EECS 182/282A Lecture 20: Pretraining and Fine-Tuning

Instructor: Anant Sahai
Scribes: Keaton Elvins, Raghav Ramanujam

1 Tokens

1. Tokenizer: In the typical approach for sequence models, the input is first passed to a tokenizer. This parses the input sequence into a series of discrete tokens (i.e. Token-1, Token-72, Token-985...). Once this sequence of discrete objects is generated, they are then passed to a learnable look-up table.

   > **Aside:** For NLP, the byte pair encoding scheme is typically used for this step, which entails repeatedly grouping the most common occurring pair of characters together until the desired "token budget" is reached (similar to the grouping in Huffman encoding but in reverse). This process addresses the issue of encountering out-of-vocabulary words since they will just be broken down into common character sub-groups (read more `here`).

2. Look-Up Table: At this step, each token gets mapped to a vector, which becomes the actual input that the transformer/sequence model sucks up. These vectors are all learnable, so this effectively adds (# of tokens × size of input vector) parameters.

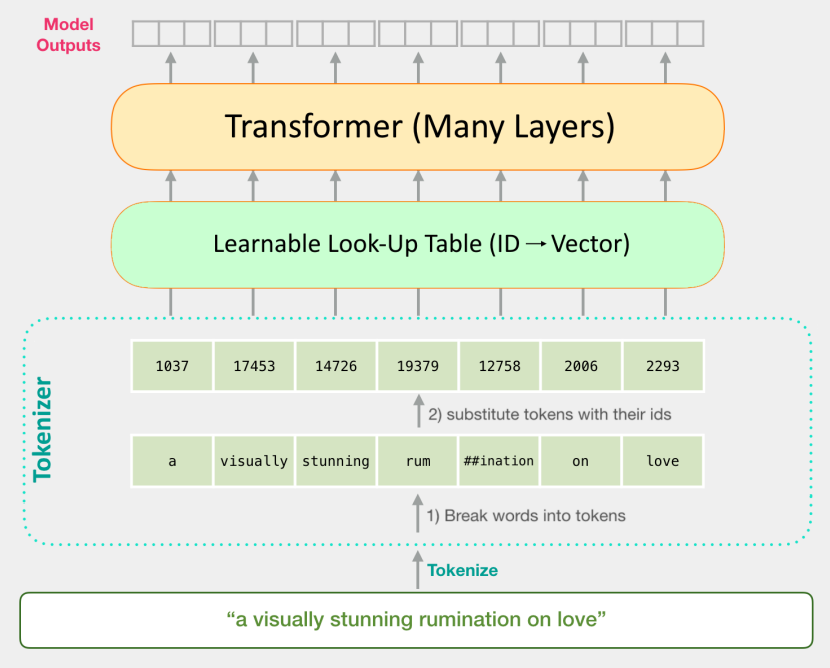


Figure 1: Example of input processing for transformer architecture [1]

2 Word2vec

2.1 Creating a tractable version of the problem

As we learned in the last lecture, the objective with word embeddings is to find vector representations of each word such that similar words $a, b \in V$ produce similar embeddings v_a, v_b. An initial approach could be to set up the optimization problem

$$\arg \max_{u_1,\cdots,u_n,v_1,\cdots,v_n} \sum_{c,o} \log p(o|c)$$

where

$$p(o|c) = \frac{\exp\left(u_o^\top v_c\right)}{\sum_{w \in V} \exp\left(u_w^\top v_c\right)}$$

The intuition for this approach stems from clustering: we want words that are roughly interchangeable in context to be placed near each other in the embedding. However, unlike K-means, our set of words is fixed, meaning we don't have to worry about fitting new words in (although we could always do this by recomputing as before).

One large problem with this approach is that when we attempt to run gradient descent, the denominator of the probability function is extremely costly to compute with a large vocabulary. To mitigate this, we can consider redefining the problem as something closer to binary classification. To do so, let's try a new probability function

$$p(\text{o is the right word}|c) = \sigma(u_o^\top v_c) = \frac{1}{1 + \exp\left(-u_o^\top v_c\right)}$$

The problem with this approach, however, is that it consists of only positive examples! Therefore, the optimization algorithm is incentivized to make all the embeddings line up to achieve really high probabilities. To address this, we can add

$$p(\text{w is the wrong word}|c) = \sigma(-u_w^\top v_c) = \frac{1}{1 + \exp\left(u_w^\top v_c\right)}$$

and optimize the function

$$\arg \max_{u_1,\cdots,u_n,v_1,\cdots,v_n} \sum_{c,o} \left(\log p(\text{o is right} \mid c) + \sum_w \log p(\text{w is wrong} \mid c) \right)$$

But this runs into the same problem as before! Now we're just summing over a huge number of negative examples, and the loss from these could just dominate. In order to balance this tug of war, we can instead just randomly sample a few words to be used for the "w is wrong" negative examples. This approach incorporates both an attractive force for words that occur together (push v_c towards u_o) and a repulsive force for those that do not (push v_c away from vectors u_w), and we are left with our Word2vec optimization problem

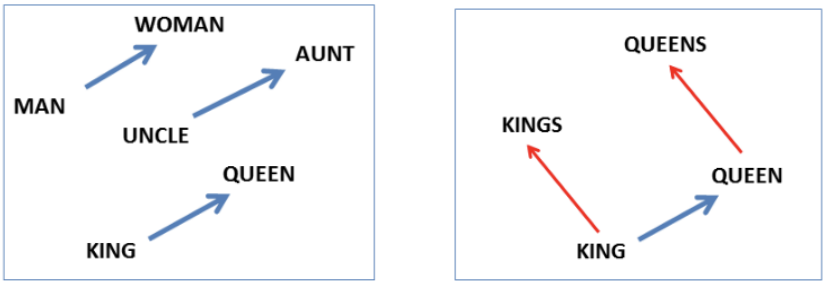
$$\arg \max_{u_1,\cdots,u_n,v_1,\cdots,v_n} \sum_{c,o} \left(\log \sigma(u_o^\top v_c) + \sum_w \log \sigma(-u_w^\top v_c) \right)$$

2.2 Interpreting the learned embeddings

After the development of Word2vec, researchers began to look at the actual embeddings to find if any interesting properties were present that might reflect the underlying structure of a language. After some exploration, they discovered that algebraic relations seemed to have some meaning with the embeddings. For example,

$$\text{vec}(\text{“woman”}) - \text{vec}(\text{“man”}) \simeq \text{vec}(\text{“aunt”}) - \text{vec}(\text{“uncle”})$$

$$\text{vec}(\text{“woman”}) - \text{vec}(\text{“man”}) \simeq \text{vec}(\text{“queen”}) - \text{vec}(\text{“king”})$$



(a) Gendered pairs have matching differences

(b) Plurality is captured similarly

Figure 2: Visualization of grammatical structure in the embedded space [2]

While these relationships are slightly idealized, the learned embeddings from Word2vec were able to capture a surprising amount of grammatical structure. Realizing this, researchers started to use the model to solve analogies (i.e. If women $\rightarrow$ man, aunt $\rightarrow$?). In practice, this returned a mix of nonsense responses and some actually interesting results.

Relationship	Example 1	Example 2	Example 3
France - Paris	Italy: Rome	Japan: Tokyo	Florida: Tallahassee
big - bigger	small: larger	cold: colder	quick: quicker
Miami - Florida	Baltimore: Maryland	Dallas: Texas	Kona: Hawaii
Einstein - scientist	Messi: midfielder	Mozart: violinist	Picasso: painter
Sarkozy - France	Berlusconi: Italy	Merkel: Germany	Koizumi: Japan
copper - Cu	zinc: Zn	gold: Au	uranium: plutonium
Berlusconi - Silvio	Sarkozy: Nicolas	Putin: Medvedev	Obama: Barack
Microsoft - Windows	Google: Android	IBM: Linux	Apple: iPhone
Microsoft - Ballmer	Google: Yahoo	IBM: McNealy	Apple: Jobs
Japan - sushi	Germany: bratwurst	France: tapas	USA: pizza

Figure 3: Examples of Word2vec's performance on more complex analogies [2]

Aside: Following this, concerns began to develop around whether all the stuff the embeddings were learning was desired (e.g. was it learning sexist/racist/homophobic ideas from the training data). To address this issue, researchers tried going through datasets to censor out nasty unwanted training points and applying data augmentation to make the embeddings invariant to these regularities. However, this is still a field of ongoing interest, as these approaches may not be enough.

3 Pretrained Language Models

3.1 Contextual representations

One big advantage of word embeddings over one-hot encodings is the incorporation of latent information about the language/word that might otherwise have to be learned by a downstream model. However, since the embeddings are constant in word2vec, the same word used in two different contexts will produce the same embedding.

Figure 4: Same word in two difference contexts [2]

This implies that we need some form of context-specific representation for our model. In order to address this problem, we can

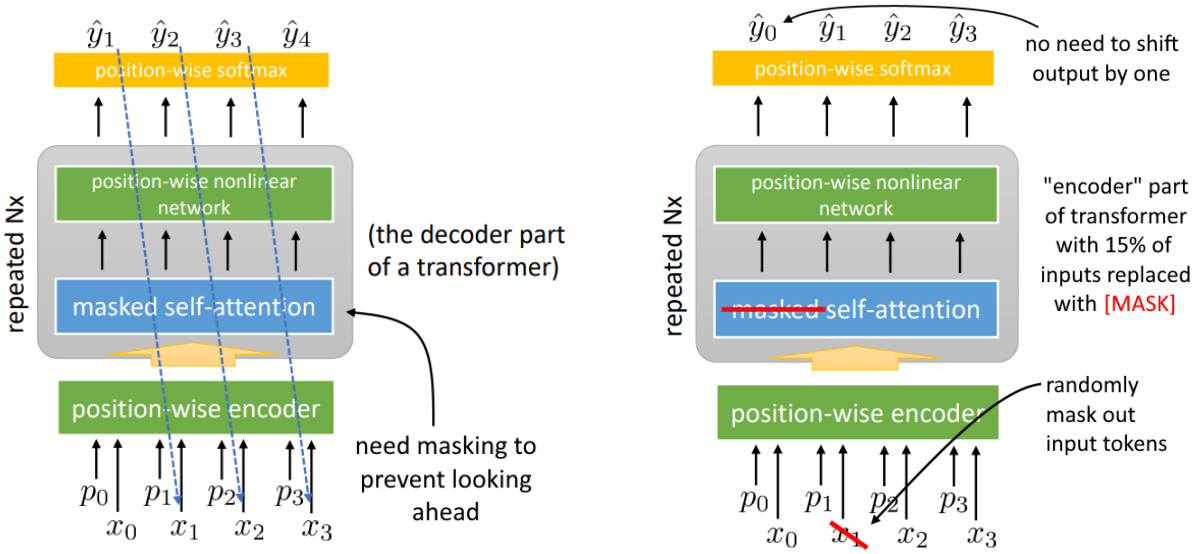
1. Train a language model on a surrogate task
2. Run it on a sentence
3. Take the hidden state from the model and treat it as the embedding

But this raises the complication: how do we train the best language model to get a high-quality embedding? What architecture should we use, and how does the surrogate task affect performance?

One approach is to train on the task of predicting the next word in a sentence while using a decoder-style transformer architecture (autoregressive idea used by GPT). However, since we don't want the attention mechanism to just be able to look ahead in the sentence and know exactly what to output, we have to implement **masked self-attention**. In practice, this is done by setting the relevant inner products to negative infinity, which reduces the contribution of their values to zero via the softmax.

Masked self-attention works for many tasks but introduces an interesting limitation: the model can only build context for a word by attending to the ones that came before it. In Word2vec, however, we were able to look at things on both sides of the center word to build context. How can we do something similar here?

3.2 Bidirectional Transformer Language Models



(a) One-directional transformer used in GPT

(b) Bidirectional transformer used in BERT

Figure 5: Differences between popular transformer-based language models [2]

We can modify our approach by changing the surrogate task! Instead of predicting the next word, we can mask out a small percentage of the input (replace with a [MASK] token) and train the model on predicting what those words would have been. This self-supervised task is very similar to what we did with masking in autoencoders, which we analyzed under the lens of low-rank approximation using PCA. Due to this change, we no longer need masked self-attention, and the model can build context for a given token using input that both precedes and follows it.

It is also important to distinguish that there are two losses we can use here: loss based on the predictions for the masked tokens, and loss for the rest of the sequence. It is critical to find a balance here, as letting the second option apply too much gradient pressure would result in the model just learning the identity function and giving up on the masked predictions. So in practice, the main goal is just to fill in the blanks, although BERT did modify this scheme slightly to find the right balance.

While BERT was training, not all of the 15% of tokens were actually replaced with the corresponding [MASK] token. While 80% of them still were, 10% were instead replaced with some random wrong word (similar to a denoising autoencoder), and the remaining 10% were just the correct word unchanged [3]. By leaving the token as is sometimes, we allow the second loss from above to apply a small amount of gradient pressure. This encourages the model to also take into account the current token, rather than being entirely context based, and strikes the appropriate balance between the two losses.

3.3 Training BERT

While training BERT (as seen in Figure 4a), the researchers took an additional step to try and force the model to learn sentence-level representations. They passed in two sentences at a time, separated by a [SEP] token, and randomly swapped the order of the sentences 50% of the time. They then added a surrogate binary classification task, denoted Next Sentence Prediction or NSP, and asked it to predict if the sentence order had been swapped or not (in addition to the previously stated task). In order to accomodate for this extra task, the researchers added a special [CLS] token at the start of each sentence pair, with its corresponding output from BERT being the output for the NSP task. This little tweak during pretraining turned out to be very beneficial for downstream tasks like question answering and natural language inference, as the model was forced to learn both context-dependent word-level and sentence-level representations.

Aside: Some people have tried using Word2vec as a starting point for the token-to-vector encoding for models like BERT, but it isn't an exact match since not all tokens are words. While they often are, these models have some token budget they must stay within, so sometimes complicated/rare words are broken up. Check out `https://beta.openai.com/tokenizer` for an example of how OpenAI's GPT family of models process text into tokens.

4 Fine-Tuning

4.1 Using BERT

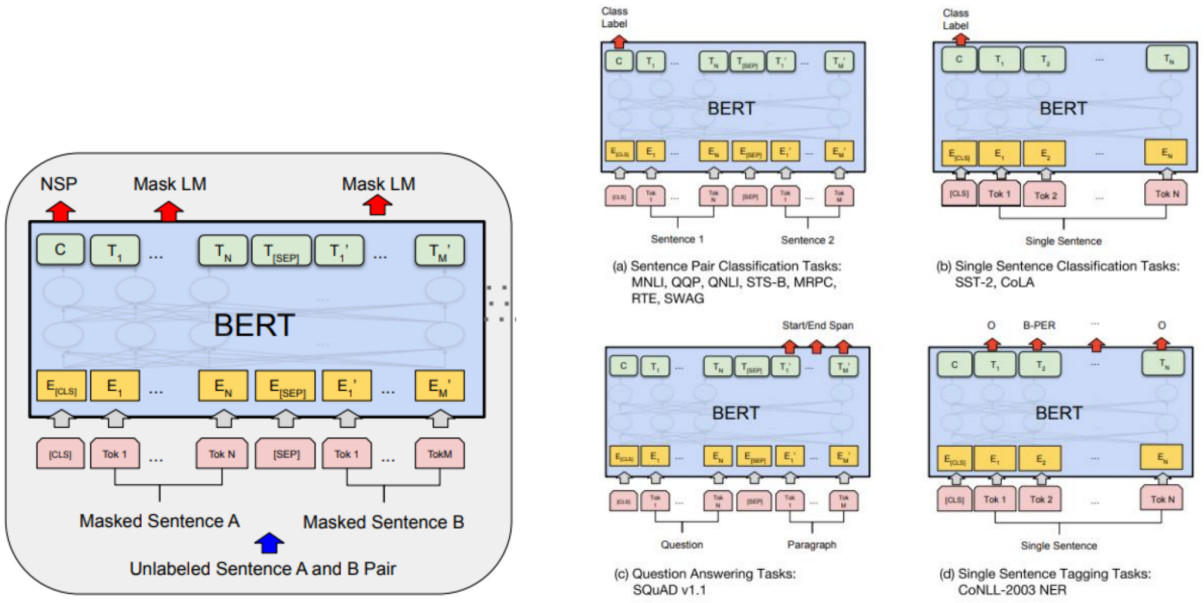
When it comes to actually trying to use BERT on some downstream task, we can think back to what we did with PCA: use an autoencoder approach, train the model, chop off the decoder part, and just use the embedding produced by the encoder on something new. The same works with BERT! One example would be entailment classification, or whether or not one sentence is logically a consequence of another. Since this is more related to the NSP task from training, we can cut off whatever classifier/linear-layer was used to make the NSP prediction, pass the embedded representation that BERT built to a new model, and train on our entailment task. However, now we have two options for this last fine-tuning step:

1. Freeze BERT and only train whatever classifier we've added at the end: This approach is similar to how we thought about the autoencoder in the PCA context. The entire encoding network is frozen and can be thought of as some input featurization for the final component to run inference on.

2. Train BERT end-to-end on the new task: Sometimes this approach of fine-tuning the entire model works better, but it also takes significantly more compute. In addition, the ratio of unlabeled to labeled data is often enormous, so trying to train such an over-parameterized network on a small set of labeled data may not be worthwhile.

4.2 Additional Tasks

Since the above procedure only really uses the first output from BERT, it is natural to wonder what uses the other outputs might have. It turns out that BERT is useful for many tasks, it simply depends on which components the user wants to use.



(a) BERT model during training

(b) BERT model during fine-tuning. Utilized outputs depend on use case

Figure 6: BERT model during training vs fine-tuning [2]

As seen in Figure 4b, we can build on top of the class label for classification tasks like above, or we can also select different outputs depending on the given task. For example, if we are trying to identify the span of contents in a paragraph that answer a given question, we can pass in the question followed by a separator token and the paragraph. We can then build on top of the outputs from the paragraph and fine-tune a model to choose the start/end of the span. The final example in the figure is entity-labeling, or identifying things like people's names, locations, and other categories.

4.3 Using BERT for feature generation

One question that may arise is how to use BERT to get features like we did with Word2vec. Since BERT has many layers, we have a choice of which hidden state to use. In practice, it is worth testing out the embedding from different layers, as well as the sum of different layers, to find the best possible contextualized representation. This idea is visualized below. More info can also be found at `http://jalammar.github.io/illustrated-bert/`.

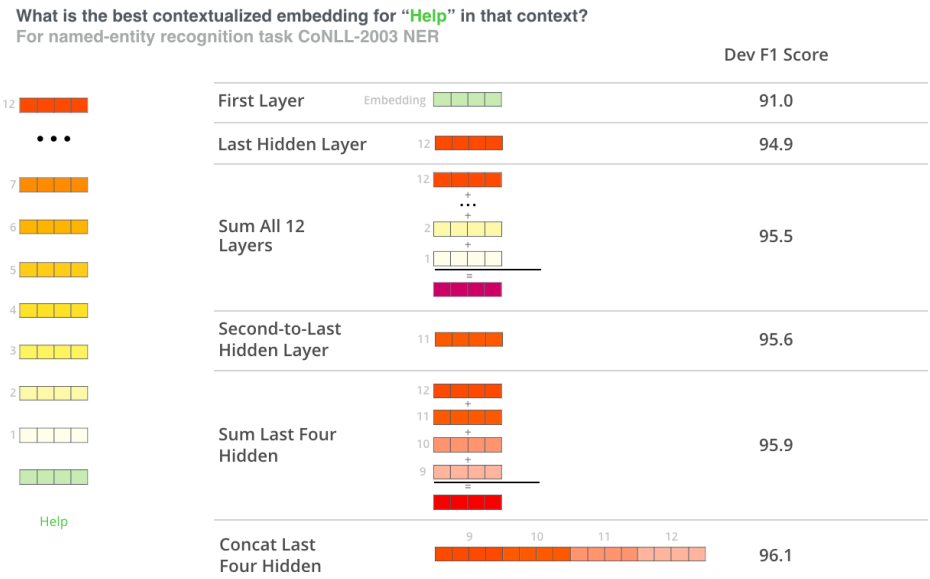


Figure 7: Representation scores of possible choices for embedding [4]

Aside: When exploring this, one may find the second-to-last layer works better than the last for feature generation. While the exact reason is up for debate, intuition for this could be that the last layer is more specific to the surrogate task while the second-to-last layer contains more general information.

5 Surprising Results With GPT

In the problem of text generation, we give the model some input for context and ask it to spit out a continuation (e.g. given the first paragraph, finish this article). Due to BERT being bidirectional and trained with context on both sides, it is not particularly suited or well-performing in this task. However, this problem is perfect for autoregressive models like the one-directional transformer architecture used in GPT (partially visualized in Figure 3a).

In fact, we can frame many tasks as text generation and see how GPT performs without any additional training (and therefore without performing further gradient descent). For the example of machine translation, one could input "The translation of 'she' to Spanish is 'ella'. The translation of 'ball' to Spanish is..." to GPT, and find that it would return the correct translation ('pelota') more times than just luck would suggest. The implications of such properties are still being figured out.

Food for thought: How would you frame the task of article summary to GPT?

References

[1] Alammar, J (2018). A Visual Guide to Using BERT for the First Time. https://jalammar.github.io/a-visual-guide-to-using-bert-for-the-first-time/

[2] Levine, Sergey. NLP Applications: CS 182 Lecture Slides. https://cs182sp21.github.io/static/slides/lec-13.pdf

[3] Devlin, Jacob and Chang, Ming-Wei and Lee, Kenton and Toutanova, Kristina. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. 2018. https://arxiv.org/abs/1810.04805

[4] Alammar, J (2018). The Illustrated Bert, ELMo, and co. [Blog post]. Retrieved from http://jalammar.github.io/illustrated-bert/

What we wish this lecture also had to make things clearer

The slides and the visuals were great, but we think the mechanics of byte pair encoding got somewhat brushed over. We would also have liked to see some visuals for BERT word embeddings similar to the ones we got for Word2vec, as well as a brief exploration of the main idea behind ELMo.