Natural Language Processing

Zhongjian Lin
University of Georgia

November 13, 2024

Bag-of-Words

- Examples clearly show that word order matters: manual engineering of order-based features, such as bigrams, yields a nice accuracy boost.
- The history of deep learning is that of a move away from manual feature engineering, toward letting models learn their own features from exposure to data alone.
- What if, instead of manually crafting order-based features, we exposed the model to raw word sequences and let it figure out such features on its own? This is what *sequence models* are about.
- The vectorization of strings has large sparse matrices and the training is very slow. One better way is *word embeddings*.

Word embeddings

One-hot encoding makes a feature-engineering decision. You're injecting into your model a fundamental assumption about the structure of your feature space. That assumption is that the different tokens you're encoding are all independent from each other: indeed, one-hot vectors are all orthogonal to one another. And in the case of words, that assumption is clearly wrong. Words form a structured space: they share information with each other. The words "movie" and "film" are interchangeable in most sentences, so the vector that represents "movie" should not be orthogonal to the vector that represents "film"—they should be the same vector, or close enough.

- The geometric relationship between two word vectors should reflect the semantic relationship between these words.
- For instance, in a reasonable word vector space, you would expect synonyms to be embedded into similar word vectors, and in general, you would expect the geometric distance (such as the cosine distance or L2 distance) between any two word vectors to relate to the "semantic distance" between the associated words. Words that mean different things should lie far away from each other, whereas related words should be closer.
- *Word embeddings* are vector representations of words that achieve exactly this: they map human language into a structured geometric space.

Word embeddings

The vectors obtained through one-hot encoding are binary, sparse (mostly made of zeros), and very high-dimensional (the same dimensionality as the number of words in the vocabulary), word embeddings are low-dimensional floating-point vectors (that is, dense vectors, as opposed to sparse vectors). It's common to see word embeddings that are 256-dimensional, 512-dimensional, or 1,024-dimensional when dealing with very large vocabularies. On the other hand, one-hot encoding words generally leads to vectors that are 20,000-dimensional or greater

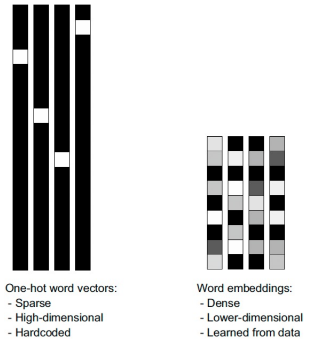


Figure 11.2 Word representations obtained from one-hot encoding or hashing are sparse, high-dimensional, and hardcoded. Word embeddings are dense, relatively low-dimensional, and learned from data.

Word embeddings

Besides being dense representations, word embeddings are also structured representations, and their structure is learned from data. Similar words get embedded in close locations, and further, specific directions in the embedding space are meaningful. To make this clearer, let's look at a concrete example.

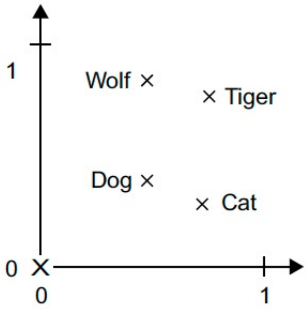


Figure 11.3 A toy example of a word-embedding space

We have "from pet to wild animal" vector and "from canine to feline" vector.

Word embeddings

In real-world word-embedding spaces, common examples of meaningful geometric transformations are "gender" vectors and "plural" vectors. For instance, by adding a "female" vector to the vector "king," we obtain the vector "queen." By adding a "plural" vector, we obtain "kings." Word-embedding spaces typically feature thousands of such interpretable and potentially useful vectors. There are two ways to obtain word embeddings:

- Learn word embeddings jointly with the main task you care about (such as document classification or sentiment prediction). In this setup, you start with random word vectors and then learn word vectors in the same way you learn the weights of a neural network.
- Load into your model word embeddings that were precomputed using a different machine learning task than the one you're trying to solve. These are called *pretrained word embeddings*.

Embedding Layers

- What makes a good word-embedding space depends heavily on your task: the perfect word-embedding space for an English-language movie-review sentiment-analysis model may look different from the perfect embedding space for an English-language legal-document classification model, because the importance of certain semantic relationships varies from task to task.
- It's thus reasonable to learn a new embedding space with every new task. Fortunately, backpropagation makes this easy, and *Keras* makes it even easier. It's about learning the weights of a layer: the Embedding layer.
- The Embedding layer is best understood as a dictionary that maps integer indices (which stand for specific words) to dense vectors. It takes integers as input, looks up these integers in an internal dictionary, and returns the associated vectors.

$$\text{Word Index} \rightarrow \text{Embedding layer} \rightarrow \text{Corresponding word vector}$$

Illustration

```
inputs = keras.Input(shape=(None,), dtype="int64")
embedded = tf.one_hot(inputs, depth=max_tokens)
x = layers.Bidirectional(layers.LSTM(32))(embedded)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
model.compile(optimizer="rmsprop",
  loss="binary_crossentropy",
  metrics=["accuracy"])
model.summary()
callbacks = [
  keras.callbacks.ModelCheckpoint("one_hot_bidir_lstm.tf",save_best_only=True)
]
model.fit(int_train_ds, validation_data=int_val_ds, epochs=10,
  callbacks=callbacks)
model = keras.models.load_model("one_hot_bidir_lstm.tf")
print(f"Test acc: {model.evaluate(int_test_ds)[1]:.3f}")
--------------------------------------------------------------------------
inputs = keras.Input(shape=(None,), dtype="int64")
embedded = layers.Embedding(input_dim=max_tokens, output_dim=256)(inputs)
x = layers.Bidirectional(layers.LSTM(32))(embedded)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
model.compile(optimizer="rmsprop",
  loss="binary_crossentropy",
  metrics=["accuracy"])
model.summary()
callbacks = [
  keras.callbacks.ModelCheckpoint("embeddings_bidir_gru.tf",save_best_only=True)
]
model.fit(int_train_ds, validation_data=int_val_ds, epochs=10,callbacks=callbacks)
model = keras.models.load_model("embeddings_bidir_gru.tf")
print(f"Test acc: {model.evaluate(int_test_ds)[1]:.3f}")
```

The Transformer architecture

- Starting in 2017, a new model architecture started overtaking recurrent neural networks across most natural language processing tasks: the Transformer.
- Transformers were introduced in the seminal paper "Attention is all you need" by Vaswani et al. The gist of the paper is right there in the title: as it turned out, a simple mechanism called "neural attention" could be used to build powerful sequence models that didn't feature any recurrent layers or convolution layers.
- This finding unleashed nothing short of a revolution in natural language processing— and beyond. Neural attention has fast become one of the most influential ideas in deep learning.
- The Transformer is effective for sequence data. We'll then leverage self-attention to create a Transformer encoder, one of the basic components of the Transformer architecture, and we'll apply it to the IMDB movie review classification task.

Self-Attention

As you're going through this book, you may be skimming some parts and attentively reading others, depending on what your goals or interests are. What if your models did the same? It's a simple yet powerful idea: not all input information seen by a model is equally important to the task at hand, so models should "pay more attention" to some features and "pay less attention" to other features.

- Max pooling in convnets looks at a pool of features in a spatial region and selects just one feature to keep. That's an "all or nothing" form of attention: keep the most important feature and discard the rest.
- TF-IDF normalization assigns importance scores to tokens based on how much information different tokens are likely to carry. Important tokens get boosted while irrelevant tokens get faded out. That's a continuous form of attention.

Self-Attention

There are many different forms of attention you could imagine, but they all start by computing importance scores for a set of features, with higher scores for more relevant features and lower scores for less relevant ones (see figure 11.5). How these scores should be computed, and what you should do with them, will vary from approach to approach.

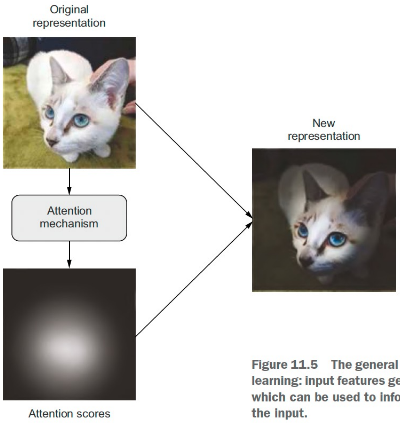
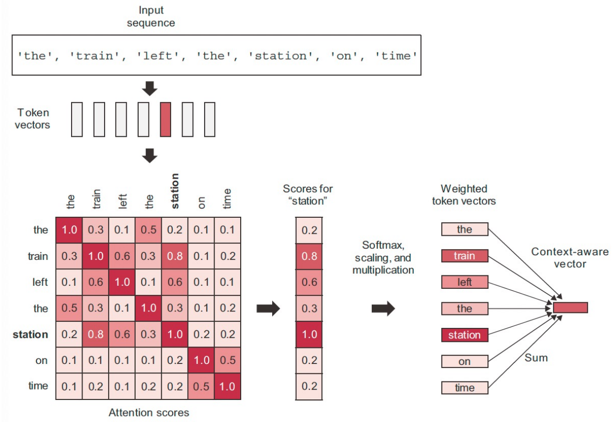


Figure 11.5 The general concept of "attention" in deep learning: input features get assigned "attention scores," which can be used to inform the next representation of the input.

Context-Specific Self-Attention

Clearly, a smart embedding space would provide a different vector representation for a word depending on the other words surrounding it. That's where *self-attention* comes in. The purpose of self-attention is to modulate the representation of a token by using the representations of related tokens in the sequence. This produces context-aware token representations.



	the	train	left	the	**station**	on	time
the	1.0	0.3	0.1	0.5	0.2	0.1	0.1
train	0.3	1.0	0.6	0.3	0.8	0.1	0.2
left	0.1	0.6	1.0	0.1	0.6	0.1	0.1
the	0.5	0.3	0.1	1.0	0.3	0.1	0.2
station	0.2	0.8	0.6	0.3	1.0	0.2	0.2
on	0.1	0.1	0.1	0.1	0.2	1.0	0.5
time	0.1	0.2	0.1	0.2	0.2	0.5	1.0

Scores for "station"
0.2
0.8
0.6
0.3
1.0
0.2
0.2

Weighted token vectors
the
train
left
the
station
on
time

Figure 11.6 Self-attention: attention scores are computed between "station" and every other word in the sequence, and they are then used to weight a sum of word vectors that becomes the new "station" vector.

The Transformer encoder

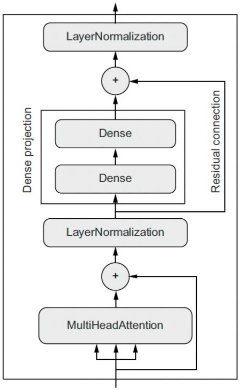


Figure 11.9 The `TransformerEncoder` chains a `MultiHeadAttention` layer with a dense projection and adds normalization as well as residual connections.

Factoring outputs into multiple independent spaces, adding residual connections, adding normalization layers—all of these are standard architecture patterns that one would be wise to leverage in any complex model. Together, these bells and whistles form the Transformer encoder—one of two critical parts that make up the Transformer architecture

When to use sequence models over bag-of-words models

You may sometimes hear that bag-of-words methods are outdated, and that Transformerbased sequence models are the way to go, no matter what task or dataset you're looking at. This is definitely not the case: a small stack of Dense layers on top of a bag-ofbigrams remains a perfectly valid and relevant approach in many cases. In fact, among the various techniques that we've tried on the IMDB dataset throughout this chapter, the best performing so far was the bag-of-bigrams!
It turns out that when approaching a new text-classification task, you should pay close attention to the ratio between the number of samples in your training data and the mean number of words per sample. In other words, sequence models work best when lots of training data is available and when each sample is relatively short.

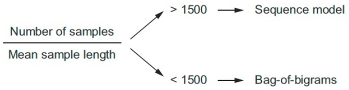


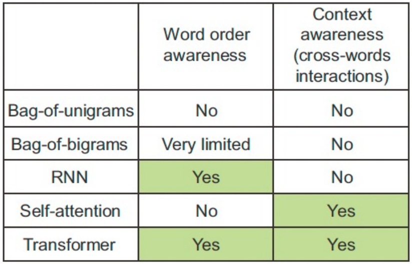
Figure 11.11 A simple heuristic for selecting a text-classification model: the ratio between the number of training samples and the mean number of words per sample

Sequence-to-sequence learning

Text Classification is an extremely popular use case, but there's a lot more to NLP than classification. For instance, sequence-to-sequence models. A sequence-to-sequence model takes a sequence as input (often a sentence or paragraph) and translates it into a different sequence. This is the task at the heart of many of the most successful applications of NLP:

1. *Machine translation*-Convert a paragraph in a source language to its equivalent in a target language.
2. *Text summarization*-Convert a long document to a shorter version that retains the most important information.
3. *Question answering*-Convert an input question into its answer.
4. *Chatbots*-Convert a dialogue prompt into a reply to this prompt, or convert the history of a conversation into the next reply in the conversation.
5. *Text generation*-Convert a text prompt into a paragraph that completes the prompt.

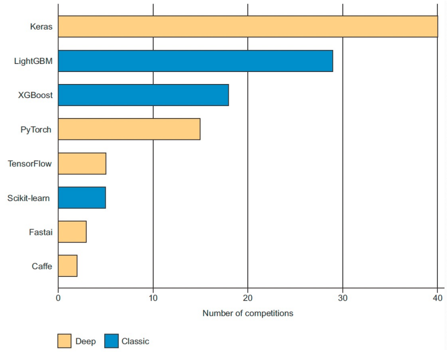
Natural Language Processing

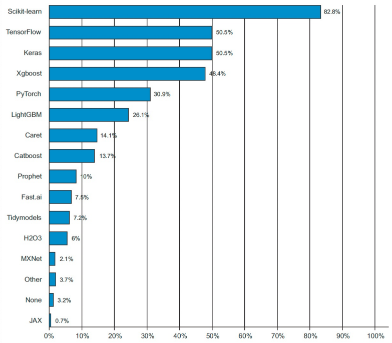


	Word order awareness	Context awareness (cross-words interactions)
Bag-of-unigrams	No	No
Bag-of-bigrams	Very limited	No
RNN	Yes	No
Self-attention	No	Yes
Transformer	Yes	Yes

Figure 11.10 Features of different types of NLP models

Trends and Framework


Keras
LightGBM
XGBoost
PyTorch
TensorFlow
Scikit-learn
Fastai
Caffe
Number of competitions
Deep
Classic


Scikit-learn 82.8%
TensorFlow 50.5%
Keras 50.5%
Xgboost 48.4%
PyTorch 30.9%
LightGBM 26.1%
Caret 14.1%
Catboost 13.7%
Prophet 10%
Fast.ai 7.5%
Tidymodels 7.2%
H2O3 6%
MXNet 2.1%
Other 3.7%
None 3.2%
JAX 0.7%