Dealing with Sequences: Recurrent Neural Network (RNN) layer

For a function that takes sequence $\mathbf{x^{(i)}}$ as input and creates sequence $\mathbf{y}$ as output we had two choices for implementing the function.

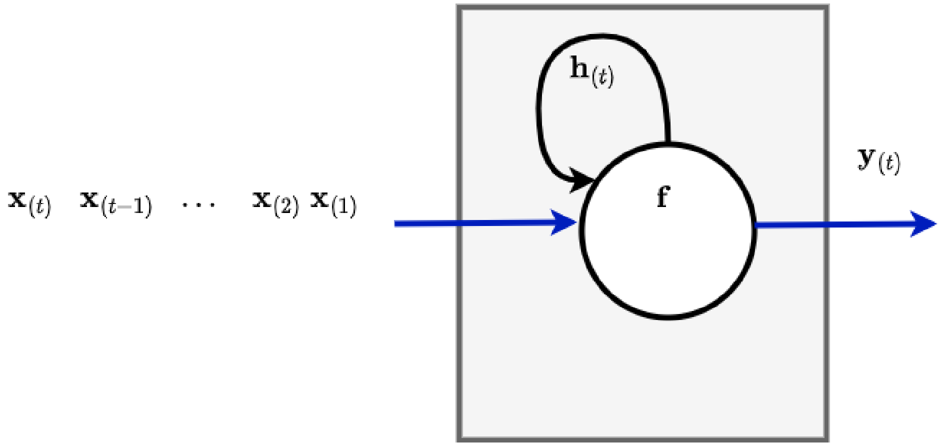
The RNN implements the function as a "loop"

- A function that taking **a single** $\mathbf{x}_{(t)}$ as input a time
- Outputting $\mathbf{y}_{(t)}$
- Using a "latent state" $\mathbf{h}_{(t)}$ to summarize the prefix $\mathbf{x}_{(1...t)}$
- Repeat in a loop over t

$p(\mathbf{h}_{(t)}|\mathbf{x}_{(t)}, \mathbf{h}_{(t-1)})$ latent variable $\mathbf{h}_{(t)}$encodes $[\mathbf{x}_{(1)} \dots \mathbf{x}_{(t)}]$

$p(\mathbf{y}_{(t)}|\mathbf{h}_{(t)})$ prediction contingent on latent variable

Loop with latent state


$\mathbf{x}_{(t)}$ $\mathbf{x}_{(t-1)}$ $\dots$ $\mathbf{x}_{(2)}$ $\mathbf{x}_{(1)}$
$\mathbf{h}_{(t)}$
$\mathbf{f}$
$\mathbf{y}_{(t)}$

"Unrolling" the loop makes it equivalent to a multi-layer network

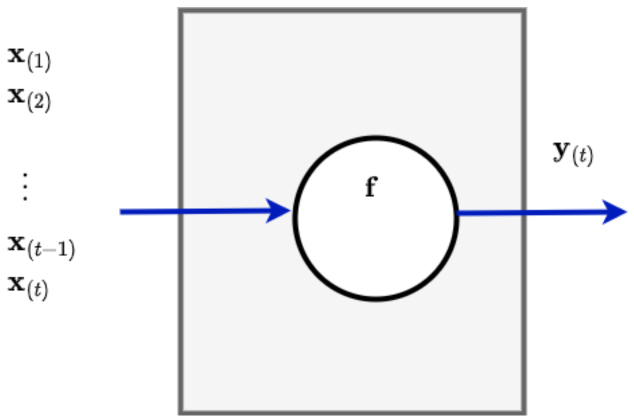
RNN unrolled

Transformer layer

The alternative to the loop was to create a "direct function"

- Taking a **sequence** $\mathbf{x}_{(1...t)}$ as input
- Outputting $\mathbf{y}_{(t)}$

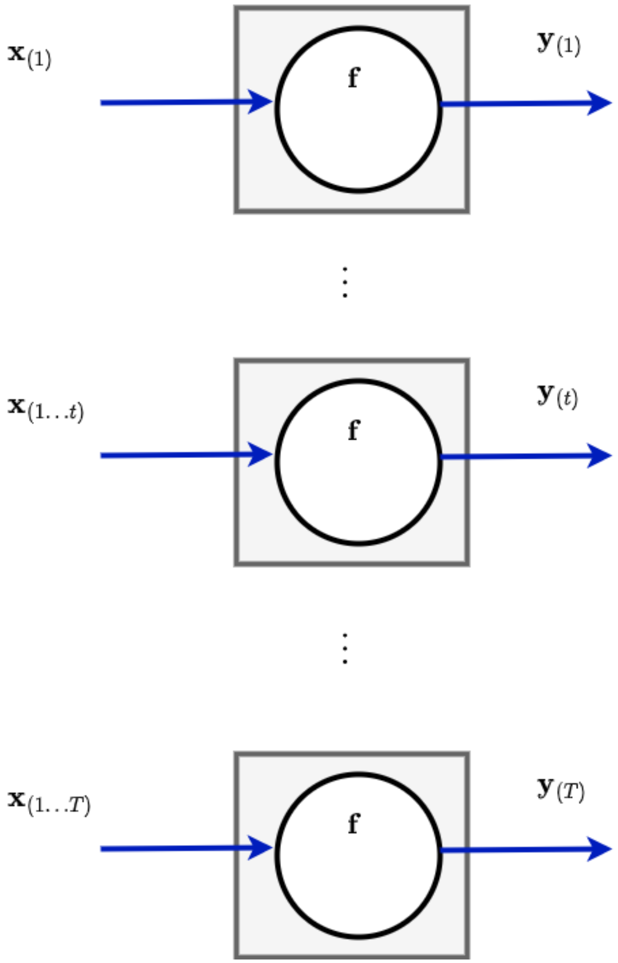
Direct function



In order to output the sequence $\mathbf{y}_{(1)} \ldots \mathbf{y}_{(T)}$ we create T copies of the function (one for each $\mathbf{y}_{(t)}$)

- computes each $\mathbf{y}_{(t)}$ in **parallel**, not sequentially as in the loop

Direct function, in parallel (masked input)


$\mathbf{x}_{(1)}$
f
$\mathbf{y}_{(1)}$
$\vdots$
$\mathbf{x}_{(1...t)}$
f
$\mathbf{y}_{(t)}$
$\vdots$
$\mathbf{x}_{(1...T)}$
f
$\mathbf{y}_{(T)}$

The parallel units constitute a *Transformer layer*

Transformer layer (masked)

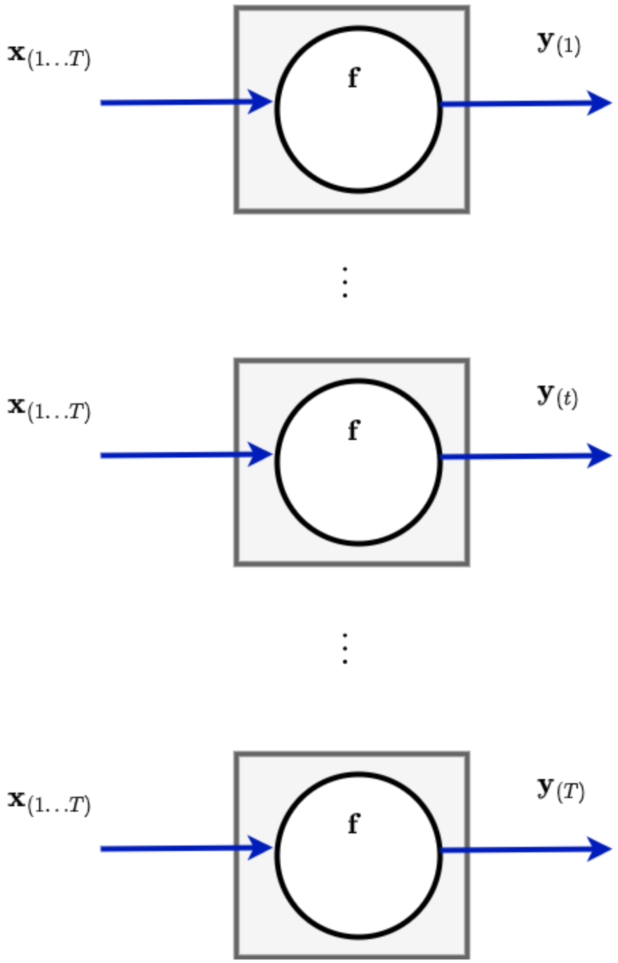
Compared to the unrolled RNN, the Transformer layer

- Has **no** data (e.g., $\mathbf{h}_{(t)}$) passing from the computation between time steps (e.g., from t to $(t+1)$)
- Takes a **sequence** $\mathbf{x}_{(1..t)}$ as input
    - Because $\mathbf{y}_{(t)}$ is computed as a *direct* function of the prefix $\mathbf{x}_{(1..t)}$ rather than recursively
- Outputs generated in parallel, not sequentially
- No gradients flowing backward over time

With this architecture, we can compute more general functions

- where each $\mathbf{y}_{(t)}$ depends on the entire $\mathbf{x}_{(1...T)}$ rather than a prefix $\mathbf{x}_{(1...t)}$

Direct function, in parallel (un-masked input)


$\mathbf{x}_{(1...T)}$
f
$\mathbf{y}_{(1)}$
$\mathbf{x}_{(1...T)}$
f
$\mathbf{y}_{(t)}$
$\mathbf{x}_{(1...T)}$
f
$\mathbf{y}_{(T)}$

Transformer layer

To illustrate the different types of functions:

- In a "predict the next" element function, we restrict the input to a causal prefix
- In a "summarize" the sequence function, we allow access to the full sequence
    - Context Sensitive Encoding of a word within a sentence
    - Same effect as a bi-directional RNN

For generality

- The Transformer input is usually the entire sequence $\mathbf{x}_{(1...T)}$
- When we need to "hide" part of the sequence, we can use an **input mask**
- The mask can be arbitrary
- The particular input mask restricting to a prefix implements **causal** masking
    - Can't "look into the future inputs"

Self Attention

If we look inside the box computing the direct function, we will find several layers

- An Attention layer
    - To influence which elements of the input sequence $\mathbf{x}$ to attend/focus when outputting $\mathbf{y}_{(t)}$
- A Feed Forward Network (FF) layer to compute the function

Transformer Layer (Encoder)

An Attention layer that attends/focus on its inputs implements what is called *Self-attention*

- We will soon see the possibility of attending to other values

If the function for $\mathbf{y}_{(t)}$ is restricted to prefix $\mathbf{x}_{(1 \ldots t)}$ the Attention layer can use causal masking.

This is referred to as *Masked Self-Attention*.

Cross Attention

It is common to use two Transformers in an Encoder-Decoder configuration.

Recall the Encoder-Decoder architecture (using RNN's rather than Transformers in the diagram)

Encoder-Decoder for language translation

The Decoder in the Encoder-Decoder architecture is *generative*

- Outputs $\hat{\mathbf{y}}_{(t)}$ for a single t at a time
- Appending output $\hat{\mathbf{y}}_{(t)}$ to the input available to output the next $\hat{\mathbf{y}}_{(t+1)}$

The Encoder in the Encoder-Decoder architecture creates a latent state $\bar{\mathbf{h}}_{(t)}$ which summarizes the input prefix $\mathbf{x}_{(1...t)}$.

In the above diagram the Decoder only has access to $\bar{\mathbf{h}}_{(\bar{T})}$, the final latent state

- summarizing the entire input sequence

This is very restrictive, forcing $\bar{\mathbf{h}}_{(\bar{T})}$ to encode a lot of information.

But we motivated Attention by suggesting that the Decoder have access to *each* $\bar{\mathbf{h}}_{(t)}$ for $1 \le t \le \bar{T}$.

- and use the Attention mechanism to decide which $\bar{\mathbf{h}}_{(t)}$ to focus on when generating $\hat{\mathbf{y}}_{(t)}$

Decoder: Attention

Thus the Decoder Transformer can also attend to the output of the Encoder.

This is called *Cross Attention* (Encoder-Decoder Attention).

Transformer Layer (Decoder)

Transformer Layer (Encoder/Decoder)

Explanation of diagram

- The Encoder uses Self-attention (wide Green arrow) to attend to input sequence $\mathbf{x}$
- The Decoder uses Masked Self-attention (wide Red arrow) to attend to its input
    - It's input is the prefix of the output sequence $\mathbf{y}$
    - Limited to prefix of length t by **masking**
- The Decoder uses Cross Attention (between Encoder and Decoder) (wide Blue arrow)
    - To enable Decoder to focus on which Encoder latent state $\bar{\mathbf{h}}_{(t)}$ to atttend to
- The dotted (thin Blue arrow) indicates that the output $\hat{\mathbf{y}}_{(t)}$ is appended to the input that is available when generating $\hat{\mathbf{y}}_{(t+1)}$

Stacked Transformer

Just as with many other layer types (e.g., RNN), we may stack Transformer layers.

- Each layer creating alternate representations of the input of increasing complexity

In fact, stacking $N > 1$ Transformer layers is typical.

$N = 6$ was the choice of the original paper.

Stacked Transformer Layers (Encoder/Decoder)

Full Encoder-Decoder Transformer architecture

There are other components of the Encoder and Decoder that we have yet to describe.

We will do so briefly.

(The Transformer was introduced in the paper [Attention is all you Need (https://arxiv.org/pdf/1706.03762.pdf)](https://arxiv.org/pdf/1706.03762.pdf)

Transformer (Encoder/Decoder)

Embedding layers

We will motivate and describe Embeddings in the NLP module.

For now:

- an embedding is an encoding of a categorical value that is shorter than OHE

It is used in the Transformer to

- encode the input sequence of words
- encode the output sequence of words

Positional Encoding

The inputs are ordered (i.e., sequences) and thus describe a relative ordering relationship between elements.

But inputs to most layer types (e.g., Fully Connected) are unordered.

The Positional Encoding is a way of encoding the the relative ordering of elements.

Add and Norm

We have seen each of these layer types before

- Norm: Batch (or other) Normalization layers
- Add: the part of the residual network that joins outputs of multiple previous layers

The diagram shows an Encoder-Decoder pair.

You will notice that each element of the pair is different.

- It is possible to use each element independently as well.
- But first we need to understand the source of the differences and their implications.

Encoder-style Transformer

The Transformer for the Encoder and Decoder of an Encoder-Decoder Transformer are slightly different.

They can also be used individually as well as in pairs.

It's important to understand the differences in order to know when to use each individually.

The Encoder side of the pair **does not** restrict the order in which it's inputs are accessed.

- Self-attention **without** causal masking

So the Encoder is appropriate for tasks that require a context-sensitive representation of each input element.

For example: the meaning of the word "**it**" changes with a small change to a subsequent word in the following sentences:

- "The animal didn't cross the road because **it** was too tired"
- "The animal didn't cross the road because **it** was too wide"

Some tasks with this characteristic are

- Sentiment
- Masked Language Modeling: fill-in the masked word
- Semantic Search
    - compare a summary of the sequence that is the context-sensitive representation of
        - query sentence
        - document sentences
    - Each summary is a kind of **sentence embedding**
    - Summary
        - pooling over each word
        - final token

Decoder-style Transformer

One notable aspect of the Decoder is its recurrent (generative) architecture

- Output $\mathbf{y}_{(t-1)}$ is appended to the Decoder inputs available at step t.
    - The Decoder inputs are $\mathbf{y}_{(1..T)}$, where T is the full length of the Decoder output
    - **But** Causal Masking ensures that only $\mathbf{y}_{(1..t)}$ is *available* at step t.

Thus, the Decoder is appropriate for *generative* tasks

- Text generation
- Predict the next word in a sentence

Advantages of a Transformer compared to an RNN

Among the most important advantages of the Transformer over an RNN

- are its ability to capture long-term dependencies
- because all elements of the sequence are processed in parallel
    - no vanishing gradient or truncated back propagation

This has made the Transformer the architecture of choice for NLP.

The computational advantages (detailed in next section) are many:

- Time: All steps computed in parallel
    - $O(1)$ sequential steps versus $O(T)$
- Fewer operations: faster training
    - $O(T^2 * d)$ versus $O(T * d^2)$, where d is size of latent state and length of a single input element
        - e.g., $\mathbf{x}_{(t)}$ replaced by an embedding of dimension d
    - Transformer has fewer operations when $T < d$
- Similar number of parameters
    - When $T < \sqrt{d}$: Self attention has about the same number of parameters

Note that, because of TBTT, T is the length of a *chunk* rather than the full input length

- Typical $T = 64$, $d \geq 256$

So under the special case (that applies to sequences) that chunk length is short relative to representation size, it is not "crazy" to perform all elements of $\mathbf{x}$ with separate FC's.

The faster training enables

- larger datasets
- deeper models

Detailed computational comparison of architectures

Type	Parameters	Operations	Path length
CNN	$k * d^2$	$T * k * d^2$	T
RNN	d^2	$T * d^2$	T
Self-attention	$T^2 * d$	$T^2 * d$	1

Here's the details of the math

Attention involves a dot product (of vectors of length d)

- Each input matched against all others: $T * T$
- So $T^2 * d$ operations

RNN

- T sequential steps: path length T
- Each step evaluates
$$\mathbf{h}_{(t)} = \phi(\mathbf{W}_{xh}\mathbf{x}_{(t)} + \mathbf{W}_{hh}\mathbf{h}_{(t-1)} + \mathbf{b}_h)$$
- $\mathbf{h}_{(t)}$ has multiple elements, assume $||\mathbf{h}|| = O(d)$
    - Computing updated hidden state element j (i.e., $\mathbf{h}_{(t),j}$) involves dot product of vectors of length d (size of $\mathbf{x}_{(t)}$)
    - d multiplications per element of $\mathbf{h}$, times $O(d)$ elements of $\mathbf{h}$ is $O(d^2)$ per step
    - So $T * d^2$ operations
- $\mathbf{W}_{hh}$ matrix: d^2 parameters
    - $|\mathbf{h}| = d$

CNN

- path length T
  - each kernel multiplication connects only k elements of $\mathbf{x}$
  - since kernels overlap inputs, can't parallelize, hence $O(T/k)$ path length
    - can reduce to $\log(T)$ with tree structure

- Parameters

  - kernel size k
  - number of input channels = number of output channels = d
  - $k * d$ parameters for kernel of one channel
  - $k * d^2$ parameters for kernel for all d output channels

- Operations

  - for a single output channel: k per input channel
    - There are d input channels, so $k * d$ for each dot product of *one* output channel
    - There are d output channels, so $k * d^2$ per time step
  - T time steps so $T * k * d^2$ number of operations

A free lunch ? Almost !

Transformers offer the possibility of great improvements in training speed

- Parallelism
- Fewer operations

Sounds too good to be true. Is there such a thing as a free lunch ?

Almost

- RNN can handle sequences of arbitrary length (T unbounded)
- Transformer has a fixed number of parallel units, which limits the length of sequences

But, in practice: RNN uses *Truncated* Back Propagation Through Time

- So the maximum distance between input sequence elements is bounded by k, the truncation length

Some other advantages

- Can learn long-range dependencies
    - Gradients within a layer don't flow backwards: always a single step
        - Can't vanish or explode
    - The output $\mathbf{y}_{(t)}^{[l]}$ of layer l (for stacked Transformer layers) is a function of **all** inputs
    $$\mathbf{y}_{(t')}^{[l-1]} \text{ for } 1 \le t' \le T$$
        - so can directly access a distant input
        - not diminished by passing through multiple intermediate time steps

Some drawbacks

- The output $\mathbf{y}_{(t)}^{[l]}$ of layer l (for stacked Transformer layers) is a function of **all** inputs, **always**
    - Perhaps less efficient
- Unless you add positional encoding, you lose ordering relationships between inputs

```
In [ ]: print("Done")
```