Deep Learning

Vazgen Mikayelyan

YSU, Krisp

November 13, 2019

Outline

1. Transformers
2. Dilated and Transposed Convolutions

Problems with RNNs

- Sequential computation prevents parallelization.

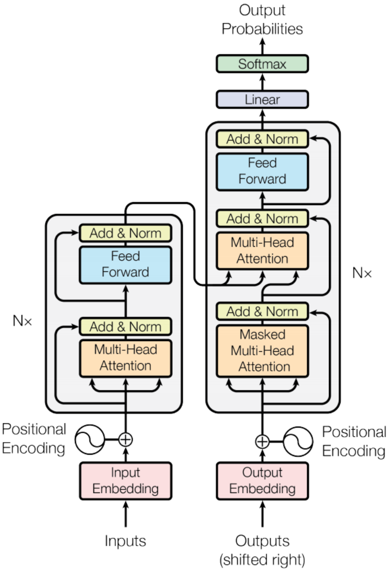
Problems with RNNs

- 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 the 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 encoding of another word and gives a new encoding.

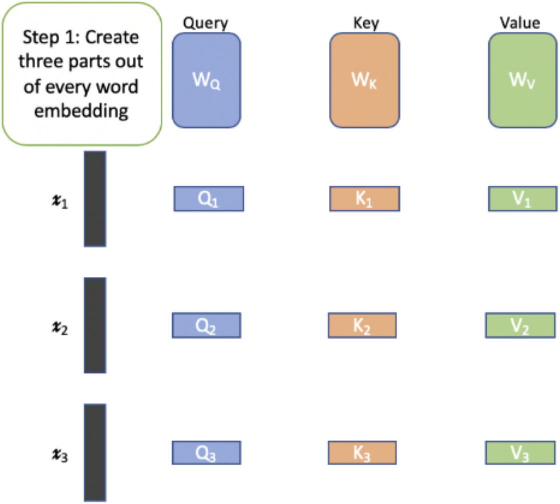
Self Attention Layer

- This layer aims to encode a word based on all other words in the sequence. It measures the encoding of the word against the encoding 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 encoding 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 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 be a single number.

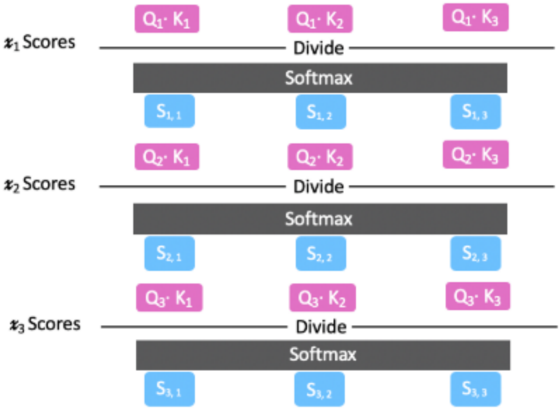
Self Attention Layer

- 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 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 be 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 be 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

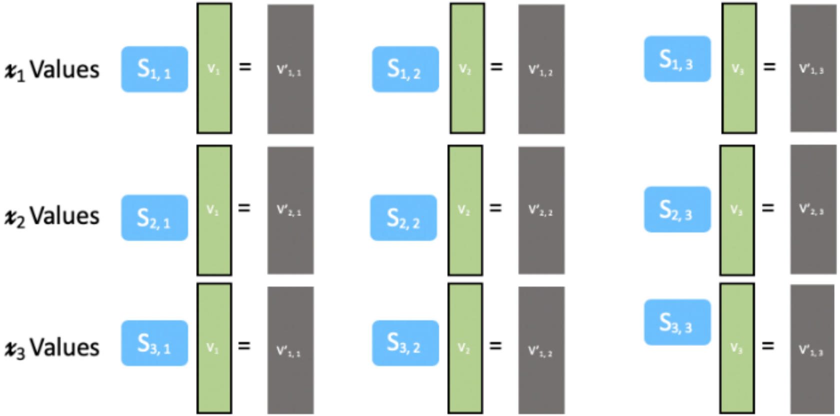


- x_1 will now use this score and the 'value' of the corresponding word to get a new value of itself with respect to that word.

Self Attention Layer

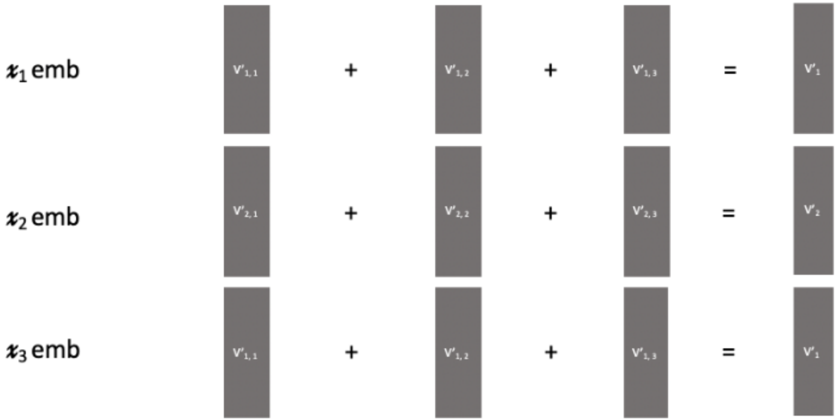
- x_1 will now use this score and the 'value' of the corresponding word to get a new value of itself with respect to that word.
- If the word is not relevant to x_1 then the score will be small and the corresponding value will be reduced a factor of that score and similarly the significant words will 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

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

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,

Self Attention

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

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

$$\text{Attention}\left(q, K, V\right) = \sum_i \frac{e^{q \cdot k_i}}{\sum\limits_j e^{q \cdot k_j}} v_i$$

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.

Self Attention

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

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

Self Attention

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

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

- 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 smaller.

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 smaller.
- 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.

$$\text{Multihead}\left(Q, K, V, h\right) = \text{Concat}\left(\text{head}_1, \dots, \text{head}_h\right) W^o$$

where

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

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

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

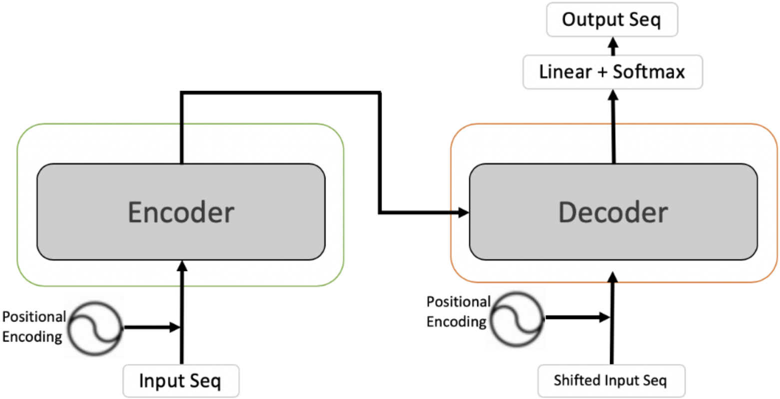
- At any position, a word may depend on both the words before it as well as the ones after it.
- This is why in the self-attention layer, the query was performed with all words against all words.
- But at the time of decoding, when trying to predict the next word in the sentence, logically, it should 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

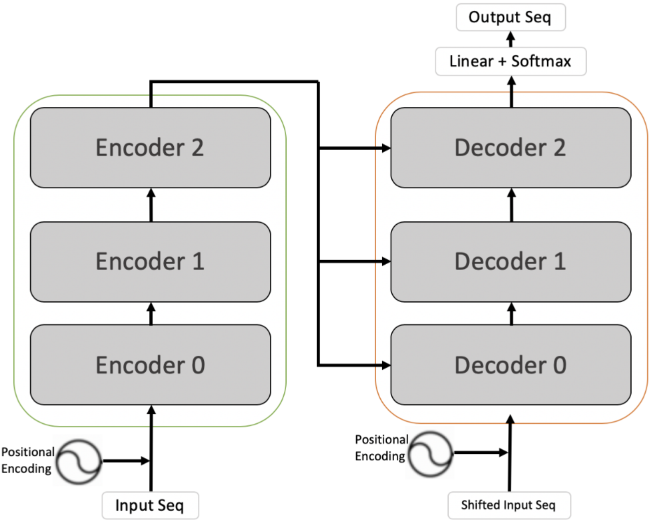
this part is a position free neural network, which consists of two fully connected layers with a ReLU activation in between:

$$\mathrm{FFN}(x) = W_2 \cdot \mathrm{ReLU}(W_1 x + b_1) + b_2$$

Encoder-Decoder Architecture



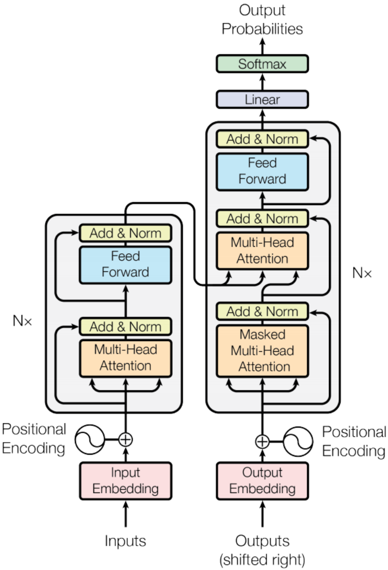
Encoder-Decoder Architecture



Encoder-Decoder Architecture

Queries	Decoder Input
Keys	Encoder output
Values	Encoder output

Transformer



Outline

Dilated/Atrous Convolution

Definition 1

Let $F : \mathbb{Z}^2 \rightarrow \mathbb{R}$ *be a discrete function. Let* $\Omega_r : [-r, r] \cap \mathbb{Z}^2$ *and let* $k : \Omega_r \rightarrow \mathbb{R}$ *be a discrete filter of size* $(2r+1)^2$*. The discrete convolution operator* $*$ *can be defined as*

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

Dilated/Atrous Convolution

Definition 1

Let $F : \mathbb{Z}^2 \to \mathbb{R}$ be a discrete function. Let $\Omega_r : [-r, r] \cap \mathbb{Z}^2$ and let $k : \Omega_r \to \mathbb{R}$ be a discrete filter of size $(2r+1)^2$. The discrete convolution operator $*$ can be defined as

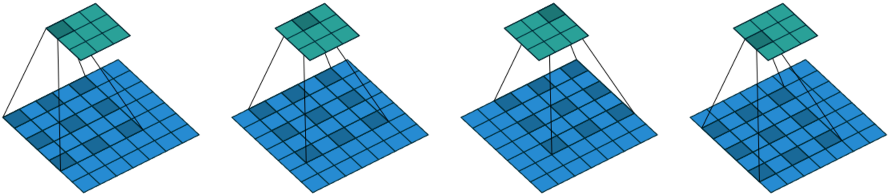
$$(F * k)(p) = \sum_{s+t=p} F(s)\, k(t)$$

Definition 2

Let $F : \mathbb{Z}^2 \to \mathbb{R}$ be a discrete function. Let $\Omega_r : [-r, r] \cap \mathbb{Z}^2$ and let $k : \Omega_r \to \mathbb{R}$ be a discrete filter of size $(2r+1)^2$. The discrete l-dilated convolution operator $*_l$ can be defined as

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

Dilated/Atrous Convolution



1D Dilated Convolution

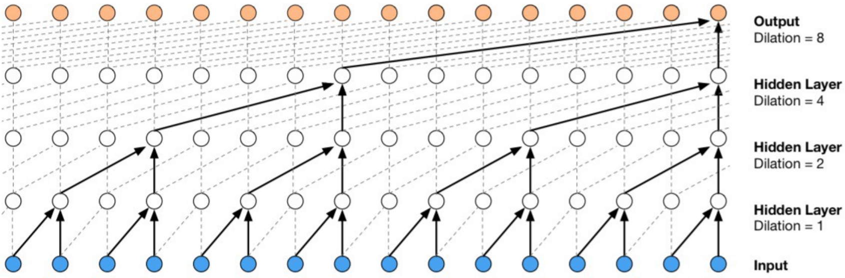


Figure 3: Visualization of a stack of *dilated* causal convolutional layers.