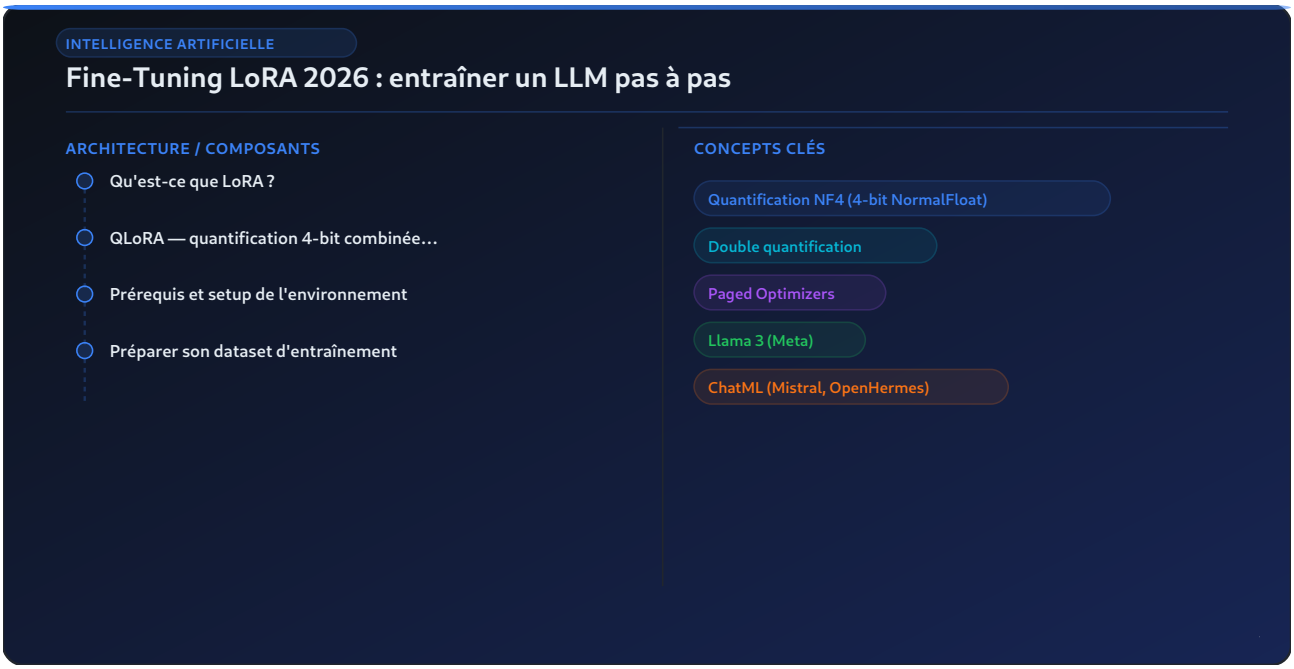
Fine-Tuning LoRA 2026 : Entraîner un LLM pas à pas avec QLoRA

> Fine-tuner un LLM avec LoRA en 2026 : QLoRA, PEFT, HuggingFace Transformers. Dataset, entraînement GPU, évaluation ROUGE. Exemple complet Llama 3/Mistral.

Le fine-tuning d'un grand modèle de langage (LLM) a longtemps été réservé aux équipes disposant de clusters GPU de plusieurs dizaines de milliers de dollars. LoRA — Low-Rank Adaptation — a changé la donne en rendant l'adaptation de modèles comme Llama 3, Mistral ou Phi-4 accessible sur un seul GPU grand public. L'idée centrale est élégante : plutôt que de mettre à jour les milliards de paramètres d'un modèle pré-entraîné, LoRA ajoute de petites matrices de rang faible aux couches d'attention et n'entraîne que ces matrices additionnelles, représentant typiquement moins de 1 % des paramètres totaux. QLoRA pousse encore plus loin en combinant la quantification 4 bits du modèle de base avec l'adaptation LoRA, permettant de fine-tuner un modèle de 70 milliards de paramètres sur un seul GPU de 48 Go de VRAM. En 2026, ces techniques sont devenues le standard industriel pour adapter des LLMs à des domaines spécialisés : classification d'alertes de sécurité, génération de rapports de pentest, extraction d'IoC, conformité réglementaire. Ce guide vous emmène de la théorie mathématique simplifiée à un entraînement complet fonctionnel, avec évaluation quantitative et export du modèle final prêt pour la production. Nous couvrirons les prérequis techniques, la préparation du dataset au format instruction tuning, la configuration BitsAndBytes pour la quantification 4 bits, le SFTTrainer de la bibliothèque TRL, l'évaluation ROUGE et le merge des poids LoRA dans le modèle de base pour un déploiement sans dépendance à PEFT.



Qu'est-ce que LoRA ?

LoRA (Low-Rank Adaptation of Large Language Models, Hu et al. 2021) repose sur une observation mathématique : lors du fine-tuning classique, la mise à jour des poids ΔW a un rang intrinsèquement faible. C'est-à-dire que la nouveauté apprise par le modèle peut être approximée par une matrice de rang beaucoup plus petit que la matrice de poids originale.

Formellement, si W est une matrice de poids pré-entraînée de dimension d×k, LoRA approxime la mise à jour comme suit :

```
W' = W + delta_W = W + B x A
  où B de dimension (d x r), A de dimension (r x k), rang r bien inférieur à min(d, k)
```

Pendant l'entraînement, W est gelée (frozen), seules les matrices A et B sont entraînées. B est initialisée à zéro et A avec une distribution gaussienne aléatoire, ce qui garantit que delta_W = 0 au début de l'entraînement. Un hyperparamètre `lora_alpha` scale le produit BA : la mise à jour effective est (alpha/r) × BA, permettant de contrôler l'amplitude de l'adaptation.

Le nombre de paramètres entraînables est r × (d + k) au lieu de d × k. Pour une couche d'attention avec d=4096 et k=4096, un rang r=16 réduit les paramètres de 16,7M à 131k — soit

99,2 % de réduction. C'est cette propriété qui rend LoRA si attractif : on entraîne 100 fois moins de paramètres tout en obtenant des résultats proches du full fine-tuning.

Comparaison : Full Fine-Tuning vs LoRA vs QLoRA

Méthode	Paramètres entraînés	VRAM (7B)	VRAM (13B)	Qualité relative
Full Fine-Tuning	100 %	~80 Go	>160 Go	Optimale (référence)
LoRA (bf16)	0,5 à 2 %	~20 Go	~40 Go	Très proche (-1 %)
QLoRA (4-bit NF4)	0,5 à 2 %	~8 Go	~14 Go	Légèrement inférieure (-2 %)

QLoRA — quantification 4-bit combinée à LoRA

QLoRA (Dettmers et al., 2023) combine deux innovations pour réduire encore davantage la VRAM requise :

⚡ **Retour terrain**

J'ai supervisé un fine-tuning LoRA pour un éditeur de logiciels juridiques qui voulait adapter Mistral 7B à la terminologie du droit OHADA. Le jeu de données initial de 800 exemples produisait un modèle qui sur-apprenait — il reproduisait les formules exactes sans généraliser. Doubler le corpus en paraphrasant les exemples existants et baisser le learning rate à 2e-5 a stabilisé les métriques d'évaluation.

Quantification NF4 (4-bit NormalFloat) : les poids du modèle de base sont quantifiés en 4 bits au lieu de 16 bits (bfloat16) ou 32 bits (float32). NF4 est une représentation optimale pour des

distributions de poids gaussiennes, préservant mieux la précision que la quantification entière classique INT4.

Double quantification : les constantes de quantification elles-mêmes sont quantifiées à leur tour, économisant environ 0,5 bits supplémentaires par paramètre — soit environ 512 Mo de moins pour un modèle 7B.

Paged Optimizers : la mémoire de l'optimiseur (états Adam) est gérée en mode paginé entre GPU et CPU RAM, évitant les OOM lors des pics de mémoire pendant le backward pass.

En pratique, QLoRA permet de fine-tuner Llama 3 8B sur un RTX 3090 (24 Go VRAM) ou même sur un RTX 4070 Ti (12 Go VRAM) avec un rang LoRA réduit. La perte de qualité par rapport à LoRA en bf16 est généralement inférieure à 2 % sur les benchmarks standards (MT-Bench, MMLU), ce qui la rend imperceptible pour la plupart des cas d'usage métier.

La bibliothèque `bitsandbytes` de Tim Dettmers implémente ces techniques en CUDA. Elle s'intègre directement avec `transformers` via le paramètre `quantization_config` de `from_pretrained`. En 2026, cette bibliothèque supporte également la quantification 8-bit (LLM.int8()) et la quantification FP8 pour les GPU H100/H200, offrant un spectre complet de compromis précision/mémoire.

Prérequis et setup de l'environnement

Avant de lancer l'entraînement, l'environnement doit être configuré avec soin. Les versions des bibliothèques importent énormément dans l'écosystème HuggingFace : une incompatibilité entre `transformers`, `peft` et `trl` peut provoquer des erreurs silencieuses ou des résultats dégradés. Voici le stack validé pour juin 2026.

```
 # Installation des dependances (Python 3.10+, CUDA 12.1+)
pip install torch==2.3.0 --index-url https://download.pytorch.org/whl/cu121
pip install transformers==4.44.0
pip install peft==0.12.0          # PEFT : Parameter-Efficient Fine-Tuning
pip install trl==0.10.1           # TRL : Supervised Fine-Tuning Trainer
pip install bitsandbytes==0.43.3  # Quantification 4-bit et 8-bit
pip install datasets==2.20.0      # HuggingFace Datasets
pip install accelerate==0.33.0    # Gestion multi-GPU et CPU offloading
pip install evaluate rouge_score  # Metriques d'evaluation ROUGE
```

VRAM requise selon le modèle et le rang LoRA

Modèle	Paramètres	LoRA r=8 (bf16)	QLoRA r=8 (4bit)	QLoRA r=16 (4bit)
Phi-4 Mini Instruct	3,8B	~10 Go	~5 Go	~6 Go
Llama 3.2 8B Instruct	8B	~20 Go	~8 Go	~10 Go
Mistral 7B Instruct v0.3	7B	~18 Go	~7 Go	~9 Go
Llama 3 70B Instruct	70B	~160 Go	~48 Go	~52 Go

Pour les équipes qui souhaitent expérimenter sans GPU dédié, Google Colab Pro+ (A100 40 Go) ou RunPod (RTX 4090 24 Go) permettent de fine-tuner Llama 3 8B avec QLoRA à moins de 5 euros par run d'entraînement. Pour les déploiements en inférence locale après entraînement, consultez notre comparatif des LLM en local avec Ollama, LMStudio et vLLM.

Préparer son dataset d'entraînement

La qualité du dataset est le facteur le plus déterminant dans la réussite d'un fine-tuning. Un modèle entraîné sur 500 exemples de haute qualité surpasse systématiquement un modèle entraîné sur 10 000 exemples mal structurés ou incohérents. Le format recommandé en 2026 est l'instruction tuning : chaque exemple est un triplet (instruction, input optionnel, output attendu), formaté selon le chat template du modèle de base.

Format Instruction Tuning et Chat Templates

Deux chat templates dominent l'écosystème open source en 2026 :

Llama 3 (Meta) : utilise des tokens spéciaux `<|begin_of_text|>`, `<|start_header_id|>`, `<|end_header_id|>` et `<|eot_id|>` pour délimiter les rôles system, user et assistant.

ChatML (Mistral, OpenHermes) : utilise `<|im_start|>` et `<|im_end|>` comme délimiteurs de tour de conversation.

La bibliothèque `transformers` applique automatiquement le bon template via `tokenizer.apply_chat_template()` si le tokenizer du modèle de base le définit. Il est fortement recommandé d'utiliser cette méthode plutôt que de construire les templates manuellement pour éviter des artefacts d'entraînement.

```
 from datasets import Dataset

# Exemple de dataset pour classification d'alertes de securite
raw_data = [
    {
        "instruction": "Classe cette alerte de securite et explique ta decision.",
        "input": "Alert: Outbound connection to 185.220.101.45:443 from workstation WS-042. "
                 "Process: powershell.exe. Parent: winword.exe.",
        "output": (
            "Severite: CRITIQUE. Classification: Compromission initiale probable "
            "(T1566.001 - Phishing via macro Word). Indicateurs : powershell.exe lance "
            "par winword.exe (process injection typique), connexion sortante vers IP "
            "Tor exit node connue (185.220.101.45). "
            "Action : isolation immediate de WS-042, collecte forensique, "
            "analyse du document Word source."
        )
    },
    # Ajouter 200 a 500 exemples similaires pour un fine-tuning efficace
]

def format_instruction(example):
    """Formater au format Alpaca (compatible avec la plupart des tokenizers)."""
    if example.get("input"):
        text = (
            "### Instruction:\n" + example["instruction"] + "\n\n"
            "### Input:\n" + example["input"] + "\n\n"
            "### Response:\n" + example["output"]
        )
    else:
        text = (
            "### Instruction:\n" + example["instruction"] + "\n\n"
            "### Response:\n" + example["output"]
        )
    return {"text": text}

dataset = Dataset.from_list(raw_data)
dataset = dataset.map(format_instruction)

# Division train/test (80/20)
dataset = dataset.train_test_split(test_size=0.2, seed=42)
print(f"Train: {len(dataset['train'])} exemples")
print(f"Test: {len(dataset['test'])} exemples")
```

Pour un cas d'usage cybersécurité, les sources de données publiques incluent : les CTI reports publics de Mandiant et CrowdStrike, les writeups CTF sur GitHub, les rapports ANSSI publics, et des datasets HuggingFace spécialisés. Les embeddings générés par le modèle fine-tuné peuvent ensuite être utilisés dans un pipeline RAG — voir notre guide sur les embeddings vs tokens. Pour comprendre l'architecture d'indexation vectorielle qui en découle, consultez notre article sur l'indexation vectorielle.

Entraînement QLoRA pas à pas

Voici le code d'entraînement complet, commenté pour la production. Il utilise `BitsAndBytesConfig` pour la quantification 4-bit, `get_peft_model` de la bibliothèque PEFT pour injecter les adaptateurs LoRA dans le modèle, et `SFTTrainer` (Supervised Fine-Tuning Trainer) de TRL pour piloter la boucle d'entraînement.

```
 import torch
from transformers import (
    AutoTokenizer, AutoModelForCausalLM,
    BitsAndBytesConfig
)
from peft import LoraConfig, get_peft_model, prepare_model_for_kbit_training
from trl import SFTTrainer, SFTConfig

# 1. Configuration de la quantification 4-bit (QLoRA)
bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_quant_type="nf4",           # NormalFloat4 : optimal pour distributions gaussiennes
    bnb_4bit_compute_dtype=torch.bfloat16,  # Calculs en bf16 pour stabilite numerique
    bnb_4bit_use_double_quant=True       # Double quantification pour economiser ~512 Mo
)

# 2. Chargement du modele de base quantifie
MODEL_ID = "meta-llama/Meta-Llama-3.1-8B-Instruct"  # ou mistralai/Mistral-7B-Instruct-v0.3

tokenizer = AutoTokenizer.from_pretrained(MODEL_ID)
tokenizer.pad_token = tokenizer.eos_token  # Necessaire pour le padding lors du batching
tokenizer.padding_side = "right"           # Padding a droite pour les modeles causaux

model = AutoModelForCausalLM.from_pretrained(
    MODEL_ID,
    quantization_config=bnb_config,
    device_map="auto",       # Distribue automatiquement sur les GPUs disponibles
    trust_remote_code=False  # Securite : ne pas executer de code arbitraire du repo
)

# Prepare le modele pour l'entrainement en k-bit (cast des LayerNorm en fp32)
model = prepare_model_for_kbit_training(model)

# 3. Configuration LoRA
lora_config = LoraConfig(
    r=16,              # Rang : 8-32 est le range usuel. Plus grand = plus de parametres
    lora_alpha=32,     # Scale factor : souvent 2x r pour un bon equilibre
    target_modules=[   # Modules cibles : couches d'attention et MLP
        "q_proj", "k_proj", "v_proj", "o_proj",  # Self-attention
        "gate_proj", "up_proj", "down_proj"       # MLP Feed-Forward
    ],
    lora_dropout=0.05, # Dropout pour regularisation
    bias="none",       # Ne pas adapter les biais (economie de parametres)
```

```
    task_type="CAUSAL_LM"
)

model = get_peft_model(model, lora_config)
model.print_trainable_parameters()
# Sortie typique : trainable params: 83,886,080 || all params: 8,113,475,584 || 1.03%

# 4. Configuration de l'entrainement
training_args = SFTConfig(
    output_dir="./llama3-cyber-qlora",
    num_train_epochs=3,
    per_device_train_batch_size=4,
    gradient_accumulation_steps=4,      # Effective batch size = 4 x 4 = 16
    gradient_checkpointing=True,        # Reduit la VRAM en recalculant les activations
    optim="paged_adamw_32bit",          # Paged AdamW : gestion memoire des etats optimiseur
    learning_rate=2e-4,
    lr_scheduler_type="cosine",         # Decroissance cosinus pour meilleure convergence
    warmup_ratio=0.05,
    logging_steps=10,
    save_strategy="epoch",
    evaluation_strategy="epoch",
    bf16=True,
    max_grad_norm=0.3,                  # Gradient clipping pour stabiliser l'entrainement
    report_to="none",
    max_seq_length=2048,
    dataset_text_field="text",
    packing=False
)

# 5. Lancement de l'entrainement
trainer = SFTTrainer(
    model=model,
    train_dataset=dataset["train"],
    eval_dataset=dataset["test"],
    args=training_args,
)

trainer.train()
trainer.save_model("./llama3-cyber-qlora/final")
tokenizer.save_pretrained("./llama3-cyber-qlora/final")
print("Entrainement termine. Modele sauvegarde.")
```

La durée d'entraînement typique sur un RTX 4090 (24 Go) pour Llama 3 8B avec 500 exemples, 3 epochs, batch size effectif 16 est d'environ 45 minutes. Sur un A100 40 Go, le même

entraînement prend moins de 20 minutes. Les logs montrent la training loss et la validation loss à chaque epoch : surveiller une divergence entre les deux (overfitting) et réduire `num_train_epochs` ou ajouter du dropout si nécessaire.

Évaluation et merge des poids LoRA

Un modèle fine-tuné doit être évalué quantitativement avant tout déploiement en production. Deux métriques complémentaires sont utilisées : ROUGE pour les tâches de génération de texte (mesure le recouvrement de n-grammes entre sortie générée et référence), et la perplexité pour la mesure de la qualité générale du modèle sur le corpus de test. Après validation, le merge des poids LoRA produit un modèle autonome sans dépendance à PEFT.

```
 import evaluate
import torch
from transformers import pipeline
from peft import PeftModel
from transformers import AutoModelForCausalLM, AutoTokenizer

# --- Evaluation ROUGE sur le set de test ---
rouge = evaluate.load("rouge")

pipe = pipeline(
    "text-generation",
    model="./llama3-cyber-qlora/final",
    tokenizer=tokenizer,
    torch_dtype=torch.bfloat16,
    device_map="auto",
    max_new_tokens=512,
    temperature=0.1
)

predictions = []
references = []

for example in dataset["test"].select(range(50)):
    prompt = example["text"].split("### Response:")[0] + "### Response:\n"
    output = pipe(prompt)[0]["generated_text"]
    generated = output.split("### Response:")[-1].strip()
    expected = example["text"].split("### Response:")[-1].strip()
    predictions.append(generated)
    references.append(expected)

scores = rouge.compute(predictions=predictions, references=references)
print(f"ROUGE-1: {scores['rouge1']:.4f}")  # Recouvrement unigrams
print(f"ROUGE-2: {scores['rouge2']:.4f}")  # Recouvrement bigrams
print(f"ROUGE-L: {scores['rougeL']:.4f}")  # Plus longue sous-sequence commune

# --- Merge des poids LoRA dans le modele de base ---
MODEL_ID = "meta-llama/Meta-Llama-3.1-8B-Instruct"

base_model = AutoModelForCausalLM.from_pretrained(
    MODEL_ID,
    torch_dtype=torch.bfloat16,
    device_map="auto"
)
```

```
# Charger et fusionner les adaptateurs LoRA
peft_model = PeftModel.from_pretrained(base_model, "./llama3-cyber-qlora/final")
merged_model = peft_model.merge_and_unload()  # Fusionne BA dans W, supprime les adaptateurs

# Sauvegarder le modele merge (standard HuggingFace, sans dependance PEFT)
merged_model.save_pretrained("./llama3-cyber-final-merged", safe_serialization=True)
tokenizer.save_pretrained("./llama3-cyber-final-merged")
print("Merge termine. Modele pret pour la production.")
```

Le modèle mergé est un modèle HuggingFace standard, sans dépendance à PEFT. Il peut être converti en format GGUF pour Ollama ou llama.cpp, ou chargé directement avec vLLM pour l'inférence haute performance en production. Des scores ROUGE-L supérieurs à 0,4 indiquent généralement un fine-tuning réussi pour des tâches de génération structurée ; pour de la classification, préférer l'accuracy ou le F1-score comme métriques principales.

Cas d'usage métier — Cybersécurité

Le fine-tuning LoRA en cybersécurité s'applique à plusieurs cas d'usage à forte valeur où les LLMs généralistes montrent leurs limites : vocabulaire technique très spécialisé, formats de sortie stricts (JSON, STIX, XML), et contraintes de confidentialité qui imposent de ne pas envoyer les données vers des API cloud tierces.

Classification d'alertes SIEM

Un modèle fine-tuné sur un corpus d'alertes SIEM labellisées (vrai positif / faux positif / sévérité / TTP MITRE) peut automatiser le triage de niveau 1 avec une précision supérieure à 90 %, contre 70 à 75 % pour un LLM généraliste en zero-shot sur le même benchmark interne. L'avantage décisif du modèle fine-tuné est la cohérence du format de sortie JSON, parsable en quasi-totalité, là où un LLM généraliste varie parfois la structure de sa réponse. Ce type d'agent de triage s'intègre dans un workflow de détection plus large — voir notre analyse du Cyber Threat Landscape France 2026 et notre comparatif des solutions EDR/XDR 2025.

Génération de rapports de pentest

Un modèle fine-tuné sur un corpus de rapports de pentest anonymisés peut générer des sections standardisées (description de la vulnérabilité, impact, recommandation, références CVE/CWE) à partir d'une fiche de finding structurée. Les équipes qui utilisent cette approche rapportent un gain de temps de 40 à 60 % sur la rédaction. Pour les tests de sécurité cloud, consultez notre guide sur le pentest cloud AWS, Azure et GCP. Les méthodologies d'audit de GCP spécifiquement sont couvertes dans notre article sur l'audit de sécurité GCP.

Extraction d'IoC depuis des rapports CTI

Un modèle fine-tuné avec LoRA peut extraire automatiquement les indicateurs de compromission (IPs, domaines, hashes MD5/SHA256, CVE, TTPs MITRE) depuis des rapports de threat intelligence en texte libre. Fine-tuner un modèle d'encodeur (RoBERTa, CamemBERT pour le français) avec LoRA est encore plus accessible que les modèles génératifs : quelques centaines d'exemples annotés avec Label Studio suffisent pour atteindre des performances de production sur cette tâche de NER spécialisé.

FAQ — Fine-Tuning LoRA

Combien d'exemples faut-il pour un fine-tuning LoRA efficace ?

La règle empirique en 2026 est de commencer avec 200 à 500 exemples de haute qualité pour des tâches de formatage et de classification, et 1 000 à 5 000 exemples pour des tâches de génération complexes. La qualité prime sur la quantité : un dataset de 300 exemples parfaitement formatés et diversifiés surpasse systématiquement un dataset de 3 000 exemples bruités. Il est recommandé d'évaluer sur un set de validation dès 100 exemples d'entraînement, puis d'augmenter progressivement jusqu'à convergence de la validation loss.

Quelle est la différence de qualité entre LoRA et QLoRA ?

La différence de qualité entre LoRA en bfloat16 et QLoRA en 4-bit NF4 est généralement inférieure à 2 % sur les benchmarks standards comme MT-Bench ou MMLU. Pour des tâches très spécialisées sur un domaine étroit — classification d'alertes, génération de code de sécurité — la différence tend à s'estomper encore davantage car le fine-tuning compense en grande partie la perte de précision due à la quantification. QLoRA est recommandé par défaut car il permet d'utiliser des modèles plus grands avec le même budget GPU, ce qui compense largement la légère perte de précision.

Peut-on fine-tuner un modèle LoRA pour plusieurs tâches simultanément ?

Oui, via le multitask fine-tuning : le dataset contient des exemples de plusieurs tâches, chacune identifiée par un préfixe dans l'instruction. Le modèle apprend à switcher entre les tâches selon le contexte. Une alternative plus modulaire est d'entraîner des adaptateurs LoRA séparés par tâche et de les switcher dynamiquement via PEFT — cette approche est particulièrement utile quand les tâches ont des styles de sortie très différents (JSON structuré vs prose narrative, par exemple).

Comment éviter le catastrophic forgetting lors du fine-tuning LoRA ?

LoRA réduit intrinsèquement le catastrophic forgetting par rapport au full fine-tuning puisque les poids de base restent gelés. Néanmoins, un dataset très étroit peut trop spécialiser le modèle et dégrader ses capacités générales. Les mitigations recommandées sont : mélanger 10 à 20 % d'exemples généraux au dataset de fine-tuning, utiliser un faible learning rate (inférieur ou égal à 2e-4), surveiller les perplexités sur un set de tâches générales pendant l'entraînement, et utiliser des techniques comme EWC (Elastic Weight Consolidation) si la préservation des capacités générales est critique pour le cas d'usage.

À RETENIR

Points clés à retenir

LoRA réduit les paramètres entraînables à moins de 1 % en ajoutant des matrices de rang faible (W' = W + B×A) aux couches d'attention ; les poids du modèle de base restent gelés pendant tout l'entraînement.

QLoRA combine quantification 4-bit NF4 et LoRA pour fine-tuner des modèles de 7 à 8 milliards de paramètres sur un GPU de 8 Go de VRAM, avec une perte de qualité inférieure à 2 % par rapport au full fine-tuning.

Le stack de référence 2026 est : HuggingFace Transformers + PEFT + TRL (SFTTrainer) + BitsAndBytes — compatibles avec Llama 3, Mistral, Phi-4 et la quasi-totalité des modèles open source.

La qualité du dataset prime sur sa taille : 300 exemples soigneusement annotés suffisent pour des tâches de classification ou de formatage structuré en cybersécurité.

Après entraînement, `merge_and_unload()` fusionne les adaptateurs LoRA dans les poids de base, produisant un modèle standard déployable avec vLLM, Ollama ou llama.cpp sans dépendance à PEFT.

Références et ressources

Hugging Face — Transformers documentation

arXiv — Recherches récentes sur les LLMs

NIST AI — Ressources Intelligence Artificielle

