TTIC 31230, Fundamentals of Deep Learning

David McAllester, Fall 2023

Language Modeling

Machine Translation

Recurrent Neural Networks

The Transformer

Language Modeling

The recent progress on NLP benchmarks is due to pretraining on language modeling.

Language modeling is based on unconditional cross-entropy minimiztion.

$$\Phi^* = \underset{\Phi}{\operatorname{argmin}} \; E_{y \sim \mathrm{Pop}} \; [-\ln P_\Phi(y)]$$

In language modeling y is a sentence (or fixed length block of text).

Language Modeling

Let V be some finite vocabulary of tokens.

Each token is a character sequence that can be used as a part of a rare word such a name in a foreign language. Most English words are a single token.

Tokens typically do not cross word boundaries.

We are interested in probability distributions over V^* (the finite squanteces of tokens).

Language Modeling

Let Pop be a population distribution over sequences of tokens.

We want to train a model $P_\Phi(y)$ for token sequences y

$$\Phi^* = \underset{\Phi}{\operatorname{argmin}} \; E_{y \sim \mathrm{Pop}} \; [-\ln P_\Phi(y)]$$

A structured object, such as a token sequence or an image, has an exponentially small probability.

Autoregressive Models

An autoregressive model uses the chain rule to represent a distribution on sequences in terms of the conditional probability for each token given the earlier tokens tokens.

$$P_\Phi(w_0, w_1, \cdots, w_T) = \prod_{t=0}^{T} P_\Phi(w_t \mid w_0, \ldots, w_{t-1})$$

Modern language models are actually defining probability distributions on long sequences (thousands of tokens).

The end-of-squence token

We want to define a probability distribution over sentence of different length.

For this we require that each sentence is "terminated" with an end of sequence token `<EOS>`.

Training

$$P_\Phi(w_0, w_1, \cdots, w_T) = \prod_{t=0}^{T} P_\Phi(w_t \mid w_0, \ldots, w_{t-1})$$

For training we need to compute the log loss on a training example.

The log loss on a sequence is the sum of the log losses for each token generation.

Generation

$$P_\Phi(w_0, w_1, \cdots, w_T) = \prod_{t=0}^{T} P_\Phi(w_t \mid w_0, \ldots, w_{t-1})$$

We can generate from an autoregressive language model by generating one word at a time.

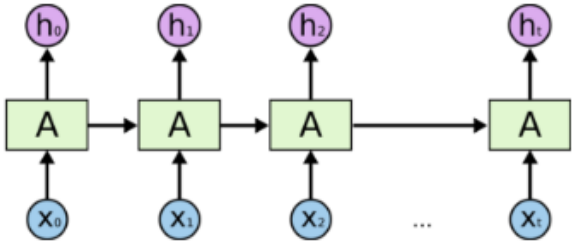
To generate we sample from a probability distribution over the first word. Once this word is generated we compute a probability distribution for the second word and so on.

Word Embeddings

Each word w is associated with a vector $e(w)[I]$ called the embedding of word w.

The matrix e can be viewed as a dictionary assigning each word w the vector $e(w)[I]$.

Recurrent Neural Network (RNN) Language Modeling

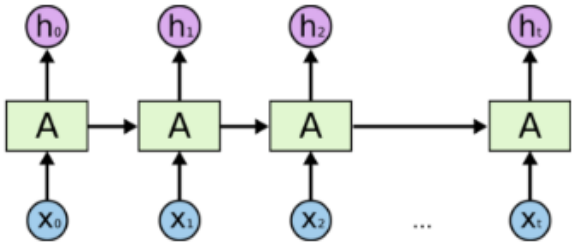


[Christopher Olah]

A typical RNN neural language model has the form

$$P_\Phi(w_t \mid w_0, \cdots, w_{t-1}) = \underset{w_t}{\text{softmax}}\, e(w_t)[I]h[t-1, I]$$

Vanilla RNNs



[Christopher Olah]

A Vanilla RNN uses two-input linear threshold units.

$$h[t,j] = \sigma\left(W^{h,h}[j,I]h[t-1,I] + W^{x,h}[j,K]x[t,K] - B[j]\right)$$

Exploding and Vanishing Gradients

If we avoid saturation of the activation functions then we get exponentially growing or shrinking eigenvectors of the weight matrix.

Note that if the forward values are bounded by sigmoids or tanh then they cannot explode.

However the gradients can still explode.

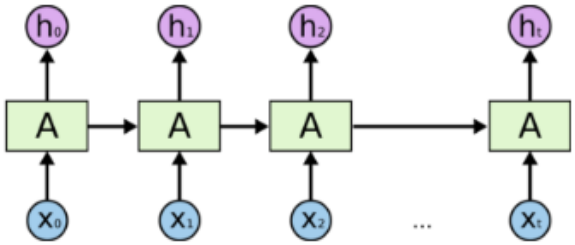
Exploding Gradients: Gradient Clipping

We can dampen the effect of exploding gradients by clipping them before applying SGD.

$$W.\text{grad}' = \begin{cases} W.\text{grad} & \text{if } ||W.\text{grad}|| \leq n_{\max} \\ \\ n_{\max}\, W.\text{grad}/||W.\text{grad}|| & \text{otherwise} \end{cases}$$

See `torch.nn.utils.clip_grad_norm`

Time as Depth



[Christopher Olah]

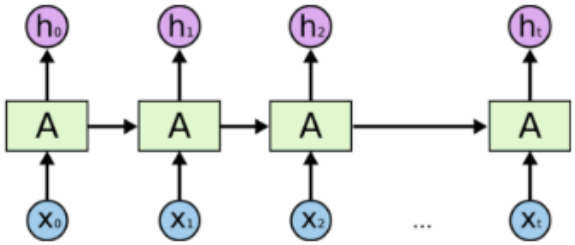
We would like the RNN to remember and use information from much earlier inputs.

All the issues with depth now occur through time.

However, for RNNs at each time step we use the same model parameters.

In CNNs at each layer uses its own model parameters.

"Residual Connections" Through Time

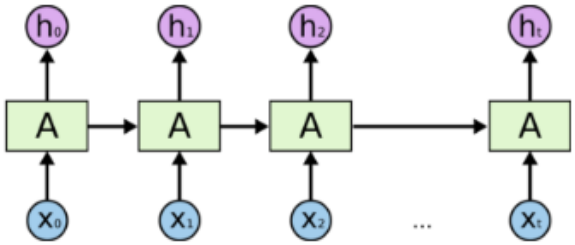


[Christopher Olah]

We would like to have residual connections through time.

However, we have to handle the fact that the same model parameters are used at every time step.

Gated RNNs



[Christopher Olah]

$$h[t,j] = G_t[t,j]h[t-1,j] + (1 - G[t,j])R[t,j]$$

This is analogous to a residual connection.

Rather than add the "next layer" $R[t,j]$ to the input $h[t-1,j]$ as in a residual connection, we take a convex combination determined by a computed "gate" $G[t,j] \in [0,1]$.

Update Gate RNN (UGRNN)

$$
\begin{aligned}
R[t,j] &= \tanh\left(W^{h,R}[j,I]h[t-1,I] + W^{x,R}[j,K]x[t,K] - B^R[j]\right) \\
G[t,j] &= \sigma\left(W^{h,G}[j,I]h[t-1,I] + W^{x,G}[j,K]x[t,K] - B^R[j]\right) \\
h[t,j] &= G[t,j]h[t-1,j] + (1 - G[t,j])R[t,j] \\
\Phi &= (W^{h,R}, W^{x,R}, B^R, W^{h,G}, W^{x,G}, B^G)
\end{aligned}
$$

$$\tanh(x) \in (-1,1) \qquad \sigma(x) \in (0,1)$$

Hadamard product

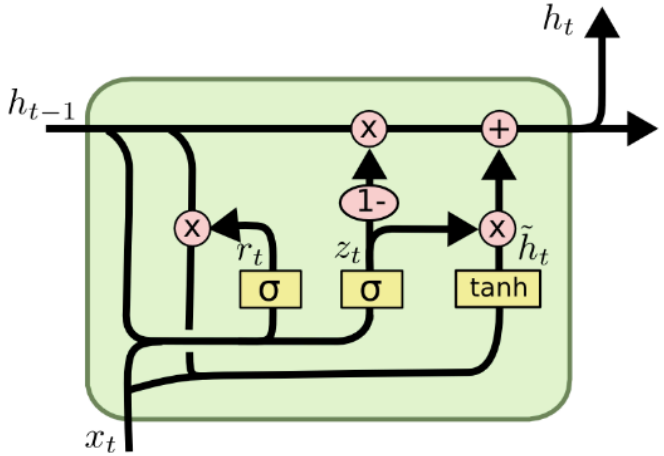
$$h[t, j] = G[t, j]h[t-1, j] + (1 - G[t, j])R[t, j]$$

is sometimes written as

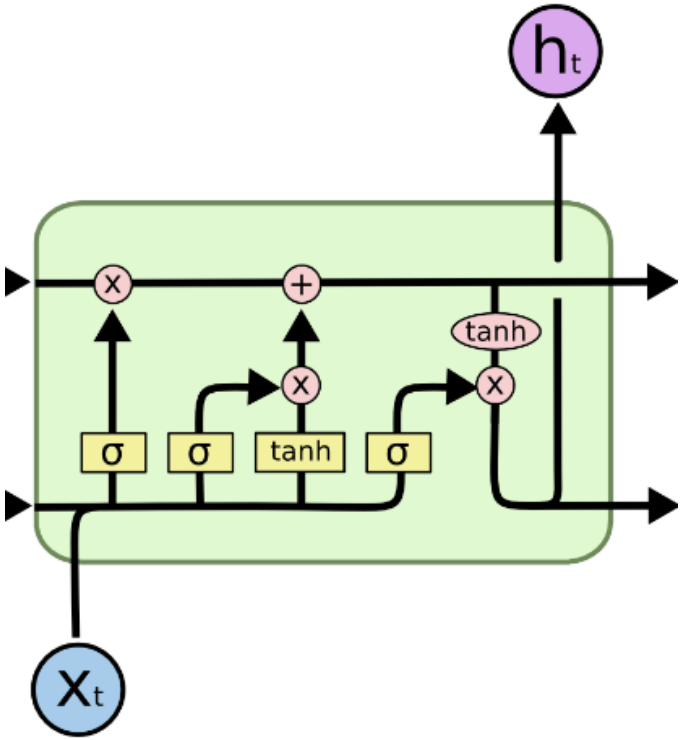
$$h[t, J] = G[t, J] \odot h[t-1, J] + (1 - G[t, J]) \odot R[t, J]$$

$\odot$ is the Hadamard product (componentwise product) on vectors.

Gated Recurrent Unity (GRU) by Cho et al. 2014



Long Short Term Memory (LSTM)

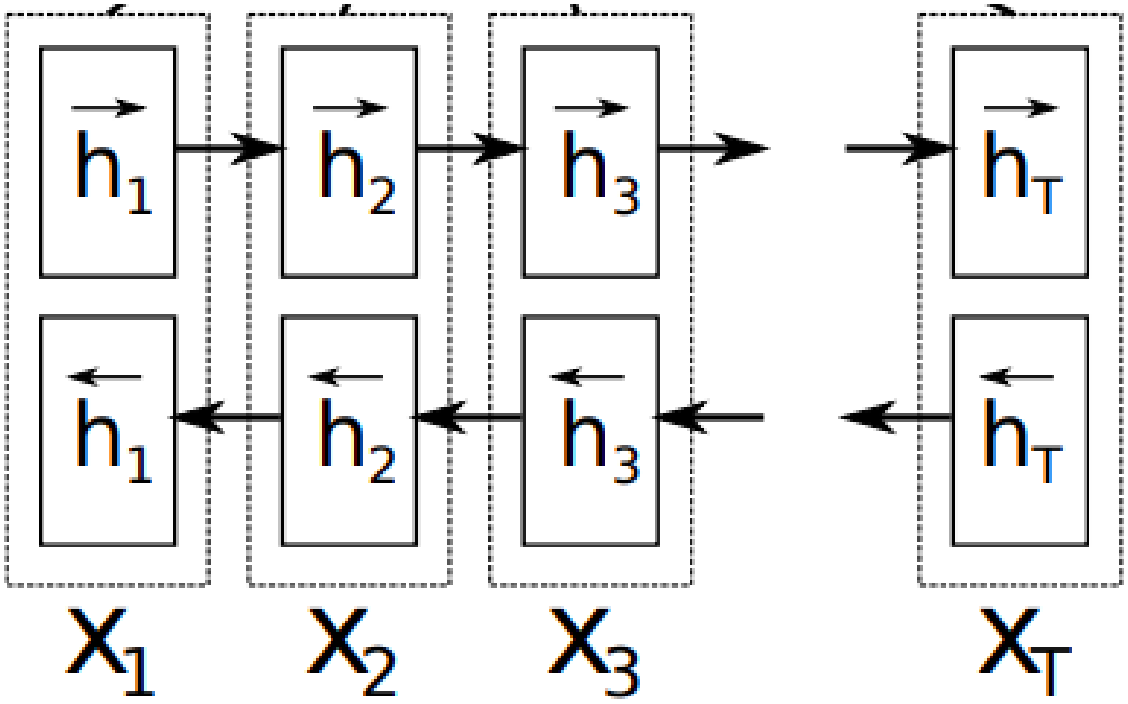


[LSTM: Hochreiter&Shmidhuber, 1997]

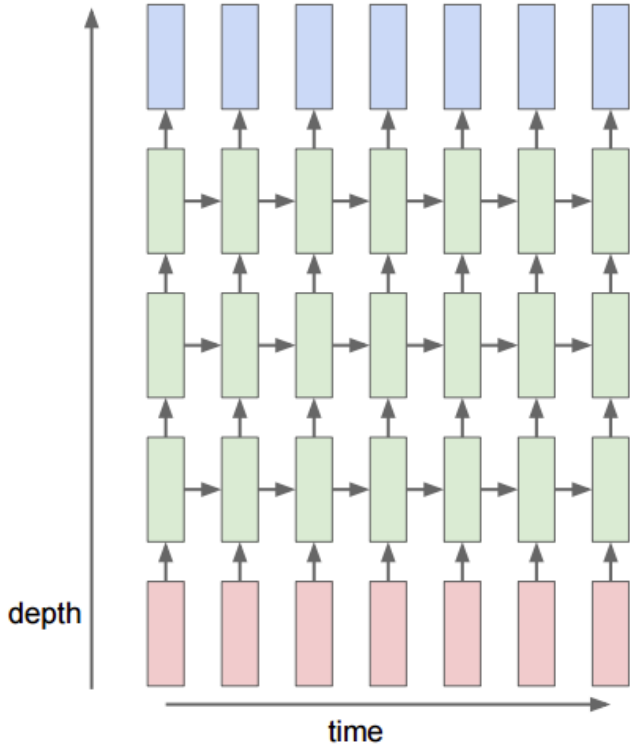
UGRNN vs. GRUs vs. LSTMs

[Collins, Dickstein and Sussulo 2016] found that the number of parameters is more important than the choice of RNN architectures.

bidirectional RNNS



Multi-Layer RNNs



Modern versions would stack layers using residual connections.

Machine Translation

$$w_0, \ldots, w_{T_{\text{in}}} \Rightarrow \tilde{w}_0, \ldots, \tilde{w}_{T_{\text{out}}}$$

Translation is a **sequence to sequence** (seq2seq) task.

Sequence to Sequence Learning with Neural Networks, Sutskever, Vinyals and Le, NeurIPS 2014, arXiv Sept 10, 2014.

Machine Translation

We define a model

$$P_\Phi\left(\tilde{w}_0, \ldots, \tilde{w}_{T_{\text{out}}} \mid w_0, \ldots, w_{T_{\text{in}}}\right)$$

$$\Phi^* = \underset{\Phi}{\text{argmin}}\ E_{\langle x, y\rangle \sim \text{Pop}}\ [-\ln P_\Phi(y|x)]$$

Translation Using Thought Vectors

The final state of a **right-to-left (backward)** RNN is viewed as a **"thought vector"** representation of the input sentence.

We use the thought vector for the niput sentence as the initial hidden state of a **left-to-right (forward)** RNN language model generating the output sentence.

Computing the input thought vector backward provides a good start to the forward generation of the output.

The Introduction of Attention

Neural Machine Translation by Jointly Learning to Align and Translate Dzmitry Bahdanau, Kyunghyun Cho, Yoshua Bengio, ICLR 2015 (arXiv Sept. 1, 2014)

Attention

As we generate each word in the output translation we compute an attention over the input.

Intuitively, we want to define an "alignment" between the words in the output and the words in the input.

In modern terminology this is calles a "cross attention" — one thing (the output) attending to a different thing (the input).

This is different from the "self attention" used in transformers.

Encoder-Decoder Models with Cross-Attention

Let h_{thought} be the thought vector for the input sentence.

Let $h_{\text{in}}(t_{\text{in}})$ be a sequence of vectors generated by the encoder for the sequence of input words.

Let $h_{\text{out}}(t_{\text{out}})$ be a thought vector for the first t_{out} words in the output sentence.

$$\text{Attention} \quad \alpha[t_{\text{out}}, t_{\text{in}}] = \text{softmax}_{t_{\text{in}}} \; e(w_{t_{\text{out}}})[N] \; h_{\text{in}}(t_{\text{in}})[N]$$

$$\text{Weighted Sum :} \quad \tilde{h}_{\text{out}}(t_{\text{out}}) = \alpha[t_{\text{out}}, T_{\text{in}}] \; h_{\text{in}}(T_{\text{in}})$$

$$\text{generate :} \quad w_{\text{t}_{\text{out}}+1} \sim P_{\Phi}(w_{t_{\text{out}}+1} \; | \tilde{h}_{\text{though}}, h_{\text{out}}(t_{\text{out}}), \tilde{h}_{\text{out}}(t_{\text{out}})),$$

Cross Attention in Image Captioning

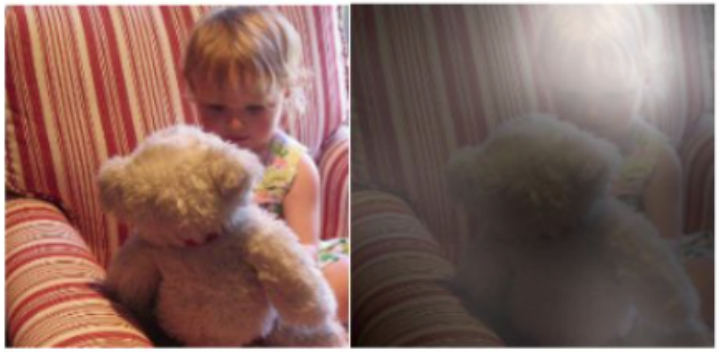
We can treat image captioning as translating an image into a caption.

In translation with attention involves an attention over the input aligning output words with positions in the input.

For each output word we get an attention over the image positions.

Attention in Image Captioning

A woman is throwing a frisbee in a park.



A little girl sitting on a bed with a teddy bear.

Xu et al. ICML 2015

The Transformer: Self Attention

Attention is All You Need, Vaswani et al., June 2017

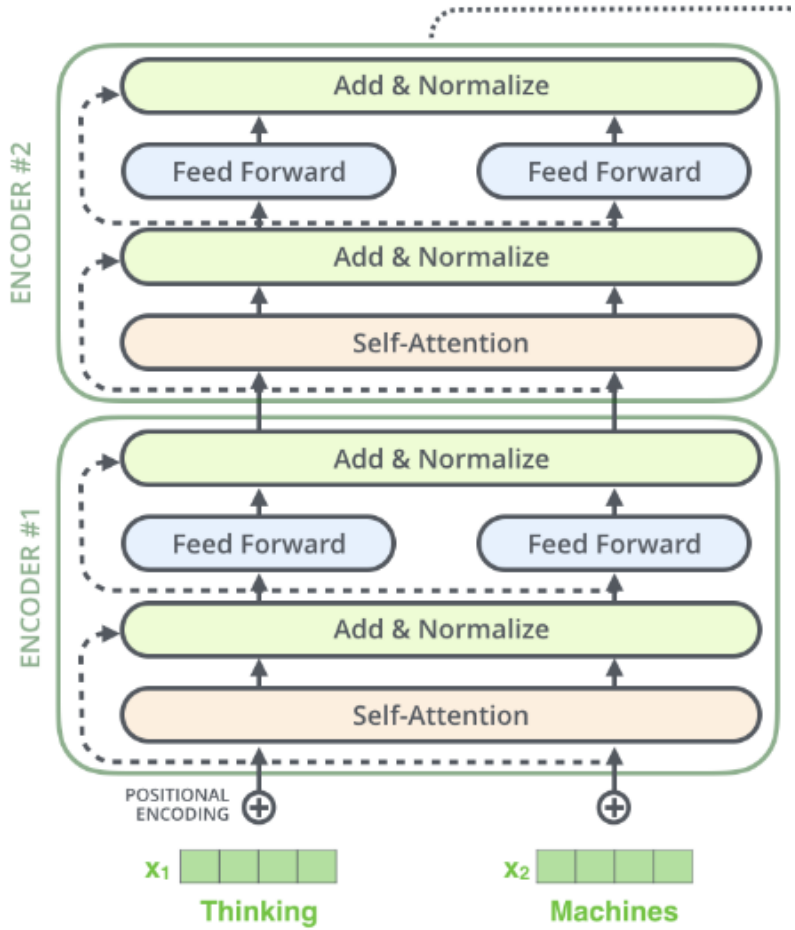
We replace the RNNs with self attention.

For the encoder we will have a "residual stack" $h_{0,\text{in}}(t_{\text{in}}), \ldots, h_{N,\text{in}}(t_{\text{in}})$.

Set $h_{0,\text{in}}(t_{\text{in}}) = e(w_{t_{\text{in}}}); \text{pos}(t_{\text{t}_{\text{in}}})$.

Here semicolon denotes vector concatenation and $\text{pos}(t_{\text{in}})$ is a "position encoding" for the position t.

The Residual stack



Parallel Layer Computation

However, in the transformer we can compute the layer $L_{\ell+1}[T, J]$ from $L_\ell[T, J]$ in parallel.

This is an important difference from RNNs which compute sequentially over time.

In this respect the transformer is more similar to a CNN than to an RNN.

Self-Attention

The fundamental innovation of the transformer is the self-attention layer.

For each position t in the sequence we compute an attention over the other positions in the sequence.

Transformer Heads

There is an intuitive analogy between the Transformer's self attention and a dependency parse tree.

In a dependency parse cibsists if edges between words labeled with grammatical roles such as "subject-of" or "object-of".

The self attention layers of the transformer we have "heads" which can be viewed as labels for dependency edges.

Self attention constructs a tensor $\alpha[k, t_1, t_2]$ — the strength of the attention weight (edge weight) from t_1 to t_2 with head (label) k.

Query-Key Attention

For each head k and position t we compute a key vector and a query vector with dimension I typically smaller than dimension J.

$$\text{Query}_{\ell+1}[k, t, I] = W^Q_{\ell+1}[k, I, J] L_\ell[t, J]$$

$$\text{Key}_{\ell+1}[k, t, I] = W^K_{\ell+1}[k, I, J] L_\ell[t, J]$$

$$\alpha_{\ell+1}[k, t_1, t_2] = \underset{t_2}{\text{softmax}} \frac{1}{\sqrt{I}} \text{Query}_{\ell+1}[k, t_1, I] \text{Key}_{\ell+1}[k, t_2, I]$$

Computing the Output

$$\begin{aligned}
\text{Value}_{\ell+1}[k,t,I] &= W^V_{\ell+1}[k,I,J]L_\ell[t,J] \\
h^1_{\ell+1}[k,t,I] &= \alpha[k,t,T]\text{Value}[k,T,I] \\
h^2_{\ell+1}[t,C] &= h^1_{\ell+1}[0,t,I];\cdots;h^1_{\ell+1}[K-1,t,I] \\
L_{\ell+1}[t,J] &= W^0_{\ell+1}[J,C]h^2[t,C]
\end{aligned}$$

Here semicolon denotes vector concatenation.

The Transformer Layer

Each transformer block in the residual pathway of six "sublayers" the first of which is the self-attention layer.

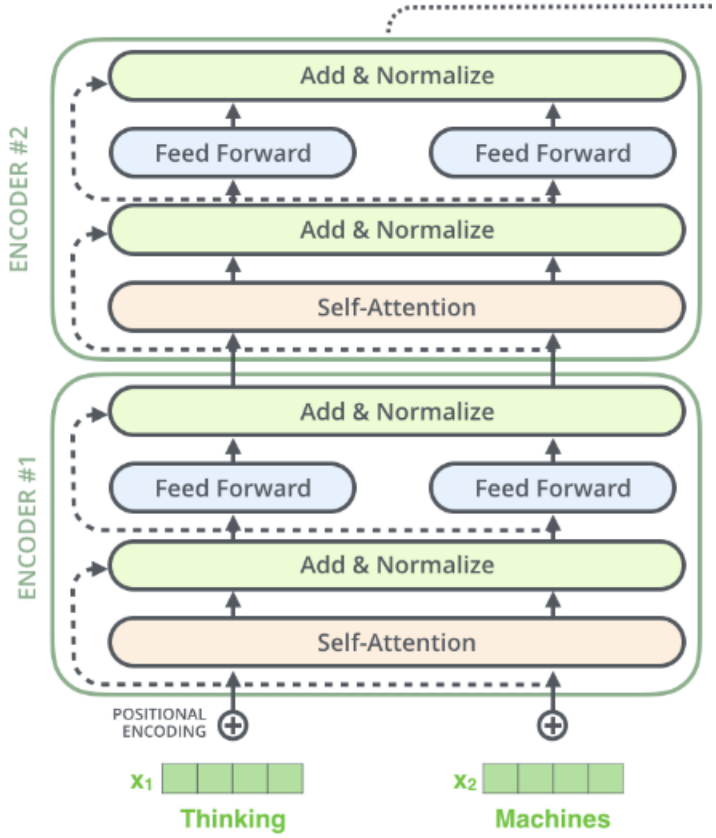
Feed-Forward Layers

The feed-forward layers apply a two-level multi-layer perceptron (MLP) to the vector at each time position independently.

$$h_{\ell+1}[t,i] = \text{ReLU}(W^{\text{FF1}}_{\ell+1}[i,J]\; L_\ell[t,J] - B^{\text{FF1}}_{\ell+1}[i])$$

$$L_{\ell+1}[t,j] = W^{\text{FF2}}_{\ell+1}[j,I]\; h_{\ell+1}[t,I] - B^{\text{FF2}}_{\ell+1}[j]$$

The Transformer



Jay Alammar's blog

END