Time Series Analysis

Ahmed Ebid

Abstract

This report takes a look into the architectures of three models that deal with time series data. The models are IndRNN [S. Li, W. Li, Cook, Zhu, et al. 2018], Transformer [Vaswani et al. 2017], and Transformer-XL [Dai et al. 2019]. The models are compared with RNNs and LSTMs, which are the de facto standard for dealing with sequential data.

1 Introduction

It has been established that when it comes to modeling data of a sequential nature, a RNN offers a clear architectural advantage over a standard feed-forward neural network. The structure of a RNN takes previous elements of a sequence into consideration when processing any element of a sequence. However, RNNs suffer from some well-known issues:

- Vanishing and exploding gradients. In the first case, the gradients go to zero exponentially fast, which results in halting the training. In the second case, the gradients go to infinity exponentially fast, which results in the network being unstable. Both issues are due to the recurrence, which involves multiplying the back-propagation gradient several times.
- Long sequences. If a sequence requires an element to draw a connection to another element that appears much earlier in the sequence, which is common in NLP tasks, RNNs suffer to model such dependencies.

LSTMs are RNNs. However, a LSTM cell includes a cell state and several gates. That makes it more capable of establishing longer-term dependencies. Nevertheless, LSTMs also suffer from some well-known issues:

- LSTMs deals with the problem of vanishing gradients better than standard RNNs. However, they do not remove the problem completely.

- Training LSTMs is time-consuming and requires a lot of resources.
- Some NLP tasks require to establish longer-term dependencies that go beyond the capacity of LSTMs.

RNNs input data need to be passed sequentially one after the other, which is time-consuming. In a Transformer model, the input sequence can be passed in parallel. The Transformer replaces recurrence with Attention. Attention is a mechanism that assigns for each element in 5 a sequence a score with respect to all elements in the sequence. That score indicates how helpful would another elements be in encoding the current element of interest. This introduces the possibility of using GPUs for parallelism which makes it a much faster model.

When using a Transformer model in the context of language modeling, the naive chunking of sequences into fixed-size segments results in a problem known as context fragmentation. Transformer-XL implemented a recurrent connection between segments which solved this issue.

IndRNN introduced a RNN where neurons in a layer are independent of each other, which simplified the RNN architecture. This resulted in a simpler gradient computation, and easier regulation of the recurrent weights, thus tackling the vanishing and exploding gradient issues.

2 Related Work

When it comes to training RNNs, one issue is particularly dominant. That issue is gradient vanishing and exploding. To address this issue, variants of RNNs have been proposed. The LSTM [Hochreiter and Schmidhuber 1997] and the Gated Recurrent Unit (GRU) [Cho et al. 2014] are two popular ones. Both LSTM and GRU employ a recurrent connection along the time steps, and gates to regulate information flow through the network. However, the reliance on gates makes the network computationally complex and not parallelizable. Quasi-Recurrent Neural Network [Bradbury et al. 2016] applies convolutional layers in parallel across time steps, fixing the recurrent connections. This strategy greatly simplifies the computational complexity, but it reduces its capability since the recurrent connections are no longer trainable.

The Transformer model aimed to reduce sequential computation. Other models with the same goal are ByteNet [Kalchbrenner et al. 2017] and ConvS2S [Gehring et al. 2017]. Both models compute hidden representations in parallel for all input and output positions using CNNs. For these models, it is difficult to learn dependencies between distant positions, because the number of operations required to relate signals from two arbitrary input or output positions grow as the distance between positions increases. It grows linearly for ConvS2S and logarithmically for ByteNet. For a Transformer, however, this cost is reduced to a constant number of operations.

When it comes to language modeling, context is of extreme importance. Some papers explored feeding networks a representation of a wider context as an input. [Mikolov and Zweig 2012] manually defines the context representations. It uses a contextual real-valued input vector in association with each word, to convey contextual information about the sentence being modeled. [Dieng et al. 2016] relies on document-level topics learned from data. It directly captures the global semantic meaning, relating words in a document via latent topics.

3 Time Series Data

Time series data do not follow the common assumption needed for most machine learning data sets. Time series data samples are not independent and identically distributed. Each sample is dependent on the previous one. Traditional neural networks can not model this dependence between samples. However, RNNs can.

3.1 RNNs

Recurrent neural networks (RNNs) are a class of neural networks that incorporate the concept of time into its structure. It is designed such that a single sample is a sequence. The term that describes the number of elements in that sequence is called the time steps.

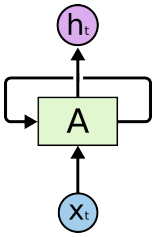


Figure 1: RNNs have a recurrent connection (Image credit goes to *Understanding LSTM Networks* 2015)

A RNN has a recurrent connection where the previous hidden state is an input to the current state. The update of states can be described as follows:

$$h_t = \sigma(Wx_t + Uh_{t-1} + b) \tag{1}$$

where $x_t \in \mathbb{R}^M$ and $h_t \in \mathbb{R}^N$ are the input and hidden state at time step t, respectively. $W \in \mathbb{R}^{N\times M}$, $U \in \mathbb{R}^{N\times N}$ and $b \in \mathbb{R}^N$ are the weights for the current input and the recurrent input, and the bias of the neurons, respectively. σ is an element-wise activation function of the neurons, and N is the number of neurons in this RNN layer [S. Li, W. Li, Cook, Zhu, et al. 2018].

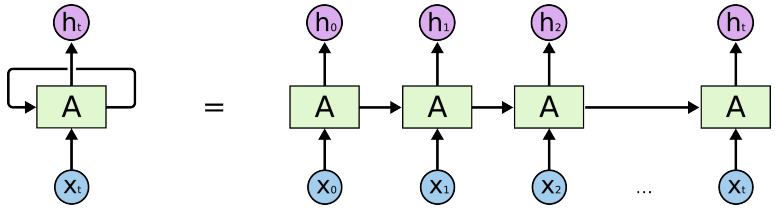


Figure 2: An unrolled RNN layer (Image credit goes to *Understanding LSTM Networks* 2015)

Shown above is a RNN, or a single-layer RNN. A RNN layer is composed of τ RNN cells, where τ indicates the number of time steps and each RNN-cell takes in the input corresponding to a specific time step and the hidden state of the previous RNN-cell.

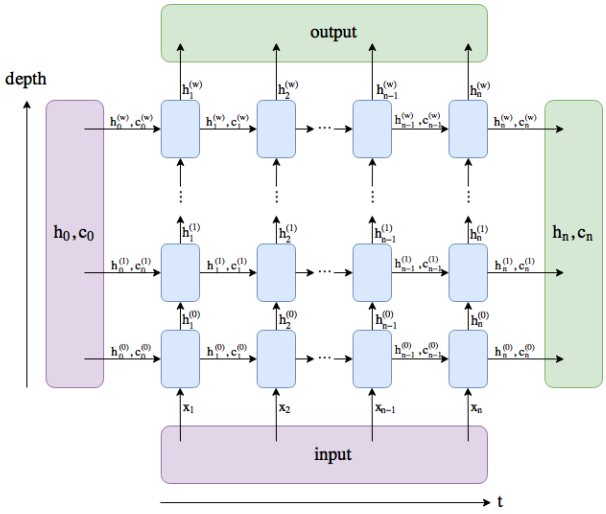


Figure 3: Stacked RNN (Image credit goes to *Pytorch [Basics] — Intro to RNNs* 2020)

A Stacked RNN is formed by vertically stacking RNNs, or single-layer RNNs on top if each other.

The main limitation of RNNs is that they are not capable of learning long term dependencies. As mentioned before, a single input to a RNN is a sequence of a size equal to the number of time steps. At each time step, we have an element. Now, if we are dealing with a long sequence, it becomes hard for a RNN to connect an element that appears late in the sequence to an element that appears much earlier. To tackle this issue, Long Short Term Memory (LSTM) networks were introduced.

LSTM solves complex, artificial long time lag tasks that have never been solved by previous recurrent network algorithms [Hochreiter and Schmidhuber 1997].

3.2 LSTMs

LSTMs are a special kind of RNNs capable of learning long-term dependencies. All RNNs have the form of a chain of repeating modules. In standard RNNs, this repeating module has a single neural network layer with a simple activation function like the tanh.

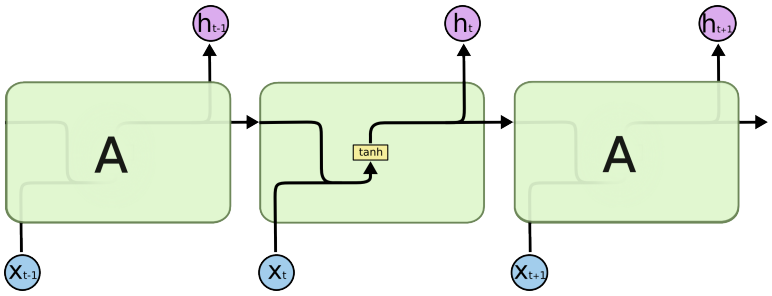


Figure 4: The repeating module of a RNN (Image credit goes to *Understanding LSTM Networks* 2015)

Like standard RNNs, LSTMs have the same chain-like structure of repeating modules. However, the repeating module has four different layers interacting together.

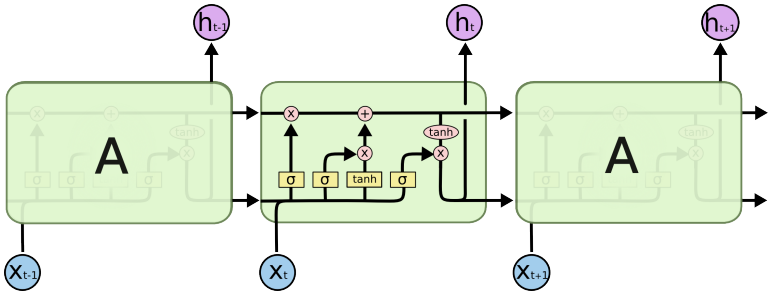


Figure 5: The repeating module of a LSTM (Image credit goes to *Understanding LSTM Networks* 2015)

The main feature of a LSTM cell is the cell state. The cell state runs down the entire layer connecting cells of the layer. The cell state is modified by minor linear interactions, which allows for information to easily flow through it.

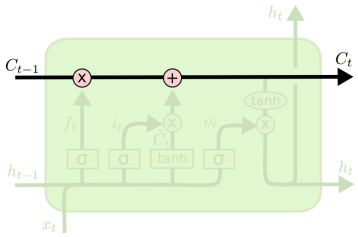
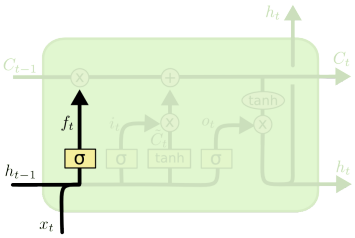


Figure 6: LSTM Cell State (Image credit goes to *Understanding LSTM Networks* 2015)

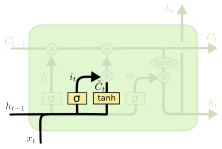
The first step in a LSTM cell is to decide based on the current input and the previous hidden state, how much of the previous cell state should be kept. This is achieved by using a sigmoid activation. The previous cell state is then multiplied by output of the sigmoid.



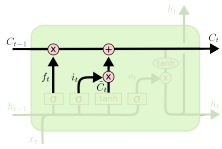
$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

Figure 7: Forget Gate Layer (Image credit goes to *Understanding LSTM Networks* 2015)

The second step is to decide on the information we want to add to the cell state, and how much of it would we like to add. That is, based on the current input and the previous hidden state, by using a *sigmoid* activation decide on how much information, and by using a tanh activation, decide on what information to add to the cell state.



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

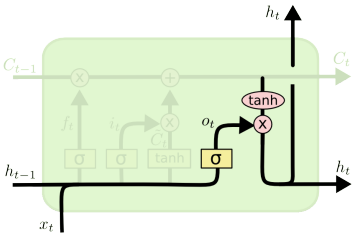


$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Figure 8: Input Gate Layer and Cell Update Layer (Image credit goes to *Understanding LSTM Networks* 2015)

The third and final step is to decide on what to output as a hidden state. The cell state is passed through a tanh layer and the output is multiplied by the factor

generated by the $sigmoid$ layer when the current input and previous hidden state are passed to it.



$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

Figure 9: Ouput Gate and Output Hidden State (Image credit goes to *Understanding LSTM Networks* 2015)

4 Transformer

The Transformer model was proposed by [Vaswani et al. 2017]. It was mainly introduced to deal with Seq2Seq problems. That is, taking a sequence as an input, and returning a sequence as an output. The Transformer uses an Attention mechanism which eliminates the recurrence required in RNNs and allows the model to employ parallelism.

The go-to architecture for dealing with Seq2Seq problems is illustrated in the figure below. For this section, machine translation will be used as a running example of Seq2Seq problems.

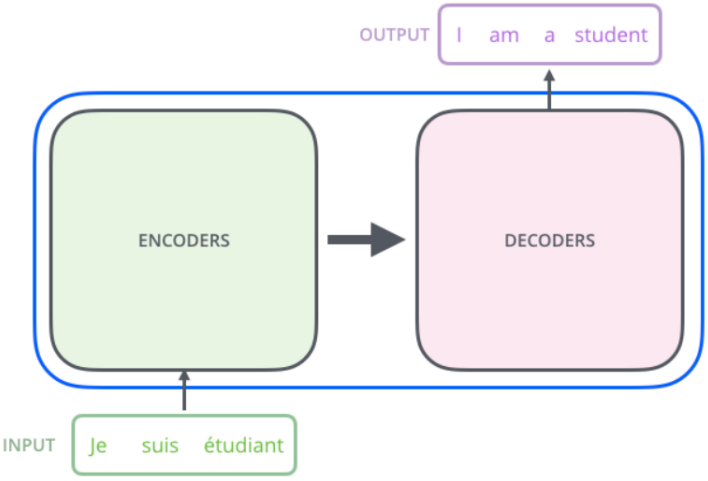


Figure 10: Machine translation as an example of a Seq2Seq problem (Image credit goes to *The Illustrated Transformer* 2018)

Tackling a Seq2Seq problem using a RNN architecture is illustrated in the figure below.

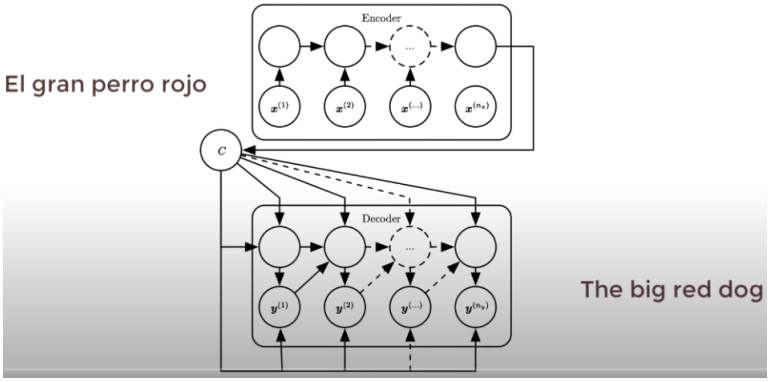


Figure 11: Seq2Seq problem using an RNN architecture (Image credit goes to *Transformer Neural Networks - EXPLAINED! (Attention is all you need)* n.d.)

RNNs input data need to be passed sequentially or serially one after the other. We need inputs of the previous state to make any operations on the current state. Such sequential flow does not make use of today's GPUs very well, which are designed for parallel computation. We need to make use of parallelization for sequential data.

The figure below shows the Transformer architecture.

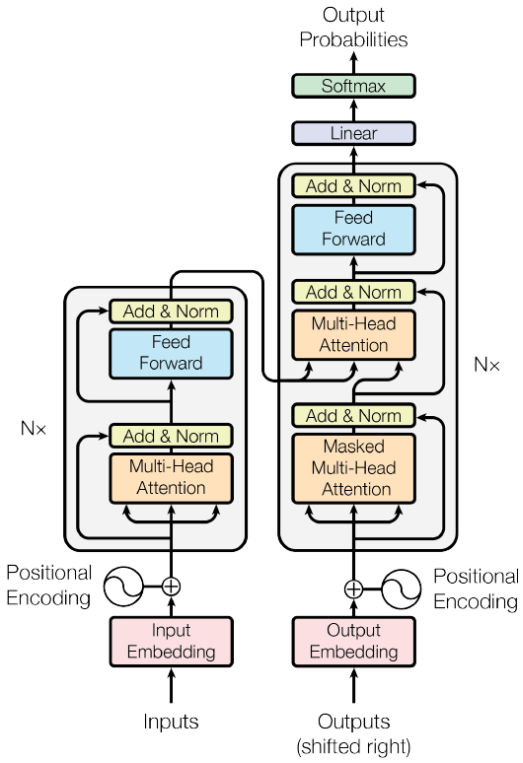


Figure 12: Transformer Architecture (Image credit goes to Vaswani et al. 2017)

The Transformer neural network architecture employs an encoder-decoder architecture much like RNNs. The difference is that the input sequence can be passed in parallel. Consider translating a sentence from French to English. With a RNN encoder we pass an input French sentence one word after the other. The current word's hidden state has dependencies on the previous word's hidden state. The words embeddings are generated one time step at a time. With a Transformer encoder on the other hand there is no concept of time step for the input we pass in all the words of the sentence simultaneously and determine the words embeddings simultaneously.

The figure below illustrates how the Transformer stacks the encoders and decoders. Each encoder and decoder in the figure below contains the components shown in the previous figure. The encoding component is a stack of six encoders, and the decoding component is a stack of decoders of the same number.

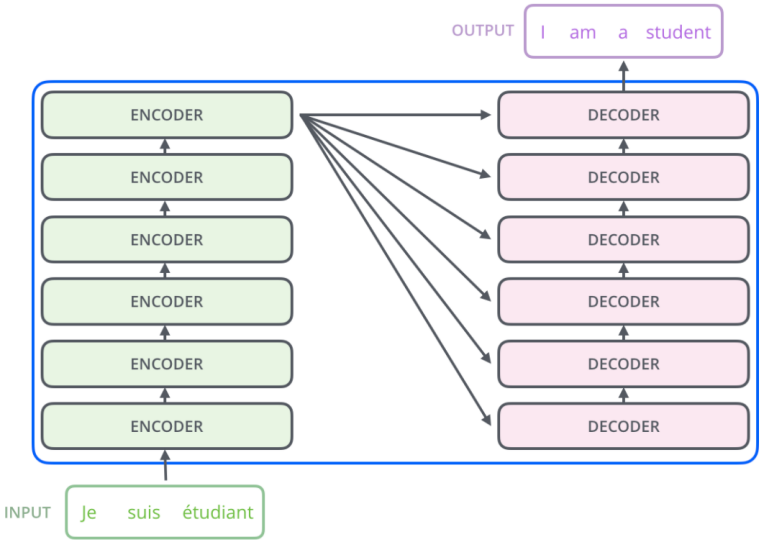


Figure 13: Encoder and Decoder stack in a Transformer (Image credit goes to *The Illustrated Transformer* 2018)

A word embedding is a technique used to represent a word as a fixed size vector. Every word is mapped to a point in space where similar words in meaning are physically closer to each other, this space is called an embedding space. This embedding space maps a word to a vector but the same word in different sentences may have different meanings. This is where positional encoders come in. It is a vector that has information on distances between words and the sentence. The paper uses a sine and cosine function to generate this vector. After passing the French sentence through the input embedding and applying the positional encoding, we get word vectors that have positional information. The encoded input is then passed to the Multi-Head Attention block. The Multi-Head Attention block combines the output from eight Self-Attention blocks.

For the following illustrations we use Self-Attention blocks to explain the concepts behind it. The actual architecture uses Multi-Head Attention blocks instead.

The encoder's inputs first flow through a Self-Attention layer. The outputs of the Self-Attention layer are then fed to a feed-forward neural network. The decoder has both those layers, but between them is an Encoder-Decoder Attention layer that helps the decoder focus on relevant parts of the input sentence.

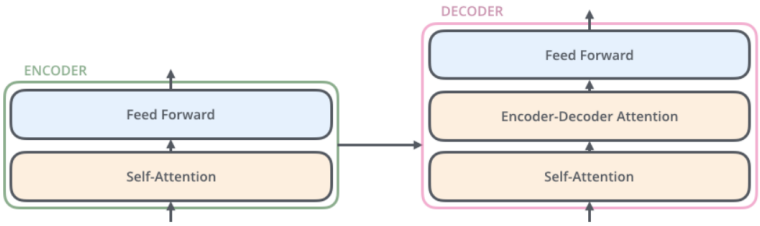


Figure 14: Encoder and Decoder Internal Layers (Image credit goes to *The Illustrated Transformer* 2018)

To understand the concept of Attention, lets a take a look at an example. Consider the following sentence: "The animal didn't cross the street because it was too tired". Attention is a mechanism that assigns for each word in a sentence a score with respect to all words in the sentence. That score indicates how helpful would another words be in encoding the current word of interest. The figure below shows the Attention scores of the word "it" with respect to all words in the sentence.

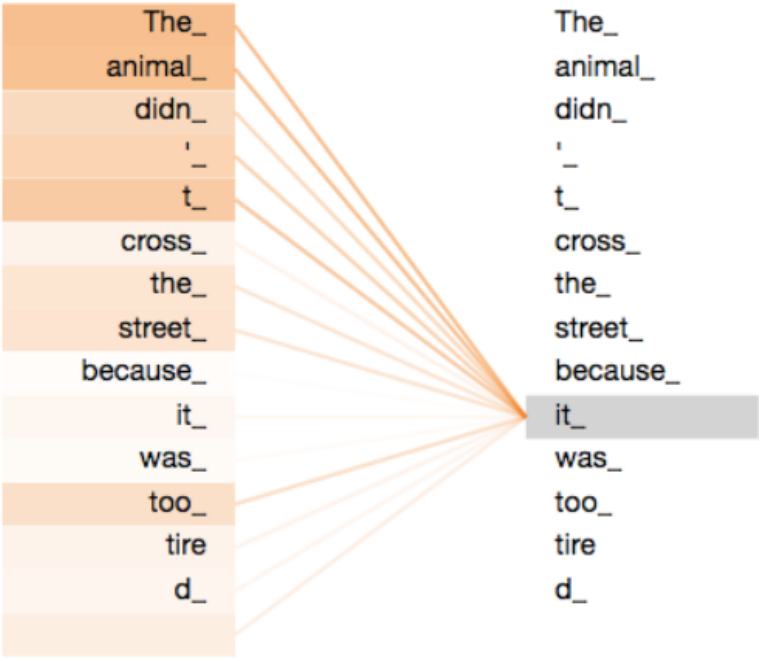


Figure 15: Attention Example (Image credit goes to *The Illustrated Transformer* 2018)

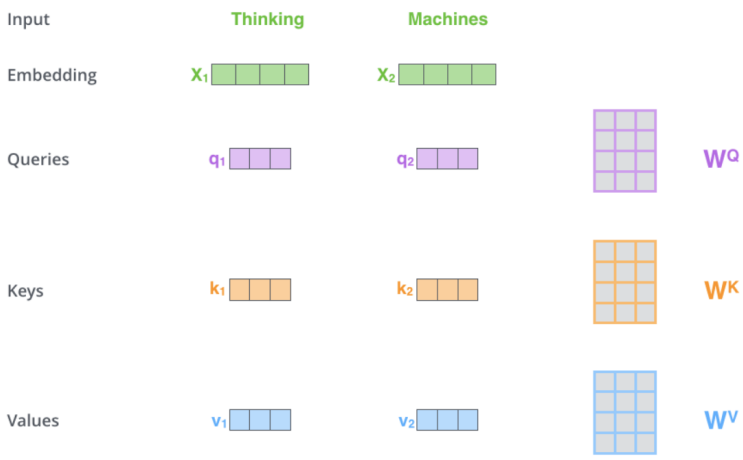


Figure 16: Computing the Queries, Keys, and Values (Image credit goes to *The Illustrated Transformer* 2018)

The figure above shows the first step of computing Self-Attention. For the example sentence above "Thinking Machines". Each word is represented as a vector, and for each vector three other vectors are computed. Namely, the query, key, and value. They are computed by applying a word vector to each of the query, key, and value matrices. The query, key, and value vectors values are determined by the matrix multiplication with weight matrices to be learned. The way Attention is computed using these vectors explains the name of each vector. The illustration above shows the query, key, and value vectors with a smaller dimensionality than the embedding vector. Their dimensionality in the paper is 64, while the embedding and encoder input/output vectors have dimensionality of 512. This is an architecture choice to make the computation of the Multi-Head Attention constant.

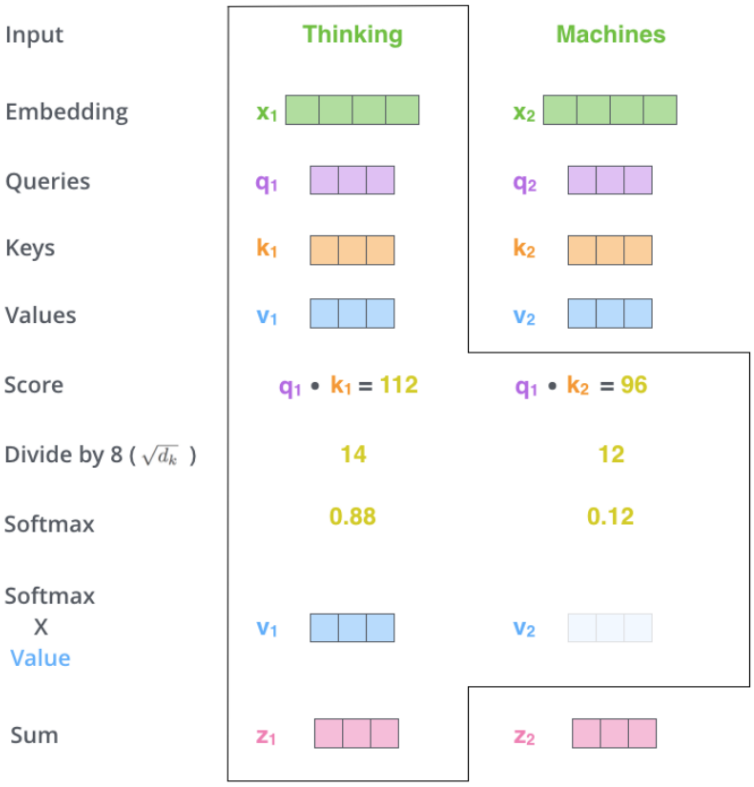


Figure 17: Self-Attention Example (Image credit goes to *The Illustrated Transformer* 2018)

The figure above shows how to compute Self-Attention for the word "Thinking" in the sentence "Thinking Machines". After computing the query, key, and value vectors, we compute a dot product of the query and key vectors. Recall that the dot product of two vectors indicates how similar the two vector are. That is, we are getting a score that represents how similar the query and key vectors are. The score is then divided by 8 (the square root of the dimension of the key vectors used in the paper – 64). This leads to having more stable gradients. Softmax is then applied to normalize the scores so they're all positive and add up to 1.

The value vectors are then multiplied by the softmax score. The intuition behind doing that is to keep the values of words we want to focus on significant and drown out the values of other irrelevant words. The weighted value vectors are then summed up. This final vector represents the Self-Attention vector. The example illustrated in the figure shows the computation of the Self-Attention vector for the first word.

The example illustrated above showed the computation for one word using vectors. In reality, matrix multiplication is used and all words of a sentence are combined in a matrix.

That was the Self-Attention block. The Multi-Head Attention block combines eight Self-Attention blocks to allow the attention vectors of a word to focus on other words also, instead of only focusing on itself. This mechanism provides multiple "representation subspaces". The Multi-Head Attention block has a set of eight query/key/value weight matrices. 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.

The figure below shows how a Multi-Head Attention block combines the eight Self-Attention blocks.

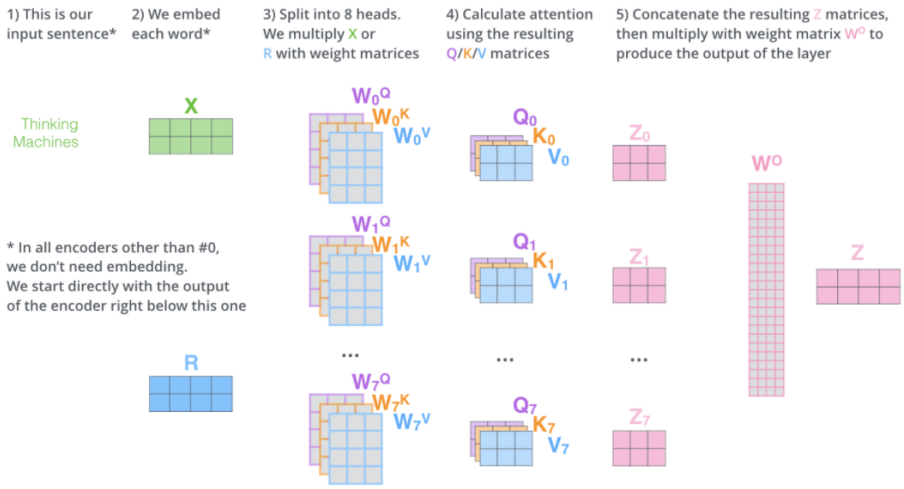


Figure 18: Multi-Head Attention (Image credit goes to *The Illustrated Transformer* 2018)

After encoding, comes decoding. The encoder component starts by processing the input sequence. The output of the top encoder in the encoders stack is then transformed into a set of attention vectors K (keys) and V (values). These are to be used by each decoder in its "encoder-decoder attention" layer which helps the decoder focus on appropriate places in the input sequence. The decoding phase outputs an element of the output sequence one by one until it outputs a special token indicating the end of sentence. The output of each decoding step is fed to the bottom decoder after passing through word embedding and positional encoding. The Self-Attention layers in the decoder operate in a slightly different way than the one in the encoder. The paper calls the attention block in the decoding side masked attention block. It is named as such because while generating the

next output word we can use all the words from the input sentence but only the previous words of the output sentence. If we are going to use all the words in the output sentence, then there would be no learning it would just spit out the next word. So, while performing parallelization with matrix operations we make sure that the matrix will mask the words appearing later so the attention network can't use them.

The "Encoder-Decoder Attention" layer works just like Multi-Head Attention, except it creates its queries matrix from the layer below it, and takes the keys and values matrices from the output of the encoder stack.

The figure below illustrates a step in the decoder translation of the sentence "Je suis étudiant" in French to "I am a student<EOS>" in English.

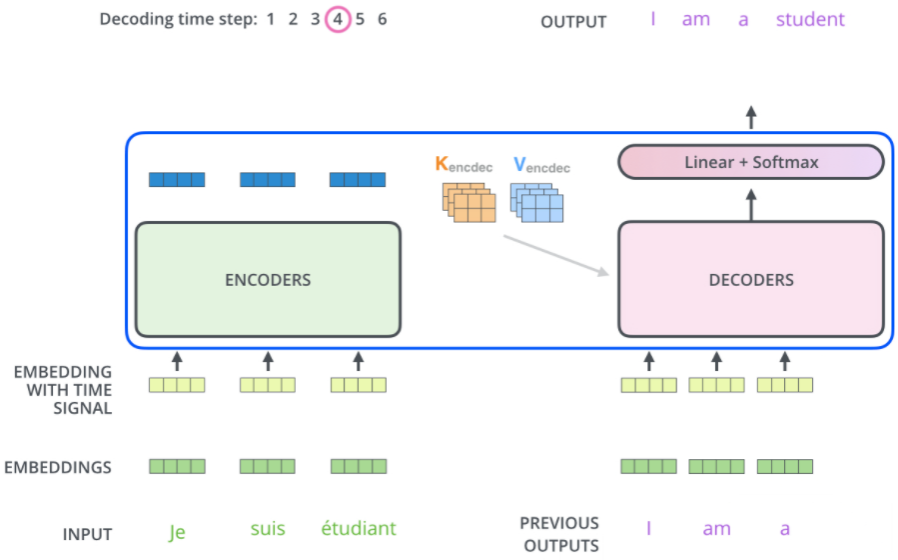


Figure 19: Decoder Stack in Action (Image credit goes to *The Illustrated Transformer* 2018)

To finally go from the output vector of the decoders to a word, the output vector goes through the linear layer, which is a fully connected neural network the outputs a logits vector. The logits vector is of size equal to the number of words the model learned from the training dataset. Each element of the logits vector corresponds to the score of a unique word. The softmax layer then transforms the logits scores into probabilities. The element with the highest probability is chosen, and the word associated with it is produced as the output for this time step. The figure below illustrates an example.

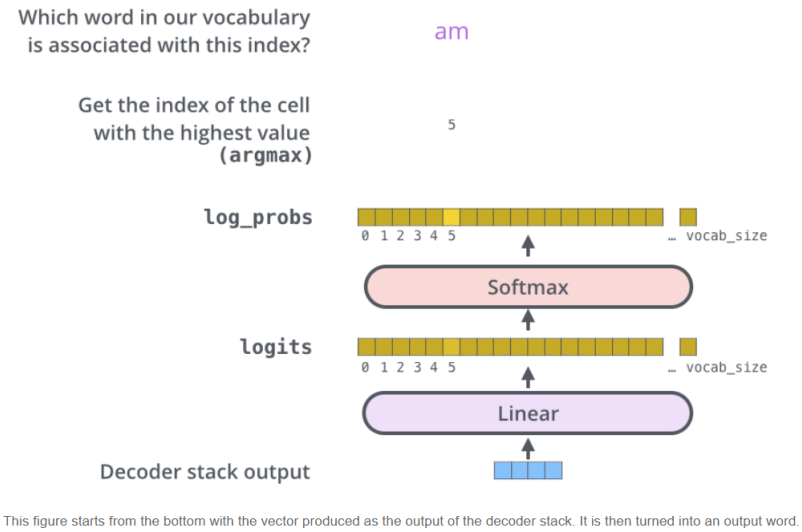


Figure 20: From Decoder Output to Final Output (Image credit goes to *The Illustrated Transformer* 2018)

5 Transformer-XL

Transformer-XL (meaning extra-long) was introduced by [Dai et al. 2019] to deal with the language modeling problem. Language modeling is the task of assigning a probability to sentences in a language. Language models also assign a probability for the likelihood of a given word (or a sequence of words) to follow a sequence of words. Language modeling is particularly difficult, because it requires modeling long-term dependencies. When it comes to modeling long-term dependencies in sequential data, RNNs, despite being the standard solution to language modeling, are difficult to optimize due to gradient vanishing and explosion. LSTMs are better, but LSTMs are not sufficient to fully address this issue. On the other hand, the direct connections between long-distance word pairs built in attention mechanisms might ease optimization and enable the learning of long-term dependencies.

[Al-Rfou et al. 2018] tackles the language modeling problem using Transformers. Training is performed on separated fixed-length segments of a few hundred characters. Without any information flow across segments, the model cannot capture any long-term dependencies beyond the fixed context length. Moreover, the fixed-length segments are constructed by choosing a consecutive chunk of symbols without paying attention to the sentence or semantic boundaries. This problem is referred to as *context fragmentation*. The model lacks necessary contextual information needed to accurately predict the first few symbols, leading to inefficient optimization and inferior performance.

Tranformer-XL mainly offers two contributions. The first is the segment-level recurrence mechanism. Tranformer-XL introduces the notion of recurrence by reusing the hidden states obtained from the previous segment instead of computing the hidden states from scratch for each new segment. A recurrent connection between the segments is built by reusing the previous hidden state as a memory for the current state. Through these recurrent connections, information flows allowing long-term dependencies to be modeled. Tranformer-XL resolves the *context fragmentation* problem, because it is able to pass information from the previous segments to the current segment. The second is a novel positional encoding scheme. To avoid temporal confusion when reusing states, relative positional encodings rather than absolute ones are used to generalize to attention lengths longer than the one observed during training.

The task of language modeling is to estimate the joint probability of a given corpus of tokens $\boldsymbol{x} = (x_1, \ldots, x_T)$. The joint probability is $P(x) = \prod_t P(x_t|x_{<t})$. The standard neural approach is to model the conditional probability. Specifically, a neural network is used to encode the context $x_{<t}$ into a fixed size hidden state, which is multiplied with the word embeddings to obtain the logits. We then get a probability distribution over the next token by passing the logits to a Softmax function.

[Al-Rfou et al. 2018] tackles the language modeling problem by splitting the corpus into fixed-size segments, and only train the model within each segment, ignoring all contextual information from previous segments. This paper calls this model the "vanilla model". Following this training scheme does not allow information to flow across segments in both forward and backward passes. By using a fixed-length context, the vanilla model suffers from two critical limitations. First, the model cannot establish any dependency that is longer than the segment length. Second, the naive chunking of the corpus into segments results in *context fragmentation*. The evaluation step of the vanilla model is performed by processing a segment of the same length used in training, and only making a prediction at the last position. Too generate the next prediction, the segment is shifted to the right by one position, and that new segment has to be processed from scratch. Obviously, this evaluation procedure is very expensive. It does that though to ensure that each prediction utilizes the longest possible context exposed during training. The evaluation phase also suffers from *context fragmentation*. The figure below shows the training and evaluation phases of the vanilla Transformer.

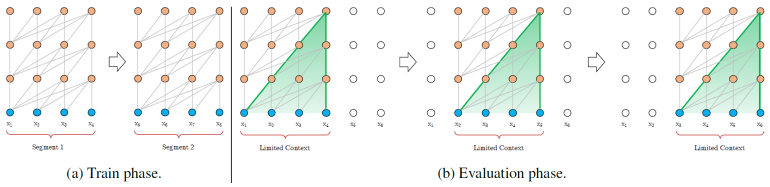


Figure 21: Vanilla Model with a Segment Length of Size 4 (Image credit goes to Dai et al. 2019)

Transformer-XL introduces a recurrence mechanism to mitigate the limitations of a fixed-length context. When processing a segment, the previous segment's hidden state is fixed and cached to be reused as an extended segment. However, there is no recurrent back propagation. Unlike a RNN, the back propagation does not flow through the time steps. The previous segment is only used in the forward pass. Even though the gradient does not flow back to the previous segment, using the previous hidden state as an additional input allows the network to exploit information from the past. This equips the model with better capabilities of modeling longer-term dependencies and avoiding *context fragmentation*.

Formally, let the two consecutive segments of length L be $\boldsymbol{s_\tau} = (x_{\tau,1}, \ldots, x_{\tau,L})$ and $\boldsymbol{s_{\tau+1}} = (x_{\tau+1,1}, \ldots, x_{\tau+1,L})$ respectively. Denoting the n-th layer hidden state sequence produced for the τ-th segment s_τ by $h_\tau^n \in \mathbb{R}^{L \times d}$, where d is the hidden dimension. Then, the n-th layer hidden state for segment $s_{\tau+1}$ is produced (schematically) as follows,

$$
\begin{aligned}
\tilde{\mathbf{h}}_{\tau+1}^{n-1} &= \left[\text{SG}\left(\mathbf{h}_{\tau}^{n-1}\right) \circ \mathbf{h}_{\tau+1}^{n-1}\right], \\
\mathbf{q}_{\tau+1}^{n}, \mathbf{k}_{\tau+1}^{n}, \mathbf{v}_{\tau+1}^{n} &= \mathbf{h}_{\tau+1}^{n-1}\mathbf{W}_q^\top, \widetilde{\mathbf{h}}_{\tau+1}^{n-1}\mathbf{W}_k^\top, \widetilde{\mathbf{h}}_{\tau+1}^{n-1}\mathbf{W}_v^\top, \\
\mathbf{h}_{\tau+1}^{n} &= Transformer - Layer\left(\mathbf{q}_{\tau+1}^{n}\mathbf{k}_{\tau+1}^{n}, \mathbf{v}_{\tau+1}^{n}\right)
\end{aligned}
\tag{2}
$$

where the function $SG(\cdot)$ stands for stop-gradient, the notation $[h_u \cdot h_v]$ indicates the concatenation of two hidden sequences along the length dimension, and W denotes model parameters. Compared to the standard Transformer, the critical difference lies in that the key $k_{\tau+1}^n$ and value $v_{\tau+1}^n$ are conditioned on the extended context $\tilde{h}_{\tau+1}^{n-1}$ and hence h_τ^{n-1} cached from the previous segment. [Dai et al. 2019]

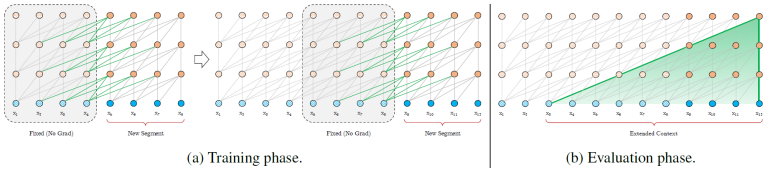


Figure 22: Transformer-XL with a Segment Length of Size 4 (Image credit goes to Dai et al. 2019)

A segment-level recurrence is created in the hidden states because of the recurrence mechanism that is applied to every two consecutive segments of a corpus. Analogous to the concept of the receptive field in convolutional layers, the effective context utilized can go beyond just two segments. Unlike the same-layer recurrence in conventional RNN-LMs, the recurrent dependency between $h_{\tau+1}^{n}$ and h_{τ}^{n-1} shifts one layer downwards per-segment. As a result, the largest possible dependency length grows linearly w.r.t. the number of layers as well as the segment length, i.e., $O(N \times L)$, as visualized by the shaded area in figure 22(b). This recurrence scheme also facilities significantly faster evaluation. Unlike the vanilla model that re-computes every segment from scratch, Transformer-XL uses the cached representations of the previous segments.

The problem with reusing the previous segments is that the positional encodings would not be coherent. For example, if the previous segment has the following contextual positions: $[0, 1, 2, 3]$, when a new segment is processed, the contextual positions for the combination of the two segments would be $[0, 1, 2, 3, 0, 1, 2, 3]$. Clearly, the semantics of each position id is incoherent throughout the combined sequence. To deal with this issue, Transformer-XL introduces a novel relative positional encoding scheme. Recall the computation of the attention score shown in figure 23.

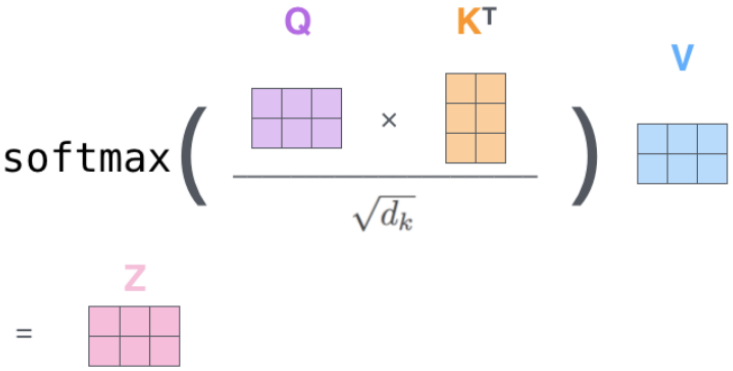


Figure 23: Attention Score Computation (Image credit goes to *The Illustrated Transformer* 2018)

In the novel positional encoding scheme introduced in Transformer-XL, the simple multiplication of $(Q_i \times K_j)$ is expanded to include four terms. The first term represents the *content weight*, that is the original $(Q_i \times K_j)$. Keep in mind that up to this point there is no positional encoding. The usual positional encoding applied on the input before going through the encoder stack is not applied. The second term is a *positional bias* with respect to the current query (Q_i). Q_i is encoding information about the current token. The positional bias term measures

the distance between the tokens i and j, instead of using the absolute position of the current token. Recall that positional encoding in the Transformer model uses a sinusoidal function that takes in the absolute position of a token. However, in Transformer-XL, the sinusoidal function takes in distance between tokens. The third term is a learned *global content bias*. The model adds a learned vector that measures the importance of the other token content. The fourth and final term is a learned *global bias*. It is also a learned vector, but it measures the importance based only on the distance between the tokens.

6 IndRNN

Independently Recurrent Neural Network (IndRNN) was proposed by [S. Li, W. Li, Cook, Zhu, et al. 2018] with an new architecture, where neurons in the same layer are independent of each other and are instead connected across layers. RNNs are diffcult to train, because of the gradient vanishing and exploding problem, which is caused by the recurrent connections with repeated multiplication of the recurrent weight matrix. Moreover, a RNN-cell within a RNN-layer applies the same activation shown in equation (1). This means that each neuron within a RNN-cell has a recurrent connection to all other neurons of the previous hidden state. This dependence makes it difficult to understand and interpret the patterns each neuron responds to without considering the other neurons. Additionally, because of the recurrent connections, with each time step, matrix multiplication is computed. Due to the recurrence, this computation can not be parallelized. This results in a very time-consuming process when the RNN deals with a big number of time steps.

IndRNN processes the recurrent inputs as follows:

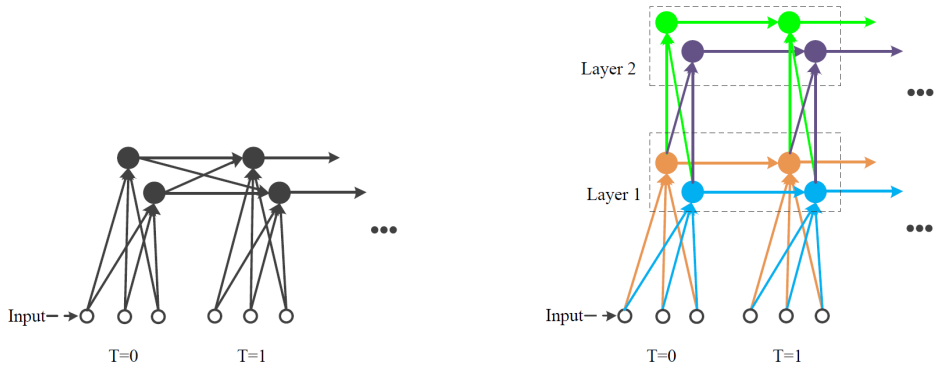
$$h_t = \sigma(W x_t + u \odot h_{t-1} + b) \tag{3}$$

where $x_t \in \mathbb{R}^M$ and $h_t \in \mathbb{R}^N$ are the input and hidden state at time step t, respectively. $W \in \mathbb{R}^{N \times M}$, $u \in \mathbb{R}^N$ and $b \in \mathbb{R}^N$ are the weights for the current input and the recurrent input, and the bias of the neurons, respectively. $\odot$ is the Hadamard product, σ is an element-wise activation function of the neurons, and N is the number of neurons in this RNN layer. Compared with the traditional RNN where the recurrent weight U is a matrix and processes the recurrent input using matrix product, the recurrent weight u in IndRNN is a vector and processes the recurrent input with element-wise vector product. Each neuron in one layer is independent from others, thus termed as "independently recurrent". For the n-th neuron, the hidden state $h_{n,t}$ can be obtained as:

$$h_{n,t} = \sigma(w_n x_t + u_n h_{n,t-1} + b_n) \tag{4}$$

here w_n, u_n and b_n are the n-th row of the input weight, recurrent weight and bias, respectively [S. Li, W. Li, Cook, Zhu, et al. 2018].

The Hadamard product is element-wise matrix multiplication. It can only be applied on matrices with same dimension. For vectors, it is an element-wise vector product. The figure below schematically demonstrates the difference in the recurrent connections between traditional RNNs and IndRNN.



(a) The conventional simple RNN with recurrent connections connecting all neurons over time.

(b) The proposed IndRNN with independent recurrent connections making each neuron in one layer independent while connected over layers.

Figure 24: Illustration of a conventional simple RNN and the proposed IndRNN unfolded in time. Each solid dot represents a neuron in a layer and each line represents a connection. (Image credit goes to S. Li, W. Li, Cook, Gao, et al. 2019)

This new formulation results in the following advantages when compared to traditional RNNs:

- IndRNN is able to process longer sequences. The recurrent weights u are regulated, which solves the gradient vanishing and exploding problem. Experiments showed that IndRNN is capable of dealing with sequences with over 5000 time steps.
- Unlike traditional RNNs, we can use a non-saturated activation function such as the $ReLU$. This results in better behaved gradient back propagation through time.
- Neurons within an IndRNN layer can be interpreted easily. This is due to the independence between neurons in each layer. Every neuron's behaviour can be interpreted individually.
- IndRNN offers reduced complexity. The matrix product applied on recurrent connections is replaced with element-wise vector product, which is much more efficient.

The gradient back propagation through time for neurons in and IndRNN layer can be performed independently for each neuron, because the neurons in each layer are independent of each other. Ignoring the bias, the activation for the n-th neuron is $h_{n,t} = \sigma(w_n x_t + u_n h_{n,t-1})$. Let J_n be the objective to be minimized at time step T. Assume the time steps follow the following order $[1 \dots t \dots T \dots]$. The gradient back propagated to the time step t is shown below.

$$\begin{aligned} \frac{\partial J_n}{\partial h_{n,t}} &= \frac{\partial J_n}{\partial h_{n,T}} \frac{\partial h_{n,T}}{\partial h_{n,t}} = \frac{\partial J_n}{\partial h_{n,T}} \prod_{k=t}^{T-1} \frac{\partial h_{n,k+1}}{\partial h_{n,k}} \\ &= \frac{\partial J_n}{\partial h_{n,T}} \prod_{k=t}^{T-1} \sigma'_{n,k+1} u_n = \frac{\partial J_n}{\partial h_{n,T}} u_n^{T-t} \prod_{k=t}^{T-1} \sigma'_{n,k+1} \end{aligned} \tag{5}$$

where $\sigma'_{n,k+1}$is the derivative of the element-wise activation function.

The gradient involves three terms. The first is mainly influenced by the objective function, the second is an exponential term of a scalar value u_n which can be easily regulated, and the third is the and the gradient of the activation function which is often bounded in a certain range. In a traditional RNN, this gradient is $\frac{\partial J}{\partial h_T} \prod_{k=t}^{T-1} diag(\sigma'(h_{k+1}))U^T$ where $diag(\sigma'(h_{k+1}))$ is the Jacobian matrix of the element-wise activation function.

The gradient of IndRNN is directly dependent on the recurrent weight (u_n) which is changed by a small magnitude according to the learning rate. On the other hand, the gradient of a RNN directly depends on a matrix product, which is mainly determined by its eigenvalues and can be changed significantly even if the change to each matrix entries is small [S. Li, W. Li, Cook, Zhu, et al. 2018]. This equips IndRNN with a more robust training when compared with a traditional RNN.

By regulating the exponential term $u_n^{T-t} \prod_{k=t}^{T-1} \sigma'_{n,k+1}$ to an appropriate range, and ignoring the gradient back propagated from the objective at time step T, the gradient exploding and vanishing problem over time can be solved.

7 Results

This section take a look at some of the experiments carried out by each paper that introduced the models.

7.1 Transformer

The Transformer model was tested on the WMT 2014 English-to-German translation task, and the WMT 2014 English-to-French translation task. The metrics used in this benchmark are BLEU (bilingual evaluation understudy) and FLOPS (floating point operations per second). BLEU evaluates the quality of machine-translated text from one language to another [*BLEU* n.d.]. Higher is better. FLOPS

measures computer performance in making computations that require floating-point calculations [*FLOPS* n.d.]. Lower is better. [Vaswani et al. 2017] estimated the number of floating point operations used to train a model by multiplying the training time, the number of GPUs used, and an estimate of the sustained single-precision floating-point capacity of each GPU. At the time of its introduction, the Transformer model achieved state-of-the-art results in that benchmark. The figure below shows that benchmark results.

Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [18]	23.75			
Deep-Att + PosUnk [39]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [38]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [9]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [32]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [39]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [38]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [9]	26.36	**41.29**	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer (base model)	27.3	38.1	$\mathbf{3.3 \cdot 10^{18}}$	
Transformer (big)	**28.4**	**41.8**	$2.3 \cdot 10^{19}$	

Figure 25: WMT 2014 English-to-German and WMT 2014 English-to-French (Image credit goes to Vaswani et al. 2017)

7.2 Transformer-XL

The Transformer-XL was tested on both word-level and character-level language modeling datasets.

WikiText-103 is the largest available word-level language modeling benchmark with long-term dependency. The results on that benchmark is shown in the figure below.

Model	#Param	PPL
Grave et al. (2016b) - LSTM	-	48.7
Bai et al. (2018) - TCN	-	45.2
Dauphin et al. (2016) - GCNN-8	-	44.9
Grave et al. (2016b) - Neural cache	-	40.8
Dauphin et al. (2016) - GCNN-14	-	37.2
Merity et al. (2018) - QRNN	151M	33.0
Rae et al. (2018) - Hebbian + Cache	-	29.9
Ours - Transformer-XL Standard	151M	**24.0**
Baevski and Auli (2018) - Adaptive Input$^{\diamond}$	247M	20.5
Ours - Transformer-XL Large	257M	**18.3**

Figure 26: WikiText-103 (Image credit goes to Dai et al. 2019)

The metric for that benchmark is perplexity. Perplexity measures how well a probability model predicts a sample [*Perplexity Intuition (and its derivation)* n.d.]. Lower is better. At the time of its introduction, the Transformer-XL model achieved state-of-the-art results in that benchmark.

enwik8 is a character-level language modeling benchmark. The results on that benchmark is shown in the figure below.

Model	#Param	bpc
Ha et al. (2016) - LN HyperNetworks	27M	1.34
Chung et al. (2016) - LN HM-LSTM	35M	1.32
Zilly et al. (2016) - RHN	46M	1.27
Mujika et al. (2017) - FS-LSTM-4	47M	1.25
Krause et al. (2016) - Large mLSTM	46M	1.24
Knol (2017) - cmix v13	-	1.23
Al-Rfou et al. (2018) - 12L Transformer	44M	1.11
Ours - 12L Transformer-XL	41M	**1.06**
Al-Rfou et al. (2018) - 64L Transformer	235M	1.06
Ours - 18L Transformer-XL	88M	1.03
Ours - 24L Transformer-XL	277M	**0.99**

Figure 27: enwik8 (Image credit goes to Dai et al. 2019)

The metric for that benchmark is BPC (Bits-per-character). BPC measures the average number of bits needed to encode on character [*Evaluation Metrics for*

Language Modeling n.d.]. Lower is better. At the time of its introduction, the Transformer-XL model achieved state-of-the-art results in that benchmark.

7.3 IndRNN

An experiment used to showcase the IndRNN's speed is the adding problem. In the adding problem, the input is two sequences of the same length. The first sequence values are randomly sampled from a uniform [0, 1] distribution. The second sequence contains two entries with the value one, and the rest are zeros. The output is the sum of the two entries in the first sequence indexed by the entries where the values are one in the second sequence. To show the compatibilities of modeling long-term memory, the experiment was applied with sequence lengths of 100, 500, and 1000. The RNN models included in the experiments for comparison are the traditional RNN with tanh, LSTM, and IRNN.

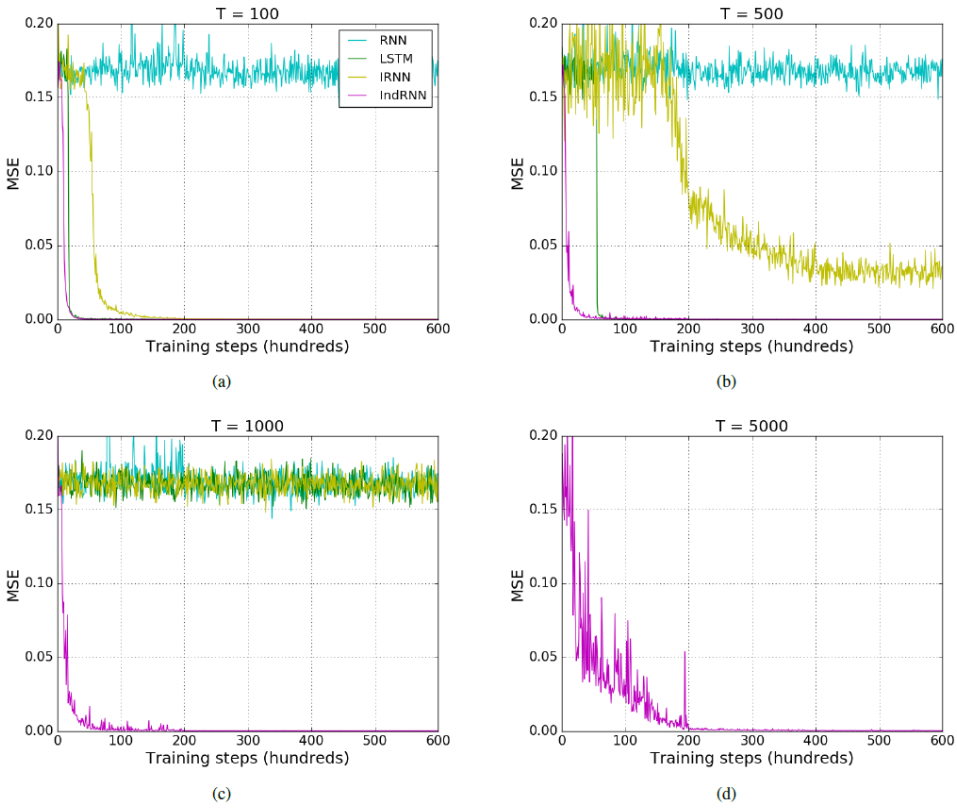


Figure 28: Adding Problem (Image credit goes to S. Li, W. Li, Cook, Zhu, et al. 2018)

For (T=100), all models except the RNN converged. For (T=500), the LSTM takes more time to converge compared with the IndRNN. The IRNN takes even longer,

and does not fully converge. For (T=1000), the IRNN and LSTM can no longer converge. The IndRNN, however, manages to converge to a small loss value very quickly. Even for (T=5000), the IndRNN manages to converge in a reasonable time.

8 Models Comparison

8.1 RNN vs LSTM

RNNs are flexible. What facilities this flexibility is the usage of time steps at each layer. By varying the number of time steps considered at the input and output layers, one can formulate different types of problems. Namely, one-to-one, one-to-many, many-to-one, and many-to-many.

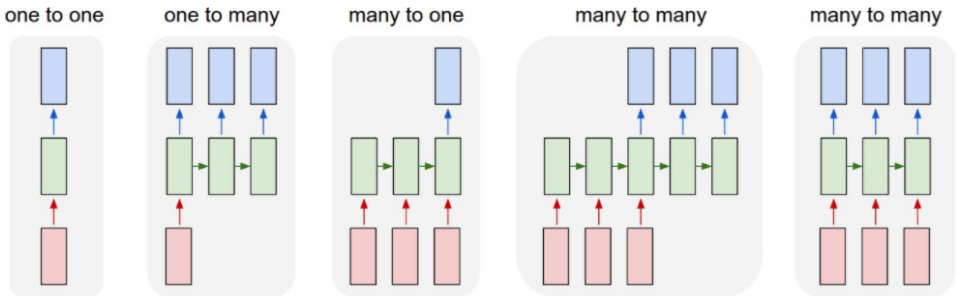


Figure 29: Flexibility of RNNs. (Image credit goes to *The Unreasonable Effectiveness of Recurrent Neural Networks* 2015)

A LSTM is a RNN. Whatever applies in terms of functionality to a RNN also applies to a LSTM. Afterall, a LSTM-cell is just a more complex RNN-cell. One with several gates.

8.2 RNN vs IndRNN

In terms of functionalities, an IndRNN is still a RNN. They share the same functionalities. The IndRNN offers the following advantages over a standard RNN:

- IndRNN is able to process longer sequences. The recurrent weights u are regulated, which solves the gradient vanishing and exploding problem. Experiments showed that IndRNN is capable of dealing with sequences with over 5000 time steps.
- Unlike traditional RNNs, we can use a non-saturated activation function such as the $ReLU$. This results in better behaved gradient back propagation through time.
- Neurons within an IndRNN layer can be interpreted easily. This is due to the independence between neurons in each layer. Every neuron's behaviour can be interpreted individually.

Table 1: Summary of advantages and disadvantages of RNNs and LSTMs

	Advantages	Disadvantages
RNNs	• Can deal with sequential data • Flexible problem formulation	• Slow training • Vanishing and exploding gradient • Can not model long sequences
LSTMs	• Can model longer sequences when compared with a standard RNN • Does not suffer from vanishing and exploding gradient as much as a standard RNN	• Slow training • Some sequences are too long for LSTMs

- IndRNN offers reduced complexity. The matrix product applied on recurrent connections is replaced with element-wise vector product, which is much more efficient.

8.3 RNN vs Transformer

The Transformer's attention mechanism offers the following advantages over a standard RNN:

- RNNs input data need to be passed sequentially. We need inputs of the previous state to make any operations on the current state. However, with Attention, the entire input can be passed at once.
- With RNNs, the direct connection is between a time step and the previous one. However, with attention, an attention score is computed for each element in the input sequence with respect to all other elements in the input sequence.

8.4 Transformer vs Transformer-XL

We use the context of language modeling to compare between the vanilla Transformer [Al-Rfou et al. 2018] and Transformer-XL [Dai et al. 2019]. The vanilla Transformer model suffers from the following limitations:

- Training is applied on separated fixed-length segments of a few hundred characters. Without any information flow across segments, the model cannot capture any long-term dependencies beyond the fixed context length.

- Fixed-length segments are constructed by choosing a consecutive chunk of symbols without paying attention to the sentence or semantic boundaries. This problem is referred to as context fragmentation.

Transformer-XL introduces the notion of recurrence by reusing the hidden states obtained from the previous segment instead of computing the hidden states from scratch for each new segment. Through these recurrent connections, information flows allowing long-term dependencies to be modeled. Transformer-XL resolves the context fragmentation problem, because it is able to pass information from the previous segments to the current segment.In addition, because of the recurrent connection,it has better capabilities of modeling longer-term dependencies.

9 Conclusion

The two main issues involved with modeling sequential data are vanishing and exploding gradients and the sequence being too long. These were the issues each model reviewed in this report tried to tackle. The Transformer model introduced Attention and exploited parallelism. The Transformer-XL model solved the context fragmentation issue of the vanilla Transformer model by using a recurrent connection between segments. The IndRNN simplified the hidden state formulation and the gradient computation, making it easy to regulate the gradients.

References

BLEU (n.d.). URL: https://en.wikipedia.org/wiki/BLEU.

Bradbury, James, Stephen Merity, Caiming Xiong, and R. Socher (2016). "Quasi-Recurrent Neural Networks." In: *arXiv:1611.01576v2*.

Cho, Kyunghyun, B. V. Merrienboer, Çaglar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio (2014). "Learning phrase representations using rnn encoder-decoder for statistical machine translation." In: *arXiv preprint arXiv:1406.1078*.

Dai, Zihangi, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V. Le, and Ruslan Salakhutdinov (2019). "Transformer-XL: Attentive Language Models Beyond a Fixed-Length Context." In: *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*.

Dieng, Adji B, Chong Wang, Jianfeng Gao, and John Paisley (2016). "Topicrnn: A recurrent neural network with long-range semantic dependency." In: *arXiv preprint arXiv:1611.01702*.

Evaluation Metrics for Language Modeling (n.d.). URL: https://thegradient.pub/understanding-evaluation-metrics-for-language-models.

FLOPS (n.d.). URL: https://en.wikipedia.org/wiki/FLOPS.

Gehring, Jonas, M. Auli, David Grangier, Denis Yarats, and Yann Dauphin (2017). "Convolutional sequence to sequence learning." In: *arXiv preprint arXiv:1705.03122v2*.

Hochreiter, Sepp and Jurgen Schmidhuber (1997). "LONG SHORT-TERM MEMORY." In: *Neural Computation 9(8):1735-1780*.

Kalchbrenner, Nal, Lasse Espeholt, K. Simonyan, A. Oord, A. Graves, and K. Kavukcuoglu (2017). "Neural machine translation in linear time." In: *arXiv preprint arXiv:1610.10099v2*.

Li, Shuai, Wanqing Li, Chris Cook, Yanbo Gao, and Ce Zhu (2019). "Deep Independently Recurrent Neural Network(IndRNN)." In: *arXiv:1910.06251v2 [cs.CV]*.

Li, Shuai, Wanqing Li, Chris Cook, Ce Zhu, and Yanbo Gao (2018). "Independently Recurrent Neural Network (IndRNN): Building A Longer and Deeper RNN." In: *arXiv:1803.04831v3 [cs.CV]*.

Mikolov, Tomas and Geoffrey Zweig (2012). "Context dependent recurrent neural network language model." In: *SLT, 12(234-239):8*.

Perplexity Intuition (and its derivation) (n.d.). URL: https://towardsdatascience.com/perplexity-intuition-and-derivation-105dd481c8f3.

Pytorch [Basics] — Intro to RNNs (2020). URL: https://towardsdatascience.com/pytorch-basics-how-to-train-your-neural-net-intro-to-rnn-cb6ebc594677.

Al-Rfou, Rami, Dokook Choe, Noah Constant, Mandy Guo, and Llion Jones (2018). "Character-level language modeling with deeper self-attention." In: *arXiv preprint arXiv:1808.04444*.

The Illustrated Transformer (2018). URL: http://jalammar.github.io/illustrated-transformer/.

The Unreasonable Effectiveness of Recurrent Neural Networks (2015). URL: http://karpathy.github.io/2015/05/21/rnn-effectiveness/.

Transformer Neural Networks - EXPLAINED! (Attention is all you need) (n.d.). URL: https://www.youtube.com/watch?v=TQQlZhbC5ps.

Understanding LSTM Networks (2015). URL: http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin (2017). "Attention Is All You Need." In: *arXiv:1706.03762v5 [cs.CL]*.