

TL09 PEFT

Índice

1. Introducción
2. PEFT
3. Fine-tuning con LoRA de un MLP
4. Fine-tuning con LoRA de un ViT para food101
5. Ejercicio: fine-tuning con LoRA de un ViT para CIFAR10

1 Introducción

Documentación de transformers:

- **Get started:** TL02
- **Clases base:** TL03
- **Inferencia:**
 - **API Pipeline:** TL03
 - **LLMs:** TL06
 - **Chat with models:** TL07
- **Entrenamiento:**
 - **API Trainer:** TL08

Otras librerías y recursos:

- **pytorch** TL01
- **datasets:** TL04
- **evaluate:** TL04
- Benchmarking: TL06
- **accelerate:** TL08
- **PEFT:** secciones siguientes

Objetivo: introducir PEFT

2 PEFT

Documentación de PEFT:

- **Get started:** PEFT; quicktour; instalación
- **Tutorial:** configuración y modelos; integraciones
- **PEFT method guides:** Prompt-based methods; LoRA methods; IA3
- **Developer guides:** Model merging; Quantization; LoRA; ...
- **Accelerate integrations:** DeepSpeed; Fully Sharded Data Parallel
- **Conceptual guides:** Adapters; Soft prompts; IA3; OFT/BOFT
- **Referencia:** Main classes; Adapters; Utilities

Github de PEFT:

- [Ejemplos](#) incluye cuadernos ejemplo en los que podemos basarnos
- ...

3 Fine-tuning con LoRA de un MLP

Fine-tuning a multilayer perceptron using LoRA and 🤖 PEFT: ejemplo en el que se basa esta sección

Librerías:

```
In [ ]: import copy; import os; import peft; import torch; from torch import nn; import torch.nn.functional as F
os.environ["BITSANDBYTES_NOWELCOME"] = "1" # ignore bnb warnings
```

Dataset: 800 vectores 20-dim aleatorios de clase 0 o 1 aleatoria, a procesar en lotes de 64

```
In [ ]: torch.manual_seed(0); X = torch.rand((1000, 20)); y = (X.sum(1) > 10).long()
n_train = 800; batch_size = 64
train_dataloader = torch.utils.data.DataLoader(torch.utils.data.TensorDataset(X[:n_train], y[:n_train]),
        batch_size=batch_size, shuffle=True)
eval_dataloader = torch.utils.data.DataLoader(torch.utils.data.TensorDataset(X[n_train:], y[n_train:]),
        batch_size=batch_size)
X[0], y[0]
```

```
Out[ ]: (tensor([0.4963, 0.7682, 0.0885, 0.1320, 0.3074, 0.6341, 0.4901, 0.8964, 0.4556,
        0.6323, 0.3489, 0.4017, 0.0223, 0.1689, 0.2939, 0.5185, 0.6977, 0.8000,
        0.1610, 0.2823]),
        tensor(0))
```

Modelo: MLP de 2 capas ReLU ocultas y capa de salida binaria LogSoftmax

```
In [ ]: class MLP(nn.Module):
        def __init__(self, num_units_hidden=2000):
            super().__init__()
            self.seq = nn.Sequential(
                nn.Linear(20, num_units_hidden), nn.ReLU(),
                nn.Linear(num_units_hidden, num_units_hidden), nn.ReLU(),
                nn.Linear(num_units_hidden, 2), nn.LogSoftmax(dim=-1))
        def forward(self, X):
            return self.seq(X)
```

Entrenamiento: parámetros y función básicos

```
In [ ]: lr = 0.002; batch_size = 64; max_epochs = 10
device = torch.accelerator.current_accelerator().type if torch.accelerator.is_available() else "cpu"
device
```

```
Out[ ]: 'cuda'
```

```
In [ ]: def train(model, optimizer, criterion, train_dataloader, eval_dataloader, epochs):
    for epoch in range(epochs):
        model.train()
        train_loss = 0
        for xb, yb in train_dataloader:
            xb = xb.to(device); yb = yb.to(device); outputs = model(xb)
            loss = criterion(outputs, yb); train_loss += loss.detach().float()
            loss.backward(); optimizer.step(); optimizer.zero_grad()
        model.eval(); eval_loss = 0
        for xb, yb in eval_dataloader:
            xb = xb.to(device); yb = yb.to(device)
            with torch.no_grad():
                outputs = model(xb)
            loss = criterion(outputs, yb); eval_loss += loss.detach().float()
        eval_loss_total = (eval_loss / len(eval_dataloader)).item()
        train_loss_total = (train_loss / len(train_dataloader)).item()
        print(f"epoch=:{<2} {train_loss_total=:.4f} {eval_loss_total=:.4f}")
```

Entrenamiento: sin LoRA

```
In [ ]: module = MLP().to(device)
optimizer = torch.optim.Adam(module.parameters(), lr=lr)
criterion = nn.CrossEntropyLoss()
```

```
In [ ]: %time train(module, optimizer, criterion, train_dataloader, eval_dataloader, epochs=max_epochs)
```

```
epoch=0   train_loss_total=0.7970   eval_loss_total=0.6472
epoch=1   train_loss_total=0.5597   eval_loss_total=0.4898
epoch=2   train_loss_total=0.3696   eval_loss_total=0.3323
epoch=3   train_loss_total=0.2364   eval_loss_total=0.5454
epoch=4   train_loss_total=0.2428   eval_loss_total=0.2843
epoch=5   train_loss_total=0.1251   eval_loss_total=0.2514
epoch=6   train_loss_total=0.0952   eval_loss_total=0.2068
epoch=7   train_loss_total=0.0831   eval_loss_total=0.2395
epoch=8   train_loss_total=0.0655   eval_loss_total=0.2524
epoch=9   train_loss_total=0.0380   eval_loss_total=0.3650
CPU times: user 193 ms, sys: 112 ms, total: 305 ms
Wall time: 325 ms
```

Entrenamiento: con LoRA

```
In [ ]: [(n, type(m)) for n, m in MLP().named_modules()]
```

```
Out[ ]: [('', __main__.MLP),  
        ('seq', torch.nn.modules.container.Sequential),  
        ('seq.0', torch.nn.modules.linear.Linear),  
        ('seq.1', torch.nn.modules.activation.ReLU),  
        ('seq.2', torch.nn.modules.linear.Linear),  
        ('seq.3', torch.nn.modules.activation.ReLU),  
        ('seq.4', torch.nn.modules.linear.Linear),  
        ('seq.5', torch.nn.modules.activation.LogSoftmax)]
```

```
In [ ]: config = peft.LoraConfig(r=8, target_modules=["seq.0", "seq.2"], modules_to_save=["seq.4"])
```

```
In [ ]: module = MLP().to(device)  
module_copy = copy.deepcopy(module) # we keep a copy of the original model for later  
peft_model = peft.get_peft_model(module, config)  
optimizer = torch.optim.Adam(peft_model.parameters(), lr=lr)  
criterion = nn.CrossEntropyLoss()  
peft_model.print_trainable_parameters()
```

```
trainable params: 52,162 || all params: 4,100,164 || trainable%: 1.2722
```

```
In [ ]: %time train(peft_model, optimizer, criterion, train_dataloader, eval_dataloader, epochs=max_epochs)
```

```
epoch=0   train_loss_total=0.6873   eval_loss_total=0.6671  
epoch=1   train_loss_total=0.6247   eval_loss_total=0.5831  
epoch=2   train_loss_total=0.4716   eval_loss_total=0.4771  
epoch=3   train_loss_total=0.3213   eval_loss_total=0.2492  
epoch=4   train_loss_total=0.2283   eval_loss_total=0.6788  
epoch=5   train_loss_total=0.3126   eval_loss_total=0.4314  
epoch=6   train_loss_total=0.2784   eval_loss_total=0.2741  
epoch=7   train_loss_total=0.2071   eval_loss_total=0.2370  
epoch=8   train_loss_total=0.1257   eval_loss_total=0.2654  
epoch=9   train_loss_total=0.0841   eval_loss_total=0.2018  
CPU times: user 153 ms, sys: 5.23 ms, total: 158 ms  
Wall time: 175 ms
```

Entrenamiento: efecto de LoRA en términos de parámetros añadidos y originales actualizados

```
In [ ]: for name, param in peft_model.base_model.named_parameters():
        if "lora" not in name:
            continue
        print(f"New parameter {name:<13} | {param.numel():>5} parameters | updated")
```

```
New parameter model.seq.0.lora_A.default.weight | 160 parameters | updated
New parameter model.seq.0.lora_B.default.weight | 16000 parameters | updated
New parameter model.seq.2.lora_A.default.weight | 16000 parameters | updated
New parameter model.seq.2.lora_B.default.weight | 16000 parameters | updated
```

```
In [ ]: params_before = dict(module_copy.named_parameters())
        for name, param in peft_model.base_model.named_parameters():
            if "lora" in name:
                continue
            name_before = (name.partition(".")[0].replace("base_layer.", "").replace("original_", "")\
                           .replace("module.", "").replace("modules_to_save.default.", ""))
            param_before = params_before[name_before]
            if torch.allclose(param, param_before):
                print(f"Parameter {name_before:<13} | {param.numel():>7} parameters | not updated")
            else:
                print(f"Parameter {name_before:<13} | {param.numel():>7} parameters | updated")
```

```
Parameter seq.0.weight | 40000 parameters | not updated
Parameter seq.0.bias | 2000 parameters | not updated
Parameter seq.2.weight | 4000000 parameters | not updated
Parameter seq.2.bias | 2000 parameters | not updated
Parameter seq.4.weight | 4000 parameters | not updated
Parameter seq.4.bias | 2 parameters | not updated
Parameter seq.4.weight | 4000 parameters | updated
Parameter seq.4.bias | 2 parameters | updated
```

4 Fine-tuning con LoRA de un ViT para food101

```
In [ ]: import numpy as np; import torch; import transformers; import evaluate; import accelerate; import peft
```

```
In [ ]: model_checkpoint = "google/vit-base-patch16-224-in21k"
```

```
In [ ]: from datasets import load_dataset
full_dataset = load_dataset("food101", split="train")
dataset = full_dataset.shuffle(seed=7).select(range(50000)); del full_dataset
```

```
In [ ]: labels = dataset.features["label"].names; label2id, id2label = dict(), dict()
for i, label in enumerate(labels):
    label2id[label] = i; id2label[i] = label
```

```
In [ ]: from transformers import AutoImageProcessor
image_processor = AutoImageProcessor.from_pretrained(model_checkpoint)
```

```
In [ ]: from torchvision.transforms import (CenterCrop, Compose, Normalize, RandomHorizontalFlip, RandomResizedCrop,
    Resize, ToTensor)
normalize = Normalize(mean=image_processor.image_mean, std=image_processor.image_std)
train_transforms = Compose([RandomResizedCrop(image_processor.size["height"]),
    RandomHorizontalFlip(), ToTensor(), normalize])
val_transforms = Compose([Resize(image_processor.size["height"]), CenterCrop(image_processor.size["height"]),
    ToTensor(), normalize])
def preprocess_train(example_batch):
    """Apply train_transforms across a batch."""
    example_batch["pixel_values"] = [train_transforms(image.convert("RGB")) for image in example_batch["image"]]
    return example_batch
def preprocess_val(example_batch):
    """Apply val_transforms across a batch."""
    example_batch["pixel_values"] = [val_transforms(image.convert("RGB")) for image in example_batch["image"]]
    return example_batch
splits = dataset.train_test_split(test_size=0.1)
train_ds = splits["train"]; val_ds = splits["test"]
train_ds.set_transform(preprocess_train); val_ds.set_transform(preprocess_val)
```

Modelo:

```
In [ ]: from transformers import AutoModelForImageClassification, TrainingArguments, Trainer
model = AutoModelForImageClassification.from_pretrained(model_checkpoint, label2id=label2id, id2label=id2label)
```

Entrenamiento: parámetros entrenables

```
In [ ]: def print_trainable_parameters(model):
        """
        Prints the number of trainable parameters in the model.
        """
        trainable_params = 0
        all_param = 0
        for _, param in model.named_parameters():
            all_param += param.numel()
            if param.requires_grad:
                trainable_params += param.numel()
        print(f"trainable params: {trainable_params} || all params: {all_param} ||",
              f"trainable%: {100 * trainable_params / all_param:.2f}")
```

```
In [ ]: print_trainable_parameters(model)

trainable params: 85876325 || all params: 85876325 || trainable%: 100.00
```

Entrenamiento: modelo LoRA y parámetros entrenables

```
In [ ]: from peft import LoraConfig, get_peft_model
        config = LoraConfig(
            r=16,
            lora_alpha=16,
            target_modules=["query", "value"],
            lora_dropout=0.1,
            bias="none",
            modules_to_save=["classifier"])
        lora_model = get_peft_model(model, config)
```

```
In [ ]: print_trainable_parameters(lora_model)
```

```
trainable params: 667493 || all params: 86543818 || trainable%: 0.77
```

Entrenamiento: del modelo LoRA

```
In [ ]: from transformers import TrainingArguments, Trainer
model_name = model_checkpoint.split("/")[-1]; batch_size = 128
args = TrainingArguments(
    f"{model_name}-finetuned-lora-food101",
    remove_unused_columns=False,
    eval_strategy="epoch",
    save_strategy="epoch",
    learning_rate=5e-3,
    per_device_train_batch_size=batch_size,
    gradient_accumulation_steps=4,
    per_device_eval_batch_size=batch_size,
    fp16=True,
    num_train_epochs=10,
    logging_steps=10,
    load_best_model_at_end=True,
    metric_for_best_model="accuracy",
    push_to_hub=False,
    label_names=["labels"])
```

```
In [ ]: metric = evaluate.load("accuracy")
def compute_metrics(eval_pred):
    """Computes accuracy on a batch of predictions"""
    predictions = np.argmax(eval_pred.predictions, axis=1)
    return metric.compute(predictions=predictions, references=eval_pred.label_ids)
```

```
In [ ]: def collate_fn(examples):
    pixel_values = torch.stack([example["pixel_values"] for example in examples])
    labels = torch.tensor([example["label"] for example in examples])
    return {"pixel_values": pixel_values, "labels": labels}
```

```
In [ ]: trainer = Trainer(
    lora_model,
    args,
    train_dataset=train_ds,
    eval_dataset=val_ds,
    processing_class=image_processor,
    compute_metrics=compute_metrics,
    data_collator=collate_fn)
train_results = trainer.train()
```

 [880/880 54:40, Epoch 10/10]

Epoch	Training Loss	Validation Loss	Accuracy
1	0.901400	0.719933	0.804400
2	0.796000	0.649857	0.821600
3	0.672100	0.628907	0.830600
4	0.629100	0.598930	0.835000
5	0.557100	0.595112	0.840000
6	0.507900	0.594549	0.841200
7	0.482000	0.573726	0.847000
8	0.432400	0.573374	0.851200
9	0.405100	0.570710	0.848600
10	0.387200	0.567812	0.850600

```
In [ ]: trainer.log_metrics("train", train_results.metrics)
```

```
***** train metrics *****
epoch                =          10.0
total_flos           = 32757989277GF
train_loss           =          0.617
train_runtime        =    0:54:44.97
train_samples_per_second =      136.987
train_steps_per_second  =          0.268
```

```
In [ ]: metrics = trainer.evaluate()
trainer.log_metrics("eval", metrics)
```

 [40/40 00:29]

```
***** eval metrics *****
epoch                =          10.0
eval_accuracy        =          0.8512
eval_loss            =          0.5734
eval_runtime         =    0:00:32.37
eval_samples_per_second =      154.431
eval_steps_per_second  =          1.235
```

5 Ejercicio: fine-tuning con LoRA de un ViT para CIFAR10

```
In [ ]: import numpy as np; import torch; import transformers; import evaluate
```

```
In [ ]: model_checkpoint = "google/vit-base-patch16-224-in21k"
```

```
In [ ]: from datasets import load_dataset
train_ds, test_ds = load_dataset('cifar10', split=['train[:50000]', 'test[:10000]']) # mucho menos para pruebas
```

```
In [ ]: id2label = {id:label for id, label in enumerate(train_ds.features['label'].names)}
label2id = {label:id for id,label in id2label.items()}
```

```
In [ ]: from transformers import AutoImageProcessor
image_processor = AutoImageProcessor.from_pretrained(model_checkpoint)
```

```
In [ ]: from torchvision.transforms import (CenterCrop, Compose, Normalize, RandomHorizontalFlip, RandomResizedCrop,
      Resize, ToTensor)
normalize = Normalize(mean=image_processor.image_mean, std=image_processor.image_std)
train_transforms = Compose([RandomResizedCrop(image_processor.size["height"]),
      RandomHorizontalFlip(), ToTensor(), normalize])
val_transforms = Compose([Resize(image_processor.size["height"]), CenterCrop(image_processor.size["height"]),
      ToTensor(), normalize])
def preprocess_train(example_batch):
    example_batch["pixel_values"] = [train_transforms(image.convert("RGB")) for image in example_batch["img"]]
    return example_batch
def preprocess_val(example_batch):
    example_batch["pixel_values"] = [val_transforms(image.convert("RGB")) for image in example_batch["img"]]
    return example_batch
train_ds.set_transform(preprocess_train); test_ds.set_transform(preprocess_val)
```

```
In [ ]: from transformers import AutoModelForImageClassification, TrainingArguments, Trainer
model = AutoModelForImageClassification.from_pretrained(model_checkpoint, label2id=label2id, id2label=id2label)
```

Ejercicio: completa el experimento para hallar la precisión en test y el coste temporal del entrenamiento; compara resultados con el fine-tuning convencional

Solución:

```
In [ ]: from peft import LoraConfig, get_peft_model
config = LoraConfig(
    r=16,
    lora_alpha=16,
    target_modules=["query", "value"],
    lora_dropout=0.1,
    bias="none",
    modules_to_save=["classifier"])
lora_model = get_peft_model(model, config)
```


```
In [ ]: model_name = model_checkpoint.split("/")[-1]; batch_size = 32
args = TrainingArguments(
    f"{model_name}-finetuned-lora-cifar10",
    remove_unused_columns=False,
    eval_strategy = "epoch",
    save_strategy = "epoch",
    learning_rate=5e-5,
    per_device_train_batch_size=batch_size,
    gradient_accumulation_steps=4,
    per_device_eval_batch_size=batch_size,
    num_train_epochs=3,
    warmup_ratio=0.1,
    logging_steps=10,
    load_best_model_at_end=True,
    metric_for_best_model="accuracy",
    push_to_hub=False)
```

```
In [ ]: metric = evaluate.load("accuracy")
def compute_metrics(eval_pred):
    """Computes accuracy on a batch of predictions"""
    predictions = np.argmax(eval_pred.predictions, axis=1)
    return metric.compute(predictions=predictions, references=eval_pred.label_ids)
```

```
In [ ]: def collate_fn(examples):
    pixel_values = torch.stack([example["pixel_values"] for example in examples])
    labels = torch.tensor([example["label"] for example in examples])
    return {"pixel_values": pixel_values, "labels": labels}
```

```
In [ ]: trainer = Trainer(
    lora_model,
    args,
    train_dataset=train_ds,
    eval_dataset=test_ds,
    processing_class=image_processor,
    compute_metrics=compute_metrics,
    data_collator=collate_fn)
```

```
In [ ]: train_results = trainer.train()
```

 [1173/1173 35:31, Epoch 3/3]

Epoch	Training Loss	Validation Loss	Accuracy
1	1.160000	1.034516	0.957400
2	0.680800	0.475599	0.969600
3	0.522400	0.383807	0.969900

```
In [ ]: trainer.log_metrics("train", train_results.metrics)
```

```
***** train metrics *****
epoch                =          3.0
total_flos           = 10901671102GF
train_loss           =          1.0917
train_runtime        =          0:35:34.90
train_samples_per_second =          70.261
train_steps_per_second  =           0.549
```

```
In [ ]: metrics = trainer.evaluate()
trainer.log_metrics("eval", metrics)
```

 [313/313 01:37]

```
***** eval metrics *****
epoch                =          3.0
eval_accuracy        =          0.9699
eval_loss            =          0.3838
eval_runtime         = 0:01:37.46
eval_samples_per_second =          102.596
eval_steps_per_second  =           3.211
```