Deep Learning

Vazgen Mikayelyan

December 8, 2020

FAST DISCOVERING THE FUTURE

Outline

Problems with RNNs

Sequential computation prevents parallelization.

Problems with RNNs

Sequential computation prevents parallelization.

Despite GRUs and LSTMs, RNNs still need attention mechanism to deal with long range dependencies path length for codependent computation between states grows with sequence.

Problems with RNNs

- Sequential computation prevents parallelization.
- Despite GRUs and LSTMs, RNNs still need attention mechanism to deal with long range dependencies path length for codependent computation between states grows with sequence.
- But if attention gives us access to any state, maybe we don't need t RNN?

Transformer

Self Attention Layer

This layer aims to encode a word based on all other words in the sequence. It measures the encoding of the word against the encodi of another word and gives a new encoding.

Self Attention Layer

This layer aims to encode a word based on all other words in the sequence. It measures the encoding of the word against the encodi of another word and gives a new encoding.

Given an embedding x, it learns three separate smaller embeddings from it   query, key and value.

Self Attention Layer

- This layer aims to encode a word based on all other words in the sequence. It measures the encoding of the word against the encodi of another word and gives a new encoding.
- Given an embedding x, it learns three separate smaller embeddings from it query, key and value.
- During the training phase, the W_q, W_k, and W_v matrices are learnt to get the query, key and value embeddings.

Self Attention Layer

Self Attention Layer

Say x_1 wants to know its value with respect to x_2. So it will `query' x_2.

Self Attention Layer

Say x_1 wants to know its value with respect to x_2. So it will `query' x_2.

x_2 will provide the answer in the form of its own `key', which can the be used to get a score representing how much it values x_1 by taking a dot product with the query. Since both have the same size, this will a single number.

Self Attention Layer

Say x_1 wants to know its value with respect to x_2. So it will `query' x_2.

x_2 will provide the answer in the form of its own `key', which can the be used to get a score representing how much it values x_1 by taking a dot product with the query. Since both have the same size, this will a single number.

Then x_1 will take all these scores and perform softmax.

Self Attention Layer

Say x_1 wants to know its value with respect to x_2. So it will `query' x_2.

x_2 will provide the answer in the form of its own `key', which can then be used to get a score representing how much it values x_1 by taking a dot product with the query. Since both have the same size, this will a single number.

Then x_1 will take all these scores and perform softmax.

This step will be performed with every word.

Self Attention Layer

Self Attention Layer

x_1 will now use this score and the `value' of the corresponding word get a new value of itself with respect to that word.

Self Attention Layer

x_1 will now use this score and the `value' of the corresponding word get a new value of itself with respect to that word.

If the word is not relevant to x_1 then the score will be small and the corresponding value will be reduced a factor of that score and simila the signi cant words will get their values bolstered by the score.

Self Attention Layer

Self Attention Layer

Finally, the word x1 will create a new `value' for itself by summing up the values received. This will be the new embedding of the word.

Self Attention

$$\text{Attention}(q; K; V) = \sum_i \frac{e^{q \cdot k_i}}{\sum_j e^{q \cdot k_j}} v_i$$

where

inputs: a query q and a set of key-value (K-V) pairs to an output,

Self Attention

$$\text{Attention}(q; K; V) = \sum_i \frac{e^{q \cdot k_i}}{\sum_j e^{q \cdot k_j}} v_i$$

where

inputs: a query q and a set of key-value (K-V) pairs to an output,

query, keys, values and output are all vectors,

Self Attention

$$\text{Attention}(q; K; V) = \sum_i \frac{e^{q\, k_i}}{\sum_j e^{q\, k_j}} v_i$$

where

inputs: a query q and a set of key-value (K-V) pairs to an output,

query, keys, values and output are all vectors,

output is a convex combination of values,

Self Attention

$$\text{Attention}(q; K; V) = \sum_i \frac{e^{q \cdot k_i}}{\sum_j e^{q \cdot k_j}} v_i$$

where

- inputs: a query q and a set of key-value (K-V) pairs to an output,
- query, keys, values and output are all vectors,
- output is a convex combination of values,
- weight of each value is computed by an inner product of query and corresponding key,

Self Attention

$$\text{Attention}(q; K; V) = \sum_i \frac{e^{q \cdot k_i}}{\sum_j e^{q \cdot k_j}} v_i$$

where

- inputs: a query q and a set of key-value (K-V) pairs to an output,
- query, keys, values and output are all vectors,
- output is a convex combination of values,
- weight of each value is computed by an inner product of query and corresponding key,
- queries and keys have the same dimensionality d_k, values have d_v.

When we have multiple queries q, we stack them in a matrix Q:

$$\text{Attention}(Q;K;V) = \text{Softmax}\ QK^T\ \ V:$$

Self Attention

When we have multiple queries q, we stack them in a matrix Q:

$$\text{Attention}(Q; K; V) = \text{Softmax}\left(QK^T\right) V:$$

Problem: as d_k gets large, the variance of $q^T k$ increases, thus some values inside the softmax gets large, hence its gradients gets smalle

Self Attention

When we have multiple queries q, we stack them in a matrix Q:

$$\text{Attention}(Q; K; V) = \text{Softmax}\left(QK^T\right) V:$$

Problem: as d_k gets large, the variance of $q^T k$ increases, thus some values inside the softmax gets large, hence its gradients gets small

Solution:

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

Multi-Head Attention Layer

Multihead attention assumes that all inputs and outputs have the same length d_{model}. If inputs hasn't length d_{model}, we pass it through one fully connected layer.

Multi-Head Attention Layer

Multihead attention assumes that all inputs and outputs have the same length d_{model}. If inputs hasn't length d_{model}, we pass it through one fully connected layer.

$$\text{Multihead} = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)\, W^o$$

where

$$\text{head}_i = \text{Attention}\left(xW_i^Q, xW_i^K, xW_i^V\right)$$

$$W_i^Q \in \mathbb{R}^{d_{model} \times d_k}, W_i^K \in \mathbb{R}^{d_{model} \times d_k}, W_i^V \in \mathbb{R}^{d_{model} \times d_v}, W^O \in \mathbb{R}^{hd_v \times d_{model}}$$

Masked Multi-Head Attention Layer

At any position, a word may depend on both the words before it as well as the ones after it.

Masked Multi-Head Attention Layer

- At any position, a word may depend on both the words before it as well as the ones after it.
- This is why in the self-attention layer, the query was performed with all words against all words.

Masked Multi-Head Attention Layer

- At any position, a word may depend on both the words before it as well as the ones after it.
- This is why in the self-attention layer, the query was performed with all words against all words.
- But at the time of decoding, when trying to predict the next word in the sentence, logically, it should not know what are the words which are present after the word we are trying to predict.

Masked Multi-Head Attention Layer

- At any position, a word may depend on both the words before it as well as the ones after it.
- This is why in the self-attention layer, the query was performed with all words against all words.
- But at the time of decoding, when trying to predict the next word in the sentence, logically, it should not know what are the words which are present after the word we are trying to predict.
- This is why the embeddings for all these are masked by multiplying with 0.

Feed-Forward Network

this part is a position free neural network, which consists of two fully connected layers with a ReLU activation in between:

$$FFN(x) = W_2 \ ReLU(W_1 x + b_1) + b_2$$

Encoder-Decoder Architecture

Encoder-Decoder Architecture

Encoder-Decoder Architecture

Transformer

Outline

Dilated/Atrous Convolution

De nition 1

Let $F : Z^2 \; ! \;\; R$ be a discrete function. Let $_r = [\;\; r ; r] \setminus \;\; Z^2$ and let $k : \;\; _r \; ! \;\; R$ be a discrete lter of size $(2r + 1)^2$. The discrete convolution operator can be de ned as

$$(F \;\; k)(p) = \sum_{s + t = p} F(s)\, k(t)$$

Dilated/Atrous Convolution

De nition 1

Let $F : Z^2 \,!\quad R$ be a discrete function. Let $_r = [\quad r ; r] \backslash\ Z^2$ and let $k : \ _r \,!\quad R$ be a discrete lter of size $(2r + 1)^2$. The discrete convolution operator can be de ned as

$$(F \quad k)(p) = \sum_{s+t=p} F(s)\,k(t)$$

De nition 2

Let $F : Z^2 \,!\quad R$ be a discrete function. Let $_r = [\quad r ; r] \backslash\ Z^2$ and let $k : \ _r \,!\quad R$ be a discrete lter of size $(2r + 1)^2$. The discrete l-dilated convolution operator $_l$ can be de ned as

$$(F \ _l\, k)(p) = \sum_{s+lt=p} F(s)\,k(t)$$

Dilated/Atrous Convolution

Dilated/Atrous Convolution

1D Dilated Convolution

Convolution as a Matrix Operation

Convolution as a Matrix Operation

This linear operation takes the input matrix attened as a 16-dimensiona vector and produces a 4-dimensional vector that is later reshaped as the 2   2 output matrix.

Transposed Convolution (stride=0)

Transposed Convolution (stride=0)

Transposed Convolution (stride=1)

Transposed Convolution (stride=1)