CS 182/282A Deep Neural Networks **April 7th th, Spring 2023**

Lecture 21: BERT & Fine-tuning

Lecturer: Anant Sahai *Scribes: Jens Amund Kristoffersen, Peter Zhang*

21.1 Tokenization

Before we get into BERT and fine-tuning, it is important to understand how tokenization is done. Tokenization is necessary because textual information is given as a sequence of strings from a discrete alphabet. However, for deep learning, we need a sequence of vectors.

We break this problem down into two steps

1. Parse the strings to a sequence of tokens
2. Map those tokens into vectors

A natural approach would be to forego step one and use the letters of the alphabet as tokens. However, a single letter is not a meaningful token. Consider the word "dog". The naive tokenization would represent it as follows:

$$dog \rightarrow \begin{bmatrix} \vec{d} \\ \vec{o} \\ \vec{g} \end{bmatrix}$$

Every letter occupies separate dimensions, but together they represent the concept. Thus, our model has to learn how these dimensions together represent the word.

If we could have each token represent a meaningful unit instead, we effectively do this work for the model. Since each meaningful unit of language is of variable length, we need a way to map a variable-length string to a fixed-length token. We do that via a lookup table, where each of the m possible inputs is mapped to an index.

Variable length input	Token index
s_0	0
s_1	1
.	.
.	.
.	.
s_{m-1}	m-1

Table 21.1: Mapping from input string to token index

The left column of this table must contain all possible input strings. In addition, each input must have an unambiguous match in the left column. Consider the input **a**, and the two possible matches **a** and **an**. Which should we choose? One could break the tie through a predefined rule, for instance choosing the longest match. However, the simplest way is to ensure prefix freeness, i.e. that no string is the prefix of another.

Once we have the token as an index, we must map those tokens to vectors. This is done via another lookup table.

Token index	$\mathbb{R}^d$
0	$\vec{v}_0$
1	$\vec{v}_1$
.	.
.	.
.	.
m-1	$\vec{v}_{m-1}$

Table 21.2: Mapping from token index to vector

Since the right side of this table contains vectors of numbers, they can be learned through gradient descent. Thus, we initialize and update these vectors the same way we update our weights.

Note that the table is not a linear mapping, since the left column contains discrete objects, not vectors in a vector space. However, the lookup operation can be thought of as a matrix multiplication. To do so, each index i on the left side is represented as a one-hot-encoded vector with zero in all but the ith spot. We then multiply it by the matrix representing the right side of the table.

The mapping in table **??** cannot be learned through gradient descent. How, then, can we learn it from data? As it turns out, the problem is very analogous to compression. When doing lossless compression, we want to find a way to map variable-length sub-sequences of our data to fixed-length tokens in such a way that we minimize the size of the resulting sequence of tokens. To do this, frequent longer sub-sequences get their own token, such that they can be represented efficiently.

One way to figure out this mapping from a corpus of data is Byte pair encoding, which is related to the famous Huffman Coding algorithm [**huffman1952method**]. Byte pair encoding works as follows:

1. Choose how many tokens you want.
2. Place each alphabet symbol in its own bucket as a candidate token.
3. Estimate the frequency of each possible pair of tokens from a given corpus.
4. Create a new bucket with the most frequent pair. If all possible pairs of one of the original tokens now have their own bucket, remove the token.
5. Repeat step 3 and 4 until you have reached the target number of tokens.

Why do we use a compression algorithm to do our tokenization? The standard deep learning answer is that there is no good reason. A more intuitive explanation is that symbols that occur commonly together do so because they represent a semantically meaningful unit. Therefore, we should represent them by one vector to save our model from having to learn to associate the combination of many different vectors with one meaning.

It is important to note that tokenization is done *before* learning starts. In practice, off-the-shelf tokenizers like OpenAI's tiktoken [**tiktoken**] are used. As a result, design decisions like vocabulary and vector size are made by a third party. If you are operating under constraints that force you to control these variables, you have to build your own tokenizer. The full deep learning setup can be seen in figure **??**

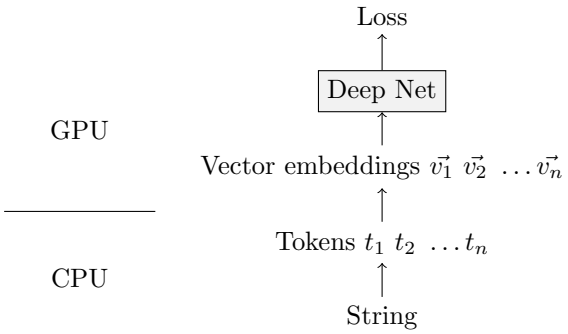

Figure 21.1: Full learning setup with tokenization of input

21.2 Word2vec

Except for the decision tree family of algorithms, every other machine-learning technique requires the input to be in the form of vectors. Thus, if we want to do machine learning, we need to do a token-to-vector mapping.

Ideally, the resulting vectors should reflect the semantic "meaning" of the words/tokens. To achieve this, we exploit the fact that words in a corpus always appear in the context of other words. If two words often appear in similar contexts, they should be close in semantic space. The clearest example of this is synonyms. Since they mean the same thing, they appear in very similar contexts. Thus, their vectors should be close to each other.

Now, to use this insight in practice, we need to represent it as a training strategy with a loss function such that we can perform gradient descent. We achieve this in the following way:

1. Randomly initialize two vectors $\vec{u}_i$ and $\vec{v}_i$ for each word.
2. Use $\vec{u}_o^T \vec{v}_c$ to measure a score for o as a likely neighbor for c.
3. Train using "logistic loss style" loss, i.e.

   $$log(\frac{1}{1 + exp(\pm score)})$$

   with randomly selected word c. We compare c with positive examples that are its neighbors and random negative examples. We need to train with both positive and negative examples in order for the vectors to not all be $\vec{0}$, which would minimize the inner product between all vectors.
4. Use the average of $\vec{u}_i$ and $\vec{v}_i$ as the final embedding.

Empirically, using two vectors $\vec{u}$ and $\vec{v}$ for each word achieves better results. The training setup tries to move likely neighbors into alignment with each other while forcing random words to be orthogonal to each other.

We pick the dimension of the embedding the same way we pick any other hyperparameter. You could do cross-validation on a downstream task using the embeddings or simply use loss on held-out data.

In contrast to words, vectors live in vector spaces. As a result, we can now use vector operations on them. For instance, we can add and subtract words from each other. Interestingly, these operations have human understandable effects. Consider the following equation.

$$woman - man + uncle$$

Here, we are adding the difference between a woman and a man to an uncle. Intuitively, the result should be an aunt. Amazingly, the following holds in the Word2Vec vector space:

$$Vec(woman) - Vec(man) + Vec(uncle) \approx Vec(aunt)$$

This is surprising since making analogies was never a part of the training objective. We have an emergent property, where abilities not optimized for directly emerge from our model.

21.3 BERT

BERT is an **encoder-style** transformer, which means self-attention is not causal [**devlin2018bert**]. Text input is first transformed into a vector (with learnable parameters) and concatenated to a positional embedding (without learnable parameters). The core model is composed of a stack of attention blocks with serial (or parallel) interconnections to NLP. At the end is a task-specific MLP that maps the embedding to scores for each token.

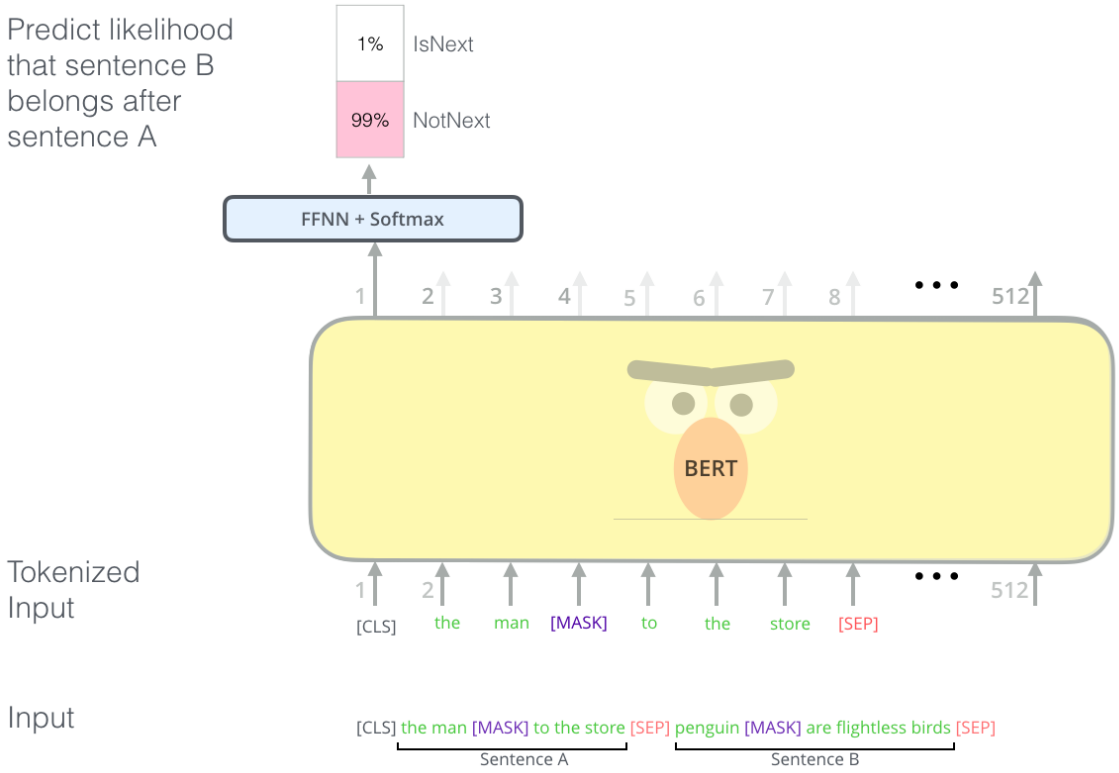

Figure 21.2: BERT Diagram

BERT is originally pre-trained on two tasks:

1. **Masked denoising**. Before words are tokenized, words are randomly masked (replaced with a `<MASK>` token) or swapped for a randomly selected word, say 10% of the time. For them ask, the goal is to predict the missing input; the motivation behind replacing the word is to force the encoder to look at the rest of the context. The goal is to reconstruct the original sentence which encourages learning the context. But masked denoising might only teach models to reason at the word level within a sentence.
2. **Sentence order**. Input pairs of sentences at a time and occasionally swap the sentences. Given a pair of sentence separated with a `<SEP>` character, predict whether the sentences are swapped. The prediction corresponds to the `<CLS>` token, which has a corresponding output that is fed into a task-specific MLP. The second task is designed to teach language models to learn the relationships between sentences, although there is controversey over whether this second task adds any value with larger models.

When using BERT on downstream tasks, there are two approaches:

1. **Feature extractors**. Treat the embeddings of the language model as a feature extractor and use any combination of the activations as an input to a separate model. While you could just use the penultimate layer, you could also use the last N layers and concatenate or average them. Selecting a subset of features becomes its own hyperparameter search.
2. **Fine-tuning**. Use BERT as a building block. Replace the task head (in this context, the head refers to the MLP fine-tuned to a specific task) with a new MLP. Fine-tuning can apply to just the final layer or to the entire model, but fine-tuning the entire model can actually degrade performance. This is because if the final layer's weights are poor, the backpropagated weight updates may also be noisy and suboptimal. The present best practice is to freeze the pre-trained model first, and then train everything.

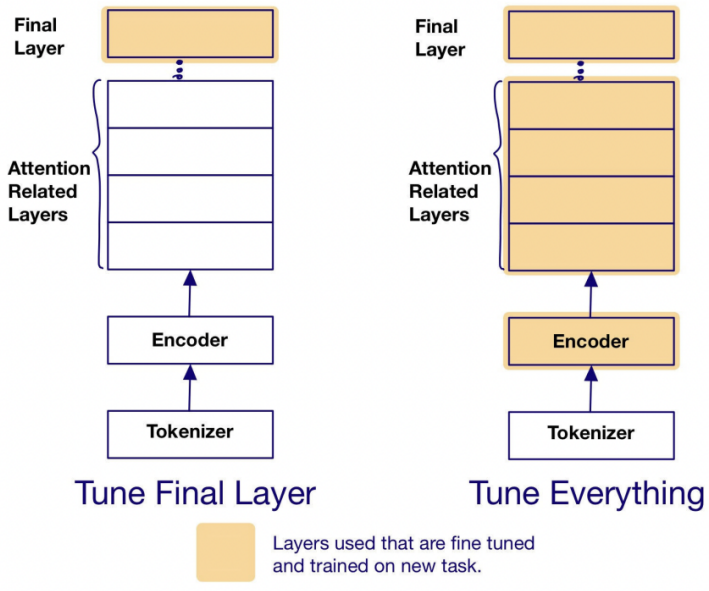

Figure 21.3: Fine-tuning the final layer versus everything