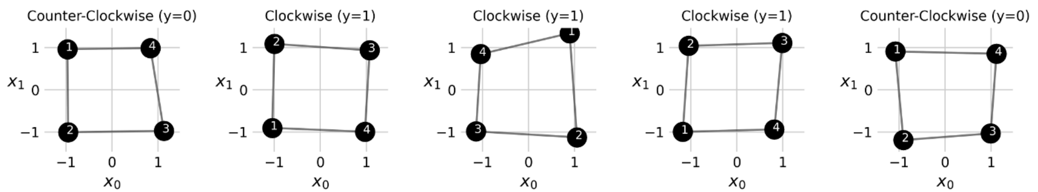


Figure 9.3 - Sequence dataset

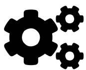
The corners show the **order** in which they were drawn. In the third square, the drawing **started at the top-right corner** (corresponding to the *blue C* corner) and followed a **clockwise direction** (corresponding to the *CDAB* sequence). The **source sequence** for that square would include corners ***C* and *D* (1 and 2)**, while the **target sequence** would include corners ***A* and *B* (3 and 4)**, in that order.

In order to **output a sequence** we need a more complex architecture; we need an…

Encoder-Decoder Architecture

The encoder-decoder is a combination of **two models**: the **encoder** and the **decoder**.

Encoder



The **encoder's goal** is to **generate a representation** of the **source sequence**; that is, to **encode it**.

"*Wait, we've done that already, right?*"

Absolutely! That's what the **recurrent layers** did: They generated a **final hidden state** that was a **representation of the input sequence**. Now you know *why* I insisted *so much* on this idea and repeated it *over and over again* in Chapter 8 :-)

At evaluation / prediction time we *only* have the source sequence, and, in our example, we use its last element as input for the decoder:

```
inputs = source_seq[:, -1:]
trg_masks = subsequent_mask(1)
out = decself(inputs, target_mask=trg_masks)
out
```

Output

```
tensor([[[0.4132, 0.3728]]], grad_fn=<AddBackward0>)
```

The mask is not actually masking anything in this case, and we get a prediction for the coordinates of $\mathbf{x}_2$ as expected. Previously, this prediction would have been used as the next input, but things are a *bit* different now.

The **self-attention decoder** expects the **full sequence** as **"query,"** so we **concatenate the prediction** to the **previous "query."**

```
inputs = torch.cat([inputs, out[:, -1:, :]], dim=-2)
inputs
```

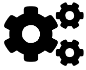
Output

```
tensor([[[-1.0000,  1.0000],
         [ 0.4132,  0.3728]]], grad_fn=<CatBackward>)
```

Now there are **two data points** for querying the decoder, so we adjust the mask accordingly:

$$
2^{\text{nd}}\ \text{Step}
\left\{
\begin{array}{c|cc}
 & \text{source} & \\
\text{target} & x_1 & x_2 \\
\hline
h_2 & \alpha_{21} & 0 \\
h_3 & \alpha_{31} & \alpha_{32}
\end{array}
\right.
$$

Equation 9.19 - Decoder's (masked) attention scores for the second target



The **mask** guarantees that the **predicted** x_2 (in the first step) **won't change** the **predicted** x_2 (in the second step), because predictions are made based on **past data points only**.

```
trg_masks = subsequent_mask(2)
out = decself(inputs, target_mask=trg_masks)
out
```

Output

```
tensor([[[0.4137, 0.3728],
         [0.4132, 0.3728]]], grad_fn=<AddBackward0>)
```

These are the **predicted coordinates** of both x_2 and x_3. They are very close to each other, but that's just because we're using an *untrained model* to illustrate the mechanics of using target masks for prediction. The **last prediction** is, once again, **concatenated** to the previous "query."

```
inputs = torch.cat([inputs, out[:, -1:, :]], dim=-2)
inputs
```

Output

```
tensor([[[-1.0000,  1.0000],
         [ 0.4132,  0.3728],
         [ 0.4132,  0.3728]]], grad_fn=<CatBackward>)
```

But, since we're *actually* done with the predictions (the desired target sequence has a length of two), we simply *exclude* the first data point in the query (the one coming from the source sequence), and are left with the **predicted target sequence**:

```
inputs[:, 1:]
```

On the right, the encoder uses a *norm-first wrapper*, and its output (the encoder's **states**) is given by:

$$\text{outputs}_{\text{norm-first}} = \underbrace{\text{inputs} + \text{att}(\text{norm}(\text{inputs}))}_{\text{Output of SubLayer}_0} + \text{ffn}(\text{norm}(\underbrace{\text{inputs} + \text{att}(\text{norm}(\text{inputs}))}_{\text{Output of SubLayer}_0}))$$

Equation 10.6 - Encoder's output: norm-first

The **norm-first wrapper** allows the **inputs to flow unimpeded** (the inputs aren't normalized) all the way to the top while adding the results of each "sub-layer" along the way (the *last* normalization of *norm-first* happens outside of the "sub-layers," so it's not included in the equation).

"Which one is best?"

There is no straight answer to this question. It actually reminds me of the discussions about placing the batch normalization layer *before* or *after* the activation function. Now, once again, there is no "right" and "wrong," and the order of the different components is not etched in stone.

In PyTorch, the encoder "layer" is implemented as `nn.TransformerEncoderLayer`, and its constructor method expects the following arguments (`d_model`, `nhead`, `dim_feedforward`, and `dropout`) and an optional `activation` function for the feed-forward network, similar to our own `EncoderLayer`. Its `forward()` method has **three main arguments**:

- `src`: the **source sequence**; that's the `query` argument in our class

IMPORTANT: PyTorch's **Transformer layers** use **sequence-first** shapes for their inputs (L, N, F) by default but, since v1.9, there's a `batch_first` argument you can set.

- `src_key_padding_mask`: the mask for **padded data points**; that's the `mask` argument in our class
- `src_mask`: This mask is used to **purposefully hide some of the inputs** in the source sequence—we're not doing that, so our class doesn't have a corresponding argument—a technique that can be used for training **language models** (more on that in Chapter 11).

In PyTorch, the decoder "layer" is implemented as `nn.TransformerDecoderLayer`, and its constructor method expects the following arguments (`d_model`, `nhead`, `dim_feedforward`, and `dropout`) and an optional `activation` function for the feed-forward network.

Its `forward()` method, though, has **six main arguments**. Three of them are equivalent to those arguments in our own `forward()` method:

- `tgt`: the **target sequence**; that's the `query` argument in our class (required)

IMPORTANT: PyTorch's **Transformer layers** use **sequence-first** shapes for their inputs (L, N, F) by default but, since v1.9, there's a `batch_first` argument you can set.

- `memory_key_padding_mask`: the mask for **padded data points** in the **source sequence**; that's the `source_mask` argument in our class (optional), and the same as the `src_key_padding_mask` of `nn.TransformerEncoderLayer`
- `tgt_mask`: the mask used to **avoid cheating**; that's the `target_mask` argument in our class (although quite important, this argument is still considered *optional*)

Then, there is the **other required argument**, which corresponds to the `states` argument of the `init_keys()` method in our own class:

- `memory`: the **encoded states** of the **source sequence** as returned by the **encoder**

The remaining two arguments *do not exist* in our own class:

- `memory_mask`: This mask is used to **purposefully hide some of the encoded states** used by the decoder.
- `tgt_key_padding_mask`: This mask is used for **padded data points** in the **target sequence**.

- First, and most important, PyTorch implements **norm-last "sub-layer" wrappers** by default, normalizing the output of each "sub-layer." You may switch it to norm-first using the `norm_first` argument introduced in version 1.10, though.

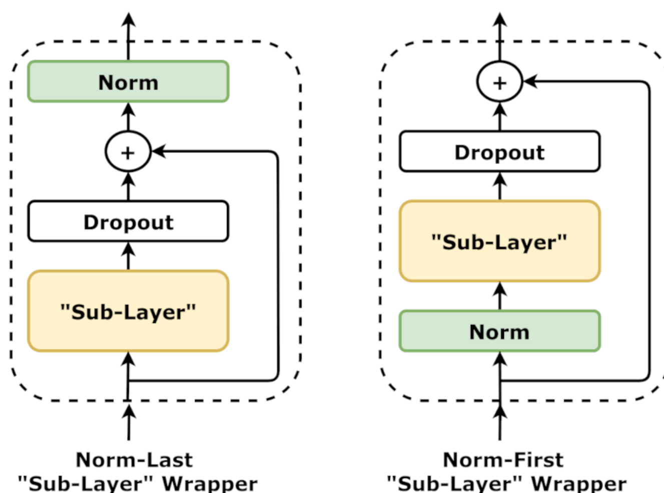


Figure 10.17 - "Sub-Layer"—norm-last vs norm-first

- It **does not** implement **positional encoding**, the **final linear layer**, or the **projection layer**, so we have to handle those ourselves.

Let's take a look at its constructor and `forward()` methods. The constructor expects *many arguments* because PyTorch's Transformer actually **builds both encoder and decoder by itself**:

- `d_model`: the **number of (projected) features**, that is, the **dimensionality of the model** (remember, this number will be *split* among the attention heads, so it must be a multiple of the number of heads; its default value is 512)
- `nhead`: the number of **attention heads** in each **attention mechanism** (default is eight, so each attention head gets 64 out of the 512 dimensions)
- `num_encoder_layers`: the number of "layers" in the encoder (the Transformer uses six layers by default)
- `num_decoder_layers`: the number of "layers" in the decoder (the Transformer uses six layers by default)
- `dim_feedforward`: the number of **units** in the **hidden layer** of the **feed-forward network** (default is 2048)

- `dropout`: the **probability** of dropping out inputs (default is 0.1)
- `activation`: the activation function to be used in the feed-forward network (ReLU by default)
- `batch_first`: It switches from **sequence-first** (L, N, F) to **batch-first** (N, L, F) shapes. This argument was introduced in PyTorch 1.9.
- `norm_first`: It switches from **norm-last** to **norm-first** "sub-layers." This argument was introduced in PyTorch 1.10.

It is also possible to use a **custom encoder or decoder** by setting the corresponding arguments: `custom_encoder` and `custom_decoder`.

The `forward()` method expects both sequences, **source** and **target**, and **all sorts of (optional) masks**.

There are **masks** for **padded data points**:

- `src_key_padding_mask`: the mask for **padded data points** in the **source sequence**
- `memory_key_padding_mask`: It's also a mask for **padded data points** in the **source sequence** and should be, in most cases, the *same* as `src_key_padding_mask`.
- `tgt_key_padding_mask`: This mask is used for **padded data points** in the **target sequence**.

And there are **masks** to **purposefully hide some of the inputs**:

- `src_mask`: It hides inputs in the **source sequence**, this can be used for training **language models** (more on that in Chapter 11).
- `tgt_mask`: That's the mask used to **avoid cheating** (although quite important, this argument is still considered *optional*).
  - The Transformer has a method named `generate_square_subsequent_mask()` that generates the appropriate mask given the size (length) of the sequence.
- `memory_mask`: It hides **encoded states** used by the decoder.

Also, notice that there is no **memory argument** anymore: The encoded states are

handled internally by the Transformer and fed directly to the decoder part.

In our own code, we'll be replacing the two former methods, `encode()` and `decode()`, with a single one, `encode_decode()`, that calls the **Transformer** itself and runs its output through the **last linear layer** to transform it into coordinates. The Transformer must be configured to use **batch-first** shapes to make it work.

```
def encode_decode(self, source, target,
                  source_mask=None, target_mask=None):
    # Projections
    src = self.preprocess(source)
    tgt = self.preprocess(target)

    out = self.transf(src, tgt,
                      src_key_padding_mask=source_mask,
                      tgt_mask=target_mask)

    # Linear
    out = self.linear(out) # N, L, F
    return out
```

By the way, we're keeping the masks to a minimum for the sake of simplicity: Only `src_key_padding_mask` and `tgt_mask` are used.

Moreover, we're implementing a `preprocess()` method that takes an **input sequence** and

- **projects** the original features into the model dimensionality;
- adds **positional encoding** and
- **(layer) normalizes** the result (remember that PyTorch's norm-last default *does not normalize the inputs*, so we have to do it ourselves).

The full code looks like this:

Transformer

```
 1 class TransformerModel(nn.Module):
 2     def __init__(self, transformer,
 3                  input_len, target_len, n_features):
 4         super().__init__()
 5         self.transf = transformer
 6         self.input_len = input_len
 7         self.target_len = target_len
 8         self.trg_masks = \
 9             self.transf.generate_square_subsequent_mask(
10                 self.target_len
11             )
12         self.n_features = n_features
13         self.proj = nn.Linear(n_features, self.transf.d_model) ①
14         self.linear = nn.Linear(self.transf.d_model,           ②
15                                 n_features)
16
17         max_len = max(self.input_len, self.target_len)
18         self.pe = PositionalEncoding(max_len,
19                                      self.transf.d_model)      ③
20         self.norm = nn.LayerNorm(self.transf.d_model)          ③
21
22     def preprocess(self, seq):
23         seq_proj = self.proj(seq)                              ①
24         seq_enc = self.pe(seq_proj)                            ③
25         return self.norm(seq_enc)                              ③
26
27     def encode_decode(self, source, target,
28                       source_mask=None, target_mask=None):
29         # Projections
30         src = self.preprocess(source)                          ③
31         tgt = self.preprocess(target)                          ③
32
33         out = self.transf(src, tgt,
34                           src_key_padding_mask=source_mask,
35                           tgt_mask=target_mask)
36
37         # Linear
38         out = self.linear(out) # N, L, F                       ②
```

```
39          return out
40
41      def predict(self, source_seq, source_mask=None):
42          inputs = source_seq[:, -1:]
43          for i in range(self.target_len):
44              out = self.encode_decode(
45                  source_seq, inputs,
46                  source_mask=source_mask,
47                  target_mask=self.trg_masks[:i+1, :i+1]
48              )
49              out = torch.cat([inputs, out[:, -1:, :]], dim=-2)
50              inputs = out.detach()
51          outputs = out[:, 1:, :]
52          return outputs
53
54      def forward(self, X, source_mask=None):
55          self.trg_masks = self.trg_masks.type_as(X)
56          source_seq = X[:, :self.input_len, :]
57
58          if self.training:
59              shifted_target_seq = X[:, self.input_len-1:-1, :]
60              outputs = self.encode_decode(
61                  source_seq, shifted_target_seq,
62                  source_mask=source_mask,
63                  target_mask=self.trg_masks
64              )
65          else:
66              outputs = self.predict(source_seq, source_mask)
67
68          return outputs
```

① Projecting features to model dimensionality

② Final linear transformation from model to feature space

③ Adding positional encoding and normalizing inputs

Its constructor takes an instance of the `nn.Transformer` class followed by the typical sequence lengths *and* the number of features (so it can map the predicted sequence back to our feature space; that is, to coordinates). Both `predict()` and `forward()` methods are roughly the same, but they call the `encode_decode()` method now.

Model Configuration & Training

Let's train PyTorch's **Transformer**! We start by creating an instance of it to use as an argument of our `TransformerModel` class, followed by the same initialization scheme as before, and the typical training procedure:

Model Configuration

```
1 torch.manual_seed(42)
2 transformer = nn.Transformer(d_model=6,
3                              nhead=3,
4                              num_encoder_layers=1,
5                              num_decoder_layers=1,
6                              dim_feedforward=20,
7                              dropout=0.1,
8                              batch_first=True)
9 model_transformer = TransformerModel(transformer, input_len=2,
10                                     target_len=2, n_features=2)
11 loss = nn.MSELoss()
12 optimizer = torch.optim.Adam(model_transformer.parameters(),
13                              lr=0.01)
```

Weight Initialization

```
1 for p in model_transformer.parameters():
2     if p.dim() > 1:
3         nn.init.xavier_uniform_(p)
```

Model Training

```
1 sbs_seq_transformer = StepByStep(
2     model_transformer, loss, optimizer
3 )
4 sbs_seq_transformer.set_loaders(train_loader, test_loader)
5 sbs_seq_transformer.train(50)
```

```
fig = sbs_seq_transformer.plot_losses()
```

```
class ViT(nn.Module):
    def __init__(self, encoder, img_size,
                 in_channels, patch_size, n_outputs):
        super().__init__()
        self.d_model = encoder.d_model
        self.n_outputs = n_outputs
        self.encoder = encoder
        self.mlp = nn.Linear(encoder.d_model, n_outputs)

        self.embed = PatchEmbed(img_size, patch_size,
                                in_channels, encoder.d_model)
        self.cls_token = nn.Parameter(
            torch.zeros(1, 1, encoder.d_model)
        )

    def preprocess(self, X):
        # Patch embeddings
        # N, C, H, W -> N, L, D
        src = self.embed(X)
        # Special classifier token
        # 1, 1, D -> N, 1, D
        cls_tokens = self.cls_token.expand(X.size(0), -1, -1)
        # Concatenates CLS tokens -> N, 1 + L, D
        src = torch.cat((cls_tokens, src), dim=1)
        return src

    def encode(self, source):
        # Encoder generates "hidden states"
        states = self.encoder(source)
        # Gets state from first token: CLS
        cls_state = states[:, 0]  # N, 1, D
        return cls_state

    def forward(self, X):
        src = self.preprocess(X)
        # Featurizer
        cls_state = self.encode(src)
        # Classifier
        out = self.mlp(cls_state) # N, 1, outputs
        return out
```

Additional Setup

This is a *special chapter* when it comes to its setup: We won't be using *only* PyTorch but rather a handful of other packages as well, including the *de facto* standard for NLP tasks—HuggingFace.

Before proceeding, make sure you have all of them installed by running the commands below:

```
!pip install gensim==4.3.3
!pip install flair==0.13.1
!pip install torchvision==0.18.1
# HuggingFace
!pip install transformers==4.42.4
!pip install datasets==2.18.0
```

Some packages, like `flair`, may have *strict dependencies* and eventually require the **downgrading** of some other packages in your environment, even PyTorch itself.

Even though the packages above are pinned to specific versions, you may use newer ones if you want. Regardless of which versions you're using, though, reproducibility is not guaranteed and you should expect small differences in the produced outputs.

Imports

For the sake of organization, all libraries needed throughout the code used in any given chapter are imported at its very beginning. For this chapter, we'll need the following imports:

```
import os
import json
import errno
import requests
import numpy as np
from copy import deepcopy
from operator import itemgetter
```

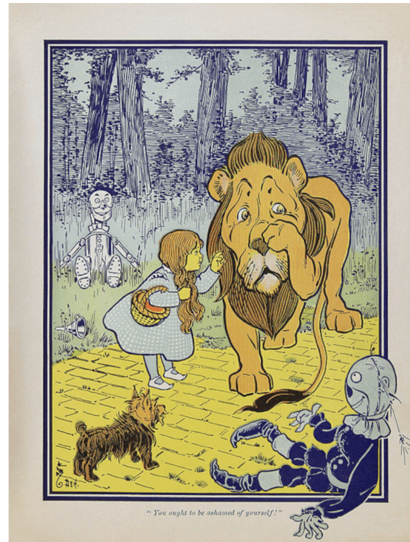


Figure 11.1 - Left: "Alice and the Baby Pig" illustration by John Tenniel, from Alice's Adventures in Wonderland (1865). Right: "Dorothy meets the Cowardly Lion" illustration by W. W. Denslow, from The Wonderful Wizard of Oz (1900).

The direct links to both texts are alice28-1476.txt[66] (*https://tinyurl.com/yfcjzy68*, we're naming it `ALICE_URL`) and wizoz10-1740.txt[67] (*https://tinyurl.com/3rtywhz6*, we're naming it `WIZARD_URL`). You can download both of them to a local folder using the helper function `download_text()` (included in `data_generation.nlp`):

Data Loading

```
1 localfolder = 'texts'
2 download_text(ALICE_URL, localfolder)
3 download_text(WIZARD_URL, localfolder)
```

If you open these files in a text editor, you'll see that there is a *lot* of information at the beginning (and some at the end) that has been added to the original text of the books for legal reasons. We need to remove these additions to the original texts:

Downloading Books

```
1 fname1 = os.path.join(localfolder, 'alice28-1476.txt')
2 with open(fname1, 'r') as f:
3     alice = ''.join(f.readlines()[104:3704])
4 fname2 = os.path.join(localfolder, 'wizoz10-1740.txt')
5 with open(fname2, 'r') as f:
6     wizard = ''.join(f.readlines()[310:5100])
```

Global Vectors (GloVe)

The Global Vectors model was proposed by Pennington, J. et al. in their 2014 paper "GloVe: Global Vectors for Word Representation."[87] It combines the **skip-gram** model with **co-occurrence statistics** at the **global level** (hence the name). We're not diving into its inner workings here, but if you're interested in knowing more about it, check its official website: https://nlp.stanford.edu/projects/glove/.

The pre-trained GloVe embeddings come in many sizes and shapes: Dimensions vary between 25 and 300, and vocabularies vary between 400,000 and 2,200,000 words. Let's use Gensim's `downloader` to retrieve the smallest one: `glove-wiki-gigaword-50`. It was trained on Wikipedia 2014 and Gigawords 5, it contains 400,000 words in its vocabulary, and its embeddings have 50 dimensions.

Downloading Pre-trained Word Embeddings

```
1 from gensim import downloader
2 glove = downloader.load('glove-wiki-gigaword-50')
3 len(glove.key_to_index)
```

Output

```
400000
```

Let's check the embeddings for "***alice***" (the vocabulary is uncased):

```
glove['alice']
```

Output

```
array([ 0.16386,  0.57795, -0.59197, -0.32446,  0.29762,  0.85151,
       -0.76695, -0.20733,  0.21491, -0.51587, -0.17517,  0.94459,
        0.12705, -0.33031,  0.75951,  0.44449,  0.16553, -0.19235,
        0.06553, -0.12394,  0.61446,  0.89784,  0.17413,  0.41149,
        1.191  , -0.39461, -0.459  ,  0.02216, -0.50843, -0.44464,
        0.68721, -0.7167 ,  0.20835, -0.23437,  0.02604, -0.47993,
        0.31873, -0.29135,  0.50273, -0.55144, -0.06669,  0.43873,
       -0.24293, -1.0247 ,  0.02937,  0.06849,  0.25451, -1.9663 ,
        0.26673,  0.88486], dtype=float32)
```

Only 82 out of 50,802 words in the text corpora cannot be matched to the vocabulary of the word embeddings. That's an impressive 99.84% coverage!

The helper function below can be used to compute the **vocabulary coverage** given a **Gensim's `Dictionary`** and **pre-trained embeddings**:

Method for Vocabulary Coverage

```
1 def vocab_coverage(gensim_dict, pretrained_wv,
2                    special_tokens=('[PAD]', '[UNK]')):
3     vocab = list(gensim_dict.token2id.keys())
4     unknown_words = sorted(
5         list(set(vocab).difference(
6              set(pretrained_wv.key_to_index)))
7     )
8     unknown_ids = [gensim_dict.token2id[w]
9                    for w in unknown_words
10                   if w not in special_tokens]
11    unknown_count = np.sum([gensim_dict.cfs[idx]
12                            for idx in unknown_ids])
13    cov = 1 - unknown_count / gensim_dict.num_pos
14    return cov
```

```
vocab_coverage(dictionary, glove)
```

Output

```
0.9983858903192788
```

Tokenizer

Once we're happy with the **vocabulary coverage** of our pre-trained embeddings, we can **save the vocabulary of the embeddings to disk** as a plain-text file, so we can use it with the HF's tokenizer:

Method to Save a Vocabulary from Pre-trained Embeddings

```
1 def make_vocab_from_wv(wv, folder=None, special_tokens=None):
2     if folder is not None:
3         if not os.path.exists(folder):
4             os.mkdir(folder)
5
6     words = wv.index_to_key
7     if special_tokens is not None:
8         to_add = []
9         for special_token in special_tokens:
10            if special_token not in words:
11                to_add.append(special_token)
12        words = to_add + words
13
14    with open(os.path.join(folder, 'vocab.txt'), 'w') as f:
15        for word in words:
16            f.write(f'{word}\n')
```

Saving GloVe's Vocabulary to a File

```
1 make_vocab_from_wv(glove,
2                    'glove_vocab/',
3                    special_tokens=['[PAD]', '[UNK]'])
```

We'll be using the `BertTokenizer` class once again to create a tokenizer based on GloVe's vocabulary:

Creating a Tokenizer using GloVe

```
1 glove_tokenizer = BertTokenizer('glove_vocab/vocab.txt')
```

One more time: The (pre-trained) tokenizer you'll use for real with a (pre-trained) BERT model **does not need a vocabulary**.

Now we can use its `encode()` method to get the indices for the tokens in a sentence:

```
glove_tokenizer.encode('alice followed the white rabbit',
                       add_special_tokens=False)
```

Output

```
[7101, 930, 2, 300, 12427]
```

These are the **indices** we'll use to **retrieve the corresponding word embeddings**. There is **one small detail** we need to take care of first, though…

Special Tokens' Embeddings

Our vocabulary has 400,002 tokens now, but the original pre-trained word embeddings has only 400,000 entries:

```
len(glove_tokenizer.vocab), len(glove.vectors)
```

Output

```
(400002, 400000)
```

The difference is due to the **two special tokens**, `[PAD]` and `[UNK]`, that were **prepended to the vocabulary** when we saved it to disk. Therefore, we need to **prepend their corresponding embeddings** too.

"How would I know the embeddings for these tokens?"

That's actually easy; these embeddings are just 50-dimensional vectors of **zeros**, and we concatenate them to the GloVe's pre-trained embeddings, making sure that the special embeddings come first:

Adding Embeddings for the Special Tokens

```
1 special_embeddings = np.zeros((2, glove.vector_size))
2 extended_embeddings = np.concatenate(
3     [special_embeddings, glove.vectors], axis=0
4 )
5 extended_embeddings.shape
```

Output

```
(400002, 50)
```

Now, if we encode "*alice*" to get its corresponding index, and use that index to retrieve the corresponding values from our *extended embeddings*, they should match the original GloVe embeddings:

```
alice_idx = glove_tokenizer.encode(
    'alice', add_special_tokens=False
)
np.all(extended_embeddings[alice_idx] == glove['alice'])
```

Output

```
True
```

OK, it looks like we're set! Let's put these embeddings to good use and *finally* train a model in PyTorch!

I want to introduce you to…

ELMo

Born in 2018, ELMo is able to understand that words may have different meanings in different contexts. If you feed it a sentence, it will give you back embeddings for each of the words while taking the full context into account.

Embeddings from **L**anguage **Mo**dels (ELMo, for short) was introduced by Peters, M. et al. in their paper "Deep contextualized word representations"[88] (2018). The model is a **two-layer bidirectional LSTM encoder** using **4,096 dimensions in its cell states** and was trained on a **really large corpus** containing **5.5 billion words**. Moreover, ELMo's representations are **character-based**, so it can easily handle unknown (out-of-vocabulary) words.

You can find more details about its implementation, as well as its **pre-trained weights**, at AllenNLP's ELMo[89] site. You can also check the "ELMo"[90] section of Lilian Weng's great post, "Generalized Language Models."[91]

"*Cool, are we loading a pre-trained model then?*"

Well, we *could*, but ELMo embeddings can be conveniently retrieved using yet another library: flair.[92] `flair` is an NLP framework built on top of PyTorch that offers a **text embedding library** that provides **word embeddings** and **document embeddings** for popular Muppets, oops, models like **ELMo** and **BERT**, as well as classical word embeddings like GloVe.

Unfortunately, as of December 16, 2022, AllenNLP's repository is archived. It doesn't make sense to use AllenNLP for retrieving ELMo embeddings anymore because it would require pinning `flair` and PyTorch itself to older versions. The code in this section was originally written in 2020, and it will be kept in this revision because of the historical value of ELMo.

For a complete list of available tasks, please check HuggingFace's `pipeline` [112] documentation.

Let's run the first sentence of our training set through the sentiment analysis pipeline:

```
sentence = train_dataset[0]['sentence']
print(sentence)
print(sentiment(sentence))
```

Output

```
And, so far as they knew, they were quite right.
[{'label': 'POSITIVE', 'score': 0.9998356699943542}]
```

Positive, indeed!

If you're curious about **which model** is being used under the hood, you can check the `SUPPORTED_TASKS` dictionary. For sentiment analysis, it uses the `distilbert-base-uncased-finetuned-sst-2-english` model:

```
from transformers.pipelines import SUPPORTED_TASKS
SUPPORTED_TASKS['text-classification']
```

Output

```
{'impl': transformers.pipelines.text_classification
.TextClassificationPipeline, ...
 'pt': (transformers.models.auto.modeling_auto
.AutoModelForSequenceClassification,),
 'default': {'model': {'pt': ('distilbert/distilbert-base-uncased-
finetuned-sst-2-english',
    'af0f99b'), ...},
 'type': 'text'}
```

"*What about **text generation**?*"

```
SUPPORTED_TASKS['text-generation']
```

Output

```
{'impl': transformers.pipelines.text_generation
.TextGenerationPipeline,
 'pt': (transformers.models.auto.modeling_auto
.AutoModelForCausalLM,),
 'default': {'model': {'pt': ('openai-community/gpt2', '6c0e608'),
...}},
 'type': 'text'}
```

That's the **famous GPT-2** model, which we'll discuss briefly in the next, and last, section of this chapter.

GPT-2

The **G**enerative **P**retrained **T**ransformer **2**, introduced by Radford, A. et al. in their paper "Language Models are Unsupervised Multitask Learners"[113] (2018), made headlines with its impressive ability to **generate text** of high quality in a variety of contexts. Just like BERT, it is a **language model**; that is, it is trained to **fill in the blanks** in sentences. But, while BERT was trained to fill in the blanks in the middle of sentences (thus correcting corrupted inputs), **GPT-2** was trained to **fill in blanks at the end of sentences**, effectively **predicting the next word in a given sentence**.

Predicting the **next element in a sequence** is exactly what a **Transformer decoder** does, so it should be no surprise that **GPT-2 is actually a Transformer decoder**.

It was trained on more than 40 GB of Internet text spread over 8 million web pages. Its largest version has **48 "layers"** (the original Transformer had only six), **twelve attention heads**, and **1,600 hidden dimensions**, totaling **1.5 billion parameters**, and it was released in November 2019.[114]

"Don't train this at home!"

On the other end of the scale, the smallest version has *only* twelve "layers," twelve attention heads, and 768 hidden dimensions, totaling 117 million parameters (the smallest GPT-2 is still a bit larger than the original BERT!). This is the version automatically loaded in the `TextGenerationPipeline`.