

Listing 7.5 Implementing a custom batch collate function

```

def custom_collate_fn(
    batch,
    pad_token_id=50256,
    ignore_index=-100,
    allowed_max_length=None,
    device="cpu"
):
    batch_max_length = max(len(item)+1 for item in batch)
    inputs_lst, targets_lst = [], []

    for item in batch:
        new_item = item.copy()
        new_item += [pad_token_id]

        padded = (
            new_item + [pad_token_id] *
            (batch_max_length - len(new_item))
        )
        inputs = torch.tensor(padded[:-1])
        targets = torch.tensor(padded[1:])

        mask = targets == pad_token_id
        indices = torch.nonzero(mask).squeeze()
        if indices.numel() > 1:
            targets[indices[1:]] = ignore_index

        if allowed_max_length is not None:
            inputs = inputs[:allowed_max_length]
            targets = targets[:allowed_max_length]

        inputs_lst.append(inputs)
        targets_lst.append(targets)

    inputs_tensor = torch.stack(inputs_lst).to(device)
    targets_tensor = torch.stack(targets_lst).to(device)
    return inputs_tensor, targets_tensor

```

Pads sequences to max_length

Truncates the last token for inputs

Shifts +1 to the right for targets

Replaces all but the first padding tokens in targets by ignore_index

Optionally truncates to the maximum sequence length

Again, let's try the collate function on the sample batch that we created earlier to check that it works as intended:

```

inputs, targets = custom_collate_fn(batch)
print(inputs)
print(targets)

```

The results are as follows, where the first tensor represents the inputs and the second tensor represents the targets:

```

tensor([[ 0,      1,      2,      3,      4,
         5,      6, 50256, 50256, 50256],
         [ 7,      8,      9, 50256, 50256]])

```

```
tensor([[ 1, 2, 3, 4, 50256],
       [ 6, 50256, -100, -100, -100],
       [ 8, 9, 50256, -100, -100]])
```

The modified collate function works as expected, altering the target list by inserting the token ID `-100`. What is the logic behind this adjustment? Let's explore the underlying purpose of this modification.

For demonstration purposes, consider the following simple and self-contained example where each output logit corresponds to a potential token from the model's vocabulary. Here's how we might calculate the cross entropy loss (introduced in chapter 5) during training when the model predicts a sequence of tokens, which is similar to what we did when we pretrained the model and fine-tuned it for classification:

```
logits_1 = torch.tensor(
    [[-1.0, 1.0],           predictions for 1st token
     [-0.5, 1.5]]          predictions for 2nd token
)
targets_1 = torch.tensor([0, 1]) # Correct token indices to generate
loss_1 = torch.nn.functional.cross_entropy(logits_1, targets_1)
print(loss_1)
```

The loss value calculated by the previous code is `1.1269`:

```
tensor(1.1269)
```

As we would expect, adding an additional token ID affects the loss calculation:

```
logits_2 = torch.tensor(
    [[-1.0, 1.0],           New third token
     [-0.5, 1.5],           ID prediction
     [-0.5, 1.5]]
)
targets_2 = torch.tensor([0, 1, 1])
loss_2 = torch.nn.functional.cross_entropy(logits_2, targets_2)
print(loss_2)
```

After adding the third token, the loss value is `0.7936`.

So far, we have carried out some more or less obvious example calculations using the cross entropy loss function in PyTorch, the same loss function we used in the training functions for pretraining and fine-tuning for classification. Now let's get to the interesting part and see what happens if we replace the third target token ID with `-100`:

```
targets_3 = torch.tensor([0, 1, -100])
loss_3 = torch.nn.functional.cross_entropy(logits_2, targets_3)
print(loss_3)
print("loss_1 == loss_3:", loss_1 == loss_3)
```

The resulting output is

```
tensor(1.1269)
loss_1 == loss_3: tensor(True)
```

The resulting loss on these three training examples is identical to the loss we calculated from the two training examples earlier. In other words, the cross entropy loss function ignored the third entry in the `targets_3` vector, the token ID corresponding to -100. (Interested readers can try to replace the -100 value with another token ID that is not 0 or 1; it will result in an error.)

So what's so special about -100 that it's ignored by the cross entropy loss? The default setting of the cross entropy function in PyTorch is `cross_entropy(..., ignore_index=-100)`. This means that it ignores targets labeled with -100. We take advantage of this `ignore_index` to ignore the additional end-of-text (padding) tokens that we used to pad the training examples to have the same length in each batch. However, we want to keep one 50256 (end-of-text) token ID in the targets because it helps the LLM to learn to generate end-of-text tokens, which we can use as an indicator that a response is complete.

In addition to masking out padding tokens, it is also common to mask out the target token IDs that correspond to the instruction, as illustrated in figure 7.13. By masking out the LLM's target token IDs corresponding to the instruction, the cross entropy loss is only computed for the generated response target IDs. Thus, the model is trained to focus on generating accurate responses rather than memorizing instructions, which can help reduce overfitting.

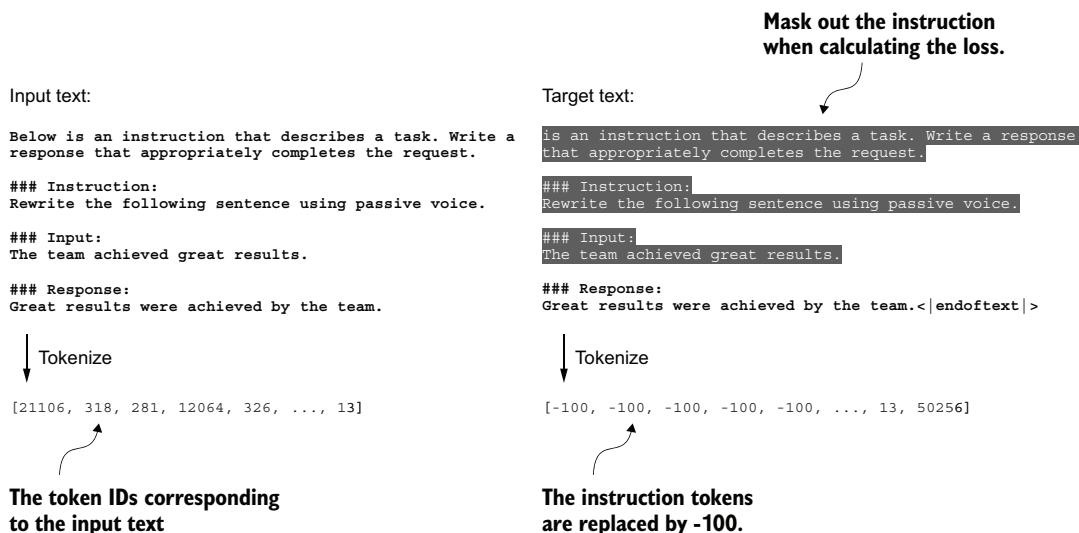


Figure 7.13 Left: The formatted input text we tokenize and then feed to the LLM during training. Right: The target text we prepare for the LLM where we can optionally mask out the instruction section, which means replacing the corresponding token IDs with the -100 `ignore_index` value.