Attention In Natural Language Processing

BILICI, M. Şafak
safakk.bilici.2112@gmail.com

Contents

1 RNN Recap

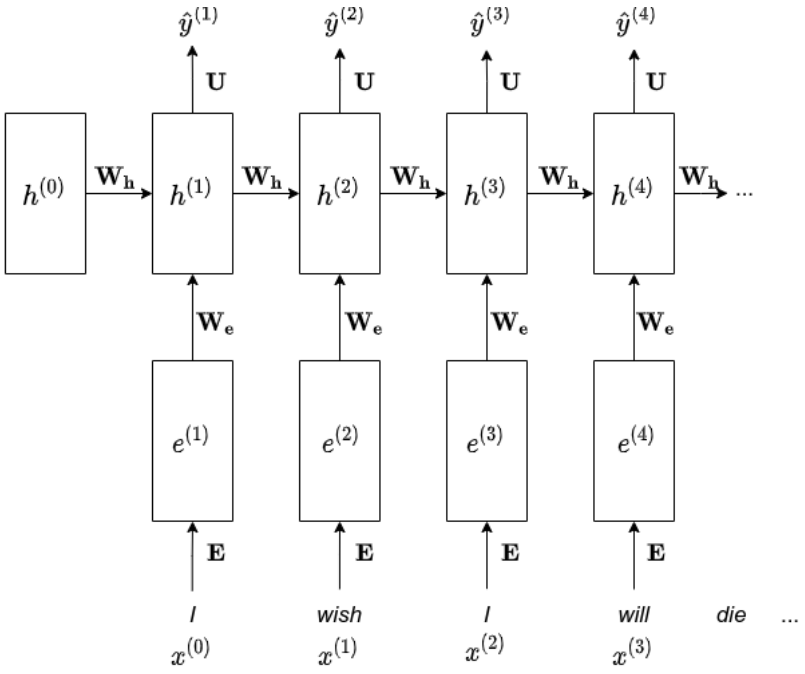


Figure 1: Simple RNN

RNNs have the forward propagation of,

$$x^{(t)} \in \mathbb{R}^{|V|}$$

$$e^{(t)} = E \cdot x^{(t)}$$

$$h^{(0)} \rightarrow \text{initial state (can be learned)}$$
$$h^{(t)} = \sigma(W_h \cdot h^{(t-1)} + W_e \cdot e^{(t)} + b_1)$$
$$\hat{y}^{(t)} = softmax(U \cdot h^{(t)} + b_2) \in \mathbb{R}^{|V|}$$

and the optimal parameters can be learned with backpropagation through time (BPTT). For example, for $\hat{y}^{(1)}$, the loss $J^{(1)}(\theta)$ is negative log-probability of word "wish". So the overall loss is the average of negative log-likelihoods

$$J(\theta) = \frac{1}{T}\sum_{t=1}^{T} J^{(t)}(\theta) = \frac{1}{T}\sum_{t=1}^{T} -\log \hat{y}^{(t)}_{x_{t+1}}$$

For example derivative of loss respect to W_h is

$$\frac{\partial J^{(t)}}{\partial W_h} = \sum_{(i=1)}^{t} \left.\frac{\partial J^{(t)}}{\partial W_h}\right|_{(i)}$$

2 Neural Machine Translation (NMT)

2.1 Training

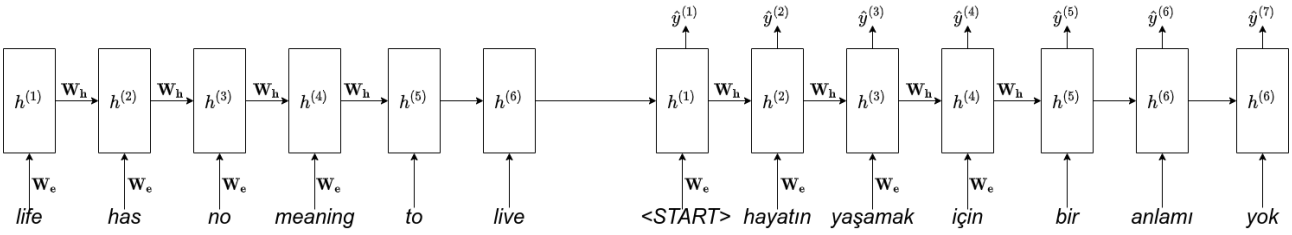


Figure 2: NMT

At training, each loss is computed as in vanilla RNN. For example $J^{(1)}(\theta)$ is negative log-probability of "hayatın" (end-to-end training). Or, $J^{(1)}(\theta)$ is negative log-probability of special token "<END>". NMT directly calculates $p(y \mid x)$:

$$p(y \mid x) = p(y^{(1)} \mid x) \cdot p(y^{(2)} \mid y^{(1)}, x) \cdot p(y^{(3)} \mid y^{(2)}, y^{(1)}, x) \cdot ... \cdot p(y^{(T)} \mid y^{(T-1)}, y^{(T-2)}, ..., y^{(1)}, x)$$

Each component is probability of next target word given target words so far and source sentence x. The objective is, as can be seen,

$$\arg\max_{y} p(y \mid x)$$

2.2 Testing

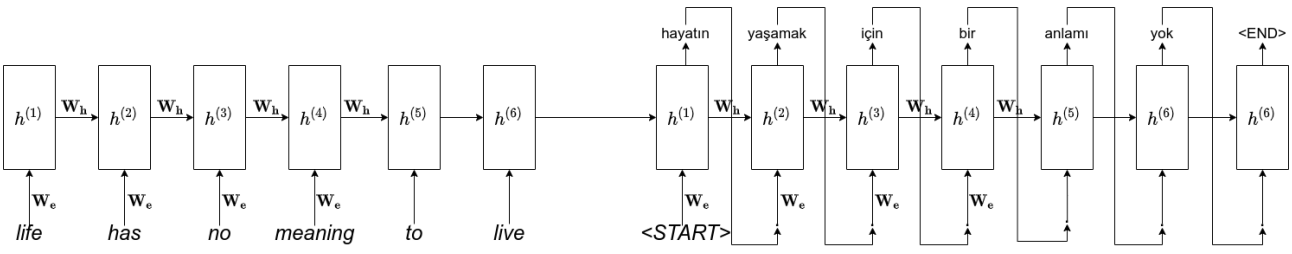


Figure 3: NMT

We call it, this is **greedy decoding**. Taking the most probable word on each step. There are several decoding strategies in natural language processing. So, what is wrong with greedy decoding?

3 Greedy Decoding

Greedy decoding has no way to undo decisions. Also we are not guaranteed that, there will be <END> token at decoding state. On the other hand, if we get a <END> token, there is still a possibility that it may not be the most optimal solution.

input: life has no meaning to live.
→ hayatın .....
→ hayatın yaşamak .....
→ hayatın yaşamak için .....
→ hayatın yaşamak için turtası ..... (???? no going back now)

What we are doing in greedy decoding actually is taking arg max of each output, so we have one output with highest probability, but this does not mean that it will converge to highest overall probability.

4 Exhaustive Search

Could we try computing all possible sentences y? Well yes but actually no. This has $O(\mid V \mid^T)$ complexity.

5 Beam Search

On each step of decoder, keep track of the k most probable partial translations (which we call hypotheses).

- k is the beam size (in practice: 5-10).
- A hypothesis $y_1, ..., y_t$ has a score which is log probability.
  - We search for high-scoring hypotheses, tracking top k on each step.
- Beam search is not guaranteed to find optimal solution.

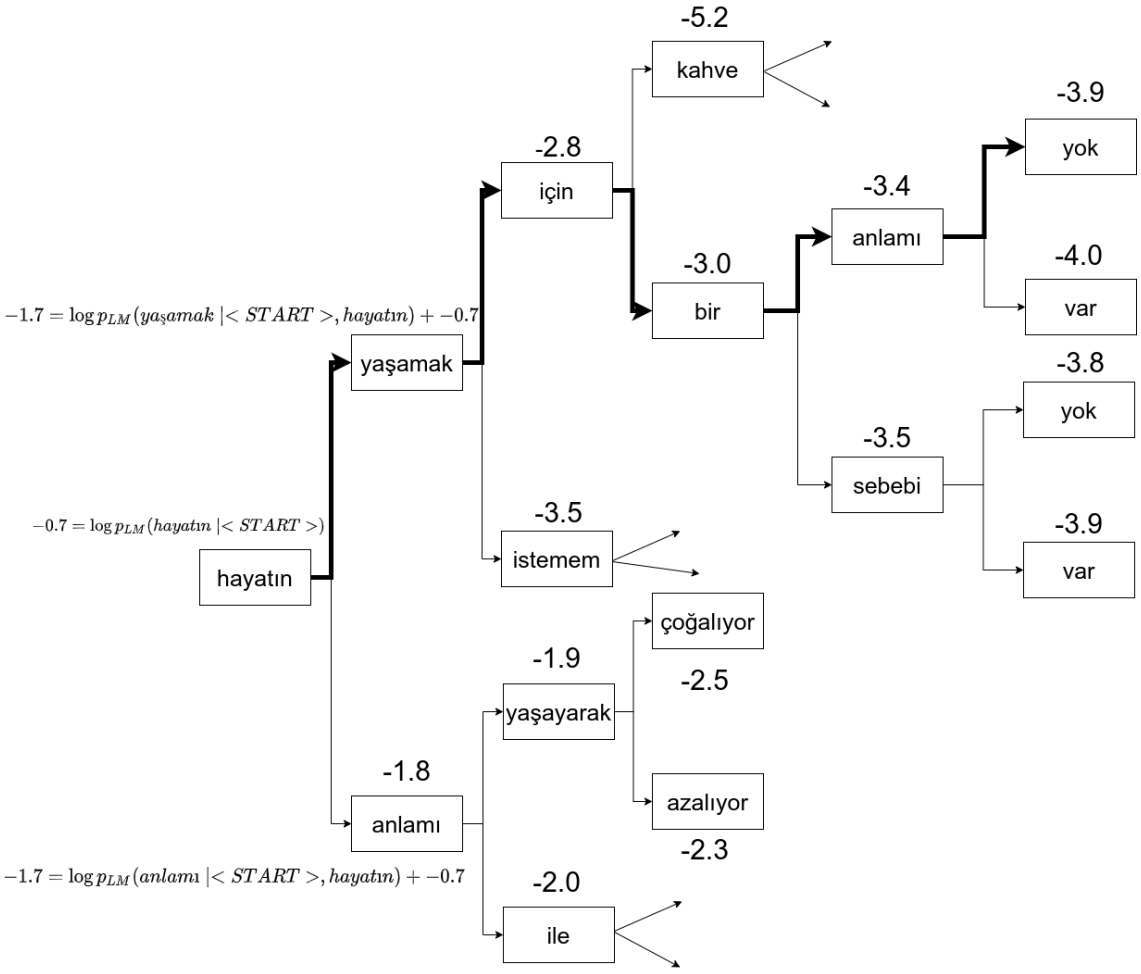


Figure 4: Beam Search

As you can see, Greedy Decoding chooses [”hayatın”, ”yaşamak”, ”için”, ”bir”, ”anlamı”, ”yok”] with score of -3.9. But it is not the most probable sentence. With Beam Search, with beam size = 2, we can select the most probable sentence (in the space of beam size = 2), [”hayatın”, ”yaşamak”, ”için”, ”bir”, ”sebebi”, ”yok”].

5.1 Beam Search: Stopping Criterion

- In greedy decoding, we usually decode until the model produces <END> token.
- In Beam Search, different hypotheses may produce <END> tokens on different timesteps.
  - When a hypothesis produces <END>, that hypothesis is complete.
  - Place it aside and continue exploring other hypotheses via Beam Search.
- Usually we continue Beam Search until:
  - We reach timestep T (pre-defined) or,
  - We reach at least n completed hypotheses (pre-defined).

5.2 Beam Search: Finishing Up

- We have our list of completed hypotheses.

- How to select top one with highest score?
- **Choosing shortest one is a problem**: longer hypotheses have lower scores.
- So, normalize by length!

$$\frac{1}{t}\sum_{i=1}^{t}\log p_{\text{LM}}(y_i \mid y_{i-1}, ..., y_1, x)$$

6 BLEU Score

6.1 Example

Turkish: kedi paspasın üzerinde
Reference 1: the cat is on the mat
Reference 2: there is a cat on the mat
Candidate: the cat the cat on the mat

6.1.1 Bigram Precision

Unigram	Shown In References?
the	1
cat	1
the	1
cat	1
on	1
the	1
mat	1

Then, merge the unigram counts:

Unique Unigram	Count
the	3
cat	2
on	1
mat	1

The total number of counts for the unique unigrams in the candidate sentence is 7, and the total number of unigrams in the candidate sentence is 7. The unigram precision is $7/7 = 1.0$

6.1.2 Bigram Precision

Bigram	Shown In References?
the cat	1
cat the	0
the cat	1
cat on	1
on the	1
the mat	1

Then, merge the bigram counts:

Unique Bigram	Count
the cat	2
cat the	0
cat on	1
on the	1
the mat	1

The total number of counts for the unique bigrams in the candidate sentence is 5, and the total number of bigrams in the candidate sentence is 6. The bigram precision is $5/6 = 0.833$.

6.2 Calculating BLEU

$$\text{BLEU} = \text{BP} \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right)$$

and

$$\text{BP} = \begin{cases} 1 & \text{if } c > r \\ \exp\left(1 - \frac{r}{c}\right) & \text{if } c \leq r \end{cases}$$

Where p_n is the precision of n-gram, with natural logarithm, w_n is the weight between 0 and 1, with constraint of $\sum_{i=1}^{n} w_i = 1$, and BP is the brevity penalty to penalize short machine translations. Where c is the number of unigrams (length) in all the candidate sentences, and r is the best match lengths for each candidate sentence in the corpus. BLEU generally uses $N = 4$ and $w_n = \frac{1}{n}$.

6.3 Proof Of BLEU $\in [0, 1]$

$$\begin{aligned} \exp\left(\sum_{n=1}^{N} w_n \log p_n\right) &= \prod_{n=1}^{N} \exp\left(w_n \log p_n\right) \\ &= \prod_{n=1}^{N} \left[\exp\left(\log p_n\right)\right]^{w_n} = \prod_{n=1}^{N} {p_n}^{w_n} \\ &\in [0, 1] \end{aligned}$$

7 Attention

7.1 The Problem Of Vanishing Gradients

Recall that from RNNs,

$$h^{(t)} = \sigma'(W_h \cdot h^{(t-1)} + W_x \cdot x^{(t)} + b_1)$$

Therefore,

$$\frac{\partial h^{(t)}}{\partial h^{(t-1)}} = diag(\sigma'(W_h \cdot h^{(t-1)} + W_x \cdot x^{(t)} + b_1))W_h$$

Consider the gradient of loss $J^{(i)}(\theta)$ on step i, with respect to the hidden state $h^{(j)}$ on the same previous step j:

$$\begin{aligned} \frac{\partial J^{(i)}(\theta)}{\partial h^{(j)}} &= \frac{\partial J^{(i)}(\theta)}{\partial h^{(i)}} \cdot \prod_{j<t\leq i} \frac{\partial h^{(t)}}{\partial h^{(t-1)}} \\ &= \frac{\partial J^{(i)}(\theta)}{\partial h^{(i)}} \cdot W_h^{(i-j)} \prod_{j<t\leq i} diag(\sigma'(W_h \cdot h^{(t-1)} + W_x \cdot x^{(t)} + b_1)) \end{aligned}$$

So, if W_h is small, then this term gets vanishingly small as$i$ and j get further apart. Consider L2 matrix norms,

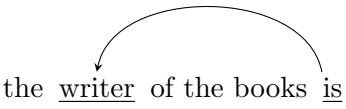
$$\left\|\frac{\partial J^{(i)}(\theta)}{\partial h^{(j)}}\right\| \leq \left\|\frac{\partial J^{(i)}(\theta)}{\partial h^{(i)}}\right\| \cdot \|W_h\|^{(i-j)} \cdot \prod_{j<t\leq i} \left\|diag(\sigma'(W_h \cdot h^{(t-1)} + W_x \cdot x^{(t)} + b_1))\right\|$$

If the $\max_{\lambda_i} \lambda_i \leq 1$ then the gradient will shrink exponentially.

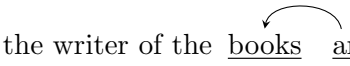
7.2 The Problem Of Recency

As in the Vanishing Gradient, the problem of recency is getting important while sentence length is increasing.

7.2.1 syntactic recency



7.2.2 sequential recency



Recurrent Neural Network Language Model is sequential recency rather than syntactic recency, due to long sequences, information lost and vanishing gradient...

7.3 The Problem Of Alignment

Since Statistical Machine Translation, alignment between source sentence's words and target sentence's words is a main problem in research. There are many alingment types and some of them are complex.

7.3.1 one-to-one Alignment

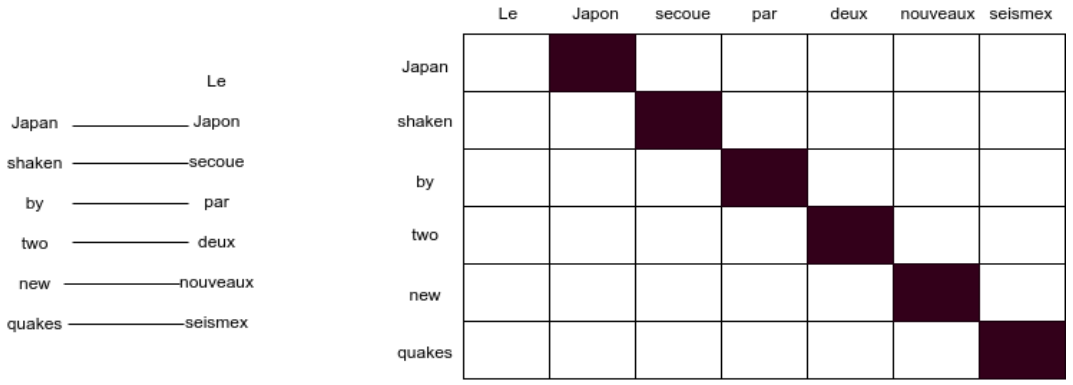


Figure 5: one-to-one

We call "Le" as "spurios" word.

7.3.2 many-to-one Alignment

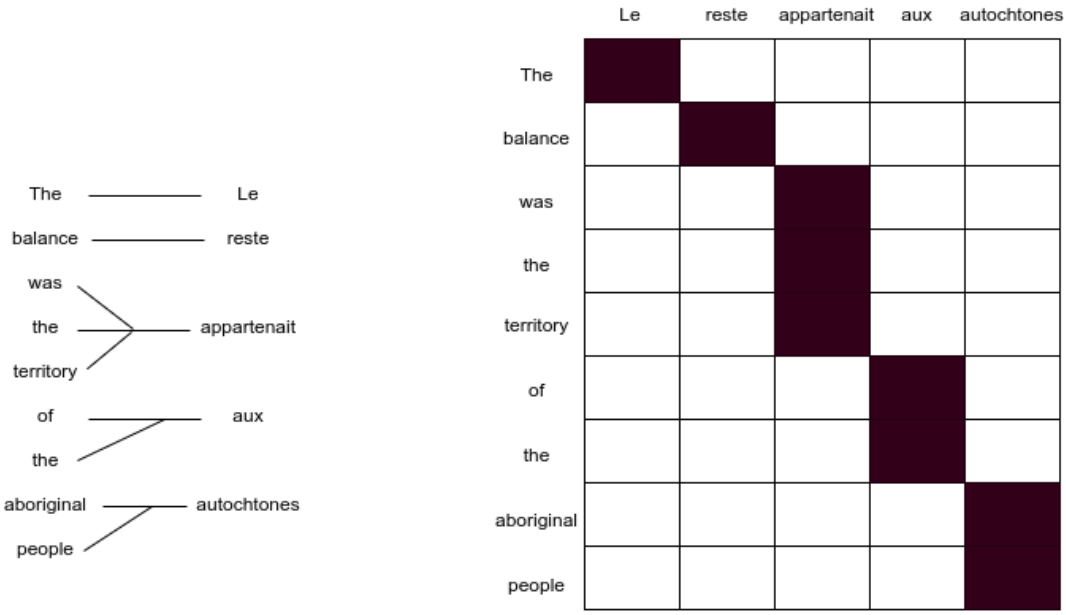


Figure 6: many-to-one

7.3.3 one-to-many Alignment

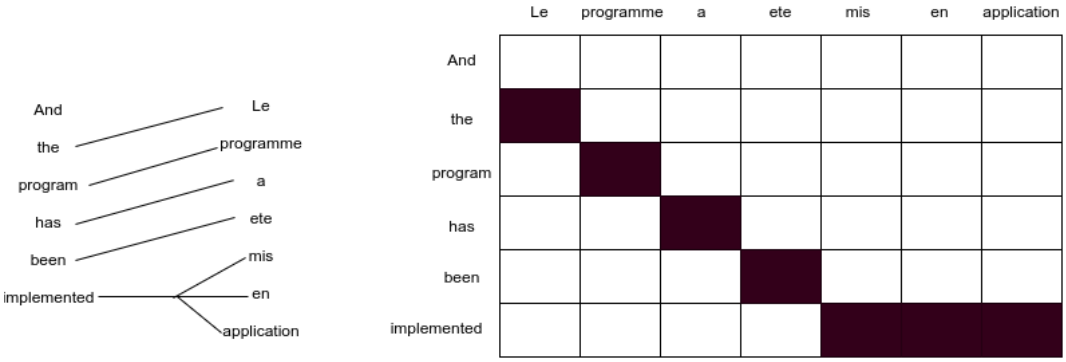


Figure 7: one-to-many

We call "implemented" as "fertile" word.

7.3.4 many-to-many Alignment

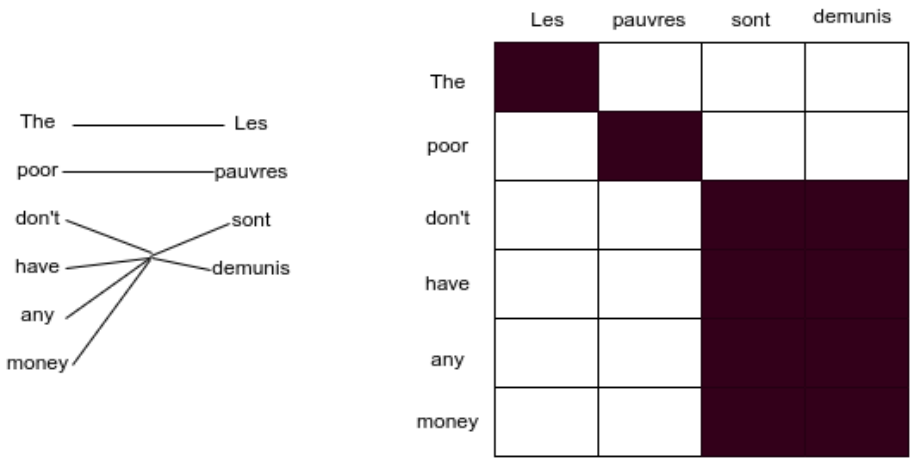


Figure 8: many-to-many

7.4 Fixing Problems With Attention Mechanism

The attention mechanism was born to help memorize long source sentences in neural machine translation (NMT). Rather than building a single context vector out of the encoder's last hidden state, the secret sauce invented by attention is to create shortcuts between the context vector and the entire source input. The weights of these shortcut connections are customizable for each output element. While the context vector has access to the entire input sequence, we don't need to worry about forgetting. The alignment between the source and target is learned and controlled by the context vector. Essentially the context vector consumes three pieces of information; encoder hidden states, decoder hidden states, alignment between source and target.

7.4.1 Neural Machine Translation by Jointly Learning to Align and Translate

Neural Machine Translation with attention mechanism allows us to learn alignments insted of defining phrase based alignments as in Statistical Machine Translation. Attention mechanism is not only for NMT, it can be applied into Language Modeling, Question Answering and other tasks in NLP. Also, since we showed alignments in Statistical Machine Translation, the attentions are binary, not weighted. In linguistics, words are related and dependent, and dependencies may have importance. Learning attention in NMT also allows us to learn those weighted dependencies. In Neural Machine Translation by Jointly Learning to Align and Translate (Bahdanau et al., 2014), the additive attention is proposed. The illustration of this mechanism can be shown,

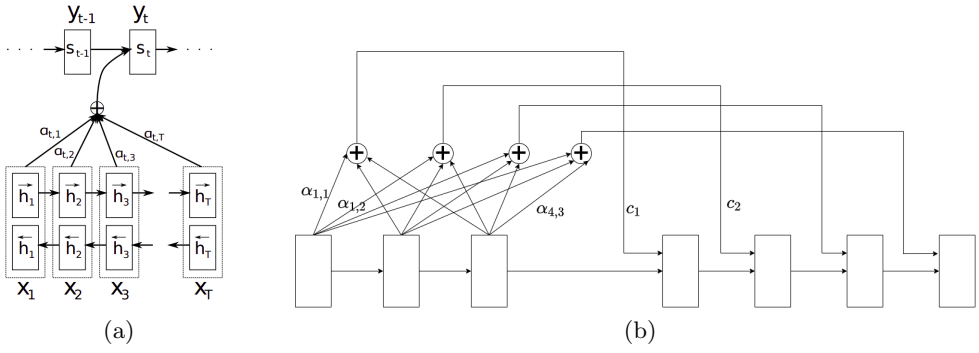


Figure 9: Additive Attention

To make a formal definition, we have input sequence $\mathbf{x}$ with length of n and output sequence $\mathbf{y}$ with length of m,

$$\mathbf{x} = [x_1, x_2, ..., x_n]$$

$$\mathbf{y} = [y_1, y_2, ..., x_m]$$

Bi-directional encoder state is

$$\boldsymbol{h}_i = [\overrightarrow{\boldsymbol{h}}_i^\top; \overleftarrow{\boldsymbol{h}}_i^\top]^\top, i = 1, \ldots, n$$

Let's denote decoder hidden states as s_t which is a function of

$$\boldsymbol{s}_t = f(\boldsymbol{s}_{t-1}, y_{t-1}, \mathbf{c}_t)$$

The context vector c_t is defined as

$$\mathbf{c}_t = \sum_{i=1}^{n} \alpha_{t,i} \mathbf{h_i}$$

The term $\alpha_{t,i}$ represents "How well two words y_t and x_i are aligned", and defined as

$$\alpha_{t,i} = \text{align}(y_t, x_i) = \frac{\exp(\text{score}(\mathbf{s}_{t-1}, \mathbf{h}_i))}{\sum_{j=1}^{n} \exp(\text{score}(\mathbf{s}_{t-1}, \mathbf{h}_j))}$$

The alignment model assigns a score $\alpha_{t,i}$ to the pair of input at position i and output at position t, (y_t, x_i), based on how well they match. The set of $\alpha_{t,i}$ are weights defining how much of each source hidden state should be considered for each output. Since this alignment is built with softmax, it can be seen as a classification between pairs. Score function is a scoring function between (y_t, x_i) pairs.

$$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \mathbf{v}_a^\top \tanh(\mathbf{W}_a[\boldsymbol{s}_t; \boldsymbol{h}_i])$$

where both $\mathbf{v}$ and $\mathbf{W}_a$ are weight matrices to be learned in the alignment model. The matrix of alignment scores is a nice byproduct to explicitly show the correlation between source and target words.

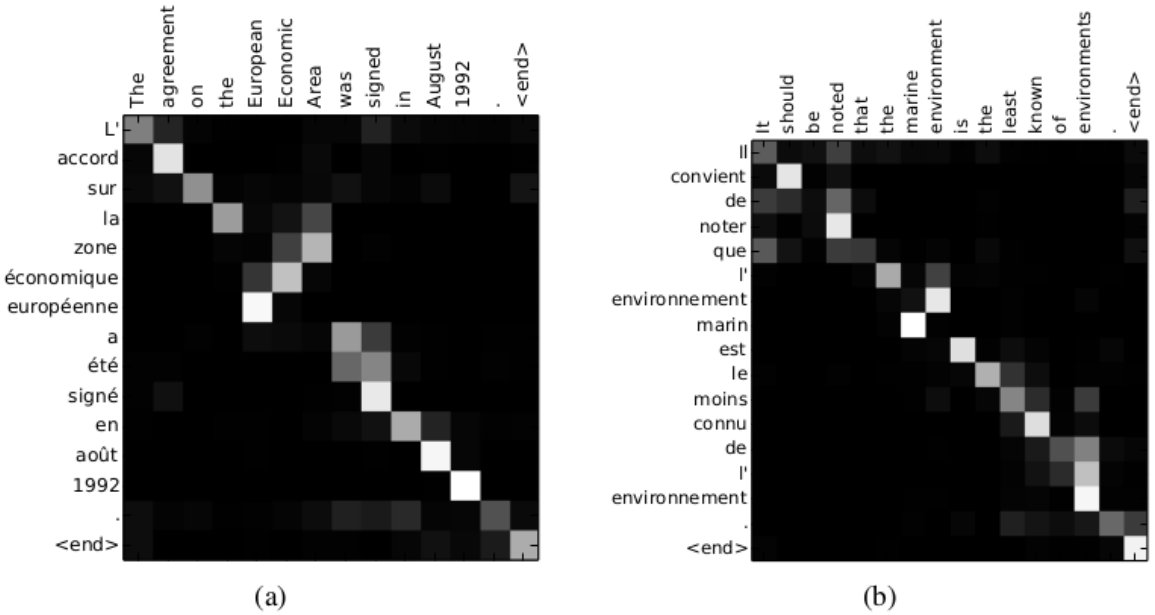
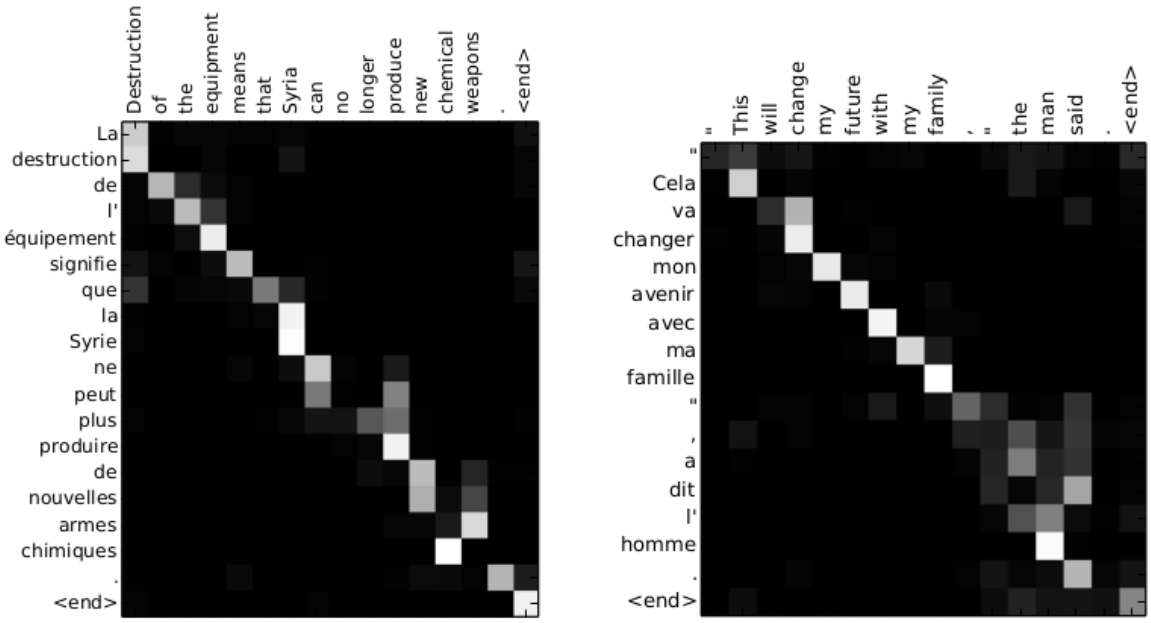


Figure 10: illustrated attention, Neural Machine Translation by Jointly Learning to Align and Translate (Bahdanau et al., 2014)



(c) (d)

Figure 11: illustrated attention, Neural Machine Translation by Jointly Learning to Align and Translate (Bahdanau et al., 2014)

7.4.2 Types Of Attention Mechanism

- **Content-base attention**

$$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \text{cosine}[\boldsymbol{s}_t, \boldsymbol{h}_i]$$

- **Location-Base**:

$$\alpha_{t,i} = \text{softmax}(\mathbf{W}_a \boldsymbol{s}_t)$$

- **General**:

$$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \boldsymbol{s}_t^\top \mathbf{W}_a \boldsymbol{h}_i$$

- **Dot-Product**:

$$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \boldsymbol{s}_t^\top \boldsymbol{h}_i$$

- **Scaled Dot-Product**:

$$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \frac{\boldsymbol{s}_t^\top \boldsymbol{h}_i}{\sqrt{n}}$$

8 Deep Contextualized Word Representations

If we're using this GloVe representation, then the word "stick" would be represented by this vector no-matter what the context was. And yet, "stick" has multiple meanings depending on where it's used. Why not give it an embedding based on the context it's used in – to both capture the word meaning in that context as well as other contextual information?". And so, contextualized word-embeddings were born.

8.1 ELMo (Embeddings from Language Models)

Figure 12: source

ELMo (Deep contextualized word representations (Peters et al., 2018)) was proposed to use contextualized representations of words (in addition, ELMo was not the first one that used contextualized representations, see Semi-supervised sequence tagging with bidirectional language models (Peters et al., 2017) and Learned in Translation: Contextualized Word Vectors (McCann et al., 2017)). ELMo uses vectors derived from a bidirectional LSTM that is trained with a coupled language model (LM) objective on large text corpus.
ELMo word representations are functions of the ENTIRE input sentence. They are computed on top of two-layer biLM with character convolutions.

8.1.1 Bidirectional Language Model

Given a sequence of N tokens $(t_1, t_2, ..., t_N)$, a forward language model computes the probability of the sequence by modeling the probability of token t_k given the history $(t_1, t_2, ...t_{k-1})$:

$$p(t_1, t_2, ..., t_N) = \prod_{k=1}^{N} p(t_k \mid t_1, t_2, ..., t_{k-1})$$

At each position k, each LSTM layer outputs a context-dependent representation $\vec{\mathbf{h}}_{k,j}^{LM}$ where $j = 1, 2, ..., L$ (number of layers). Top layer LSTM output, $\vec{\mathbf{h}}_{k,L}^{LM}$ is used to predict next token t_{k+1} with softmax.

The equations that we formulated above are for forward LM. The backward LM is similar to forward LM.

$$p(t_1, t_2, ..., t_N) = \prod_{k=1}^{N} p(t_k \mid t_{k+1}, t_{k+2}, ..., t_N)$$

with each backward LSTM layer j in a L layer deep model producing representations $\overleftarrow{\mathbf{h}}_{k,j}^{LM}$ of t_k given $(t_{k+1}, t_{k+2}, ..., t_N)$.

And the formulation jointly maximizes the log-likelihood of the forward and backward directions:

$$\sum_{k=1}^{N} \Big(\log p(t_k \mid t_1, t_2, ..., t_{k-1}; \Theta_x, \vec{\Theta}_{LSTM}, \Theta_s) + \log p(t_k \mid t_{k+1}, t_{k+2}, ..., t_N; \Theta_x, \overleftarrow{\Theta}_{LSTM}, \Theta_s) \Big)$$

8.1.2 Using biLM For Representation

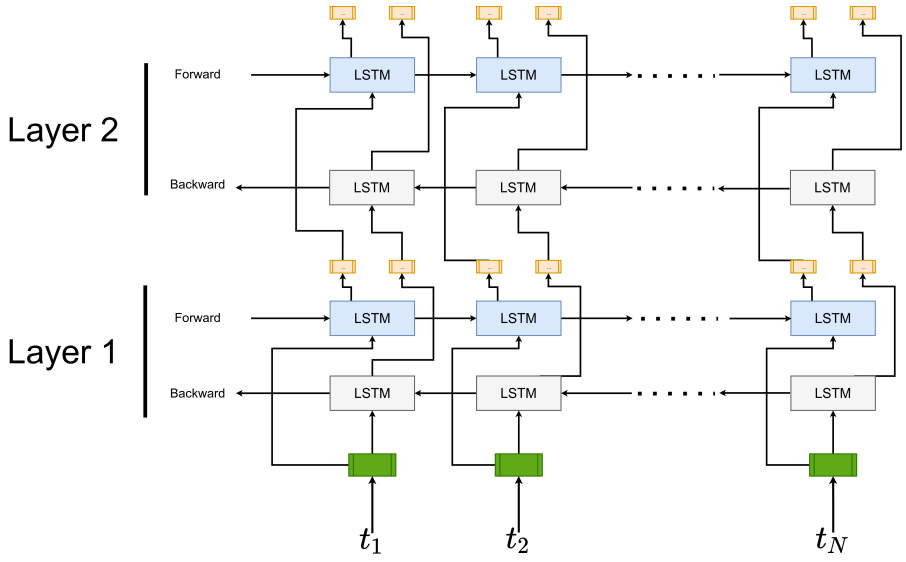


Figure 13: ELMo illustrated

ELMo is a task specific combination of the intermediate layer representations in the biLM. Higher-level LSTM states capture context-dependent aspects of word meaning (word sense disambiguation tasks) while lower level states model aspects of syntax (POS tagging).

For each token t_k, a L-layer biLM computes a set of $2L + 1$ representations (forward + backward + x_k)

$$R_k = \{\mathbf{x}_k^{LM}, \vec{\mathbf{h}}_{k,j}^{LM}, \overleftarrow{\mathbf{h}}_{k,j}^{LM} \mid j = 1, 2, ..., L\}$$

$$= \{\mathbf{h}_{k,j}^{LM} \mid j = 0, 1, ..., L\}$$

where $\mathbf{h}_{k,0}^{LM}$ is the token layer and $\mathbf{h}_{k,j}^{LM} = [\vec{\mathbf{h}}_{k,j}^{LM}; \overleftarrow{\mathbf{h}}_{k,j}^{LM}]$, for each biLSTM layer.
For inclusion in a downstream model, ELMo collapses all layers in R into single vector, $\text{ELMo}_\mathbf{k} = E(R_k; \Theta_e)$. In the simplest case, ELMo just selects the top layer $E(R_k) = \mathbf{h}_{k,l}^{LM}$.
More generally, we compute a task specific weighting of all biLM layers:

$$\text{ELMo}_k^{task} = E(R_k; \Theta^{task}) = \gamma^{task} \sum_{j=0}^{L} s_j^{task} \mathbf{h}_{k,l}^{LM}$$

9 Transformers

The Transformer was proposed in the paper Attention Is All You Need (Vaswani et al., 2017). Transformer uses one of the attention mechanism called self-attention. Transformer still has an

encoder and a decoder like in seq2seq RNNs. But it is purely built on attention mechanism. There are no sequential calculations. All happens at once, hence, it is more parallelizable and requiring significantly less time to train.

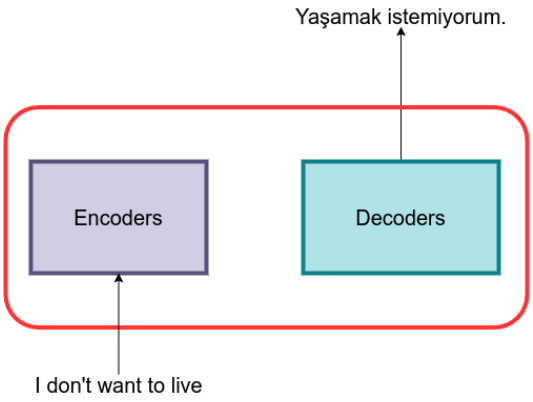


Figure 14: Transformers

The encoding component is a stack of encoders. The decoding component is a stack of decoders of the same number.

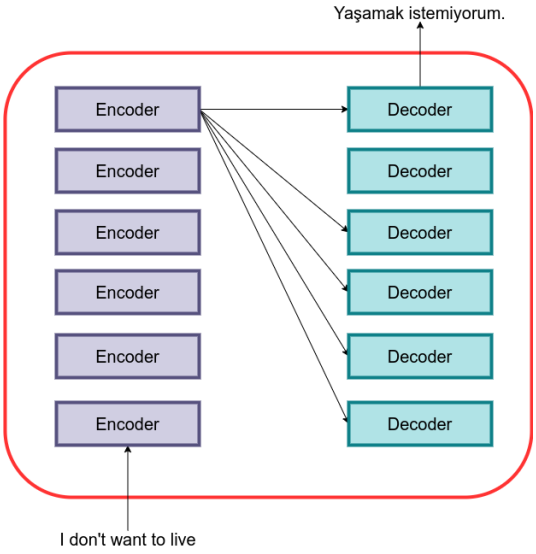


Figure 15: Transformers

But, what are the components of this encoder/decoder stacks?

Encoder discovers interesting things about the source sentence, attends at once pick and choose which word you look at more or less, which we call self-attention.

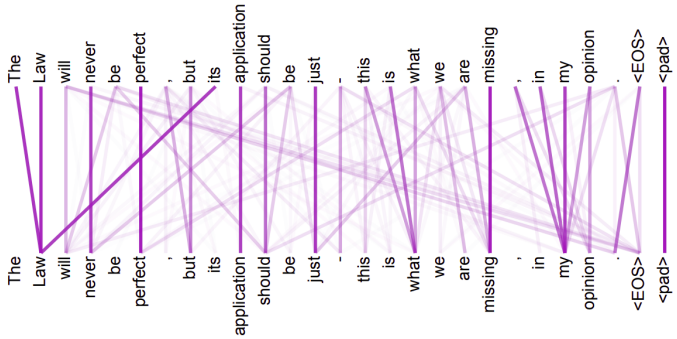


Figure 16: Transformers

9.1 Keys, Queries & Values

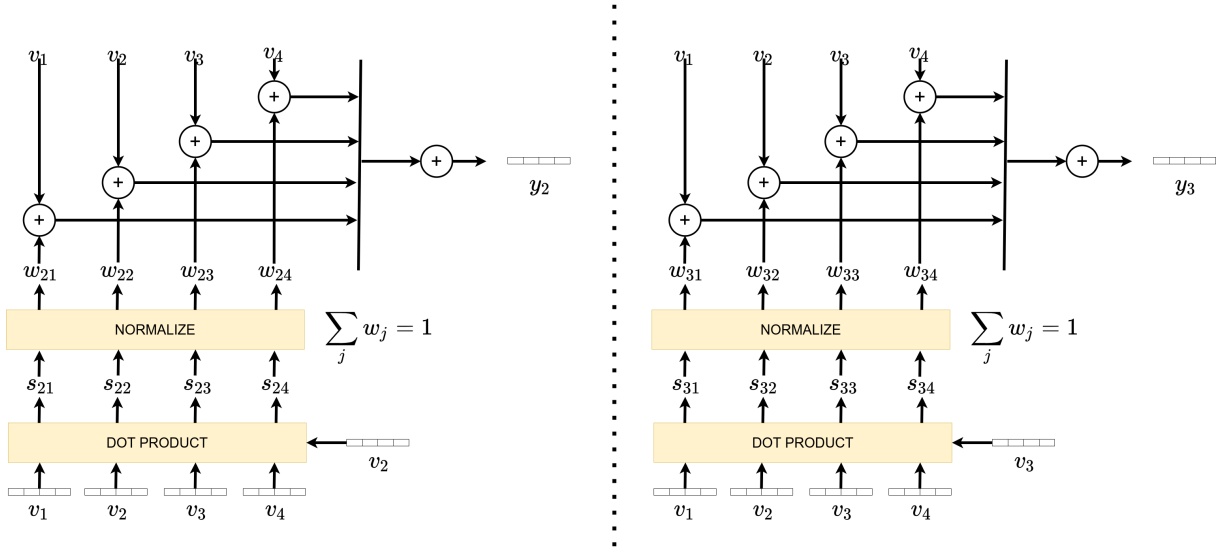


Figure 17: Attention

Self-attention actually is a reweighting all vectors (vectors of tokens in our case), towards current vector. For example for v_2, reweighting scheme is

$$v_2 \cdot v_1 = \tilde{w}_{11} \rightarrow normalize(w_{11})$$

$$v_2 \cdot v_2 = \tilde{w}_{12} \rightarrow normalize(w_{12})$$

$$v_2 \cdot v_3 = \tilde{w}_{13} \rightarrow normalize(w_{13})$$

$$v_2 \cdot v_4 = \tilde{w}_{14} \rightarrow normalize(w_{14})$$

Then

$$y_1 = w_{11} \cdot v_1 + w_{12} \cdot v_2 + w_{13} \cdot v_3 + w_{14} \cdot v_4$$

And this procedure applies to all v_i.

$$y_2 = w_{21} \cdot v_1 + w_{22} \cdot v_2 + w_{23} \cdot v_3 + w_{24} \cdot v_4$$

$$y_3 = w_{31} \cdot v_1 + w_{32} \cdot v_2 + w_{33} \cdot v_3 + w_{34} \cdot v_4$$

$$y_4 = w_{41} \cdot v_1 + w_{42} \cdot v_2 + w_{43} \cdot v_3 + w_{44} \cdot v_4$$

So we have now y_1, y_2, y_3, y_4 with better and more context. We need to learn this reweighting operation with linguistics features.

In Transformers, instead using same v_i vectors, we use "keys, queries and values" to represent embeddings in 3 different ways. It can be seen as key-value pairs in hashing (hash table). We select most related key with queries, then select values with this key. The first step in calculating self-attention is to create three vectors from each of the encoder's input vectors (in this case, the embedding of each word).

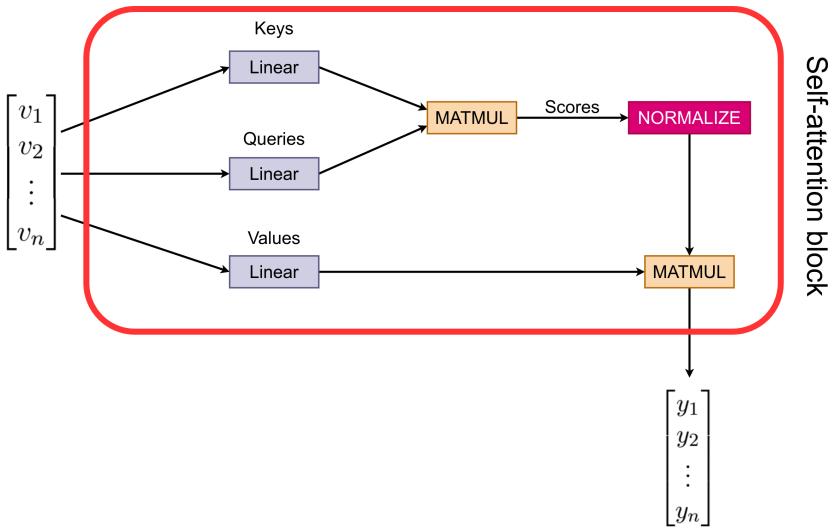


Figure 18: Self-attention block

So for each word, we create a Query vector, a Key vector, and a Value vector. These vectors are created by multiplying the embedding by three matrices that we trained during the training process. Notice that these new vectors are smaller in dimension than the embedding vector. Their dimensionality is 64, while the embedding and encoder input/output vectors have dimensionality of 512. They don't have to be smaller, this is an architecture choice to make the computation of multiheaded attention (mostly) constant.

9.2 Multihead Attention

The paper further refined the self-attention layer by adding a mechanism called "multi-headed" attention. This improves the performance of the attention layer in two ways:

- It expands the model's ability to focus on different positions
- It gives the attention layer multiple "representation subspaces". With multi-headed attention we have not only one, but multiple sets of Query/Key/Value weight matrices (the Transformer uses eight attention heads, so we end up with eight sets for each encoder/decoder). Each of these sets is randomly initialized. Then, after training, each set is used to project the input embeddings (or vectors from lower encoders/decoders) into a different representation subspace.

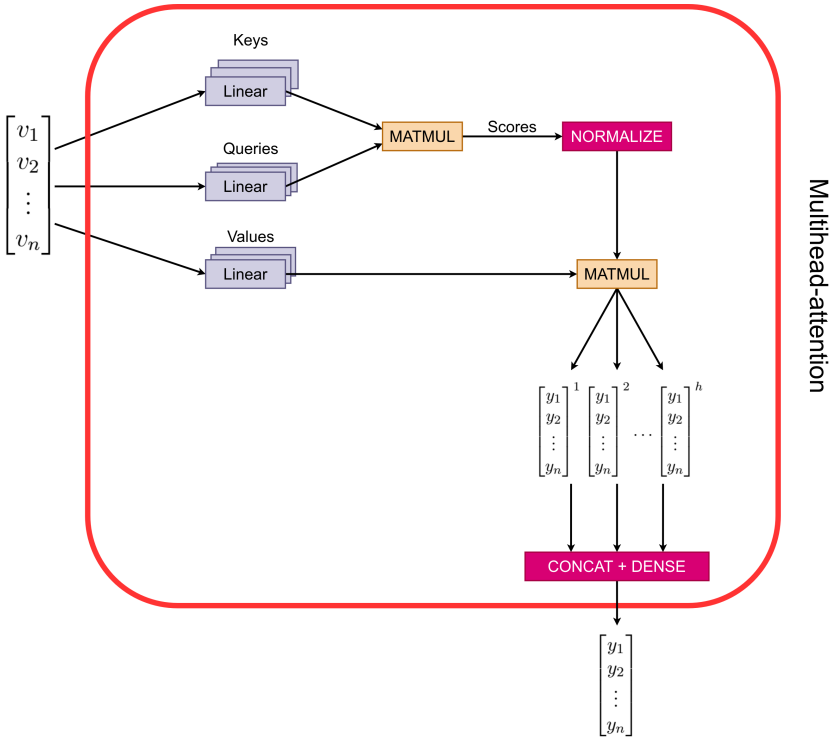
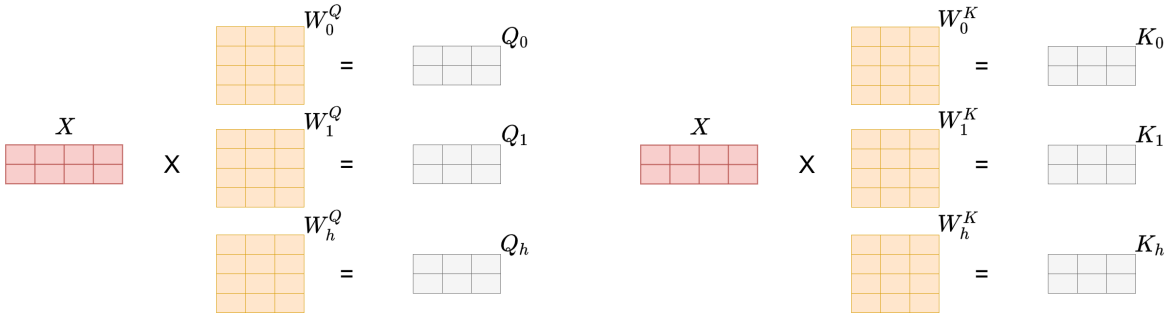


Figure 19: Self-attention block

9.3 Self-Attention Formulation In Nutshell



(a) query

(b) key

(c) value

Figure 20: Linear mapping of Key, Queries & Values

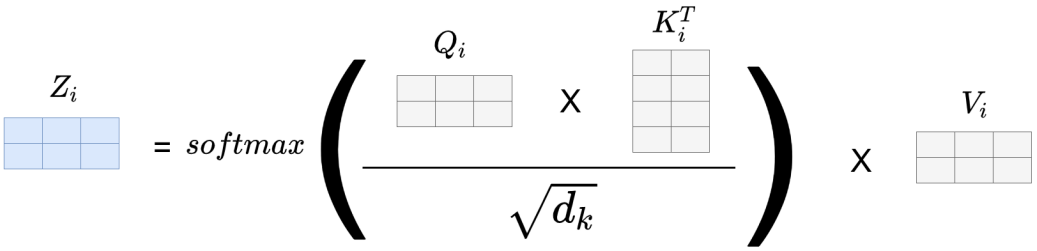


Figure 21: scaled dot product

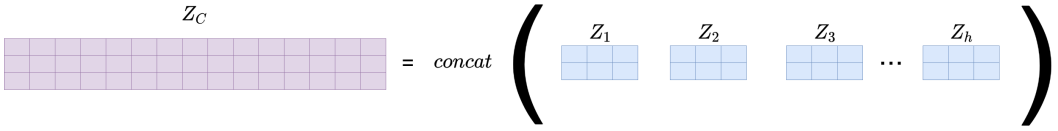


Figure 22: concatenating Z_i

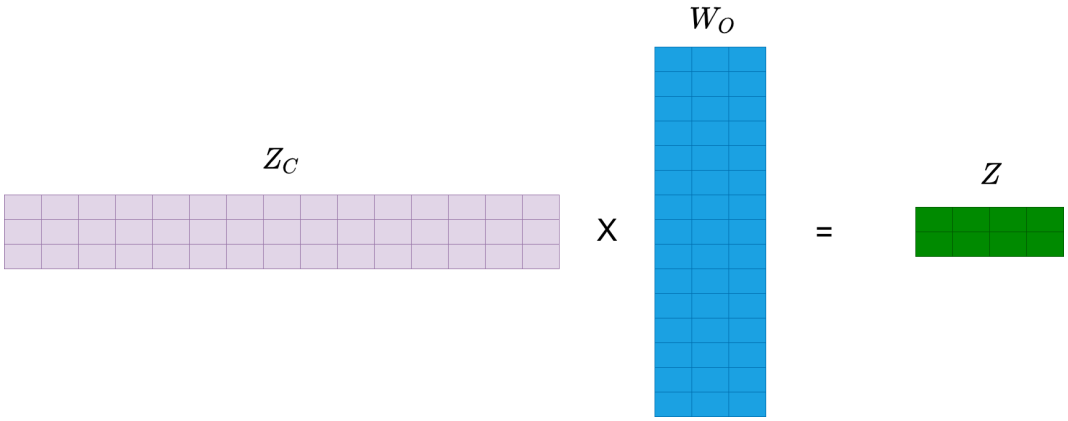


Figure 23: dense

9.4 Self-Attention Dimensions In Nutshell

- $X \in \mathbb{R}^{input\_length \times 512}$
- $W_i^K, W_i^Q, W_i^V \in \mathbf{R}^{512 \times 64}$
- $K_i, Q_i, V_i \in \mathbf{R}^{input\_length \times 64}$
- $Z_i \in \mathbf{R}^{input\_length \times 64}$
- $Z_c \in \mathbf{R}^{input\_length \times (64 \times h)}$
- $W_O \in \mathbf{R}^{(64 \times h) \times 512}$
- $Z \in \mathbf{R}^{input\_length \times 512}$

When the general structure of multihead attention is learned, this structure of Transformers is learned.

9.5 Attention Is All You Need

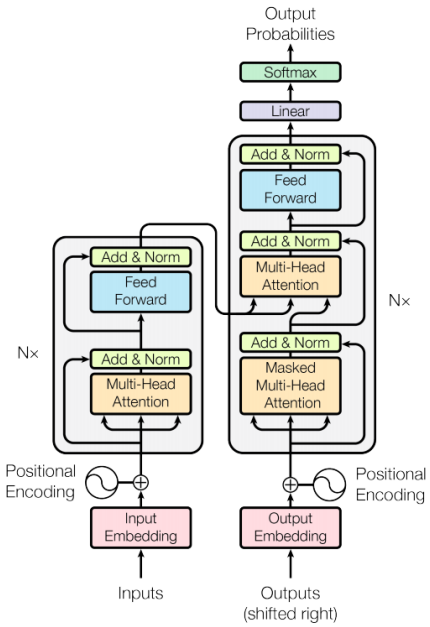


Figure 24: Transformers architecture in Attention Is All You Need (Vaswani et al., 2017)

The transformer-based encoder-decoder model was introduced by Vaswani et al. in the famous Attention is all you need paper and is today the de-facto standard encoder-decoder architecture in natural language processing. Now let's define more formally and detailed the encoder and the decoder of Transformers.

9.5.1 Encoder

The encoder layer has multihead attention + residual connections + feedforward layer. It is nearly described in previous sections. The 3 arrows, inputs of multihead attention, are keys, kueries & values which was explained.

Encoder has an input of word vectors. The output of the encoder is contextualized encoding sequence:

$$f_{\theta_{\text{encoder}}} : (x_1, x_2, ..., x_n) \rightarrow (z_1, z_2, ..., z_n)$$

Each encoder block consists of a bi-directional self-attention layer, followed by two feed-forward layers.

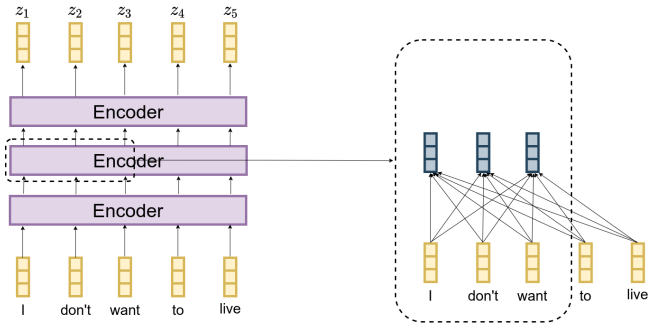


Figure 25: Encoder

As seen in the figure 25, the contextual representation of "don't" is the up-second block at right-most part of figure. The best part of this bidirectionality is now, the word "want" is depends directly on all input word "I", "don't", "want", "to", "live".

9.5.2 Decoder

The decoder structure is nearly same as in the encoder. But the decoder auto-regressively generate the outputs. In the decoder, the self-attention layer is only allowed to attend to earlier positions in the output sequence. This is done by masking future positions (setting them to -inf) before the softmax step in the self-attention calculation.

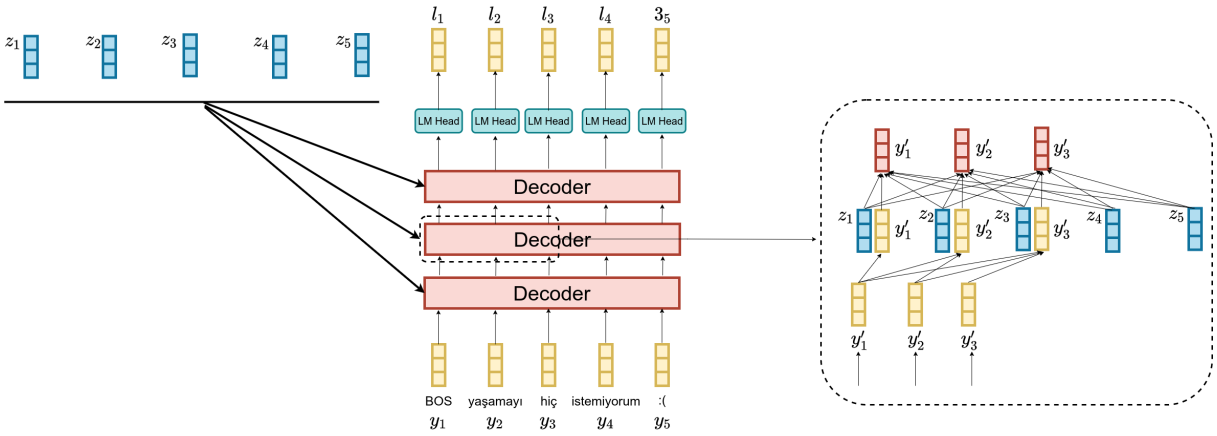


Figure 26: Decoder

The transformer-based decoder hereby maps the sequence of encoded hidden states $z_1, z_2, ..., z_n$ and all previous target vectors $y_1, y_2, ..., y_{i-1}$ to the logit vector l_i. he logit vector l_i is then processed by the softmax operation to define the conditional distribution $p_{\theta_{\text{decoder}}}(y_i \mid y_1, y_2, ..., y_{i-1}, z_1, z_2, ..., z_n)$ just as it is done for RNN-based decoders.
In contrast to transformer-based encoders, in transformer-based decoders, the encoded output vector y_i should be a good representation of the next target vector y_{i+1} and not of the input vector itself. Additionally, the encoded output vector y_i should be conditioned on all contextualized encoding sequence $x_1, x_2, ..., x_n$. To meet these requirements each decoder block consists of a uni-directional self-attention layer, followed by a cross-attention layer and two feed-forward layers.

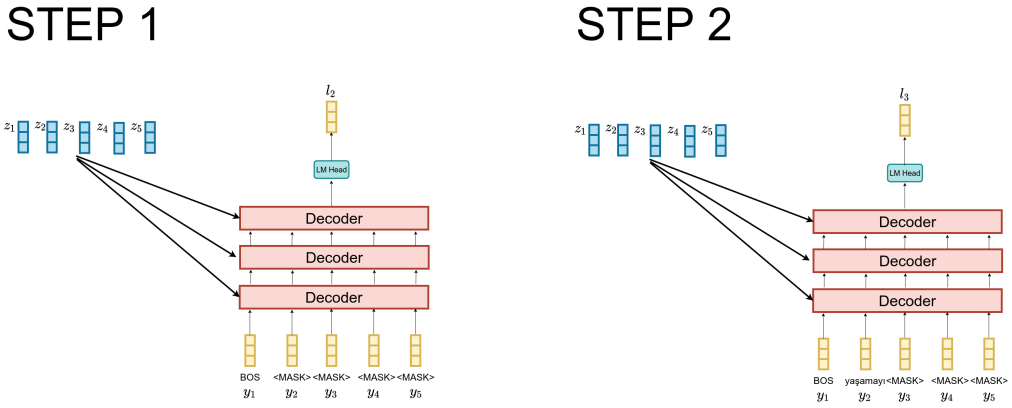


Figure 27: Decoder

So why is it important that we use uni-directional self-attention in the decoder instead of bi-directional self-attention? In our example, the input vector $y_2 =$ yaşamayı is mapped to the logit vector l_3 which is then used to predict the input vector y_3. Thus, if y_2' would have access to the following input vectors y_3', y_4', y_5' the decoder would simply copy the vector representation of "hiç" to be its output y_1''.

10 BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding

BERT is designed to pre-train deep bidirectional representations from unlabeled text by jointly conditioning on both left and right context in all layers. As a result, the pre-trained BERT model can be fine-tuned with just one additional output layer to create state-of-the-art models for a wide range of tasks.

BERT is pre-trained on Masked Language Model (MLM) objective. THe masked language model randomly masks some of the tokens from the input, and the objective is to predict the original vocabulary id of the masked word based on only its context.

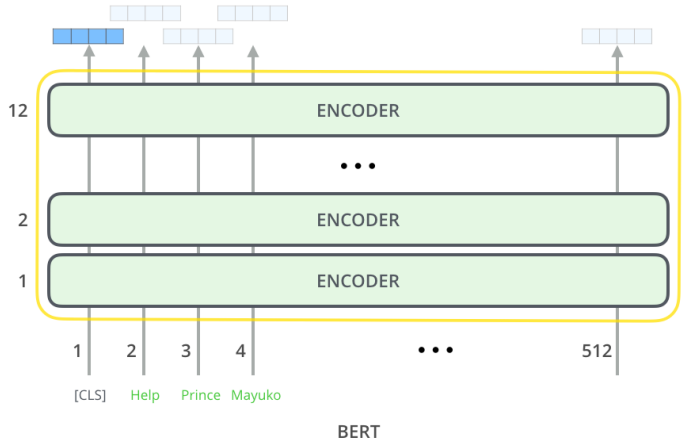


Figure 28: BERT

- $\text{BERT}_{\text{BASE}}$
  - The Number Of Layers $(L) = 12$
  - The Hidden Size $(H) = 768$
  - The Number Of Self-Attention Heads $(A) = 12$
  - Total Parameters = 110M
- $\text{BERT}_{\text{LARGE}}$
  - The Number Of Layers $(L) = 24$
  - The Hidden Size $(H) = 1024$
  - The Number Of Self-Attention Heads $(A) = 16$
  - Total Parameters = 340M

10.1 Pre-Training And Fine-Tuning

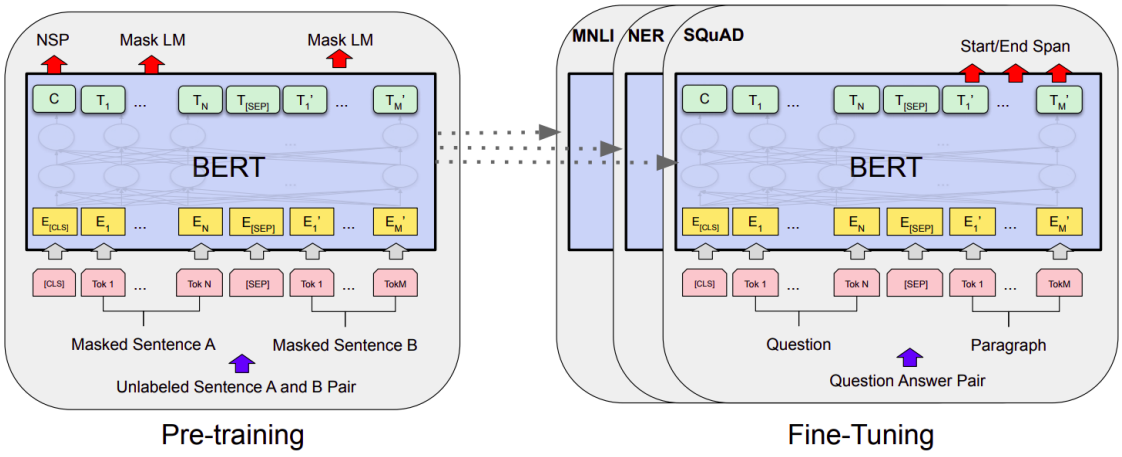
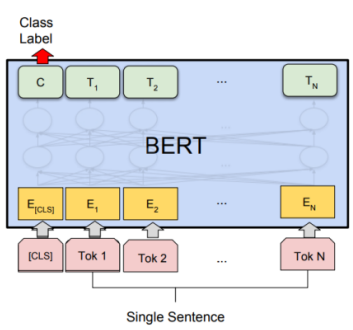
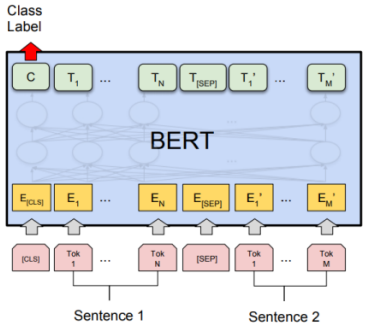


Figure 29: BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding

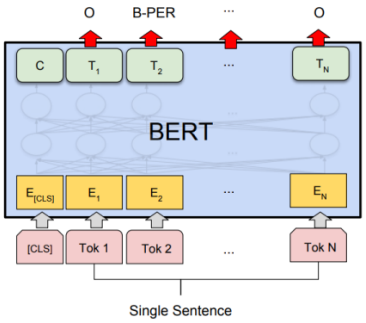
After pre-training (MLM and next sentence prediction), input representations are used to finetune on downstream tasks.
The first token of every sequence is always a special classification tokem (`[CLS]`). The final hidden state corresponding to this token is used as the aggregate sequence representation for classification tasks. Sentence pairs are packed together into a single sequence.



(a) sequence classification



(b) sentence pair classification



(c) single sentence tagging

Figure 30: BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding

11 Resources

- **Books**
  - Speech and Language Processing, Daniel Jurafsky and James H. Martin
  - Turkish Natural Language Processing, Kemal Oflazer and Murat Saraçlar
  - Linguistics: An Introduction to Linguistic Theory, Victoria A. Fromkin.
- **Video Series**
  - Stanford CS224n
  - Dan Jurafsky's Natural Language Processing Series
  - CMU CS 11-747, Spring 2020 Neural Networks for NLP
  - Algorithms for NLP
- **Blogs**
  - Sebastian Ruder's Blog
  - Jay Alammar's Blog
  - Lilian Weng's Blog
  - Chris Olah's Blog
- **Softwares**
  - huggingface/transformers, huggingface/tokenizers, fairseq, NLTK, spaCy, Gensim, Spark NLP.

12 References

[1] The Mathematics of Statistical Machine Translation: Parameter Estimation, Brown et al., 1994.

[2] How to generate text: using different decoding methods for language generation with Transformers, Patrick von Platen, 2020. URL: https://huggingface.co/blog/how-to-generate

[3] Stanford CS224n: Natural Language Processing with Deep Learning (Machine Translation, Attention). Christopher Manning, Richard Socher, Guillaume Genthial, Lucas Liu, Barak Oshri, Kushal Ranjan. URL: http://web.stanford.edu/class/cs224n/readings/cs224n-2019-notes06-NMT_seq2seq_attention.pdf

[4] Bilingual Evaluation Understudy (BLEU), Lei Mao. URL: https://leimao.github.io/blog/BLEU-Score/

[5] Stanford CS224n: Natural Language Processing with Deep Learning (Vanishing Gradients). Christopher Manning, Richard Socher, Guillaume Genthial, Lucas Liu, Barak Oshri, Kushal Ranjan. URL: http://web.stanford.edu/class/cs224n/readings/cs224n-2019-notes05-LM_RNN.pdf

[6] Neural Machine Translation by Jointly Learning to Align and Translate, Bahdanau et al., 2014. arXiv:1409.0473

[7] Deep contextualized word representations., Peters et al., 2018. arXiv:1802.05365

[8] The Illustrated BERT, ELMo, and co. (How NLP Cracked Transfer Learning), Jay Alammar. URL: http://jalammar.github.io/illustrated-bert/

[9] Attention Is All You Need., Vaswani et al., 2017. arXiv:1706.03762

[10] The Illustrated Transformer, Jay Alammar. URL: http://jalammar.github.io/illustrated-transformer/

[11] NLP for Developers and the Algorithm Whiteboard (attention1, attention2, attention3), RASA. URL: https://www.youtube.com/playlist?list=PL75e0qA87dlG-za8eLI6t0_Pbxafk-cxb

[12] BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding, Devlin et al., 2018. arXiv:1810.04805