

Réseaux de neurones, et davantage

Module TAL - Master HN PSL

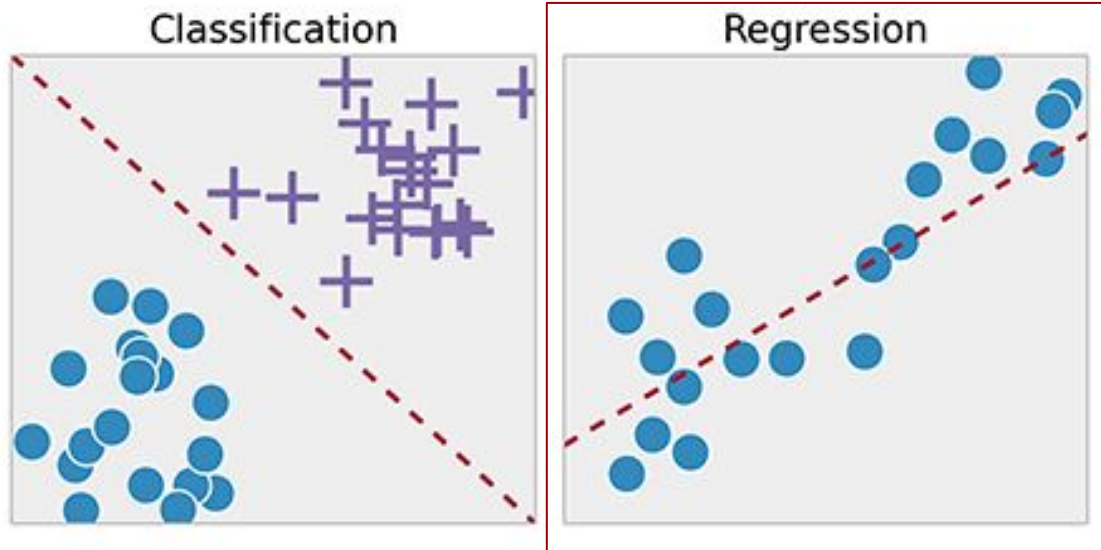
Carmen Brando
carmen.brand@ehess.fr
Ingénieure de recherche en humanités numériques
Ecole des hautes études en sciences sociales
Collaboration avec le **Laboratoire LATTICE**

Paris, 16 mars 2022

Plan (notions clés)

- Régression linéaire
- Réseaux de neurones
- Réseaux de neurones récurrents
- Réseaux à large « mémoire court-terme » (LSTM)
- Transformers
- Modèles pré-entraînés et fine tuning

Régression linéaire



En machine learning, le type de sortie que l'on attend de notre programme peut être une valeur continue (un nombre) ou bien une valeur discrète (une catégorie). Le premier cas est appelé une régression, le second une classification.

Par la suite, nous allons nous intéresser au 2e cas qui est la base des réseaux de neurones.

Régression linéaire

- Le but de la régression linéaire est d'**ajuster (fit) le meilleur modèle linéaire** à un ensemble de données
- A partir de **m échantillons de données** qui sont dans un **espace à n dimensions**. Chacun de ces échantillons a **n caractéristiques** qui correspondent à une seule valeur de sortie.
- Nous avons accès aux données **d'entrée et de sortie**, mais nous voulons **déterminer s'il existe une relation linéaire** qui mappe les données d'entrée sur les données de sortie.

$$h_{\theta}(x) = \theta^T x \approx \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_n x_n$$

θ – the coefficients we are trying to solve for

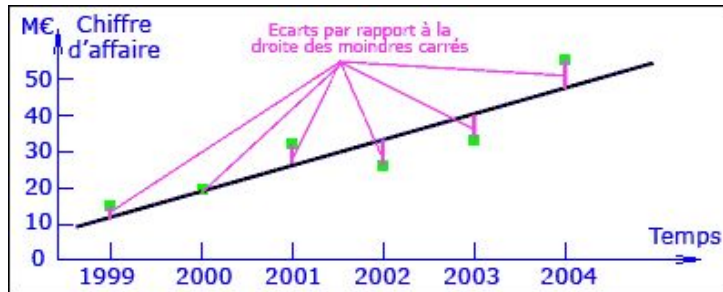
θ_0 – the bias term added to $\theta^T x$

x – the n-dimensional vector of an input sample

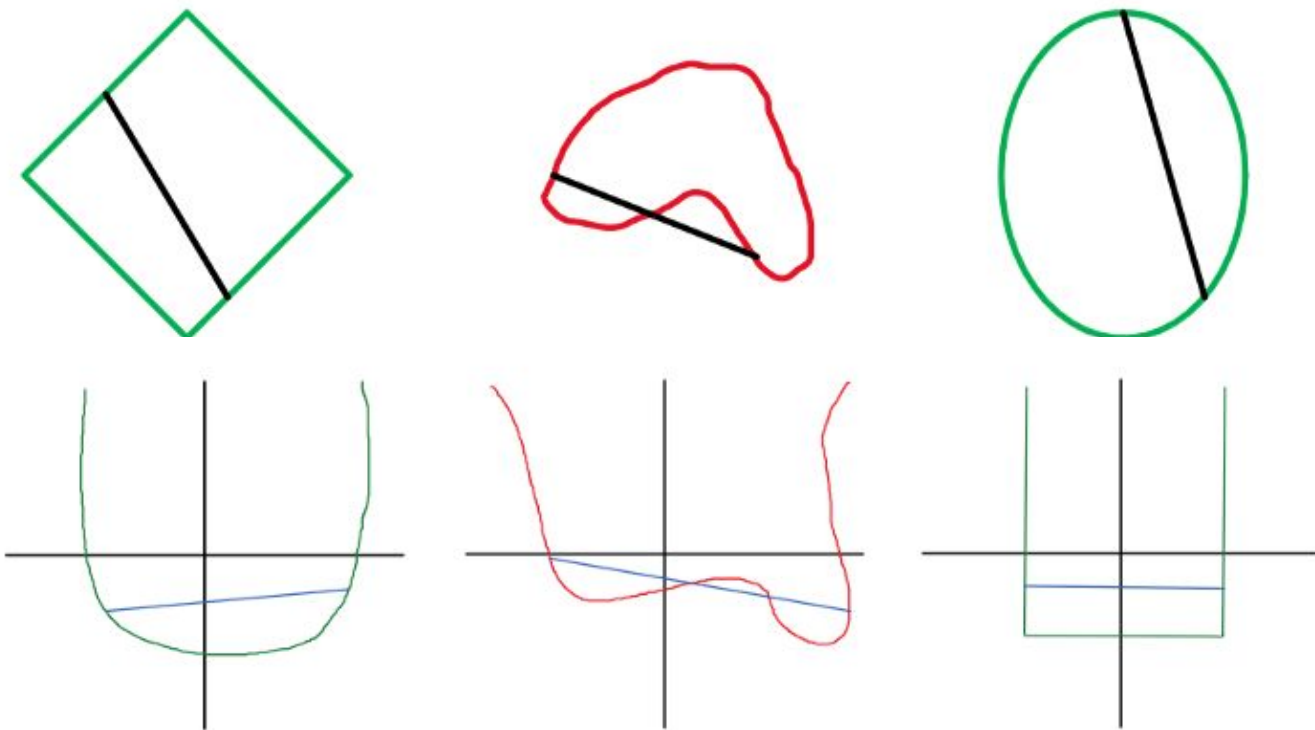
$h_{\theta}(x)$ – the estimated output from an input sample

Régression linéaire (2)

- on détermine le meilleur modèle linéaire par la résolution des **coefficients du modèle qui minimisent l'erreur** entre nos valeurs de sortie estimées et les valeurs de sortie réelles
- Nous pouvons utiliser la **fonction des moindres carrés linéaires** pour y parvenir (voir ci-dessous). Nous appelons cette fonction **une fonction de coût** parce que nous calculons l'erreur totale ou le coût entre notre estimation et la valeur réelle
- Puisque le problème des moindres carrés linéaires est une fonction quadratique, nous pourrions minimiser cette fonction de coût de manière analytique
- Cependant, avec un ensemble de données volumineux, on utilise la méthode itérative connue sous le nom de **descente de gradient pour trouver les meilleurs coefficients**.



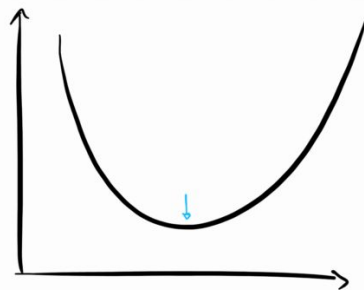
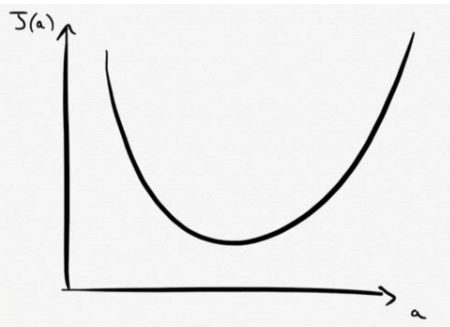
La méthode des moindres carrés consiste à trouver la droite qui minimise la somme des carrés des distances entre chaque point et la droite.



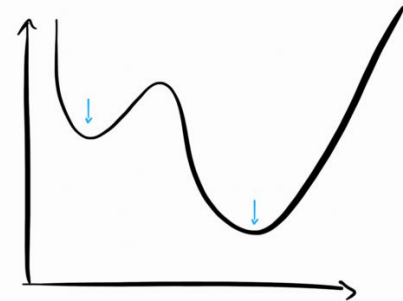
Notion de **Convexité** : En deux dimensions, nous pouvons penser à un ensemble convexe comme une forme où quelle que soit la ligne que vous tracez reliant deux points de l'ensemble, aucune partie de la ligne ne sera en dehors de l'ensemble.

La descente de gradient

- algorithme d'optimisation qui permet de trouver le **minimum** de n'importe quelle **fonction convexe** en convergeant progressivement vers celui-ci
- En Machine Learning, on va utiliser cet algorithme dans les problèmes d'apprentissage supervisé pour **minimiser la fonction coût**, qui justement est une fonction convexe (par exemple l'erreur quadratique moyenne)
- C'est grâce à cet algorithme que la **machine apprend**, c'est-à-dire, trouve le meilleur modèle



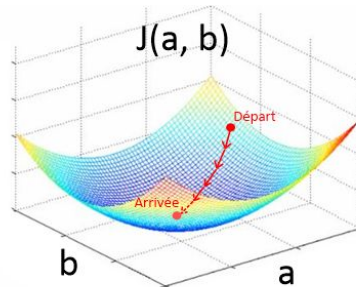
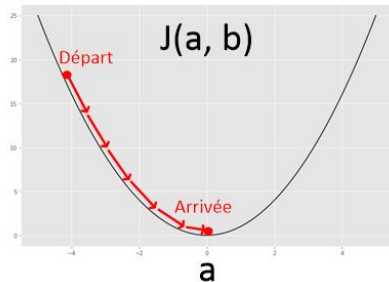
Fonction Convexe



Fonction Non-Convexe

La descente de gradient (algorithme itératif)

1. **Calcul de la dérivée de la Fonction Coût.** Nous partons d'un point initial aléatoire puis nous mesurons la valeur de la pente (**dérivée**) en ce point.
2. **Mise à jour des paramètres du modèle.** On progresse ensuite d'une certaine distance α dans la direction de la pente qui descend. On appelle cette distance taux d'apprentissage, que l'on pourrait traduire par vitesse d'apprentissage. Cette opération a pour résultat de modifier la valeur des paramètres de notre modèle
3. On répète ces deux étapes en boucle.



Par exemple, l'algorithme permet de trouver la valeur idéale pour les paramètres a et b de la fonction coût $J(a, b)$ que l'on a développé pour une régression linéaire

La descente de gradient stochastique et les réseaux de neurones

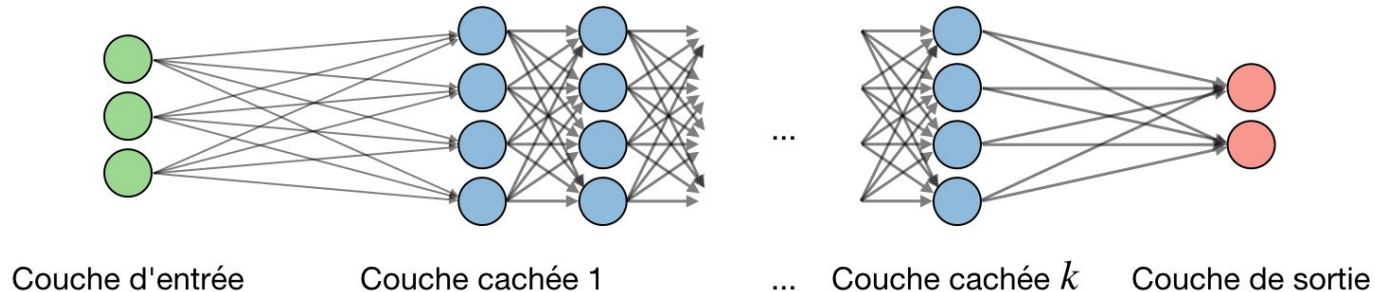
- La minimisation d'une fonction convexe peut se faire avec un **algorithme de descente de gradient** qui permet de converger vers l'unique minimum.
- Compte tenu des courbes d'erreurs très accidentées dessinées par les réseaux de neurones, il existe une **multitude de minima locaux**.
- De ce fait, l'**apprentissage global converge rarement vers le minimum global de la fonction de coût** lorsqu'on applique les algorithmes basés sur le gradient global.
- L'apprentissage **avec gradient stochastique** est une solution permettant de mieux explorer ces courbes d'erreurs et cet algorithme s'utilise notamment pour l'**apprentissage des réseaux de neurones**
- Non convexe, aucune garantie, peut nécessiter un temps excessif, mais, en pratique, cette technique parvient souvent à trouver une bonne solution

Les réseaux de neurones (multi-couches)

- **neurones** = fonctions dont les paramètres sont **ajustés** en fonction de données d'entrée afin de fournir la meilleure réponse possible. Un neurone fait une combinaison linéaire des entrées qu'il reçoit, à laquelle il ajoute une valeur appelée biais
- Une **fonction dite d'activation**, (comme par exemple tangente hyperbolique) est alors appliquée à la valeur de sortie. Cette **valeur est ensuite transmise à la couche de neurone suivante**.
- Chaque **neurone effectue ainsi un calcul très rudimentaire**, et c'est la **succession des couches de neurones** qui permet d'obtenir des réseaux complexes.
- Durant cette phase dite « d'entraînement », le **réseau va ajuster automatiquement les paramètres de chaque neurone**, c'est-à-dire **les valeurs des poids et du biais** afin de **minimiser l'erreur moyenne calculée sur l'ensemble des exemples entre la sortie attendue et celle observée**.

Les réseaux de neurones (multi-couches)

- L'hypothèse est qu'après cette phase d'entraînement, le réseau sera capable de traiter de manière satisfaisante de nouveaux exemples, dont la sortie est inconnue, en fonction de ce qu'il a « appris »
- Dans un réseau de neurones à deux couches, la première couche est constituée d'un ensemble de neurones connectés en parallèle et fournissant un ensemble de sorties, elles-mêmes combinées pour devenir les entrées d'un nouvel ensemble de neurones formant une seconde couche.



Les réseaux récurrents (ou RNN pour Recurrent Neural Networks)

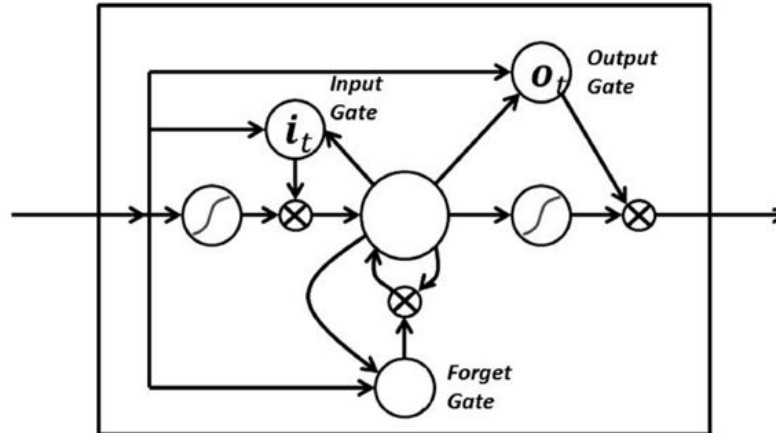
- sont des réseaux de neurones dans lesquels l'information peut se propager **dans les deux sens**, y compris des couches profondes aux premières couches
- Ces réseaux possèdent des connexions récurrentes au sens où **elles conservent des informations en mémoire** (une sorte de **mémoire à court terme**) : ils peuvent prendre en compte à un instant t un certain nombre d'états passés
- les RNNs sont particulièrement adaptés aux applications faisant intervenir le contexte, et plus particulièrement au traitement des séquences temporelles, c'est à dire quand les **données forment une suite et ne sont pas indépendantes les unes des autres**
- En effet, les RNNs « classiques » ne sont capables de mémoriser que le passé dit proche, et commencent à « **oublier** » au bout de plusieurs itérations
- Le principal **inconvénient** : le transfert d'information à double sens rend leur entraînement beaucoup plus compliqué

Inconvénients des RNN

- ne parviennent pas à stocker les informations pendant **une période plus longue**. Parfois, une référence à certaines informations stockées il y a assez longtemps est nécessaire pour prédire la sortie actuelle. Les RNN sont absolument incapables de gérer de telles **dépendances à long terme**
- il n'y a **pas de contrôle** plus fin sur quelle partie du contexte doit être reportée et quelle part du passé doit être « oubliée »
- D'autres problèmes avec les RNN sont l'**explosion et la disparition des gradients** qui se produisent pendant le processus de formation d'un réseau par le biais du retour en arrière
 - Problème posé par l'augmentation très rapide des valeurs des gradients pendant la descente entraînant un dépassement de la capacité de la représentation interne des nombres et l'arrêt de l'apprentissage.
- ainsi, les réseaux à large « mémoire court-terme » (LSTM) ont été introduits.

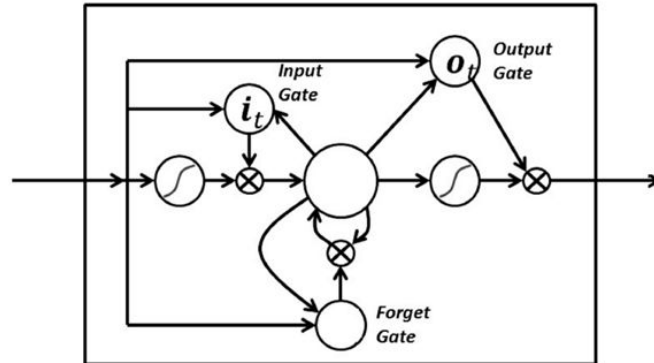
Les réseaux à large mémoire court-terme (LSTM)

- extension pour les RNN, qui étend leur mémoire, conçus pour **modéliser les séquences chronologiques et leurs dépendances à longue portée** plus précisément que les RNN
- stockent leurs informations dans une **mémoire**, ce qui ressemble beaucoup à la mémoire d'un ordinateur parce que le LSTM peut **lire, écrire et supprimer des informations** de sa mémoire



Les réseaux à large mémoire court-terme (2)

- cette mémoire peut être vue comme une **cellule *gated***, où *gated* signifie que la cellule décide de stocker ou de supprimer des informations en fonction de **l'importance qu'elle attribue à l'information**
- l'attribution de l'importance se fait à travers **des poids, qui sont également appris par l'algorithme**. Cela signifie simplement qu'il **apprend avec le temps quelle information est importante et laquelle ne l'est pas**
- les problèmes posés par la disparition des gradients sont résolus grâce à LSTM, car les gradients sont assez raides et **l'entraînement est relativement court et la précision élevée**.

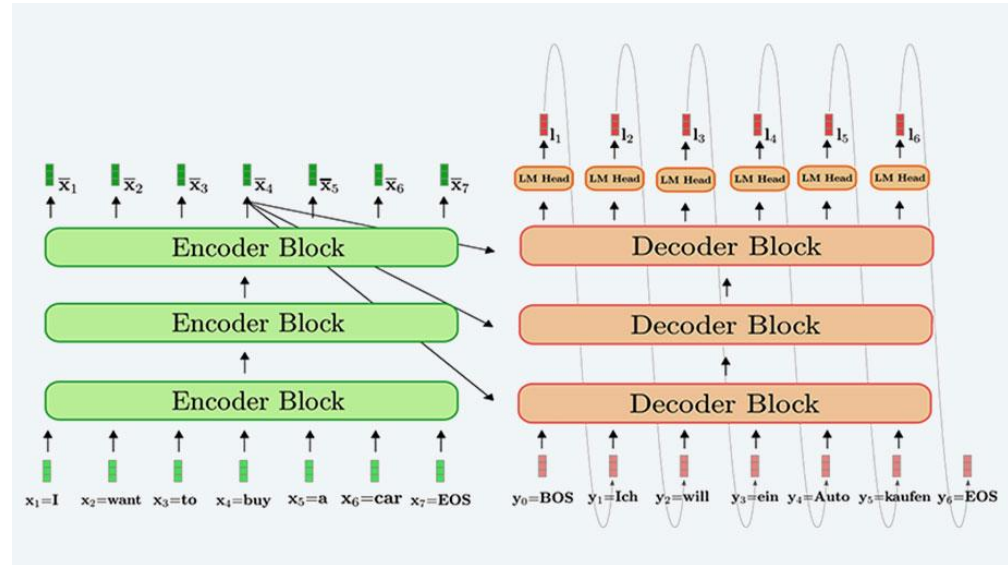


Transformers

- un réseau de neurones de type **séquence à séquence** (seq2seq) comme les RNN dont les LSTM vus précédemment
- Il se différencie par le fait de n'utiliser que **le mécanisme d'attention** et aucun réseau récurrent
- Contrairement aux RNN, les transformers **n'exigent pas que les données séquentielles soient traitées dans l'ordre.**
- c'est grâce à cette fonctionnalité, que le transformer permet une **parallélisation** des calculs
- Les modèles sans contexte tels que **word2vec** ou **GloVe** génèrent une unique représentation vectorielle pour chaque mot du vocabulaire. Les **modèles contextuels** génèrent au contraire une représentation de chaque mot qui se base sur les autres mots de la phrase, ce qui rend ces **modèles profondément bidirectionnels.**

Transformers : l'architecture encoder-decoder

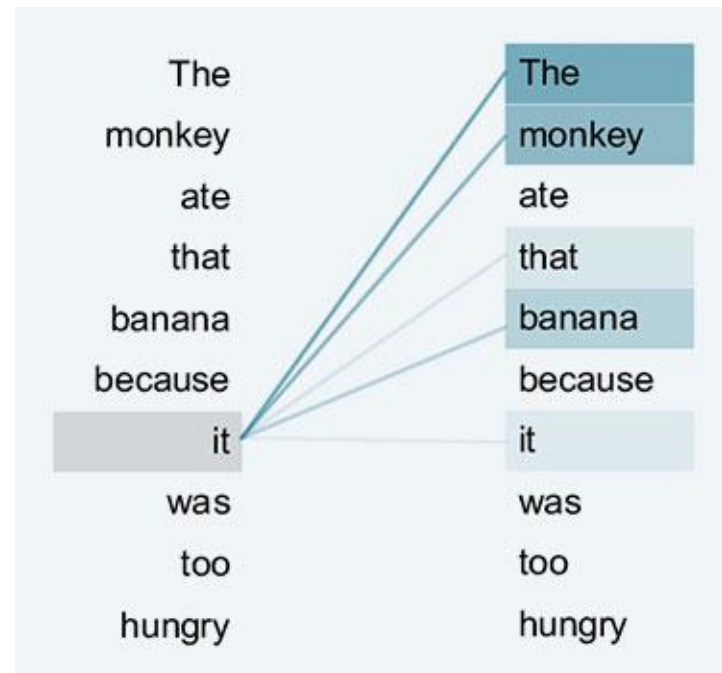
- Les modèles séquence à séquence utilisent une **architecture Encoder-Decoder**
- l'encodeur crée une **représentation vectorielle (embedding)** d'une **séquence d'entrée**
- le décodeur retourne une séquence à partir de l'embedding.



(Von Platen)

Transformers : mécanisme d'attention

- L'idée du transformer est de **conserver l'interdépendance des mots** (tokens) d'une séquence en utilisant le mécanisme d'attention qui est au centre de son architecture
- Ce concept d'attention **mesure le lien entre deux éléments de deux séquences**. Ainsi, le mécanisme d'attention permet de transmettre l'information au modèle, afin qu'il **porte son attention au bon endroit** sur les mots de la séquence A, quand on traite un mot de la séquence B
- La figure ci-dessous, montre que pour le mot « it », le **coefficient d'attention** est élevé pour « The monkey ». Ce qui veut dire que le modèle devra s'appuyer sur « The monkey » pour encoder « it ».



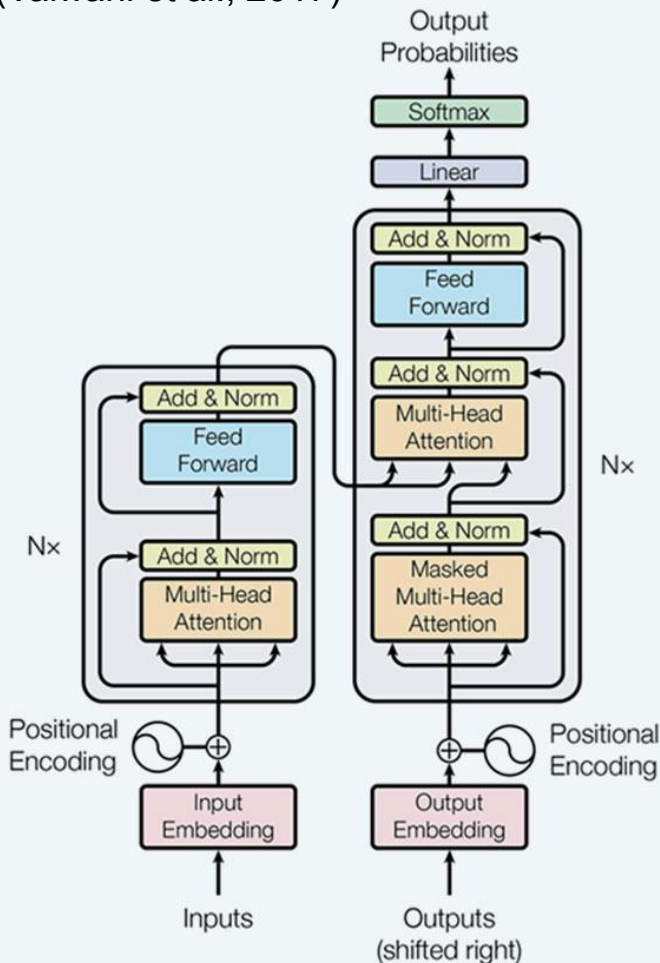
(Xie et al. 2020).

Transformers :

l'architecture globale

- Les couches sont constituées de **Multi-Head Attention** afin que chaque élément de la séquence d'entrée ait un coefficient d'attention
- La **couche Feed Forward** est un réseau de neurones multicouches qui traite la sortie d'une couche d'attention de manière à mieux adapter l'entrée de la couche d'attention suivante.
- L'encodeur est donc appliqué N fois
- Dans le **décodeur**, le « Masked-Multi-Head Attention » permet de limiter l'accès d'un token de la séquence aux tokens suivants. Ainsi, ce masque évite d'incorporer les informations des tokens qui seront décodés plus tard
- Le **positional encoding** ajoute au token l'information de sa position au sein de la séquence. En effet, lors de la traduction d'une phrase, la position des mots dans la phrase est importante.

(Valwani et al., 2017)



Modèles de langue pré-entraînés et fine-tuning

- L'un des plus grands défis en TAL est le **manque de données d'entraînement** et le nombreuses tâches distinctes, la plupart des ensembles de données spécifiques aux tâches ne contiennent que quelques milliers ou quelques centaines de milliers d'**exemples d'entraînement** étiquetés par des humains.
- Pour aider à combler ce manque de données, il est possible d'entraîner des modèles de langue (pré-entraînés) en utilisant l'énorme quantité de texte non annoté sur le web (pré-entraînement)

Modèles de langue pré-entraînés et fine-tuning (2)

- les modèles “transformers” (BERT, GPT), une fois **pré-entraîné**, de façon **non supervisée** (sur des données textuelles non labellisées comme le corpus anglophone de Wikipédia), il possède une "représentation" linguistique qui lui est propre
 - cela nécessite des gigas octets de données textuelles, une infrastructure GPU, ce qui représente un coût important.
- Il est ensuite possible, sur la base de cette représentation initiale, de le **customiser (fine-tuning) pour une tâche particulière**
- Il peut être entraîné en mode incrémental pour **spécialiser le modèle rapidement et avec peu de données**, ce qui permet d'améliorer considérablement la précision par rapport à l'entraînement à partir de zéro.

Transformers : exemples

- BERT (Bidirectional Encoder Representations from Transformers) introduit en 2019 par Google
- GPT-3 (Generative Pre-trained Transformer)
- RoBERTa, version de BERT pour laquelle certains hyperparamètres du pré-entraînement ont été modifiés, capable d'améliorer l'objectif de modélisation du langage masqué par rapport à BERT
- Malgré leurs succès, la plupart des modèles disponibles ont été formés soit sur des corpus textuels en langue anglaise, soit sur la concaténation de données dans plusieurs langues. Cela rend l'utilisation pratique de tels modèles limitée en France par exemple, pour cela, CamemBERT et FlauBERT ont vu le jour.



Hugging Face

Search models, datasets, users...

Models

Datasets

Spaces

Docs

Solutions

Pricing



Log In

Sign Up

Jean-Baptiste / **camembert-ner**

like 9



Token Classification



PyTorch



Transformers

Jean-Baptiste/wikiner_fr

fr

camembert



AutoNLP Compatible



Model card



Files and versions



Train



Deploy



Use in Transformers

camembert-ner: model fine-tuned from camemBERT for NER task.

Introduction

[camembert-ner] is a NER model that was fine-tuned from camemBERT on wikiner-fr dataset. Model was trained on wikiner-fr dataset (~170 634 sentences). Model was validated on emails/chat data and overperformed other models on this type of data specifically. In particular the model seems to work better on entity that don't start with an upper case.

Training data

Training data was classified as follow:

Downloads last month
20,379



Hosted inference API



Token Classification

Examples



Je m'appelle jean-baptiste et je vis à montréal

Compute

Computation time on cpu: cached

Je m'appelle jean-baptiste **PER** et je vis à montréal **LOC**

JSON Output

Maximize

Accès simplifié via HuggingFace

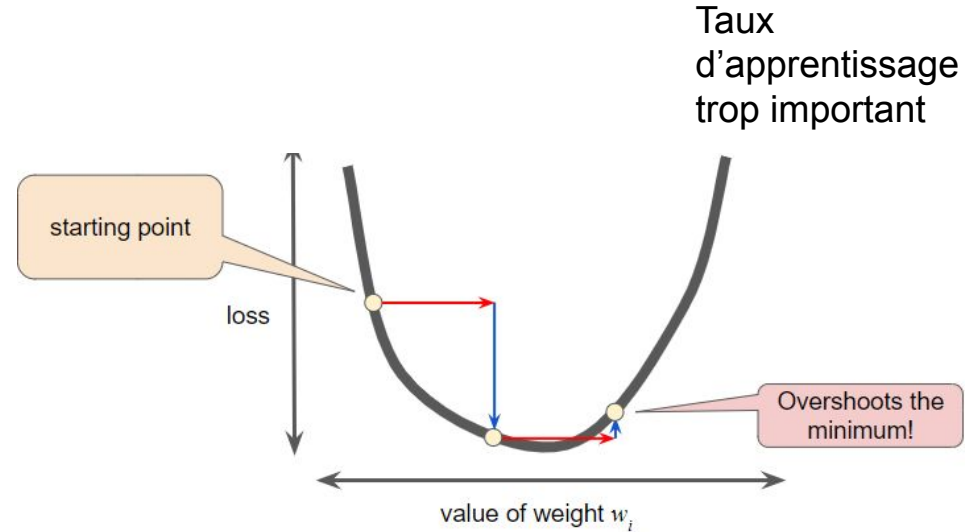
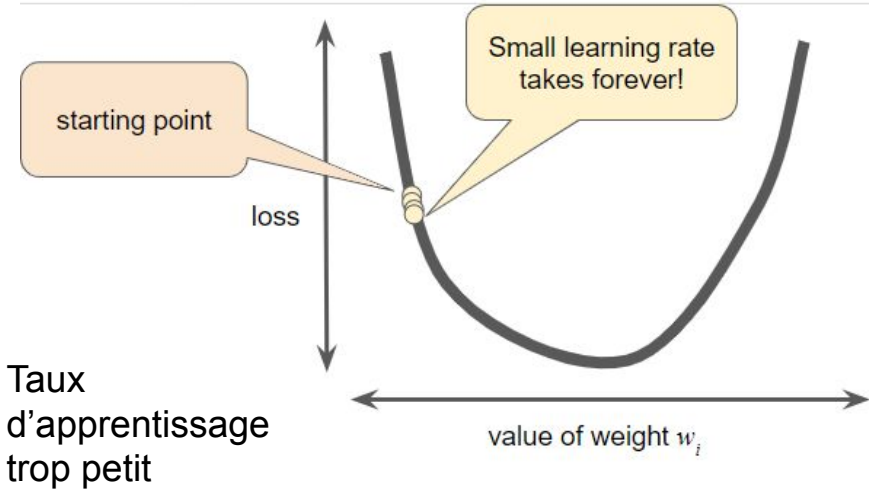
TD : utiliser des modèles transformers pré-entraînés disponibles sur HuggingFace

https://github.com/cvbrandoe/coursTAL/blob/master/2022/Utiliser_des_transformers_via_HuggingFace.ipynb

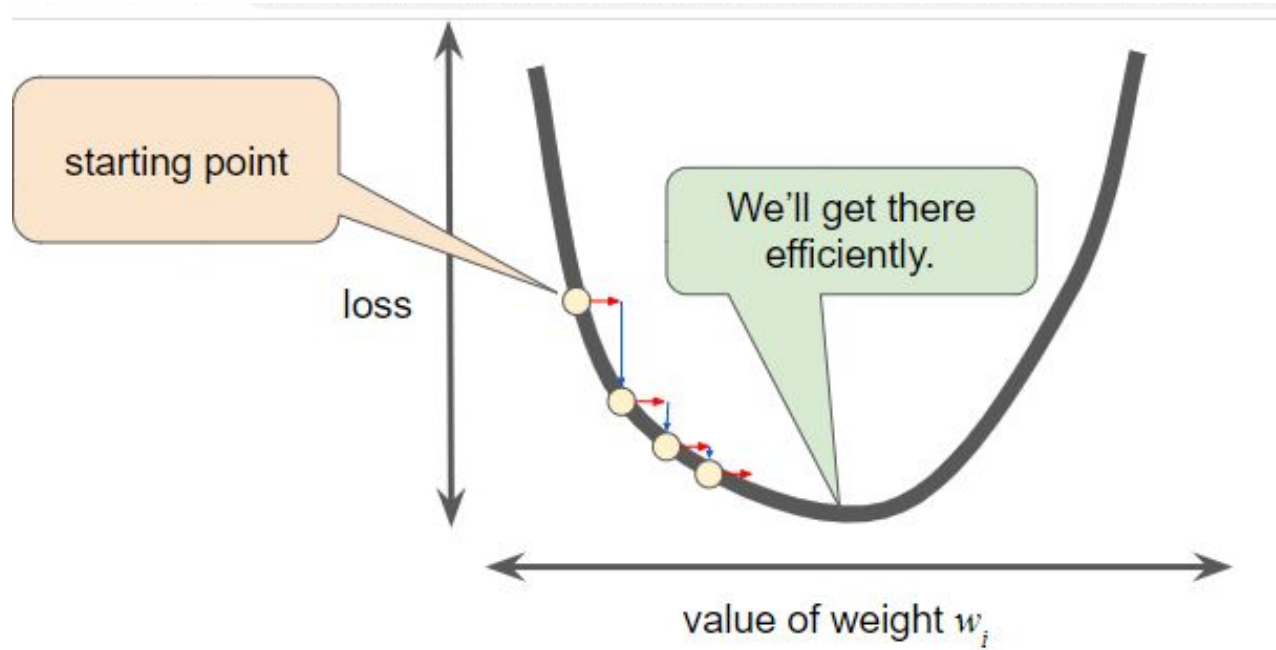
Hyperparamètres lors de l'apprentissage

- Le **taux d'apprentissage** (appelé en anglais learning rate) indique la vitesse à laquelle les coefficients évoluent. Cette quantité peut être fixe ou variable. L'une des méthodes les plus populaires à l'heure actuelle s'appelle **Adam**, qui a un taux d'apprentissage qui s'adapte au fil du temps
- **LOSS**, une mesure de la distance entre les prédictions d'un modèle et son étiquette. Ou, pour le formuler de manière plus pessimiste, une mesure de la qualité du modèle
- **Dropout**, technique qui est destinée à empêcher le sur-ajustement sur les données de training en abandonnant des unités dans un réseau de neurones

hyperparamètres (taux d'apprentissage et loss)



hyperparamètres (taux d'apprentissage et loss)



Sur-ajustement (overfitting) d'un modèle

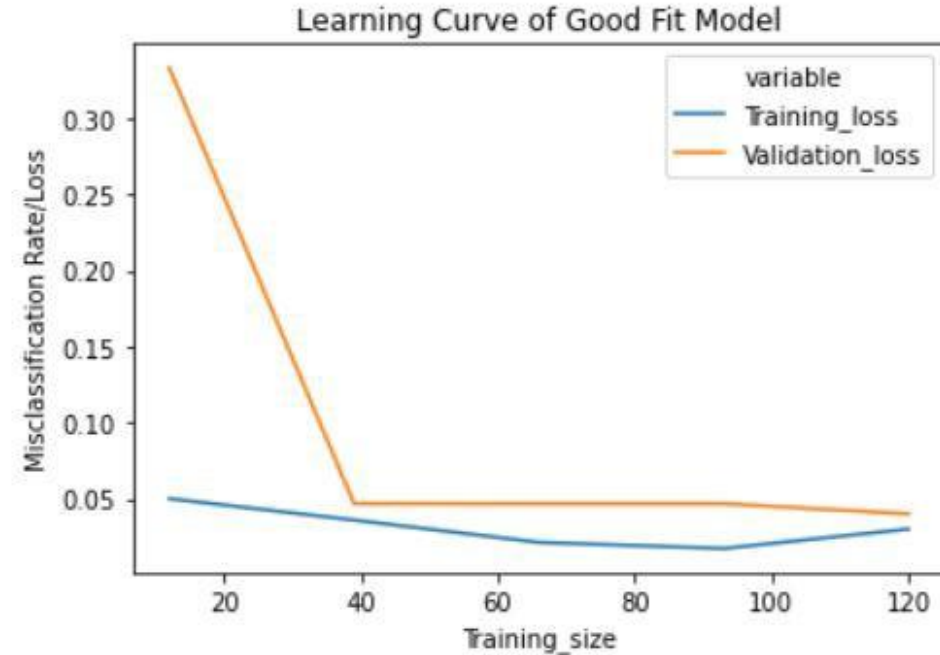
- On dit qu'un modèle est surajusté s'il est surentraîné sur les données de telle sorte qu'il en apprend même le bruit
- Un modèle surajusté apprend chaque exemple si parfaitement qu'il classe mal un exemple nouveau ou non vu
- Pour un modèle surajusté, nous avons **une excellente précision (accuracy) lors de la phase d'apprentissage et faible lors de la phase de validation**
- Causes :
- Utilisation d'un modèle complexe pour un problème simple qui capte le bruit des données
- Petits ensembles de données, car l'ensemble d'apprentissage peut ne pas être une bonne représentation de l'univers

Sous-ajustement (underfitting) d'un modèle

- On dit d'un modèle qu'il est sous-ajusté s'il est incapable d'apprendre correctement les modèles des données.
- Un modèle sous-ajusté n'apprend pas complètement chaque exemple de l'ensemble de données.
- Dans de tels cas, nous constatons **un faible score à la fois lors de l'apprentissage et de la validation**
- Causes :
- L'utilisation d'un modèle simple pour un problème complexe qui n'apprend pas tous les modèles dans les données.
- Les données sous-jacentes ne présentent aucun modèle inhérent

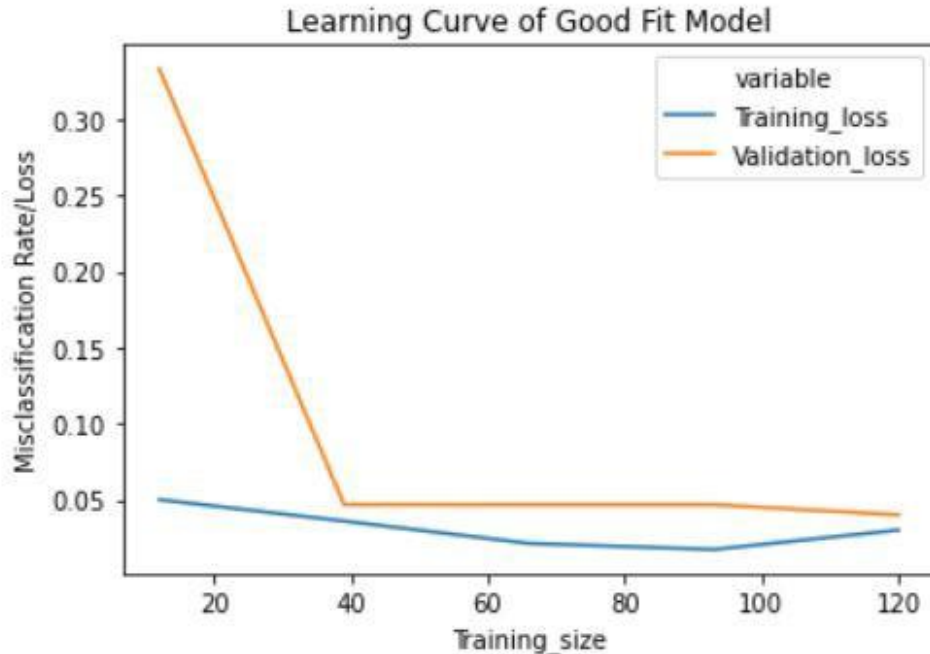
Courbes d'apprentissage

- Les courbes d'apprentissage représentent la perte (LOSS) d'entraînement et de validation d'un échantillon d'exemples d'entraînement **par l'ajout progressif de nouveaux exemples**
- Les courbes d'apprentissage nous aident à **déterminer si l'ajout d'exemples d'apprentissage supplémentaires améliorerait le score de validation** (score sur les données non observées).

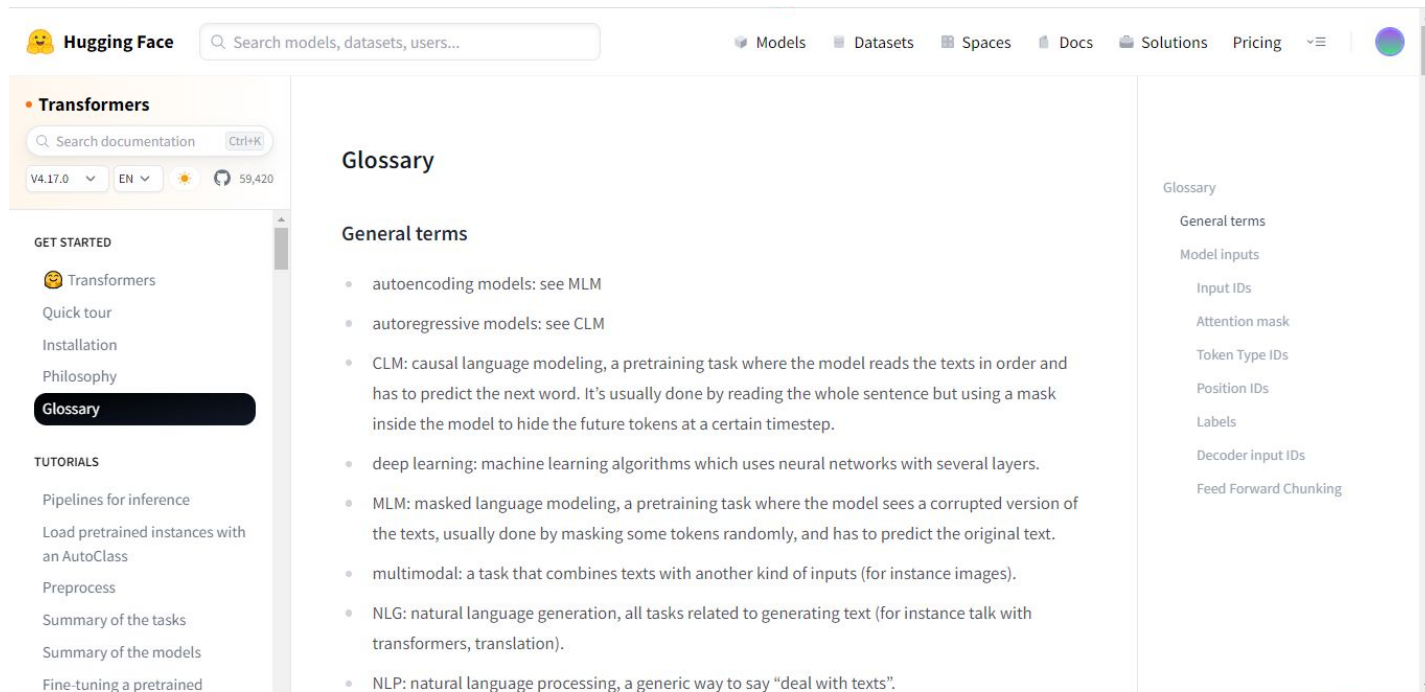


Courbes d'apprentissage

- training loss : la courbe d'apprentissage d'un modèle bien ajusté présente une perte d'apprentissage modérément élevée au début, qui diminue progressivement en ajoutant des exemples d'apprentissage et s'aplatit graduellement, ce qui indique que **l'ajout d'exemples d'apprentissage supplémentaires n'améliore pas les performances du modèle sur les données d'apprentissage**
- validation loss : La courbe d'apprentissage d'un modèle bien ajusté présente une perte de validation élevée au début, qui diminue progressivement lors de l'ajout d'exemples d'entraînement et s'aplatit graduellement, ce qui indique que **l'ajout d'exemples d'entraînement supplémentaires n'améliore pas les performances du modèle sur les données non observées**.
- Nous pouvons également constater qu'en ajoutant un nombre raisonnable d'exemples d'apprentissage, **les pertes d'apprentissage et de validation se rapprochent l'une de l'autre, la perte de validation étant légèrement supérieure à la perte d'apprentissage**



<https://huggingface.co/docs/transformers/glossary>



The screenshot shows the Hugging Face website's Glossary page. The top navigation bar includes the Hugging Face logo, a search bar, and links for Models, Datasets, Spaces, Docs, Solutions, and Pricing. The left sidebar is divided into 'Transformers' (with a search bar and version/language filters) and 'TUTORIALS' (listing various guides). The main content area is titled 'Glossary' and 'General terms', featuring a bulleted list of definitions for various NLP concepts. A right sidebar provides a table of contents for the Glossary section.

Hugging Face Search models, datasets, users...

Transformers

Search documentation Ctrl+K

V4.17.0 EN 59,420

GET STARTED

- Transformers
- Quick tour
- Installation
- Philosophy
- Glossary**

TUTORIALS

- Pipelines for inference
- Load pretrained instances with an AutoClass
- Preprocess
- Summary of the tasks
- Summary of the models
- Fine-tuning a pretrained

Glossary

General terms

- autoencoding models: see MLM
- autoregressive models: see CLM
- CLM: causal language modeling, a pretraining task where the model reads the texts in order and has to predict the next word. It's usually done by reading the whole sentence but using a mask inside the model to hide the future tokens at a certain timestep.
- deep learning: machine learning algorithms which uses neural networks with several layers.
- MLM: masked language modeling, a pretraining task where the model sees a corrupted version of the texts, usually done by masking some tokens randomly, and has to predict the original text.
- multimodal: a task that combines texts with another kind of inputs (for instance images).
- NLG: natural language generation, all tasks related to generating text (for instance talk with transformers, translation).
- NLP: natural language processing, a generic way to say "deal with texts".

Glossary

- General terms
- Model inputs
 - Input IDs
 - Attention mask
 - Token Type IDs
 - Position IDs
 - Labels
- Decoder input IDs
- Feed Forward Chunking

TD : fine-tuner un modèle pré-entraîné avec un jeu de données utilisateur pour un tâche de classification de séquences de mots

https://github.com/cvbrandoe/coursTAL/blob/master/2022/HuggingFace_finetune_custom_datasets_Sequence_classification.ipynb

TD : fine-tuner un modèle pré-entraîné avec un jeu de données utilisateur pour un tâche de classification de mots

https://github.com/cvbrandoe/coursTAL/blob/master/2022/FineTuningBERT_NER_UN.ipynb

The main use case for pretrained transformer models is transfer learning. You load in a large generic model pretrained on lots of text, and start training on your smaller dataset with labels specific to your problem. The spacy-transformers package has custom pipeline components that make this especially easy. We provide an example component for the development of analogous components for other tasks should be straightforward.

<https://hackernoon.com/train-a-ner-transformer-model-with-just-a-few-lines-of-code-via->

Paramètres du notebook

Accélérateur matériel

GPU

Pour tirer le meilleur parti de Colab, évitez d'utiliser un GPU si vous n'en avez pas besoin. [En savoir plus](#)

☐ Exécution en arrière-plan

Vous souhaitez que votre notebook continue de fonctionner même après que vous avez fermé votre navigateur ? [Passer à Colab Pro+](#)

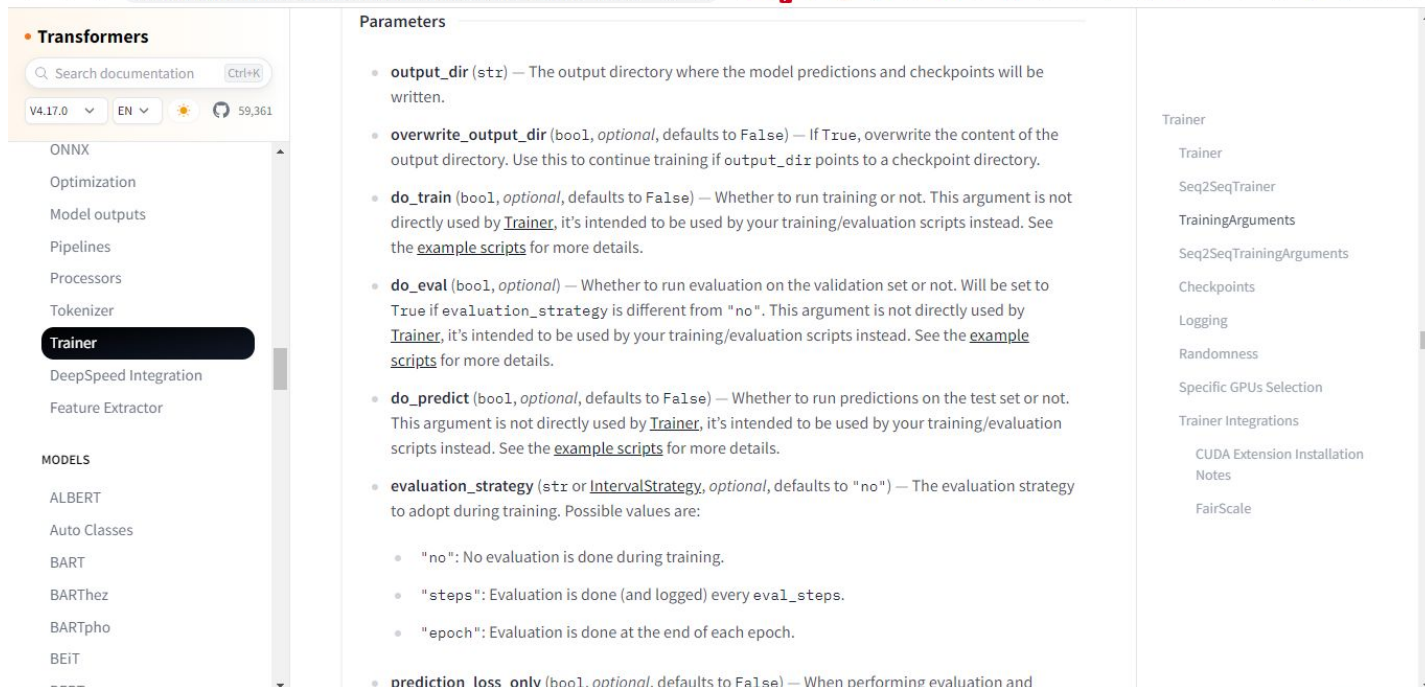
☐ Omettre l'élément de sortie des cellules de code lors de l'enregistrement de ce notebook

Annuler [Enregistrer](#)

HuggingFace : interface Trainer (arguments clés)

```
TrainingArguments(  
    output_dir="./results",  
    evaluation_strategy="epoch",  
    learning_rate=2e-5,  
    per_device_train_batch_size=16,  
    per_device_eval_batch_size=16,  
    num_train_epochs=3,  
    weight_decay=0.01,
```

HuggingFace : interface Trainer (arguments clés)



The screenshot displays the HuggingFace Transformers documentation interface. On the left, a sidebar under the 'Transformers' header includes a search bar, version (V4.17.0), language (EN), and a list of categories. The 'Trainer' category is highlighted. Below it, a 'MODELS' section lists various models like ALBERT, Auto Classes, BART, BARThez, BARTpho, BEiT, and more. The main content area is titled 'Parameters' and lists key arguments for the Trainer class:

- output_dir** (`str`) — The output directory where the model predictions and checkpoints will be written.
- overwrite_output_dir** (`bool`, *optional*, defaults to `False`) — If `True`, overwrite the content of the output directory. Use this to continue training if `output_dir` points to a checkpoint directory.
- do_train** (`bool`, *optional*, defaults to `False`) — Whether to run training or not. This argument is not directly used by `Trainer`, it's intended to be used by your training/evaluation scripts instead. See the [example scripts](#) for more details.
- do_eval** (`bool`, *optional*) — Whether to run evaluation on the validation set or not. Will be set to `True` if `evaluation_strategy` is different from "no". This argument is not directly used by `Trainer`, it's intended to be used by your training/evaluation scripts instead. See the [example scripts](#) for more details.
- do_predict** (`bool`, *optional*, defaults to `False`) — Whether to run predictions on the test set or not. This argument is not directly used by `Trainer`, it's intended to be used by your training/evaluation scripts instead. See the [example scripts](#) for more details.
- evaluation_strategy** (`str` or `IntervalStrategy`, *optional*, defaults to "no") — The evaluation strategy to adopt during training. Possible values are:
 - "no": No evaluation is done during training.
 - "steps": Evaluation is done (and logged) every `eval_steps`.
 - "epoch": Evaluation is done at the end of each epoch.
- prediction_loss_only** (`bool`, *optional*, defaults to `False`) — When performing evaluation and

On the right side of the page, a 'Trainer' section lists related classes and modules: `Trainer`, `Seq2SeqTrainer`, `TrainingArguments`, `Seq2SeqTrainingArguments`, `Checkpoints`, `Logging`, `Randomness`, `Specific GPUs Selection`, `Trainer Integrations`, `CUDA Extension Installation Notes`, and `FairScale`.

Welcome

Create a new model repo
From the website

Model hub documentation
How to add metadata to your model

Transformers documentation
About model sharing & uploading

Getting started with our `git` and `git-lfs` interface

If you need to create a model repo from the command line (skip if you created a repo from the website)

```
$ pip install huggingface_hub
# Or use transformers-cli if you have transformers

$ huggingface-cli login
# Log in using the same credentials as huggingface.co/join
# Create a model repo from the CLI if needed
$ huggingface-cli repo create model_name
```

Clone your model locally

```
# Make sure you have git-lfs installed
# (https://git-lfs.github.com)
$ git lfs install
$ git clone https://huggingface.co/username/model_name
```

Clone your model locally

```
# Make sure you have git-lfs installed  
# (https://git-lfs.github.com)  
$ git lfs install  
$ git clone https://huggingface.co/username/model_name
```

Then add, commit and push weights, tokenizer and config

```
# save files via '.save_pretrained()' or move them here  
$ git add .  
$ git commit -m "commit from $USER"  
$ git push
```

Your model will then be accessible through its identifier: `username/model_name`

Anyone can load it from code:

```
tokenizer = AutoTokenizer.from_pretrained("username/model_name")  
model = AutoModel.from_pretrained("username/model_name")
```

