Language models, attention, and Transformer networks.

Introduction

The application of deep neural networks for NLP does not end with fully-connected networks. It barely starts with them. While you definitely can use them to process a simple text representation like CountVectors, the full potential of Deep Learning opens up when a model can learn complex features based on a raw sequence of tokens.

Bag-of-words approach is a good classical strategy, but it is generally applicable only to a handful of NLP tasks like classification and search. If you have a more complex task, like **part-of-speech tagging**, your output space is not just a single number (class index), but a sequence of numbers labeling each token what part of speech it is. If you just have a single fixed-size vector that describes your sentence “a quick brown fox jumps over the lazy dog”, it is not quite obvious how to transform this vector into a sequence “DET ADJ ADJ NOUN VERB PREP ADJ NOUN”. And if this vector does not have any information of the **word position**, this task is absolutely impossible, because it would not be able to distinguish between this sentence and a sentence “a dog jumps over the quick brown lazy fox”. They both contain the same words and if the representation does not account for the word position, they will have exactly the same vector, while the expected answer “DET NOUN VERB PREP DET ADJ ADJ ADJ NOUN” is quite different.

Representing texts as sequences of word vectors

Ideally, for this task we would like to have a **sequence of word vectors** instead of a single vector representing the whole text. A natural way of doing this would be to stack a bunch of word representations together into a matrix of size [seq_len, word_dim]. Next step is to insert each of these vectors, independently, into a linear layer or a neural network with input size word_dim and output size n_tags to classify it over different part-of-speech tags. What vectors can we use here?

Both one-hot or word2vec representations would work as a baseline. But what do they both lack? Consider a pair of sentences “we have argued for a salary increase” and “please increase the volume”. The word increase in the first sentence is a NOUN, but in the second it is a VERB. However, if we use a one-hot vector to represent it, would the vector be the same in both situations? Yes. The same applies to word2vec representations, which are static, context-independent, one vector per word in your vocabulary. If the vectors are the same and

we just feed them into a neural network independently of each other, there is no way to distinguish between “increase” as a VERB and as a NOUN.

A baseline for contextualized word representation

One very simple and straightforward way of solving this issue is what we call a context window representation. It is very similar to the approach that is used to train word2vec in a sense that we only consider a narrow context around the word. Let’s say just two words to the right and two words to the left. In this case we will have 5 vectors representing the word in its context. Let’s put them into a matrix of shape [5, word_dim] for simplicity. Now we need to somehow combine these into a single vector.

One way to do that is to average all vectors, like we did in word2vec. Is it a good idea here? Can it cause any kind of potential issues? If we just average the vectors for these 5 words we actually lose a lot of information. First of al, we lose positional information. But the second problem is even more frustrating. Such a vector does not indicate for which word we need to predict the part-of-speech tag. If you see a sum of vectors for words “volume+the+increase+please+music” and you need to predict a part-of-speech tag, what should it be? The part of the speech tag for the word “increase”? Or “volume”? Or “the”? There is no way of telling this. So avegating is not a good approach.

Instead, however, we can just concatenate these 5 vectors into a single vector of size 5 * word_dim. If we maintain the order of the vectors during concatenation and the vector of the first word is always placed to the indices 0 to word_dim-1, the second — word_dim to 2*word_dim-1, third — 2*word_dim to 3*word_dim-1 and so on the positional information is not lost. More than that, because we are always predicting the POS tag for the center word (which is the third word), the network understands that when it sees a vector of words [please, increase, the, music, volume] it needs to predict the POS tag for the word “the”.

Let’s look at what is happening inside our model. Assume we have these word vectors:

please = [7, 9, 1]
increase = [0, 1, 5]
the = [1, 1, 2]
mucis = [-1, 0, 5]
volume = [2, -1, -7]

After concatenation we will have a vector like this:

x = [7, 9, 1, 0, 1, 5, 1, 1, 2, -1, 0, 5, 2, -1, -7]

Assume we are using a linear model here and we have 4 classes “Determiner”, “Noun”, “Verb” and “other”. In this case, our matrix W will have the shape

[word_dim * (1 + context_size * 2), n_classes] = [3 * (1 + 2 * 2), 4] = [15, 4].

Let’s say Wij is the i-th j-th element of this matrix. In this case the weights W0,0, W1,0, … to W14,0 will be the weights we use to compute the score for the first class — “Determiner”. Let’s matrix-multiply x @ W and just look at the score computation for this class.

score = W0,0 * 7 + W1,0 * 9 + W2,0 * 1 + W3,0 * 0 + W4,0 * 1 + … + W13,0 * -1 + W14,0 * -7

Looking at this, we can say that in general, for all classes the weights W0,j to W2,j will correspond to the first word, W3,j to W5,j to the second and so on.

> In the case of a neural network, we can have multiple hidden layers in-between input and class scores. The very first layer has word-position specific weights, but the rest don’t. This is not a problem, the first layer does the job distinguishing between word positions and the rest just learn interactions between the features of these words.

Now, when we know how to process one word. We just apply the same technique to every word in the text. So, given context_size = 1 and a sentence “please increase the volume”, for the words “increase” we build a vector from words [“please”, “increase”, “the”]. For word “the” we build a vector from [“increase”, “the”, “volume”]. Now, what should we do with the words “please” and “volume” which do not have left and right contexts? One way to deal with that is to add special tokens which we call padding tokens and transform the sentence into “PAD please increase the volume PAD”. Now, nothing changes for “increase” and “the” and we can build vector representations for “please” as [“PAD”, “please”, “increase”] and for “volume” as [“the”, “volume”, “PAD”]. This PAD symbol should have its own vector. We can learn it or we can just say that its vector is all zeros.

This approach actually works and you can get reasonable results with it. But for some cases you might want to have more context. Maybe 3 words in each direction. Or 5 or 10 or 100. While in part-of-speech, you probably don’t regularly need this many words and there are other part-of-speech specific issues like out-of-vocabulary words, in many applications words can affect each other over large distances. For example, pronouns like “this” can relate to a word from a different sentence or in legal documents words like “tenant” can relate to a person who’s only mentioned in the beginning and the end of a multiple-page document.

Growing N

However, as N grows, word-window models become more and more difficult to handle. Even with a reasonable N of ~100, you will start having lots of issues.

First, your input layer will be extremely large, as your input vector grows linearly with N. Second, you will start having a lot of padding for the words on the edge of your text. Third, even if your text is of size 16, you still need to build a large padded vector of size 100 * word_dim. Fourth, because you need to predict a tag for every single word in your text, your total input tensor will have a potentially extremely large shape [text_len, N~200 * word_dim]. This is computationally inefficient, but this is just one side of the problem.

The second part is that your input layer parameters are per-word specific. Why is this a problem? Because your text lengths probably follow an exponential distribution, meaning there are a few very short texts, a lot of medium-sized texts and just a few long texts. It's quite probable that for a large N, a significant part of the input layer parameters will not be trained well, because they will mostly see padding.

And the list goes on. If you shift your text just by one word, it can have a completely different representation, while we might not necessarily need that. For example removing "brown" from "a quick brown fox jumps over the lazy dog" changes the weights that are used for all words after the second one, while meaning stays mostly the same.

Given all of these issues of this very simple method, how can we make a better architecture that can extract more flexible and more long-distance features while being computationally and parameter-wise more effective at scale?

Attention is all you need

Let's look at the problem one more time. We want to build a matrix of contextualized word vectors. When we have it, we're kind of done. If the vectors are good and account for position, semantics and context, we can just project them into labels space of POS-tags or, if we are doing classification, to average them and then project to a space of classes.

So really, we want the word representations to be able to be able to communicate. Maybe even multiple times, to catch more complex dependencies. How can we do that?

Let's assume that the word "needs" information from some other word if it is *in any way* related to it. So far it makes sense, but how do we detect this relation? Better question, how do we make our network able to learn relations?

Let's start with our usual word embeddings. They even don't have to be word2vec, we can start from random ones and then backpropagate right into word embeddings to learn task-specific representations. Let's say we have a matrix of word embeddings for our fox sentence [w_a, w_quick, w_brown, w_fox, w_jumps, w_over, w_the, w_lazy, w_dog].

Now, let's say we want to find the relations of the word fox in this sentence. This is a search problem in a way. We use the word fox to query the rest of the words. To start the search we

need to form a query from the embedding of this word. Because fully-connected neural networks can learn any function, we can use one here.

q_fox = Query(w_fox) = FCN(w_fox)

Next step is to use this query to find how much each word is related to the query. One way of doing this is another neural network that accepts a concatenated vector of the query and the other word

relation_score_with_a = FCN2([q_fox:w_a])
relation_score_with_quick FCN2([q_fox:w_quick])
…

There is, however, a simpler approach. Because we essentially look for similarity, we can use some similarity score instead. Let's do dot-product. If it's large, two vectors are similar, if it's small — the vectors are very different. Although this is not as flexible as a separate neural network, because we already have a complex transformation of Query(w_fox) we can expect it to give us a good search vector.

relation_score_with_a = q_fox @ w_a.T
relation_score_with_quick q_fox @ w_quick.T

After applying this operation to all of the words (let's leave fox here for a general case when there is no relation to other words) we get a vector of scores with shape [seq_len=9].

x = [w_a, …, w_dog] # [seq_len, word_dim]
s = q_fox @ x.T # [seq_len]

Now, we have choices. One way would be just to select the most-scored relation.

relation_id = argmax(s)

However, what if the word relates to multiple words in the same way. Like "fox" relating to both "quick" and "brown". We need to allow our model to work with such relations. One way of doing this is to replace the argmax operation which we can call "hard" maximum, because if finds a single word with a softmax operation, which we call "soft maximum" which gives us a list of numbers which are all more than 0 and sum to 1.

p = softmax(s) = exp(s) / sum(exp(s))

Let's say our relation scores of word fox with the rest of the words are

[w_a, w_quick, w_brown, w_fox, w_jumps, w_over, w_a, w_lazy, w_dog].
[4, 6, 6.1, 4, 6, 1, -3, -4, -3]

Argmax returns us just the index 2 for the word brown, because its relation score is the highest. Equivalently, we can say it returns us this vector

[0, 0, 1, 0, 0, 0, 0, 0, 0]

Compared to that, softmax would return us something like this

p = [0.05, 0.36, 0.40, 0.06, 0.13, 0.00, 0.00, 0.00, 0.00]

Now we have normalized scores (probabilities) of relation between the word fox and the rest of the words and we can use them to introduce more context to the embedding of the word fox. The property that the probabilities sum up to one gives us a nice interpretation that if the distribution has a very distinctive peak, only this word will affect the context of our word fox. But the distribution is more smooth, many words can contribute, but with the total weight of a single word that gets distributed between them. These probabilities are commonly referred to as “attention probabilities”

To do that, we can sum up all of the words with the weights equal to the softmax probabilities giving us something like this

context_fox = p @ x.T = 0.05 * w_a + 0.36 * w_quick + 0.40 * w_brown + 0.06 * w_fox + 0.13 * w_jumps + 0 * w_over + 0 * w_the + 0 * w_lazy + 0 * w_dog

To summarize, our contextualizing operation on a sequence of word embeddings x

1. Transforms the word “fox” into a query
2. Queries all of the words using dot-product between query_fox and x and gets relation scores
3. Applies softmax to the relation scores to get attention probabilities
4. Sums up word vectors with the weights equal to the probabilities

Now, let’s see if we can make this function even more flexible.

When we perform a context extraction operation on step 4, we might want our context vectors to differ from the original word embedding vectors. Why? Direct answer would be “it works better in practice”. A hand-wavy explanation would be “the generic meaning of the word without the context is slightly different from the meaning that this word brings to the sentence.

This modification replaces context_fox with the following two equations

v = FCN(x)
context_fox = p @ v

Now, let's simplify everything a bit. Let's use a single liner layer instead of every fully-connected network we had. It limits the expressiveness of the operation, but we will gain it back soon.

```
q = w_fox @ W_q  # [1, word_dim]
v = x @ W_v  # [seq_len, word_dim]

s = q @ x.T  # [1, seq_len]
p = softmax(s)  # [1, seq_len]
context_fox = p @ v  # [seq_len, word_dim]
```

This operation is applied to every single word in a text, not just to the word "fox". And because all of these operations are essentially just matrix multiplications, we can do all of them at the same time.

```
Q = x @ W_q  # [seq_len, word_dim]
V = x @ W_v  # [seq_len, word_dim]

S = Q @ x.T  # [seq_len, seq_len]
P = softmax(S, dim=1)  # [seq_len, seq_len]
context_x = P @ V  # [seq_len, word_dim]
```

The very last step is to additionally transform x into what we call "key" vectors that the query will use instead of x at the step 3. The reason for that is again, it works slightly better in practice, but additionally it allows the size of query and key vectors to be different from the size of the input, because now they are both feeded to linear layers and these linear layers can transform them from word_dim to some hidden with hidden != word_dim. It will be important later.

```
Q = x @ W_q  # [seq_len, hidden]
V = x @ W_v  # [seq_len, hidden]
K = x @ W_k  # [seq_len, hidden]

S = Q @ K.T  # [seq_len, seq_len]
P = softmax(S, dim=1)  # [seq_len, seq_len]
context_x = P @ V  # [seq_len, hidden]
```

This operation we got is called "attention". Each vector attends to every vector in the sequence and it is the main contextualizing operation used in post-2018 NLP. You can actually write it in just a single line.

```
softmax(Q @ K.T) @ V
```

~~Notice that v = x @ W_v does not depend on the index i and we can move it outside the for loop. Loops are slow and we need to try to get rid of them as much as we can. So let's try to~~

~~move out some other operations as well. We can compute all of the queries outside the loop q = x @ W_q and then just index them q_i = q[i]. However, we don't even need to do that, because instead of q_i @ x.T we can also do matrix multiplication s = q @ x.T. Same goes for softmax operation, which is vectorized and can accept tensor of any shape, not just a single vector. The only thing you need to provide is to tell it over which~~

Why K, Q, V

> Why do we call these variables K, Q, and V? This is due to an analogy that is very commonly used to explain attention.
>
> Imagine you have some database. It is not even related to neural networks directly. Let's say this is some SQL database that stores values V over which you can search. Every database uses some hashing algorithm to produce keys K of these values — the search index. This is done, because the values in real-world databases might be very large, like images, long texts and so on. And every time you query your database with a set of queries Q, the query gets compared to the keys which is a very quick operation.

Now let's look at some examples of attention maps. Of course we initialize these matrices with random numbers and in the beginning they don't do anything. However, after training to some task. It might be a classification task, or a tagging task, or, more commonly these days, a language modeling task which we will cover at the end of the lecture. After training, we can see that this relatively simple mechanism which does not specify which relations we should extract, extracts useful and sometimes even interpretable relations.

https://arxiv.org/abs/1906.04341 (Figure 5)

What do we mean by useful? Attention-based networks have been keeping the state of the art on almost all of the NLP tasks for the last 3 years.

Multiple heads are better than one

Now, let's think about how we can make our attention operation even more flexible. In our example with a fox you can say that this word is related to quick and brown in one way, but it is also related to jumps and this relation is different. To improve the ability of attention to catch more diverse dependencies and to catch multiple dependencies we can introduce the concept of attention heads.

Multi-head attention is essentially just a number of attentions running in parallel. For example, for two heads, we compute

```
K1 = x @ W_k1  # [seq_len, hidden / 2]
Q1 = x @ W_q1
```

```
V1 = x @ W_v1

K2 = x @ W_k2  # [seq_len, hidden / 2]
Q2 = x @ W_q2
V2 = x @ W_v2

h1 = softmax(Q1 @ K1.T) @ V1.T  # the output of the first head
h2 = softmax(Q2 @ K2.T) @ V2.T  # the output of the second head

h = [h1:h2] @ H  # concatenate and mix them together
```

Note that x, the input to attention operation, is the same for all of them, but the weights W_k1, W_k2, W_q1, W_q2, W_v1, W_v2 are different, which allows to attend to different parts of the same sequence.

The final operation of concatenating different head outputs and applying the matrix H to them is performed to achieve the desired dimensionality and to “mix” the information from the different heads into a united representation.

Multiple heads improve attention in two ways:

1. It expands the model’s ability to focus on different positions. It is very common to see the attending word itself dominate in the attention distribution. Having multiple heads mitigates this issue.
2. It gives the attention layer multiple “representation subspaces”. Each of W_k, W_q, W_v matrices are randomly initialized. Then, after training, each set is used to project the input embeddings into a different representation subspace.

In practice, this approach actually can detect different dependencies and some heads sometimes specialize on particular dependency types. For example one head can do a very simple operation of just looking at the previous word, a different one can detect words that word pronouns like “it” relate to.

Stack more layers

Attention operation is a very powerful mechanism that allows us to construct contextualized word representations. However, a single attention operation performed on uncontextualized embeddings which we have in the beginning cannot extract multi-hop relations. Meaning, all relations that we extract are direct. These relations happen only between a word and another word (or a small group of words), but not between different groups of words (no many-to-many) relations. An illustrative example would be

Ban on nude dancing on Governor’s desk

If you only consider the relations between “ban” and “on” or “on” and “desk” you might come to only one of the meanings of this sentence. However, it’s quite likely that this meaning (dancing on desk) is not the one we are looking for. We would want to extract the second-order relation between “ban-on” and “on-desk” which allows us to understand that “ban” might be a NOUN and this ban, as a document, is on the Governor’s desk.

The way we approach these second- and third- and n-th order relations is having multiple layers of attention stack on top of each other.

```
h_1 = Attention(x)
h_2 = Attention(h1)
...
h_n = Attention (h_n-1)
```

Notice that shape-wise we can easily do this, because the output of the attention layer has almost the same shape as it’s input.

```
x                       # [seq_len, word_dim]
h_1 = Attention(x)      # [seq_len, hidden_1]
h_2 = Attention(h1)     # [seq_len, hidden_2]
...
h_n = Attention (h_n-1) # [seq_len, hidden_3]
```

This approach of multiple attention layers works incredibly in practice. However, in order for everything to work, we need to make attention mindful of the word positions.

Accounting for word positions

More weird examples! Let’s take a sentence

Can a canner can a can?

And apply attention operation to it. Notice that all of the words “can” originally have exactly the same embedding, because it is the same word. After transforming it with Key, Query, and Value layers

```
k = w_can @ W_k
q = w_can @ W_q
v = w_can @ W_v
```

We still get exactly the same kqv vectors for all of the words “can”. This means that the attention operation cannot distinguish between them, thus attention cannot detect that one of these is a

verb meaning the ability to do something, the other one is the verb meaning to open a can and the other one is, well, a can.

This is happening, because attention is position-invariant. If we would insert a sequence of words

Can can can a a canner?

It would produce exactly the same vectors for every word as it would in the previous example.

A common way to avoid this problem is modify every word embedding according to its position. For example, we can have a special set of vectors that does not correspond to words, but instead it corresponds to positions. Next, we would always add the first position embedding to the first word in a sentence, the next one to the second word and so on.

w_can + p_0, w_a + p_1, w_canner + p2, w_can + p_3, a + p_4, can + p_5

Now, the first word “can” has the embedding that is a sum of the embedding of the word “can” and the embedding “p_0” which means the first position. And it is different from the embedding of the second word “can" which is equal to w_can + p_3.

How do we get these embeddings? We just initialize them randomly, and then update them with gradients computed via backpropagation using our loss function for the task the network is solving. Commonly this approach is called “positional embedding”.

The very last trick is extremely simple. In attention computation we divide the attention scores with a square root of the hidden size. It helps gradient flow when the hidden size is large and you will show this in your homework.

And now, when we understand the idea of multi-head attention, why it’s useful to have multiple layers of it, and how to add account for positions in attention, let’s discuss one of the most impactful neural architectures of 21st century that will be our trusted friend for pretty much the rest of the NLP journey.

Transformer

NLP was quite different just a couple of years ago. The concept of attention that we discussed today first came out in 2014 in the paper “Neural Machine Translation by Jointly Learning to Align and Translate”. That paper used attention as an additional mechanism to recurrent neural networks and only for the machine translation task. This version of attention was quite different from the one we are discussing today. A modern version, with multiple heads was introduced in the paper with a quite memorable title “Attention is all you need” in December 2017. Its main idea was to make attention the main component of the network instead of an auxiliary mechanism it was before. Let’s take a look at the architecture.

It starts with multiple layers of attention. These attentions contextualize our word embeddings and make them more aware of each other with every layer.

The outputs of every self-attention layer are followed by a two-layer neural network. It is applied to each of the embeddings independently, and uses the same weights for all of the positions. Different layers have different weights.

Because each word is fed into the network independently, this layer does not extract any relations and instead it changes the word embeddings in a non-linear way allowing the next layer to extract more complex dependencies.

Basically, attention layers allow the embeddings to interact and feed-forward layers introduce the non-linearity the network needs to learn more complex functions. This process of transforming word-embeddings into new, more complex and contextualized representations is commonly called encoding.

To account for word position, a positional embedding is applied before the very first attention layer.

Finally, the very last layer is followed by a linear layer which projects our contextualized word embeddings to the space of our classes in case of tagging. In the case of classification, we can first average or max-pool these embeddings into a single vector and then insert it into a linear layer which outputs the scores of our classes. Then we just apply softmax-cross-entropy loss to train this and backpropagate to every single parameter of the network, because every single operation we have here is differentiable.

Conceptually, this is the main idea of Transformer Encoder. The rest of the details are mostly technical and allow us, as machine learning practitioners, to actually train these networks without vanishing gradients and instabilities.

First thing is to add Layer Normalization between every layer. It renormalizes the statistics of hidden representations allowing the network to learn quicker and more smoothly. You can literally see the difference in smoothness on the loss plots when using Layer Norm and not using it.

Next concept is called residual or skip connections. The idea behind it is that unless the network is initialized perfectly, which is not practically possible. Your gradients from the loss, when you have many layers (more than ~5), are going to quickly fade out. To mitigate this, every single time we apply an attention operation of a fully-connected network, we add the **input** of this layer to the **output** of this layer.

This does two things: first, it makes it easier for the network to learn identify function in case this particular layer does not want to modify its input. Second, when looking at the backpropagation,

we can see that the gradient takes two ways — inside the FCN or Attention layer which can potentially zero some of it out, or skips it, without any modification.

Finally, this is the full architecture of the Transformer Encoder that, even though relatively complex, is extremely important to be familiar with, because it works incredibly in practice. Now, let's take a quick look at the task that is just as important as this architecture.

Language modeling

The task of language modeling is to predict the next word in a text. At a first look, this task might seem to be very academic, specific and not widely applicable, but in a post-2018 NLP it is a very important task, because it can serve as a proxy to any other NLP task. But for now, let's just concentrate on language modeling as a task of its own.
Formally speaking, language modeling is a task of building a probability distribution of the next word t_k over all possible words from our vocabulary given some prefix t_0, …, t_k-1 using some dataset of texts (corpus) D.

P(t_k|t_0, …, t_k-1)

For example, let's say we want to predict the next word given the prefix "a quick brown fox". To do that we need to compute probabilities for every possible word in our vocabulary.

P(a|a quick brown fox)
P(the|a quick brown fox)
…
P(jumps|a quick brown fox)
…
P(watermelon|a quick brown fox)

The most straightforward approach to this would be just to count how many times you see each of these texts "a quick brown fox a", "a quick brown fox the", "a quick brown fox jumps", …, "a quick brown fox watermelon" in our collection of texts D. Then divide all of them by the number of times you see "a quick brown fox" and get your probability.

P(jumps|a quick brown fox) = Count(a quick brown fox jumps) / Count(a quick brown fox)

However, the language is very diverse and creative. Even if you download all of the books people ever wrote, and the whole internet, it still would not contain all of the possible grammatically and semantically correct texts and you will have 0 probabilities for them, even though they are possible.

The first mentions of this task date way before the computer era, down to the first decade of the 20th century when a Russian mathematician Andrey Markov developed what we now call Markov Chains and applied it to predict the next word in a poem “Ruslan and Ludmila”. As computers became more common and started to process texts a very similar approach was a predominant way to do language modeling up until the 2000s or even 2010s. In was based on word coocurrence counting and estimating probabilities of word A following word B as Count(BA) / Count(B). Next step is to generalize this approach from only using bi-grams (AB) to using N-grams (A…N). The general idea is pretty simple and you can read about it in the Appendix.

Language modeling is a great example of the task where you do need N to be large (>>100 tokens) and until the neural era this was really hard to achieve.

Applying Transformer to Language Modeling

Even though you can directly apply a transformer as a classifier model over all of the prefix words to predict a single next word, this is not what we usually do in practice. It is way more efficient to predict the next word for every prefix of our prefix, meaning we try to predict the next word every time in our sequence. For example, given

Input = A quick brown fox jumps over a lazy

We would want to predict

output = quick brown fox jumps over a lazy dog

This is basically a tagging task where the space of our classes is our vocabulary. Every single word tries to predict the next word. We use exactly the same softmax-cross-entropy loss, but it is applied to every prediction independently. We just feed the input and train it to predict the output.

However, if we apply a transformer to this task directly, it is just going to cheat. Because attention can get any information from the word context, it can just always look at the next word in the input. Well, except the very last word which is not part of the input.

The solution to that is to restrict attention from looking at the future tokens. Given token position i we do this via zeroing out the attention probabilities for all future positions j > i. A common trick here is to compute the attention scores in a regular way

s = Q @ K.T

But then to add a matrix that has 0 on the main diagonal and -inf above the main diagonal.

s += lower_triangular_matrix(lower=0, upper=-inf)

After applying softmax operation all probabilities of scores of -inf will be equal to 0 and this does not allow past tokens like “fox” to attend to the future tokens like “jumps”. This technique is called causal masking, because it masks the scores with -inf values and introduces temporal causality to our predictions (past cannot see the future).

Appendix A: N-gram language models

The first mentions of language modeling task date way before the computer era, down to the first decade of the 20th century when a Russian mathematician Andrey Markov developed what we now call Markov Chains and applied it to predict the next word in a poem “Ruslan and Ludmila”. As computers became more common and started to process texts a very similar approach was a predominant way to do language modeling up until the 2000s or even 2010s.

Let’s take a quick look at the approach. Then we apply the definition of conditional probability P(A|B) = P(A, B) / P(B) which yields

Equation 1: P(t_k|t_0, …, t_k-1) = P(t_0, …, t_k) / P(t_0, …, t_k-1)

Then we assume that the next word only depends on the previous words and now we can apply the chain rule of probabilities (do not confuse with the chain rule for derivatives) which looks like that

Equation 2: P(t_0, … t_k) =
P(t_0) * P(t_1|t_0) * P(t_2|t_0, t_1) * … * P(t_k-2|t_0, …, t_k-3) * P(t_k|t_0, …, t_k-1) =
prod(P(t_j|t_0, …, t_j-1), j = 0..k-1)

Why did we do this? This formula ends with P(t_k|t_0, .., t_k-1) and this is what we looked for originally. Seems to make no sense so far.

Our next step is another, stronger, assumption. The assumption is that the next word only depends on the previous N words where N is some constant. This is called the markov property of order N. Let’s say N = 1, then

P(t_k|t_0, …, t_k-1) = P(t_k|t_k-1)

The trick is, we only make this assumption for Equation 2, not Equation 1 (otherwise, we are back to estimating probability using direct counts). This allows us to compute P(t_k|t_0, …, t_k-1) for k >> N without the need of computing the counts of sequences of length k. Finally, we say that

P(t_k|t_0, …, t_k-1) = P(t_0, …, t_k) / P(t_0, …, t_k-1)

P(t_0, … t_k) = P(t_0) * P(t_1|t_0) * P(t_2|t_1) * P(t_3| t_2) * P(t_4|t_3) * … * P(t_k|t_k-1)

What do we do next? We go to our corpus, count how many times we see every possible unigram and set

P(t) = Count(t) / Sum(Count(w) for w in vocabulary) = Count(t) / total count of words in D

and count all bi-grams setting

P(t_2|t_1) = Count(t_1 t_2) / Sum(Count(t_1 t_k) for k in vocabulary) =
Count(t_2 t_1) / Count(t_1)

Such models are called n-gram language models. While initially they do look simple, they become more complex as you want them to work well.

Language modeling is a great example of the task where you do need N to be large (~500 tokens). However, this is not possible for n-gram models, because as N grows, the number of possible n-grams increases exponentially (vocab^N) and for a large N, most of the n-grams won't be present in the dataset even if they are grammatical and semantically meaningful.

Appendix B: Technical details of multi-head attention

When implementing anything in deep learning, we want to avoid loops. Say we have h heads, doing a for-loop would be a terrible idea because GPUs are highly parallelizable and they are extremely efficient with operations like matrix multiplication, but can be significantly slower when used with sequential operations, like for-loops.

Fortunately, it is possible to implement multi-head attention using just matrix multiplications. First of all, when we compute K1, …, Kh (and Q, and V) we can use a single W_K matrix of size [word_dim, hidden]. This, given the input of shape [seq_len, word_dim] will produce a tensor K of shape [seq_len, hidden].

Next step is to create a head dimension. We will reshape the tensor K (and Q and V) into shape [seq_len, num_heads, hidden / num_heads]. Then, we need to transpose (not reshape!) it over dimensions 0 and 1 to get a tensor of shape [num_heads, seq_len, hidden / num_heads] and then we pretty much perform the same operations as before, just our tensors K, Q, V have an extra head dimension. At this point you can imagine that these are just different examples in a batch (and many multi-head attention implementations do mix the dimensions of batch and heads). Now we just perform pretty much the same operations as before.

s = Q @ K.T

The transposition here is applied between the last (hidden/heads) and second-to-last (seq) dimensions. The matrix multiplication here happens over the last two dimensions. Meaning Q has the shape [heads, seq, hidden/heads] and K.T has the shape [heads, hidden/heads, seq] and the output of this has the shape [heads, seq, seq]. This operation is commonly called batch-matrix multiplication.

Next step of applying softmax to get the probabilities and computing the weighted sum of value vectors stays almost the same. Only the shapes of the h and s tensors are different and you need to remember that the transposition of V is applied to switch the order of the last two dimensions.

Note that for attention we need a lot of transpositions for all of the matrix multiplications to work out shape-wise. ~~There are ways to make the notation more readable such as Named Tensor or ein~~

h = softmax(s) @ V # [num_heads, seq_len, hidden / num_heads]

Finally, we merge the heads together using one more transposition (over num_heads and seq_len) and a reshape

h = h.transpose(0, 1).reshape(1, 2) # [seq_len, hidden]

This gives us a tensor of shape [seq_len, hidden] which we insert into a matrix H which we will call a “mixing matrix”.

Appendix C: N-gram neural language model and its problems

It is possible to apply a word-window classification approach to language modeling. If you look at the language modeling objective closely, it is just a classification objective over our vocabulary given text prefix and this approach works reasonably well. It even has it’s name as “neural n-gram models”.

However, this approach has a couple of drawbacks as N grows. First of all, the number of parameters of the input layer grows with N linearly, as the size of the input vector increases. Second, say your N=512. Let’s say you want to apply your network to the text of size 16, you still need to pad the input and compute an expensive first layer. Third, your history length is N and no more than N.