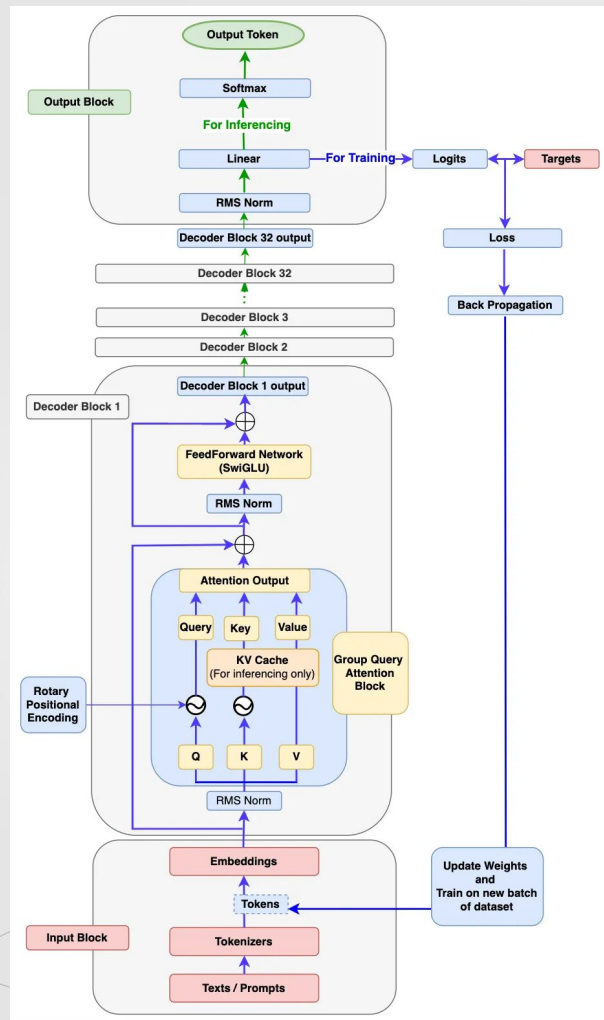


The background features a complex network of thin grey lines and dots, forming a web-like structure on the left side. Scattered across the entire background are various triangles of different sizes and orientations, some with solid outlines and others with dashed or dotted outlines. The overall aesthetic is minimalist and technical.

Adapter les LLMs

CEPE Octobre 2024
Alexandre Tuel
atuel@galeio.fr



**LES DIFFÉRENTES
MÉTHODES
D'ADAPTATION**

1

**RETRIEVAL
AUGMENTED
GENERATION**

2

SUPERVISED FINE-TUNING

3

STRUCTURE DU COURS

4

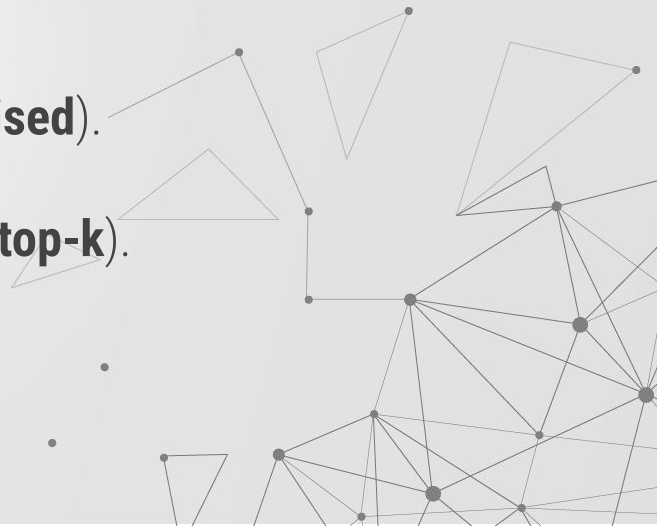
RLHF

5

ANNEXES

Récapitulatifs des éléments importants

- **Tâches prétextes** qui déterminent différents modèles. Next-token prediction/masked language modeling.
- La notion **d'embedding** d'un token/mot/phrased : **uni/bi directionnel**.
- LLM et **foundation model**.
- LLM et entraînement **auto-supervisé (self-supervised)**.
- Processus de génération (**sampling, température, top-k**).
- **Tokénisation**.



Récapitulatifs des éléments importants

- **Context window** et coût quadratique.
- **Zero-shot/few-shot learning**
- CPU/GPU
- **Quantization**
- Les **LLMs génératifs** ne sont pas la solution à tout.
- Il y a de nombreux LLMs différents.



Objectifs d'aujourd'hui

- Mieux comprendre les différentes catégories de LLMs à travers des cas pratiques.
- Maîtriser le Retrieval Augmented Generation de A à Z.



1

Les différentes méthodes d'adaptation



Adapter les LLMs

- **Foundation Model** = possibilité d'utiliser modèles existants pour des tâches sur lesquelles ils n'ont pas été entraînés.
- Mais on veut aller **plus loin que le zero-shot learning/few-shot learning**. Il existe de nombreuses approches (complémentaires) :
 - Retrieval Augmented Generation RAG.
 - Chain-of-thought reasoning
 - Supervised fine-tuning.
 - Unsupervised fine-tuning : très proche du pretraining mais prudemment.
 - RLHF : Reinforcement Learning Human Feedback.
 - Et de nombreuses autres méthodes...



Adapter les LLMs

Mais avant de faire beaucoup d'efforts, il faut aussi **savoir choisir parmi tous les modèles déjà disponibles**. Par exemple, **pour du Q&A**, vaut-il mieux **faire du fine-tuning d'un decoder-only ou utiliser un encoder-decoder ?**

- BERT ? RoBERTA?
- BART ?
- GPT-3 ou Mixtral ?
- Version quantized existe-t-elle déjà ?
- etc..



Retrieval Augmented Generation

- **RAG = Retrieval Augmented Generation.**
- C'est une façon d'aller **chercher automatiquement du texte** qui est **similaire à un prompt** afin d'enrichir (**augmented**) ce prompt.
- C'est une première façon d'**adapter un LLM** à une **tâche spécifique** avec plusieurs cas d'usages
 - Faire du prompt-engineering de façon automatique
 - Chercher des informations que le LLM ne connaît pas
 - Chercher de la donnée actualisée
 - Rendre LLM plus robuste aux hallucinations
 - Raisonnement plus transparent



Retrieval Augmented Generation

- En particulier, pour un RAG, il n'y a pas besoin d'entraîner/fine-tuner un réseau de neurones.
- C'est de toute façon la **première étape** dans l'adaptation d'un LLM à un cas d'usage particulier.



Supervised Fine-Tuning

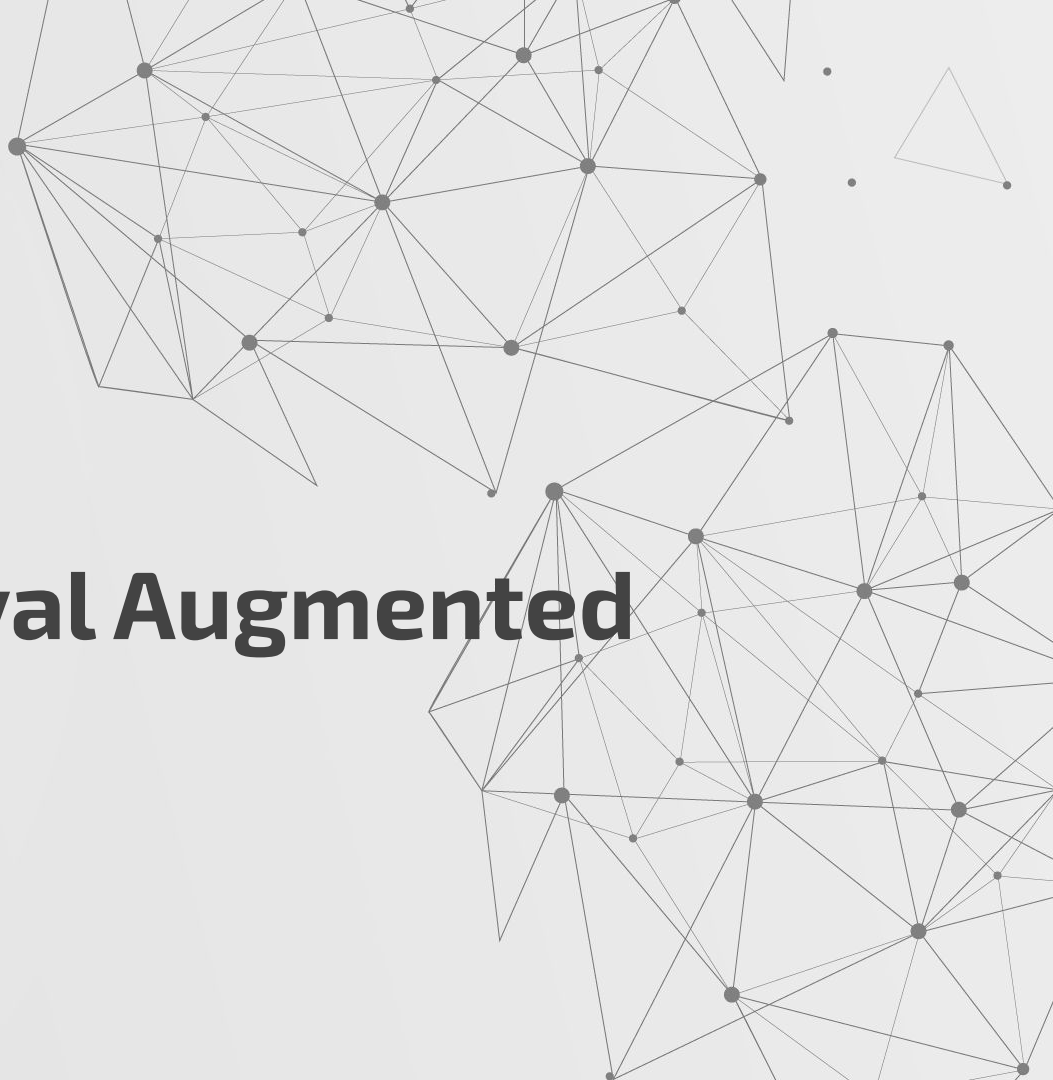
```
training_params = TrainingArguments(  
    output_dir="./results",  
    num_train_epochs=1,  
    per_device_train_batch_size=4,  
    gradient_accumulation_steps=1,  
    optim="paged_adamw_32bit",  
    save_steps=25,  
    logging_steps=25,  
    learning_rate=2e-4,  
    weight_decay=0.001,  
    fp16=False,  
    bf16=False,  
    max_grad_norm=0.3,  
    max_steps=-1,  
    warmup_ratio=0.03,  
    group_by_length=True,  
    lr_scheduler_type="constant",  
    report_to="tensorboard"  
)
```

- Beaucoup de détails sont complètement cachés.
- En revanche il y a de nombreux paramètres à comprendre.
- Il faut donc expliquer comment fonctionne l'entraînement d'un réseau de neurones en général.



2

Pratique : Retrieval Augmented Generation



Code pour RAG assez compact

```
DOC_PATH = "test_data/arxiv_example.pdf"
CHROMA_PATH = "database_RAG/db_arxiv_example"

loader = PyPDFLoader(DOC_PATH)
pages = loader.load()

text_splitter = RecursiveCharacterTextSplitter(chunk_size=500, chunk_overlap=50)
chunks = text_splitter.split_documents(pages)

embeddings = OpenAIEmbeddings(openai_api_key=OPENAI_API_KEY)
db_chroma = Chroma.from_documents(chunks, embeddings, persist_directory=CHROMA_PATH)

query = 'Does this article has many authors ? Does this article talk about climate change ?'

docs_chroma = db_chroma.similarity_search_with_score(query, k=5)

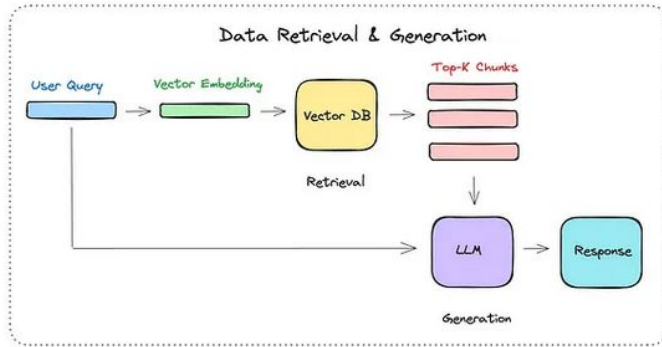
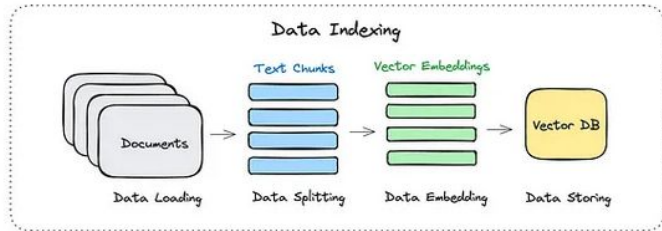
context_text = "\n\n".join([doc.page_content for doc, _score in docs_chroma])

PROMPT_TEMPLATE = """
Answer the question based only on the following context:
{context}
Answer the question based on the above context: {question}.
"""

prompt_template = ChatPromptTemplate.from_template(PROMPT_TEMPLATE)
prompt = prompt_template.format(context=context_text, question=query)
```

Les étapes du RAG

Basic RAG Pipeline



1. Préparer une base de documents.
2. Indexer cette base de documents dans une **base vectorielle** en associant un embedding à chaque text chunks.
3. Trouver les chunks similaires à la question (Pour Q&A LLM).
4. Intégrer ces chunks dans le contexte de la question.

Remarques générales

- Pas de difficulté structurelle : un bon RAG c'est avant tout une bonne exécution.
- Indexer cette base de documents dans une **base vectorielle** en associant un embedding à chaque chunk de texte.
- Trouver les chunks similaires à la question (Pour Q&A LLM).
- Intégrer ces chunks dans le contexte de la question.





2.1

Préparer un corpus de PDF

Différents packages pour extraire du texte



- De nombreuses méthodes d'**OCR : Optical Character Recognition**.
- **NOUGAT** : en local mais pas usage commercial. Meta.
- **MARKER** : en local mais pas usage commercial.
- **Mathpix** : en API
- Et bien d'autres



Extraire des images de PDFs

```
DOC_PATH = "test_data/arxiv_example.pdf"

doc = fitz.open(DOC_PATH)
image_count = 0
image_dir = 'extracted_images'

if not os.path.exists(image_dir):
    os.makedirs(image_dir)

for i in range(len(doc)):
    for img in doc.get_page_images(i):
        xref = img[0]
        base_image = doc.extract_image(xref)
        image_bytes = base_image["image"]
        image = Image.open(io.BytesIO(image_bytes))
        image.save(f"{image_dir}/image_{image_count}.png")
        image_count += 1

print(f"Extracted {image_count} images")
```

Extracted 18 images



Extraire les équations : utilisation de marker

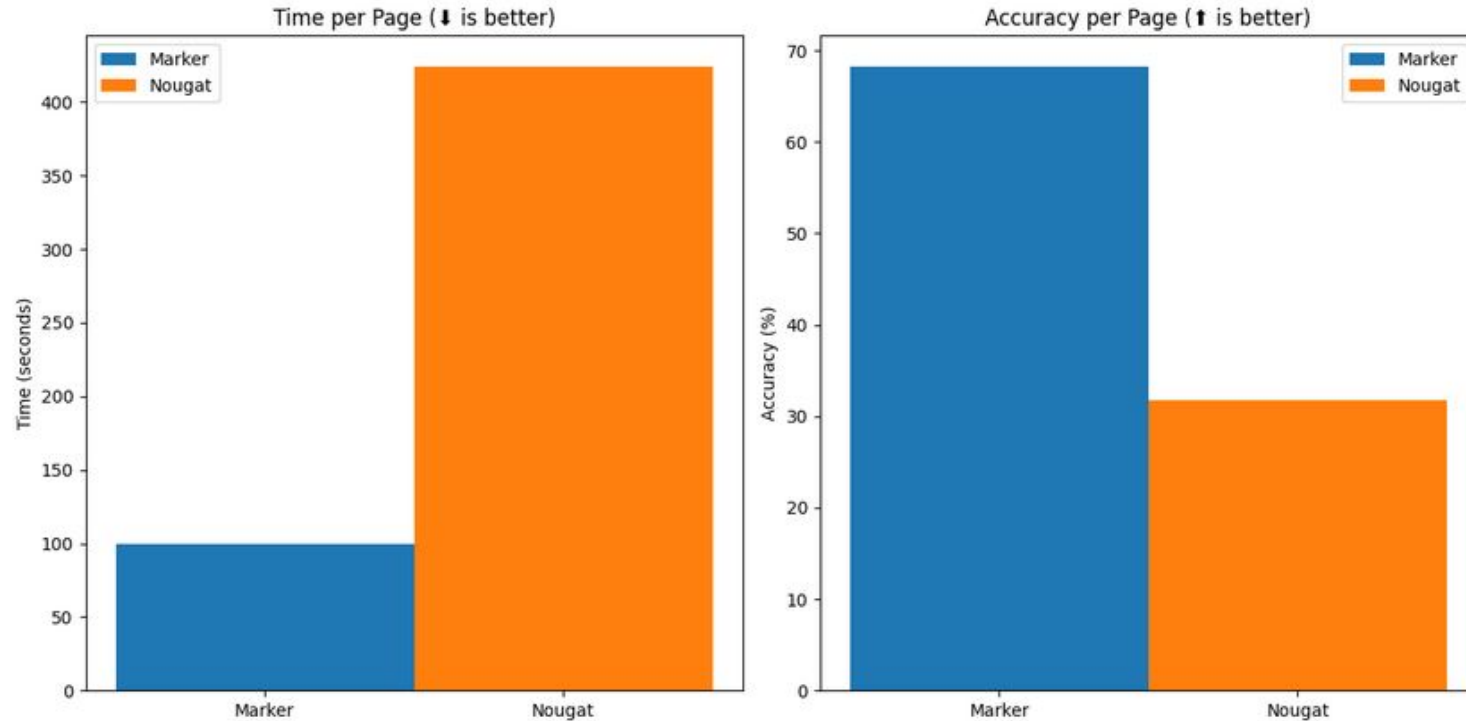
```
from marker.convert import convert_single_pdf
from marker.models import load_all_models
```

```
model_lst = load_all_models()
```

```
DOC_PATH_EQUATION = 'fausse_equation_CEPE.pdf'
# Attention la numerotation de pages commence a zero
full_text, images, out_meta = convert_single_pdf(
    DOC_PATH_EQUATION, model_lst, max_pages=10,
    langs=None, batch_multiplier=2, start_page=0)
```

```
Detecting bboxes: 100%|██████████| 1/1 [00:00<00:00, 11.94it/s]
Detecting bboxes: 100%|██████████| 1/1 [00:00<00:00, 11.89it/s]
Finding reading order: 100%|██████████| 1/1 [00:00<00:00, 18.40it/s]
```

Extraire les équations : utilisation de marker



The above results are with marker and nougat setup so they each take ~4GB of VRAM on an A6000.

Extraire les équations

- Il y a plein de LLMs open-source, par contre des solutions performantes de traitement de PDF peuvent être encore compliquées à mettre en oeuvre !
- La question du **temps de processing** peut être importante, notamment lorsqu'il s'agit de faire du RAG sur de très grandes bases de données d'archives
- Encore de l'effort pour avoir en open-source des outils pour lire parfaitement les PDFs, ou scans de livres





2.2

Sentence Embeddings

Sentence embedding

```
from sentence_transformers import SentenceTransformer

model = SentenceTransformer("all-MiniLM-L6-v2")
texts = [
    "Paris is hosting Olympic games",
    "There is no blue dog."]

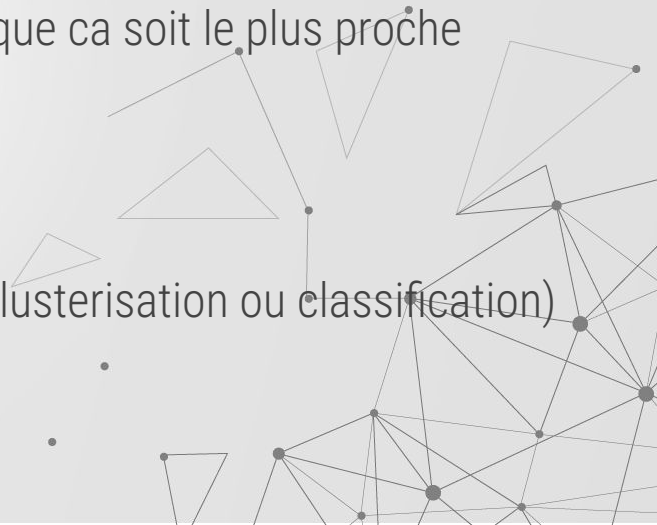
text_embeddings = model.encode(texts)
```

- Embedding des documents sans appel à des calls API.
- Attention de faire les embeddings des documents et de la query avec le **même modèle**.
- **Sentence embedding** = on travaille directement au niveau de la phrase plutôt que du token.
- **all-MiniLM-L6-v2** produit de petits embeddings de dimension 384

Le choix du sentence embedding

Il y a de **nombreux types d'embeddings disponibles**, et on va regarder plusieurs critères comme :

- La **taille de l'embedding** (et donc du modèle associé). Par exemple, si passage à l'échelle/real-time est un aspect critique,
- Sur quel **type de données** l'embedding a été entraîné pour que ca soit le plus proche possible du genre de document que nous allons embedder.
- Le **type de langage** utilisé.
- Le **type de tâches** que l'on va effectuer (semantic search, clusterisation ou classification)



Exemples de sentence embedding

Models 124

Collapse Sort: Recently updated

 Sentence Similarity • Updated 26 days ago • \pm 284k • 183	 Sentence Similarity • Updated about 1 month ago • \pm 29.6M • 1.96k
 Sentence Similarity • Updated May 7 • \pm 5.88k • 1	 Sentence Similarity • Updated May 7 • \pm 7.99k
 Sentence Similarity • Updated May 7 • \pm 1.22k	 Sentence Similarity • Updated May 7 • \pm 663
 Sentence Similarity • Updated May 7 • \pm 253k • 14	 Sentence Similarity • Updated May 7 • \pm 4.34k • 3
 Sentence Similarity • Updated May 7 • \pm 68.4k • 10	 Sentence Similarity • Updated May 7 • \pm 3.46k • 13
 Sentence Similarity • Updated May 7 • \pm 33.6k • 1	 Sentence Similarity • Updated May 7 • \pm 565k • 143
 Sentence Similarity • Updated Mar 27 • \pm 2.16k • 25	 Sentence Similarity • Updated Mar 27 • \pm 3.17k • 23
 Sentence Similarity • Updated Mar 27 • \pm 2.93k • 1	 Sentence Similarity • Updated Mar 27 • \pm 1.34k
 Sentence Similarity • Updated Mar 27 • \pm 13	 Sentence Similarity • Updated Mar 27 • \pm 99



Exemples de sentence embedding


- **msmarco-distilbert-base-v4** : entraîné sur MSMARCO, avec **knowledge distillation**.
- **nli-bert-large-cls-token** : modèle de type **BERT** fine-tuned sur des **NLI** datasets
- **paraphrase-distilroberta-base-v1** : model distillé à partir de **RoBERTa** et fine-tuné sur des tâches de paraphrasing.
- **xlm-r-100langs-bert-base-nli-stsb-mean-tokens** : XLM model entraîné sur des tâches type **NLI** et **STS** ...
- Et bien d'autres...



Un peu de vocabulaire supplémentaire

Les modèles contiennent donc des informations sur 1) le type d'architecture, 2) le type de tâche d'entraînement/fine-tuning ou 3) le type de méthode d'entraînement.

Exemple d vocabulaire à connaître / savoir reconnaître :

- **Distillation** : entraîner un petit modèle à reproduire le comportement d'un gros modèle (Knowledge Distillation).
 - **NLI (Natural Language Inference)** : des datasets pour juger/améliorer les capacités de logique d'un modèle.
 - **STS (Semantic Textual Similarity)** : déterminer le degré de similarité entre phrases.
 - **RoBERTa** : entraîner BERT de façon plus robuste.
 - **CLS token** : un token spécial (voir plus tard)
- 

Natural Language Inference

premise
string · lengths



hypothesis
string · lengths



label
class label



A person on a horse jumps over a broken down airplane.	A person is training his horse for a competition.	1 neutral
A person on a horse jumps over a broken down airplane.	A person is at a diner, ordering an omelette.	2 contradiction
A person on a horse jumps over a broken down airplane.	A person is outdoors, on a horse.	0 entailment
Children smiling and waving at camera	They are smiling at their parents	1 neutral
Children smiling and waving at camera	There are children present	0 entailment
Children smiling and waving at camera	The kids are frowning	2 contradiction
A boy is jumping on skateboard in the middle of a red bridge.	The boy skates down the sidewalk.	2 contradiction
A boy is jumping on skateboard in the middle of a red bridge.	The boy does a skateboarding trick.	0 entailment



2.3

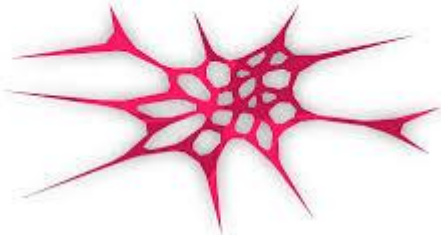
Bases de données vectorielles

Base de données vectorielles

**Semantic Search with
Open-Source ChromaDB**



FAISS
Scalable Search With Facebook AI



Les bases de données vectorielles permettent de chercher de façon efficace des vecteurs similaires.

- Algorithmes approximatifs pour résoudre cette tâche efficacement
- Fonctionnement différent des bases de données relationnelles.
- Utile pour le RAG ou juste la semantic search.

Omniprésent en pratique, au coeur de plein de cas d'usages.

De nombreux frameworks

Il y a de **nombreux frameworks** de bases de données vectorielles proposées par les différents acteurs, optimisés pour différent types de hardware, pour différents cas d'usages ou utilisant des algorithmes de recherche approximée différents :

- **FAISS** (Facebook AI Similarity Search) : Meta
- **Annoy** (Approximate Nearest Neighbors Oh Yeah) : Spotify
- **HNSW** (Hierarchical Navigable Small World)
- Et bien d'autres...



La fonction d'embedding

```
from chromadb.utils import embedding_functions

embedding_func = embedding_functions.SentenceTransformerEmbeddingFunction(
    model_name=EMBED_MODEL
)

collection = client.create_collection(
    name=COLLECTION_NAME,
    embedding_function=embedding_func)
```

- Attention, **la DB stocke le texte et non pas les embeddings**, c'est pour cela qu'il faut préciser la **fonction d'embedding**.
- Très bonne interopérabilité avec **SentenceTransformer** de HuggingFace.

Approximate k-nn

```
# Exemple de question
```

```
query = 'Does this article has many authors ? Does this article talk about climate change ?'
```

```
# On recupere les 5 chunks les plus proches de la question
```

```
# (Par default Langchain utilise la cosine distance metric)
```

```
docs_chroma = db_chroma.similarity_search_with_score(query, k=5)
```



Ajouter des métadonnées

```
documents = [
    "The latest iPhone model comes with impressive features and a powerful camera.",
    "Exploring the beautiful beaches and vibrant culture of Bali is a dream for many travelers.",
    "Einstein's theory of relativity revolutionized our understanding of space and time."

genres = [
    "technology",
    "travel",
    "science"]

embedding_func = embedding_functions.SentenceTransformerEmbeddingFunction(
    model_name=EMBED_MODEL
)

collection = client.create_collection(
    name='db_with_meta',
    embedding_function=embedding_func)

collection.add(
    documents=documents,
    ids=[f"id{i}" for i in range(len(documents))],
    metadatas=[{"genre": g} for g in genres])
```

Query Hybride

```
collection.query(  
  query_texts=["Teach me about music history"],  
  where={"genre": {"$eq": "music"}},  
  n_results=1)
```

```
collection.query(  
  query_texts=["Teach me about music history"],  
  where={"genre": {"$in": ["music", "history"]}},  
  n_results=1)
```

- Prendre le temps de construire une base de données vectorielles avec des métadonnées.
- Cela permet ensuite de faire des **recherches hybrides**. Intéressant pour faire un moteur de recherche sur données privées.

Cas d'usages

- Les RAGs
- Les systèmes de recommandation (Spotify a même son propre framework de base de données vectorielle !)
- Semantic search (text, image, etc...) : avec applications sur données perso



Mise en oeuvre opérationnelle

- **Service-based :**

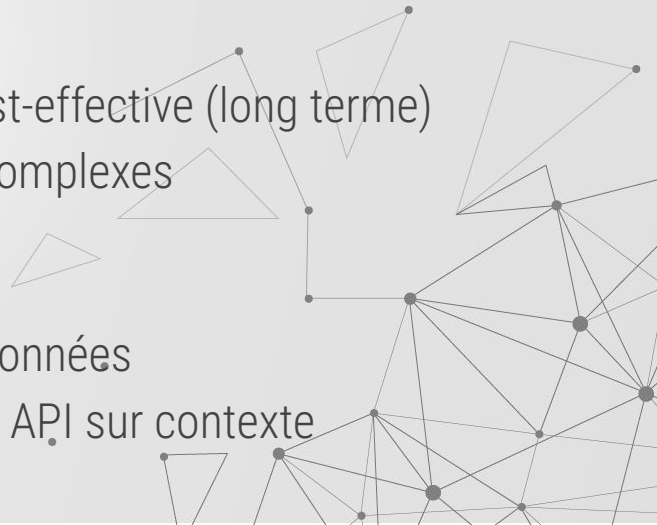
- LLM-as-a-service, fourni par cloud provider/entreprise spécialisée
- Les + : rapide, facile, peu coûteux (initialement)
- Les - : vendor lock-in, pas de contrôle sur l'architecture ou les données, latence

- **In-house :**

- Faire tourner le modèle sur ses serveurs
- Les + : contrôle total, customisation, optimisation, cost-effective (long terme)
- Les - : expertise technique, ressources, mises à jour complexes

- **Hybride :**

- Mix service/in-house ; calls API sur une fraction des données
- Exemple : RAG hybride avec embedding en local, calls API sur contexte





TP : Mettre en place son propre RAG.



Fine-tuned RAGs !

```
from transformers import RagTokenizer, RagRetriever, RagSequenceForGeneration, Trainer, TrainingArguments

# Load pre-trained RAG model and tokenizer
tokenizer = RagTokenizer.from_pretrained("facebook/rag-sequence-base")
retriever = RagRetriever.from_pretrained("facebook/rag-sequence-base", index_name="exact")
model = RagSequenceForGeneration.from_pretrained("facebook/rag-sequence-base", retriever=retriever)

# Define training arguments
training_args = TrainingArguments(
    output_dir="./results",
    evaluation_strategy="epoch",
    learning_rate=5e-5,
    per_device_train_batch_size=2,
    per_device_eval_batch_size=2,
    num_train_epochs=3,
    weight_decay=0.01,
)
```


Trucs et astuces (1)

- LLMs à grande context window peuvent surpasser un RAG (donner tout le document !), mais : (1) très gourmand en ressources et (2) pas de focus sur l'info vraiment importante.
- Tester la remontée des chunks pertinents en les classant par ordre d'apparition dans le document (au lieu d'ordre décroissant de score de similarité) peut améliorer considérablement la qualité des réponses.

In Defense of RAG in the Era of Long-Context Language Models

Tan Yu
NVIDIA
Santa Clara, California
United States
tayu@nvidia.com

Anbang Xu
NVIDIA
Santa Clara, California
United States
anbangx@nvidia.com

Rama Akkiraju
NVIDIA
Santa Clara, California
United States
rakkiraju@nvidia.com

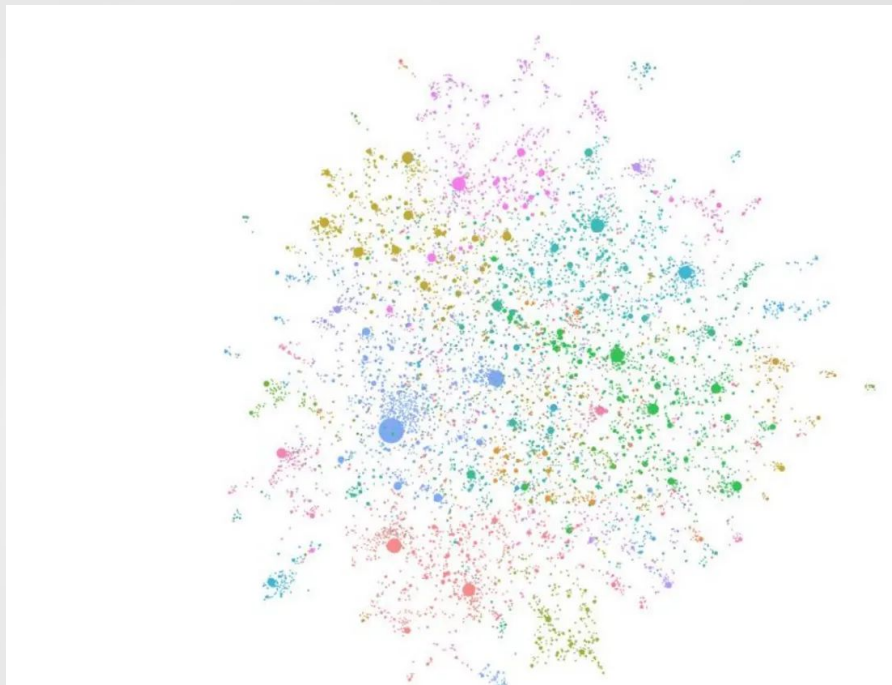
Trucs et astuces (2)

- Si la base de connaissances est trop volumineuse par rapport au nombre de chunks maximum à remonter, on peut la réduire ou la diviser en plusieurs bases pour créer plusieurs RAGs spécialisés performants, plutôt qu'un seul RAG.
- Le nombre de chunks pertinents à remonter et leur longueur doivent être adaptés au nombre de paramètres du LLM choisi, ou inversement.



Au-delà du simple RAG

- RAGChecker
- EfficientRAG
- FlashRAG
- Agentic RAGs
- GraphRAG



GraphRAG

- Améliore la capacité à prendre en compte des informations/contextes différents.
- **Indexing stage :**
 - Découper en chunks (unités de texte)
 - Entity extraction (noms, localisations, dates, organisations, etc.)
 - Clustering hiérarchique sur la base des entités pour construire un graphe
 - Générer des résumés de chaque cluster du graphe (entités, relations, propriétés, etc.)
- **Query stage :**
 - Global/local search dans les clusters en fonction de la question
 - Aller chercher du contexte à l'échelle du sous-cluster (communauté), du cluster, ou de plusieurs clusters
 - Agréger les contextes pour obtenir une réponse





3

Supervised Fine-Tuning

Les différents packages



Les paramètres pour fine-tuner un LLM

```
[ ]: training_params = TrainingArguments(  
    output_dir="./results",  
    num_train_epochs=1,  
    per_device_train_batch_size=4,  
    gradient_accumulation_steps=1,  
    optim="paged_adamw_32bit",  
    save_steps=25,  
    logging_steps=25,  
    learning_rate=2e-4,  
    weight_decay=0.001,  
    fp16=False,  
    bf16=False,  
    max_grad_norm=0.3,  
    max_steps=-1,  
    warmup_ratio=0.03,  
    group_by_length=True,  
    lr_scheduler_type="constant",  
    report_to="tensorboard"  
)
```

- La plupart des détails d'entraînement sont **sous le capot**.
- Certains paramètres sont liés à l'optimisation : **batch, optim, gradient_accumulation_steps, weight_decay, learning_rate**
- Souvent **on gardera les valeurs par défaut**, mais il faut être capable d'interpréter les résultats en fonction du **learning rate** par exemple.



3.1

Entraîner un réseau de neurones

Objectif

Break-down ce code typique d'entraînement d'un réseau de neurones.

- **enumerate(training_loader)**: on visite progressivement tous les exemples de la base de données d'entraînement. A comprendre.
- **optimizer.zero_grad()** : c'est juste une fonction pour remettre à zéro les **gradients**.
- **loss.backward()** : on calcule les **gradients**. A comprendre.
- **optimizer.step()** : on fait un pas de **gradient**. A comprendre.

```
import torch # pytorch
```

```
for i, data in enumerate(training_loader):
```

```
    inputs, labels = data
```

```
    # Zero your gradients for every batch!  
    optimizer.zero_grad()
```

```
    # Make predictions for this batch  
    outputs = model(inputs)
```

```
    # Compute the loss and its gradients  
    loss = loss_fn(outputs, labels)  
    loss.backward()
```

```
    # Adjust learning weights  
    optimizer.step()
```

```
# clf = classifier
```

```
clf = RandomForestClassifier(max_depth=2, random_state=0)  
clf.fit(X_train, y_train)
```

Pourquoi parler d'optimisation?

On utilise θ pour noter l'ensemble des paramètres d'un réseau de neurone.

Que l'on fasse de la régression ou classification on cherche les θ de qui minimisent **le risque empirique**

$$\min_{\theta} J(\theta) = \sum_{i=1}^n \underbrace{L(f_{\theta}(x_i), y_i)}_{:=J_i(\theta)}.$$

C'est un **problème d'optimisation difficile** parce que

- n peut être très grand
- Avec un réseau de neurone, les $J_i(\theta)$ sont déjà très complexes (beaucoup de dimension et non-convexe)

```
>>> from sklearn import linear_model
>>> reg = linear_model.Lasso(alpha=0.1)
>>> reg.fit([[0, 0], [1, 1]], [0, 1])
Lasso(alpha=0.1)
>>> reg.predict([[1, 1]])
array([0.8])
```

Un .fit() ca ne suffit plus!

L'optimisation en boîte noire?

On doit résoudre ce problème de minimisation de risque, mais pourquoi ne pas le faire sous le capot?

$$\min_{\theta} J(\theta) = \sum_{i=1}^n \underbrace{L(f_{\theta}(x_i), y_i)}_{:= J_i(\theta)}.$$

- Il n'y a pas de **solution simple** que l'on peut automatiser
- **Bien optimiser = bien apprendre**
- Beaucoup de concepts d'optimisation dans le code classique
- Lien direct avec le hardware (batch, GPU)

```
import torch # pytorch

for i, data in enumerate(training_loader):

    inputs, labels = data

    # Zero your gradients for every batch!
    optimizer.zero_grad()

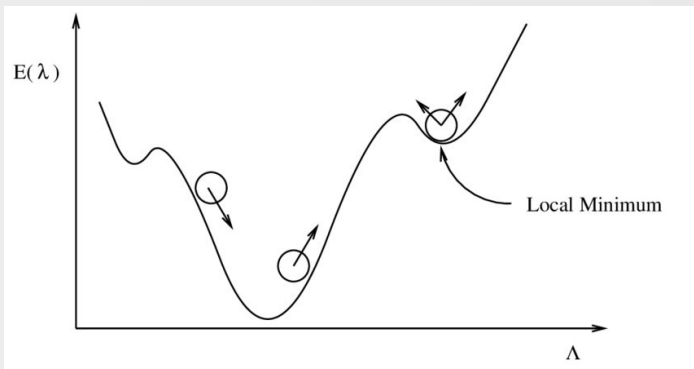
    # Make predictions for this batch
    outputs = model(inputs)

    # Compute the loss and its gradients
    loss = loss_fn(outputs, labels)
    loss.backward()

    # Adjust learning weights
    optimizer.step()
```

Qu'est-ce que le gradient?

Intuitivement: le gradient (la dérivée) c'est la direction à suivre pour descendre le plus rapidement possible dans la pente.



Formellement: le gradient permet d'approximer linéairement une fonction $F : \mathbb{R}^d \rightarrow \mathbb{R}$, pour h petit

$$F(w + h) \approx F(w) + \langle \nabla F(w); h \rangle.$$

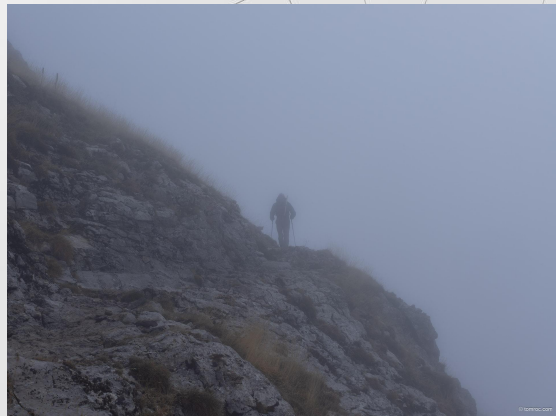
La descente de gradient

Et donc si en partant d'un point w_t , pour trouver un point w_{t+1} tel que $F(w_{t+1})$ soit plus petit que $F(w_t)$, il suffit **d'aller un peu dans la direction du gradient**:

$$w_{t+1} = w_t - \lambda_t \nabla F(w_t) .$$

C'est la stratégie que l'on prendrait naturellement pour descendre une montagne dans le brouillard!

- Comment calculer le gradient?
- Comment choisir **'de combien on suit le gradient'**?
- Pour nous, **w** ce sera les poids du réseau de neurone

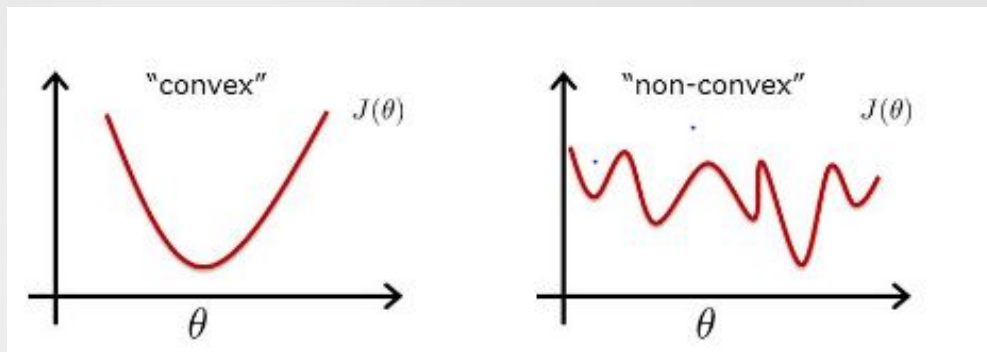


Deux difficultés: non-convexité et dimension

Il y a deux premières difficultés pour minimiser le risque empirique via une descente de gradient:

- La dimension des données (de θ) parce que pour l'instant on n'imagine minimiser qu'avec **une ou deux dimensions**.
- La non-convexité de la fonction $J(\theta)$.

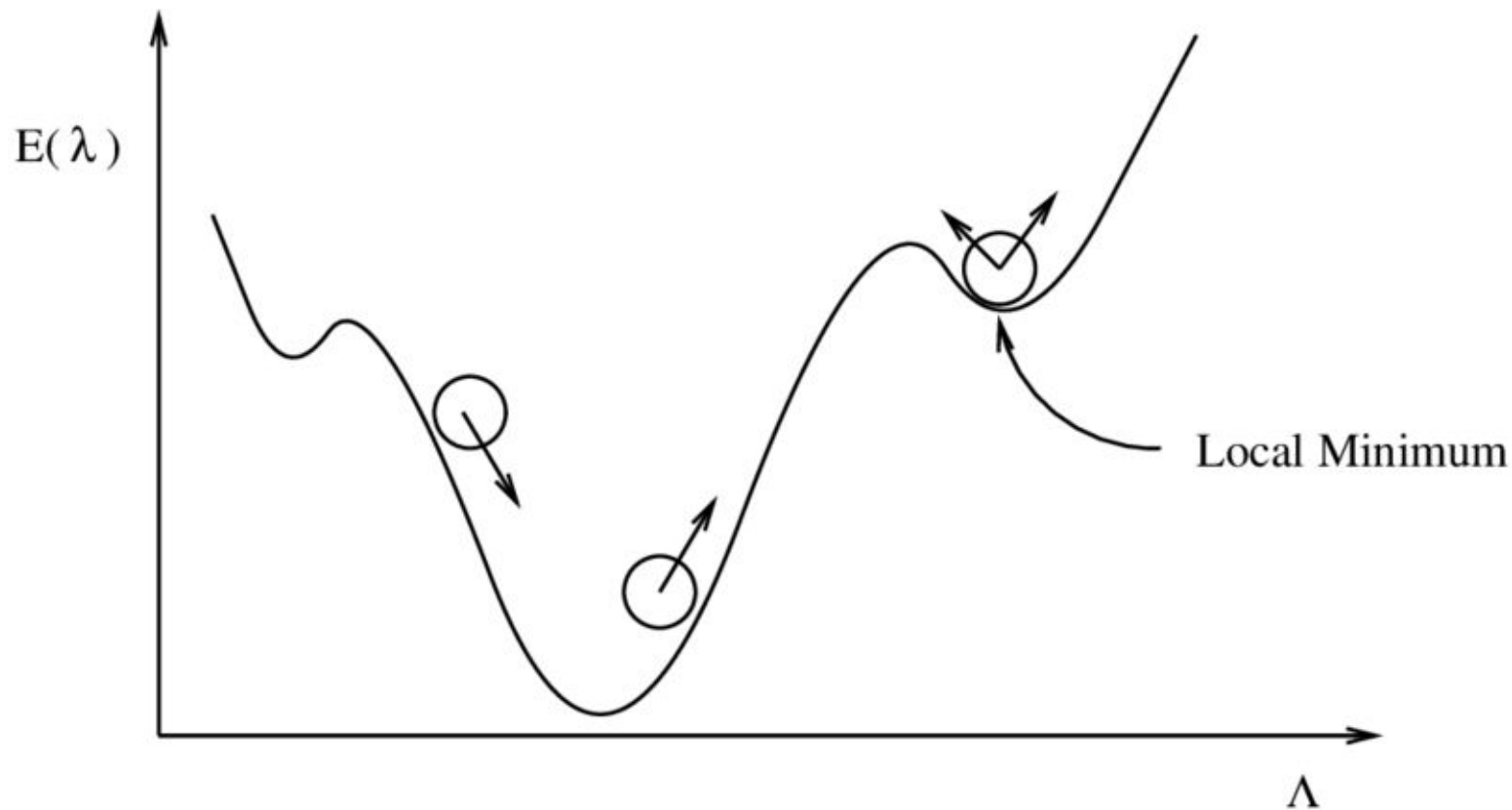
$$\min_{\theta} J(\theta) = \sum_{i=1}^n \underbrace{L(f_{\theta}(x_i), y_i)}_{:= J_i(\theta)}.$$



C'est précisément à cause de cette non-convexité que le monde du deep learning est **un domaine empirique**.

- **Côté alchimiste/mécano**: donc il faut comprendre les détails
- Globalement, nous n'avons **pas de certitude mathématiques** sur comment ça fonctionne.

Les difficultés: non-convexité et dimension



Descente de gradient pour minimiser le risque empirique?

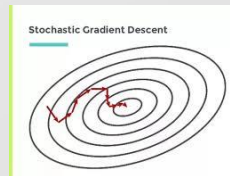
Troisième difficulté = calculer le gradient $\nabla J(\theta)$ pour **minimiser le risque empirique**

$$\min_{\theta} J(\theta) = \sum_{i=1}^n \underbrace{L(f_{\theta}(x_i), y_i)}_{:= J_i(\theta)}.$$

$$\nabla J(\theta) = \frac{1}{n} \sum_{i=1}^n \nabla_{\theta} L(f_{\theta}(x_i), y_i).$$

A chaque itération de la descente de gradient il va falloir

- Calculer **chacun des gradients** à l'intérieur de la somme ?
- En faire la somme même quand **n** est très grand?





La différentiation automatique

La **différentiation automatique** = PyTorch permet de automatiquement faire le calcul du gradient:

$$\nabla_{\theta} L(f_{\theta}(x_i), y_i)$$

- **Considérer cela comme une boîte noire**
- En anglais on appelle cela la **backpropagation** et faire le calcul des gradients une **backward pass**.
- Le gradient calculé **couche par couche en partant de la fin**, c'est juste une généralisation de

$$(f \circ g)' = g' \cdot f' \circ g$$

- Cela a un **coût computationnel important**, on veut en faire le moins possible

```
import torch # pytorch

for i, data in enumerate(training_loader):

    inputs, labels = data

    # Zero your gradients for every batch!
    optimizer.zero_grad()

    # Make predictions for this batch
    outputs = model(inputs)

    # Compute the loss and its gradients
    loss = loss_fn(outputs, labels)
    loss.backward()

    # Adjust learning weights
    optimizer.step()
```

Descente de gradient stochastique

Pour implémenter la descente de gradient classique il faudrait donc faire **n** backward pass à chaque itérations, même lorsque **n** est très grand?

$$\nabla J(\theta) = \frac{1}{n} \sum_{i=1}^n \nabla_{\theta} L(f_{\theta}(x_i), y_i).$$

Solution : la descente de gradient stochastique (= aléatoire). On tire aléatoirement B (avec $B \ll n$) éléments de la base de données et on considère l'approximation

$$\frac{1}{B} \sum_{j \in (i_1, \dots, i_B)} \nabla L(f_{\theta_t}(x_j), y_j) \approx \frac{1}{n} \sum_{i=1}^n \nabla L(f_{\theta_t}(x_i), y_i)$$

La descente de gradient stochastique ressemble alors à

$$\theta_{t+1} = \theta_t - \lambda_t \frac{1}{B} \sum_{j \in (i_1, \dots, i_B)} \nabla L(f_{\theta_t}(x_j), y_j) .$$



Descente de gradient en pytorch

```
model = ClassifieurMultiCouche(28*28, 100, 50, 10)
# SGD = Stochastic Gradient Descent.
optimizer = torch.optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
```

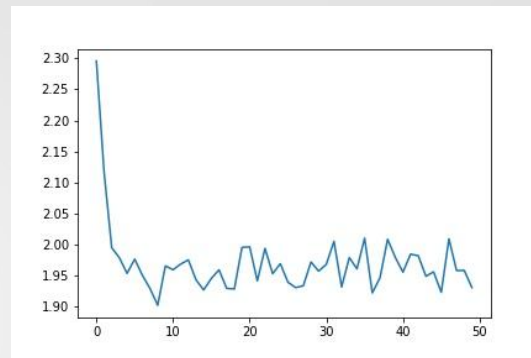
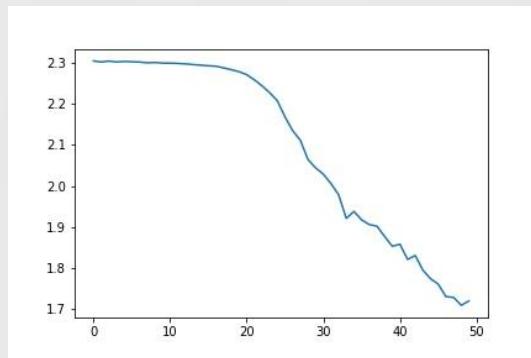
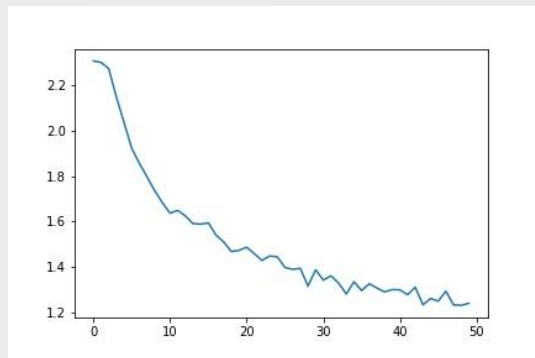
L'optimizer va se charger de tous les détails d'optimisation pour nous

- Il faut **spécifier les paramètres qui vont être modifiés**
- **optimiser.step()** fera un pas de descente de gradient
- Chaque pas modifie **TOUT les paramètres** (mais très légèrement)
- le paramètre lr = **learning-rate. Super important.**
- momentum = pousser la bille



La question du learning rate

Trois learning rate différents: un trop grand, un trop petit et un bon. Qui est qui?



```
model = ClassifieurMultiCouche(28*28, 100, 50, 10)
# SGD = Stochastic Gradient Descent.
optimizer = torch.optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
```

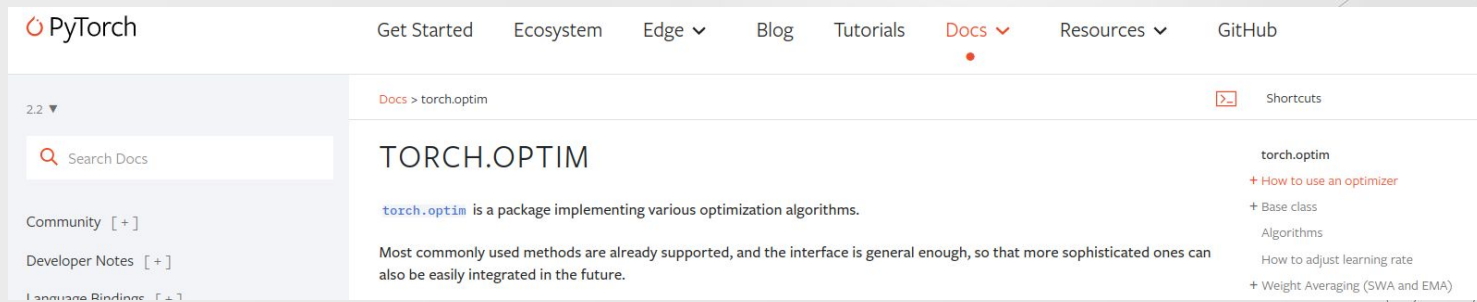
Plein de méthodes possibles

On n'a parlé que de la descente de gradient stochastique, mais il y a de nombreux autres exemples de d'algorithmes de descente de gradient

- **Adam**: Ada pour adaptive.
- **AdaGrad**: mais en pratique le choix de l'optimiser c'est du deuxième ordre..
- **RMSProp**

Il y a aussi de nombreux paramètres à varier et méthodes (pas besoin de rentrer dans ce détail)

- **Momentum** : lancer la bille
- **Learning rate schedule**: faire varier la taille du learning rate pendant l'entraînement
- **gamma** : le facteur multiplicatif pour décroître le learning rate.



The screenshot shows the PyTorch documentation website. The top navigation bar includes links for 'Get Started', 'Ecosystem', 'Edge', 'Blog', 'Tutorials', 'Docs' (highlighted with a red dot), 'Resources', and 'GitHub'. The left sidebar shows the version '2.2' and a search bar. The main content area is titled 'TORCH.OPTIM' and describes the `torch.optim` package as implementing various optimization algorithms. It notes that most commonly used methods are supported and the interface is general enough for integration in the future. A right sidebar lists shortcuts for `torch.optim`, including 'How to use an optimizer', 'Base class', 'Algorithms', 'How to adjust learning rate', and 'Weight Averaging (SWA and EMA)'.



3.2

Dataset et DataLoader

Pratique: le pytorch.DataLoader

Deux objets importants: le **dataset** et le **DataLoader**

Le dataset c'est un objet python de base qui permet d'indexer les datapoints.

```
import torchvision
import torchvision.transforms as transforms

training_set = torchvision.datasets.MNIST("./", train=True, transform=transforms.ToTensor(), download=True)

print(len(training_set))

60000
```

Le paramètre transform est très important. On en discutera en détail au moment des réseaux convolutionnels.

```
data, label = training_set[5]
print(data.shape)
print(label)

torch.Size([1, 28, 28])
2
```

Pratique: le pytorch.DataLoader

Le DataLoader est une façon de penser l'entraînement des réseaux de neurones

Analogie de l'enfant qui apprend au fur et à mesure en feuilletant les pages d'un livre.

C'est un objet optimisé pour l'efficacité du chargement des données

```
•[13]: import torch
import torchvision # contient les datasets
import torchvision.transforms as transforms # permet de facilement faire des operations

training_set = torchvision.datasets.MNIST("./", train=True, transform=transforms.ToTensor(), download=True)
training_loader = torch.utils.data.DataLoader(training_set, batch_size=4, shuffle=True)

[14]: for i, data in enumerate(training_loader):
    X,y = data
    print(X.shape)
    print(y.shape)

torch.Size([4, 1, 28, 28])
torch.Size([4])
```


Descente de gradient stochastique

- A chaque itération, le **pytorch.DataLoader** va échantillonner au hasard B éléments de la base de donnée
- La backward pass calcule le gradient pour chacun de ces échantillons
- On retient que le nombre B s'appelle le **batch-size**
- Pour entraîner on va faire de nombreuses itérations

```
import torch # pytorch

for i, data in enumerate(training_loader):
    inputs, labels = data

    # Zero your gradients for every batch!
    optimizer.zero_grad()

    # Make predictions for this batch
    outputs = model(inputs)

    # Compute the loss and its gradients
    loss = loss_fn(outputs, labels)
    loss.backward()

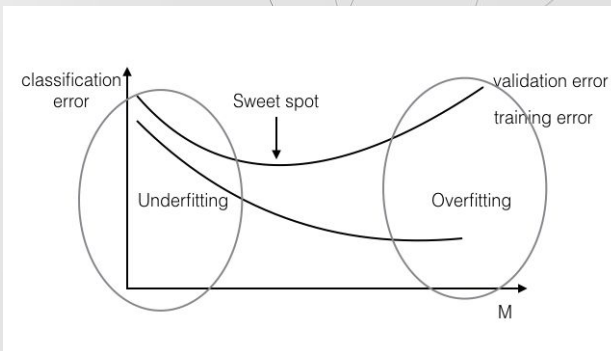
    # Adjust learning weights
    optimizer.step()
```

Nombre d'itérations et choix du batch-size

On utilise le concept d'époques (**epoch**) pour avoir une idée du nombre d'itérations nécessaires afin d'avoir de bonnes performances

- Une epoch = quand on a vu au moins une fois tous les éléments de la base de donnée
- On entraîne sur **de nombreuses epoch** mais il n'y a pas de règle a priori (hyper-paramètre!)
- Analogie : un écolier a besoin de lire un livre plusieurs fois pour l'avoir bien compris. **Trop souvent = surapprentissage!**

Choix du batch-size = en première approximation c'est surtout une question de mémoire RAM disponible dans le GPU.



Récapitulatif sur le snippet d'entraînement

- Chaque itération va changer tous les poids du réseau de neurone mais un tout petit peu.
- Apprentissage et optimization sont intriqués et il faut comprendre les différentes subtilités.
- Est-ce que toutes les lignes de code sont plus ou moins claires?

```
import torch # pytorch

for i, data in enumerate(training_loader):

    inputs, labels = data

    # Zero your gradients for every batch!
    optimizer.zero_grad()

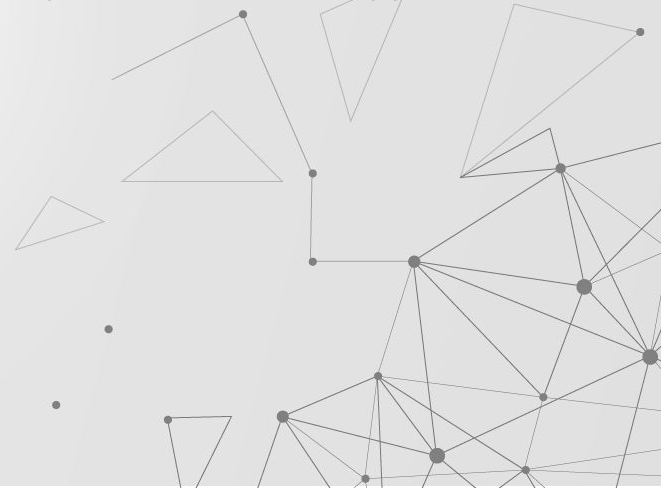
    # Make predictions for this batch
    outputs = model(inputs)

    # Compute the loss and its gradients
    loss = loss_fn(outputs, labels)
    loss.backward()

    # Adjust learning weights
    optimizer.step()
```

Vocabulaire à maîtriser

- **Epoch**: visiter une fois toutes les éléments de la base de donnée.
- **batch-size** : le nombre de données que le réseau considère à chaque itération.
- **lr**: le learning rate = de combien l'alpiniste suit le gradient.
- **Backward pass** : façon de calculer en interne le gradient pour l'optimisation. Ca s'appelle aussi **backpropagation**.
- **Forward pass**: juste calculer $f(x)$, c.a.d. de l'inférence
- Et surement d'autres mots!



La question du device

Avec pytorch, on peut très facilement indiquer dans le code dans **quel espace mémoire nous voulons placer les tenseurs**. Il va y avoir deux tenseurs :

- Les tenseurs des données
- Les tenseurs associés aux paramètres du réseau de neurone

Quand on a accès à une GPU, on va vouloir faire toutes nos opérations sur ce GPU (dans la limite évidemment de la mémoire disponible)

```
[7]: # On demande a python a quel device on a acces (on a par default  
# toujours acces au CPU)  
  
device = 'cuda' if torch.cuda.is_available() else 'cpu'  
print(device)  
  
cuda
```





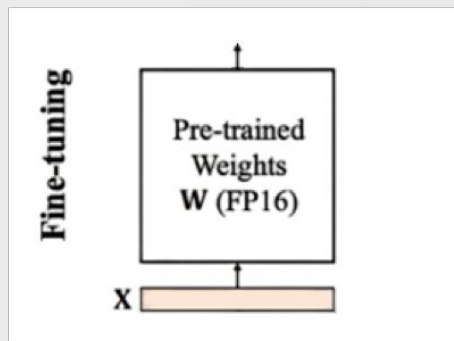
3.3

LLM fine-tuning

Stratégies

Full Fine-Tuning

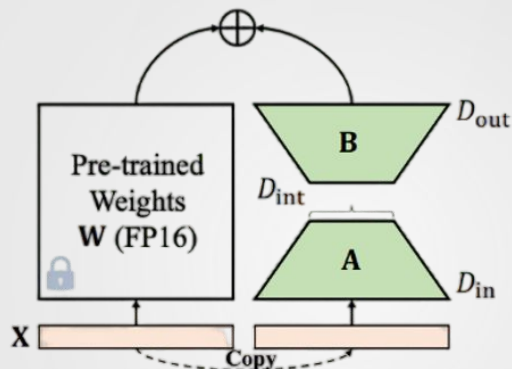
16-bit precision



- ✓ Best performance
- ✗ Very high VRAM usage

LoRA

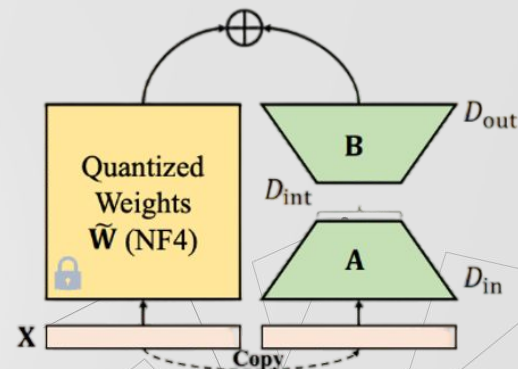
16-bit precision



- ✓ Quick training
- ✗ Still costly

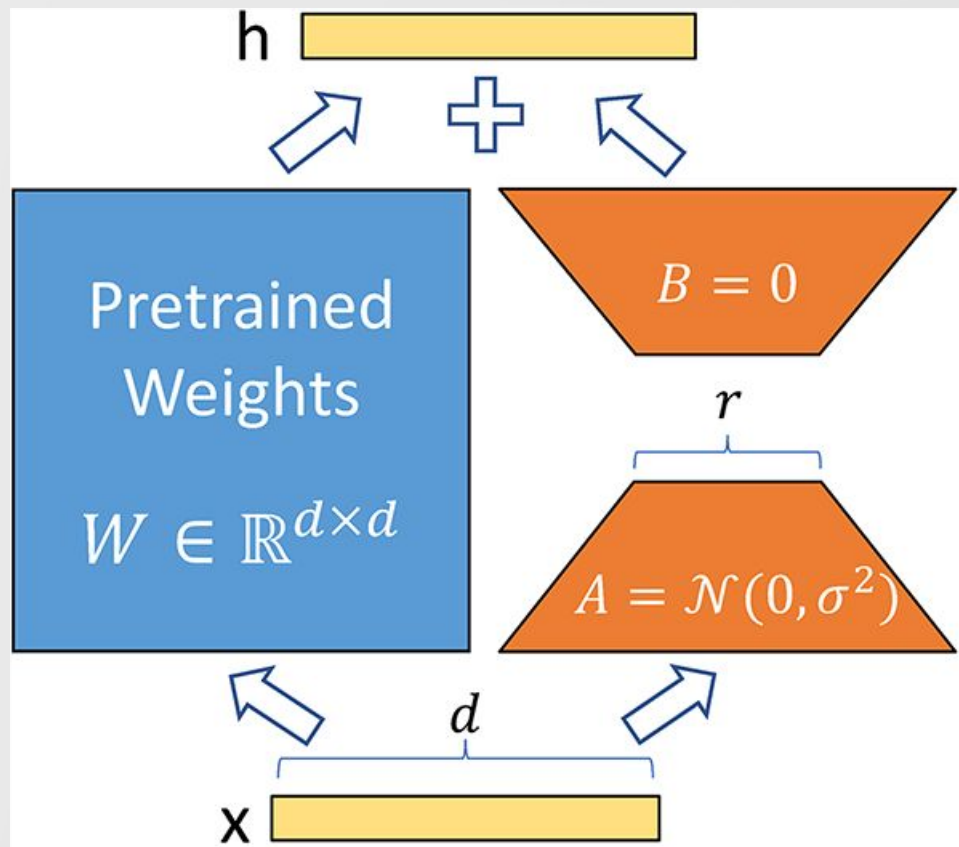
QLoRA

4-bit precision

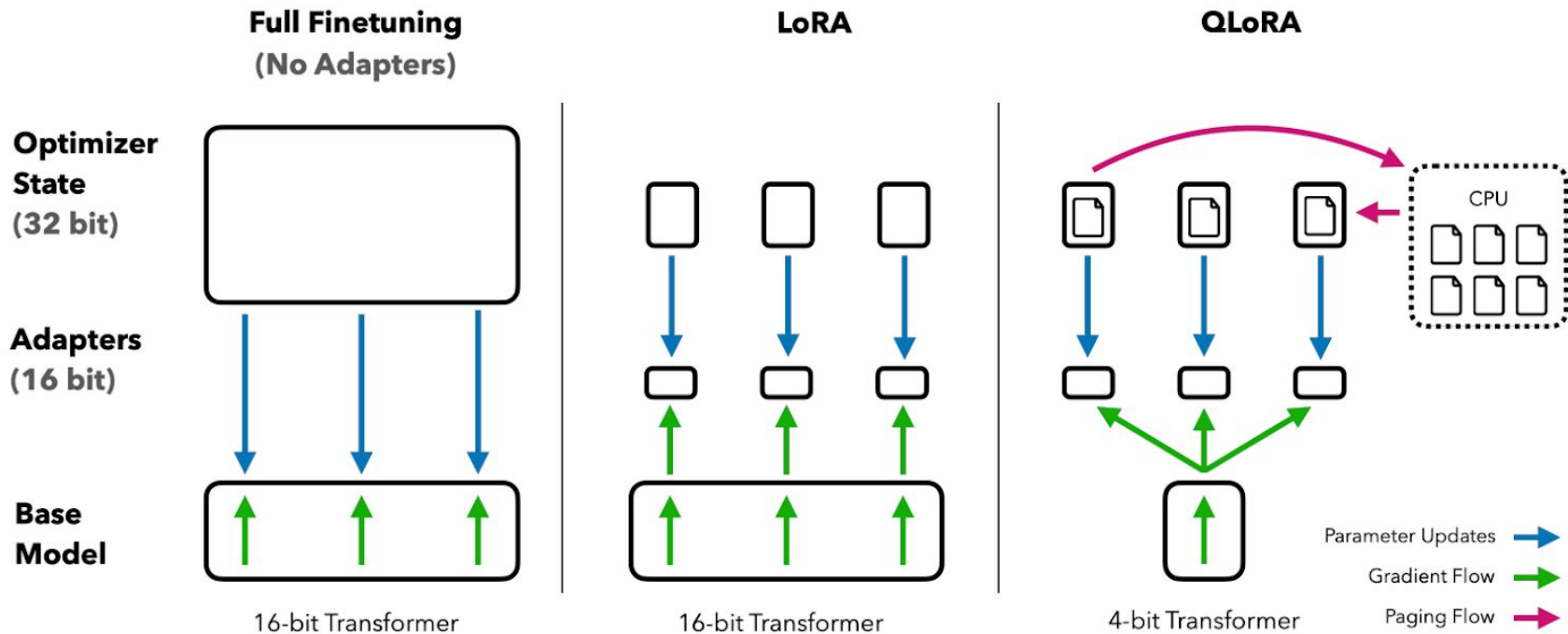


- ✓ Low VRAM usage
- ✗ Degrades performance

Low Rank Adaptation Training (LoRA)

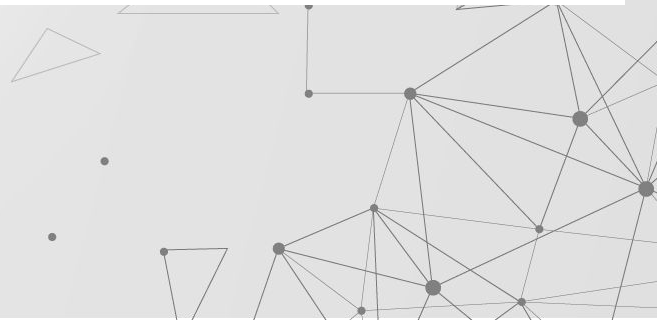


Quantized LoRA (QLoRA)



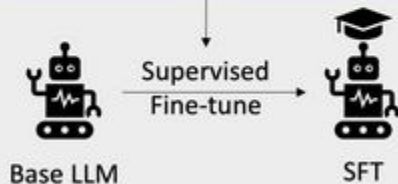
Code super simple

```
peft_params = LoraConfig(  
    lora_alpha=16,  
    lora_dropout=0.1,  
    r=64,  
    bias="none",  
    task_type="CAUSAL_LM",  
)
```

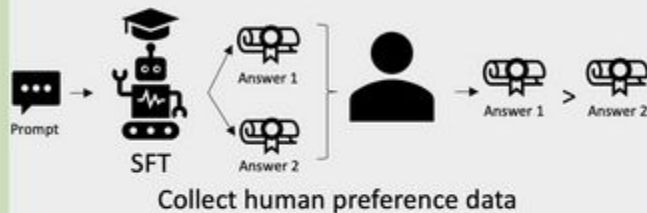


RLHF pour adapter LLM

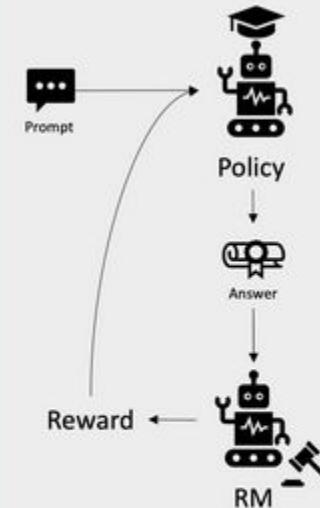
Step 1 Supervised Fine-Tuning



Step 2 Training a Reward Model



Step 3 Optimize Policy



PPO



4

Annexes

Quelques limitations

- Ressources de calcul/inférence
- Coût environnemental
- Hallucinations
- Performance statistique



Quelques blogs/ressources

<https://huyenchip.com/blog/>

<https://www.rungalileo.io/blog>

<https://lightning.ai/pages/llm-learning-lab/>

<https://mlabonne.github.io/blog/>

<https://www.llmwatch.com/>

<https://huggingface.co/docs>

<https://www.llamaindex.ai/blog>

<https://superlinked.com/vectorhub>

<https://qdrant.tech/articles/>

<https://eugeneyan.com/writing/>

