```
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 LoRALayer 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.

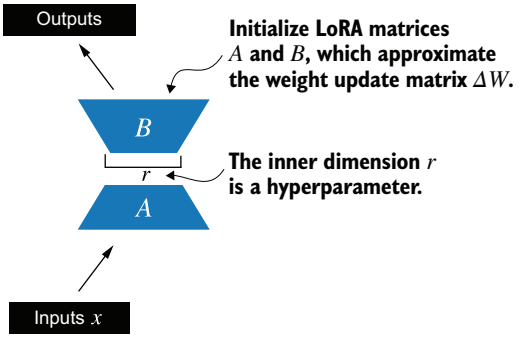

Figure E.2 The LoRA matrices *A* and *B* are applied to the layer inputs and are involved in computing the model outputs. The inner dimension *r* of these matrices serves as a setting that adjusts the number of trainable parameters by varying the sizes of *A* and *B*.

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, `alpha`, 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.

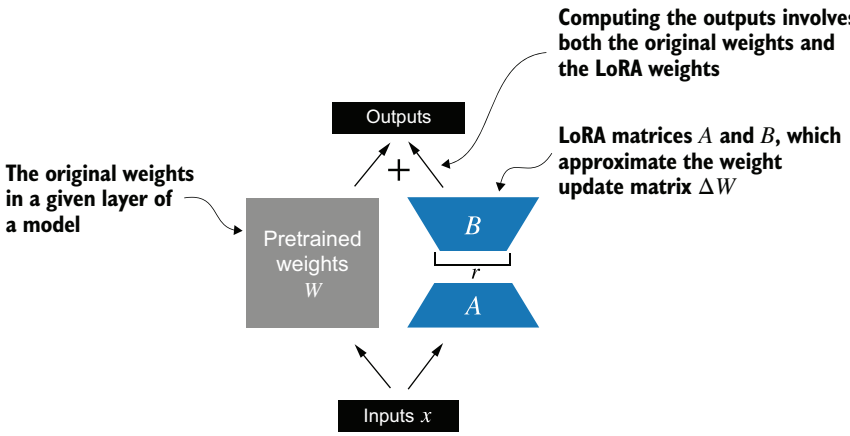

Figure E.3 The integration of LoRA into a model layer. The original pretrained weights (*W*) of a layer are combined with the outputs from LoRA matrices (*A* and *B*), which approximate the weight update matrix (Δ*W*). The final output is calculated by adding the output of the adapted layer (using LoRA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