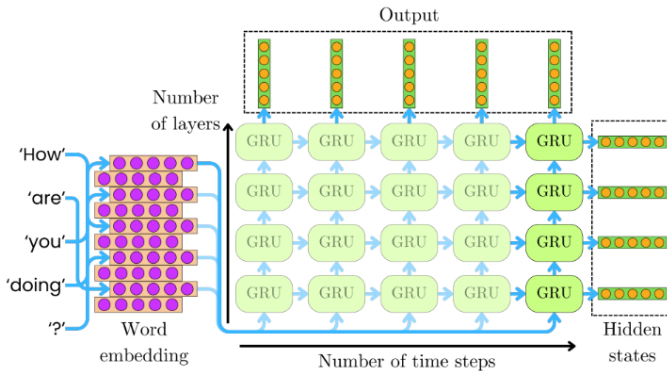


Figure: Comparaison entre l'encodeur-décodeur RNN et l'attention de Bahdanau.

$$e_{tt'} = \mathbf{v}^\top \tanh(W_h \mathbf{h}_t^L + W_s \mathbf{s}_{t'}^L)$$

- W_h et W_s : matrices carrées.
- \mathbf{v} : couche linéaire unidimensionnelle.
- Scores normalisés par Softmax.

La couche d'attention

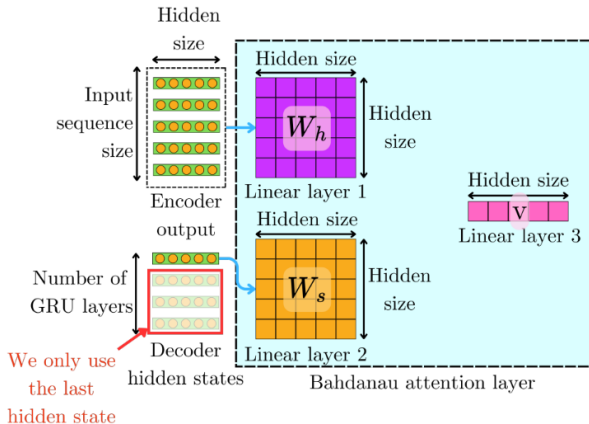


Figure: Composition de la couche d'attention.

$$\mathbf{c}_{t'} = \sum_{t=1}^T a_{tt'} \mathbf{h}_t^L$$

- Moyenne pondérée des vecteurs de sortie de l'encodeur.
- Résumé dynamique de la séquence d'entrée.

- Initialisation avec le token $\langle \text{SOS} \rangle$.
- Concaténation du vecteur de contexte et des entrées du décodeur.
- Utilisation des GRU pour prédire les tokens suivants.

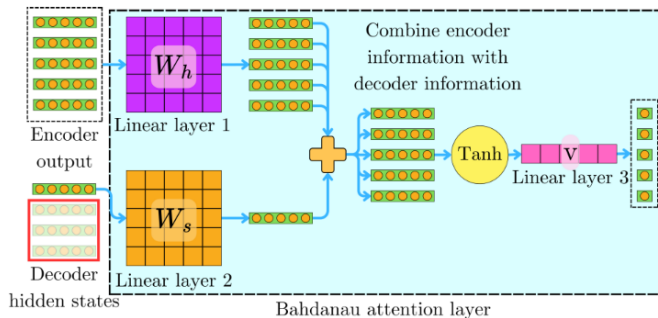


Figure: Calcul du premier vecteur de contexte.

vecteur de contexte et token

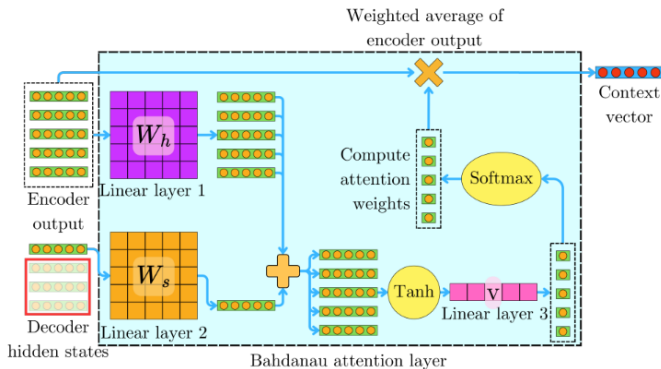


Figure: Concaténation du vecteur de contexte et du token <SOS>.

- Calcul des poids d'attention avec les états cachés du décodeur.
- Concaténation du nouveau vecteur de contexte.
- Prédiction du token suivant.

vecteur de contexte

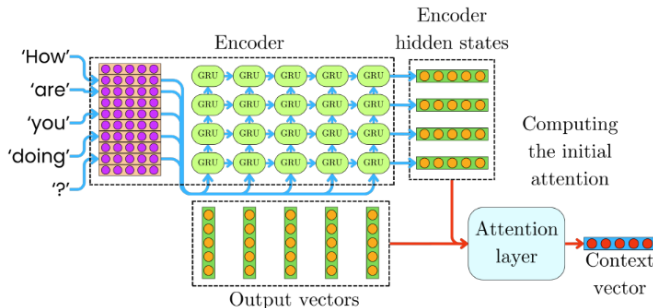


Figure: Calcul du deuxième vecteur de contexte.

prédiction du mot suivant

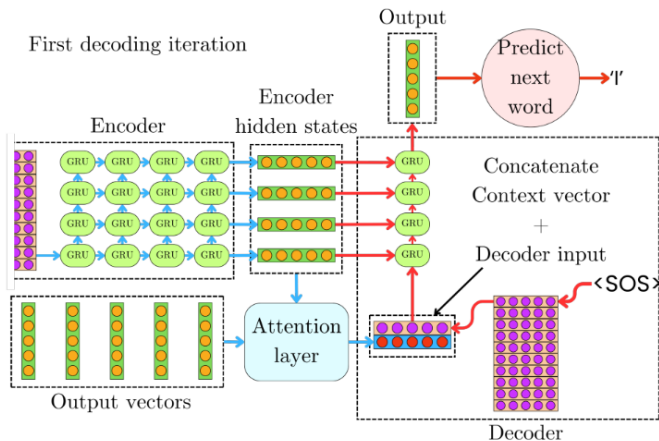


Figure: Concaténation du vecteur de contexte et prédiction du mot suivant.

prédiction du mot suivant

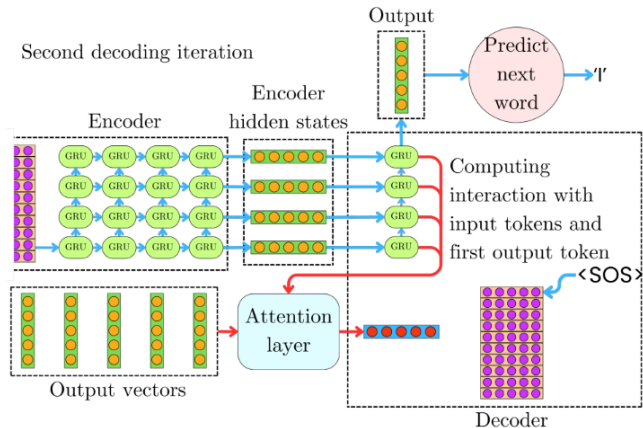


Figure: Calcul du deuxième vecteur de contexte.

prédiction du mot suivant

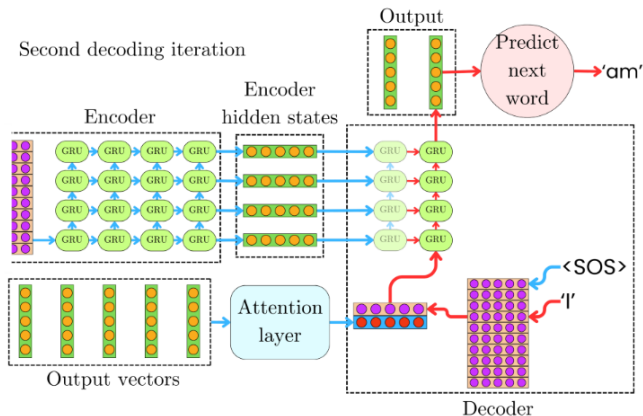


Figure: Concaténation du vecteur de contexte et prédiction du mot suivant.

Avantages de l'attention

- Pas de dépendance à une représentation de taille fixe.
- Visualisation des poids d'attention.
- Meilleure gestion des longues séquences.

Équations clés du mécanisme d'attention

- Score d'alignement :

$$e_{tt'} = \mathbf{v}^\top \tanh(W_h \mathbf{h}_t^L + W_s \mathbf{s}_{t'}^L)$$

- Poids d'attention (Softmax) :

$$a_{tt'} = \frac{\exp(e_{tt'})}{\sum_{k=1}^T \exp(e_{kt'})}$$

- Vecteur de contexte :

$$\mathbf{c}_{t'} = \sum_{t=1}^T a_{tt'} \mathbf{h}_t^L$$

- Initialisation du vecteur de contexte :

$$\mathbf{c}_1 = \text{Attention}(\mathbf{h}_1^L, \dots, \mathbf{h}_T^L; \mathbf{h}_T^L)$$

- Concaténation avec l'entrée du décodeur :

$$\mathbf{y}'_1 = [\mathbf{y}_1, \mathbf{c}_1] \quad \text{concaténation}$$

- Mise à jour des états cachés du décodeur :

$$\mathbf{s}_1^i = \begin{cases} \text{GRU}_{\text{dec}}^i(\mathbf{y}'_1, \mathbf{h}_T^i) & \text{si } i = 1, \\ \text{GRU}_{\text{dec}}^i(\mathbf{s}_1^{i-1}, \mathbf{h}_T^i) & \text{si } i > 1, \end{cases}$$

Visualisation des poids d'attention

- Les poids d'attention $a_{tt'}$ montrent comment le décodeur se concentre sur les tokens de l'entrée.
- Ces poids peuvent être visualisés pour comprendre les alignements entre les tokens source et cible.

- L'attention de Bahdanau améliore la gestion des longues séquences.
- Elle permet une meilleure modélisation des dépendances distantes.
- Les poids d'attention offrent une interprétabilité accrue.

- Développée en 2015, similaire à l'attention de Bahdanau.
- Introduit des mécanismes d'attention globale et locale.
- Se concentre ici sur l'attention globale.
- Propose des mécanismes de calcul d'alignement multiplicatifs, réduisant les coûts de calcul.

- Mécanisme d'attention multiplicatif avec trois variantes.
- Première variante : Scores d'alignement par produit scalaire.

$$e_{tt'} = \text{score}(\mathbf{h}_t^L, \mathbf{s}_{t'}^L) = \mathbf{h}_t^{L^T} \mathbf{s}_{t'}^L, \quad \text{produit scalaire}$$

L'Architecture

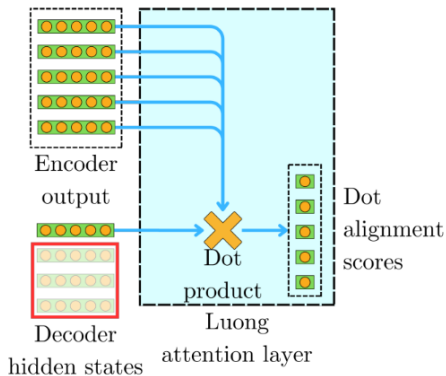


Figure: Calcul des scores d'alignement par produit scalaire dans la couche d'attention de Luong

- Deuxième variante : Scores d'alignement généraux avec paramètres d'apprentissage W_s .

$$e_{tt'} = \text{score}(\mathbf{h}_t^L, \mathbf{s}_{t'}^L) = \mathbf{h}_t^{L^T} W_s \mathbf{s}_{t'}^L, \quad \text{général}$$

L'Architecture

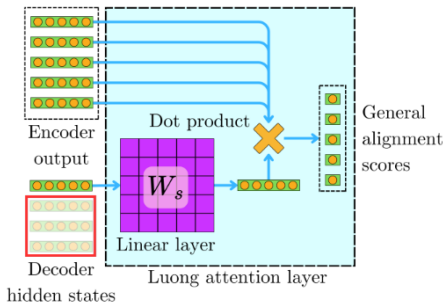


Figure: Calcul des scores d'alignement généraux dans la couche d'attention de Luong

- Troisième variante : Concaténation des sorties de l'encodeur et de l'état caché du décodeur.

$$e_{tt'} = \text{score}(\mathbf{h}_t^L, s_{t'}^L) = \mathbf{v}^\top \tanh \left(W_s \left[\mathbf{h}_t^L; s_{t'}^L \right] \right), \quad \text{concaténation}$$

(6)

L'Architecture

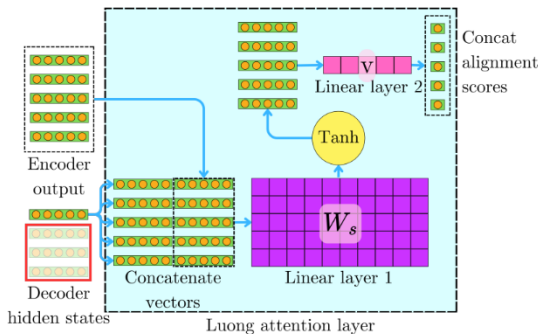


Figure: Calcul des scores d'alignement par concaténation dans la couche d'attention de Luong

- Poids d'attention calculés avec Softmax :

$$a_{tt'} = \frac{\exp(e_{tt'})}{\sum_{k=1}^T \exp(e_{kt'})}$$

(7)

- Vecteur de contexte comme moyenne pondérée des vecteurs de sortie de l'encodeur :

$$\mathbf{c}_{t'} = \sum_{t=1}^T a_{tt'} \mathbf{h}_t^L \quad (2.71)$$

Poids d'Attention et Vecteur de Contexte

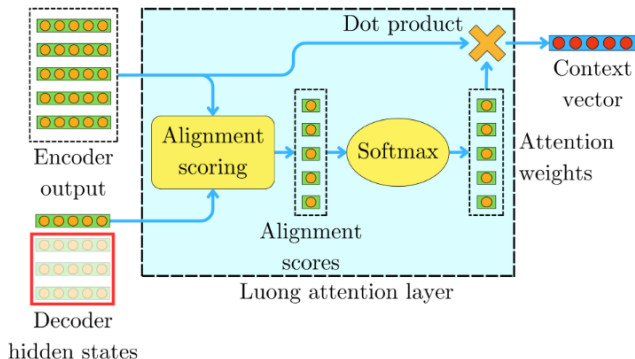


Figure: Après le calcul des scores d'alignement, ils sont normalisés en poids d'attention par une transformation Softmax et utilisés pour calculer une moyenne pondérée des vecteurs de sortie de l'encodeur

Le Processus de Décodage

- Similaire au réseau de Bahdanau, avec des différences dans l'ordre des opérations.
- Initialisation des états cachés du décodeur avec les états cachés de l'encodeur \mathbf{h}_T^i .
- Calcul du premier vecteur de sortie du décodeur \mathbf{s}_1^L en utilisant le token $\langle \text{SOS} \rangle$:

$$\mathbf{s}_1^i = \begin{cases} \text{GRU}_{\text{dec}}^i(\mathbf{y}_1, \mathbf{h}_T^i) & \text{si } i = 1, \\ \text{GRU}_{\text{dec}}^i(\mathbf{s}_1^{i-1}, \mathbf{h}_T^i) & \text{si } i > 1, \end{cases}$$

- Calcul du vecteur de contexte en utilisant les états cachés du décodeur et la sortie de l'encodeur :

$$\mathbf{c}_1 = \text{Attention}(\mathbf{h}_1^L, \dots, \mathbf{h}_T^L; \mathbf{s}_1^L)$$

Le Processus de Décodage (suite)

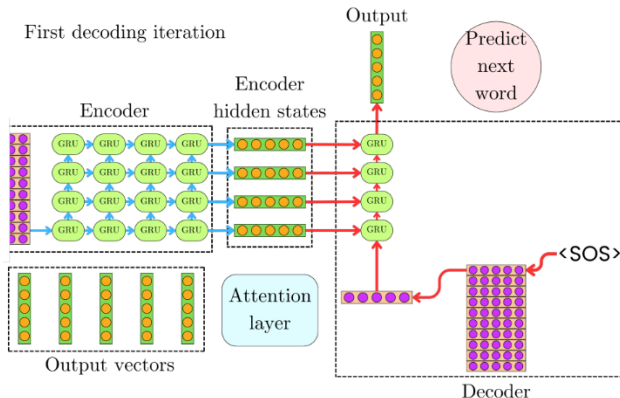


Figure: Lors de la première itération du processus de décodage, la sortie du RNN du décodeur est d'abord calculée à partir des états cachés de l'encodeur et du token **<SOS>**.

Le Processus de Décodage (suite)

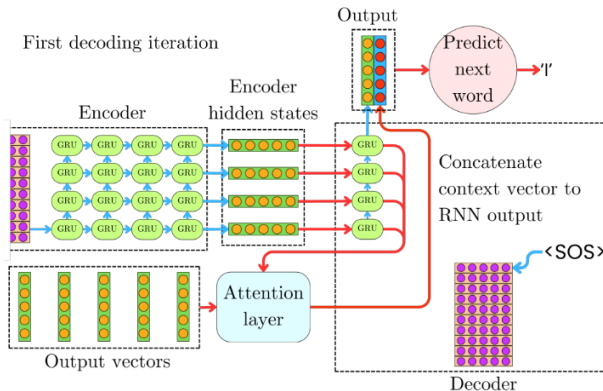


Figure: Le premier vecteur de contexte est calculé à partir des premiers états cachés du décodeur et est concaténé à la sortie du RNN du décodeur comme entrée de la tête de prédiction.

- Concaténation du vecteur de contexte à la sortie du décodeur pour la prédiction :

$$\text{Mot suivant} = \arg \max_{\text{vocab}} W_p \left[\mathbf{s}_1^L, \mathbf{c}_1 \right]$$

- Assure une perte d'information moindre pour prédire le mot suivant.

- Les modèles encodeur-décodeur basés sur RNN avec les attentions de Bahdanau et Luong ont amélioré les tâches de séquence à séquence.
- La focalisation dynamique sur des parties spécifiques de l'entrée a amélioré les performances.
- Les défis liés aux dépendances à long terme et à la parallélisation des calculs subsistaient.
- L'architecture Transformer est apparue, abandonnant la récurrence et utilisant l'auto-attention.