Top-k sampling, when combined with probabilistic sampling and temperature scaling, can improve the text generation results. In top-k sampling, we can restrict the sampled tokens to the top-k most likely tokens and exclude all other tokens from the selection process by masking their probability scores, as illustrated in figure 5.15.

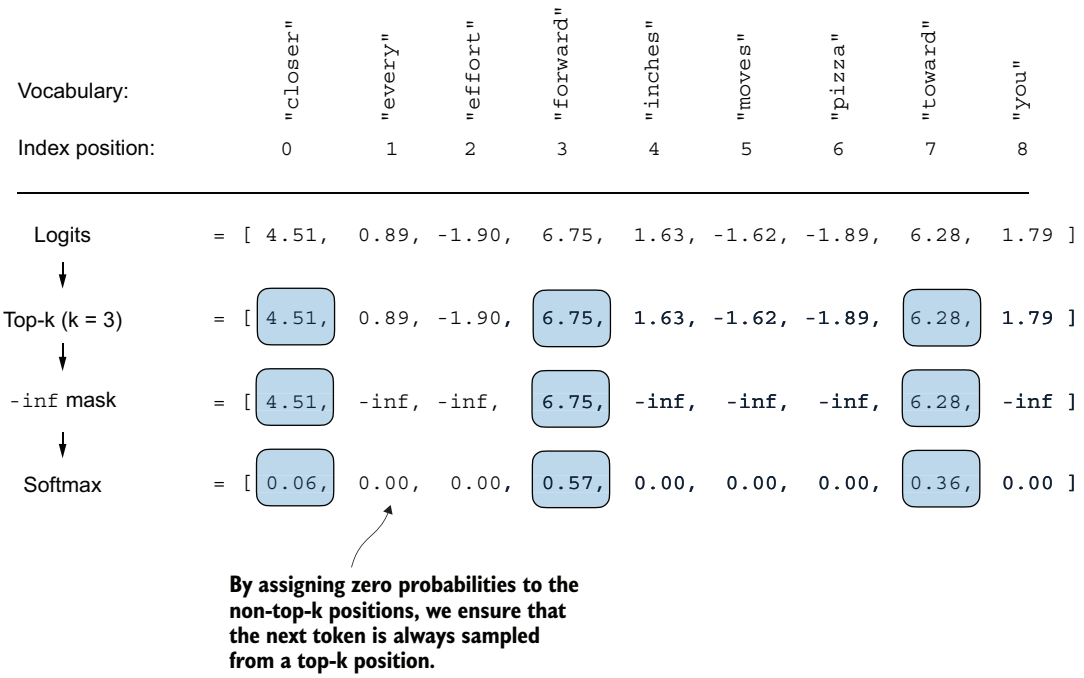

Figure 5.15 Using top-k sampling with k = 3, we focus on the three tokens associated with the highest logits and mask out all other tokens with negative infinity (`-inf`) before applying the `softmax` function. This results in a probability distribution with a probability value 0 assigned to all non-top-k tokens. (The numbers in this figure are truncated to two digits after the decimal point to reduce visual clutter. The values in the "Softmax" row should add up to 1.0.)

The top-k approach replaces all nonselected logits with negative infinity value (`-inf`), such that when computing the softmax values, the probability scores of the non-top-k tokens are 0, and the remaining probabilities sum up to 1. (Careful readers may remember this masking trick from the causal attention module we implemented in chapter 3, section 3.5.1.)

In code, we can implement the top-k procedure in figure 5.15 as follows, starting with the selection of the tokens with the largest logit values:

```
top_k = 3
top_logits, top_pos = torch.topk(next_token_logits, top_k)
print("Top logits:", top_logits)
print("Top positions:", top_pos)
```

The logits values and token IDs of the top three tokens, in descending order, are

```
Top logits: tensor([6.7500, 6.2800, 4.5100])
Top positions: tensor([3, 7, 0])
```

Subsequently, we apply PyTorch's `where` function to set the logit values of tokens that are below the lowest logit value within our top-three selection to negative infinity (`-inf`):

```
new_logits = torch.where(
    condition=next_token_logits < top_logits[-1],
    input=torch.tensor(float('-inf')),
    other=next_token_logits
)
print(new_logits)
```

Identifies logits less than the minimum in the top 3

Assigns –inf to these lower logits

Retains the original logits for all other tokens

The resulting logits for the next token in the nine-token vocabulary are

```
tensor([4.5100,   -inf,   -inf, 6.7500,   -inf,   -inf,   -inf, 6.2800,
     -inf])
```

Lastly, let's apply the `softmax` function to turn these into next-token probabilities:

```
topk_probas = torch.softmax(new_logits, dim=0)
print(topk_probas)
```

As we can see, the result of this top-three approach are three non-zero probability scores:

```
tensor([0.0615, 0.0000, 0.0000, 0.5775, 0.0000, 0.0000, 0.0000, 0.3610,
        0.0000])
```

We can now apply the temperature scaling and multinomial function for probabilistic sampling to select the next token among these three non-zero probability scores to generate the next token. We do this next by modifying the text generation function.

5.3.3 Modifying the text generation function

Now, let's combine temperature sampling and top-k sampling to modify the `generate_text_simple` function we used to generate text via the LLM earlier, creating a new `generate` function.

Listing 5.4 A modified text generation function with more diversity

```
def generate(model, idx, max_new_tokens, context_size,
             temperature=0.0, top_k=None, eos_id=None):
    for _ in range(max_new_tokens):
        idx_cond = idx[:, -context_size:]
        with torch.no_grad():
            logits = model(idx_cond)
        logits = logits[:, -1, :]
```

The for loop is the same as before: gets logits and only focuses on the last time step.

```
        if top_k is not None:                                    # Filters logits with top_k sampling
            top_logits, _ = torch.topk(logits, top_k)
            min_val = top_logits[:, -1]
            logits = torch.where(
                logits < min_val,
                torch.tensor(float('-inf')).to(logits.device),
                logits
            )
        if temperature > 0.0:                                    # Applies temperature scaling
            logits = logits / temperature
            probs = torch.softmax(logits, dim=-1)
            idx_next = torch.multinomial(probs, num_samples=1)
        else:                                                    # Carries out greedy next-token selection as before when temperature scaling is disabled
            idx_next = torch.argmax(logits, dim=-1, keepdim=True)
        if idx_next == eos_id:                                   # Stops generating early if end-of-sequence token is encountered
            break
        idx = torch.cat((idx, idx_next), dim=1)
    return idx
```

Let's now see this new `generate` function in action:

```
torch.manual_seed(123)
token_ids = generate(
    model=model,
    idx=text_to_token_ids("Every effort moves you", tokenizer),
    max_new_tokens=15,
    context_size=GPT_CONFIG_124M["context_length"],
    top_k=25,
    temperature=1.4
)
print("Output text:\n", token_ids_to_text(token_ids, tokenizer))
```

The generated text is

```
Output text:
 Every effort moves you stand to work on surprise, a one of us had gone
 with random-
```

As we can see, the generated text is very different from the one we previously generated via the `generate_simple` function in section 5.3 (`"Every effort moves you know," was one of the axioms he laid...!`), which was a memorized passage from the training set.

Exercise 5.2

Play around with different temperatures and top-k settings. Based on your observations, can you think of applications where lower temperature and top-k settings are desired? Likewise, can you think of applications where higher temperature and top-k settings are preferred? (It's recommended to also revisit this exercise at the end of the chapter after loading the pretrained weights from OpenAI.)