Chapter 10. Creating Text Embedding Models

Text embedding models lie at the foundation of many powerful natural language processing applications. They lay the groundwork for empowering already impressive technologies such as text generation models. We have already used embedding models throughout this book in a number of applications, such as supervised classification, unsupervised classification, semantic search, and even giving memory to text generation models like ChatGPT.

It is nearly impossible to overstate the importance of embedding models in the field as they are the driving power behind so many applications. As such, in this chapter, we will discuss a variety of ways that we can create and fine-tune an embedding model to increase its representative and semantic power.

Let's start by discovering what embedding models are and how they generally work.

Embedding Models

Embeddings and embedding models have already been discussed in quite a number of chapters (Chapters 4, 5, and 8) thereby demonstrating their usefulness. Before going into training such a model, let's recap what we have learned with embedding models.

Unstructured textual data by itself is often quite hard to process. They are not values we can directly process, visualize, and create actionable results from. We first have to convert this textual data to something that we can easily process: numeric representations. This process is often referred to as *embedding* the input to output usable vectors, namely *embeddings*, as shown in Figure 10-1.

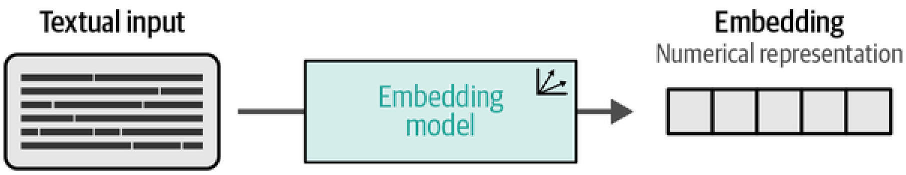

Figure 10-1. We use an embedding model to convert textual input, such as documents, sentences, and phrases, to numerical representations, called embeddings.

This process of embedding the input is typically performed by an LLM, which we refer to as an *embedding model*. The main purpose of such a model is to be as accurate as possible in representing the textual data as an embedding.

However, what does it mean to be accurate in representation? Typically, we want to capture the *semantic nature*—the meaning—of documents. If we can capture the core of what the document communicates, we hope to have captured what the document is about. In practice, this means that we expect vectors of documents that are similar to one another to be similar, whereas the embeddings of documents that each discuss something entirely different should be dissimilar. We've seen this idea of semantic similarity several times already in this book, and it is visualized in Figure 10-2. This figure is a simplified example. While two-dimensional visualization helps illustrate the proximity and similarity of embeddings, these embeddings typically reside in high-dimensional spaces.

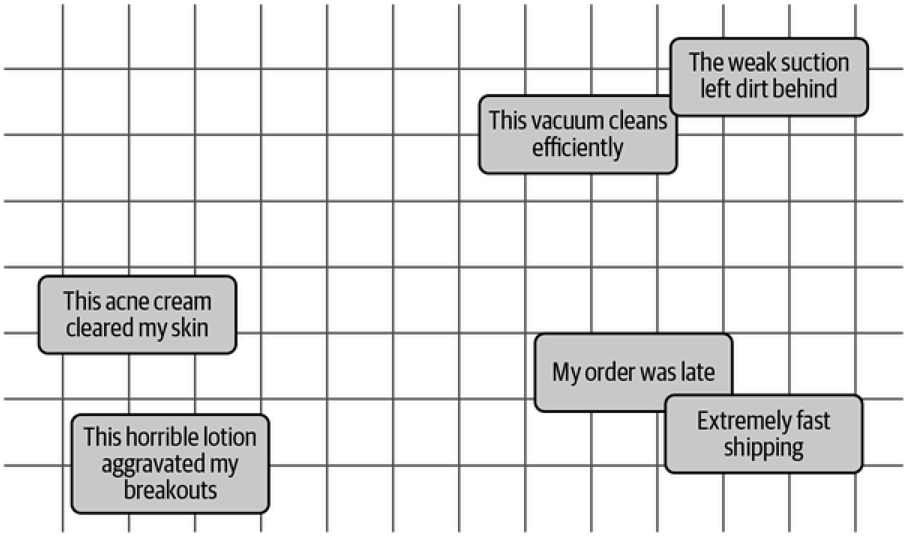
The weak suction left dirt behind
This vacuum cleans efficiently
This acne cream cleared my skin
My order was late
Extremely fast shipping
This horrible lotion aggravated my breakouts

Figure 10-2. The idea of semantic similarity is that we expect textual data with similar meanings to be closer to each other in n-dimensional space (two dimensions are illustrated here).

An embedding model, however, can be trained for a number of purposes. For example, when we are building a sentiment classifier, we are more interested in the sentiment of texts than their semantic similarity. As illustrated in Figure 10-3, we can fine-tune the model such that documents are closer in n-dimensional space based on their sentiment rather than their semantic nature.

Either way, an embedding model aims to learn what makes certain documents similar to one another and we can guide this process. By presenting the model with enough examples of semantically similar documents, we can steer toward semantics whereas using examples of sentiment would steer it in that direction.

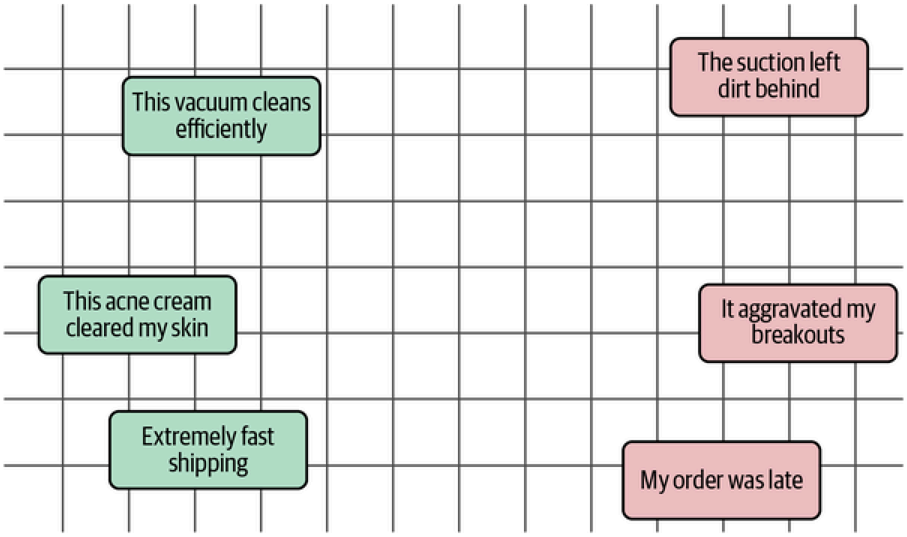
This vacuum cleans efficiently
The suction left dirt behind
This acne cream cleared my skin
It aggravated my breakouts
Extremely fast shipping
My order was late

Figure 10-3. In addition to semantic similarity, an embedding model can be trained to focus on sentiment similarity. In this figure, negative reviews (red) are close to one another and dissimilar to positive reviews (green).

There are many ways in which we can train, fine-tune, and guide embedding models, but one of the strongest and most widely used techniques is called contrastive learning.

What Is Contrastive Learning?

One major technique for both training and fine-tuning text embedding models is called *contrastive learning*. Contrastive learning is a technique that aims to train an embedding model such that similar documents are closer in vector space while dissimilar documents are further apart. If this sounds familiar, it's because it's very similar to the word2vec method from Chapter 2. We have seen this notion previously in Figures 10-2 and 10-3.

The underlying idea of contrastive learning is that the best way to learn and model similarity/dissimilarity between documents is by feeding a model examples of similar and dissimilar pairs. In order to accurately capture the semantic nature of a document, it often needs to be contrasted with another document for a model to learn what makes it different or similar. This contrasting procedure is quite powerful and relates to the context in which documents are written. This high-level procedure is demonstrated in Figure 10-4.

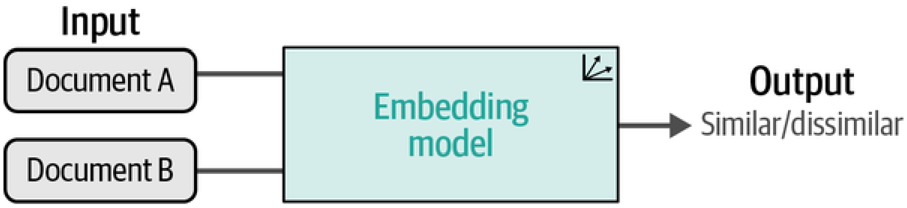

Figure 10-4. Contrastive learning aims to teach an embedding model whether documents are similar or dissimilar. It does so by presenting groups of documents to a model that are similar or dissimilar to a certain degree.

Another way to look at contrastive learning is through the nature of explanations. A nice example of this is an anecdotal story of a reporter asking a robber "Why did you rob a bank?" to which he answers, "Because that is where the money is."[1] Although a factually correct answer, the intent of the question was not why he robs banks specifically but why he robs at all. This is called *contrastive explanation* and refers to understanding a particular case, "Why P?" in contrast to alternatives, "Why P and not Q?"[2] In the example, the question could be interpreted in a number of ways and may be best modeled by providing an alternative: "Why did you rob a bank (P) instead of obeying the law (Q)?"

The importance of alternatives to the understanding of a question also applies to how an embedding learns through contrastive learning. By showing a model similar and dissimilar pairs of documents, it starts to learn what makes something similar/dissimilar and more importantly, why.

For example, you could teach a model to understand what a dog is by letting it find features such as "tail," "nose," "four legs," etc. This learning process can be quite difficult since features are often not well-defined and can be interpreted in a number of ways. A being with a "tail," "nose," and "four legs" can also be a cat. To help the model steer toward what we are interested in, we essentially ask it, "Why is this a dog and not a cat?" By providing the contrast between two concepts, it starts to learn the features that define the concept but also the features that are not related. We get more information when we frame a question as a contrast. We further illustrate this concept of contrastive explanation in Figure 10-5.

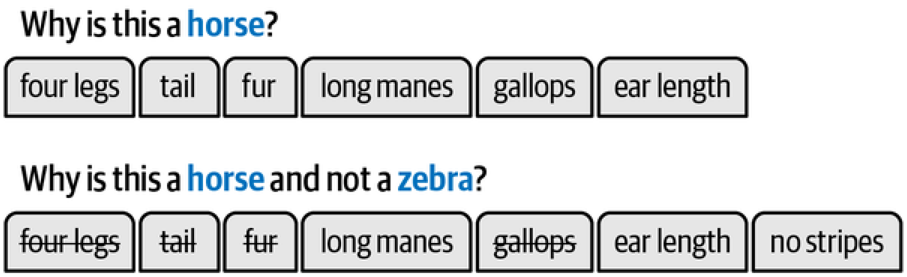

Figure 10-5. When we feed an embedding model different contrasts (degrees of similarity), it starts to learn what makes things different from one another and thereby the distinctive characteristics of concepts.

NOTE

One of the earliest and most popular examples of contrastive learning in NLP is actually word2vec, as we discussed in Chapters 1 and 2. The model learns word representations by training on individual words in a sentence. A word close to a target word in a sentence will be constructed as a positive pair whereas randomly sampled words constitute dissimilar pairs. In other words, positive examples of neighboring words are contrasted with randomly selected words that are not neighbors. Although not widely known, it is one of the first major breakthroughs in NLP that leverages contrastive learning with neural networks.

There are many ways we can apply contrastive learning to create text embedding models but the most well-known technique and framework is `sentence-transformers`.

SBERT

Although there are many forms of contrastive learning, one framework that has popularized the technique within the natural language processing community is `sentence-transformers`.[3] Its approach fixes a major problem with the original BERT implementation for creating sentence embeddings, namely its computational overhead. Before `sentence-transformers`, sentence embeddings often used an architectural structure called cross-encoders with BERT.

A cross-encoder allows two sentences to be passed to the Transformer network simultaneously to predict the extent to which the two sentences are similar. It does so by adding a classification head to the original architecture that can output a similarity score. However, the number of computations rises quickly when you want to find the highest pair in a collection of 10,000 sentences. That would require n·(n−1)/2 = 49,995,000 inference computations and therefore generates significant overhead. Moreover, a cross-encoder generally does not generate embeddings, as shown in Figure 10-6. Instead, it outputs a similarity score between the input sentences.

A solution to this overhead is to generate embeddings from a BERT model by averaging its output layer or using the `[CLS]` token. This, however, has shown to be worse than simply averaging word vectors, like GloVe.[4]

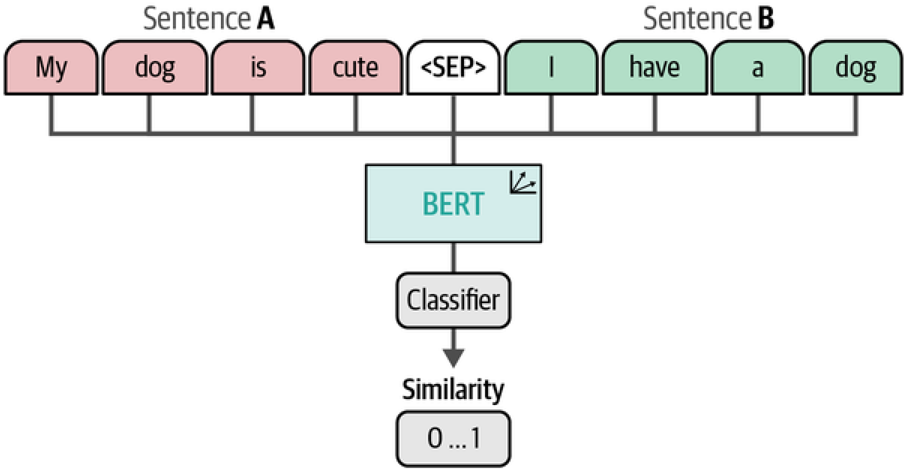

Figure 10-6. The architecture of a cross-encoder. Both sentences are concatenated, separated with a `<SEP>` token, and fed to the model simultaneously.

Instead, the authors of `sentence-transformers` approached the problem differently and searched for a method that is fast and creates embeddings that can be compared semantically. The result is an elegant alternative to the original cross-encoder architecture. Unlike a cross-encoder, in `sentence-transformers` the classification head is dropped, and instead mean pooling is used on the final output layer to generate an embedding. This pooling layer averages the word embeddings and gives back a fixed dimensional output vector. This ensures a fixed-size embedding.

The training for `sentence-transformers` uses a Siamese architecture. In this architecture, as visualized in Figure 10-7, we have two identical BERT models that share the same weights and neural architecture. These models are fed the sentences from which embeddings are generated through the pooling of token embeddings. Then, models are optimized through the similarity of the sentence embeddings. Since the weights are identical for both BERT models, we can use a single model and feed it the sentences one after the other.

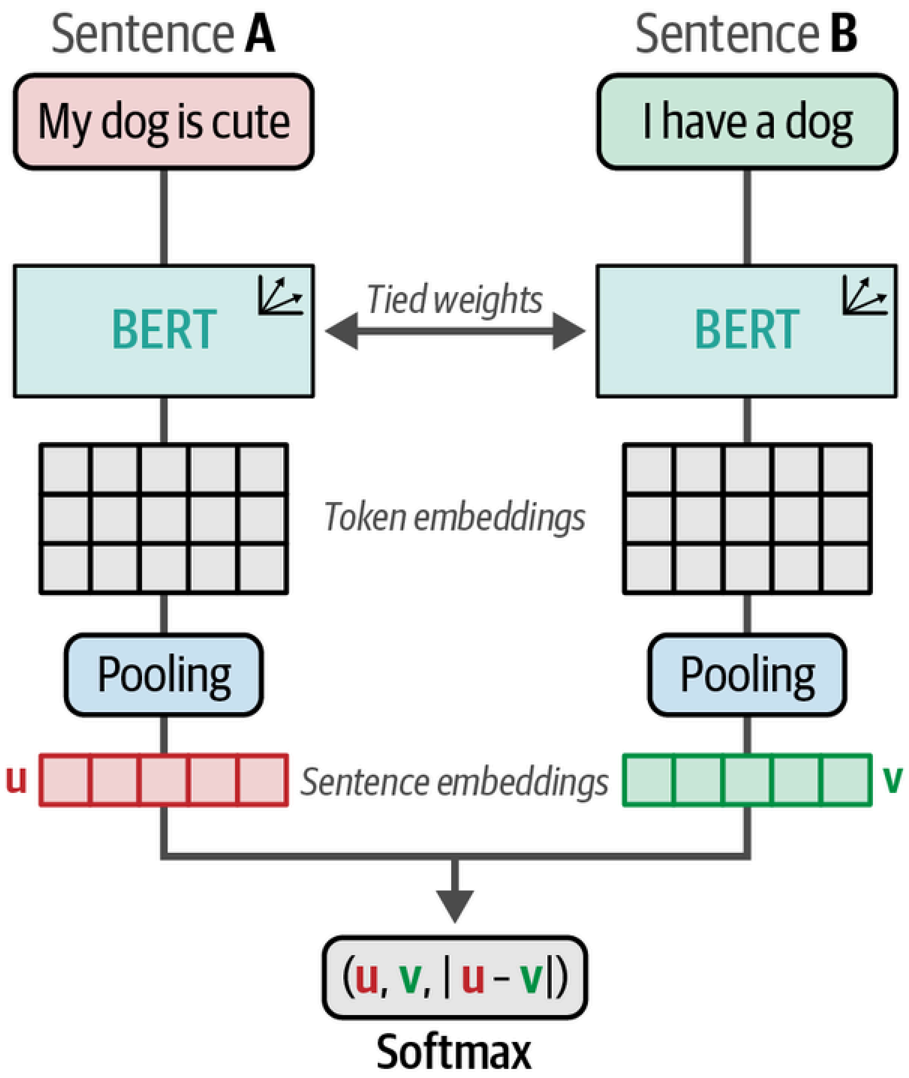

Figure 10-7. The architecture of the original `sentence-transformers` model, which leverages a Siamese network, also called a bi-encoder.

The optimization process of these pairs of sentences is done through loss functions, which can have a major impact on the model's performance. During training, the embeddings for each sentence are concatenated together with the difference between the embeddings. Then, this resulting embedding is optimized through a softmax classifier.

The resulting architecture is also referred to as a bi-encoder or SBERT for sentence-BERT. Although a bi-encoder is quite fast and creates accurate sentence representations, cross-encoders generally achieve better performance than a bi-encoder but do not generate embeddings.

The bi-encoder, like a cross-encoder, leverages contrastive learning; by optimizing the (dis)similarity between pairs of sentences, the model will eventually learn the things that make the sentences what they are.

To perform contrastive learning, we need two things. First, we need data that constitutes similar/dissimilar pairs. Second, we will need to define how the model defines and optimizes similarity.

Creating an Embedding Model

There are many methods through which an embedding model can be created but generally, we look toward contrastive learning. This is an important aspect of many embedding models as the process allows it to efficiently learn semantic representations.

However, this is not a free process. We will need to understand how to generate contrastive examples, how to train the model, and how to properly evaluate it.

Generating Contrastive Examples

When pretraining your embedding model, you will often see data being used from natural language inference (NLI) datasets. NLI refers to the task of investigating whether, for a given premise, it entails the hypothesis (entailment), contradicts it (contradiction), or neither (neutral).

For example, when the premise is “He is in the cinema watching *Coco*” and the hypothesis “He is watching *Frozen* at home,” then these statements are contradictions. In contrast, when the premise is “He is in the cinema watching *Coco*” and the hypothesis “In the movie theater he is watching the Disney movie *Coco*,” then these statements are considered entailment. This principle is illustrated in Figure 10-8.

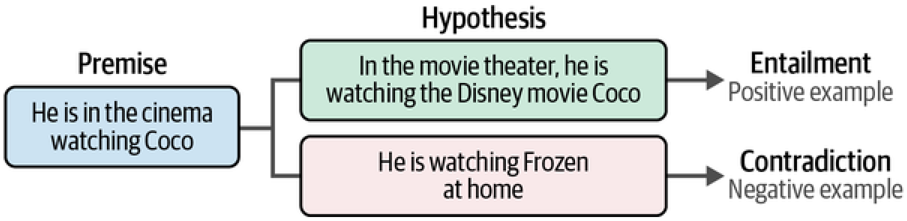
Premise
He is in the cinema watching Coco
Hypothesis
In the movie theater, he is watching the Disney movie Coco
He is watching Frozen at home
Entailment
Positive example
Contradiction
Negative example

Figure 10-8. We can leverage the structure of NLI datasets to generate negative examples (contradiction) and positive examples (entailments) for contrastive learning.

If you look closely at entailment and contradiction, then they describe the extent to which two inputs are similar to one another. As such, we can use NLI datasets to generate negative examples (contradictions) and positive examples (entailments) for contrastive learning.

The data that we are going to be using throughout creating and fine-tuning embedding models is derived from the General Language Understanding Evaluation benchmark (GLUE). This GLUE benchmark consists of nine language understanding tasks to evaluate and analyze model performance.

One of these tasks is the Multi-Genre Natural Language Inference (MNLI) corpus, which is a collection of 392,702 sentence pairs annotated with entailment (contradiction, neutral, entailment). We will be using a subset of the data, 50,000 annotated sentence pairs, to create a minimal example that does not need to be trained for hours on end. Do note, though, that the smaller the dataset, the more unstable training or fine-tuning an embedding model is. If possible, larger datasets are preferred assuming it is still quality data:

```python
from datasets import load_dataset

# Load MNLI dataset from GLUE
# 0 = entailment, 1 = neutral, 2 = contradiction
train_dataset = load_dataset(
    "glue", "mnli", split="train"
).select(range(50_000))
train_dataset = train_dataset.remove_columns("idx")
```

Next, we take a look at an example:

```python
dataset[2]
```

```
{'premise': 'One of our number will carry out your instructions minutely.',
 'hypothesis': 'A member of my team will execute your orders with immense precision.',
 'label': 0}
```

This shows an example of an entailment between the premise and the hypothesis as they are positively related and have near identical meanings.

Train Model

Now that we have our dataset with training examples, we will need to create our embedding model. We typically choose an existing `sentence-transformers` model and fine-tune that model, but in this example, we are going to train an embedding from scratch.

This means that we will have to define two things. First, a pretrained Transformer model that serves as embedding individual words. We will use the BERT base model (uncased) as it is a great introduction model. However, many others exist that also have been evaluated using `sentence-transformers`. Most notably, `microsoft/mpnet-base` often gives good results when used as a word embedding model.

```
from sentence_transformers import SentenceTransformer

# Use a base model
embedding_model = SentenceTransformer('bert-base-uncased')
```

NOTE

By default, all layers of an LLM in `sentence-transformers` are trainable. Although it is possible to freeze certain layers, it is generally not advised since the performance is often better when unfreezing all layers.

Next, we will need to define a loss function over which we will optimize the model. As mentioned at the beginning of this section, one of the first instances of `sentence-transformers` uses softmax loss. For illustrative purposes, we are going to be using that for now, but we will go into more performant losses later on:

```
from sentence_transformers import losses

# Define the loss function. In softmax loss, we will also need to explicitly set the number
train_loss = losses.SoftmaxLoss(
    model=embedding_model,
    sentence_embedding_dimension=embedding_model.get_sentence_embedding_dimension(),
    num_labels=3
)
```

Before we train our model, we define an evaluator to evaluate the model's performance during training, which also determines the best model to save.

We can perform evaluation of the performance of our model using the Semantic Textual Similarity Benchmark (STSB). It is a collection of human-labeled sentence pairs, with similarity scores between 1 and 5.

We use this dataset to explore how well our model scores on this semantic similarity task. Moreover, we process the STSB data to make sure all values are between 0 and 1:

```
from sentence_transformers.evaluation import EmbeddingSimilarityEvaluator

# Create an embedding similarity evaluator for STSB
val_sts = load_dataset("glue", "stsb", split="validation")
evaluator = EmbeddingSimilarityEvaluator(
    sentences1=val_sts["sentence1"],
    sentences2=val_sts["sentence2"],
    scores=[score/5 for score in val_sts["label"]],
    main_similarity="cosine",
)
```

Now that we have our evaluator, we create `SentenceTransformerTrainingArguments`, similar to training with Hugging Face Transformers (as we will explore in the next chapter):

```
from sentence_transformers.training_args import SentenceTransformerTrainingArguments

# Define the training arguments
args = SentenceTransformerTrainingArguments(
    output_dir="base_embedding_model",
    num_train_epochs=1,
    per_device_train_batch_size=32,
    per_device_eval_batch_size=32,
    warmup_steps=100,
    fp16=True,
    eval_steps=100,
    logging_steps=100,
)
```

Of note are the following arguments:

num_train_epochs
: The number of training rounds. We keep this at 1 for faster training but it is generally advised to increase this value.

per_device_train_batch_size
: The number of samples to process simultaneously on each device (e.g., GPU or CPU) during evaluation. Higher values generally means faster training.

per_device_eval_batch_size
: The number of samples to process simultaneously on each device (e.g., GPU or CPU) during evaluation. Higher values generally

means faster evaluation.

warmup_steps

The number of steps during which the learning rate will be linearly increased from zero to the initial learning rate defined for the training process. Note that we did not specify a custom learning rate for this training process.

fp16

By enabling this parameter we allow for mixed precision training, where computations are performed using 16-bit floating-point numbers (FP16) instead of the default 32-bit (FP32). This reduces memory usage and potentially increases the training speed.

Now that we have defined our data, embedding model, loss, and evaluator, we can start training our model. We can do that using `SentenceTransformerTrainer`:

```
from sentence_transformers.trainer import SentenceTransformerTrainer

# Train embedding model
trainer = SentenceTransformerTrainer(
    model=embedding_model,
    args=args,
    train_dataset=train_dataset,
    loss=train_loss,
    evaluator=evaluator
)
trainer.train()
```

After training our model, we can use the evaluator to get the performance on this single task:

```
# Evaluate our trained model
evaluator(embedding_model)
```

```
{'pearson_cosine': 0.5982288436666162,
 'spearman_cosine': 0.6026682018489217,
 'pearson_manhattan': 0.6100690915500567,
 'spearman_manhattan': 0.617732600131989,
 'pearson_euclidean': 0.6079280934202278,
 'spearman_euclidean': 0.6158926913905742,
 'pearson_dot': 0.38364924527804595,
 'spearman_dot': 0.37008497926991796,
 'pearson_max': 0.6100690915500567,
 'spearman_max': 0.617732600131989}
```

We get several different distance measures. The one we are interested in most is `'pearson_cosine'`, which is the cosine similarity between centered vectors. It is a value between 0 and 1 where a higher value indicates higher degrees of similarity. We get a value of 0.59, which we consider a baseline throughout this chapter.

TIP

Larger batch sizes tend to work better with multiple negative rankings (MNR) loss as a larger batch makes the task more difficult. The reason for this is that the model needs to find the best matching sentence from a larger set of potential pairs of sentences. You can adapt the code to try out different batch sizes and get a feeling of its effects.

In-Depth Evaluation

A good embedding model is more than just a good score on the STSB benchmark! As we observed earlier, the GLUE benchmark has a number of tasks for which we can evaluate our embedding model. However, there exist many more benchmarks that allow for the evaluation of embedding models. To unify this evaluation procedure, the Massive Text Embedding Benchmark (MTEB) was developed.[5] The MTEB spans 8 embedding tasks that cover 58 datasets and 112 languages.

To publicly compare state-of-the-art embedding models, a leaderboard was created with the scores of each embedding model across all tasks:

```
from mteb import MTEB

# Choose evaluation task
evaluation = MTEB(tasks=["Banking77Classification"])

# Calculate results
results = evaluation.run(model)
```

```
{'Banking77Classification': {'mteb_version': '1.1.2',
 'dataset_revision': '0fd18e25b25c072e09e0d92ab615fda904d66300',
 'mteb_dataset_name': 'Banking77Classification',
 'test': {'accuracy': 0.4926298701298701,
 'f1': 0.49083335791288685,
 'accuracy_stderr': 0.010217785746224237,
 'f1_stderr': 0.010265814957074591,
 'main_score': 0.4926298701298701,
 'evaluation_time': 31.83}}}
```

This gives us several evaluation metrics for this specific task that we can use to explore its performance.

The great thing about this evaluation benchmark is not only the diversity of the tasks and languages but that even the evaluation time is saved. Although many embedding models exist, we typically want those that are both accurate and have low latency. The tasks for which embedding models are used, like semantic search, often benefit from and require fast inference.

Since testing your model on the entire MTEB can take a couple of hours depending on your GPU, we will use the STSB benchmark throughout this chapter instead for illustration purposes.

TIP

Whenever you are done training and evaluating your model, it is important to *restart* the notebook. This will clear your VRAM up for the next training examples throughout this chapter. By restarting the notebook, we can be sure that all VRAM is cleared.

Loss Functions

We trained our model using softmax loss to illustrate how one of the first `sentence-transformers` models was trained. However, not only is there a large variety of loss functions to choose from, but softmax loss is generally not advised as there are more performant losses.

Instead of going through every single loss function out there, there are two loss functions that are typically used and seem to perform generally well, namely:

- Cosine similarity
- Multiple negatives ranking (MNR) loss

NOTE

There are many more loss functions to choose from than just those discussed here. For example, a loss like MarginMSE works great for training or fine-tuning a cross-encoder. There are a number of interesting loss functions implemented in the `sentence-transformers` framework.

Cosine similarity

The cosine similarity loss is an intuitive and easy-to-use loss that works across many different use cases and datasets. It is typically used in semantic textual similarity tasks. In these tasks, a similarity score is assigned to the pairs of texts over which we optimize the model.

Instead of having strictly positive or negative pairs of sentences, we assume pairs of sentences that are similar or dissimilar to a certain degree. Typically, this value lies between 0 and 1 to indicate dissimilarity and similarity, respectively (Figure 10-9).

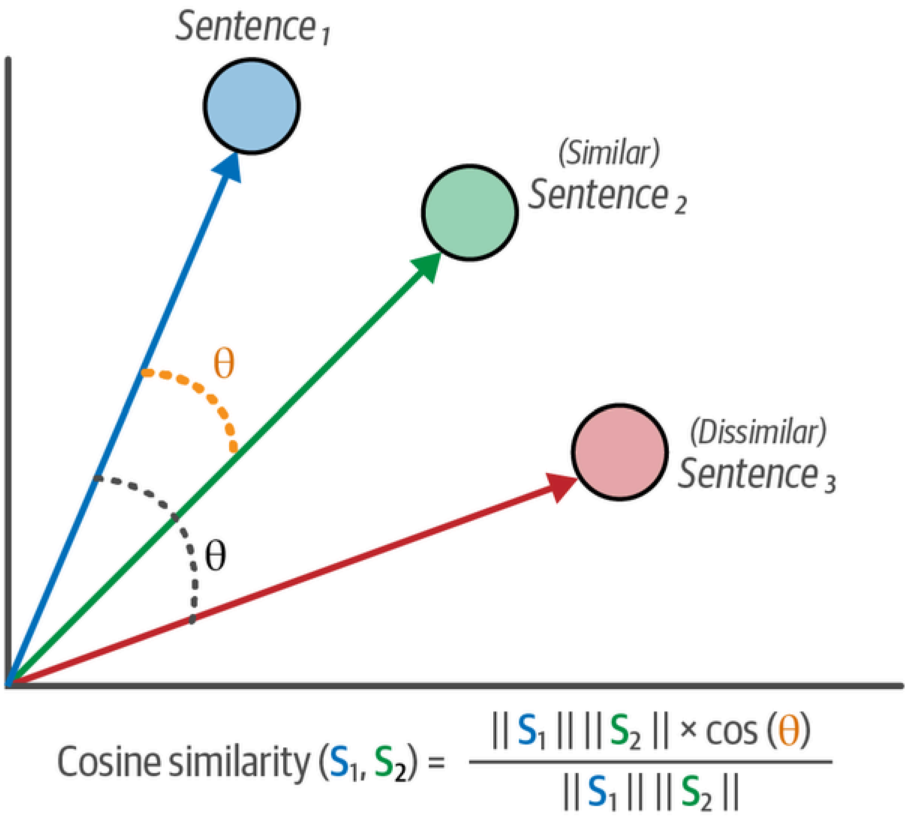
Sentence 1
(Similar)
Sentence 2
(Dissimilar)
Sentence 3
θ
θ
Cosine similarity (S1, S2) = || S1 || || S2 || × cos (θ) / || S1 || || S2 ||

Figure 10-9. Cosine similarity loss aims to minimize the cosine distance between semantically similar sentences and to maximize the distance between semantically dissimilar sentences.

Cosine similarity loss is straightforward—it calculates the cosine similarity between the two embeddings of the two texts and compares that to the labeled similarity score. The model will learn to recognize the degree of similarity between sentences.

Cosine similarity loss intuitively works best using data where you have pairs of sentences and labels that indicate their similarity between 0 and 1. To use this loss with our NLI dataset, we need to convert the entailment (0), neutral (1), and contradiction (2) labels to values between 0 and 1. The entailment represents a high similarity between the sentences, so we give it a similarity score of 1. In contrast, since both neutral and contradiction represent dissimilarity, we give these labels a similarity score of 0:

```
from datasets import Dataset, load_dataset

# Load MNLI dataset from GLUE
# 0 = entailment, 1 = neutral, 2 = contradiction
train_dataset = load_dataset(
    "glue", "mnli", split="train"
).select(range(50_000))
train_dataset = train_dataset.remove_columns("idx")
```

```
# (neutral/contradiction)=0 and (entailment)=1
mapping = {2: 0, 1: 0, 0:1}
train_dataset = Dataset.from_dict({
    "sentence1": train_dataset["premise"],
    "sentence2": train_dataset["hypothesis"],
    "label": [float(mapping[label]) for label in train_dataset["label"]]
})
```

As before, we create our evaluator:

```
from sentence_transformers.evaluation import EmbeddingSimilarityEvaluator

# Create an embedding similarity evaluator for stsb
val_sts = load_dataset("glue", "stsb", split="validation")
evaluator = EmbeddingSimilarityEvaluator(
    sentences1=val_sts["sentence1"],
    sentences2=val_sts["sentence2"],
    scores=[score/5 for score in val_sts["label"]],
    main_similarity="cosine"
)
```

Then, we follow the same steps as before but select a different loss instead:

```
from sentence_transformers import losses, SentenceTransformer
from sentence_transformers.trainer import SentenceTransformerTrainer
from sentence_transformers.training_args import SentenceTransformerTrainingArguments

# Define model
embedding_model = SentenceTransformer("bert-base-uncased")

# Loss function
train_loss = losses.CosineSimilarityLoss(model=embedding_model)

# Define the training arguments
args = SentenceTransformerTrainingArguments(
    output_dir="cosineloss_embedding_model",
    num_train_epochs=1,
    per_device_train_batch_size=32,
    per_device_eval_batch_size=32,
    warmup_steps=100,
    fp16=True,
    eval_steps=100,
    logging_steps=100,
)

# Train model
trainer = SentenceTransformerTrainer(
    model=embedding_model,
    args=args,
    train_dataset=train_dataset,
    loss=train_loss,
    evaluator=evaluator
)
trainer.train()
```

Evaluating the model after training gives us the following score:

```
# Evaluate our trained model
evaluator(embedding_model)
```

```

{'pearson_cosine': 0.7222322163831805,
 'spearman_cosine': 0.7250508271229599,
 'pearson_manhattan': 0.7338163436711481,
 'spearman_manhattan': 0.7323479193408869,
 'pearson_euclidean': 0.7332716434966307,
 'spearman_euclidean': 0.7316999722750905,
 'pearson_dot': 0.660366792336156,
 'spearman_dot': 0.6624167554844425,
 'pearson_max': 0.7338163436711481,
 'spearman_max': 0.7323479193408869}
```

A Pearson cosine score of 0.72 is a big improvement compared to the softmax loss example, which scored 0.59. This demonstrates the impact the loss function can have on performance.

Make sure to *restart* your notebook so we can explore a more common and performant loss, namely multiple negatives ranking loss.

Multiple negatives ranking loss

Multiple negatives ranking (MNR) loss,[6] often referred to as InfoNCE[7] or NTXentLoss,[8] is a loss that uses either positive pairs of sentences or triplets that contain a pair of positive sentences and an additional unrelated sentence. This unrelated sentence is called a negative and represents the dissimilarity between the positive sentences.

For example, you might have pairs of question/answer, image/image caption, paper title/paper abstract, etc. The great thing about these pairs is that we can be confident they are hard positive pairs. In MNR loss (Figure 10-10), negative pairs are constructed by mixing a positive pair with another positive pair. In the example of a paper title and abstract, you would generate a negative pair by combining the title of a paper with a completely different abstract. These negatives are called *in-batch negatives* and can also be used to generate the triplets.

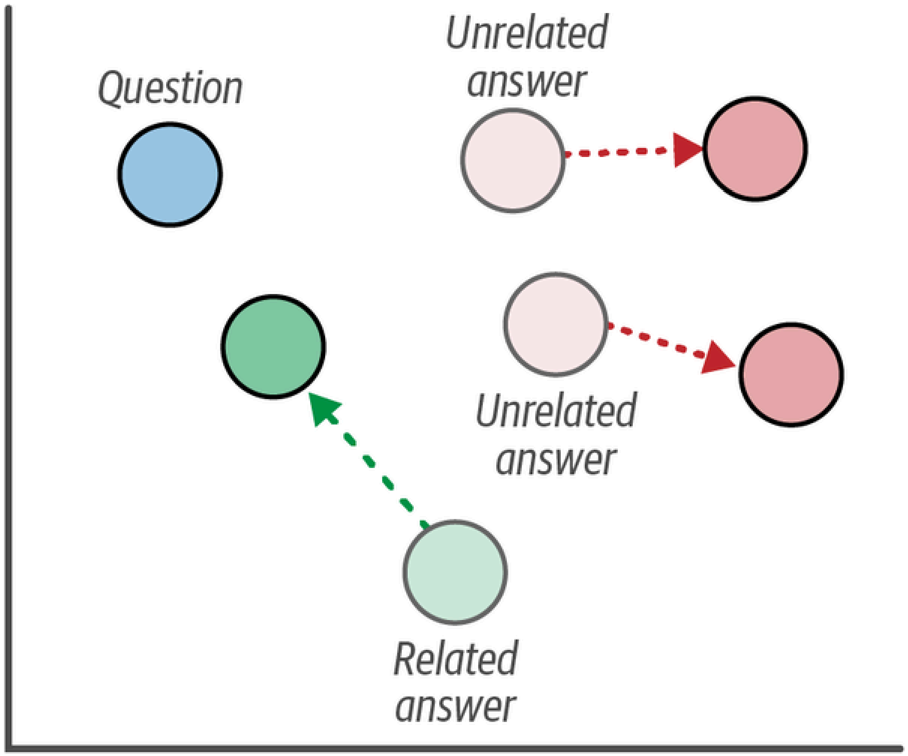

Figure 10-10. Multiple negatives ranking loss aims to minimize the distance between related pairs of text, such as questions and answers, and maximize the distance between unrelated pairs, such as questions and unrelated answers.

After having generated these positive and negative pairs, we calculate their embeddings and apply cosine similarity. These similarity scores are then used to answer the question, are these pairs negative or positive? In other words, it is treated as a classification task and we can use cross-entropy loss to optimize the model.

To make these triplets we start with an anchor sentence (i.e., labeled as the "premise"), which is used to compare other sentences. Then, using the MNLI dataset, we only select sentence pairs that are positive (i.e., labeled as "entailment"). To add negative sentences, we randomly sample sentences as the "hypothesis."

```
import random
from tqdm import tqdm
from datasets import Dataset, load_dataset

# # Load MNLI dataset from GLUE
mnli = load_dataset("glue", "mnli", split="train").select(range(50_000))
mnli = mnli.remove_columns("idx")
mnli = mnli.filter(lambda x: True if x["label"] == 0 else False)

# Prepare data and add a soft negative
train_dataset = {"anchor": [], "positive": [], "negative": []}
soft_negatives = mnli["hypothesis"]
random.shuffle(soft_negatives)
for row, soft_negative in tqdm(zip(mnli, soft_negatives)):
    train_dataset["anchor"].append(row["premise"])
    train_dataset["positive"].append(row["hypothesis"])
    train_dataset["negative"].append(soft_negative)
train_dataset = Dataset.from_dict(train_dataset)
```

Since we only selected sentences labeled with “entailment,” the number of rows reduced quite a a bit from 50,000 to 16,875 rows.

Let’s define the evaluator:

```
from sentence_transformers.evaluation import EmbeddingSimilarityEvaluator
# Create an embedding similarity evaluator for stsb
val_sts = load_dataset("glue", "stsb", split="validation")
evaluator = EmbeddingSimilarityEvaluator(
    sentences1=val_sts["sentence1"],
    sentences2=val_sts["sentence2"],
    scores=[score/5 for score in val_sts["label"]],
    main_similarity="cosine"
)
```

We then train as before but with MNR loss instead:

```
from sentence_transformers import losses, SentenceTransformer
from sentence_transformers.trainer import SentenceTransformerTrainer
from sentence_transformers.training_args import SentenceTransformerTrainingArguments

# Define model
embedding_model = SentenceTransformer('bert-base-uncased')

# Loss function
train_loss = losses.MultipleNegativesRankingLoss(model=embedding_model)

# Define the training arguments
args = SentenceTransformerTrainingArguments(
    output_dir="mnrloss_embedding_model",
    num_train_epochs=1,
    per_device_train_batch_size=32,
    per_device_eval_batch_size=32,
    warmup_steps=100,
    fp16=True,
    eval_steps=100,
    logging_steps=100,
)

# Train model
trainer = SentenceTransformerTrainer(
    model=embedding_model,
    args=args,
    train_dataset=train_dataset,
    loss=train_loss,
    evaluator=evaluator
)
trainer.train()
```

Let's see how this dataset and loss function compare to our previous examples:

```
# Evaluate our trained model
evaluator(embedding_model)
```

```
{'pearson_cosine': 0.8093892326162132,
 'spearman_cosine': 0.8121064796503025,
 'pearson_manhattan': 0.8215001523827565,
 'spearman_manhattan': 0.8172161486524246,
 'pearson_euclidean': 0.8210391407846718,
 'spearman_euclidean': 0.8166537141010816,
 'pearson_dot': 0.7473360302629125,
 'spearman_dot': 0.7345184137194012,
 'pearson_max': 0.8215001523827565,
 'spearman_max': 0.8172161486524246}
```

Compared to our previously trained model with softmax loss (0.72), our model with MNR loss (0.80) seems to be much more accurate!

TIP

Larger batch sizes tend to be better with MNR loss as a larger batch makes the task more difficult. The reason for this is that the model needs to find the best matching sentence from a larger set of potential pairs of sentences. You can adapt the code to try out different batch sizes and get a feeling of the effects.

There is a downside to how we used this loss function. Since negatives are sampled from other question/answer pairs, these in-batch or "easy" negatives that we used could potentially be completely unrelated to the question. As a result, the embedding model's task of then finding the right answer to a question becomes quite easy. Instead, we would like to have negatives that are very related to the question but not the right answer. These negatives are called *hard negatives*. Since this would make the task more difficult for the embedding model as it has to learn more nuanced representations, the embedding model's performance generally improves quite a bit.

A good example of a hard negative is the following. Let's assume we have the following question: "How many people live in Amsterdam?" A related answer to this question would be: "Almost a million people live in Amsterdam." To generate a good hard negative, we ideally want the answer to contain something about Amsterdam and the number of people living in this city. For example: "More than a million people live in Utrecht, which is more than Amsterdam." This answer relates to the question but is not the actual answer, so this would be a good hard negative. Figure 10-11 illustrates the differences between easy and hard negatives.

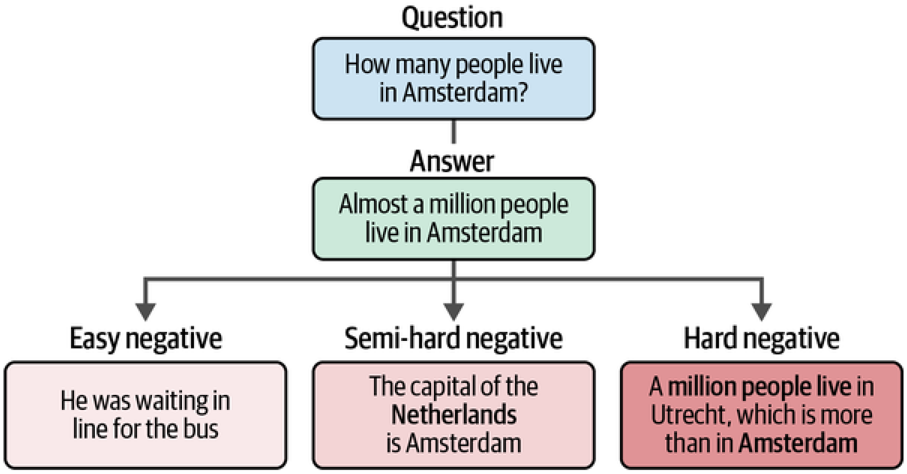
Question
How many people live in Amsterdam?
Answer
Almost a million people live in Amsterdam
Easy negative
He was waiting in line for the bus
Semi-hard negative
The capital of the Netherlands is Amsterdam
Hard negative
A million people live in Utrecht, which is more than in Amsterdam

Figure 10-11. An easy negative is typically unrelated to both the question and answer. A semi-hard negative has some similarities to the topic of the question and answer but is somewhat unrelated. A hard negative is very similar to the question but is generally the wrong answer.

Gathering negatives can roughly be divided into the following three processes:

Easy negatives

Through randomly sampling documents as we did before.

Semi-hard negatives

Using a pretrained embedding model, we can apply cosine similarity on all sentence embeddings to find those that are highly related. Generally, this does not lead to hard negatives since this method merely finds similar sentences, not question/answer pairs.

Hard negatives

These often need to be either manually labeled (for instance, by generating semi-hard negatives) or you can use a generative model to either judge or generate sentence pairs.

Make sure to *restart* your notebook so we can explore the different methods of fine-tuning embedding models.

Fine-Tuning an Embedding Model

In the previous section, we went through the basics of training an embedding model from scratch and saw how we could leverage loss functions to further optimize its performance. This approach, although quite powerful, requires creating an embedding model from scratch. This process can be quite costly and time-consuming.

Instead, the `sentence-transformers` framework allows nearly all embedding models to be used as a base for fine-tuning. We can choose an embedding model that was already trained on a large amount of data and fine-tune it for our specific data or purpose.

There are several ways to fine-tune your model, depending on the data availability and domain. We will go through two such methods and demonstrate the strength of leveraging pretrained embedding models.

Supervised

The most straightforward way to fine-tune an embedding model is to repeat the process of training our model as we did before but replace the `'bert-base-uncased'` with a pretrained `sentence-transformers` model. There are many to choose from but generally, `all-MiniLM-L6-v2` performs well across many use cases and due to its small size is quite fast.

We use the same data as we used to train our model in the MNR loss example but instead use a pretrained embedding model to fine-tune. As always, let's start by loading the data and creating the evaluator:

```
from datasets import load_dataset
from sentence_transformers.evaluation import EmbeddingSimilarityEvaluator

# Load MNLI dataset from GLUE
# 0 = entailment, 1 = neutral, 2 = contradiction
train_dataset = load_dataset(
    "glue", "mnli", split="train"
).select(range(50_000))
train_dataset = train_dataset.remove_columns("idx")

# Create an embedding similarity evaluator for stsb
val_sts = load_dataset("glue", "stsb", split="validation")
evaluator = EmbeddingSimilarityEvaluator(
    sentences1=val_sts["sentence1"],
    sentences2=val_sts["sentence2"],
    scores=[score/5 for score in val_sts["label"]],
    main_similarity="cosine"
)
```

The training steps are similar to our previous examples but instead of using `'bert-base-uncased',` we can use a pretrained embedding model instead:

```
from sentence_transformers import losses, SentenceTransformer
from sentence_transformers.trainer import SentenceTransformerTrainer
from sentence_transformers.training_args import SentenceTransformerTrainingArguments

# Define model
embedding_model = SentenceTransformer('sentence-transformers/all-MiniLM-L6-v2')

# Loss function
```

```
train_loss = losses.MultipleNegativesRankingLoss(model=embedding_model)

# Define the training arguments
args = SentenceTransformerTrainingArguments(
    output_dir="finetuned_embedding_model",
    num_train_epochs=1,
    per_device_train_batch_size=32,
    per_device_eval_batch_size=32,
    warmup_steps=100,
    fp16=True,
    eval_steps=100,
    logging_steps=100,
)

# Train model
trainer = SentenceTransformerTrainer(
    model=embedding_model,
    args=args,
    train_dataset=train_dataset,
    loss=train_loss,
    evaluator=evaluator
)
trainer.train()
```

Evaluating this model gives us the following score:

```
# Evaluate our trained model
evaluator(embedding_model)
```

```

{'pearson_cosine': 0.8509553350510896,
 'spearman_cosine': 0.8484676559567688,
 'pearson_manhattan': 0.8503896832470704,
 'spearman_manhattan': 0.8475760325664419,
 'pearson_euclidean': 0.8513115442079158,
 'spearman_euclidean': 0.8484676559567688,
 'pearson_dot': 0.8489553386816947,
 'spearman_dot': 0.8484676559567688,
 'pearson_max': 0.8513115442079158,
 'spearman_max': 0.8484676559567688}
```

Although a score of 0.85 is the highest we have seen thus far, the pre-trained model that we used for fine-tuning was already trained on the full MNLI dataset, whereas we only used 50,000 examples. It might seem redundant but this example demonstrates how to fine-tune a pretrained embedding model on your own data.

TIP

Instead of using a pretrained BERT model like `'bert-base-uncased'` or a possible out-of-domain model like `'all-mpnet-base-v2'`, you can also perform masked language modeling on the pretrained BERT model to first adapt it to your domain. Then, you can use this fine-tuned BERT model as the base for training your embedding model. This is a form of domain adaptation. In the next chapter, we will apply masked language modeling on a pretrained model.

Note that the main difficulty of training or fine-tuning your model is finding the right data. With these models, we not only want to have very large datasets, but the data in itself needs to be of high quality. Developing positive pairs is generally straightforward but adding hard negative pairs significantly increases the difficulty of creating quality data.

As always, *restart* your notebook to free up VRAM for the following examples.

Augmented SBERT

A disadvantage of training or fine-tuning these embedding models is that they often require substantial training data. Many of these models are trained with more than a billion sentence pairs. Extracting such a high number of sentence pairs for your use case is generally not possible as in many cases, there are only a couple of thousand labeled data points available.

Fortunately, there is a way to augment your data such that an embedding model can be fine-tuned when there is only a little labeled data available. This procedure is referred to as *Augmented SBERT*.[9]

In this procedure, we aim to augment the small amount of labeled data such that they can be used for regular training. It makes use of the slow and more accurate cross-encoder architecture (BERT) to augment and label a larger set of input pairs. These newly labeled pairs are then used for fine-tuning a bi-encoder (SBERT).

As shown in Figure 10-12, Augmented SBERT involves the following steps:

1. Fine-tune a cross-encoder (BERT) using a small, annotated dataset (gold dataset).
2. Create new sentence pairs.
3. Label new sentence pairs with the fine-tuned cross-encoder (silver dataset).
4. Train a bi-encoder (SBERT) on the extended dataset (gold + silver dataset).

Here, a gold dataset is a small but fully annotated dataset that holds the ground truth. A silver dataset is also fully annotated but is not necessarily the ground truth as it was generated through predictions of the cross-encoder.

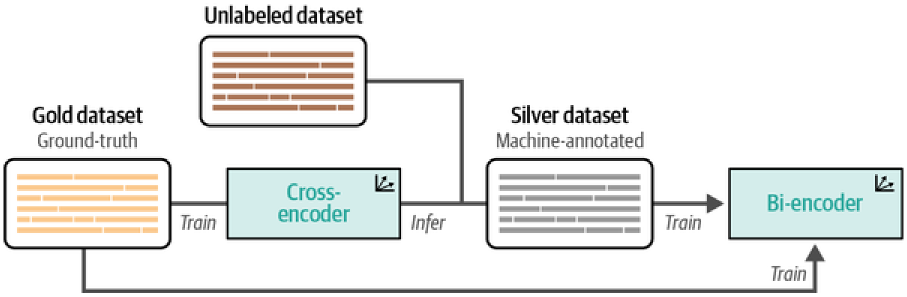

Figure 10-12. Augmented SBERT works through training a cross-encoder on a small gold dataset, then using that to label an unlabeled dataset to generate a larger silver dataset. Finally, both the gold and silver datasets are used to train the bi-encoder.

Before we get into the preceding steps, let's first prepare the data. Instead of our original 50,000 documents, we take a subset of 10,000 documents to simulate a setting where we have limited annotated data. As we did in our example with cosine similarity loss, give entailment a score of 1 whereas neutral and contradiction get a score of 0:

```
import pandas as pd
from tqdm import tqdm
from datasets import load_dataset, Dataset
from sentence_transformers import InputExample
from sentence_transformers.datasets import NoDuplicatesDataLoader

# Prepare a small set of 10000 documents for the cross-encoder
dataset = load_dataset("glue", "mnli", split="train").select(range(10_000))
mapping = {2: 0, 1: 0, 0:1}

# Data loader
gold_examples = [
    InputExample(texts=[row["premise"], row["hypothesis"]], label=mapping[row["label"]])
    for row in tqdm(dataset)
]
gold_dataloader = NoDuplicatesDataLoader(gold_examples, batch_size=32)

# Pandas DataFrame for easier data handling
gold = pd.DataFrame(
    {
    "sentence1": dataset["premise"],
    "sentence2": dataset["hypothesis"],
    "label": [mapping[label] for label in dataset["label"]]
    }
)
```

This is the gold dataset since it is labeled and represents our ground truth.

Using this gold dataset, we train our cross-encoder (step 1):

```
from sentence_transformers.cross_encoder import CrossEncoder

# Train a cross-encoder on the gold dataset
cross_encoder = CrossEncoder("bert-base-uncased", num_labels=2)
cross_encoder.fit(
    train_dataloader=gold_dataloader,
    epochs=1,
    show_progress_bar=True,
    warmup_steps=100,
    use_amp=False
)
```

After training our cross-encoder, we use the remaining 400,000 sentence pairs (from our original dataset of 50,000 sentence pairs) as our silver dataset (step 2):

```
# Prepare the silver dataset by predicting labels with the cross-encoder
silver = load_dataset(
    "glue", "mnli", split="train"
).select(range(10_000, 50_000))
pairs = list(zip(silver["premise"], silver["hypothesis"]))
```

TIP

If you do not have any additional unlabeled sentence pairs, you can randomly sample them from your original gold dataset. To illustrate, you can create a new sentence pair by taking the premise from one row and the hypothesis from another. This allows you to easily generate 10 times as many sentence pairs that can be labeled with the cross-encoder.

This strategy, however, likely generates significantly more dissimilar than similar pairs. Instead, we can use a pretrained embedding model to embed all candidate sentence pairs and retrieve the top-k sentences for each input sentence using semantic search. This rough reranking process allows us to focus on sentence pairs that are likely to be more similar. Although the sentences are still chosen based on an approximation since the pretrained embedding model was not trained on our data, it is much better than random sampling.

Note that we assume that these sentence pairs are unlabeled in this example. We will use our fine-tuned cross-encoder to label these sentence pairs (step 3):

```
import numpy as np

# Label the sentence pairs using our fine-tuned cross-encoder
output = cross_encoder.predict(
    pairs, apply_softmax=True,
show_progress_bar=True
)
silver = pd.DataFrame(
    {
        "sentence1": silver["premise"],
        "sentence2": silver["hypothesis"],
        "label": np.argmax(output, axis=1)
    }
)
```

Now that we have a silver and gold dataset, we simply combine them and train our embedding model as we did before:

```
# Combine gold + silver
data = pd.concat([gold, silver], ignore_index=True, axis=0)
data = data.drop_duplicates(subset=["sentence1", "sentence2"], keep="first")
train_dataset = Dataset.from_pandas(data, preserve_index=False)
```

As always, we need to define our evaluator:

```
from sentence_transformers.evaluation import EmbeddingSimilarityEvaluator

# Create an embedding similarity evaluator for stsb
val_sts = load_dataset("glue", "stsb", split="validation")
evaluator = EmbeddingSimilarityEvaluator(
    sentences1=val_sts["sentence1"],
    sentences2=val_sts["sentence2"],
    scores=[score/5 for score in val_sts["label"]],
    main_similarity="cosine"
)
```

We train the model the same as before except now we use the augmented dataset:

```
from sentence_transformers import losses, SentenceTransformer
from sentence_transformers.trainer import SentenceTransformerTrainer
from sentence_transformers.training_args import SentenceTransformerTrainingArguments

# Define model
embedding_model = SentenceTransformer("bert-base-uncased")

# Loss function
train_loss = losses.CosineSimilarityLoss(model=embedding_model)

# Define the training arguments
args = SentenceTransformerTrainingArguments(
    output_dir="augmented_embedding_model",
    num_train_epochs=1,
    per_device_train_batch_size=32,
    per_device_eval_batch_size=32,
    warmup_steps=100,
    fp16=True,
    eval_steps=100,
    logging_steps=100,
)

# Train model
trainer = SentenceTransformerTrainer(
    model=embedding_model,
    args=args,
    train_dataset=train_dataset,
    loss=train_loss,
    evaluator=evaluator
)
trainer.train()
```

Finally, we evaluate the model:

```
evaluator(embedding_model)
```

```
{'pearson_cosine': 0.7101597020018693,
 'spearman_cosine': 0.7210536464320728,
 'pearson_manhattan': 0.7296749443525249,
 'spearman_manhattan': 0.7284184255293913,
 'pearson_euclidean': 0.7293097297208753,
 'spearman_euclidean': 0.7282830906742256,
 'pearson_dot': 0.6746605824703588,
 'spearman_dot': 0.6754486790570754,
 'pearson_max': 0.7296749443525249,
 'spearman_max': 0.7284184255293913}
```

The original cosine similarity loss example had a score of 0.72 with the full dataset. Using only 20% of that data, we managed to get a score of 0.71!

This method allows us to increase the size of datasets that you already have available without the need to manually label hundreds of thousands of sentence pairs. You can test the quality of your silver data by also training your embedding model only on the gold dataset. The difference in performance indicates how much your silver dataset potentially adds to the quality of the model.

You can *restart* your notebook a final time for the last example, namely unsupervised learning.

Unsupervised Learning

To create an embedding model, we typically need labeled data. However, not all real-world datasets come with a nice set of labels that we can use. We instead look for techniques to train the model without any predetermined labels—unsupervised learning. Many approaches exist, like Simple Contrastive Learning of Sentence Embeddings (SimCSE),[10] Contrastive Tension (CT),[11] Transformer-based Sequential Denoising Auto-Encoder (TSDAE),[12] and Generative Pseudo-Labeling (GPL).[13]

In this section, we will focus on TSDAE, as it has shown great performance on unsupervised tasks as well as domain adaptation.

Transformer-Based Sequential Denoising Auto-Encoder

TSDAE is a very elegant approach to creating an embedding model with unsupervised learning. The method assumes that we have no labeled data at all and does not require us to artificially create labels.

The underlying idea of TSDAE is that we add noise to the input sentence by removing a certain percentage of words from it. This “damaged” sentence is put through an encoder, with a pooling layer on top of it, to map it to a sentence embedding. From this sentence embedding, a decoder tries to reconstruct the original sentence from the “damaged” sentence but without the artificial noise. The main concept here is that the more accurate the sentence embedding is, the more accurate the reconstructed sentence will be.

This method is very similar to masked language modeling, where we try to reconstruct and learn certain masked words. Here, instead of reconstructing masked words, we try to reconstruct the entire sentence.

After training, we can use the encoder to generate embeddings from text since the decoder is only used for judging whether the embeddings can accurately reconstruct the original sentence (Figure 10-13).

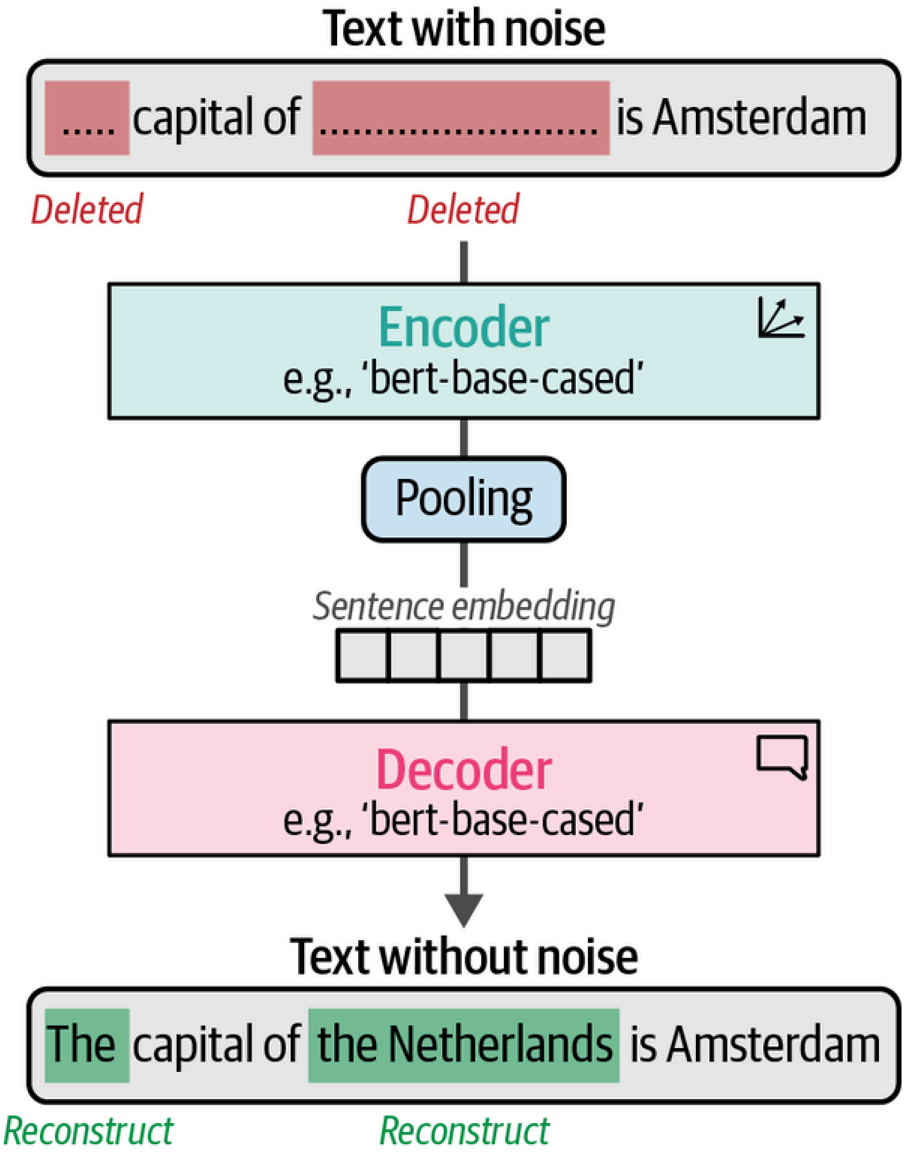

Figure 10-13. TSDAE randomly removes words from an input sentence that is passed through an encoder to generate a sentence embedding. From this sentence embedding, the original sentence is reconstructed.

Since we only need a bunch of sentences without any labels, training this model is straightforward. We start by downloading an external tokenizer, which is used for the denoising procedure:

```
# Download additional tokenizer
import nltk
nltk.download("punkt")
```

Then, we create flat sentences from our data and remove any labels that we have to mimic an unsupervised setting:

```
from tqdm import tqdm
from datasets import Dataset, load_dataset
from sentence_transformers.datasets import DenoisingAutoEncoderDataset

# Create a flat list of sentences
mnli = load_dataset("glue", "mnli", split="train").select(range(25_000))
flat_sentences = mnli["premise"] + mnli["hypothesis"]
```

```
# Add noise to our input data
damaged_data = DenoisingAutoEncoderDataset(list(set(flat_sentences)))

# Create dataset
train_dataset = {"damaged_sentence": [], "original_sentence": []}
for data in tqdm(damaged_data):
    train_dataset["damaged_sentence"].append(data.texts[0])
    train_dataset["original_sentence"].append(data.texts[1])
train_dataset = Dataset.from_dict(train_dataset)
```

This creates a dataset of 50,000 sentences. When we inspect the data, notice that the first sentence is the damaged sentence and the second sentence the original:

```
train_dataset[0]
```

```
{'damaged_sentence': 'Grim jaws are.',
 'original_sentence': 'Grim faces and hardened jaws are not people-friendly.'}
```

The first sentence shows the “noisy” data whereas the second shows the original input sentence. After creating our data, we define our evaluator as before:

```
from sentence_transformers.evaluation import EmbeddingSimilarityEvaluator

# Create an embedding similarity evaluator for stsb
val_sts = load_dataset("glue", "stsb", split="validation")
evaluator = EmbeddingSimilarityEvaluator(
    sentences1=val_sts["sentence1"],
    sentences2=val_sts["sentence2"],
    scores=[score/5 for score in val_sts["label"]],
    main_similarity="cosine"
)
```

Next, we run the training as before but with the `[CLS]` token as the pooling strategy instead of the mean pooling of the token embeddings. In the TSDAE paper, this was shown to be more effective since mean pooling loses the position information, which is not the case when using the `[CLS]` token:

```
from sentence_transformers import models, SentenceTransformer

# Create your embedding model
word_embedding_model = models.Transformer("bert-base-uncased")
pooling_model = models.Pooling(word_embedding_model.get_word_embedding_dimension(), "cls")
embedding_model = SentenceTransformer(modules=[word_embedding_model, pooling_model])
```

Using our sentence pairs, we will need a loss function that attempts to reconstruct the original sentence using the noise sentence, namely `DenoisingAutoEncoderLoss`. By doing so, it will learn how to accurately represent the data. It is similar to masking but without knowing where the actual masks are.

Moreover, we tie the parameters of both models. Instead of having separate weights for the encoder's embedding layer and the decoder's output layer, they share the same weights. This means that any updates to the weights in one layer will be reflected in the other layer as well:

```
from sentence_transformers import losses

# Use the denoising auto-encoder loss
train_loss = losses.DenoisingAutoEncoderLoss(
    embedding_model, tie_encoder_decoder=True
)
train_loss.decoder = train_loss.decoder.to("cuda")
```

Finally, training our model works the same as we have seen several times before but we lower the batch size as memory increases with this loss function:

```
from sentence_transformers.trainer import SentenceTransformerTrainer
from sentence_transformers.training_args import SentenceTransformerTrainingArguments

# Define the training arguments
args = SentenceTransformerTrainingArguments(
    output_dir="tsdae_embedding_model",
    num_train_epochs=1,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    warmup_steps=100,
    fp16=True,
    eval_steps=100,
    logging_steps=100,
)

# Train model
trainer = SentenceTransformerTrainer(
    model=embedding_model,
    args=args,
    train_dataset=train_dataset,
    loss=train_loss,
    evaluator=evaluator
)
trainer.train()
```

After training, we evaluate our model to explore how well such an unsupervised technique performs:

```
# Evaluate our trained model
evaluator(embedding_model)
```

```
{'pearson_cosine': 0.6991809700971775,
 'spearman_cosine': 0.713693213167873,
 'pearson_manhattan': 0.7152343356643568,
 'spearman_manhattan': 0.7201441944880915,
 'pearson_euclidean': 0.7151142243297436,
 'spearman_euclidean': 0.7202291660769805,
 'pearson_dot': 0.5198066451871277,
 'spearman_dot': 0.5104025515225046,
 'pearson_max': 0.7152343356643568,
 'spearman_max': 0.7202291660769805}
```

After fitting our model, we got a score of 0.70, which is quite impressive considering we did all this training with unlabeled data.

Using TSDAE for Domain Adaptation

When you have very little or no labeled data available, you typically use unsupervised learning to create your text embedding model. However, unsupervised techniques are generally outperformed by supervised techniques and have difficulty learning domain-specific concepts.

This is where *domain adaptation* comes in. Its goal is to update existing embedding models to a specific textual domain that contains different subjects from the source domain. Figure 10-14 demonstrates how domains can differ in content. The target domain, or out-domain, generally contains words and subjects that were not found in the source domain or in-domain.

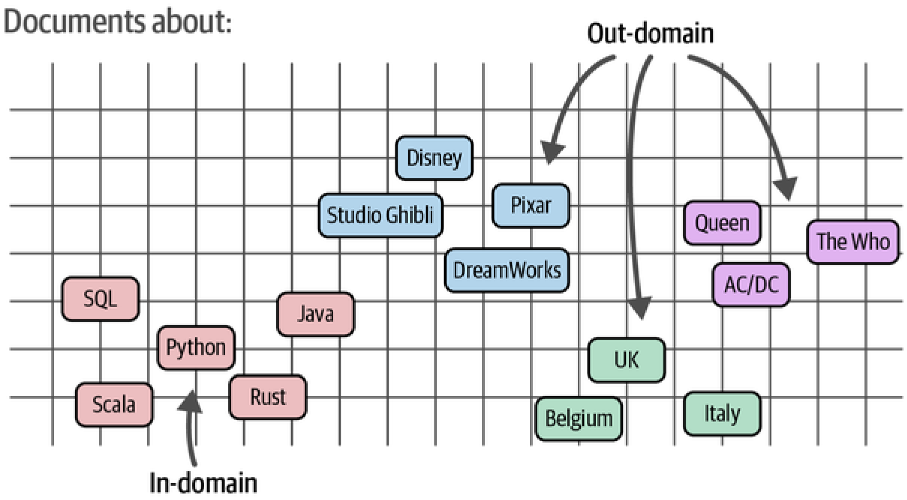

Figure 10-14. In domain adaptation, the aim is to create and generalize an embedding model from one domain to another.

One method for domain adaptation is called *adaptive pretraining*. You start by pretraining your domain-specific corpus using an unsupervised technique, such as the previously discussed TSDAE or masked language modeling. Then, as illustrated in Figure 10-15, you fine-tune that model using a training dataset that can be either outside or in your target domain. Although data from the target domain is preferred, out-domain data also works since we started with unsupervised training on the target domain.

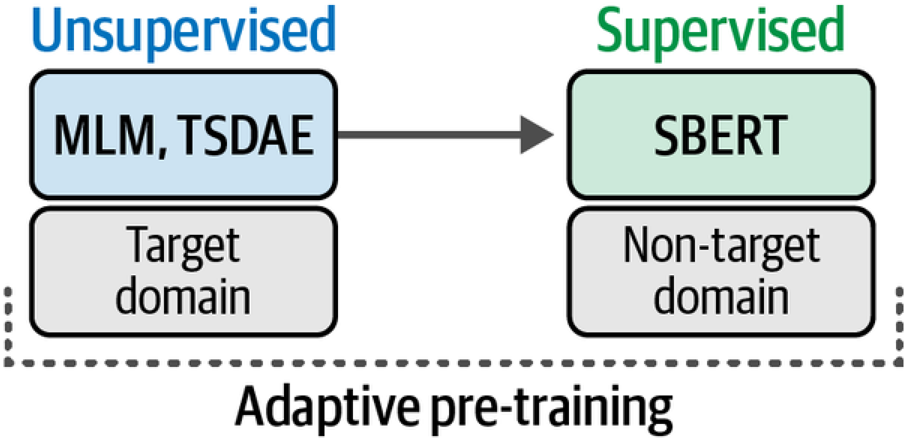

Figure 10-15. Domain adaptation can be performed with adaptive pretraining and adaptive fine-tuning.

Using everything you have learned in this chapter, you should be able to reproduce this pipeline! First, you can start with TSDAE to train an embedding model on your target domain and then fine-tune it using either general supervised training or Augmented SBERT.

Summary

In this chapter, we looked at creating and fine-tuning embedding models through various tasks. We discussed the concept of embeddings and their role in representing textual data in a numerical format. We then explored the foundational technique of many embedding models, namely contrastive learning, which learns primarily from (dis)similar pairs of documents.

Using a popular embedding framework, `sentence-transformers`, we then created embedding models using a pretrained BERT model while exploring different loss functions, such as cosine similarity loss and MNR loss. We discussed how the collection of (dis)similar pairs or triples of documents is vital to the performance of the resulting model.

In the sections that followed, we explored techniques for fine-tuning embedding models. Both supervised and unsupervised techniques were discussed such as Augmented SBERT and TSDAE for domain adaptation. Compared to creating an embedding model, fine-tuning generally needs less data and is a great way to adapt existing embedding models to your domain.

In the next chapter, methods for fine-tuning representations for classification will be discussed. Both BERT models and embedding models will make an appearance as well as a wide range of fine-tuning techniques.

[1] Alan Garfinkel. *Forms of Explanation: Rethinking the Questions in Social Theory*. Yale University Press (1982).

[2] Tim Miller. "Contrastive explanation: A structural-model approach." *The Knowledge Engineering Review* 36 (2021): e14.

[3] Nils Reimers and Iryna Gurevych. "Sentence-BERT: Sentence embeddings using Siamese BERT-networks." *arXiv preprint arXiv:1908.10084* (2019).

[4] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. "GloVe: Global vectors for word representation." *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 2014.

[5] Niklas Muennighoff et al. "MTEB: Massive Text Embedding Benchmark." *arXiv preprint arXiv:2210.07316* (2022).

[6] Matthew Henderson et al. "Efficient natural language response suggestion for smart reply." *arXiv preprint arXiv:1705.00652* (2017).

[7] Aaron van den Oord, Yazhe Li, and Oriol Vinyals. "Representation learning with contrastive predictive coding." *arXiv preprint arXiv:1807.03748* (2018).

[8] Ting Chen et al. "A simple framework for contrastive learning of visual representations." *International Conference on Machine Learning*. PMLR, 2020.

[9] Nandan Thakur et al. "Augmented SBERT: Data augmentation method for improving bi-encoders for pairwise sentence scoring tasks." *arXiv preprint arXiv:2010.08240* (2020).

[10] Tianyu Gao, Xingcheng Yao, and Danqi Chen. "SimCSE: Simple contrastive learning of sentence embeddings." *arXiv preprint arXiv:2104.08821* (2021).

[11] Fredrik Carlsson et al. "Semantic re-tuning with Contrastive Tension." *International Conference on Learning Representations, 2021*. 2021.

[12] Kexin Wang, Nils Reimers, and Iryna Gurevych. "TSDAE: Using Transformer-based Sequential Denoising Auto-Encoder for unsupervised sentence embedding learning." *arXiv preprint arXiv:2104.06979* (2021).

[13] Kexin Wang et al. "GPL: Generative Pseudo Labeling for unsupervised domain adaptation of dense retrieval." *arXiv preprint arXiv:2112.07577* (2021).