

appendix D

Adding bells and whistles to the training loop

In this appendix, we enhance the training function for the pretraining and fine-tuning processes covered in chapters 5 to 7. In particular, it covers *learning rate warmup*, *cosine decay*, and *gradient clipping*. We then incorporate these techniques into the training function and pretrain an LLM.

To make the code self-contained, we reinitialize the model we trained in chapter 5:

```
import torch
from chapter04 import GPTModel

GPT_CONFIG_124M = {
    "vocab_size": 50257,
    "context_length": 256,
    "emb_dim": 768,
    "n_heads": 12,
    "n_layers": 12,
    "drop_rate": 0.1,
    "qkv_bias": False
}

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)
model.to(device)
model.eval()
```

Vocabulary size

Shortened context length (orig: 1024)

Embedding dimension

Number of attention heads

Number of layers

Dropout rate

Query-key-value bias

After initializing the model, we need to initialize the data loaders. First, we load the “The Verdict” short story:

```

import os
import urllib.request

file_path = "the-verdict.txt"

url = (
    "https://raw.githubusercontent.com/rasbt/LLMs-from-scratch/"
    "main/ch02/01_main-chapter-code/the-verdict.txt"
)

if not os.path.exists(file_path):
    with urllib.request.urlopen(url) as response:
        text_data = response.read().decode('utf-8')
    with open(file_path, "w", encoding="utf-8") as file:
        file.write(text_data)
else:
    with open(file_path, "r", encoding="utf-8") as file:
        text_data = file.read()

```

Next, we load the `text_data` into the data loaders:

```

from previous_chapters import create_dataloader_v1

train_ratio = 0.90
split_idx = int(train_ratio * len(text_data))
torch.manual_seed(123)
train_loader = create_dataloader_v1(
    text_data[:split_idx],
    batch_size=2,
    max_length=GPT_CONFIG_124M["context_length"],
    stride=GPT_CONFIG_124M["context_length"],
    drop_last=True,
    shuffle=True,
    num_workers=0
)
val_loader = create_dataloader_v1(
    text_data[split_idx:],
    batch_size=2,
    max_length=GPT_CONFIG_124M["context_length"],
    stride=GPT_CONFIG_124M["context_length"],
    drop_last=False,
    shuffle=False,
    num_workers=0
)

```

D.1 *Learning rate warmup*

Implementing a learning rate warmup can stabilize the training of complex models such as LLMs. This process involves gradually increasing the learning rate from a very low initial value (`initial_lr`) to a maximum value specified by the user (`peak_lr`). Starting the training with smaller weight updates decreases the risk of the model encountering large, destabilizing updates during its training phase.

Suppose we plan to train an LLM for 15 epochs, starting with an initial learning rate of 0.0001 and increasing it to a maximum learning rate of 0.01:

```
n_epochs = 15
initial_lr = 0.0001
peak_lr = 0.01
warmup_steps = 20
```

The number of warmup steps is usually set between 0.1% and 20% of the total number of steps, which we can calculate as follows:

```
total_steps = len(train_loader) * n_epochs
warmup_steps = int(0.2 * total_steps)    ← 20% warmup
print(warmup_steps)
```

This prints 27, meaning that we have 20 warmup steps to increase the initial learning rate from 0.0001 to 0.01 in the first 27 training steps.

Next, we implement a simple training loop template to illustrate this warmup process:

```
optimizer = torch.optim.AdamW(model.parameters(), weight_decay=0.1)
lr_increment = (peak_lr - initial_lr) / warmup_steps

global_step = -1
track_lrs = []

for epoch in range(n_epochs):
    for input_batch, target_batch in train_loader:
        optimizer.zero_grad()
        global_step += 1

        if global_step < warmup_steps:
            lr = initial_lr + global_step * lr_increment
        else:
            lr = peak_lr

        for param_group in optimizer.param_groups:
            param_group["lr"] = lr
        track_lrs.append(optimizer.param_groups[0]["lr"])
```

Applies the calculated learning rate to the optimizer →

← **This increment is determined by how much we increase the initial_lr in each of the 20 warmup steps.**

← **Executes a typical training loop iterating over the batches in the training loader in each epoch**

← **Updates the learning rate if we are still in the warmup phase**

← **In a complete training loop, the loss and the model updates would be calculated, which are omitted here for simplicity.**

After running the preceding code, we visualize how the learning rate was changed by the training loop to verify that the learning rate warmup works as intended:

```
import matplotlib.pyplot as plt

plt.ylabel("Learning rate")
plt.xlabel("Step")
total_training_steps = len(train_loader) * n_epochs
plt.plot(range(total_training_steps), track_lrs);
plt.show()
```

The resulting plot shows that the learning rate starts with a low value and increases for 20 steps until it reaches the maximum value after 20 steps (figure D.1).

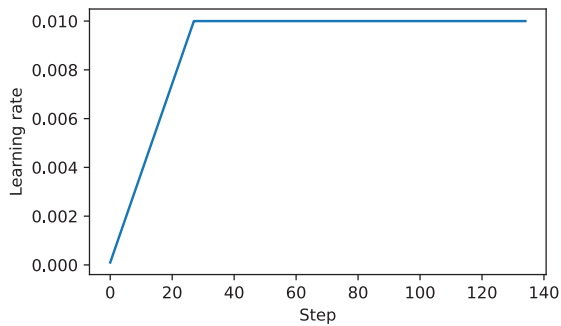


Figure D.1 The learning rate warmup increases the learning rate for the first 20 training steps. After 20 steps, the learning rate reaches the peak of 0.01 and remains constant for the rest of the training.

Next, we will modify the learning rate further so that it decreases after reaching the maximum learning rate, which further helps improve the model training.

D.2 Cosine decay

Another widely adopted technique for training complex deep neural networks and LLMs is *cosine decay*. This method modulates the learning rate throughout the training epochs, making it follow a cosine curve after the warmup stage.

In its popular variant, cosine decay reduces (or decays) the learning rate to nearly zero, mimicking the trajectory of a half-cosine cycle. The gradual learning decrease in cosine decay aims to decelerate the pace at which the model updates its weights. This is particularly important because it helps minimize the risk of overshooting the loss minima during the training process, which is essential for ensuring the stability of the training during its later phases.

We can modify the training loop template by adding cosine decay:

```
import math

min_lr = 0.1 * initial_lr
track_lrs = []
lr_increment = (peak_lr - initial_lr) / warmup_steps
global_step = -1

for epoch in range(n_epochs):
    for input_batch, target_batch in train_loader:
        optimizer.zero_grad()
        global_step += 1

        if global_step < warmup_steps:
            lr = initial_lr + global_step * lr_increment
        else:
            progress = ((global_step - warmup_steps) /
                        (total_training_steps - warmup_steps))
```

← Applies linear warmup

← Uses cosine annealing after warmup

```

lr = min_lr + (peak_lr - min_lr) * 0.5 * (
    1 + math.cos(math.pi * progress)
)

for param_group in optimizer.param_groups:
    param_group["lr"] = lr
track_lrs.append(optimizer.param_groups[0]["lr"])

```

Again, to verify that the learning rate has changed as intended, we plot the learning rate:

```

plt.ylabel("Learning rate")
plt.xlabel("Step")
plt.plot(range(total_training_steps), track_lrs)
plt.show()

```

The resulting learning rate plot shows that the learning rate starts with a linear warmup phase, which increases for 20 steps until it reaches the maximum value after 20 steps. After the 20 steps of linear warmup, cosine decay kicks in, reducing the learning rate gradually until it reaches its minimum (figure D.2).

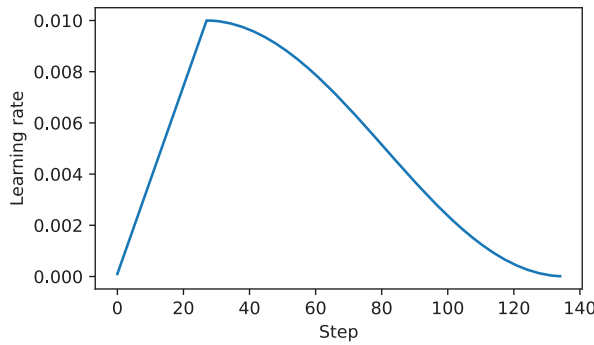


Figure D.2 The first 20 steps of linear learning rate warmup are followed by a cosine decay, which reduces the learning rate in a half-cosine cycle until it reaches its minimum point at the end of training.

D.3 Gradient clipping

Gradient clipping is another important technique for enhancing stability during LLM training. This method involves setting a threshold above which gradients are down-scaled to a predetermined maximum magnitude. This process ensures that the updates to the model’s parameters during backpropagation stay within a manageable range.

For example, applying the `max_norm=1.0` setting within PyTorch’s `clip_grad_norm_` function ensures that the norm of the gradients does not surpass 1.0. Here, the term “norm” signifies the measure of the gradient vector’s length, or magnitude, within the model’s parameter space, specifically referring to the L2 norm, also known as the Euclidean norm.

In mathematical terms, for a vector \mathbf{v} composed of components $\mathbf{v} = [v_1, v_2, \dots, v_n]$, the L2 norm is

$$|\mathbf{v}|_2 = \sqrt{v_1^2 + v_2^2 + \dots + v_n^2}$$

This calculation method is also applied to matrices. For instance, consider a gradient matrix given by

$$\mathbf{G} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

If we want to clip these gradients to a `max_norm` of 1, we first compute the L2 norm of these gradients, which is

$$|\mathbf{G}|_2 = \sqrt{1^2 + 2^2 + 2^2 + 4^2} = \sqrt{25} = 5$$

Given that $|\mathbf{G}|_2 = 5$ exceeds our `max_norm` of 1, we scale down the gradients to ensure their norm equals exactly 1. This is achieved through a scaling factor, calculated as `max_norm/|G|2 = 1/5`. Consequently, the adjusted gradient matrix \mathbf{G}' becomes

$$\mathbf{G}' = \frac{1}{5} \times \mathbf{G} = \begin{bmatrix} \frac{1}{5} & \frac{2}{5} \\ \frac{3}{5} & \frac{4}{5} \end{bmatrix}$$

To illustrate this gradient clipping process, we begin by initializing a new model and calculating the loss for a training batch, similar to the procedure in a standard training loop:

```
from chapter05 import calc_loss_batch

torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)
model.to(device)
loss = calc_loss_batch(input_batch, target_batch, model, device)
loss.backward()
```

Upon calling the `.backward()` method, PyTorch calculates the loss gradients and stores them in a `.grad` attribute for each model weight (parameter) tensor.

To clarify the point, we can define the following `find_highest_gradient` utility function to identify the highest gradient value by scanning all the `.grad` attributes of the model's weight tensors after calling `.backward()`:

```
def find_highest_gradient(model):
    max_grad = None
    for param in model.parameters():
        if param.grad is not None:
            grad_values = param.grad.data.flatten()
            max_grad_param = grad_values.max()
            if max_grad is None or max_grad_param > max_grad:
                max_grad = max_grad_param
    return max_grad
print(find_highest_gradient(model))
```

The largest gradient value identified by the preceding code is

```
tensor(0.0411)
```

Let's now apply gradient clipping and see how this affects the largest gradient value:

```
torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
print(find_highest_gradient(model))
```

The largest gradient value after applying the gradient clipping with the max norm of 1 is substantially smaller than before:

```
tensor(0.0185)
```

D.4 The modified training function

Finally, we improve the `train_model_simple` training function (see chapter 5) by adding the three concepts introduced herein: linear warmup, cosine decay, and gradient clipping. Together, these methods help stabilize LLM training.

The code, with the changes compared to the `train_model_simple` annotated, is as follows:

```

from chapter05 import evaluate_model, generate_and_print_sample

def train_model(model, train_loader, val_loader, optimizer, device,
                n_epochs, eval_freq, eval_iter, start_context, tokenizer,
                warmup_steps, initial_lr=3e-05, min_lr=1e-6):

    train_losses, val_losses, track_tokens_seen, track_lrs = [], [], [], []
    tokens_seen, global_step = 0, -1

    peak_lr = optimizer.param_groups[0]["lr"]
    total_training_steps = len(train_loader) * n_epochs
    lr_increment = (peak_lr - initial_lr) / warmup_steps

    for epoch in range(n_epochs):
        model.train()
        for input_batch, target_batch in train_loader:
            optimizer.zero_grad()
            global_step += 1

            if global_step < warmup_steps:
                lr = initial_lr + global_step * lr_increment
            else:
                progress = ((global_step - warmup_steps) /
                           (total_training_steps - warmup_steps))
                lr = min_lr + (peak_lr - min_lr) * 0.5 * (
                    1 + math.cos(math.pi * progress))

```

Retrieves the initial learning rate from the optimizer, assuming we use it as the peak learning rate

Calculates the total number of iterations in the training process

Calculates the learning rate increment during the warmup phase

Adjusts the learning rate based on the current phase (warmup or cosine annealing)

```

for param_group in optimizer.param_groups:
    param_group["lr"] = lr
track_lrs.append(lr)
loss = calc_loss_batch(input_batch, target_batch, model, device)
loss.backward()

if global_step > warmup_steps:
    torch.nn.utils.clip_grad_norm_(
        model.parameters(), max_norm=1.0
    )

optimizer.step()
tokens_seen += input_batch.numel()

if global_step % eval_freq == 0:
    train_loss, val_loss = evaluate_model(
        model, train_loader, val_loader,
        device, eval_iter
    )
    train_losses.append(train_loss)
    val_losses.append(val_loss)
    track_tokens_seen.append(tokens_seen)
    print(f"Ep {epoch+1} (Iter {global_step:06d}): "
          f"Train loss {train_loss:.3f}, "
          f"Val loss {val_loss:.3f}")

generate_and_print_sample(
    model, tokenizer, device, start_context
)

return train_losses, val_losses, track_tokens_seen, track_lrs

```

← Applies the calculated learning rate to the optimizer

← Applies gradient clipping after the warmup phase to avoid exploding gradients

← Everything below here remains unchanged compared to the `train_model_simple` function used in chapter 5.

After defining the `train_model` function, we can use it in a similar fashion to train the model compared to the `train_model_simple` method we used for pretraining:

```

import tiktoken

torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)
model.to(device)
peak_lr = 5e-4
optimizer = torch.optim.AdamW(model.parameters(), weight_decay=0.1)
tokenizer = tiktoken.get_encoding("gpt2")

n_epochs = 15
train_losses, val_losses, tokens_seen, lrs = train_model(
    model, train_loader, val_loader, optimizer, device, n_epochs=n_epochs,
    eval_freq=5, eval_iter=1, start_context="Every effort moves you",
    tokenizer=tokenizer, warmup_steps=warmup_steps,
    initial_lr=1e-5, min_lr=1e-5
)

```


The training will take about 5 minutes to complete on a MacBook Air or similar laptop and prints the following outputs:

```
Ep 1 (Iter 000000): Train loss 10.934, Val loss 10.939
Ep 1 (Iter 000005): Train loss 9.151, Val loss 9.461
Every effort moves you,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
Ep 2 (Iter 000010): Train loss 7.949, Val loss 8.184
Ep 2 (Iter 000015): Train loss 6.362, Val loss 6.876
Every effort moves you,,,,,,,,,,,,,,,,,,,,,,,, the,,,,,,,, the,,,,,,,,
the,,,,,,,,
...
Ep 15 (Iter 000130): Train loss 0.041, Val loss 6.915
Every effort moves you?" "Yes--quite insensible to the irony. She wanted him
vindicated--and by me!" He laughed again, and threw back his head to look up
at the sketch of the donkey. "There were days when I
```

Like pretraining, the model begins to overfit after a few epochs since it is a very small dataset, and we iterate over it multiple times. Nonetheless, we can see that the function is working since it minimizes the training set loss.

Readers are encouraged to train the model on a larger text dataset and compare the results obtained with this more sophisticated training function to the results that can be obtained with the `train_model_simple` function.

appendix E

Parameter-efficient fine-tuning with LoRA

Low-rank adaptation (LoRA) is one of the most widely used techniques for *parameter-efficient fine-tuning*. The following discussion is based on the spam classification fine-tuning example given in chapter 6. However, LoRA fine-tuning is also applicable to the supervised *instruction fine-tuning* discussed in chapter 7.

E.1 Introduction to LoRA

LoRA is a technique that adapts a pretrained model to better suit a specific, often smaller dataset by adjusting only a small subset of the model’s weight parameters. The “low-rank” aspect refers to the mathematical concept of limiting model adjustments to a smaller dimensional subspace of the total weight parameter space. This effectively captures the most influential directions of the weight parameter changes during training. The LoRA method is useful and popular because it enables efficient fine-tuning of large models on task-specific data, significantly cutting down on the computational costs and resources usually required for fine-tuning.

Suppose a large weight matrix W is associated with a specific layer. LoRA can be applied to all linear layers in an LLM. However, we focus on a single layer for illustration purposes.

When training deep neural networks, during backpropagation, we learn a ΔW matrix, which contains information on how much we want to update the original weight parameters to minimize the loss function during training. Hereafter, I use the term “weight” as shorthand for the model’s weight parameters.

In regular training and fine-tuning, the weight update is defined as follows:

$$W_{updated} = W + \Delta W$$

The LoRA method, proposed by Hu et al. (<https://arxiv.org/abs/2106.09685>), offers a more efficient alternative to computing the weight updates ΔW by learning an approximation of it:

$$\Delta W \approx AB$$

where A and B are two matrices much smaller than W , and AB represents the matrix multiplication product between A and B .

Using LoRA, we can then reformulate the weight update we defined earlier:

$$W_{\text{updated}} = W + AB$$

Figure E.1 illustrates the weight update formulas for full fine-tuning and LoRA side by side.

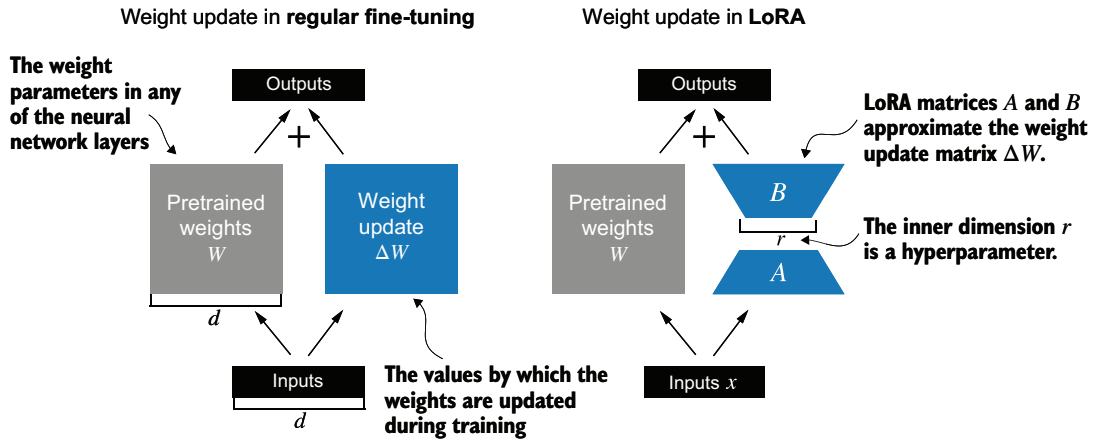


Figure E.1 A comparison between weight update methods: regular fine-tuning and LoRA. Regular fine-tuning involves updating the pretrained weight matrix W directly with ΔW (left). LoRA uses two smaller matrices, A and B , to approximate ΔW , where the product AB is added to W , and r denotes the inner dimension, a tunable hyperparameter (right).

If you paid close attention, you might have noticed that the visual representations of full fine-tuning and LoRA in figure E.1 differ slightly from the earlier presented formulas. This variation is attributed to the distributive law of matrix multiplication, which allows us to separate the original and updated weights rather than combine them. For example, in the case of regular fine-tuning with x as the input data, we can express the computation as

$$x(W + \Delta W) = xW + x\Delta W$$

Similarly, we can write the following for LoRA:

$$x(W + AB) = xW + xAB$$

Besides reducing the number of weights to update during training, the ability to keep the LoRA weight matrices separate from the original model weights makes LoRA even more useful in practice. Practically, this allows for the pretrained model weights to remain unchanged, with the LoRA matrices being applied dynamically after training when using the model.

Keeping the LoRA weights separate is very useful in practice because it enables model customization without needing to store multiple complete versions of an LLM. This reduces storage requirements and improves scalability, as only the smaller LoRA matrices need to be adjusted and saved when we customize LLMs for each specific customer or application.

Next, let's see how LoRA can be used to fine-tune an LLM for spam classification, similar to the fine-tuning example in chapter 6.

E.2 *Preparing the dataset*

Before applying LoRA to the spam classification example, we must load the dataset and pretrained model we will work with. The code here repeats the data preparation from chapter 6. (Instead of repeating the code, we could open and run the chapter 6 notebook and insert the LoRA code from section E.4 there.)

First, we download the dataset and save it as CSV files.

Listing E.1 Downloading and preparing the dataset

```
from pathlib import Path
import pandas as pd
from ch06 import (
    download_and_unzip_spam_data,
    create_balanced_dataset,
    random_split
)

url = \
    "https://archive.ics.uci.edu/static/public/228/sms+spam+collection.zip"
zip_path = "sms_spam_collection.zip"
extracted_path = "sms_spam_collection"
data_file_path = Path(extracted_path) / "SMSSpamCollection.tsv"

download_and_unzip_spam_data(url, zip_path, extracted_path, data_file_path)

df = pd.read_csv(
    data_file_path, sep="\t", header=None, names=["Label", "Text"]
)
balanced_df = create_balanced_dataset(df)
balanced_df["Label"] = balanced_df["Label"].map({"ham": 0, "spam": 1})

train_df, validation_df, test_df = random_split(balanced_df, 0.7, 0.1)
train_df.to_csv("train.csv", index=None)
```

```
validation_df.to_csv("validation.csv", index=None)
test_df.to_csv("test.csv", index=None)
```

Next, we create the SpamDataset instances.

Listing E.2 Instantiating PyTorch datasets

```
import torch
from torch.utils.data import Dataset
import tiktoken
from chapter06 import SpamDataset

tokenizer = tiktoken.get_encoding("gpt2")
train_dataset = SpamDataset("train.csv", max_length=None,
                             tokenizer=tokenizer
)
val_dataset = SpamDataset("validation.csv",
                           max_length=train_dataset.max_length, tokenizer=tokenizer
)
test_dataset = SpamDataset(
    "test.csv", max_length=train_dataset.max_length, tokenizer=tokenizer
)
```

After creating the PyTorch dataset objects, we instantiate the data loaders.

Listing E.3 Creating PyTorch data loaders

```
from torch.utils.data import DataLoader

num_workers = 0
batch_size = 8

torch.manual_seed(123)

train_loader = DataLoader(
    dataset=train_dataset,
    batch_size=batch_size,
    shuffle=True,
    num_workers=num_workers,
    drop_last=True,
)

val_loader = DataLoader(
    dataset=val_dataset,
    batch_size=batch_size,
    num_workers=num_workers,
    drop_last=False,
)

test_loader = DataLoader(
    dataset=test_dataset,
    batch_size=batch_size,
    num_workers=num_workers,
    drop_last=False,
)
```

As a verification step, we iterate through the data loaders and check that the batches contain eight training examples each, where each training example consists of 120 tokens:

```
print("Train loader:")
for input_batch, target_batch in train_loader:
    pass

print("Input batch dimensions:", input_batch.shape)
print("Label batch dimensions", target_batch.shape)
```

The output is

```
Train loader:
Input batch dimensions: torch.Size([8, 120])
Label batch dimensions torch.Size([8])
```

Lastly, we print the total number of batches in each dataset:

```
print(f"{len(train_loader)} training batches")
print(f"{len(val_loader)} validation batches")
print(f"{len(test_loader)} test batches")
```

In this case, we have the following number of batches per dataset:

```
130 training batches
19 validation batches
38 test batches
```

E.3 *Initializing the model*

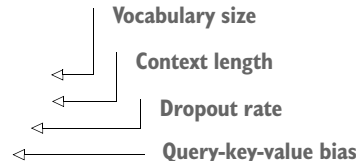
We repeat the code from chapter 6 to load and prepare the pretrained GPT model. We begin by downloading the model weights and loading them into the `GPTModel` class.

Listing E.4 Loading a pretrained GPT model

```
from gpt_download import download_and_load_gpt2
from chapter04 import GPTModel
from chapter05 import load_weights_into_gpt

CHOOSE_MODEL = "gpt2-small (124M)"
INPUT_PROMPT = "Every effort moves"

BASE_CONFIG = {
    "vocab_size": 50257,
    "context_length": 1024,
    "drop_rate": 0.0,
    "qkv_bias": True
}
```



```

model_configs = {
    "gpt2-small (124M)": {"emb_dim": 768, "n_layers": 12, "n_heads": 12},
    "gpt2-medium (355M)": {"emb_dim": 1024, "n_layers": 24, "n_heads": 16},
    "gpt2-large (774M)": {"emb_dim": 1280, "n_layers": 36, "n_heads": 20},
    "gpt2-xl (1558M)": {"emb_dim": 1600, "n_layers": 48, "n_heads": 25},
}

BASE_CONFIG.update(model_configs[CHOOSE_MODEL])

model_size = CHOOSE_MODEL.split(" ")[-1].lstrip("(").rstrip(")")
settings, params = download_and_load_gpt2(
    model_size=model_size, models_dir="gpt2"
)

model = GPTModel(BASE_CONFIG)
load_weights_into_gpt(model, params)
model.eval()

```

To ensure that the model was loaded corrected, let's double-check that it generates coherent text:

```

from chapter04 import generate_text_simple
from chapter05 import text_to_token_ids, token_ids_to_text

text_1 = "Every effort moves you"

token_ids = generate_text_simple(
    model=model,
    idx=text_to_token_ids(text_1, tokenizer),
    max_new_tokens=15,
    context_size=BASE_CONFIG["context_length"]
)

print(token_ids_to_text(token_ids, tokenizer))

```

The following output shows that the model generates coherent text, which is an indicator that the model weights are loaded correctly:

```

Every effort moves you forward.
The first step is to understand the importance of your work

```

Next, we prepare the model for classification fine-tuning, similar to chapter 6, where we replace the output layer:

```

torch.manual_seed(123)
num_classes = 2
model.out_head = torch.nn.Linear(in_features=768, out_features=num_classes)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)

```

Lastly, we calculate the initial classification accuracy of the not-fine-tuned model (we expect this to be around 50%, which means that the model is not able to distinguish between spam and nonspam messages yet reliably):

```

from chapter06 import calc_accuracy_loader

torch.manual_seed(123)
train_accuracy = calc_accuracy_loader(
    train_loader, model, device, num_batches=10
)
val_accuracy = calc_accuracy_loader(
    val_loader, model, device, num_batches=10
)
test_accuracy = calc_accuracy_loader(
    test_loader, model, device, num_batches=10
)

print(f"Training accuracy: {train_accuracy*100:.2f}%")
print(f"Validation accuracy: {val_accuracy*100:.2f}%")
print(f"Test accuracy: {test_accuracy*100:.2f}%")

```

The initial prediction accuracies are

```

Training accuracy: 46.25%
Validation accuracy: 45.00%
Test accuracy: 48.75%

```

E.4 *Parameter-efficient fine-tuning with LoRA*

Next, we modify and fine-tune the LLM using LoRA. We begin by initializing a LoRA-Layer that creates the matrices A and B , along with the alpha scaling factor and the rank (r) setting. This layer can accept an input and compute the corresponding output, as illustrated in figure E.2.

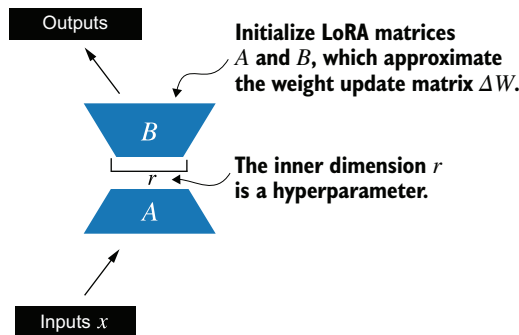


Figure E.2 The LoRA matrices A and B are applied to the layer inputs and are involved in computing the model outputs. The inner dimension r of these matrices serves as a setting that adjusts the number of trainable parameters by varying the sizes of A and B .

In code, this LoRA layer can be implemented as follows.

Listing E.5 Implementing a LoRA layer

```
import math

class LoRALayer(torch.nn.Module):
    def __init__(self, in_dim, out_dim, rank, alpha):
        super().__init__()
        self.A = torch.nn.Parameter(torch.empty(in_dim, rank))
        torch.nn.init.kaiming_uniform_(self.A, a=math.sqrt(5))
        self.B = torch.nn.Parameter(torch.zeros(rank, out_dim))
        self.alpha = alpha

    def forward(self, x):
        x = self.alpha * (x @ self.A @ self.B)
        return x
```

←
The same initialization
used for Linear layers
in PyTorch

The rank governs the inner dimension of matrices A and B . Essentially, this setting determines the number of extra parameters introduced by LoRA, which creates balance between the adaptability of the model and its efficiency via the number of parameters used.

The other important setting, α , functions as a scaling factor for the output from the low-rank adaptation. It primarily dictates the degree to which the output from the adapted layer can affect the original layer's output. This can be seen as a way to regulate the effect of the low-rank adaptation on the layer's output. The `LoRALayer` class we have implemented so far enables us to transform the inputs of a layer.

In LoRA, the typical goal is to substitute existing Linear layers, allowing weight updates to be applied directly to the pre-existing pretrained weights, as illustrated in figure E.3.

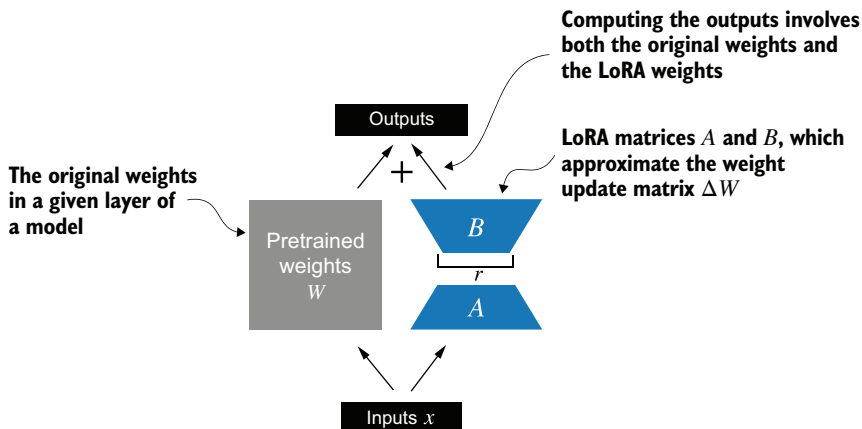


Figure E.3 The integration of LoRA into a model layer. The original pretrained weights (W) of a layer are combined with the outputs from LoRA matrices (A and B), which approximate the weight update matrix (ΔW). The final output is calculated by adding the output of the adapted layer (using LoRA weights) to the original output.

To integrate the original `Linear` layer weights, we now create a `LinearWithLoRA` layer. This layer utilizes the previously implemented `LoRALayer` and is designed to replace existing `Linear` layers within a neural network, such as the self-attention modules or feed-forward modules in the `GPTModel`.

Listing E.6 Replacing a `LinearWithLora` layer with `Linear` layers

```
class LinearWithLoRA(torch.nn.Module):
    def __init__(self, linear, rank, alpha):
        super().__init__()
        self.linear = linear
        self.lora = LoRALayer(
            linear.in_features, linear.out_features, rank, alpha
        )

    def forward(self, x):
        return self.linear(x) + self.lora(x)
```

This code combines a standard `Linear` layer with the `LoRALayer`. The `forward` method computes the output by adding the results from the original linear layer and the LoRA layer.

Since the weight matrix B (`self.B` in `LoRALayer`) is initialized with zero values, the product of matrices A and B results in a zero matrix. This ensures that the multiplication does not alter the original weights, as adding zero does not change them.

To apply LoRA to the earlier defined `GPTModel`, we introduce a `replace_linear_with_lora` function. This function will swap all existing `Linear` layers in the model with the newly created `LinearWithLoRA` layers:

```
def replace_linear_with_lora(model, rank, alpha):
    for name, module in model.named_children():
        if isinstance(module, torch.nn.Linear):
            setattr(model, name, LinearWithLoRA(module, rank, alpha))
        else:
            replace_linear_with_lora(module, rank, alpha)
```

Replaces the Linear layer with LinearWithLoRA

Recursively applies the same function to child modules

We have now implemented all the necessary code to replace the `Linear` layers in the `GPTModel` with the newly developed `LinearWithLoRA` layers for parameter-efficient fine-tuning. Next, we will apply the `LinearWithLoRA` upgrade to all `Linear` layers found in the multihead attention, feed-forward modules, and the output layer of the `GPTModel`, as shown in figure E.4.

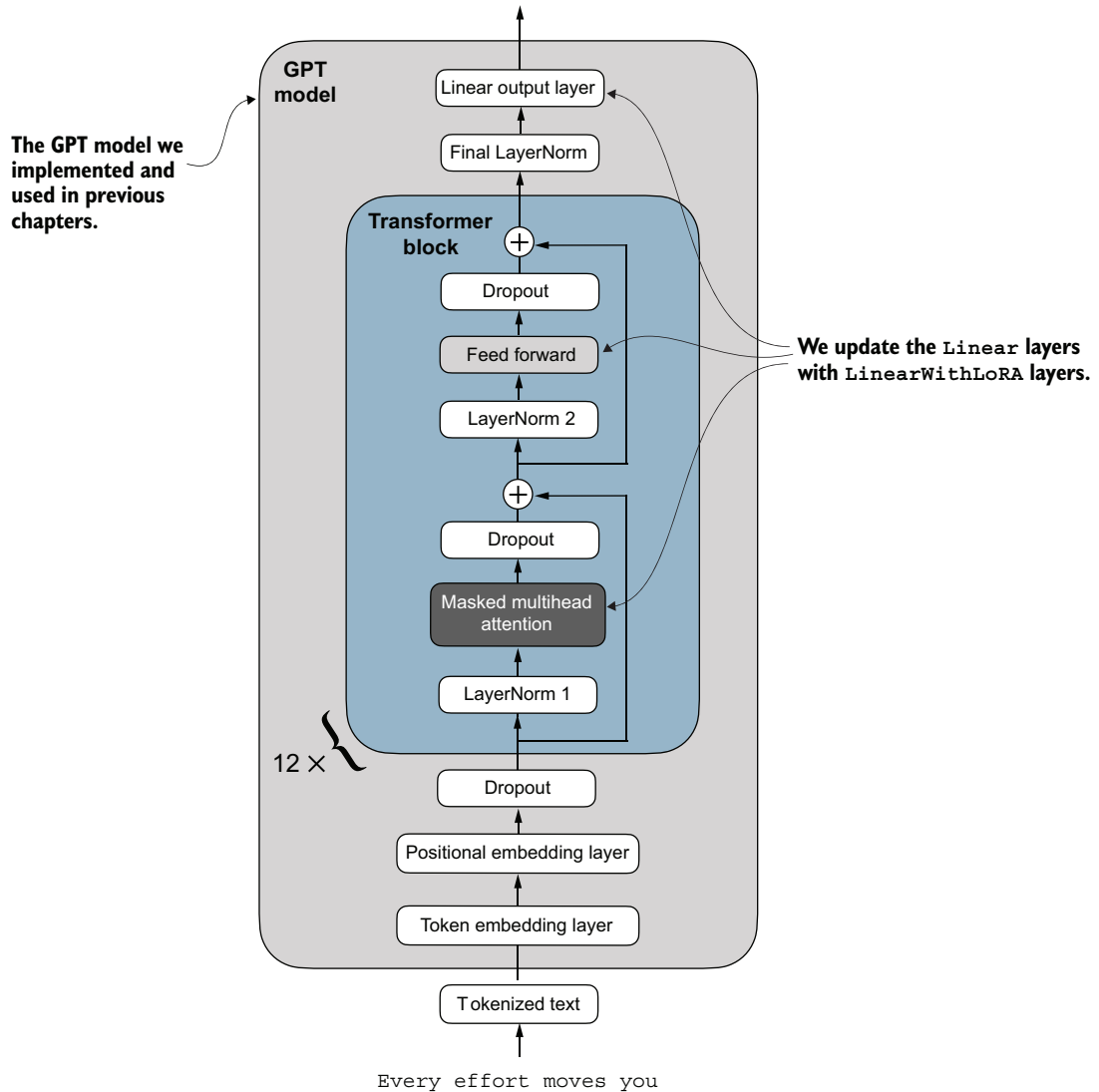


Figure E.4 The architecture of the GPT model. It highlights the parts of the model where Linear layers are upgraded to LinearWithLoRA layers for parameter-efficient fine-tuning.

Before we apply the LinearWithLoRA layer upgrades, we first freeze the original model parameters:

```
total_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f"Total trainable parameters before: {total_params:,}")
```

```
for param in model.parameters():
    param.requires_grad = False
```

```
total_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f"Total trainable parameters after: {total_params:,}")
```

Now, we can see that none of the 124 million model parameters are trainable:

```
Total trainable parameters before: 124,441,346
Total trainable parameters after: 0
```

Next, we use the `replace_linear_with_lora` to replace the Linear layers:

```
replace_linear_with_lora(model, rank=16, alpha=16)
total_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f"Total trainable LoRA parameters: {total_params:,}")
```

After adding the LoRA layers, the number of trainable parameters is as follows:

```
Total trainable LoRA parameters: 2,666,528
```

As we can see, we reduced the number of trainable parameters by almost 50× when using LoRA. A rank and alpha of 16 are good default choices, but it is also common to increase the rank parameter, which in turn increases the number of trainable parameters. Alpha is usually chosen to be half, double, or equal to the rank.

Let's verify that the layers have been modified as intended by printing the model architecture:

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
print(model)
```

The output is

```
GPTModel(
  (tok_emb): Embedding(50257, 768)
  (pos_emb): Embedding(1024, 768)
  (drop_emb): Dropout(p=0.0, inplace=False)
  (trf_blocks): Sequential(
    ...
    (11): TransformerBlock(
      (att): MultiHeadAttention(
        (W_query): LinearWithLoRA(
          (linear): Linear(in_features=768, out_features=768, bias=True)
          (lora): LoRALayer()
        )
        (W_key): LinearWithLoRA(
          (linear): Linear(in_features=768, out_features=768, bias=True)
          (lora): LoRALayer()
        )
        (W_value): LinearWithLoRA(
          (linear): Linear(in_features=768, out_features=768, bias=True)
          (lora): LoRALayer()
        )
      )
    )
  )
```

```

(out_proj): LinearWithLoRA(
  (linear): Linear(in_features=768, out_features=768, bias=True)
  (lora): LoRALayer()
)
(dropout): Dropout(p=0.0, inplace=False)
)
(ff): FeedForward(
  (layers): Sequential(
    (0): LinearWithLoRA(
      (linear): Linear(in_features=768, out_features=3072, bias=True)
      (lora): LoRALayer()
    )
    (1): GELU()
    (2): LinearWithLoRA(
      (linear): Linear(in_features=3072, out_features=768, bias=True)
      (lora): LoRALayer()
    )
  )
)
(norm1): LayerNorm()
(norm2): LayerNorm()
(drop_resid): Dropout(p=0.0, inplace=False)
)
)
(final_norm): LayerNorm()
(out_head): LinearWithLoRA(
  (linear): Linear(in_features=768, out_features=2, bias=True)
  (lora): LoRALayer()
)
)
)

```

The model now includes the new `LinearWithLoRA` layers, which themselves consist of the original `Linear` layers, set to nontrainable, and the new `LoRA` layers, which we will fine-tune.

Before we begin fine-tuning the model, let's calculate the initial classification accuracy:

```

torch.manual_seed(123)

train_accuracy = calc_accuracy_loader(
    train_loader, model, device, num_batches=10
)
val_accuracy = calc_accuracy_loader(
    val_loader, model, device, num_batches=10
)
test_accuracy = calc_accuracy_loader(
    test_loader, model, device, num_batches=10
)

print(f"Training accuracy: {train_accuracy*100:.2f}%")
print(f"Validation accuracy: {val_accuracy*100:.2f}%")
print(f"Test accuracy: {test_accuracy*100:.2f}%")

```

The resulting accuracy values are

```
Training accuracy: 46.25%
Validation accuracy: 45.00%
Test accuracy: 48.75%
```

These accuracy values are identical to the values from chapter 6. This result occurs because we initialized the LoRA matrix B with zeros. Consequently, the product of matrices AB results in a zero matrix. This ensures that the multiplication does not alter the original weights since adding zero does not change them.

Now let's move on to the exciting part—fine-tuning the model using the training function from chapter 6. The training takes about 15 minutes on an M3 MacBook Air laptop and less than half a minute on a V100 or A100 GPU.

Listing E.7 Fine-tuning a model with LoRA layers

```
import time
from chapter06 import train_classifier_simple

start_time = time.time()
torch.manual_seed(123)
optimizer = torch.optim.AdamW(model.parameters(), lr=5e-5, weight_decay=0.1)

num_epochs = 5
train_losses, val_losses, train_accs, val_accs, examples_seen = \
    train_classifier_simple(
        model, train_loader, val_loader, optimizer, device,
        num_epochs=num_epochs, eval_freq=50, eval_iter=5,
        tokenizer=tokenizer
    )

end_time = time.time()
execution_time_minutes = (end_time - start_time) / 60
print(f"Training completed in {execution_time_minutes:.2f} minutes.")
```

The output we see during the training is

```
Ep 1 (Step 000000): Train loss 3.820, Val loss 3.462
Ep 1 (Step 000050): Train loss 0.396, Val loss 0.364
Ep 1 (Step 000100): Train loss 0.111, Val loss 0.229
Training accuracy: 97.50% | Validation accuracy: 95.00%
Ep 2 (Step 000150): Train loss 0.135, Val loss 0.073
Ep 2 (Step 000200): Train loss 0.008, Val loss 0.052
Ep 2 (Step 000250): Train loss 0.021, Val loss 0.179
Training accuracy: 97.50% | Validation accuracy: 97.50%
Ep 3 (Step 000300): Train loss 0.096, Val loss 0.080
Ep 3 (Step 000350): Train loss 0.010, Val loss 0.116
Training accuracy: 97.50% | Validation accuracy: 95.00%
Ep 4 (Step 000400): Train loss 0.003, Val loss 0.151
Ep 4 (Step 000450): Train loss 0.008, Val loss 0.077
Ep 4 (Step 000500): Train loss 0.001, Val loss 0.147
Training accuracy: 100.00% | Validation accuracy: 97.50%
```

```
Ep 5 (Step 000550): Train loss 0.007, Val loss 0.094
Ep 5 (Step 000600): Train loss 0.000, Val loss 0.056
Training accuracy: 100.00% | Validation accuracy: 97.50%

Training completed in 12.10 minutes.
```

Training the model with LoRA took longer than training it without LoRA (see chapter 6) because the LoRA layers introduce an additional computation during the forward pass. However, for larger models, where backpropagation becomes more costly, models typically train faster with LoRA than without it.

As we can see, the model received perfect training and very high validation accuracy. Let's also visualize the loss curves to better see whether the training has converged:

```
from chapter06 import plot_values

epochs_tensor = torch.linspace(0, num_epochs, len(train_losses))
examples_seen_tensor = torch.linspace(0, examples_seen, len(train_losses))

plot_values(
    epochs_tensor, examples_seen_tensor,
    train_losses, val_losses, label="loss"
)
```

Figure E.5 plots the results.

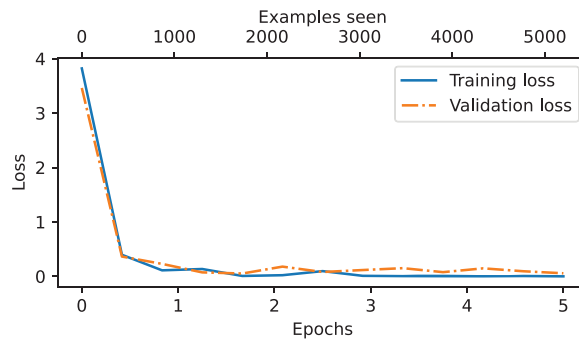


Figure E.5 The training and validation loss curves over six epochs for a machine learning model. Initially, both training and validation loss decrease sharply and then they level off, indicating the model is converging, which means that it is not expected to improve noticeably with further training.

In addition to evaluating the model based on the loss curves, let's also calculate the accuracies on the full training, validation, and test set (during the training, we approximated the training and validation set accuracies from five batches via the `eval_iter=5` setting):

```
train_accuracy = calc_accuracy_loader(train_loader, model, device)
val_accuracy = calc_accuracy_loader(val_loader, model, device)
test_accuracy = calc_accuracy_loader(test_loader, model, device)

print(f"Training accuracy: {train_accuracy*100:.2f}%")
print(f"Validation accuracy: {val_accuracy*100:.2f}%")
print(f"Test accuracy: {test_accuracy*100:.2f}%")
```

The resulting accuracy values are

```
Training accuracy: 100.00%  
Validation accuracy: 96.64%  
Test accuracy: 98.00%
```

These results show that the model performs well across training, validation, and test datasets. With a training accuracy of 100%, the model has perfectly learned the training data. However, the slightly lower validation and test accuracies (96.64% and 97.33%, respectively) suggest a small degree of overfitting, as the model does not generalize quite as well on unseen data compared to the training set. Overall, the results are very impressive, considering we fine-tuned only a relatively small number of model weights (2.7 million LoRA weights instead of the original 124 million model weights).