Published in Towards Data Science

Ruben Winastwan Follow

May 3, 2022 · 11 min read · ✦ · Listen

Named Entity Recognition with BERT in PyTorch

How to leverage a pre-trained BERT model for custom data to predict the entity of each word in a text

Photo by Aaron Burden on Unsplash

When it comes to dealing with NLP problems, BERT oftentimes comes up as a machine learning model that we can count on in terms of its performance. The fact that it's been pre-trained on more than 2,500M words and its bidirectional nature to learn information from a sequence of words makes it a powerful model to use.

I wrote about how we can leverage BERT for text classification before, and in this article, we're going to focus more on how to use BERT for named entity recognition (NER) tasks.

What is NER?

NER is a task in NLP to identify and extract meaningful information (or we can call it entities) in a sentence or text. An entity can be a single word or even a group of words that refer to the same category.

As an example, let's say we the following sentence and we want to extract information about a person's name from this sentence.

The first step of a NER task is to detect an entity. This can be a word or a group of words that refer to the same category. As an example:

- '*Bond*' ➡ an entity that consists of a single word
- '*James Bond*' ➡ an entity that consists of two words, but they are referring to the same category.

To make sure that our BERT model knows that an entity can be a single word or a group of words, then we need to provide information about the beginning and the ending of an entity on our training data via the so-called Inside-Outside-Beginning (IOB) tagging. We will see more about this on our dataset later in this article.

After detecting an entity, the next step in a NER task is to categorize the detected entity. The categories of an entity can be anything depending on our use case. Below is an example of categories of entities:

- **Person**: Bond, James Bond, Sam, Anna, Frank, Leonardo DiCaprio
- **Location**: New York, Vienna, Munich, London
- **Organization**: Google, Apple, Stanford University, Deutsche Bank
- **Location**: Central Park, Brandenburger Tor, Times Square

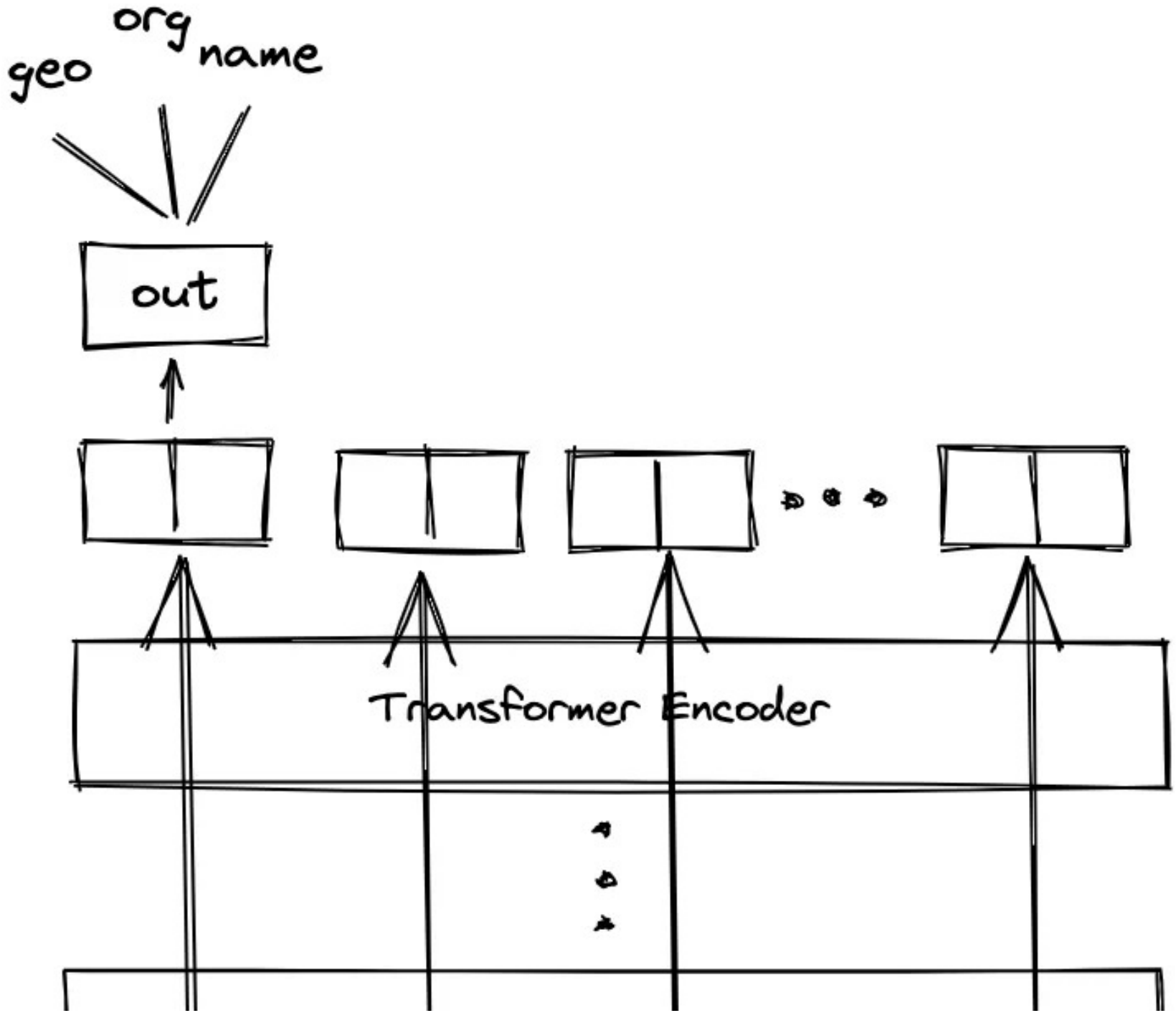
These entities are basically the label of our data during the training process of our BERT model, which we will look at in detail later in the following section.

BERT for NER

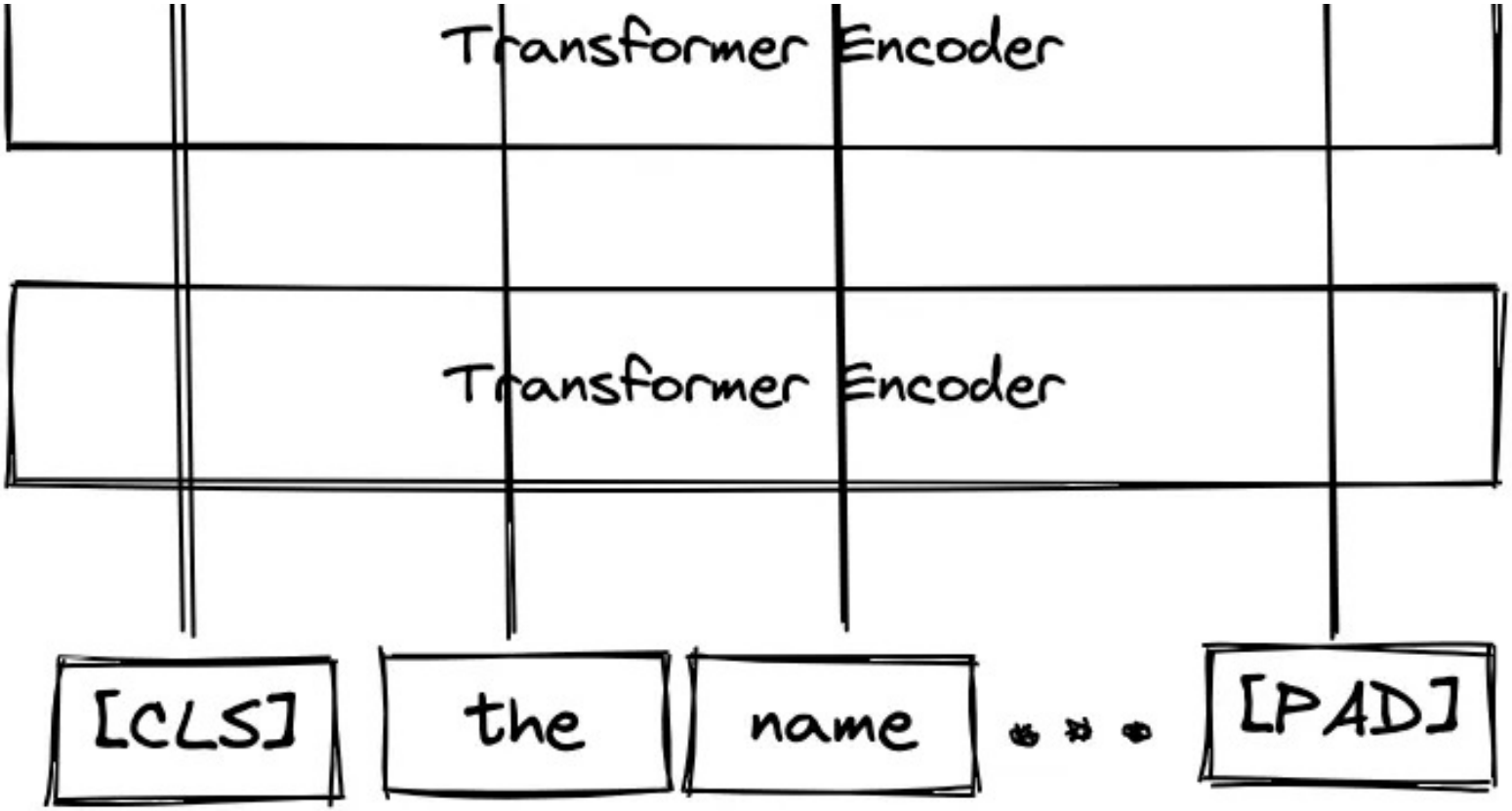
As previously mentioned, BERT is a transformers-based machine learning model that will come in pretty handy if we want to solve NLP-related tasks.

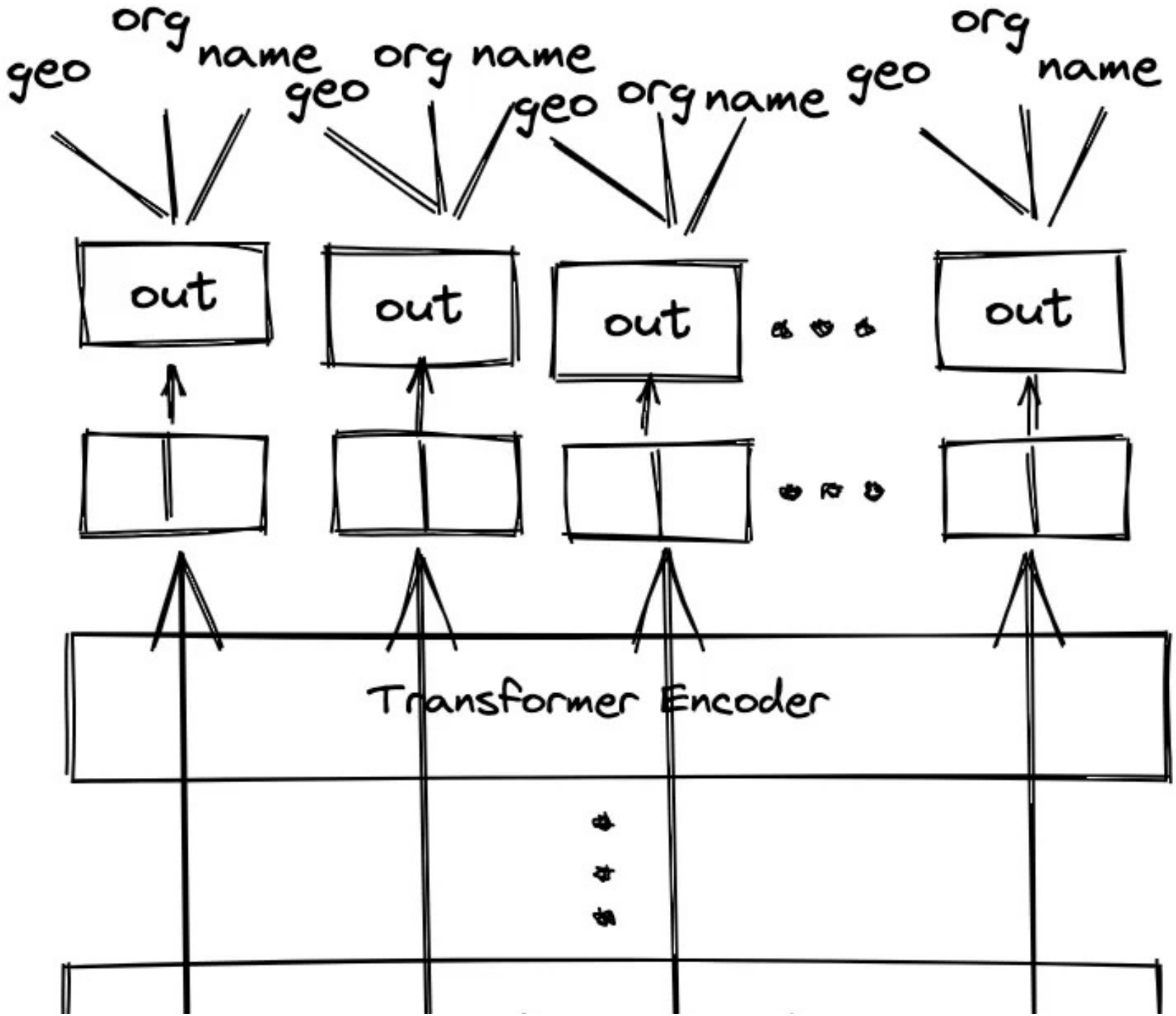
If you're not yet familiar with BERT, I recommend you to read my previous article about text classification with BERT before reading this article. There you'll find information about what BERT actually is, what kind of input data the model expects, and the output that you'll get from the model.

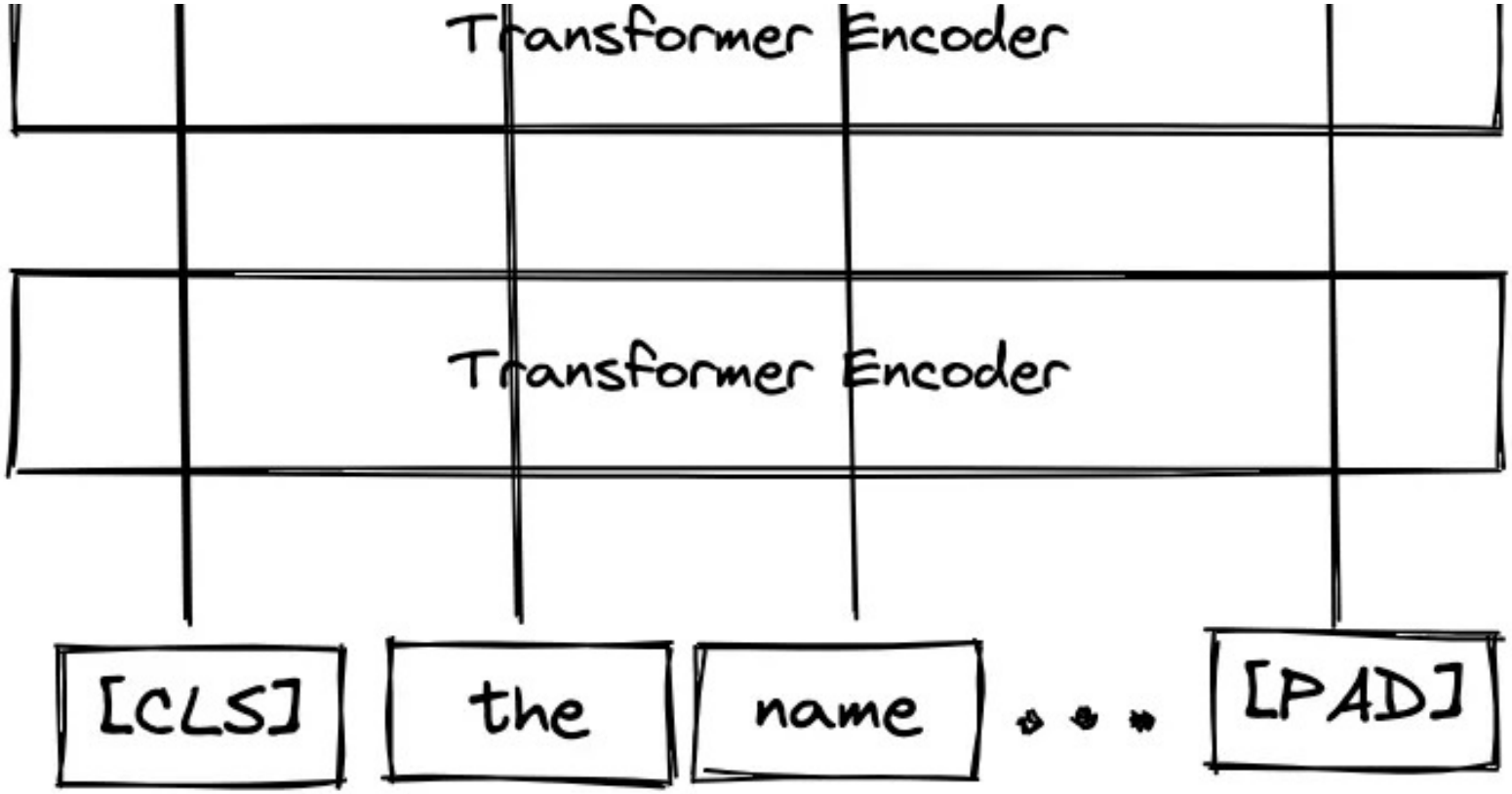
What differentiates between BERT for text classification and the NER problem is how we set the output of the model. For a text classification problem, we only use the embedding vector output from the special **[CLS]** token, as you can see in the visualization below:


geo
org
name
out
Transformer Encoder

Mea ›m
all c




geo
org
name
geo
org
name
geo
org
name
geo
org
name
out
out
out
out
Transformer Encoder



By u n
leve en.
Nov

Abo

The
spec

NER Data

Named Entity Recognition Data

www.kaggle.com

This dataset is distributed under Open Database v1.0 license, so we are free to share and use this dataset for our own purpose. Now let's take a look at what the dataset looks like.

In [1]:

```
import pandas as pd
```

In [2]:

```
df = pd.read_csv('ner.csv')
df.head()
```

Out[2]:

	text	labels
0	Thousands of demonstrators have marched throug...	O O O O O O B-geo O O O O O B-geo O O O O O B-...
1	Iranian officials say they expect to get acces...	B-gpe O O O O O O O O O O O O O B-tim O O O ...
2	Helicopter gunships Saturday pounded militant ...	O O B-tim O O O O O B-geo O O O O O B-org O O ...
3	They left after a tense hour-long standoff wit...	O O O O O O O O O O O
4	U.N. relief coordinator Jan Egeland said Sunda...	B-geo O O B-per I-per O B-tim O B-geo O B-gpe ...

In [ ]:

As we can see above, we have a dataframe which consists of the text and the label. The label corresponds to entity category of each word in a text.

In total, there are 9 entity categories, which are:

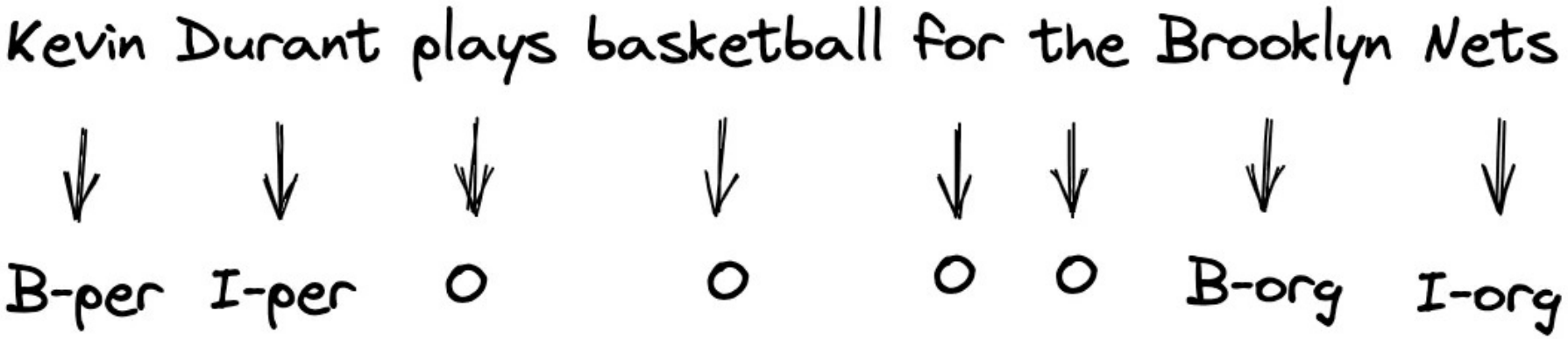
- `geo` for geographical entity
- `org` for organization entity
- `per` for person entity
- `gpe` for geopolitical entity
- `tim` for time indicator entity
- `art` for artifact entity
- `eve` for event entity
- `nat` for natural phenomenon entity
- `O` is assigned if a word doesn't belong to any entity.

Let's take a look at the unique labels available on our dataset:

```python
# Split labels based on whitespace and turn them into a list
labels = [i.split() for i in df['labels'].values.tolist()]

# Check how many labels are there in the dataset
unique_labels = set()

for lb in labels:
  [unique_labels.add(i) for i in lb if i not in unique_labels]

print(unique_labels)
>>> {'B-tim', 'B-art', 'I-art', 'O', 'I-gpe', 'I-per', 'I-nat', 'I-geo', 'B-eve', 'B-org', 'B-gpe', 'I-eve', 'B-per

# Map each label into its id representation and vice versa
labels_to_ids = {k: v for v, k in enumerate(sorted(unique_labels))}
ids_to_labels = {v: k for v, k in enumerate(sorted(unique_labels))}
print(labels_to_ids)
>>> {'B-art': 0, 'B-eve': 1, 'B-geo': 2, 'B-gpe': 3, 'B-nat': 4, 'B-org': 5, 'B-per': 6, 'B-tim': 7, 'I-art': 8, 'I
```

ner_1.py hosted with ❤ by GitHub view raw

As you might notice, each entity category is preceeded with the letter `I` or `B`. This corresponds to what previously mentioned as IOB tagging. `I` means ***Intermediate*** and `B` means ***Beginning***. Let's take a look at the following sentence to understand the concept of IOB tagging a little bit more.



- 'Kevin' has `B-pers` label since it's the beginning of a person entity
- 'Durant' has `I-pers` label because it's the continuation of a person entity
- 'Brooklyn' has `B-org` since it's the beginning of an organization entity
- 'Nets' has `I-org` label since it's the continuation of an organization entity
- Other words are assigned `O` label as they don't belong to any entity

Data Preprocessing

Before we are able to use a BERT model to classify the entity of a token, of course, we need to do data preprocessing first, which includes two parts: tokenization and adjusting the label to match the tokenization. Let's start with tokenization first.

Tokenization

Tokenization can be easily implemented with BERT, as we can use `BertTokenizerFast` class from a pretrained BERT base model with HuggingFace.

To give you an example how BERT tokenizer works, let's take a look at one of the texts from our dataset:

```
# Let's take a look at how can we preprocess the text - Take first example
text = df['text'].values.tolist()
example = text[36]

print(example)
>>> 'Prime Minister Geir Haarde has refused to resign or call for early elections.'
```

ner_2.py hosted with ❤ by GitHub view raw

Tokenizing the text above with `BertTokenizerFast` is very straightforward:

```python
from transformers import BertTokenizerFast

tokenizer = BertTokenizerFast.from_pretrained('bert-base-cased')
text_tokenized = tokenizer(example, padding='max_length', max_length=512, truncation=True, return_tensors="pt")
```

ner_3.py hosted with ❤ by GitHub view raw

We provide several arguments when calling `tokenizer` method from `BertTokenizerFast` class above:

- `padding` : to pad the sequence with a special **[PAD]** token to the maximum length that we specify. The maximum length of a sequence for a BERT model is 512.
- `max_length` : maximum length of a sequence.
- `truncation` : this is a Boolean value. If we set the value to True, then tokens that exceed the maximum length will not be used.
- `return_tensors` : the tensor type that is returned, depending on machine learning frameworks that we use. Since we're using PyTorch, then we use `pt` .

And below is the output of the tokenization process:

```
print(text_tokenized)

>>> {'input_ids': tensor([[  101,  3460,  2110,   144,  6851,  1197, 11679,  2881,  1162,  1144,
          3347,  1106, 13133,  1137,  1840,  1111,  1346,  3212,   119,   102,
             0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
             0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
             0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
             0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
                                  ..........................
             0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
             0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
             0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
             0,     0]]),
     'token_type_ids': tensor([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                 ..................................
         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0]]),
     'attention_mask': tensor([[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                 ..................................
         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
30          0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
31          0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
32          0, 0, 0, 0, 0, 0, 0, 0]])}
```

ner_4.py hosted with ❤ by GitHub view raw

the special **[CLS]** token, the id 102 is reserved for the special **[SEP]** token, and the id 0 is reserved for **[PAD]** token.

- `token_type_ids` : To identify the sequence in which a token belongs to. Since we only have one sequence per text, then all the values of `token_type_ids` will be 0.
- `attention_mask` : To identify whether a token is a real token or padding. The value would be 1 if it's a real token, and 0 if it's a **[PAD]** token.

From the `input_ids` above, we can decode the ids back into the original sequence with `decode` method as follows:

```
1  print(tokenizer.decode(text_tokenized.input_ids[0]))
2
3  >>> '[CLS] Prime Minister Geir Haarde has refused to resign or call for early elections. [SEP] [PAD] [PAD] [PAD] [PA
```

ner_5.py hosted with ❤ by GitHub view raw

We got our original sequence back after implementing `decode` method with the addition of special tokens from BERT such as **[CLS]** token at the beginning of the sequence, **[SEP]** token at the end of the sequence, and a bunch of **[PAD]** tokens to fulfill the required maximum length of 512.

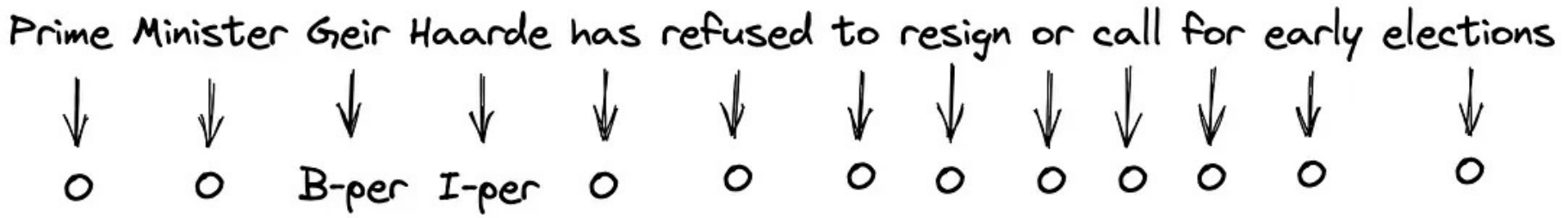
After this tokenization process, we need to proceed to the next step, which is adjusting the label of each token.

Adjusting Label After Tokenization

This is a very important step that we need to do after the tokenization process. This is because the length of the sequence is no longer matching the length of the original label after the tokenization process.

The BERT tokenizer uses the so-called word-piece tokenizer under the hood, which is a sub-word tokenizer. This means that BERT tokenizer will likely to split one word into one or more meaningful sub-words.

As an example, let's say we have the following sequence:



The sequence above has in total 13 tokens and thus, it also has 13 labels. However, after BERT tokenization, we get the following result:

```
1  print(tokenizer.convert_ids_to_tokens(text_tokenized["input_ids"][0]))
2
3  >>> ['[CLS]', 'Prime', 'Minister', 'G', '##ei', '##r', 'Ha', '##ard', '##e', 'has', 'refused', 'to', 'resign', 'or',
```

ner_6.py hosted with ❤ by GitHub view raw

There are two problems that we need to address after tokenization process:

- The addition of special tokens from BERT such as **[CLS]**, **[SEP]**, and **[PAD]**
- The fact that some tokens are splitted into sub-words.

As sub-word tokenization, word-piece tokenization splits uncommon words into their sub-words, such as '*Geir*' and '*Haarde*' in the example above. This sub-word tokenization helps the BERT model to learn the semantic meaning of related words.

The consequence of this word piece tokenization and the addition of special tokens from BERT is that the sequence length after tokenization is no longer matching the length of the initial label.

From the example above, now there are in total 512 tokens in the sequence after tokenization, while the length of the label is still the same as before. Also, the first token in a sequence is no longer the word '*Prime*', but the newly added **[CLS]** token, so we need to shift our label as well.

To solve this problem, we need to adjust the label such that it has the same length as the sequence after tokenization. To do this, we can utilize the `word_ids` method from the tokenization result as follows:

```
word_ids = text_tokenized.word_ids()
print(tokenizer.convert_ids_to_tokens(text_tokenized["input_ids"][0]))
print(word_ids)

>>> ['[CLS]', 'Prime', 'Minister', 'G', '##ei', '##r', 'Ha', '##ard', '##e', 'has', 'refused', 'to', 'resign', 'or',
>>> [None, 0, 1, 2, 2, 2, 3, 3, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, None, None, None, None, None, None, None, None,
```

ner_7.py hosted with ❤ by GitHub view raw

As you can see from the code snippet above, each splitted token shares the same `word_ids` , where special tokens from BERT such as **[CLS]**, **[SEP]**, and **[PAD]** all do not have specific `word_ids` .

These `word_ids` will be very useful to adjust the length of the label by applying either of these two methods:

1. We only provide a label to the first sub-word of each splitted token. The continuation of the sub-word then will simply have '-100' as a label. All tokens that don't have `word_ids` will also be labeled with '-100'.
2. We provide the same label among all of the sub-words that belong to the same token. All tokens that don't have `word_ids` will be labeled with '-100'.

The function in the code snippet below will do exactly the step defined above.

```python
def align_label_example(tokenized_input, labels):

        word_ids = tokenized_input.word_ids()

        previous_word_idx = None
        label_ids = []

        for word_idx in word_ids:

            if word_idx is None:
                label_ids.append(-100)

            elif word_idx != previous_word_idx:
                try:
                  label_ids.append(labels_to_ids[labels[word_idx]])
                except:
                  label_ids.append(-100)

            else:
                label_ids.append(labels_to_ids[labels[word_idx]] if label_all_tokens else -100)
            previous_word_idx = word_idx


        return label_ids
```

If you want to apply the first method, set `label_all_tokens` to False. If you want to apply the second method, set `label_all_tokens` to True, as you can see in the following code snippet:

```
label = labels[36]

#If we set label_all_tokens to True.....
label_all_tokens = True

new_label = align_label_example(text_tokenized, label)
print(new_label)
print(tokenizer.convert_ids_to_tokens(text_tokenized["input_ids"][0]))

>>> [-100, 16, 16, 6, 6, 6, 14, 14, 14, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, -100, -100, -100, -100, -100, -100,
>>> ['[CLS]', 'Prime', 'Minister', 'G', '##ei', '##r', 'Ha', '##ard', '##e', 'has', 'refused', 'to', 'resign', 'or'


#If we set label_all_tokens to False.....
label_all_tokens = False

new_label = align_label_example(text_tokenized, label)
print(new_label)
print(tokenizer.convert_ids_to_tokens(text_tokenized["input_ids"][0]))

>>> [-100, 16, 16, 6, -100, -100, 14, -100, -100, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, -100, -100, -100, -100, -
>>> ['[CLS]', 'Prime', 'Minister', 'G', '##ei', '##r', 'Ha', '##ard', '##e', 'has', 'refused', 'to', 'resign', 'or'
```

ner_9.py hosted with ❤ by GitHub view raw

In the rest of this article, we're going to implement the first method, in which we will only provide a label to the first sub-word in each token and set `label_all_tokens` to False.

Dataset Class

Before we train our BERT model for NER task, we need to create a dataset class to generate and fetch data in a batch.

```python
import torch

def align_label(texts, labels):
    tokenized_inputs = tokenizer(texts, padding='max_length', max_length=512, truncation=True)

    word_ids = tokenized_inputs.word_ids()

    previous_word_idx = None
    label_ids = []

    for word_idx in word_ids:

        if word_idx is None:
            label_ids.append(-100)

        elif word_idx != previous_word_idx:
            try:
                label_ids.append(labels_to_ids[labels[word_idx]])
            except:
                label_ids.append(-100)
        else:
            try:
                label_ids.append(labels_to_ids[labels[word_idx]] if label_all_tokens else -100)
            except:
                label_ids.append(-100)
        previous_word_idx = word_idx

    return label_ids

```

```
class DataSequence(torch.utils.data.Dataset):

    def __init__(self, df):

        lb = [i.split() for i in df['labels'].values.tolist()]
        txt = df['text'].values.tolist()
        self.texts = [tokenizer(str(i),
                                padding='max_length', max_length = 512, truncation=True, return_tensors="pt") for i
        self.labels = [align_label(i,j) for i,j in zip(txt, lb)]

```

```
import numpy as np

df = df[0:1000]
df_train, df_val, df_test = np.split(df.sample(frac=1, random_state=42),
                            [int(.8 * len(df)), int(.9 * len(df))])
```

ner_11.py hosted with ❤ by GitHub view raw

```

    def get_batch_labels(self, idx):

        return torch.LongTensor(self.labels[idx])

    def __getitem__(self, idx):

        batch_data = self.get_batch_data(idx)
        batch_labels = self.get_batch_labels(idx)

        return batch_data, batch_labels
```

ner_10.py hosted with ❤ by GitHub view raw

BERT model that will act as token-level classifiers.

```python
from transformers import BertForTokenClassification

class BertModel(torch.nn.Module):

    def __init__(self):

        super(BertModel, self).__init__()

        self.bert = BertForTokenClassification.from_pretrained('bert-base-cased', num_labels=len(unique_labels))

    def forward(self, input_id, mask, label):

        output = self.bert(input_ids=input_id, attention_mask=mask, labels=label, return_dict=False)

        return output
```

ner_12.py hosted with ❤ by GitHub view raw

In the code snippet above, first, we instantiate the model and set the output of each token classifier equal to the number of unique entities on our dataset, which in our case is 17.

Next, we will define a function for the training loop.

Training Loop

The training loop for our BERT model is the standard PyTorch training loop with a few additions, as you can see below:

```python
def train_loop(model, df_train, df_val):

    train_dataset = DataSequence(df_train)
    val_dataset = DataSequence(df_val)

    train_dataloader = DataLoader(train_dataset, num_workers=4, batch_size=BATCH_SIZE, shuffle=True)
    val_dataloader = DataLoader(val_dataset, num_workers=4, batch_size=BATCH_SIZE)

    use_cuda = torch.cuda.is_available()
    device = torch.device("cuda" if use_cuda else "cpu")

    optimizer = SGD(model.parameters(), lr=LEARNING_RATE)

    if use_cuda:
        model = model.cuda()

    best_acc = 0
    best_loss = 1000

    for epoch_num in range(EPOCHS):

        total_acc_train = 0
        total_loss_train = 0

        model.train()

        for train_data, train_label in tqdm(train_dataloader):

            train_label = train_label.to(device)
```

```
            mask = train_data['attention_mask'].squeeze(1).to(device)
            input_id = train_data['input_ids'].squeeze(1).to(device)

            optimizer.zero_grad()
            loss, logits = model(input_id, mask, train_label)

            for i in range(logits.shape[0]):

              logits_clean = logits[i][train_label[i] != -100]
              label_clean = train_label[i][train_label[i] != -100]
```

```
100%|██████████| 800/800 [02:41<00:00,  4.96it/s]
Epochs: 1 | Loss:  0.477 | Accuracy:  0.871 | Val_Loss:  0.362 | Accuracy:  0.895
100%|██████████| 800/800 [02:41<00:00,  4.96it/s]
Epochs: 2 | Loss:  0.343 | Accuracy:  0.903 | Val_Loss:  0.326 | Accuracy:  0.912
100%|██████████| 800/800 [02:41<00:00,  4.96it/s]
Epochs: 3 | Loss:  0.278 | Accuracy:  0.917 | Val_Loss:  0.290 | Accuracy:  0.915
100%|██████████| 800/800 [02:41<00:00,  4.96it/s]
Epochs: 4 | Loss:  0.216 | Accuracy:  0.932 | Val_Loss:  0.308 | Accuracy:  0.919
100%|██████████| 800/800 [02:41<00:00,  4.96it/s]
Epochs: 5 | Loss:  0.172 | Accuracy:  0.945 | Val_Loss:  0.285 | Accuracy:  0.926
```

```

        total_acc_val = 0
        total_loss_val = 0

        for val_data, val_label in val_dataloader:

            val_label = val_label.to(device)
            mask = val_data['attention_mask'].squeeze(1).to(device)
            input_id = val_data['input_ids'].squeeze(1).to(device)
```

```
            loss, logits = model(input_id, mask, val_label)

            for i in range(logits.shape[0]):

              logits_clean = logits[i][val_label[i] != -100]
              label_clean = val_label[i][val_label[i] != -100]

              predictions = logits_clean.argmax(dim=1)
              acc = (predictions == label_clean).float().mean()
              total_acc_val += acc
              total_loss_val += loss.item()

        val_accuracy = total_acc_val / len(df_val)
        val_loss = total_loss_val / len(df_val)

        print(
            f'Epochs: {epoch_num + 1} | Loss: {total_loss_train / len(df_train): .3f} | Accuracy: {total_acc_train

LEARNING_RATE = 5e-3
EPOCHS = 5
BATCH_SIZE = 2

model = BertModel()
train_loop(model, df_train, df_val)
```

```
def evaluate(model, df_test):

    test_dataset = DataSequence(df_test)

    test_dataloader = DataLoader(test_dataset, num_workers=4, batch_size=1)

    use_cuda = torch.cuda.is_available()
    device = torch.device("cuda" if use_cuda else "cpu")

    if use_cuda:
        model = model.cuda()

    total_acc_test = 0.0

    for test_data, test_label in test_dataloader:

            test_label = test_label.to(device)
            mask = test_data['attention_mask'].squeeze(1).to(device)

            input_id = test_data['input_ids'].squeeze(1).to(device)

            loss, logits = model(input_id, mask, test_label)

            for i in range(logits.shape[0]):

              logits_clean = logits[i][test_label[i] != -100]
              label_clean = test_label[i][test_label[i] != -100]

              predictions = logits_clean.argmax(dim=1)
```

```
                acc = (predictions == label_clean).float().mean()
                total_acc_test += acc

    val_accuracy = total_acc_test / len(df_test)
    print(f'Test Accuracy: {total_acc_test / len(df_test): .3f}')


evaluate(model, df_test)
```

```
def align_word_ids(texts):

    tokenized_inputs = tokenizer(texts, padding='max_length', max_length=512, truncation=True)

    word_ids = tokenized_inputs.word_ids()

    previous_word_idx = None
    label_ids = []

    for word_idx in word_ids:

        if word_idx is None:
            label_ids.append(-100)

        elif word_idx != previous_word_idx:
            try:
                label_ids.append(1)
            except:
                label_ids.append(-100)
        else:
            try:
                label_ids.append(1 if label_all_tokens else -100)
            except:
                label_ids.append(-100)
        previous_word_idx = word_idx

    return label_ids


```

```python
def evaluate_one_text(model, sentence):



    use_cuda = torch.cuda.is_available()
    device = torch.device("cuda" if use_cuda else "cpu")

    if use_cuda:
        model = model.cuda()

    text = tokenizer(sentence, padding='max_length', max_length = 512, truncation=True, return_tensors="pt")

    mask = text['attention_mask'].to(device)
    input_id = text['input_ids'].to(device)
    label_ids = torch.Tensor(align_word_ids(sentence)).unsqueeze(0).to(device)

    logits = model(input_id, mask, None)
    logits_clean = logits[0][label_ids != -100]

    predictions = logits_clean.argmax(dim=1).tolist()
    prediction_label = [ids_to_labels[i] for i in predictions]
    print(sentence)
    print(prediction_label)

evaluate_one_text(model, 'Bill Gates is the founder of Microsoft')

>>> 'Bill Gates is the founder of Microsoft'
>>> ['B-per', 'I-per', 'O', 'O', 'O', 'O', 'B-org']
```