gradients, and updating weights and concluding with monitoring steps like printing losses and generating text samples.

> **NOTE** If you are relatively new to training deep neural networks with PyTorch and any of these steps are unfamiliar, consider reading sections A.5 to A.8 in appendix A.

We can implement this training flow via the `train_model_simple` function in code.

Listing 5.3 The main function for pretraining LLMs

```
def train_model_simple(model, train_loader, val_loader,
                       optimizer, device, num_epochs,
                       eval_freq, eval_iter, start_context, tokenizer):
    train_losses, val_losses, track_tokens_seen = [], [], []
    tokens_seen, global_step = 0, -1

    for epoch in range(num_epochs):
        model.train()
        for input_batch, target_batch in train_loader:
            optimizer.zero_grad()
            loss = calc_loss_batch(
                input_batch, target_batch, model, device
            )
            loss.backward()
            optimizer.step()
            tokens_seen += input_batch.numel()
            global_step += 1

            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} (Step {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
```

Initializes lists to track losses and tokens seen

Starts the main training loop

Resets loss gradients from the previous batch iteration

Calculates loss gradients

Updates model weights using loss gradients

Optional evaluation step

Prints a sample text after each epoch

Note that the `train_model_simple` function we just created uses two functions we have not defined yet: `evaluate_model` and `generate_and_print_sample`.

The `evaluate_model` function corresponds to step 7 in figure 5.11. It prints the training and validation set losses after each model update so we can evaluate whether the training improves the model. More specifically, the `evaluate_model` function calculates the loss over the training and validation set while ensuring the model is in eval-

uation mode with gradient tracking and dropout disabled when calculating the loss over the training and validation sets:

Dropout is disabled during evaluation for stable, reproducible results.

Disables gradient tracking, which is not required during evaluation, to reduce the computational overhead

```
def evaluate_model(model, train_loader, val_loader, device, eval_iter):
    model.eval()
    with torch.no_grad():
        train_loss = calc_loss_loader(
            train_loader, model, device, num_batches=eval_iter
        )
        val_loss = calc_loss_loader(
            val_loader, model, device, num_batches=eval_iter
        )
    model.train()
    return train_loss, val_loss
```

Similar to `evaluate_model`, the `generate_and_print_sample` function is a convenience function that we use to track whether the model improves during the training. In particular, the `generate_and_print_sample` function takes a text snippet (`start_context`) as input, converts it into token IDs, and feeds it to the LLM to generate a text sample using the `generate_text_simple` function we used earlier:

```
def generate_and_print_sample(model, tokenizer, device, start_context):
    model.eval()
    context_size = model.pos_emb.weight.shape[0]
    encoded = text_to_token_ids(start_context, tokenizer).to(device)
    with torch.no_grad():
        token_ids = generate_text_simple(
            model=model, idx=encoded,
            max_new_tokens=50, context_size=context_size
        )
    decoded_text = token_ids_to_text(token_ids, tokenizer)
    print(decoded_text.replace("\n", " "))
    model.train()
```

Compact print format

While the `evaluate_model` function gives us a numeric estimate of the model's training progress, this `generate_and_print_sample` text function provides a concrete text example generated by the model to judge its capabilities during training.

AdamW

Adam optimizers are a popular choice for training deep neural networks. However, in our training loop, we opt for the *AdamW* optimizer. AdamW is a variant of Adam that improves the weight decay approach, which aims to minimize model complexity and prevent overfitting by penalizing larger weights. This adjustment allows AdamW to achieve more effective regularization and better generalization; thus, AdamW is frequently used in the training of LLMs.

Let's see this all in action by training a `GPTModel` instance for 10 epochs using an `AdamW` optimizer and the `train_model_simple` function we defined earlier:

```
torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)
model.to(device)
optimizer = torch.optim.AdamW(
     model.parameters(),
    lr=0.0004, weight_decay=0.1
)
num_epochs = 10
train_losses, val_losses, tokens_seen = train_model_simple(
    model, train_loader, val_loader, optimizer, device,
    num_epochs=num_epochs, eval_freq=5, eval_iter=5,
    start_context="Every effort moves you", tokenizer=tokenizer
)
```

The .parameters() method returns all trainable weight parameters of the model.

Executing the `train_model_simple` function starts the training process, which takes about 5 minutes to complete on a MacBook Air or a similar laptop. The output printed during this execution is as follows:

```
Ep 1 (Step 000000): Train loss 9.781, Val loss 9.933
Ep 1 (Step 000005): Train loss 8.111, Val loss 8.339
Every effort moves you,,,,,,,,,,,,.
Ep 2 (Step 000010): Train loss 6.661, Val loss 7.048
Ep 2 (Step 000015): Train loss 5.961, Val loss 6.616
Every effort moves you, and, and, and, and, and, and, and, and, and, and,
 and, and, and, and, and, and, and, and, and, and, and, and,, and, and,
[...]
Ep 9 (Step 000080): Train loss 0.541, Val loss 6.393
Every effort moves you?"  "Yes--quite insensible to the irony. She wanted
him vindicated--and by me!"  He laughed again, and threw back the
window-curtains, I had the donkey. "There were days when I
Ep 10 (Step 000085): Train loss 0.391, Val loss 6.452
Every effort moves you know," was one of the axioms he laid down across the
Sevres and silver of an exquisitely appointed luncheon-table, when, on a
later day, I had again run over from Monte Carlo; and Mrs. Gis
```

Intermediate results removed to save space

As we can see, the training loss improves drastically, starting with a value of 9.781 and converging to 0.391. The language skills of the model have improved quite a lot. In the beginning, the model is only able to append commas to the start context (`Every effort moves you,,,,,,,,,,,,`) or repeat the word `and`. At the end of the training, it can generate grammatically correct text.

Similar to the training set loss, we can see that the validation loss starts high (9.933) and decreases during the training. However, it never becomes as small as the training set loss and remains at 6.452 after the 10th epoch.

Before discussing the validation loss in more detail, let's create a simple plot that shows the training and validation set losses side by side: