Natural Language Processing
DSECL ZG565

Dr. Chetana Gavankar, Ph.D,
IIT Bombay-Monash University Australia
Chetana.gavankar@pilani.bits-pilani.ac.in

BITS Pilani
Pilani Campus

innovate achieve lead

BITS Pilani
Pilani Campus

Encoder-Decoder Models, Attention and Contextual Embedding's

Date – 2nd March 2024

These slides are prepared by the instructor, with grateful acknowledgement of Prof. Jurafsky and Martin and many others who made their course materials freely available online.

Session Content

- Encoder-Decoder with RNN
- Issues with recurrent models
- Attention
- Transformer architecture
  - Self attention and Multihead-attention
  - Position encoding
  - Masked Self attention
  - Transformer encoder-decoder
  - Layer normalization
  - Residual connection
- Pre training
- Contextual Embeddings
  - BERT

Simple feedforward sliding-window as a language model

Figure 9.1 Simplified sketch of a feedforward neural language model moving through a text. At each time step t the network converts N context words, each to a d-dimensional embedding, and concatenates the N embeddings together to get the $Nd \times 1$ unit input vector $\mathbf{x}$ for the network. The output of the network is a probability distribution over the vocabulary representing the model's belief with respect to each word being the next possible word.

- **Language Model**: A system that predicts the next word
- Embeddings as inputs
- Solve the main problem of the simple n-gram models
  - n-grams are based on words rather than embeddings
- Limitation:
  - Limited context.
  - Anything outside the context window has no impact on the decision being made
  - E.g phrase 'all the' appears in one window in the 2 and 3 positions, and in the next window in 1 and 2 positions, forcing the network to learn two separate patterns for the same item.

Deep Learning Architectures for Sequence Processing

- **Recurrent neural networks and transformer** networks
- Both capture and exploit the **temporal** nature of language
- use the **prior context**, allowing the model's decision to depend on information from words in the past.
- **The transformer uses mechanisms (self-attention and positional encodings) that help focus on how words relate to each other over long distances**

RNN Architectures for NLP Tasks

Ex: POS Tagging, Named Entity Tagging

Ex: Sentiment Analysis

Ex: Predict Next Word

Ex: Language Translation

Encoder-Decoder or Sequence-to-Sequence Networks

- **Models capable of generating contextually appropriate, arbitrary length, output sequences**
- One neural network takes input and produces a neural representation
- Another network produces output based on that neural representation
- Many NLP tasks can be phrased as sequence-to-sequence:
  - Summarization (long text → short text)
  - Dialogue (previous utterances → next utterance)
  - Parsing (input text → output parse as sequence)
  - Code generation (natural language → Python code)

Encoder-Decoder architecture

1. **Encoder**: accepts an input sequence, $\mathbf{x_{1:n}}$ and generates a corresponding sequence of contextualized representations, $\mathbf{h_{1:n}}$
2. **Context vector c**: function of $\mathbf{h_{1:n}}$ and conveys the essence of the input to the decoder.
3. **Decoder**: accepts **c** as input and generates an arbitrary length sequence of hidden states $\mathbf{h_{1:m}}$ from which a corresponding sequence of output states $\mathbf{y_{1:m}}$ can be obtained.

LSTMs, convolutional networks,and Transformers can all be employed as encoders/decoders

Simple MT example using encoder-decoder

MT with RNN based encoder-decoder

Figure 10.4 Translating a single sentence (inference time) in the basic RNN version of encoder-decoder approach to machine translation. Source and target sentences are concatenated with a separator token in between, and the decoder uses context information from the encoder's last hidden state.

Encoder - Decoder using RNN

Encoder - Decoder for Language Translation

- Encoder generates a contextualized representation of the input (last state)
- Decoder takes that state and <u>autoregressively</u> generates a sequence of outputs

word generated at each time step is conditioned on word from previous step.

Encoder - Decoder

$$
\begin{aligned}
\mathbf{c} &= \mathbf{h}_n^e \\
\mathbf{h}_0^d &= \mathbf{c} \\
\mathbf{h}_t^d &= g(\hat{y}_{t-1}, \mathbf{h}_{t-1}^d, \mathbf{c}) \\
\mathbf{z}_t &= f(\mathbf{h}_t^d) \\
y_t &= \text{softmax}(\mathbf{z}_t)
\end{aligned}
$$

Encoder - Decoder for Language Translation

Encoder - Decoder

Training

Total loss is the average cross-entropy loss per target word:

$$L = \frac{1}{T}\sum_{i=1}^{T} L_i$$

Encoder - Decoder

Teaching Forcing

- Force the system to use the gold target token from training as the next input x_{t+1}, rather than allowing it to rely on the (possibly erroneous) decoder output $\hat{y}_t$.
- Speeds up training

Issues with recurrent models

- **O(sequence length)** steps for distant word pairs to interact means:
  - Forward and backward passes have **O(sequence length)** unparallelizable operations
- RNNs are unrolled "left-to-right".
- This encodes linear locality: a useful heuristic!
  - Nearby words often affect each other's meanings
  - Hard to learn long-distance dependencies (because gradient problems!)

Vanishing gradient intuition

$$\frac{\partial J^{(4)}}{\partial h^{(1)}} = \boxed{\frac{\partial h^{(2)}}{\partial h^{(1)}}} \times \boxed{\frac{\partial h^{(3)}}{\partial h^{(2)}}} \times \boxed{\frac{\partial h^{(4)}}{\partial h^{(3)}}} \times \frac{\partial J^{(4)}}{\partial h^{(4)}}$$

chain rule!

Vanishing gradient problem:
When these are small, the gradient signal gets smaller and smaller as it backpropagates further

What happens if these are small?

Why is vanishing gradient a problem?

Gradient signal from far away is lost because it's much smaller than gradient signal from close-by.

So, model weights are basically updated only with respect to near effects, not long-term effects.

Effect of vanishing gradient on RNN-LM

- **LM task:** *When she tried to print her tickets, she found that the printer was out of toner. She went to the stationery store to buy more toner. It was very overpriced. After installing the toner into the printer, she finally printed her* _____

- To learn from this training example, the RNN-LM needs to model the dependency between *"tickets"* on the 7th step and the target word *"tickets"* at the end.

- But if the gradient is small, the model can't learn this dependency
  - So, the model is unable to predict similar long-distance dependencies at test time

- In practice a simple RNN will only condition ~7 tokens back [vague rule-of-thumb]

Encoder-decoder bottleneck

- Final state of the **E** is the only context available to **D**
- It must represent absolutely everything about the meaning of the source text
- The only thing the decoder knows about the source text is what's in this context vector

Attention ! (in RNN based encoder-decoder)

Without attention, a decoder sees the same context vector , which is a static function of all the encoder hidden states

$$\mathbf{h}_t^d = g(\hat{y}_{t-1}, \mathbf{h}_{t-1}^d, \mathbf{c})$$

Attention ! (in RNN based encoder-decoder)

Without attention, a decoder sees the same context vector , which is a static function of all the encoder hidden states

With attention, decoder to sees a different, dynamic, context, which is a function of all the encoder hidden states

$$\mathbf{h}_t^d = g(\hat{y}_{t-1}, \mathbf{h}_{t-1}^d, \mathbf{c})$$

$$\mathbf{h}_i^d = g(\hat{y}_{i-1}, \mathbf{h}_{i-1}^d, \mathbf{c}_i)$$

Attention !

Without attention, a decoder sees the same context vector , which is a static function of all the encoder hidden states

With attention, decoder to sees a different, dynamic, context, which is a function of all the encoder hidden states

$$\mathbf{h}_t^d = g(\hat{y}_{t-1}, \mathbf{h}_{t-1}^d, \mathbf{c})$$

$$\mathbf{h}_i^d = g(\hat{y}_{i-1}, \mathbf{h}_{i-1}^d, \mathbf{c}_i)$$

With attention, decoder gets information from all the hidden states of the encoder, not just the last hidden state of the encoder

Each context vector is obtained by taking a weighted sum of all the encoder hidden states.

The weights focus on ('attend to') a particular part of the source text that is relevant for the token the decoder is currently producing

Attention !

Step -1 : **Find out how relevant each encoder state is to the present decoder state** $\mathbf{h}_{i-1}^{d}$

Compute a score of similarity between $\mathbf{h}_{i-1}^{d}$ **and all the encoder states :** $score(\mathbf{h}_{i-1}^{d}, \mathbf{h}_{j}^{e})$

Dot Product Attention : $\text{score}(\mathbf{h}_{i-1}^{d}, \mathbf{h}_{j}^{e}) = \mathbf{h}_{i-1}^{d} \cdot \mathbf{h}_{j}^{e}$

Step -2 : **Normalize all the scores with softmax to create a vector of weights,** $\alpha_{i,j}$

$\alpha_{i,j}$ **indicates the proportional relevance of each encoder hidden state j to the prior hidden decoder state,** $\mathbf{h}_{i-1}^{d}$

$$\alpha_{ij} = \text{softmax}(\text{score}(\mathbf{h}_{i-1}^{d}, \mathbf{h}_{j}^{e}) \ \forall j \in e)$$

$$= \frac{\exp(\text{score}(\mathbf{h}_{i-1}^{d}, \mathbf{h}_{j}^{e})}{\sum_{k} \exp(\text{score}(\mathbf{h}_{i-1}^{d}, \mathbf{h}_{k}^{e}))}$$

Attention !

Step -3 : **Given the distribution in α, compute a fixed-length context vector for the current decoder state by taking a weighted average over all the encoder hidden states**

$$\mathbf{c}_i = \sum_j \alpha_{ij} \mathbf{h}_j^e$$

Attention !

compute a fixed-length context vector for the current decoder state by taking a weighted average over all the encoder hidden states.

$$\mathbf{h}_i^d = g(\hat{y}_{i-1}, \mathbf{h}_{i-1}^d, \mathbf{c}_i)$$

Attention is a *general* Deep Learning technique

- Attention significantly improves performance
  - It's very useful to allow decoder to focus on certain parts of the source
- Attention solves the bottleneck problem
  - Attention allows decoder to look directly at source; bypass bottleneck
- Attention helps with the vanishing gradient problem
  - Provides shortcut to faraway states
- By inspecting attention distribution, we see what the decoder was focusing on

- Attention ⇒ Ability to compare an item of interest to a collection of other items in a way that reveals their relevance in the current context.
- Self-attention ⇒

  > Set of ***comparisons*** are to other elements ***within a given sequence***

  > Use these comparisons to compute an output for the current input

Transformers

- 2017, NIPS, Vaswani et. al., **Attention Is All You Need** !!!
- Made up of **transformer blocks** in which the key component is **self-attention** layers
- Transformers are **not based on recurrent connections** ⇒ Parallel implementations possible ⇒ Efficient to scale (comparing LSTM)

Figure 9.18 A transformer block showing all the layers.

- Each Block consists of:
  - Self-attention
  - Add & Norm
  - Feed-Forward
  - Add & Norm

Self attention layer

Figure 9.15 Information flow in a causal (or masked) self-attention model. In processing each element of the sequence, the model attends to all the inputs up to, and including, the current one. Unlike RNNs, the computations at each time step are independent of all the other steps and therefore can be performed in parallel.

- Computation of y_3 is based on a set of comparisons between the input x_3 and its preceding elements x_1 and x_2, and to x_3 itself
- When processing each item in the input, the model has no access to information about inputs beyond the current one.
- This ensures that we can use this approach to create language models and use them for **autoregressive generation**

Bidirectional self-attention model

Figure 11.2 Information flow in a bidirectional self-attention model. In processing each element of the sequence, the model attends to all inputs, both before and after the current one.

Attention as a soft, averaging lookup table

We can think of **attention** as performing fuzzy lookup in a key-value store.

In a **lookup table**, we have a table of **keys** that map to **values**. The **query** matches one of the keys, returning its value.

In **attention**, the **query** matches all **keys** *softly*, to a weight between 0 and 1. The keys' **values** are multiplied by the weights and summed.

Parallelized using efficient matrix multiplication

Create three vectors from each of the encoder's input values (query, key, value)

More general definition of attention:

Given a set of vector *values*, and a vector *query*, attention is a technique to compute a weighted sum of the values, dependent on the query.

- We sometimes say that the query *attends to* the values. E.g. in the seq2seq + attention model, each decoder hidden state (query) *attends to* all the encoder hidden states (values).

Self-attention step for an entire sequence of N tokens:

$$\boxed{SelfAttention(\mathbf{Q},\mathbf{K},\mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V}}$$

D_k =dimensionality of the query and key vectors acts as regularization and improve performance of larger models

Keys and Queries Matrix

Keys and Queries Matrix

Values Matrix

Self-Attention Hypothetical Example

Image credit: https://towardsdatascience.com/attention-please-85bd0abac41#:~:text=If%20keys%2C%20values%20and%20queries,things%20at%20the%20same%20time.

Self-Attention: keys, queries, values from the same sequence

computing a single output at a single time step i

1. Transform each word embedding with weight matrices Q, K, V , each in $\mathbb{R}^{d\times d}$

$$q_i = Qx_i \text{ (queries)} \qquad v_i = Vx_i \text{ (values)}$$

$$k_i = Kx_i \text{ (keys)}$$

2. Compute pairwise similarities between keys and queries; normalize with softmax

$$e_{ij} = q_i^{\top} k_j \qquad \alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{j'} \exp(e_{ij'})}$$

3. Compute output for each word as weighted sum of values

$$o_i = \sum_j \alpha_{ij}\, v_i$$

4.The similarity between words is called **alignment**

5.The query and key vectors are used to calculate **alignment scores** that are measures of how well the query and keys match.

Figure 9.16 Calculating the value of y_3, the third element of a sequence using causal (left-to-right) self-attention.

…and these sums, which combine separate encodings for both input words, **Let's** and **go**, relative to their similarity to **Let's**, are the **Self-Attention** values for **Let's**.

The good news is that we don't need to re-calculate the **Keys** and **Values**.
Value
2.5
sum
-2.1
sum
x -0.3
x -0.2
x 1.5
x -1.0
Query
-1.0
sum
3.7
sum
x 2.4
x -2.8
x 0.6
x 1.1
Key
-4.7
sum
1.9
sum
x -1.4
x 0.9
x -1.7
x 0.5
1.9
sum
-0.2
sum
Key
x -1.4
x 0.9
x -1.7
x 0.5
-2.7
sum
1.5
sum
Value
x -0.3
x -0.2
x 1.5
x -1.0
1.87
1.09
-1.68
0.67
Self-Attention
values for Let's
2.5
-2.1
1 0 0 0
let's to go <EOS>
0 0 1 0
let's to go <EOS>

Instead, all we need to do is create the **Query** that represents the word **go**…

And this can help establish how each word in the input is related to the others.

Also, if we think of this unit, with it's **Weights** for calculating **Queries**, **Keys** and **Values**, as a **Self-Attention** cell...

The **pizza** came out of the **oven** and **it** tasted good!

Self attention Example

3 input vectors x_i
[1, 0, 1, 0]
[0, 2, 0, 2]
[1, 1, 1, 1]

Weights for Key (K)
[[0, 0, 1],
[1, 1, 0],
[0, 1, 0],
[1, 1, 0]]

Key representation for Input

$$[1, 0, 1, 0] \cdot \begin{matrix} [0, 0, 1] \\ [1, 1, 0] \\ [0, 1, 0] \\ [1, 1, 0] \end{matrix} = [0, 1, 1]$$

Key for all inputs
[0 1 1]
[4 4 0]
[2 3 1]

$k_i = Kx_i$ (keys)

Weights for query (Q)
[[1, 0, 1],
[1, 0, 0],
[0, 0, 1],
[0, 1, 1]]

Query representation for Input

$$[1, 0, 1, 0] \cdot \begin{matrix} [1, 0, 1] \\ [1, 0, 0] \\ [0, 0, 1] \\ [0, 1, 1] \end{matrix} = \mathbf{[1, 0, 2]}$$

Query for all inputs
[1, 0, 2]
[2, 2, 2]
[2, 1, 3]

$q_i = Qx_i$ (queries)

Weights for Value (V)
[[0, 2, 0],
[0, 3, 0],
[1, 0, 3],
[1, 1, 0]]

Value representation for Input

$$[1, 0, 1, 0] \cdot \begin{matrix} [0, 2, 0] \\ [0, 3, 0] \\ [1, 0, 3] \\ [1, 1, 0] \end{matrix} = [1, 2, 3]$$

Value for all inputs
[1, 2, 3]
[2, 8, 0]
[2, 6, 3]

$v_i = Vx_i$ (values)

https://towardsdatascience.com/illustrated-self-attention-2d627e33b20a

3 attention scores for first input *i* : (e_{ij})

[0, 1, 1]

[1, 0, 2] . [4, 4, 0] = [2, 4, 4]

[2, 3, 1]

$$e_{ij} = q_i^T k_j$$

3 softmax attention scores (α_{ij})

softmax([2, 4, 4]) = [0.0, 0.5, 0.5]

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{j'} \exp(e_{ij'})}$$

weighted values

1: 0.0 * [1, 2, 3] = [0.0, 0.0, 0.0]

2: 0.5 * [2, 8, 0] = [1.0, 4.0, 0.0]

3: 0.5 * [2, 6, 3] = [1.0, 3.0, 1.5]

$$\alpha_{ij} v_i$$

Output 1, which is based on the query representation from Input 1 interacting with all other keys, including itself

[0.0, 0.0, 0.0]

+ [1.0, 4.0, 0.0]

+ [1.0, 3.0, 1.5]

= [2.0, 7.0, 1.5]

$$o_i = \sum_j \alpha_{ij} v_i$$

All the outputs after first iteration

[2.0, 7.0, 1.5], # Output 1

[2.0000, 8.0, 0.0], # Output 2

[2.0, 7.8, 0.3] # Output 3

Multihead-Attention

- Different words in a sentence can relate to each other in many different ways simultaneously

  >> A single transformer block to learn to capture all of the different kinds of parallel relations among its inputs is inadequate.

- Multihead self-attention layers

  >> Heads ⇒ sets of self-attention layers, that reside in parallel layers at the same depth in a model, each with its own set of parameters.

  >> Each head learn different aspects of the relationships that exist among inputs at the same level of abstraction

$$\text{MultiHeadAttention}(\mathbf{X}) = (\mathbf{head}_1 \oplus \mathbf{head}_2 \ldots \oplus \mathbf{head}_h)\mathbf{W}^O$$

$$\mathbf{Q} = \mathbf{X}\mathbf{W}_i^Q \; ; \; \mathbf{K} = \mathbf{X}\mathbf{W}_i^K \; ; \; \mathbf{V} = \mathbf{X}\mathbf{W}_i^V$$

$$\mathbf{head}_i = \text{SelfAttention}(\mathbf{Q}, \mathbf{K}, \mathbf{V})$$

Multihead-Attention

Each of the multihead self-attention layers is provided with its own set of key, query and value weight matrices. The outputs from each of the layers are concatenated and then projected down to d, thus producing an output of the same size as the input so layers can be stacked.

Hypothetical Example of Multi-Head Attention

I went to Stanford CS 224n and learned

Self Attention Vs Cross Attention

$$\mathbf{Q} = \mathbf{X}\mathbf{W}^{\mathbf{Q}};\ \mathbf{K} = \mathbf{X}\mathbf{W}^{\mathbf{K}};\ \mathbf{V} = \mathbf{X}\mathbf{W}^{\mathbf{V}}$$

$$\boxed{\mathit{SelfAttention}(\mathbf{Q},\mathbf{K},\mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^{\mathsf{T}}}{\sqrt{d_k}}\right)\mathbf{V}}$$

$$\mathbf{Q} = \mathbf{W}^{\mathbf{Q}}\mathbf{H}^{dec[i-1]};\ \mathbf{K} = \mathbf{W}^{\mathbf{K}}\mathbf{H}^{enc};\ \mathbf{V} = \mathbf{W}^{\mathbf{V}}\mathbf{H}^{enc}$$

$$\mathit{CrossAttention}(\mathbf{Q},\mathbf{K},\mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^{\mathsf{T}}}{\sqrt{d_k}}\right)\mathbf{V}$$

Self Attention Example

...and these sums, which combine separate encodings for both input words, **Let's** and **go**, relative to their similarity to **Let's**, are the **Self-Attention** values for **Let's**.

Cross Attention Example

Cross Attention Example

Barriers and solutions for Self-Attention as a building block

Barriers	Solutions
• Doesn't have an inherent notion of order!	• Add position representations to the inputs
• No nonlinearities for deep learning magic! It's all just weighted averages	• Easy fix: apply the same feedforward network to each self- attention output.

• Need to ensure we don't "look at the future" when predicting a sequence
  • Like in machine translation
  • Or language modeling
	• Mask out the future by artificially setting attention weights to 0!

Fixing the first self-attention problem: sequence order

- With RNNs, information about the order of the inputs was built into the structure of the model.
- self-attention ditches sequential operations in favor of parallel computation
- Since self-attention doesn't build in order information, we need to encode the order of the sentence in our keys, queries, and values.
- Consider representing each **sequence index** as a **vector**

$$\boldsymbol{p}_i \in \mathbb{R}^d, \text{ for } i \in \{1,2,\dots,n\} \text{ are position vectors}$$

- Easy to incorporate this info into our self-attention block: just add the $\boldsymbol{p}_i$ to our inputs!
- $\boldsymbol{x}_i$ is the embedding of the word at index i. The positioned embedding is:

$$\tilde{\boldsymbol{x}}_i = \boldsymbol{x}_i + \boldsymbol{p}_i$$

Position encoding : simple way

Figure 9.20 A simple way to model position: simply adding an embedding representation of the absolute position to the input word embedding.

Position representation vectors through sinusoids

Each position/index is mapped to a vector

Positional Encoding Matrix with d=4, n=100

Sequence	Index of token, k	i=0	i=0	i=1	i=1
I	0	$P_{00}=\sin(0)$ = 0	$P_{01}=\cos(0)$ = 1	$P_{02}=\sin(0)$ = 0	$P_{03}=\cos(0)$ = 1
am	1	$P_{10}=\sin(1/1)$ = 0.84	$P_{11}=\cos(1/1)$ = 0.54	$P_{12}=\sin(1/10)$ = 0.10	$P_{13}=\cos(1/10)$ = 1.0
a	2	$P_{20}=\sin(2/1)$ = 0.91	$P_{21}=\cos(2/1)$ = -0.42	$P_{22}=\sin(2/10)$ = 0.20	$P_{23}=\cos(2/10)$ = 0.98
Robot	3	$P_{30}=\sin(3/1)$ = 0.14	$P_{31}=\cos(3/1)$ = -0.99	$P_{32}=\sin(3/10)$ = 0.30	$P_{33}=\cos(3/10)$ = 0.96

Positional Encoding Matrix for the sequence 'I am a robot'

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{model}})$$
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{model}})$$

d= Dimension of the output embedding space

i= Used for mapping to column indices $0 \le i < d/2$, with a single value of i maps to both sine and cosine functions

Image credit: https://machinelearningmastery.com/a-gentle-introduction-to-positional-encoding-in-transformer-models-part-1/

Position encoding : simple way

Position encoding : simple way

Position encoding : simple way

Adding nonlinearities in self-attention

- Note that there are **no elementwise nonlinearities in self-attention**; stacking more self-attention layers just re-averages **value** vectors
- Easy fix: add a **feed-forward network** to post-process each output vector.

Intuition: the FF network processes the result of attention

Masking the future in self-attention

$$SelfAttention(\mathbf{Q},\mathbf{K},\mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\intercal}{\sqrt{d_k}}\right)\mathbf{V}$$

- Calculation of the comparisons in QK results in a score for each query value to every key value, **including those that follow the query**
- Inappropriate in the setting of language modeling
- To use self-attention in **decoders**, we need to ensure we can't peek at the future.
- To enable parallelization, we **mask out attention** to future words by setting attention scores to $-\infty$

$$e_{ij} = \begin{cases} q_i^\intercal k_j, j \leq i \\ -\infty, j > i \end{cases}$$

We can look at these (not greyed out) words

For encoding these words

	[START]	The	chef	who
[START]		$-\infty$	$-\infty$	$-\infty$
The			$-\infty$	$-\infty$
chef				$-\infty$
who				

Residual connections

- **Residual connections** are a trick to help models train better.
- Pass information from a lower layer to a higher layer without going through the intermediate layer.
- Allowing information from the activation going forward and the gradient going backwards to skip a layer

Instead of $X^{(i)}$ = Layer($X^{(i-1)}$) (where i represents the layer)

We let $X^{(i)} = X^{(i-1)}$ + Layer($X^{(i-1)}$)
(so we only have to learn "the residual" from the previous layer)

These **4** features allow the **Transformer** to: encode words into numbers, encode the positions of the words, encode the relationships among the words, and relatively easily and quickly train in parallel.

Layer normalization

- **Layer normalization** is a trick to help models train faster.
  - Layer norm is a variation of the standard score, or z-score, from statistics applied to a single hidden layer
- Let $x \in \mathbb{R}^d$ be an individual (word) vector in the model.

$$\hat{\mathbf{x}} = \frac{(\mathbf{x} - \mu)}{\sigma}$$

$$\mu = \frac{1}{d_h}\sum_{i=1}^{d_h} x_i$$

$$\sigma = \sqrt{\frac{1}{d_h}\sum_{i=1}^{d_h}(x_i - \mu)^2}$$

$$\boxed{\begin{aligned} \mathbf{z} &= \text{LayerNorm}(\mathbf{x} + \text{SelfAttn}(\mathbf{x})) \\ \mathbf{y} &= \text{LayerNorm}(\mathbf{z} + \text{FFNN}(\mathbf{z})) \end{aligned}}$$

γ and β are learnable parameters

$$LayerNorm = \gamma\,\hat{\mathbf{x}} + \beta$$

The Transformer Encoder

- It receives an input and **builds a representation of it (its features): contextual embeddings.**
- **Bidirectional** model
- A stack of identical layers
- Multi-headed self attention : Models context
- Feed-forward layers :Computes non-linear hierarchical features
- Layer norm and residuals : Makes training deep networks healthy
- Positional embeddings : Allows model to learn relative positioning

The Transformer Decoder

- **A conditional language model** that attends to the encoder representation and generates the target words one by one, at each timestep
- Transformer Decoder is modified to perform **cross- attention** (also sometimes called encoder-decoder attention or source attention) to the output of the Encoder
- The decoder, works sequentially and can only pay attention to the words in the sentence that it has already translated (Masked attention layer)
- **Unidirectional model**
- **Masking**:
  - In order to parallelize operations while not looking at the future.
  - Keeps information about the future from "leaking" to the past.

The Transformer Encoder-Decoder

$$SelfAttention(\mathbf{Q},\mathbf{K},\mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V}$$

Several attention layers run in parallel

$$MultiHeadAttn(\mathbf{X}) = (\mathbf{head}_1 \oplus \mathbf{head}_2 \ldots \oplus \mathbf{head}_h)\mathbf{W}^O$$

$$\mathbf{Q} = \mathbf{X}\mathbf{W}_i^Q \; ; \; \mathbf{K} = \mathbf{X}\mathbf{W}_i^K \; ; \; \mathbf{V} = \mathbf{X}\mathbf{W}_i^V$$

$$\mathbf{head}_i = SelfAttention(\mathbf{Q},\mathbf{K},\mathbf{V})$$

Transformer Architecture

In summary…

The Pre training / Fine tuning Paradigm

Pretraining can improve NLP applications by serving as parameter initialization.

Step 1: Pretrain (on language modeling)

Lots of text; learn general things!

Step 2: Finetune (on your task)

Not many labels; adapt to the task!

… the movie was …

Pre-training for three types of architectures

Encoder-only models

- Gets bidirectional context – can condition on future!
- How do we train them to build strong representations?
- Also known as *auto-encoding models*
- For tasks that require understanding of the input, e.g sentence classification and named entity recognition
- Representatives of this family of models include:
  - ALBERT (have smaller parameter sizes when compared to the corresponding BERT models),
  - BERT,
  - DistilBERT(DistilBERT is a small, fast, cheap and light Transformer model trained by distilling Bert base)

Pre-training for three types of architectures

Decoder-only models

- Language models
- Nice to generate from; can't condition on future words
- These models are are also known as *auto-regressive models*.
- Representatives of this family of models include:
  - GPT (Generative Pre-trained Transformer), GPT-2, GPT-3 etc

Encoder-decoder models or **sequence-to-sequence models**

- For generative tasks that require an input, such as translation or summarization.
- BART (Bidirectional and Auto-Regressive Transformer)

Contextual embeddings

- Word embeddings (word2vec, GloVe) are often *pre-trained* on text corpus from co-occurrence statistics
- Problem: Word embeddings are applied in a context free manner

```
open a bank account                on the river bank

                [0.3, 0.2, -0.8, …]
```

- Solution: Train contextual représentations on text corpus

```
[0.9, -0.2, 1.6, …]                [-1.9, -0.4, 0.1, …]

open a bank account                on the river bank
```

- Given a pre trained language model and a novel input sentence, model outputs contextual embeddings for each token in the input.

Types of language modelling

- *Causal language modelling*
  - predicting the next word in a sentence in which output depends on the past and present inputs, but not the future ones
  - left-to-right
- *Masked language modeling*
  - the model predicts a masked word in the sentence
  - introduced with the BERT model
  - allows the model to see entire texts at a time, including both the right and left context.

BERT: Bidirectional Encoder Representations from Transformers

Details about BERT

- Two models were released:
  - BERT-base: 12 layers, 768-dim hidden states, 12 attention heads, 110 million params.
  - BERT-large: 24 layers, 1024-dim hidden states, 16 attention heads, 340 million params.
- Trained on:
  - BooksCorpus (800 million words)
  - English Wikipedia (2,500 million words)
- Pre training is expensive and impractical on a single GPU.
  - BERT was pre trained with 64 TPU chips for a total of 4 days.
  - (TPUs are special tensor operation acceleration hardware)
- Fine tuning is practical and common on a single GPU
  - "Pre train once, fine tune many times."

[Devlin et al., 2018]

BERT

$BERT_{BASE}$

$BERT_{LARGE}$

Model input dimension 512

BERT Pre training Task 1: masked words

- Instead of trying to predict the next word, the model learns to perform **a fill-in-the-blank task**, technically called the **cloze task e.g.**

  Please turn your homework_____.

  Please turn _____ homework in.

- Predict a random 15% of (sub)word tokens.
  - Replace input word with [MASK] 80% of the time
  - Replace input word with a random token 10% of the time
  - Leave input word unchanged 10% of the time (but still predict it!)

[Devlin et al., 2018]

BERT Pre training Task 1: masked words

Use the output of the masked word's position to predict the masked word

Randomly mask 15% of tokens

Input

Masked Language Modeling (MLM) output for BERT

$$y_i = \text{softmax}(W_V\, h_i)$$

Figure 11.5 Masked language model training. In this example, three of the input tokens are selected, two of which are masked and the third is replaced with an unrelated word. The probabilities assigned by the model to these three items are used as the training loss. (In this and subsequent figures we display the input as words rather than subword tokens; the reader should keep in mind that BERT and similar models actually use subword tokens instead.)

BERT Pretraining Task 2: Next Sentence Prediction (NSP)

- Determining the relationship between pairs of sentences
  - paraphrase detection (detecting if two sentences have similar meanings)
  - entailment (detecting if the meanings of two sentences entail or contradict each other)
  - discourse coherence (deciding if two neighboring sentences form a coherent discourse)
- To make BERT better at handling relationships between multiple sentences, the pre-training process includes an additional task: **Given two sentences (A and B), is B likely to be the sentence that follows A, or not?**

BERT Pretraining Task 2: Next Sentence Prediction (NSP)

- Two new tokens to the input representation : CLS and SEP
- **50% the second sentence of a pair was randomly selected from elsewhere in the corpus**
- Embeddings representing the first and second segments of the input are added to the word and positional embeddings to allow the model to more easily distinguish the input sentences.

BERT: Embedding layer

- The Token Embeddings layer will convert each **wordpiece** token into a 768-dimensional vector representation
- The Segment Embeddings layer only has 2 vector representations. The first vector (index 0) is assigned to all tokens that belong to input 1 while the last vector (index 1) is assigned to all tokens that belong to input 2
- Classifying whether two pieces of text are semantically similar. The pair of input text are simply concatenated and fed into the model

NSP output for BERT

$$y_i = \text{Softmax}\,(W_{NSP}\, h_i)$$

Figure 11.7 An example of the NSP loss calculation.

output vector from the final layer associated with the [CLS] token represents the next sentence prediction

BERT for different task

(a) Sentence Pair Classification Tasks: MNLI, QQP, QNLI, STS-B, MRPC, RTE, SWAG

(b) Single Sentence Classification Tasks: SST-2, CoLA

(c) Question Answering Tasks: SQuAD v1.1

(d) Single Sentence Tagging Tasks: CoNLL-2003 NER

References

- Speech and Language Processing by Daniel Jurafsky
- https://jalammar.github.io/illustrated-bert/
- https://huggingface.co/course/chapter1/4?fw=pt
- https://arxiv.org/abs/1706.03762
- https://arxiv.org/abs/1810.04805
- https://arxiv.org/abs/1406.1078
- https://arxiv.org/abs/1609.08144

Attention mechanism

- https://www.youtube.com/watch?v=OxCpWwDCDFQ&t=0s

Maths behind attention mechanism

- https://www.youtube.com/watch?v=UPtG_38Oq8o
- https://www.youtube.com/watch?v=zxQyTK8quyY&t=2077s

Best embedding for finding similarities

Best embedding for finding the next word

This embedding(s) know features of the words

- Color
- Size
- Fruitness
- Technology

This embedding knows when two words could appear in the same context

I want to buy a _____

- car
- apple
- phone

an apple and an orange

	Orange	Apple	And	An
Orange	0.4	0.3	0.15	0.15
Apple	0.3	0.4	0.15	0.15
And	0.15	0.15	0.5	0.5
An	0.15	0.15	0.5	0.5

Value matrix

=

	Orange	Apple	And	An
Orange	v_{11}	v_{12}	v_{13}	v_{14}
Apple	v_{21}	v_{22}	v_{23}	v_{24}
And	v_{31}	v_{32}	v_{33}	v_{34}
An	v_{41}	v_{42}	v_{43}	v_{44}

apple $\longrightarrow$ $0.3 \cdot$ orange
$+0.4 \cdot$ apple
$+0.15 \cdot$ and
$+0.15 \cdot$ an

apple $\longrightarrow$ $v_{21} \cdot$ orange
$+v_{22} \cdot$ apple
$+v_{23} \cdot$ and
$+v_{24} \cdot$ an

Self-attention

Scaled Dot-Product Attention

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Multi-head attention

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$
$$\text{where head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Multi-head attention

$$\mathrm{MultiHead}(Q, K, V) = \mathrm{Concat}(\mathrm{head}_1, ..., \mathrm{head}_\mathrm{h})W^O$$
$$\text{where } \mathrm{head}_\mathrm{i} = \mathrm{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$