Introduction to Transformers

D. Gueorguiev 6/24/22

Table of Contents

Preliminaries

Note 0: Knowledge of feed-forward neural networks and the back-propagation algorithm is assumed throughout this document.

It is important to understand how the *Transformer architecture* fits historically as a modeling tool. For the purpose first we will look into other architectures and models which evolved earlier compared to the *Attention mechanism* and the *Transformer architecture*.

Language Modeling

Tools for Language Modeling:

- Sequential Models
- Probabilistic Models e.g. *Latent Semantic Analysis* (LSA)

Besides used in Language Modeling the *Sequential models* are widely used in the modeling and prediction of Time-Series, Images, and Voice. Thus, we will start with a discussion on various architectures for Sequential Models which precede the *Transformer architecture*.

Sequential Modeling

In general, we build ML models where we have data points which are usually uncorrelated, and no order relation can be imposed on them.

Sequential models: In many cases such as language, voice, and time-series data a data point is dependent on a set of other data points which have already been processed. We call such stream of data points a stream of *sequential data*. Machine learning models which accept input or create an output sequence of data points are known as *sequential models*.

Previous tools for sequence modeling:

- Recurrent Neural Networks
- Long Short Term Memory
- Gated Recurrent Neural Nets

Notes on RNN (Recurrent Neural Networks)

What is RNN?

RNN is a Neural network with at least one cycle. If the network contains cycle the computation is not uniquely defined by the interconnection pattern and the *temporal dimension* must be considered. When the output of one neural node is fed back to the same node we are dealing with recursive computation. We must define what we expect from the network – is the fixed point of the recursive evaluation the desired result or one of the intermediate computations? We can assume that every computation takes a fixed amount of time and can be expressed as a certain number of time units. If the inputs of a neural node have been sent at time t, its output will be available at time $t + 1$. A recursive computation can be stopped after predetermined number of steps and the last output will be considered the result of the recursive computation.

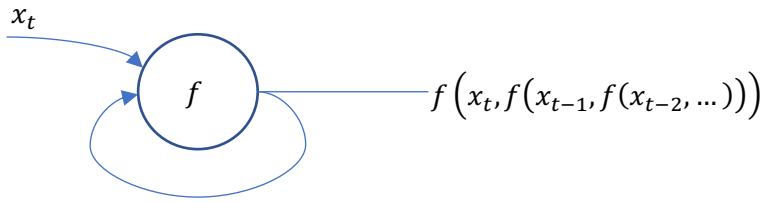


Figure 1: Node with a loop

Continuous vs Epochwise mode

Two approaches to operate and train an RNN:

- *Epochwise operation*

The network is run from a given start time until a given stopping time is reached. After reaching the stopping time the network is reset in its *initial state* for the next epoch. It is not essential that all of *the state* at the beginning of each epoch is the same. The important aspect of the *Epochwise operation* is the starting state of a new Epoch is not causally related to the ending state of the previous epoch. Thus, every epoch serves as a boundary through which learning credit cannot pass. The purpose of the epoch boundaries is to make sure that activity from one epoch is causally related to activity in another later epoch. Note that the notion of epoch is defined in a loose sense indicating only that the boundaries are present between an interval(s) and an interval of past activity is separated by boundary from the activities after the boundary.

This allows us to introduce the notion of *batch training* distinguished from the notion of *incremental training*. The difference between the batch training vs incremental training is in when the network

weights are updated. With the batch training approach weights are updated only after presenting a complete set of training examples. With the incremental approach the weights are updated after presenting each training example.

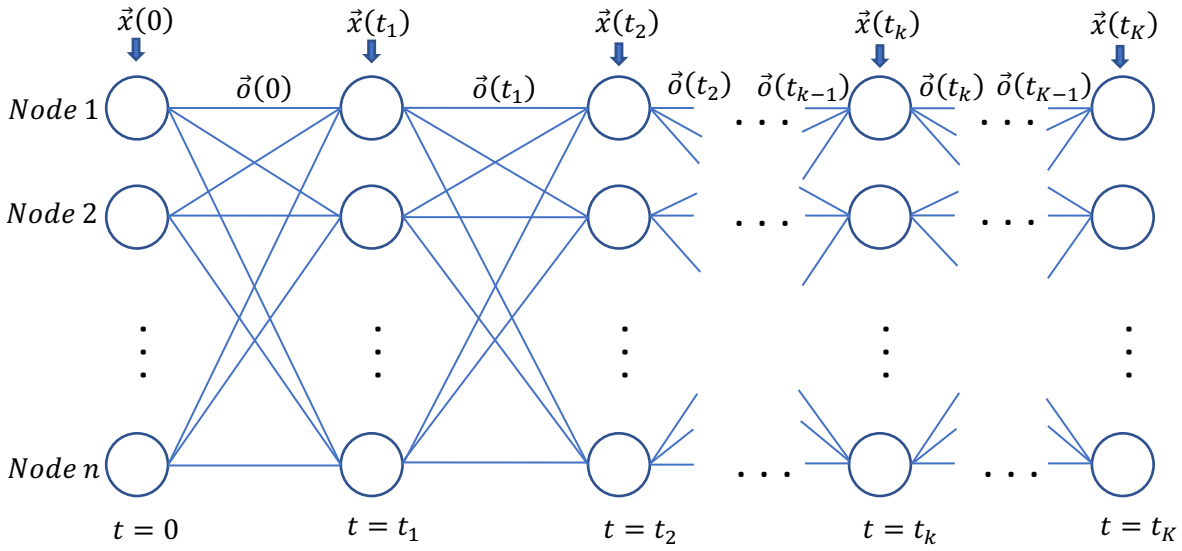
- *Continuous operation*

No manual state resets nor any barriers against the flow of training credit are imposed in the network. Continuous operation makes sense when online learning is required.

Backpropagation through time (BPTT)

This paragraph follows the notation introduced by Rojas and borrows some of the diagrams introduced in his work.

For simplicity, let us consider a finite number of iterations only. Assume that a network of n computing nodes is fully connected and that w_{ij} is the weight associated with the edge from node i to node j. We can unfold the network at times t_1, t_2, …, t_K transforming the original RNN into a feed-forward network with k stages of computation. At each discrete time $t_i = t_{i-1} + \Delta t, i = 1..K$ an external input $\vec{x}(t_i)$ is fed into the network and the outputs $\left(o_1(t_i), o_2(t_i), \dots, o_n(t_i)\right)$ of all computing nodes are recorded. We will denote vector of all network outputs at time t_i with $\vec{o}(t_i) = \left(o_1(t_i), o_2(t_i), \dots, o_n(t_i)\right)$. We



assume that $\vec{o}(0) = \vec{0}$ i.e. all network outputs at the initial moment are zeros. The unfolded network is depicted below:

Figure 2: Backpropagation through time

This unfolding strategy which converts the RNN into a feed-forward network in order to apply the classic back-propagation algorithm is called *Back-Propagation Through Time* (BPTT).

Let us denote by W the $n \times n$ matrix with the network weights $w_{j,k}$. Let us denote by W_0 the $m \times n$ matrix of interconnections between m input sites and n units. The feed-forward step is computed in the usual manner with a feed-forward network. At time $t_0 = 0$ we feed the transformed network with the m dimensional external input $\vec{x}(0)$. At each discrete time t_i there are given the network state $\vec{o}(t_i)$ (n-dimensional row vector) and the vector of derivatives of the activation function at each node:

$$\vec{o}'^{(t_i)} = \left(\frac{\partial f_1}{\partial y}\left(\vec{x}(t_i)\right), \frac{\partial f_2}{\partial y}\left(\vec{x}(t_i)\right), \ldots, \frac{\partial f_n}{\partial y}\left(\vec{x}(t_i)\right)\right) \qquad (1)$$

In (1) we have:

$$y_j(t_i) = w_{j,k}x_k(t_i) \qquad (2)$$

where $y_j(t_i)$ is the input to the activation function of the j-th node at time t_i.
Recall, we need the derivative of the activation function at each node in order to compute the back-propagated error when we are doing the back-propagation step. Refer to the figures below:

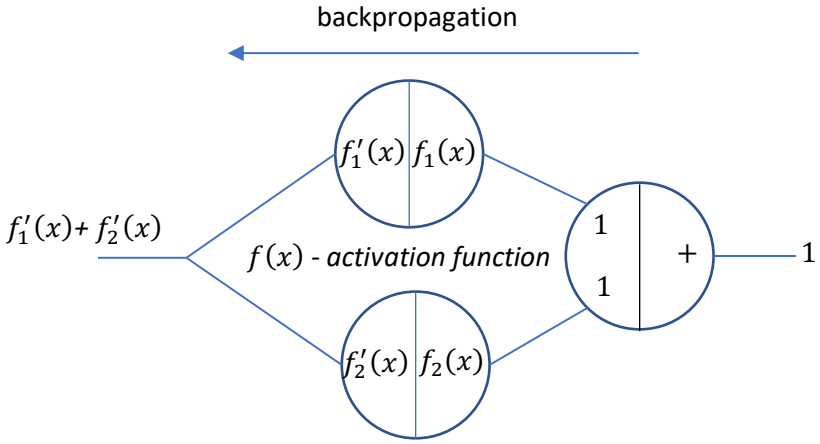


Figure 3: Result of the backpropagation step

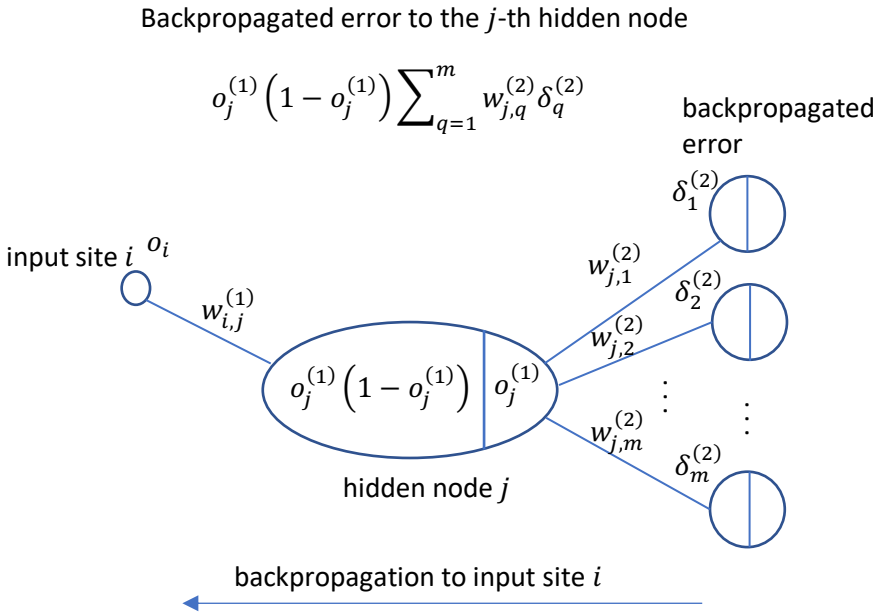


Figure 4: Backpropagation from extended output layer through a hidden node to input site i

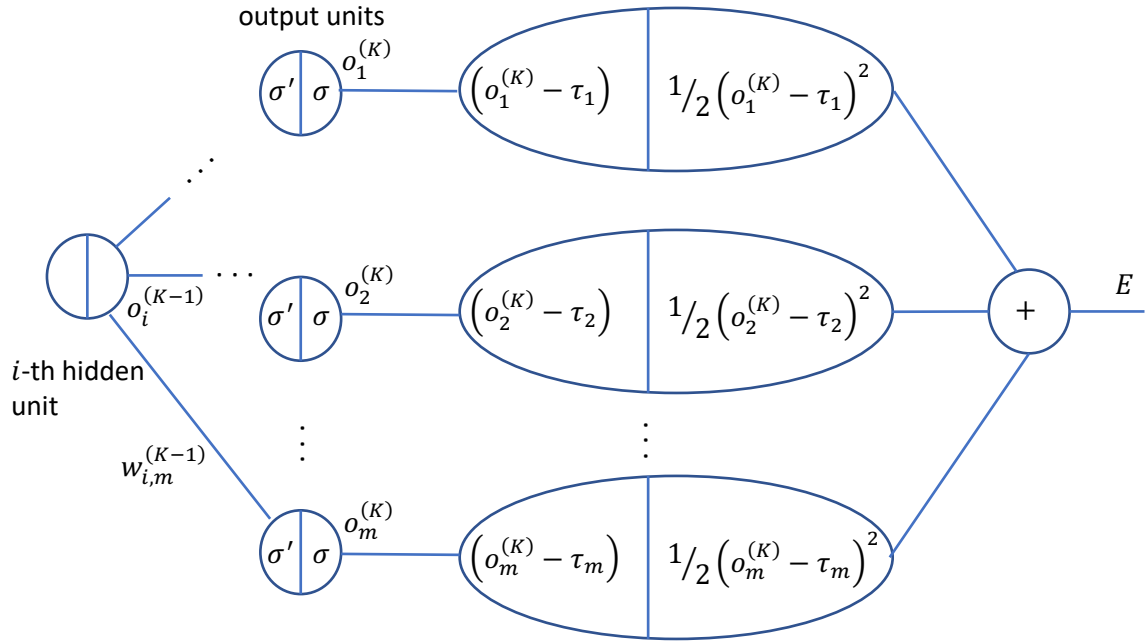
The error of the network depicted on Figure 2 can be measured after each discrete time moment t_i if a sequence of values is to be produced or just after the final moment of time t_K if only the final output is of importance. The error vector $\vec{e}$ between the m-dimensional target $\vec{\tau}$ and the output of the network $\vec{o}(t_K)$ is given with:

$\vec{e} = (\vec{o}(t_K) - \vec{\tau})^T$ (3)

which is n-dimensional column vector.

As with feed forward networks we define the network error function E to be:

$E = \frac{1}{2}\|\vec{e}\|^2 = \frac{1}{2}\sum_{i=1}^{m}(o_i(t_K) - \tau_i)^2$ (4)



In the Figure above σ denotes the activation function of the nodes. Usual choice for σ is the sigmoid error function but other choices can be used when they provide clear benefits e.g. *Rectified Linear Unit* (ReLU).

Thing to consider: each weight of the network is present at each stage of the unfolded network (Figure 2).

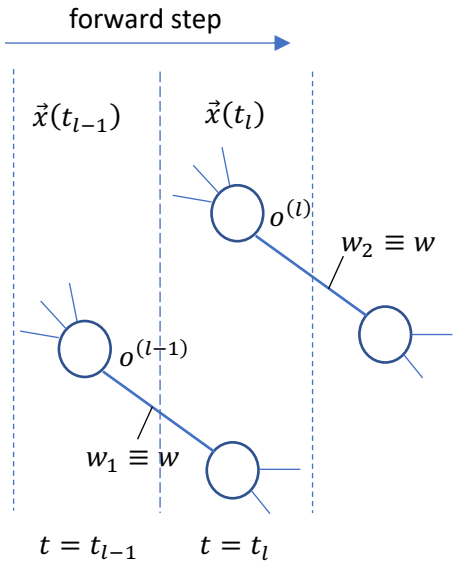
Theorem: Any network with repeated weights can be transformed into a network with unique weights.

Let us consider an unfolded feed-forward network with structure as the one shown in

Figure 5.

Weight w exists in multiple different stages of the unfolded feed-forward network and it receives different input in each stage. In the depicted on

Figure 5 stages the inputs are $\vec{o}(t_{l-2})$ and $\vec{o}(t_{l-1})$ accordingly.



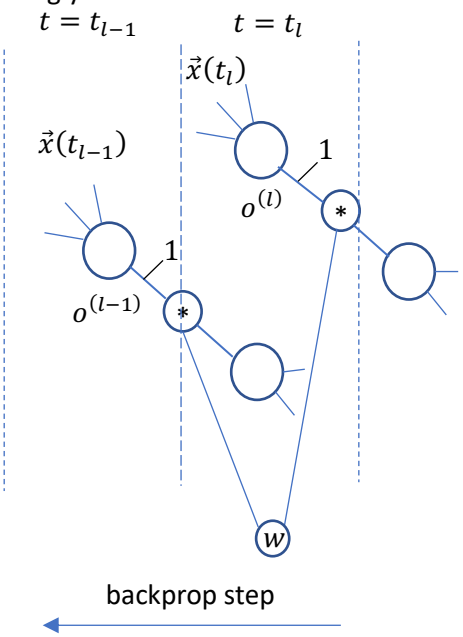


Figure 5: A duplicated weight in a network

Figure 6: Transformation guaranteeing unique weights

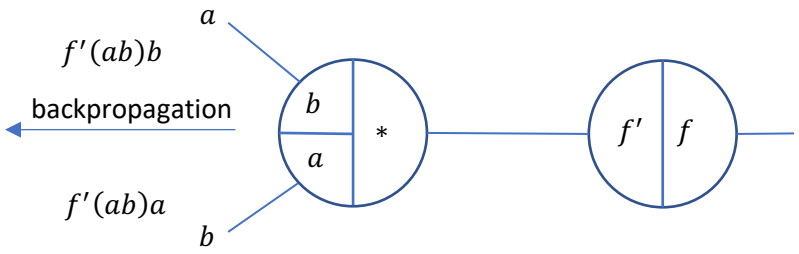


Figure 7: Multiplication as integration function

The network on

Figure 5 can be transformed as shown on Figure 6. The transformed network is indistinguishable from the original network with duplicated weight from the viewpoint of the result it computes. Note that the two edges associated with weight w previously now have weight 1 and a multiplication is performed by two additional units in the middle of the edges. In the transformed network w appears only once and we can perform backpropagation as usual. There are two groups of paths – one coming from the first multiplier (depicted with $*$) to w and the ones coming from the second. This means we can perform backpropagation as usual in the original network. Let us find the partial derivative of the error function E with the respect to the weight w. At the first edge we obtain $\partial E / \partial w_1$, at the second $\partial E / \partial w_2$ and since w_1 is the same variable as w_2 the desired partial derivative is:

$$\frac{\partial E}{\partial w} = \frac{\partial E}{\partial w_1} + \frac{\partial E}{\partial w_2} \quad (5)$$

The partial derivative can be expressed in terms of the backpropagated error and the output feed into the current node from the previous stage (Rojas, Chapter 7). Thus, we have:

$$\frac{\partial E}{\partial w_1} = \delta^{(l)} o^{(l-1)} \text{ and } \frac{\partial E}{\partial w_2} = \delta^{(l+1)} o^{(l)} \quad (6)$$

Here $\delta^{(l)}$ and $\delta^{(l+1)}$ denote the backpropagation errors for layer l and $l+1$ accordingly.

Note that the relation between the backprop error $\delta^{(l)}$ of a node in layer l and the backprop error $\delta^{(l+1)}$ of the same node in layer $l+1$ can be inferred from the general recurrence relation for the backprop errors in adjacent layers (Rojas, Chapter 7):

$$\delta_j^{(l)} = o_j^{(l)} \left(1 - o_j^{(l)}\right) \sum_{q=1}^{m} w_{jq}^{(l+1)} \delta_q^{(l+1)} \quad (7)$$

The partial derivates for the weight coefficients corresponding to layer l are given with:

$$\frac{\partial E}{\partial w_{ij}^{(l)}} = \delta_j^{(l)} o_j^{(l-1)} \quad (8)$$

In case the layer $l+1$ is the output layer we have:

$$\delta_j^{(l+1)} = o_j^{(l+1)} \left(1 - o_j^{(l+1)}\right) \left(o_j^{(l+1)} - \tau_j\right) \quad (9)$$

The partial derivatives for the weights on the output layer are given with:

$$\frac{\partial E}{\partial w_{ij}^{(l+1)}} = \delta_j^{(l+1)} o_j^{(l)} \quad (10)$$

Thus, extending (5) for the general case we can write the relation for the partial derivative of the network error function E with respect to the network weights $w_{i,j}$:

$$\frac{\partial E}{\partial w_{ij}} = \sum_{k=1}^{K} \frac{\partial E}{\partial w_{ij}^{(k)}} \quad (11)$$

In (9) the quantity τ_j denotes the j-th target value.

The correction of the weights in case of duplicate weight shared between the same node which appears in adjacent layers will be performed as:

$$\Delta w_{ij} = \sum_{k=1}^{K} \Delta w_{ij}^{(k)} \quad (12) \text{ where}$$

$$\Delta w_{ij}^{(l)} = -\gamma \delta^{(l)} o^{(l-1)} \therefore \Delta w_{ij} = -\gamma \sum_{k=1}^{K} \delta^{(k)} o^{(k-1)} \quad (13)$$

Here γ represents the learning rate.

Let us formally describe the BPTT using matrix notation
The backpropagated error at time t_K is given with :

$$\vec{\delta}(t_K) = D(t_K)\vec{e}(t_K) \quad (14)$$

Here $D(t_K)$ is a diagonal $n \times n$ matrix in which the main diagonal is occupied by the elements of $\vec{o}'^{(t_K)}$ given with (1). In general, for any moment t_k, we can write:

$$D(t_k) = \begin{pmatrix} o_1^{(k)}\left(1 - o_1^{(k)}\right) & 0 & \cdots & 0 \\ 0 & o_2^{(k)}\left(1 - o_2^{(k)}\right) & \cdots & 0 \\ \vdots & \vdots & \cdots & 0 \\ 0 & 0 & \cdots & o_n^{(k)}\left(1 - o_n^{(k)}\right) \end{pmatrix}$$

The error vector $\vec{e}(t_K)$ at moment t_K is given with (3).

Let us denote with W the matrix of weights w_{ij}, $i, j = 1..n$ where w_{ij} is the weight associated with the edge from node i to node j. With W_0 we denote the $m \times n$ matrix of interconnections between m input sites and n units. With $\vec{x}(0)$ we denote the m-dimensional external input vector.

//TODO

Real Time Recurrent Learning (RTRL)

The RTRL algorithm does not require error propagation. All the information to compute the gradient is collected as the input stream is presented to the network.

//TODO

Notes on LSTM (Long Short Term Memory)

In the BPTT step we calculate the partial derivates at each weight of the network. The RNNs are deep networks where the partial derivatives are formed as series of products as we already know. These series of products can bring the overall value of a partial derivative in an early moment of time to negligibly small value which will be useless to correct error in the weights. This leads us to the *Vanishing Gradient Problem* with RNNs.

Vanishing and Exploding Gradient Problem with RNNs

The usual choice in RNN for activation function is the sigmoid activation function:

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

The derivative of the sigmoid function is given with:

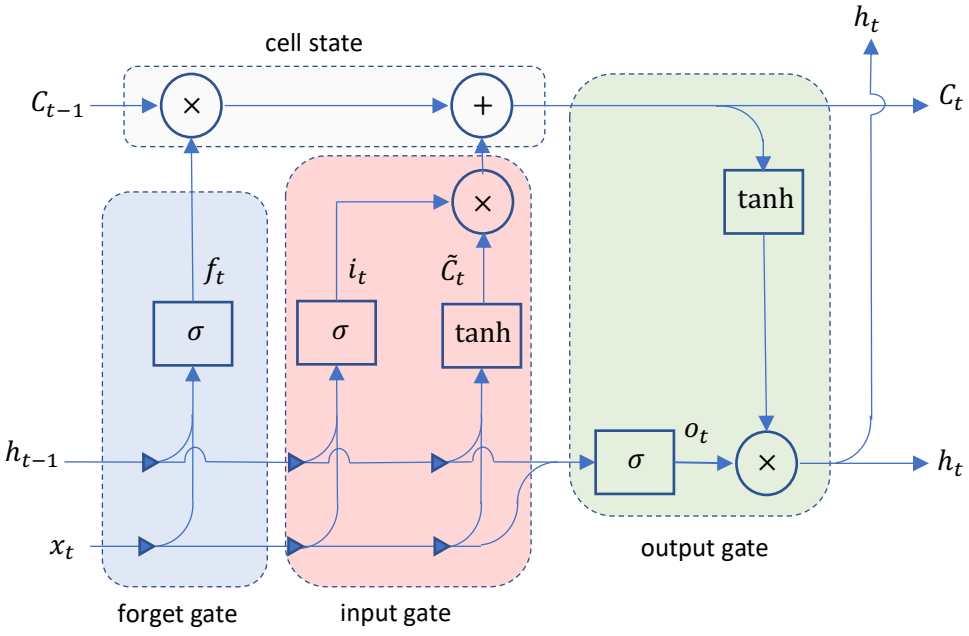
$$\sigma'(x) = \sigma(x)\big(1 - \sigma(x)\big)$$

Obviously $s'(x)$ is used in the product series of the backpropagated error in the BPTT step. Refer to (7) and (9) for details. Succinct analysis on the problem is presented in (Arbel, 2018) (link here). Indeed, it all boils down to how we calculate the partial derivative of the error function with respect to the network weights as a sum of the partial derivatives with respect the network weights for each moment of time (refer to (11)). Because each term in the right-hand side of (11) is computed with the recurrence relation (7) we end up with series of products where each term looks like:

$$o_{q_{l+1}}^{(l+1)}\left(1 - o_{q_{l+1}}^{(l+1)}\right)\left(o_{q_{l+1}}^{(l+1)} - \tau_{q_{l+1}}\right)\prod_{j=0}^{l} o_{q_j}^{(j)}\left(1 - o_{q_j}^{(j)}\right) w_{q_j q_{j+1}}^{(j+1)}$$

Those terms end up in the expression for the weight updates given with (13) and cause numerical instability leading to vanishing or exploding gradients.

Basic LSTM Architecture



LSTM (long short term memory) solves the vanishing gradient problem in backpropagation. LSTMs use gating mechanism that controls the memorizing process. Information in LSTMs can be stored, written, or read via gates that open and close. These gates store the memory in analog format, implementing element-wise multiplication by sigmoid ranges between 0 and 1.
A common LSTM unit is composed of a cell, an input gate, an output gate and forget gate.

Sigmoid function

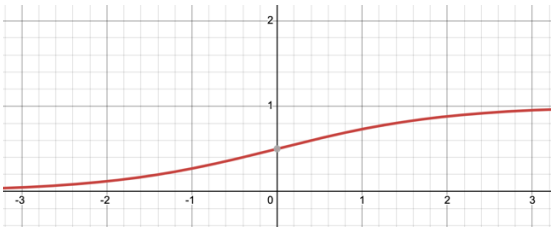
Hyperbolic tangent function

Point-by-point multiplication

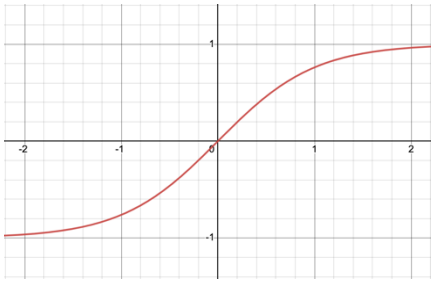
Point-by-point addition

Tanh: non-linear activation function which regulates the values flowing through the network, maintaining values between -1 and 1. To avoid information fading, a function is needed whose second derivative can survive for longer. There might be a case where some values become enormous, further causing values to be insignificant.

Sigmoid: another non-linear activation function which is used by the gates. Unlike tanh sigmoid maintains values between 0 and 1. It helps the network to *update* or *forget* data. If the multiplication results in 0 the information is considered forgotten. Similarly, the information is preserved if the value is 1.



Sigmoid



tanh

The cell remembers values over arbitrary time intervals and the three gates regulate the information into and out of the cell.

Inputs of the cell:
$x(t)$ – token at timestamp t
$h(t-1)$ – previous hidden state
$c(t-1)$ – previous cell state

Outputs of the cell:
$h(t)$ - updated hidden state
$c(t)$ - current cell state

Forget Gate

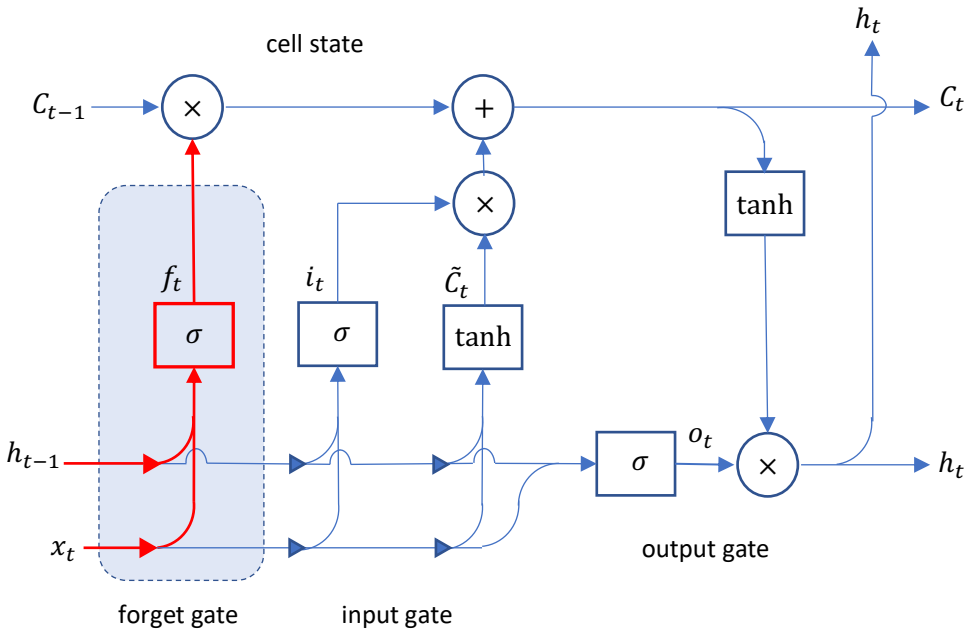


Figure 8: Forget gate operation

$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$
f_t – forget gate at timestamp t
x_t – input at timestamp t
h_{t-1} – previous hidden state
W_f – weight matrix between hidden state and input gate
b_f – connection bias at t

The forget gate decides which information needs attention and which can be ignored. The information from the current input $x(t)$ and hidden state $h(t-1)$ are passed through the sigmoid function. Sigmoid generates values in the range $(0,1)$. It concludes whether the part of the old output is necessary by producing a value closer to 1. The value of $f(t)$ will be used by the cell for point-by-point multiplication.

Input Gate

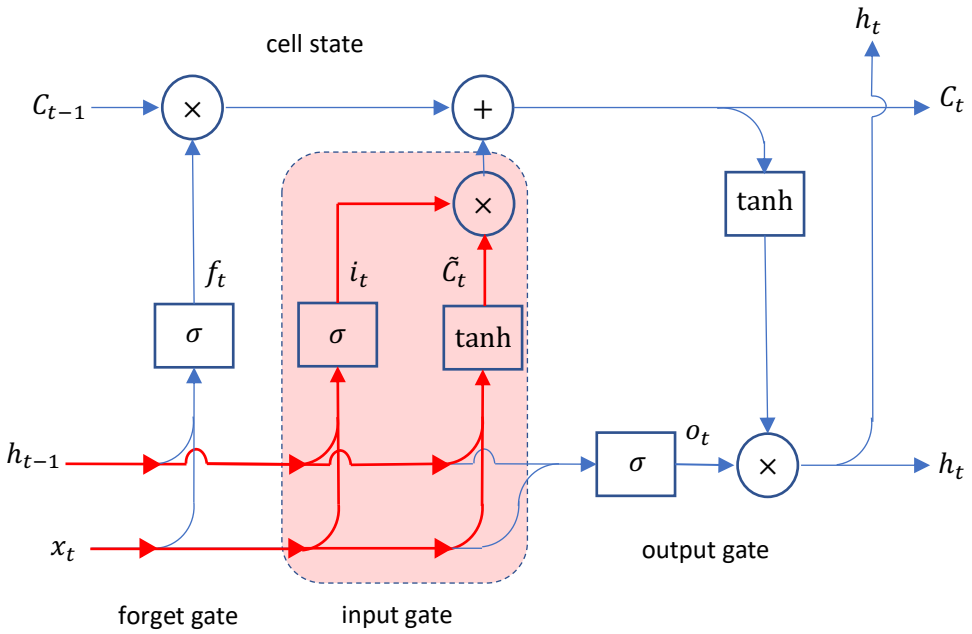


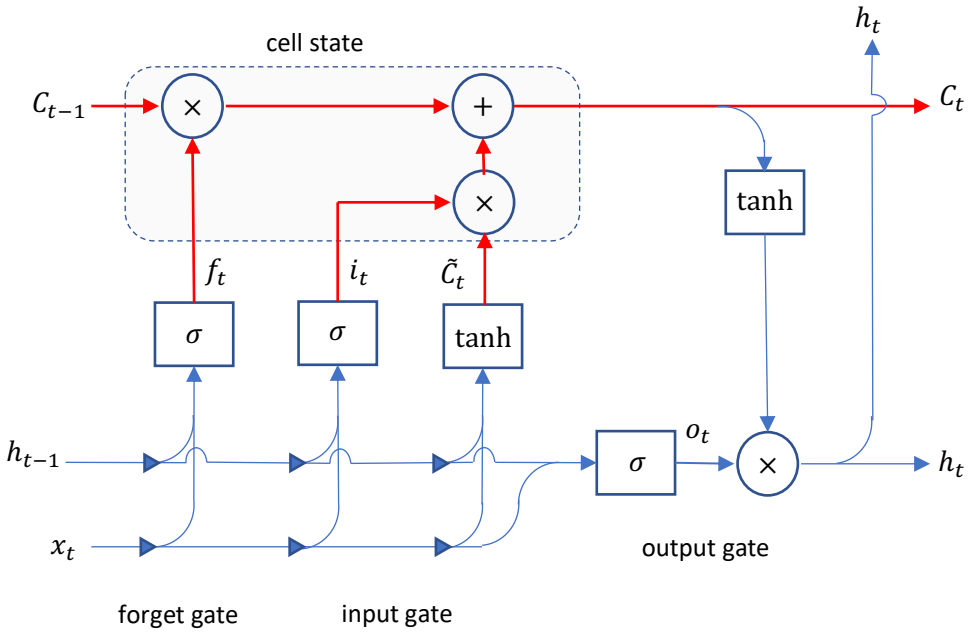
Figure 9: Input gate operation

$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$
$\tilde{C}_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c)$

i_t – input gate at moment t
W_i – weight matrix of sigmoid operator between input gate and output gate
b_i – bias vector w.r.t. W_i at moment t
$\tilde{C}_t$ – value generated by $\tanh$
W_c – weight matrix of $\tanh$ operator between cell state operation and network output
b_c – bias vector w.r.t. W_c at moment t

The input gate performs the following updates to update cell status.
First, the current state $x(t)$ and the previously hidden state $h(t-1)$ are passed into the second sigmoid function. The values are transformed between 0 (important) and 1 (not important). Next, the same information of the hidden state and current state will be passed through the $\tanh$ node. To regulate the network the $\tanh$ operator will create a vector $\tilde{C}_t$ with all the possible values between -1 and 1. The output values generated form the activation functions are ready for point-by-point multiplication.

Cell State



C_t – cell state information at discrete time moment t

//TODO

Notes on Gated Recurrent Neural Nets

//TODO

Latent Semantic Analysis (LSA)

LSA belongs to the family of *Probabilistic Methods* for analyzing and modeling the structure of the language.

//TODO

Encoder-Decoder Architectures

Became popular after 2014 , main article on encoder-decoder architectures is from Ilya Sutskever et al. Encoder-Decoder architectures were historically used as machine translation models in Natural Language Processing with encoder and decoder for each language or involve language specific encoder applied to each sentence whose outputs are then compared. An encoder neural network reads and encodes a source sentence in a fixed length vector. A decoder then outputs a translation of the encoded vector. The whole encoder-decoder system which consists of the encoder and the decoder for a language pair is jointly trained to maximize the probability of correct translation given a source sentence.

The Motivation for the Encoder-Decoder Architectures: A drawback of the classical Deep Learning architectures is that DNNs can be applied to problems whose inputs and targets can be sensibly encoded with vectors of fixed dimensionality. It is significant limitation as many important problems are best expressed with sequences whose lengths are not known a-priori. For example, speech recognition and machine translation are sequential problems.

//TODO

The Attention Mechanism

This paragraph follows the discussion on the Attention Mechanism in Galassi et al.
In many NLP problems the components of the text source have different relevance for the task which is being performed.

The Motivation for the Attention Mechanism: A potential issue with the traditional neural network-based encoder-decoder approach is that a neural network needs to be able to compress all the necessary information of a source sentence into a fixed length vector.

Examples:
Aspect Sentiment Analysis:
Words such as "*good*" or "*bad*" could be more relevant to some aspects under consideration or less relevant to other aspects.

Machine Translation:
Some words in the source text could be irrelevant in the translation of the next word.

Visual Question Answering Task:
Background pixels could be irrelevant in answering a question regarding an object in the foreground but relevant to questions regarding the scenery.

Effective solutions of such …

//TODO

Transformer Architecture

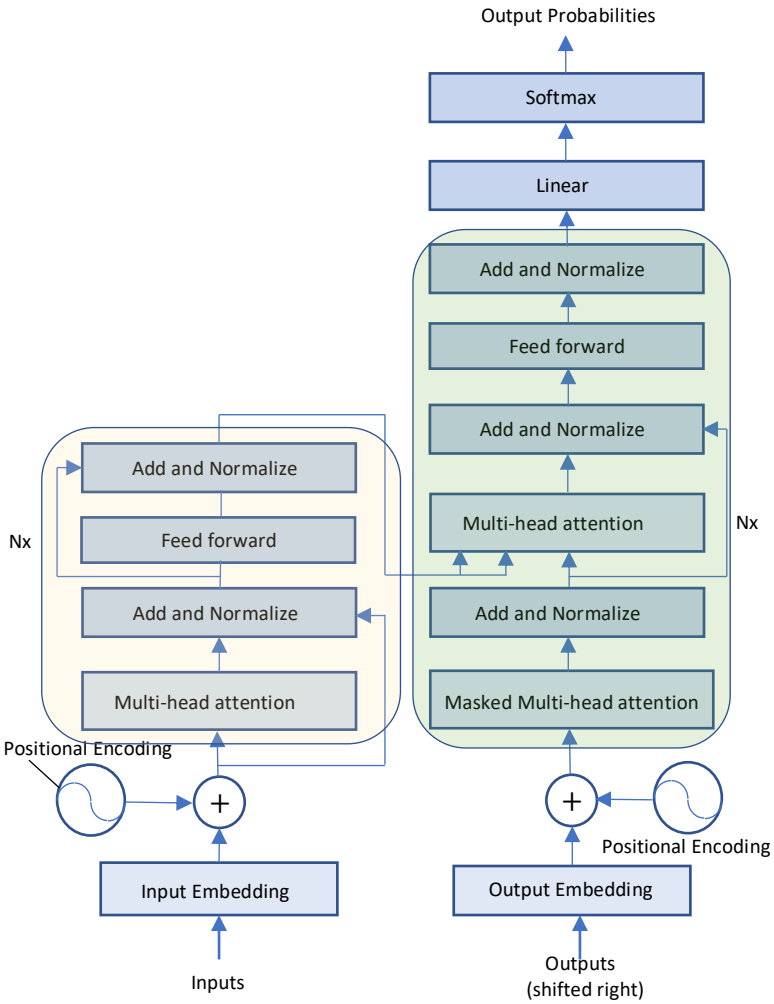


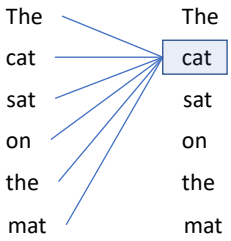
Figure: Transformer Architecture (from the original *"Attention is All you need"*)

The original Transformer model is a stack of 6 layers. The output of layer l is the input of layer $l + 1$ until the final prediction is reached. There is a 6 layer encoder stack on the left and 6 layer encoder stack on the right. On the left the inputs enter the encoder side through an Attention sub-layer and Feed-forward network sub-layer. On the right the target outputs go into the Decoder side through two Attention sub-layers and a Feed-forward network sub-layer. Notice that there is no vestige of Recurrent networks in this architecture – Recurrence have been abandoned. Attention replaces Recurrence which requires an increasing number of operations as the distance between two words increases. The Attention mechanism is a "word-to-word" operation. The Attention mechanism will find how each word is related to all other words in a sequence, including the word being analyzed itself.

Let us examine one popular example which is the following sentence:

The cat sat on the mat.

Attention will run dot products between word vectors and determine the strongest relationships of a word among all other words, including itself ("cat" and "cat").



The attention mechanism will provide a deeper relationship between words and produce better results. For each Attention sub-layer, the original Transformer model runs not one but eight Attention mechanisms in parallel to speed up the calculations. This process is named “Multi-head Attention” and it provides:

- Analysis of sequences
- Elimination of recurrence
- Parallel execution
- Each attention head learns a different aspect of the same input sequence

The Encoder Stack

The layers of the encoder and decoder of the original Transformer model are stacks of layers. Each layer of the Encoder stack has the following structure

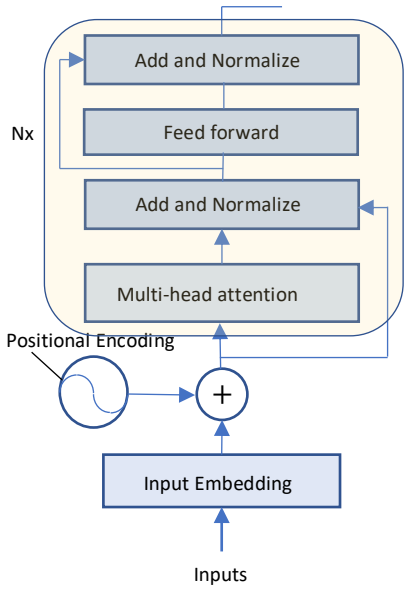


Figure: Encoder layer

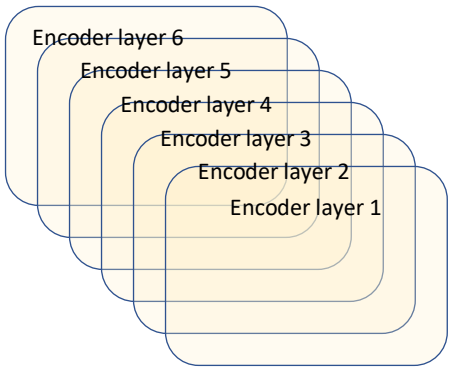


Figure: Stack of 6 Encoder layers (instances)

The original Encoder architecture remains the same for all of the 6 layers of the Transformer model. Each Encoder layer contains two main components (aka sub-layers) – Multi-headed Attention mechanism and a fully connected position-wise Feed-forward network. Notice that a residual connection surrounds each main sub-layer $Sublayer(x)$. These connections transport the unprocessed input x of a sub-layer (Attention or Feed-forward) to a normalization function layer. By doing such

bypass we make sure that key information such as positional encoding is not lost along the way. The normalized output of each layer is thus:

$$LayerNormalization\big(x + SubLayer(x)\big)$$

Though the structure of each of the six layers (instances) of the Encoder is identical, the content of each Layer is not strictly identical to the previous Layer. For example, the embedding sublayer is only present at the bottom level of the stack. The five layers (instances of the Encoder) do not contain an embedding layer and this guarantees that the encoded input is stable through all the layers.

Also, the multi-head attention sublayers perform the same functions from layer 1 to 6. However, they do not perform the same tasks. Each layer learns from the previous layer
//TODO: continue from

Embedding and Positional Encoding

In any NLP model including the Transformer there are two important pieces of information which needs to be constructed for each word in the input sequence: the meaning of each word and its position of the input sequence. The Embedding layer encodes the meaning of the word and the Position encoding layer represents the position of the word. The Transformer combines these two encodings by combining them.

Transformers, unlike RNN, are inherently parallel - that is all words in a sequence are input in parallel. This means that the position information is lost and has to be added back separately.
Let us consider the following example input sequence:

```
The black cat sat on the couch and the brown dog slept on the rug.
```

Let us denote with d_{model} the dimension of the word embedding vector. Usually, we are talking about large number of dimensions such as $256, 512, 1024$. So both the word **black** and **brown** are represented by a word embedding vector with size d_{model}. Vaswani et al are using unit sphere projections to represent positional encoding that will remain limited in magnitude but nevertheless useful in accounting for the position in each word into the corresponding word embedding vector.
For each position i in the dimensions of the word embedding vector we generate a pair of values which represent the projection on the unit sphere for the frequency corresponding to the position i.

$$pe_{(pos\ 2i)}[i] = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$

$$pe_{(pos\ 2i+1)}[i] = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$

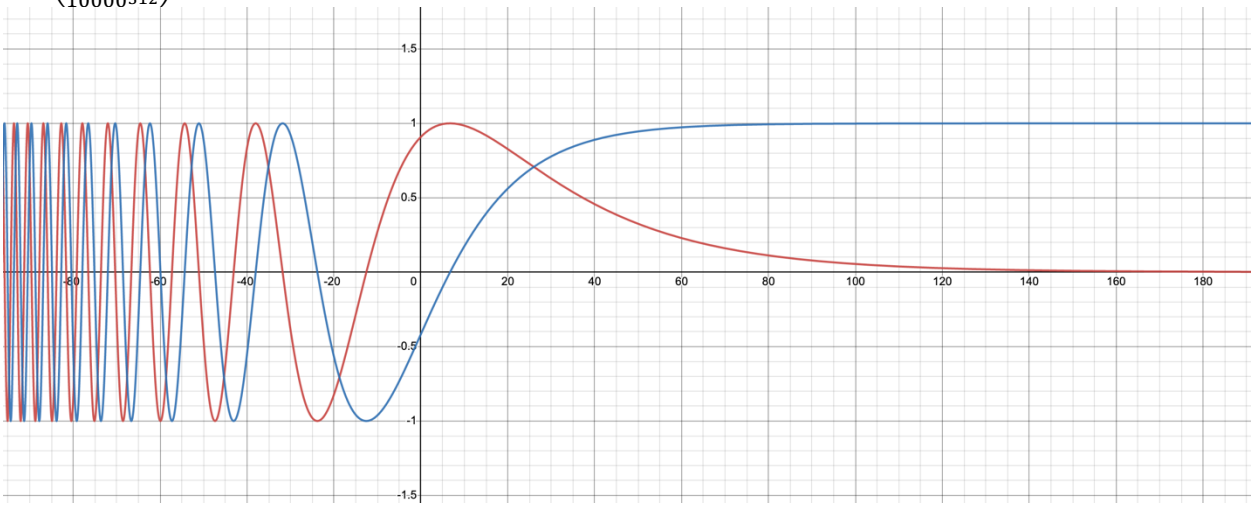
Here the sine projection is applied only to the even indices while the cosine projection is applied only to the odd indices in the word embedding vector. Thus, the positional encoding vector $\overrightarrow{pe}$ is obtained by interleaving $\overrightarrow{pe}_{(pos\ 2i)}$ with $\overrightarrow{pe}_{(pos\ 2i+1)}$. So for $\overrightarrow{pe}$ we can write:

$\overrightarrow{pe} = \left[\overrightarrow{pe}_{(pos\ 2i)} \,|||_i\, \overrightarrow{pe}_{(pos\ 2i+1)}\right]$ where the binary operator $|||_i$ denotes sequence interleaving along its index i.

For $d_{model} = 512$ and the word **black** in the example above we get:

$\sin\left(\frac{2}{10000^{\frac{2i}{512}}}\right)$ for $i = 0,2,..,510$ (——)

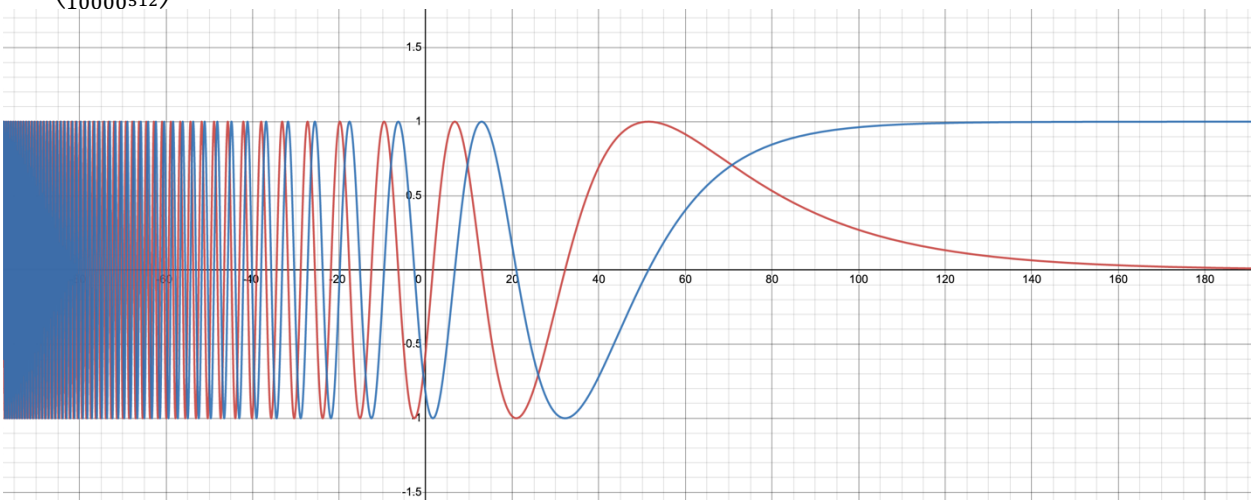
$\cos\left(\frac{2}{10000^{\frac{2i}{512}}}\right)$ for $i = 1,3,..,511$ (——)



With $d_{model} = 512$ and the word **brown** in the example above we get:

$\sin\left(\frac{2}{10000^{\frac{2i}{512}}}\right)$ for $i = 0,2,..,510$ (——)

$\cos\left(\frac{2}{10000^{\frac{2i}{512}}}\right)$ for $i = 1,3,..,511$ (——)



Definiton: *Cosine similarity*

$\vec{a} \cdot \vec{b} = \|\vec{a}\|\|\vec{b}\| \cos\theta \therefore \cos\theta = \frac{\vec{a} \cdot \vec{b}}{\|\vec{a}\|\|\vec{b}\|}$

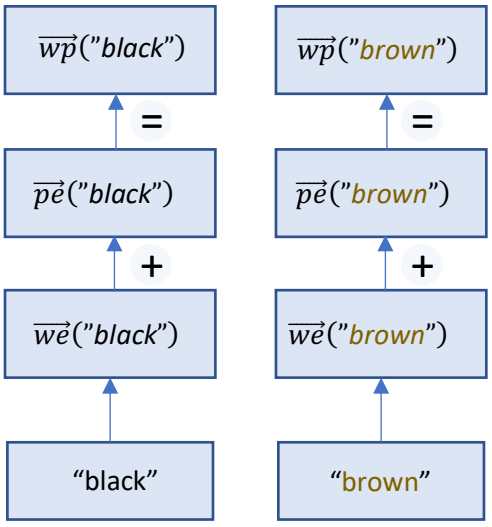
If $\cos\theta = -1$ it means the vectors are opposite in terms of semantic value, $\cos\theta = 1$ indicates they are semantically the same and $\cos\theta = 0$ indicates the vectors are uncorrelated in terms of semantic value. If $\vec{a} = \overrightarrow{pe}(\text{"black"})$ and $\vec{b} = \overrightarrow{pe}(\text{"brown"})$ then

$$\cos\left(\overrightarrow{pe}(\text{"black"}), \overrightarrow{pe}(\text{"brown"})\right) = 0.86000$$

Note that if we denote with $\overrightarrow{we}(\text{"black"})$ the word embedding vector of the word `black` and with $\overrightarrow{we}(\text{"brown"})$ the word embedding vector of the word `brown` where both are computed with *word2vec* then

$$\cos\left(\overrightarrow{we}(\text{"black"}), \overrightarrow{we}(\text{"brown"})\right) = 0.99989$$

The encoding of the position has lower similarity value than the similarity value of the word embedding for the same words. The problem is how to add the positional encoding to the word embedding vectors. The authors of the Transformer chose to implement positional encoding into the words by simply adding the word vectors $\overrightarrow{we}$ with the positional encoding vectors $\overrightarrow{pe}$. The new vector will be denoted with $\overrightarrow{wp}$ (which stands for Word Embedding with Positional Encoding) as shown on the Figure below.



If we apply the cosine similarity function to the word vectors with positional encoding $\overrightarrow{wp}$ for `black` and `brown` we will get :

$$\cos\left(\overrightarrow{wp}(\text{"black"}), \overrightarrow{wp}(\text{"brown"})\right) = 0.96270$$

Thus, we have

$$\cos\left(\overrightarrow{we}(\text{"black"}), \overrightarrow{we}(\text{"brown"})\right) > \cos\left(\overrightarrow{wp}(\text{"black"}), \overrightarrow{wp}(\text{"brown"})\right) > \cos\left(\overrightarrow{pe}(\text{"black"}), \overrightarrow{pe}(\text{"brown"})\right)$$

Initially the word embedding vectors for `black` and `brown` indicate that the two words are very closely related semantically. The positional encoding of those two words showed that those are related semantically but had larger semantic distance compared to the word embeddings and finally the semantic distances between the sum of the $\overrightarrow{we}$ and $\overrightarrow{pe}$ was found to be between the other two. This

result can be interpreted as an assertion that the word vectors with positional encoding retain the information from the word embedding and positional encoding in them.

Encoder Self-Attention

Self-attention refers to the input sequence which pays attention to *itself*.
Attention input parameters are *Query*, *Key* and *Value*. All three parameters are similar structurally with each word in a sequence represented by a vector.
The input sequence is fed into the Input Embedding and Position Encoding, which produce encoded representation of each word in the input sequence that captures the position and meaning of each word. This is fed to all three parameters, Query, Key and Value in the Self-Attention in the first Encoder which then also produces an encoded representation of each word in the input sequence that now incorporates the attention scores for each word as well. As this passes through all Encoders in the stack, each Self-Attention module also adds its own attention scores into each word's representation.

Multi-Head Attention

We divide the dimension of the word vectors d_{model} into h heads. We denote the new vector size for each of the heads with $d_h = d_{model}/h$. Let us denote with d_x the number of words in the input x. With $x_i, i = 1..d$ we will denote each individual word in x. With X we denote the matrix of word vectors with positional encoding added. Each row in that matrix represents word vector with positional encoding corresponding to an input word. Obviously X is of size $d_x \times d_{model}$.
Attention for each word $x_i, i = 1..d$ can be characterized with 3 parameters – a query vector $\vec{q}(x_i)$, key vector $\vec{k}(x_i)$, and a value vector $\vec{v}(x_i)$. With multi-head attention the query, key and value vectors have size d_k. For all words in the input we compose 3 matrices – the query matrix Q, the key matrix K and the value matrix V where each row is a corresponding vector for each word in the input:

$$Q = \begin{bmatrix} \vec{q}(x_1) \\ \vec{q}(x_2) \\ \vdots \\ \vec{q}(x_d) \end{bmatrix}; \quad K = \begin{bmatrix} \vec{k}(x_1) \\ \vec{k}(x_2) \\ \vdots \\ \vec{k}(x_d) \end{bmatrix}; \quad V = \begin{bmatrix} \vec{v}(x_1) \\ \vec{v}(x_2) \\ \vdots \\ \vec{v}(x_d) \end{bmatrix}$$

Attention is computed as:

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

Here $A(Q, K, V)$, obviously, is a square $d \times d$ matrix which is symmetric. Each element $a_{i,j}$ of the Attention matrix represents the attention between the words x_i and x_j.

To obtain those 3 attention parameters we need to train their corresponding weight matrices Q_w, K_w and V_w.

Bibliography

Arbel, N. (2018, December 21). *How LSTM networks solve the problem of vanishing gradients*. Retrieved from https://medium.datadriveninvestor.com/: https://medium.datadriveninvestor.com/how-do-lstm-networks-solve-the-problem-of-vanishing-gradients-a6784971a577

Bag, S. (2022, January 31). *The Complete LSTM Tutorial With Implementation*. Retrieved from https://www.analyticsvidhya.com/: https://www.analyticsvidhya.com/blog/2022/01/the-complete-lstm-tutorial-with-implementation/

Galassi, A., Lipp, M., & Torroni, P. (2020). Attention in Natural Language Processing. *IEEE Transactions on Neural Networks and Learning Systems*.

Hochreiter, S., & Schmidhuber, J. (1997). Long Short Term Memory. *Neural Computation 9(8)*, 1735-1780.

Rojas, R. (1996). *Neural Networks Systematic Introduction.* Berlin: Springer Verlag.

Staudemeyer, R. C., & Morris, E. R. (2019, September 12). Understanding LSTM - a tutorial into Long Short Term Memory Recurrent Neural Networks. 1-42. Schmalkalden University of Applied Sciences, Germany.

Sutskever, I., Vinyals, O., & Le, Q. V. (2014). Sequence to Sequence Learning with Neural Networks. *arXiv:1409.3215 [cs.CL]*, 9.

Vaswani, A., Shazeer, N., Paramr, N., Uszkoreit, J., Jones, L., Gomez, A., . . . Polosukhin, I. (2017). Attention Is All You Need. *31st Conference on Neural Information Processing Systems.* Long Beach: NIPS 2017.