Fine-Tuning von Large Language Models: Ein praktischer Leitfaden mit Python und Ollama

Artikel-Übersicht

Dieser Leitfaden zeigt dir, wie du Large Language Models mit Python fine-tunest und anschließend in Ollama für lokales Deployment integrierst. Wir beginnen mit den Grundkonzepten des Fine-Tunings und erklären die Vorteile gegenüber dem Training von Grund auf. Du lernst die drei Hauptanwendungsfälle kennen: konsistente Formatierung, domänenspezifisches Wissen und Kostenoptimierung. Im praktischen Teil behandeln wir zunächst die Datenvorbereitung und -formatierung mit konkreten Code-Beispielen. Danach folgt die Installation und Konfiguration des Phi-3-mini Modells mit der Unsloth-Bibliothek. Ein wichtiger Baustein ist die Verwendung von LoRA-Adaptern, die effizientes Training ermöglichen, indem nur etwa 3% der Modellparameter angepasst werden. Der Artikel führt dich durch den kompletten Trainingsprozess, vergleicht die Performance zwischen GPU und CPU, und zeigt Best Practices für die Modellvalidierung. Anschließend exportieren wir das trainierte Modell ins GGUF-Format und integrieren es in Ollama, sodass es lokal auf deinem Rechner läuft. Am Ende findest du praktische Tipps für eigene Projekte sowie einen Link zum vollständigen Code-Repository mit allen Beispielen und Konfigurationsdateien.

Einleitung: Was ist Fine-Tuning und warum ist es wichtig?

Heute zeige ich dir, wie du Large Language Models (LLMs) in Python fine-tunen und anschließend in Ollama nutzen kannst. Ja, du hast richtig gelesen – Python! Ich musste diesmal von meiner geliebten Programmiersprache C# abweichen, da sich Python im Bereich Machine Learning und KI einfach als die beste Wahl etabliert hat. Die verfügbaren Bibliotheken und die Community-Unterstützung sind hier unschlagbar.

Aber was ist Fine-Tuning überhaupt? Stell dir vor, du stellst einen erfahrenen Koch ein, der bereits alle Grundtechniken beherrscht, und bringst ihm dann die speziellen Rezepte deines Restaurants bei. Genau das machen wir beim Fine-Tuning: Wir nehmen ein vortrainiertes Modell wie GPT oder Claude, das bereits die menschliche Sprache versteht, und trainieren es auf unsere spezifischen Aufgaben.

Der entscheidende Unterschied zum Training von Grund auf? Du brauchst weder Millionen von Beispielen noch monatelange Trainingszeiten. Mit Fine-Tuning kommst du oft schon mit ein paar hundert oder tausend Beispielen aus und bist in Minuten oder Stunden fertig.

Fine-Tuning vs. Training von Grund auf: Die Vorteile erklärt

Lass uns die Unterschiede zwischen Fine-Tuning und dem Training von Grund auf genauer betrachten. Beim Parameter-Tuning justierst du nur Einstellungen wie Temperature oder Top-K – das ist wie die Lautstärke am Radio zu regeln. Fine-Tuning hingegen ist, als würdest du deinem Auto beibringen, in einer völlig neuen Nachbarschaft zu fahren.

Die Vorteile des Fine-Tunings sind beeindruckend:

- **Ressourcen**: Statt Millionen von Beispielen reichen oft Hunderte bis Tausende
- **Zeit**: Training in Minuten oder Stunden statt Wochen oder Monaten

- **Kosten**: Deutlich günstiger durch geringeren Rechenaufwand
- **Qualität**: Du profitierst vom Vorwissen des Modells

Es gibt allerdings einen wichtigen Punkt zu beachten: Durch das Fine-Tuning wird dein Modell zwar besser bei deiner spezifischen Aufgabe, kann aber bei allgemeinen Aufgaben schlechter werden. Das ist ein Trade-off, den du bewusst eingehen musst.

Wann sollte man Fine-Tuning einsetzen? Drei Hauptszenarien

Fine-Tuning ist nicht immer die richtige Lösung. Es gibt drei Hauptszenarien, in denen es besonders sinnvoll ist:

1. Konsistente Formatierung und Stil

Wenn du ein Modell brauchst, das immer in einem bestimmten Format antwortet – etwa strukturierte JSON-Ausgaben für deine API oder Berichte in einem spezifischen Stil. Prompting allein reicht hier oft nicht aus, um wirklich konsistente Ergebnisse zu erzielen.

2. Domänenspezifisches Wissen

Du arbeitest mit hochspezialisierten Daten, die das Modell noch nie gesehen hat? Ob fortgeschrittene medizinische Berichte, interne Kundenservice-Logs oder proprietäre Geschäftsprozesse – Fine-Tuning macht dein Modell zum Experten in deinem Fachgebiet.

3. Kostenoptimierung

Statt ein riesiges, teures Modell zu nutzen, kannst du ein kleineres Modell fine-tunen. Das spart nicht nur Geld bei der Inferenz, sondern macht deine Anwendung auch schneller und skalierbarer.

Vorbereitung: Datensammlung und -formatierung mit Code-Beispielen

Der wichtigste Schritt beim Fine-Tuning ist die Datenvorbereitung. Schlechte Daten führen zu einem schlechten Modell – so einfach ist das. Lass uns anschauen, wie du deine Daten richtig vorbereitest.

Das richtige Datenformat

Für unser Beispiel verwenden wir eine HTML-Extraktionsaufgabe. Die Daten sollten als JSON-Datei strukturiert sein:

```
{
  "input": "<div class='produkt'><h2>Samsung Galaxy Tab</h2><span
class='preis'>€ 599</span><span class='kategorie'>Tablet</span><span
class='marke'>Samsung</span></div>",
  "output": {
    "name": "Samsung Galaxy Tab",
    "preis": "€ 599",
    "kategorie": "Tablet",
    "marke": "Samsung"
  }
}
```

Prompt-Formatierung im Code

Schauen wir uns an, wie unser Python-Code die Daten für das Training formatiert:

```
def formatiere_prompt(self, beispiel):
    """
    Formatiert ein Beispiel für das Training

    Args:
        beispiel: Dictionary mit 'input' und 'output'

    Returns:
        Formatierter String für das Training
    """
    return f"### Eingabe: {beispiel['input']}\n### Ausgabe:
{json.dumps(beispiel['output'], ensure_ascii=False)}<|endoftext|>"
```

Diese Methode aus der `PhiFineTuner`-Klasse zeigt, wie wir die Rohdaten in ein Format bringen, das das Modell versteht. Wichtig sind:

- Klare Trennung zwischen Eingabe und Ausgabe
- Das `<|endoftext|>` Token am Ende, um dem Modell das Ende zu signalisieren
- `ensure_ascii=False` für korrekte Darstellung von Umlauten

Wie viele Daten brauchst du?

Die Anzahl hängt von deiner Aufgabe ab:

- **Einfache Formatierungsaufgaben**: 100-500 Beispiele
- **Domänenspezifisches Wissen**: 500-5000 Beispiele
- **Komplexe Aufgaben**: 5000+ Beispiele

In unserem Beispiel verwenden wir 500 Beispiele für die HTML-Extraktion – ein guter Mittelweg für diese Art von Aufgabe.

Das Phi-3-mini Modell: Installation und Setup mit Unsloth

Für unser Fine-Tuning verwenden wir das Phi-3-mini Modell mit Unsloth. Warum diese Kombination? Phi-3-mini ist klein genug für schnelles Training, aber leistungsstark genug für ernsthafte Aufgaben. Unsloth macht das Training extrem schnell und speichereffizient.

Installation der Abhängigkeiten

Zuerst müssen wir die notwendigen Pakete installieren:

```
pip install unsloth datasets transformers trl torch
```

GPU-Erkennung und Setup

Unser Code prüft automatisch, ob eine GPU verfügbar ist:

```
class PhiFineTuner:
    def __init__(self, datei_pfad="produktdaten_deutsch.json",
ausgabe_ordner="ausgabe"):
        # ... andere Initialisierungen ...

        # GPU-Verfügbarkeit prüfen
        self.gpu_verfuegbar = torch.cuda.is_available()
        if self.gpu_verfuegbar:
            print(f"✓ GPU erkannt: {torch.cuda.get_device_name(0)}")
            print(f"  CUDA Version: {torch.version.cuda}")
        else:
            print("⚠ Keine GPU erkannt - Training läuft auf CPU
(langsamer)")
```

Modell laden

Das Laden des Modells ist mit Unsloth denkbar einfach:

```
def lade_modell(self, modell_name="unsloth/Phi-3-mini-4k-instruct-bnb-
4bit"):
    """Lädt das Basismodell und den Tokenizer"""
    print(f"\nLade Modell: {modell_name}")

    self.model, self.tokenizer = FastLanguageModel.from_pretrained(
        model_name=modell_name,
        max_seq_length=2048,
        dtype=None,  # Automatische Erkennung
        load_in_4bit=True,  # 4-bit Quantisierung für Speichereffizienz
    )
```

Die 4-bit Quantisierung reduziert den Speicherbedarf drastisch, ohne die Qualität merklich zu beeinträchtigen. Perfekt für unsere Zwecke!

Alternative: Google Colab

Falls du keine leistungsstarke GPU hast, empfehle ich Google Colab. Dort bekommst du kostenlos Zugriff auf eine Tesla T4 GPU. Der Workflow ist identisch – du lädst einfach dein Notebook hoch und los geht's.

LoRA-Adapter verstehen: Effizientes Fine-Tuning erklärt

Jetzt kommt der Clou: LoRA (Low-Rank Adaptation). Statt das gesamte Modell zu trainieren, was Unmengen an Speicher und Zeit kosten würde, trainieren wir nur kleine "Adapter". Das ist genial!

Was macht LoRA?

LoRA fügt trainierbare Rang-Dekompositionsmatrizen in die Transformer-Schichten ein. Klingt kompliziert? Ist es auch, aber die Vorteile sind klar:

- **Nur ~3% der Parameter werden trainiert** (bei unserem Modell 119M von 3.9B)
- **Drastisch reduzierter Speicherbedarf**
- **Schnelleres Training bei vergleichbaren Ergebnissen**

Die Konfiguration im Code

Schauen wir uns die LoRA-Konfiguration in unserem Code an:

```
def konfiguriere_lora(self):
    """Konfiguriert LoRA-Adapter für effizientes Fine-Tuning"""
    print("\nKonfiguriere LoRA-Adapter...")

    self.model = FastLanguageModel.get_peft_model(
        self.model,
        r=64,  # LoRA Rang - 64 ist ein guter Kompromiss
        target_modules=[  # Diese Module werden angepasst
            "q_proj", "k_proj", "v_proj", "o_proj",  # Attention-Layer
            "gate_proj", "up_proj", "down_proj",  # MLP-Layer
        ],
        lora_alpha=128,  # Skalierungsfaktor (2x Rang ist Standard)
        lora_dropout=0,  # Kein Dropout für maximale Leistung
        bias="none",  # Keine Bias-Anpassung nötig
        use_gradient_checkpointing="unsloth",  # Speicher-optimiert
        random_state=3407,  # Für reproduzierbare Ergebnisse
    )
```

Die wichtigsten Parameter erklärt:

- **r (Rang)**: Bestimmt die Größe der LoRA-Matrizen. Höherer Rang = mehr Kapazität, aber auch mehr Speicher
- **target_modules**: Welche Teile des Modells werden angepasst? Wir fokussieren uns auf Attention und MLP-Layer
- **lora_alpha**: Skalierungsfaktor für die Lernrate. Faustregel: 2x Rang
- **use_gradient_checkpointing**: Tauscht Rechenzeit gegen Speicher – essentiell für große Modelle

Mit dieser Konfiguration trainieren wir effizient nur die relevanten Teile des Modells.

Der Trainingsprozess: Schritt-für-Schritt mit Python-Code

Jetzt wird's ernst – wir trainieren das Modell! Der Trainingsprozess ist das Herzstück des Fine-Tunings. Lass uns durchgehen, wie das in unserem Code funktioniert.

Der SFTTrainer im Einsatz

```
def trainiere(self, epochen=3, batch_groesse=2, lernrate=2e-4):
    """Führt das Fine-Tuning durch"""
```

```
    print(f"\nStarte Training:")
    print(f"- Epochen: {epochen}")
    print(f"- Batch-Größe: {batch_groesse}")
    print(f"- Lernrate: {lernrate}")

    # Trainer konfigurieren
    trainer = SFTTrainer(
        model=self.model,
        tokenizer=self.tokenizer,
        train_dataset=self.dataset,
        dataset_text_field="text",
        max_seq_length=2048,
        dataset_num_proc=2,
        args=TrainingArguments(
            per_device_train_batch_size=batch_groesse,
            gradient_accumulation_steps=4,  # Effektive Batch-Größe = 2 *
4 = 8
            warmup_steps=10,
            num_train_epochs=epochen,
            learning_rate=lernrate,
            fp16=not torch.cuda.is_bf16_supported(),
            bf16=torch.cuda.is_bf16_supported(),
            logging_steps=25,
            optim="adamw_8bit",  # Speicher-optimierter Optimizer
            weight_decay=0.01,
            lr_scheduler_type="linear",
            output_dir=self.ausgabe_ordner,
            save_strategy="epoch",
            save_total_limit=2,
        ),
    )

    # Training durchführen
    trainer_stats = trainer.train()
```

Die wichtigsten Trainingsparameter:

- **gradient_accumulation_steps**: Simuliert größere Batches ohne mehr Speicher zu brauchen
- **warmup_steps**: Langsames Hochfahren der Lernrate für stabileres Training
- **fp16/bf16**: Automatische Auswahl der besten Präzision für deine Hardware
- **adamw_8bit**: 8-bit Optimizer spart massiv Speicher
- **save_strategy="epoch"**: Speichert nach jeder Epoche einen Checkpoint

Was passiert während des Trainings?

1. **Datenladen**: Batches werden aus deinem Dataset gezogen
2. **Forward Pass**: Das Modell macht Vorhersagen
3. **Loss-Berechnung**: Wie falsch lag das Modell?
4. **Backward Pass**: Gradienten werden berechnet
5. **Optimizer-Step**: Gewichte werden angepasst

Bei 500 Beispielen und 3 Epochen dauert das Training auf einer T4 GPU etwa 10-15 Minuten. Auf einer CPU... nun ja, geh lieber einen Kaffee trinken. Oder zwei.

GPU vs. CPU: Performance-Unterschiede und Optimierungen

Der Unterschied zwischen GPU und CPU beim Fine-Tuning ist wie der zwischen einem Formel-1-Wagen und einem Fahrrad. Beide bringen dich ans Ziel, aber die Geschwindigkeit... nun ja.

Performance-Vergleich

Bei unserem Beispiel mit 500 Datensätzen:

- **GPU (Tesla T4)**: ~10-15 Minuten
- **GPU (RTX 4090)**: ~5-7 Minuten
- **CPU (moderne Hardware)**: 2-4 Stunden

Der Code passt sich automatisch an deine Hardware an:

```
# Automatische Hardware-Erkennung
fp16=not torch.cuda.is_bf16_supported() if self.gpu_verfuegbar else False,
bf16=torch.cuda.is_bf16_supported() if self.gpu_verfuegbar else False,
```

Speicher-Optimierungen

Wenn du wenig GPU-Speicher hast, helfen diese Tricks:

1. **Kleinere Batch-Größe**: Reduziere `batch_groesse` auf 1
2. **Gradient Accumulation erhöhen**: Setze `gradient_accumulation_steps` auf 8 oder 16
3. **Gradient Checkpointing**: Bereits aktiviert mit `use_gradient_checkpointing="unsloth"`
4. **8-bit Optimizer**: Nutzen wir bereits mit `optim="adamw_8bit"`

Google Colab als Rettung

Keine GPU? Kein Problem! Google Colab bietet kostenlos Zugriff auf GPUs. Die T4 reicht für die meisten Fine-Tuning-Aufgaben völlig aus. Upload dein Notebook, verbinde dich mit einer GPU-Runtime und leg los!

Modell testen und validieren: Best Practices

Nach dem Training kommt der spannende Moment: Funktioniert das Modell? Lass uns durchgehen, wie du dein fine-getuntes Modell richtig testest.

Der Test-Code

```
def teste_modell(self, test_eingabe=None):
    """Testet das fine-getunte Modell"""
    print("\nTeste fine-getuntes Modell...")

    # Für Inferenz optimieren
    FastLanguageModel.for_inference(self.model)
```

```
    # Standard-Testbeispiel wenn keine Eingabe gegeben
    if test_eingabe is None:
        test_eingabe = """Extrahiere die Produktinformationen:
<div class='produkt'><h2>Samsung Galaxy Tab</h2><span class='preis'>€
599</span>
<span class='kategorie'>Tablet</span><span class='marke'>Samsung</span>
</div>"""

    # Chat-Template anwenden
    nachrichten = [
        {"role": "user", "content": test_eingabe}
    ]

    inputs = self.tokenizer.apply_chat_template(
        nachrichten,
        tokenize=True,
        add_generation_prompt=True,
        return_tensors="pt",
    )

    # Auf GPU verschieben wenn verfügbar
    if self.gpu_verfuegbar:
        inputs = inputs.to("cuda")

    # Antwort generieren
    outputs = self.model.generate(
        input_ids=inputs,
        max_new_tokens=256,
        use_cache=True,
        temperature=0.7,
        do_sample=True,
        top_p=0.9,
    )
```

Best Practices beim Testen

1. **Vielfältige Testfälle**: Teste mit verschiedenen HTML-Strukturen, nicht nur mit den Trainingsbeispielen
2. **Edge Cases**: Was passiert bei fehlenden Tags? Bei verschachtelten Strukturen?
3. **Konsistenz prüfen**: Führe denselben Test mehrmals aus – sind die Ergebnisse konsistent?
4. **Temperature anpassen**: Bei `temperature=0.7` bekommst du kreativere Antworten. Für konsistente Extraktion probiere `0.1–0.3`

Warnsignale

- **Overfitting**: Das Modell gibt bei neuen Beispielen nur Trainingsantworten zurück
- **Inkonsistenz**: Stark variierende Ausgaben bei gleicher Eingabe
- **Format-Fehler**: JSON-Struktur wird nicht eingehalten

Wenn du diese Probleme siehst, brauchst du möglicherweise mehr oder vielfältigere Trainingsdaten.

GGUF-Export: Das Modell für Ollama vorbereiten

Jetzt kommt der spannende Teil: Wir exportieren unser Modell im GGUF-Format, damit es mit Ollama läuft. GGUF ist das Standardformat für quantisierte Modelle und perfekt für lokale Deployment.

Der Export-Code

```
def exportiere_gguf(self, quantisierung="q4_k_m"):
    """
    Exportiert das Modell im GGUF-Format für Ollama

    Args:
        quantisierung: Quantisierungsmethode
            - q4_k_m: 4-bit Quantisierung (empfohlen, ~2.2GB)
            - q5_k_m: 5-bit Quantisierung (~2.7GB)
            - q8_0: 8-bit Quantisierung (~4GB)
    """
    print(f"\nExportiere Modell als GGUF (Quantisierung:
{quantisierung})...")

    gguf_ordner = os.path.join(self.ausgabe_ordner, "gguf_modell")
    self.model.save_pretrained_gguf(
        gguf_ordner,
        self.tokenizer,
        quantization_method=quantisierung
    )
```

Quantisierungsoptionen

Die Wahl der Quantisierung ist ein Trade-off zwischen Größe und Qualität:

- **q4_k_m** (4-bit): Beste Balance zwischen Größe (~2.2GB) und Qualität. Meine Empfehlung!
- **q5_k_m** (5-bit): Etwas größer (~2.7GB), minimal bessere Qualität
- **q8_0** (8-bit): Größte Datei (~4GB), beste Qualität

Voraussetzungen für den Export

Für den GGUF-Export brauchst du:

1. **llama.cpp**: Muss gebaut sein (mit cmake und make)
2. **mistral_common**: `pip install mistral_common`

Der Export erstellt zwei Dateien:

- `unsloth.BF16.gguf`: Volle Präzision (~7.2GB)
- `unsloth.Q4_K_M.gguf`: Quantisierte Version (~2.2GB)

Export-Dauer

Der Export kann dauern – bei unserem Modell etwa 15-25 Minuten. Der Prozess:

1. Konvertierung ins GGUF-Format
2. Quantisierung der Gewichte
3. Speichern der finalen Datei

Tipp: Die quantisierte Version reicht für die meisten Anwendungsfälle völlig aus. Die volle Präzision brauchst du nur für spezielle Fälle.

Integration in Ollama: Modelfile erstellen und ausführen

Der finale Schritt: Wir bringen unser Modell in Ollama zum Laufen! Dafür brauchen wir ein Modelfile, das Ollama sagt, wie es mit unserem Modell umgehen soll.

Schritt 1: Modelfile erstellen

Erstelle eine Datei namens `Modelfile` (genau so, mit großem M):

```
# Terminal-Befehle
cd ~/Downloads
mkdir ollama-test
cd ollama-test
# Kopiere deine GGUF-Datei hierher
touch Modelfile
nano Modelfile
```

Schritt 2: Modelfile-Inhalt

Füge folgenden Inhalt ein:

```
FROM ./unsloth.Q4_K_M.gguf

PARAMETER top_p 0.9
PARAMETER temperature 0.7
PARAMETER stop "<|endoftext|>"

TEMPLATE """{{ if .System }}{{ .System }}{{ end }}
{{ if .Prompt }}User: {{ .Prompt }}{{ end }}
Assistant: """

SYSTEM "Du bist ein hilfreicher KI-Assistent."
```

Schritt 3: Modell zu Ollama hinzufügen

```
# Modell erstellen
ollama create html-modell -f Modelfile

# Prüfen ob es da ist
ollama list
```

```
# Modell ausführen
ollama run html-modell
```

Die Modelfile-Parameter erklärt

- **FROM**: Pfad zu deiner GGUF-Datei
- **PARAMETER**: Steuerung der Generierung
  - `top_p`: Nucleus Sampling (0.9 = top 90% der Wahrscheinlichkeiten)
  - `temperature`: Kreativität (0.7 = ausgewogen)
  - `stop`: Wann soll die Generierung stoppen?
- **TEMPLATE**: Wie werden Prompts formatiert?
- **SYSTEM**: Standard-Systemnachricht

Testen in Ollama

Jetzt kannst du dein Modell testen:

```
ollama run html-modell

>>> Extrahiere: <div class='artikel'><h1>iPhone 15</h1><span
class='preis'>999€</span></div>
```

Und voilà – dein fine-getuntes Modell läuft lokal in Ollama! Du kannst es jetzt auch aus Python oder anderen Anwendungen heraus nutzen.

Zusammenfassung und praktische Tipps

Geschafft! Du hast erfolgreich ein LLM fine-getuned und in Ollama zum Laufen gebracht. Lass uns die wichtigsten Learnings zusammenfassen und ich gebe dir noch ein paar praktische Tipps mit auf den Weg.

Was wir gelernt haben

1. **Fine-Tuning ist effizient**: Mit nur wenigen hundert Beispielen kannst du beeindruckende Ergebnisse erzielen
2. **LoRA macht's möglich**: Nur 3% der Parameter trainieren und trotzdem top Ergebnisse
3. **Hardware matters**: Eine GPU macht den Unterschied zwischen Minuten und Stunden
4. **GGUF für lokales Deployment**: Quantisierung spart Speicher ohne große Qualitätsverluste

Praktische Tipps für deine Projekte

Datenqualität über Quantität

- Lieber 200 perfekte Beispiele als 2000 mittelmäßige
- Vielfalt ist wichtig: Decke Edge Cases ab
- Konsistenz im Format ist essentiell

Performance-Optimierung

- Starte mit kleinen Modellen (Phi-3-mini ist perfekt zum Lernen)
- Nutze Google Colab wenn du keine GPU hast
- 4-bit Quantisierung (q4_k_m) ist meist die beste Wahl

Häufige Fehler vermeiden

- Overfitting: Mehr Epochen ≠ besseres Modell
- Zu hohe Lernrate: Kann das Training instabil machen
- Falsche Formatierung: Achte auf konsistente Prompt-Templates

Nächste Schritte

1. **Experimentiere mit verschiedenen Modellen**: Llama, Mistral, etc.
2. **Erweitere deine Datensätze**: Je spezifischer, desto besser
3. **Automatisiere den Workflow**: Scripte für Training und Deployment
4. **Integriere in Anwendungen**: Nutze die Ollama API

Ressourcen

- Unsloth Dokumentation
- Ollama API Referenz
- Google Colab für GPU-Zugriff
- Der vollständige Code aus diesem Tutorial

Fine-Tuning öffnet eine Welt voller Möglichkeiten. Du kannst Modelle für deine spezifischen Bedürfnisse anpassen, ohne die Bank zu sprengen oder monatelang zu trainieren. Also, worauf wartest du? Leg los und erschaffe dein eigenes, spezialisiertes LLM!

Viel Erfolg beim Fine-Tuning! 🚀

Code Repository

Den vollständigen Code zum Fine-Tuning mit Phi-3-mini findest du im GitHub-Repository:

https://github.com/ChristophKind/LargeLanguageModelsPublic

Dort kannst du:

- Den kompletten Python-Code (`finetuning.py`) herunterladen und direkt ausprobieren
- Die Beispiel-Trainingsdaten für HTML-Extraktion studieren
- Das vollständige Setup mit allen Konfigurationen nachvollziehen
- Eigene Anpassungen für deine spezifischen Use Cases vornehmen

Das Repository enthält alle notwendigen Dateien inklusive requirements.txt und eine detaillierte README mit Installationsanweisungen. Perfekt als Startpunkt für deine eigenen Fine-Tuning-Projekte!