**Neural Networks – Transformers**

There are two kinds of transformers, so far: Encoder-Only transformers, and Encoder-Decoder transformers. We’ll discuss the latter first.

**Encoder-Only Transformers**

This the model that BERT (Bidirectional Encoder Representations from Transformers) uses. BERT training is ‘masked language approach’. Note weights/biases for nodes/neural networks in the first row, etc., are different, generally. But all subsequent rows are just copies of that architecture. All weights/biases are same; only inputs, and therefore outputs, are different.

A diagram of a network

Description automatically generated

(1) So we have a vocabulary of four words. Typically vocabularies are O(104). Each word has d arrows extending from it (cumulatively represented by a single line) and connecting to the (WE) node. The d arrows carry the weights that represent the word they’re extending from. The (WE) node would be a single layer neural network with d separate linear nodes. Each of the aforementioned d arrows will connect to one of the d linear nodes in (WE). See the Word2Vec file for illustration of these connections for d = 2. So the (WE) node will just output the d-dimensional vector representing the word that is activated by the 1 input. I guess I’ll call these d-dimensional vectors **WE**1, **WE**2, **WE**3, **WE**4, **WE**5. In our d = 2 case, these would be: **WE**j = [WEj(a), WEj(b)]. FWIW, it looks like the BERT model taken off of tensorflowhub uses a preliminary d = 1 WE. So preliminarily, every word/token is simply represented by a single number. *One more thing. It looks like the input phrase is set to be a given size, say, 128 tokens. If our input only comprises 27 tokens, say, then all the rest are set to 0.*

(2) and we add to the output a d-dimensional position embedding vector. The position embedding vector encodes the order of the word in the phrase. We determine the **P** vector by first stacking d/2 sets of sine and cosine functions on top of each other (or side by side), in order of increasing wavelength. Looks like the formula is given by:



where, kn = N-dn/2, and N is commonly taken to be 10, 000. Evidently, it is assumed that d is even. Then if we have m words in our phrase we lay out a sequence of m x coordinates, xj=1,2,…,m, where xj is the coordinate associated with the jth word. I think we take xj = j? And then we assign the jth word the position embedding vector **P**(xj). In our example, these would just be **P**(xj) = [sin(k0xj), cos(k0xj)]. The three d = 2 vectors in our phrase are illustrated below. The difference in curvature between n = 0 and n = 1 is greatly exaggerated, as is the position xj, relative to the wavelength of the waves, as xjkn << 1 really.

A diagram of a graph

Description automatically generated with medium confidence

Then the output of the PE node will be: **PE**j = **WE**j + **P**(xj).

(3) Next we try to build a context for our words. This is where the encoder starts. So each of our vectors goes into a QVK node, and this outputs the α = 1, 2, 3, …, 8 or 12 whatever η-dimensional vectors, **Q** = Query, **V** = Value, **K** = Key. These vectors are arbitrary linear combinations of the components of the input vector. And they’re given by:



where **W**(Q,V,K) are arbitrary ηxd tensors/matrices. Note the weights do not depend on the position, j, of the word in the phrase, and they don’t depend on the type of word. They are all the same. They only depend on whether its Q, V, or K. These arbitrary weights will be determined by minimization as usual. For instance, in our d = 2 case, if we wanted to output Q, V, K vectors with dimensionality η = 2, then we’d have something like,



Or if we wanted to output η = 1 Q, V, K values, then,



(4) Then we use these contexts to create a sort of average/contextual value vector for our jth word. This is a weighted sum of its and all other encoded words’ values. And the weight is modulated by our jth word’s Qj value. Basically **Q**j·**K**i tells us how much context the ith word has on the jth word (and i can be j). And then Cji below is the normalized contribution. Specifically,



And then we construct from these a self-attention vector using more weights. Note dim(**V**) = η, and dim(**SAtt**) = d, so **W**(SA,α) is a d×η dimensional tensor/matrix.



(5) Here the Attention vector and Position Encoded vector are combined together to create a Residual Connection vector, **RC**. This is a two-step process. First we do a linear transformation, L, on the Z-scaled **Satt** vector. So we do:



where **μ**j is a vector of the same dimension as **SAtt**j, naturally, and whose components are all the same – the average of **SAtt**j’s components. σj is the standard deviation of **SAtt**j’s components. Actually, looks like we divide not by σj, but by √(σj2 + ε), where ε is some small number there to prevent things blowing up if σj happened to be 0. β, γ are a random weight, bias, that are initialized to 1, 0, but are to be optimized along with the rest of the weights in the transformer. This normalization and scaling is done to keep the values of coming out of Self Attention node from getting out of hand – basically to keep them somewhat comparable to the input. Anyway, after this, we add to it the **PE** vector. So we have: **RC**j = L(Z(**SAtt**j)) + **PE**j. Adding the input to the output in this way helps to mitigate the vanishing gradient problem.

(6) This is fed into a neural network to output whatever, I guess we’ll call it **NN**j.

(7) And then this is combined with its input to create a new residual connection vector **RC**j = **RC**j + L(Z(**NN**j)). At this point, we can either go straight to step eight, or we can feed this into another encoder network, as is displayed in our picture. Often time we will have several encoders stacked on top of each other. I imagine the weights in the different encoders are all different. *But observe that none of the weights depends on the word j, or the specific word itself*. So we can encode any size sentence with the same number of weights. Obviously the architecture of our model does change with number of words, but that was also the case with simple RNN’s. Anyway, eventually we get to…

(8) where we connect this to a neural network (NNN), and then a softmax and output the first word of our response. This is just as we have done in previous architectures. Note our arrow has dimensionality d, so there will be d inputs to the NNN (often it’s just a single fully connected layer of d nodes with some activation function, so kind of like the WE node). And the NNN will send out connections to the d terminals of the softmax (see the Seq2Seq and Attention models for a d = 2 illustration of the connections). And the weights/biases in the NNN are different than those in the previous NN’s. But whatever they are, they will be the same across the board, for all NNN’s.

**Training**

There are two types of training one typically does for a BERT model. MLM (Masked Language Model?) consists of masking tokens in the input and having it output the masked word. This is illustrated below,

A diagram of a network

Description automatically generated

I’m not sure what the [MASK] token looks like. Well, FWIW, the BERT model taken off of tensorflowhub seems to respresent the [MASK] token with the number 1. Note that all input tokens do output something; we just don’t care what it is per se´, unless it’s the output of the [MASK] token. The loss function would be:



where d is the dimension of the word embedding, and nmask is the number of masks in the input phrase. pi(xj) is the what the probability distribution of the jth word’s masked output should be. And this would be something like (0, 0, 0, 1), presumably, for the blue-circled output, whereas fi(xj) would be the model’s conjectured probability distribution (0.03, 0.02, 0.03, 0.92) in this case. Also, having trained BERT on a large corpus, we can extract a word embedding feature vector fot the word that is being masked. This would be within the arrow circled in orange. Looks like can alternatively use any of the arrows emerging from a previous encoder, like the black one. But maybe using the orange one would have the highest context?

And then there is NSP (Next Sentence Prediction). Here our input is of form [CLS] [Set of Tokens for Sentence 1] [SEP] [Set of Tokens for Sentence 2]. And we add a SE = sentence embedding layer to the PE layer. I imagine the formula for the SE layer is the same as for the PE layer; we’d just have j = 1 for each token in sentence 1 and j = 2 for each token in sentence 2? Whatever. Again, each token produces an output. In this case, one trains the [CLS] token to output a binary prediction NSP = 1 or 0, for if the two sentences are consecutive sentences, or not. Also FWIW, the BERT model taken off of tensorflowhub seems to respresent the [CLS] token with 101, and the [SEP] token with 102.

A diagram of a network

Description automatically generated

This trains the model to understand the relationship between sentences. And the orange-circled arrow/d set of weights could be considered the embedding vector for the sentence. I guess the orange circled arrow would be the embedding with the highest incorporated context. But you can use the black circled arrow instead; this would have less context. This vector would have the same dimensionality, d, as all the other individual words. The loss function would be:



where every term is analogous to before. I guess if we have both [MASK] tokens and [CLS] tokens in our input, then we just add the loss functions?

**Testing or Whatever**

So consider the NSP BERT guy. We can take a BERT trained on a whole bunch of sentences, like all the sentences in Wikipedia, or in a bunch of books. And then BERT will have a bunch of pretrained weights and biases. Then we can feed a random sentence into it: [CLS] [Sentence Tokens]. And the [CLS] token will map to a d-dimensional vector that encodes that sentence. Basically our BERT can be used to map all sentences into a d-dimensional space. We could use this as a spam detector for instance, maybe via support vector machines algorithm, or just logistic regression.

**Decoder-Only Transformers**

These are somewhat similar to Encoder-Only Transformers. The only difference is how they handle Attention, and how they’re trained. Decoder-Only Transformers are the basis of ChatGPT (Generative Pretrained Transformer). Going to make this discussion independent of the previous one. So I’m going to repeat myself a lot. Note weights/biase for nodes/neural networks in the first row, etc., are different, generally. But all subsequent rows are just copies of that architecture. All weights/biases are same; only inputs, and therefore outputs, are different.

A diagram of a computer network

Description automatically generated

(1) So we have a vocabulary of four words. Each word has d arrows extending from it (cumulatively represented by a single line) and connecting to the (WE) node. The d arrows carry the weights that represent the word they’re extending from. The (WE) node would be a single layer neural network with d separate linear nodes. Each of the aforementioned d arrows will connect to one of the d linear nodes in (WE). See the Word2Vec file for illustration of these connections for d = 2. So the (WE) node will just output the d-dimensional vector representing the word that is activated by the 1 input. I guess I’ll call these d-dimensional vectors **WE**1, **WE**2, **WE**3, **WE**4, **WE**5. In our d = 2 case, these would be: **WE**j = [WEj(a), WEj(b)].

(2) and we add to the output a d-dimensional position embedding vector. The position embedding vector encodes the order of the word in the phrase. We determine the **P** vector by first stacking a set of d sine and cosine functions on top of each other (or side by side), in order of increasing wavelength. Looks like the formula is given by:



where, kn = N-dn/2, and N is commonly taken to be 10, 000. Then the output of the PE node will be: **PE**j = **WE**j + **P**(xj).

(3) Next we try to build a context for our words. So each of our WE vectors goes into a QVK node. The QVK node outputs α, **Q**, **V**, **K** vectors. α = 1, 2, 3, …, 8, or 12 is the number of contexts, or ‘heads’. When α > 1, we’re said to have ‘multi-headed attention’. Multihead attention is useful when, for instance, we have more than one question or sentence in the input. The dimensionality of these vectors is often smaller than d. We’ll call it η. These vectors are arbitrary linear combinations of the components of the input vector. And they’re given by:



where **W**(Q,V,K,α) are arbitrary η×d tensors.

(4) Then we use these contexts to create a sort of average/contextual value vector for our jth word. This is a weighted sum of its and all previous words’ values. And the weight is modulated by the jth word’s Qj value. Basically **Q**j·**K**i/√η tells us how much context the ith word has on the jth word (and i can be j). Division by √η is whatever it seems. And then Cji below is the normalized contribution.



And then we construct from these a single masked self-attention vector using more weights. It’s called ‘masked’ because we’re not drawing values from later words in the input. Note dim(**V**) = η, and dim(**MSAtt**) = d, so **W**(MSA,α) is/are a d×η dimensional tensor/matrix.



So if d = 1, η = 2, then we’d have:



(5) Here the Attention vector and Position Encoded vector are added together to create the Residual Connection vector: **RC**j = L(Z(**MSAtt**j)) + **PE**j. Adding the input to the output in this way helps to mitigate the vanishing gradient problem.

(6) Then we send the input into a NN, to introduce some nonlinearity. This outputs **NN**j, say.

(7) And we do **RC**j = L(Z(**NN**j)) + **RC**j. Now we can stack another decoder on top if we want. Note this is only effective if we have that NN from the previous step. That’s because the QVK and MSAtt blocks are linear operations. And linear operations on top of linear operations are still just linear operations, and so we wouldn’t be fundamentally changing the behavior/capabilities of our model. But putting the NN in there makes it different.

(8) Eventually, we get to the end of our sequence, and take the output of our RC, and feed it into another neural network (NNN), and then a softmax and output the first word of our response. This is just as we have done in previous architectures. Note our arrow has dimensionality d, so there will be d inputs to the NNN (often it’s just d nodes with some activation function, so kind of like the WE node). And the NNN will send out connections to the d terminals of the softmax (see the Seq2Seq and Attention models for a d = 2 illustration of the connections), which will output a word. And the weights/biases in the NNN are different than those in the previous NN’s. But whatever they are, they will be the same across the board, for all NNN’s. But we kind of ignore the output anyway, since we know what the next word of the sentence will be. And we input the known next word into the decoder and proceed as before. Eventually we input the EOS (end of sequence) token.

(9-16) And then we repeat all the steps with this token that we did with the previous words. Importantly, we do *not* restart the position encoding. So our EOS token would correspond to j = 4. Only difference is that now the output word will be fed into the input of the next decoder sequence. Eventually, the EOS token will be output, and the algorithm will stop.

**Training**

GPT training is autoregressive, perhaps also called *Teacher Forcing*. It predicts one word at a time. So for instance if you input ‘the cat is on’ into the encoder, and we expect a decoder output of ‘the mat’, then we’d input ‘the cat is on’ into the encoder, and calculate the loss from just the first output word, which should be ‘the’. Then we’d input ‘the cat is on’ into the encoder, and I think we’d input ‘the’ into first word of the decoder (this would be the first input into the decoder after the EOS input), and grade it on the output of ‘mat’. Anyway, our loss function would be the usual cross entropy I guess, for all n output words, xj.



**Encoder-Decoder Transformers**

And now we have Encoder-Decoder transformers. It’s mostly the same. We’d use this for something like language translation. Note weights/biase for nodes/neural networks in the first row of encoder are different, generally. But all subsequent rows of encoder are just copies of that architecture. All weights/biases are same; only inputs, and therefore outputs, are different. And all weights/biase for nodes/neural networks in the first row of decoder are different, generally. But all subsequent rows of decoder are just copies of that architecture. All weights/biases are same; only inputs, and therefore outputs, are different.

A diagram of a network

Description automatically generated

(1) So we have a vocabulary of four words. Each word has d arrows extending from it (cumulatively represented by a single line) and connecting to the (WE) node. The d arrows carry the weights that represent the word they’re extending from. The (WE) node would be a single layer neural network with d separate linear nodes. Each of the aforementioned d arrows will connect to one of the d linear nodes in (WE). See the Word2Vec file for illustration of these connections for d = 2. So the (WE) node will just output the d-dimensional vector representing the word that is activated by the 1 input. I guess I’ll call these d-dimensional vectors **WE**1, **WE**2, **WE**3, **WE**4, **WE**5. In our d = 2 case, these would be: **WE**j = [WEj(a), WEj(b)].

(2) and we add to the output a d-dimensional position embedding vector. The position embedding vector encodes the order of the word in the phrase. We determine the **P** vector by first stacking d/2 sets of sine and cosine functions on top of each other (or side by side), in order of increasing wavelength. Looks like the formula is given by:



where, kn = N-dn/2, and N is commonly taken to be 10, 000. Then the output of the PE node will be: **PE**j = **WE**j + **P**(xj).

(3) Next we try to build a context for our words. This is where the encoder starts. So each of our vectors goes into a QVK node, and this outputs the α = 1, 2, 3, …, 8 or 12 whatever η-dimensional vectors, **Q** = Query, **V** = Value, **K** = Key. These vectors are arbitrary linear combinations of the components of the input vector. And they’re given by:



where **W**(Q,V,K) are arbitrary ηxd tensors/matrices. Note the weights do not depend on the position of the word in the phrase. They are all the same. They only depend on whether its Q, V, or K. These arbitrary weights will be determined by minimization as usual.

(4) Then we use these contexts to create a sort of average/contextual value vector for our jth word. This is a weighted sum of its and all other encoded words’ values (not just previous words). And the weight is modulated by the jth word’s Qj value. Basically **Q**j·**K**i tells us how much context the ith word has on the jth word (and i can be j). And then Cji below is the normalized contribution. Specifically,



And then we construct from these a self-attention vector using more weights. Note dim(**V**) = η, and dim(**SAtt**) = d, so **W**(SA,α) is a d×η dimensional tensor/matrix.



(5) Here the Attention vector and Position Encoded vector are combined together to create a Residual Connection vector, **RC**. We have: **RC**j = L(Z(**SAtt**j)) + **PE**j. Adding the input to the output in this way helps to mitigate the vanishing gradient problem.

(6) This is fed into a neural network to output whatever, I guess we’ll call it **NN**j.

(7) And then this is combined with its input to create a new residual connection vector **RC**j = **RC**j + L(Z(**NN**j)), where the L(Z()) refers to the same linear scaling we did in prior step. At this point, we can either go straight to step eight, or we can feed this into another encoder network, as is displayed in our picture. Often time we will have several encoders stacked on top of each other. I imagine the weights in the different encoders are all different. *But observe that none of the weights depends on the word j, or the specify word itself*. So we can encode any size sentence with the same number of weights. Obviously the architecture of our model does change with number of words, but that was also the case with simple RNN’s. Anyway, eventually we get to…

(8) Now like in the Attention network, I guess we find that it isn’t sufficient to combine all the words in the encoder sequence into a single output to feed into the decoder sequence. Rather we feed each word in individually. So we prepare more Q, V, K vectors for this purpose. And we have:



The weights, **W**, will be different than the Q, V, K weights, **W**, we used in the encoders. But like those, these new weights do not depend on j. Oh, and looks like we aren’t really using the **Q**’s at all.

(9) To start the decoding process, we input the EOS sequence. And then we repeat all the steps with this token that we did with the previous words.

(10) Importantly, we *do* restart the position encoding. In our example, the first word of the decoder would correspond to position 1.

(11) Then we feed this input into our first decoder. We make Q, V, K values for the succeeding masked self-attention vector. These are new weights, different than those of the encoder.



(12) Then we create the masked self-attention vectors for the decoded words. I *think* these self-attention vectors as masked? So first we’d construct,



(where sum runs over just decoder words) And then we construct from this the masked self-attention vector:



(13) Here the Masked Self-Attention vector and Position Encoded vector are added together to create the Residual Connection vector: **RC**j = L(Z(**MSAtt**j)) + **PE**j.

(14) Then we do the Q,V,K thing again for the decoder words. But we’re about to incorporate the encoder words this time around.



(15) So now we combine the encoder Q, V, K’s and decoder Q, V, K’s. So first we create the contextual value vector (sum runs over all elements of the encoder, and then (only) the jth element of the decoder):



Will observe the similarity between this and the Attention model. Only here, in the Transformer model, it seems we separately encode the information content, V, of the word, and its relational content to the other words, Q, and K. In the former Attention model, the vector v performed both functions. Anyway, then we output the encoder-decoder attention vector. Of course W(EDA,α) is an η×d tensor/matrix.



(16) Here the Encoder-Decoder Attention vector and Position Encoded vector are added together to create the Residual Connection vector: **RC**j = L(Z(**EDAtt**j)) + **PE**j.

(17) This will then go into a NN, outputting **NN**j.

(18) And we add the input again to get a new residual connection: **RC**j = **RC**j + L(Z(**NN**j)). At this point, we can feed the input into another decoder, or we can go straight to step nineteen. Presumably, the weights in the new encoder would be all different than the present encoder.

(19) Eventually, we will take our output and feed it into another neural network (NNN), and then a softmax and output the first word of our response. This is just as we have done in previous architectures. Note our arrow has dimensionality d, so there will be d inputs to the NNN (often it’s just d nodes with some activation function, so kind of like the WE node). And the NNN will send out connections to the d terminals of the softmax (see the Seq2Seq and Attention models for a d = 2 illustration of the connections). And the weights/biases in the NNN are different than those in the previous NN’s. But whatever they are, they will be the same across the board, for all NNN’s. The output word will be fed into the next layer of our Transformer (it keeps unrolling like a RNN). And all the steps are repeated. All same-colored lines will have same weights as before; so note that none of our model’s weights will depend on how many words or what type of words we have. So we can use the same model to output any length phrase. Eventually, the EOS token will be output, and the algorithm will stop.

**Training**

I’d guess that training is also autoregressive, perhaps also called *Teacher Forcing*. It predicts one word at a time. So for instance if you input ‘I am a student’ into the encoder, and we expect a decoder output of ‘Je suis etudiant’, then we’d input ‘I am a student’ into the encoder, and calculate the loss from just the first output word, which should be ‘Je’. Then we’d input ‘I am a student’ into the encoder, and I think we’d input ‘Je’ into first word of the decoder (this would be the first input into the decoder after the EOS input), and grade it on the output of ‘suis’. And then we’d input ‘I am a student’ into the encoder, and ‘Je suis’ into the decoder, and grade it on the output of ‘etudiant’. Anyway, our loss function would be the usual cross entropy I guess, for all n output words, xj, just as before.

