**SADE CPU GÜCÜ İLE MODEL EĞİTİMİ İŞLEMİ NASIL YAPILIR**

**LORA İLE**

# %%capture  
# Gerekirse aşağıdaki satırların başındaki yorumları kaldırarak pip install yapabilirsiniz:  
#  
# pip3 install torch torchvision torchaudio --extra-index-url https://download.pytorch.org/whl/cpu  
# pip3 install transformers datasets peft trl accelerate  
# pip3 install huggingface\_hub  
  
import os  
import torch  
from datasets import load\_dataset  
from transformers import (  
 AutoTokenizer,  
 AutoModelForCausalLM,  
 DataCollatorForLanguageModeling,  
 TrainingArguments,  
)  
from peft import get\_peft\_model, LoraConfig, TaskType  
from trl import SFTTrainer  
  
# -----------------------------------------------------------  
# 1. Genel Ayarlar (CPU-Only)  
# -----------------------------------------------------------  
device = torch.device("cpu")  
print("[MODEL İNDİRİLİYOR]")  
MODEL\_NAME = "microsoft/phi-4"  
MAX\_SEQ\_LENGTH = 2048  
  
# -----------------------------------------------------------  
# 2. Tokenizer ve Modeli İndirme (CPU-Only)  
# -----------------------------------------------------------  
tokenizer = AutoTokenizer.from\_pretrained(MODEL\_NAME)  
model = AutoModelForCausalLM.from\_pretrained(MODEL\_NAME)  
model.to(device)  
  
# -----------------------------------------------------------  
# 3. LoRA (PEFT) Ayarları  
# -----------------------------------------------------------  
lora\_config = LoraConfig(  
 task\_type=TaskType.CAUSAL\_LM,  
 inference\_mode=False,  
 r=16, # CPU’da 8 veya 4 de tercih edilebilir  
 lora\_alpha=16,  
 lora\_dropout=0.0,  
 target\_modules=["q\_proj", "v\_proj", "k\_proj", "o\_proj"],  
)  
model = get\_peft\_model(model, lora\_config)  
model.to(device)  
  
# -----------------------------------------------------------  
# 4. Dataset’i Yükleme ve Tokenizasyon  
# -----------------------------------------------------------  
print("[VERİ SETİ İNECEK]")  
raw\_dataset = load\_dataset("mrkswe/llmEndpointDatasetConversation\_2", split="train")  
print("[VERİ SETİ İNDİRİLDİ]")  
  
import json  
  
def parse\_conversations(example):  
 example["conversations"] = json.loads(example["conversations"])  
 return example  
  
raw\_dataset = raw\_dataset.map(parse\_conversations, remove\_columns=[])  
  
def convert\_to\_text(examples):  
 texts = []  
 for conv in examples["conversations"]:  
 single\_text = ""  
 for turn in conv:  
 speaker, txt = turn  
 if speaker.lower() in ["user", "human"]:  
 single\_text += f"<s>user: {txt}\n"  
 else:  
 single\_text += f"assistant: {txt}</s>\n"  
 texts.append(single\_text)  
 return {"text": texts}  
  
dataset\_with\_text = raw\_dataset.map(convert\_to\_text, batched=True, remove\_columns=["conversations"])  
  
def tokenize\_fn(examples):  
 return tokenizer(  
 examples["text"],  
 truncation=True,  
 padding="max\_length",  
 max\_length=MAX\_SEQ\_LENGTH,  
 )  
  
tokenized\_dataset = dataset\_with\_text.map(tokenize\_fn, batched=True)  
tokenized\_dataset.set\_format(type="torch", columns=["input\_ids", "attention\_mask"])  
  
# -----------------------------------------------------------  
# 5. Data Collator (Causal LM için)  
# -----------------------------------------------------------  
data\_collator = DataCollatorForLanguageModeling(  
 tokenizer=tokenizer,  
 mlm=False, # “mlm=False” deyince otoregresif (causal) moda geçer  
)  
  
# -----------------------------------------------------------  
# 6. TrainingArguments (CPU-Only)  
# -----------------------------------------------------------  
print("[HİPERPARAMETRELER HAZIRLANIYOR]")  
training\_args = TrainingArguments(  
 output\_dir="./outputs\_cpu",  
 per\_device\_train\_batch\_size=1, # CPU RAM’i kısıtlıysa küçük batch  
 gradient\_accumulation\_steps=4, # Efektif batch size = 1\*4 = 4  
 warmup\_steps=5,  
 max\_steps=30, # Örnek amaçlı kısa eğitim  
 learning\_rate=2e-4,  
 logging\_steps=1,  
 fp16=False,  
 bf16=False,  
 optim="adamw\_torch", # CPU uyumlu optimizer  
 weight\_decay=0.01,  
 lr\_scheduler\_type="linear",  
 seed=3407,  
 no\_cuda=True, # CPU-only modunu zorla  
 report\_to="none",  
)  
  
# -----------------------------------------------------------  
# 7. Eğitim Başlatma  
# -----------------------------------------------------------  
print("[EĞİTİM BAŞLIYOR]")  
trainer = SFTTrainer(  
 model=model,  
 train\_dataset=tokenized\_dataset,  
 data\_collator=data\_collator,  
 args=training\_args,  
)  
  
trainer\_stats = trainer.train()  
  
# -----------------------------------------------------------  
# 8. Modeli Cihaza Kaydetme  
# -----------------------------------------------------------  
print("[MODELİ CİHAZA KAYDET]")  
save\_dir = "./fine\_tuned\_cpu\_model"  
os.makedirs(save\_dir, exist\_ok=True)  
model.save\_pretrained(save\_dir)  
tokenizer.save\_pretrained(save\_dir)  
  
# -----------------------------------------------------------  
# 9. Hugging Face Hub’a Yükleme (Opsiyonel)  
# -----------------------------------------------------------  
print("[MODELİ HUGGINGFACE’E KAYDET]")  
from huggingface\_hub import HfApi  
api = HfApi()  
api.upload\_folder(  
 folder\_path=save\_dir,  
 path\_in\_repo=".",  
 repo\_id="mrkswe/llmPhi4Try\_CPU", # Kendi kullanıcıAdınız/modelAdınız  
 repo\_type="model",  
 token="hf\_FvoGutuzdtzKLpzLORnzZsQlnXnAqWvEFa"  
)  
  
print("Fine‑tuning tamamlandı. Model ve tokenizer klasöre kaydedildi:", save\_dir)

AÇIKLAMALAR:

**❓ 1. Neden sadece adapter\_model.safetensors dosyası var ve neden bu kadar küçük?**

Sen sadece **LoRA adaptörleri** ile fine-tuning yapıyorsun, yani:

* Ana model (phi-4, ~1.8B parametre) **değiştirilmiyor**.
* Sadece birkaç milyon parametre içeren **LoRA katmanları** eğitiliyor ve kaydediliyor.

Dolayısıyla:

* adapter\_model.safetensors dosyası sadece bu küçük farkları içeriyor.
* Bu yüzden boyutu küçük: 26.2 MB (gayet normal).

💡 **Not:** Eğer ana modeli de değiştirmek isteseydin (full fine-tuning), 6–7 GB gibi büyük bir dosya oluşurdu.

### 🧪 ****2. Eğittiğin modeli nasıl test edebilirsin?****

Eğittiğin model bir **LoRA adaptörüdür**, dolayısıyla test ederken:

#### ✅ Yüklemen gerekenler:

* **Base model**: "microsoft/phi-4"
* **LoRA adaptörü**: "mrkswe/llmPhi4Try\_CPU" (senin Hugging Face'teki modelin)

#### 🔬 Test Kodu (inference):

import torch  
from transformers import AutoTokenizer, AutoModelForCausalLM  
from peft import PeftModel, PeftConfig  
  
# 1. LoRA yapılandırmasını yükle  
adapter\_model\_id = "mrkswe/llmPhi4Try\_CPU"  
config = PeftConfig.from\_pretrained(adapter\_model\_id)  
  
# 2. Base modeli ve tokenizer'ı yükle  
base\_model = AutoModelForCausalLM.from\_pretrained(config.base\_model\_name\_or\_path)  
tokenizer = AutoTokenizer.from\_pretrained(config.base\_model\_name\_or\_path)  
  
# 3. LoRA adaptörünü base model üzerine uygula  
model = PeftModel.from\_pretrained(base\_model, adapter\_model\_id)  
model.eval() # inference için  
  
# 4. Test metni  
input\_text = "<s>user: JSON filter formatı nasıl olmalı?\nassistant:"  
inputs = tokenizer(input\_text, return\_tensors="pt")  
  
# 5. Cevap üret  
with torch.no\_grad():  
 outputs = model.generate(  
 input\_ids=inputs["input\_ids"],  
 attention\_mask=inputs["attention\_mask"],  
 max\_new\_tokens=150,  
 do\_sample=True,  
 top\_p=0.95,  
 temperature=0.7,  
 eos\_token\_id=tokenizer.eos\_token\_id  
 )  
  
# 6. Cevabı yazdır  
print(tokenizer.decode(outputs[0], skip\_special\_tokens=True))

**LORA NEDİR? NASIL ÇALIŞIR?**

LoRA (Low-Rank Adaptation of Large Language Models), büyük dil modellerini (LLM) **daha hızlı, daha verimli ve daha az kaynak kullanarak** fine-tune etmek için geliştirilmiş çok akıllıca bir yöntemdir.

LoRA, bir modelin **tüm ağırlıklarını (parametrelerini) güncellemek yerine**, sadece **daha küçük “adaptasyon katmanları” ekleyerek** eğitim yapar.

Büyük bir matris yerine, onun düşük dereceli (low-rank) bir yaklaşımıyla güncelleme yapmak.

Formülsel olarak:

Wadapted=Wpretrained+BA

Burada:

* **W**: Orijinal modelin ağırlığı
* **A** ve **B**: Eğitilen düşük boyutlu LoRA matrisleri (örneğin: r=4, yani çok küçük)
* **W değişmez!** → sadece A ve B öğrenilir.

LoRA genelde **attention katmanlarındaki ağırlıklara** uygulanır:

transformer

└── attention

├── q\_proj ─ LoRA uygula

├── k\_proj ─ LoRA uygula

├── v\_proj ─ LoRA uygula

└── o\_proj ─ LoRA uygula

| **Özellik** | **Açıklama** |
| --- | --- |
| Hafif | Yalnızca küçük matrisler eğitilir |
| Az disk kullanımı | LoRA adaptörü birkaç MB olabilir |
| Hızlı eğitim | Özellikle CPU/GPU kısıtlı cihazlarda avantajlı |
| Modülerlik | Farklı LoRA'ları aynı base modele tak/çıkar yapabilirsin |
| Ekonomik | Tam fine-tune’a göre çok daha ucuz ve hızlıdır |

**LoRA Nasıl Uygulanır?**

from peft import get\_peft\_model, LoraConfig, TaskType  
  
lora\_config = LoraConfig(  
 task\_type=TaskType.CAUSAL\_LM,  
 r=8,  
 lora\_alpha=16,  
 lora\_dropout=0.1,  
 target\_modules=["q\_proj", "v\_proj"]  
)  
  
model = get\_peft\_model(base\_model, lora\_config)

Sadece q\_proj, v\_proj gibi belirli modüllere LoRA eklenir. Geri kalanlar donmuş kalır.

## 📁 **Eğitimden Sonra Ne Olur?**

 Base model değişmez (microsoft/phi-4)

 LoRA eğitilir ve şu dosyalar olur:

| **Dosya** | **Açıklama** |
| --- | --- |
| adapter\_model.safetensors | Eğitilen küçük LoRA katmanları |
| adapter\_config.json | LoRA yapılandırması (r, alpha, target modules) |
| tokenizer dosyaları | Tokenizer zaten base model ile aynı olabilir |

## 🧪 **Inference (Test) Zamanı Ne Olur?**

Model şu şekilde yüklenir:

from transformers import AutoModelForCausalLM  
from peft import PeftModel, PeftConfig  
  
config = PeftConfig.from\_pretrained("mrkswe/llmPhi4Try\_CPU")  
base\_model = AutoModelForCausalLM.from\_pretrained(config.base\_model\_name\_or\_path)  
model = PeftModel.from\_pretrained(base\_model, "mrkswe/llmPhi4Try\_CPU")

Bu sayede base model + senin adaptör birleşir = fine-tuned model gibi davranır.

## 🔁 **Birden Fazla LoRA Kullamak Mümkün mü?**

Evet! Aynı base modele farklı LoRA'lar takıp:

* **Multi-task training**
* **LoRA fusion**
* **Inference switch** yapabilirsin.

## 📚 LoRA'nın Kullanıldığı Kütüphaneler

| **Kütüphane** | **Açıklama** |
| --- | --- |
| peft | HuggingFace’in LoRA desteği |
| trl | TRL ile LoRA + SFT yapılabiliyor |
| qlora | Quantized modellerle LoRA yapar (daha hafif) |
| lit-llama | Alternatif LLM kütüphanesi |
| Axolotl, LoRAHub, Lamini | Gelişmiş LoRA framework’leri |

LoRA, büyük modelleri eğitmek yerine, sadece minik eklentiler (LoRA katmanları) ekleyerek **hem az kaynakla hem de hızlı bir şekilde** istediğin göreve uygun hale getirmeni sağlar.

**FINE TUNING METHODLARI**

### ****1. LoRA (Low-Rank Adaptation)****

**Nedir?**  
LoRA, büyük dil modellerini (LLM) yeniden eğitirken sadece çok küçük bir kısmını değiştirerek güncelleme yapma yöntemidir.

**Nasıl çalışır?**

* Büyük modelin ağırlıkları (örneğin W) değiştirilmez.
* Bunun yerine bu ağırlıklara küçük bir "ekleme" yapılır: W + ΔW
* Bu ΔW, düşük dereceli (low-rank) iki matrisin çarpımıdır: ΔW = A \* B
* Eğitim sadece bu küçük A ve B matrisleri üzerinde yapılır.

**Avantajı:**

* Bellek ve hesaplama maliyeti düşer.
* Milyarlarca parametre yerine sadece birkaç milyon parametre eğitilir.
* Büyük modelin doğruluğunu büyük oranda korur.

### ****2. QLoRA (Quantized LoRA)****

**Nedir?**  
QLoRA, LoRA'yı daha da hafif hale getirmek için modelin tamamını quantize (örneğin 4-bit'e sıkıştırma) ederek eğitir.

**Detaylar:**

* Modelin ana ağırlıkları quantize edilir (4-bit gibi).
* LoRA adaptörleri yine float (genellikle 16-bit) olarak eğitilir.
* Eğitim sırasında quantized modelle inference yapılırken LoRA adaptörleriyle öğrenme devam eder.

**Avantajı:**

* RAM ve GPU belleği ihtiyacı çok daha düşük olur.
* Bir dizüstü bilgisayarda bile büyük LLM'ler üzerine fine-tune yapılabilir.

### ****3. LoRA Fusion (LoRA Birleştirme)****

**Nedir?**  
Fine-tune işlemi tamamlandığında LoRA adaptörlerini ana modele "birleştirme" işlemidir.

**Amaç:**

* Modeli tek dosya haline getirmek.
* Deployment sırasında ayrı LoRA dosyalarıyla uğraşmamak.

**Nasıl yapılır?**

* merge\_and\_unload() gibi fonksiyonlarla base\_model + lora\_weights = yeni model
* Artık inference sırasında LoRA'yı yüklemeye gerek kalmaz.

| **Özellik** | **Full Fine-Tune** | **LoRA** |
| --- | --- | --- |
| Eğitim parametre sayısı | Tüm model (milyarlarca) | Çok az (genellikle milyonlarca) |
| Hesaplama maliyeti | Çok yüksek | Düşük |
| Bellek kullanımı | Yüksek GPU/RAM gerektirir | Düşük (hatta CPU ile yapılabilir) |
| Esneklik | Tam kontrol sağlar | Daha sınırlı ama hızlı ve verimli |
| Uygunluk | Büyük sunucular | Yerel veya düşük kaynaklı cihazlarda |
| Model değişikliği | Kalıcı (ağırlıklar güncellenir) | Geçici (ayrı dosyada tutulabilir) |

**NOT: Biz bu şekilde bir fineTuning yaptık ama düşük bellek ve sonrasında da bu şekilde birleştirip bacendde çalışması pek işimize gelmiyor. Nedeni de ollama ile entegre çalışmasını istemem. Ollama ile çalışabilmesi için gguf formatında yüklemeyi yapmak gerekiyor. Bunun için unslot, lama.cpp vs. yapılar var eilimdeki merge modeli gguf ye çevirebilmek için. Yanlız bunların arasında unslot gpu şartı var. Biz de sadece cpu ile bu işlemi nasıl yapabiliriz bunun yollarına baktım ve çözümüünü aşşağıda paylaştım.**

Yukarıdaki yapı ile dışarıya aldığımız lora modeli ile base modeli birleştirerek ve daha sonra da lama.cpp ile gguf e dönüştürüp modeli kaydedecek ve ardından da hugginface e kaydedeceğim. Biz burada tamamını cpulu sistemde yapıyoruz ama loralı modeli alıp gpulu bir cihazda da gguf e dönüştürebilirdik. Farklı seçeneklerimiz de var.ş

Merge-model-base-lora.py:

# CPU ile çıkardığımız parça lora modeli ile base modeli birleştirip tek bir model yapıyoruz  
  
from peft import PeftModel  
from transformers import AutoModelForCausalLM, AutoTokenizer  
  
  
# 1. Ana modeli yükle (Phi-4 tabanı)  
base\_model = AutoModelForCausalLM.from\_pretrained("microsoft/phi-4")  
  
# 2. LoRA modelini yükle (senin fine-tune ettiğin klasör)  
model = PeftModel.from\_pretrained(base\_model, "./fine\_tuned\_cpu\_model")  
  
# 3. LoRA ile birleşimi tamamla  
merged\_model = model.merge\_and\_unload()  
  
# 4. Birleştirilmiş modeli kaydet  
merged\_model.save\_pretrained("./merged\_model")  
  
# 5. Tokenizer'ı da aynı klasöre kaydet  
tokenizer = AutoTokenizer.from\_pretrained("microsoft/phi-4")  
tokenizer.save\_pretrained("./merged\_model")

Bu kodda lora model ile base modeli birleştirmiş olduk. Burada merged-model diye bir klasör çıkıyor

Gguf e dönüştürme işlemlerinde öncelikle farklı bir repo ile lora yı dönüştürmeyi denedim ama git reposu kaldırılmıştı. Ondan dolayı direct olarak lama.cpp kullanmaya karar verdim.

git clone https://github.com/ggerganov/llama.cpp.git

cd llama.cpp

pip install -r requirements/requirements-convert\_hf\_to\_gguf.txt

Bu, .gguf dönüşüm script’ini çalıştırmak için gerekli bağımlılıkları yükler

CPU üzerinde .gguf formatına dönüştürme:

python3 convert\_hf\_to\_gguf.py \

/tam/yol/merged\_model \

--outfile /tam/yol/phi4-finetuned-cpu.gguf \

--outtype q4\_k\_m

Burada merged-model klasörü içerisindeki modeli gguf modeline çeviriyoruz.

HuggingFace’e .gguf dosyası upload et:

# modeli hugginface yüklemek içni  
  
  
from huggingface\_hub import HfApi  
  
api = HfApi()  
  
repo\_id = "mrkswe/llmPhi4Try\_CPU\_2" # senin repo ID'n  
hf\_token = "hf\_CVnLLVoKHZoGlTiRTpPtYuzhCHJhTmfWho" # token  
  
# yeni bir model reposu oluşturmak için  
# api.create\_repo(repo\_id, exist\_ok=True, repo\_type="model")  
  
api.upload\_file(  
 path\_or\_fileobj="phi4-finetuned-cpu.gguf",  
 path\_in\_repo="phi4-finetuned-cpu.gguf",  
 repo\_id=repo\_id,  
 token=hf\_token  
)  
  
print(f"GGUF dosyası yüklendi: https://huggingface.co/{repo\_id}")

Bu şekilde de hugginfacee yüklemiş olacakğız.