Recurrent Neural Network (RNN) layer

Review

With a sequence $\mathbf{x^{(i)}}$ as input, and a sequence $\mathbf{y}$ as a potential output, the questions arises:

- How does an RNN produce $\mathbf{y}_{(t)}$, the t^{th} output ?

Some choices

- Predict $\mathbf{y}_{(t)}$ as a direct function of the prefix of $\mathbf{x}$ of length t:

$$p(\mathbf{y}_{(t)}|\mathbf{x}_{(1)} \dots \mathbf{x}_{(t)})$$

- Uses a "latent state" that is updated with each element of the sequence, then predict the output

$$\begin{aligned} &p(\mathbf{h}_{(t)}|\mathbf{x}_{(t)}, \mathbf{h}_{(t-1)}) && \text{latent variable } \mathbf{h}_{(t)} \text{encodes } [\mathbf{x}_{(1)} \dots \mathbf{x}_{(t)}] \\ &p(\mathbf{y}_{(t)}|\mathbf{h}_{(t)}) && \text{prediction contingent on latent variable} \end{aligned}$$

In our first encounter with the RNN, we made the choice to use the "latent state" approach.

Doing so enabled us to picture an RNN as a loop:

During iteration t of the loop

- We consume input $\mathbf{x}_{(t)}$
- Produce output $\mathbf{y}_{(t)}$ (which we will assume is the latent state: $\mathbf{y}_{(t)} = \mathbf{h}_{(t)}$)

We also indicated that we could "unroll" the loop

RNN unrolled

Transformer layer

What would have happened if, rather than using the latent state approach, we choose the alternative:

- Predict $\mathbf{y}_{(t)}$ as a direct function of the prefix of $\mathbf{x}$ of length t:

Then the picture would look similar to the "unrolled" loop:

Compared to the unrolled RNN, the Transformer, the computation at step t

- Has **no** data (e.g., $\mathbf{h}_{(t)}$) passing from the computation between time steps (from $(t-1)$, 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

In some instances, we may even allow the Transformer to "see" the *entire* input (not just a prefix) at each step t

- The Encoder of an Encoder-Decoder architecture
    - Context Sensitive Encoding
        - Encode based on *entire* input
        - Bi-directional RNN

Transformer layer

The Transformer uses *self-attention*

- To influence which elements of $\mathbf{x}_{(1\ldots t)}$ to attend/focus to

Looking inside the circle

Transformer Layer (Encoder)

And there are cases where we *must not* allow the Transformer to "see" the *entire* input

- The Decoder of an Encoder-Decoder architecture
  - Teacher forcing: the input of step $(t+1)$ is $\mathbf{y}_{(t)}$, the output of step t
  - Can't look ahead to something that has not yet been created !

For those cases where look-ahead is not allowed, the Transformer using **masking**

- Hide the suffix $\mathbf{y}_{(t+1 \ldots t)}$ from the attention layer

You will notice two Attention layers

- **Masked** Self Attention (on $\mathbf{y}$)
    - Allows the layer to focus on previous outputs
        - Masked to prevent look-ahead to $\mathbf{y}_{(t')}$ for $t' > t$
- Encoder-Decoder Attention (on $\bar{\mathbf{h}}_{(t)}$)
    - Allows the Decoder to attend to the entire output sequence of the Encoder

So this layer attends to

- previously generated Decoder layer outputs
- the "relevant" part of the Encoder output

The Transformer architecture just stacks N Transformer layers.

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

Stacked Transformer Layers (Encoder/Decoder)

Full Encoder-Decoder Transformer architecture

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.

The Encoder

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

The Decoder

One notable aspect of the Decoder is its recurrent 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 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 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
- 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

RNN

- $\mathbf{W}_{hh}$ matrix: d^2 parameters
    - $|\mathbf{h}| = d$
- $T * d^2$ operations (for entire sequence)
- path length T

To summarize

- for short chunk/sequence length, relative to size of hidden state
    - $|x| < 64$ typically; $d \approx 256$
- Transformer/self attention is comparable in terms of number of parameters

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.

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

$$\mathbf{y}_{(t')}^{[l-1]} \text{ for } 1 \leq t' \leq T$$

        - 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")
```