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).

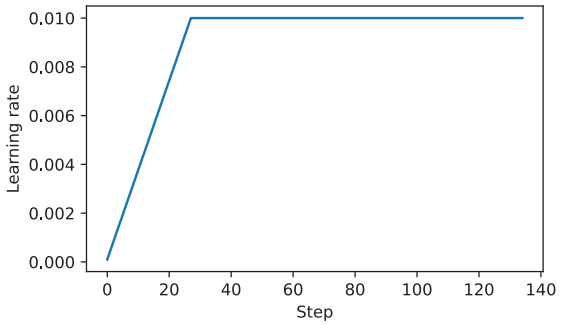

Figure D.1 The learning rate warmup increases the learning rate for the first 20 training steps. After 20 steps, the learning rate reaches the peak of 0.01 and remains constant for the rest of the training.

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).

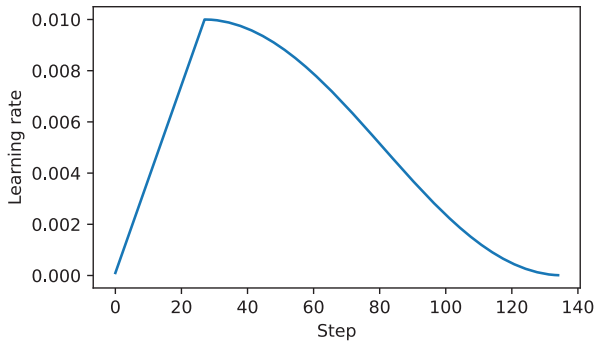

Figure D.2 The first 20 steps of linear learning rate warmup are followed by a cosine decay, which reduces the learning rate in a half-cosine cycle until it reaches its minimum point at the end of training.

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 downscaled 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 $\boldsymbol{v}$ composed of components $\boldsymbol{v} = [v_1, v_2, ..., v_n]$, the L2 norm is

$$|\boldsymbol{v}|_2 = \sqrt{v_1^2 + v_2^2 + \cdots + v_n^2}$$

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`$/|\mathbf{G}|_2 = 1/5$. Consequently, the adjusted gradient matrix **G'** becomes

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

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))
```