Back propagation through time (BPTT)

A Recurrent Neural Network (RNN)

- Can be viewed as a loop
- That can be unrolled
- Resulting in a multi-layer network
- One layer per time step

Here are the final layers of an unrolled RNN with input sequence

$$\mathbf{x}_{(1)}, \ldots, \mathbf{x}_{(T)}$$

RNN many to many API

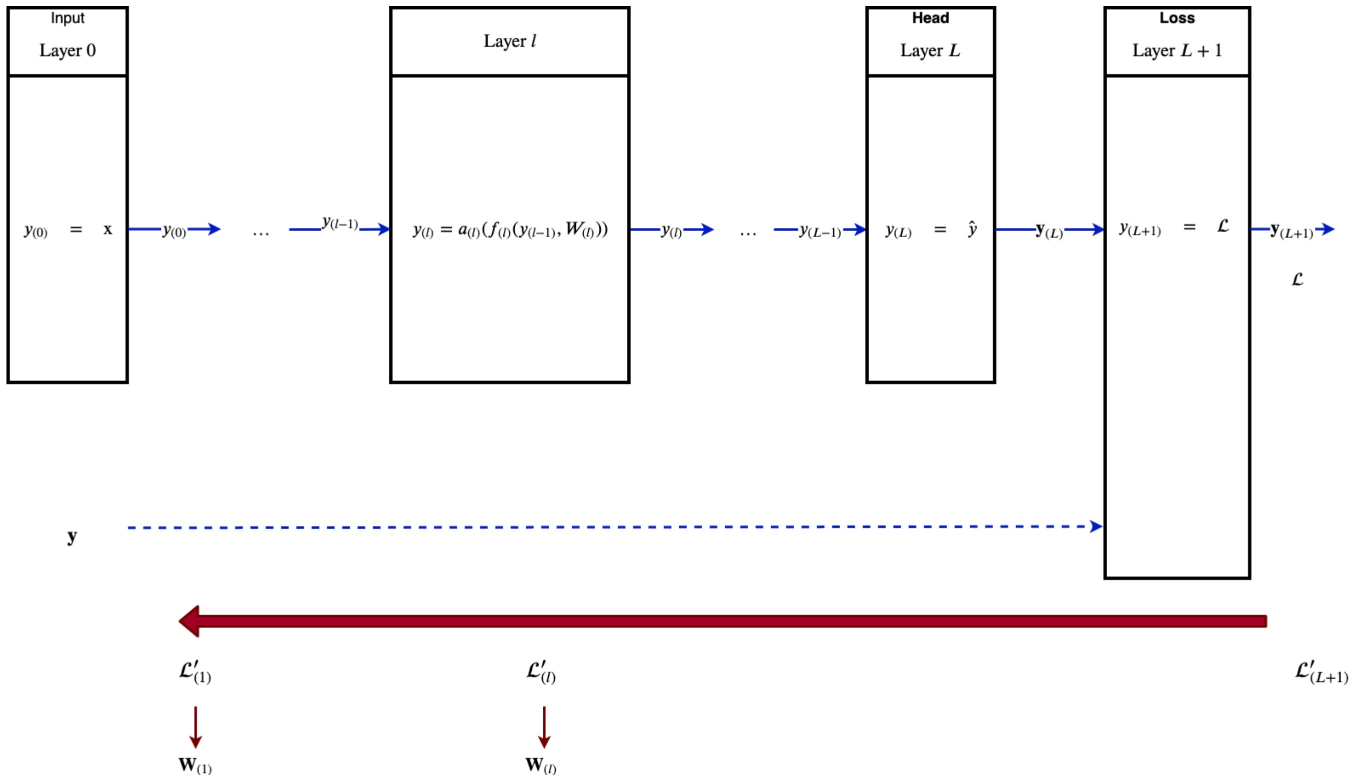
Given enough space: we would continue unrolling on the left to the Input layer

- Resulting in a network with T unrolled layers
- Plus a Loss layer

To compute the derivatives of the Loss with respect to weights

- We could, in theory, use Back Propagation
- Which is the weight update step of Gradient Descent

Backward pass: Loss to Weights



When dealing with unrolled RNN's

- We will index the "unrolled layers" with time steps, denoted by the label t
- Rather than l, which we use to index layers

This process is called *Back Propagation Through Time* (BPTT).

The only special thing to note about BPTT is that the Loss function is more complex

- There is a Loss
- Per example (as in non-recurrent layers)
- **and Per time-step** (unique to recurrent layers)

RNN Loss: Forward pass

RNN Loss: Backward pass

Truncated back propagation through time (TBPTT)

An unrolled RNN layer turns into a T layer network where T is the number of elements in the input sequence.

For long sequences (large T) this may not be practical.

First, there is the computation *time*

- t steps to compute $\mathcal{L}_{(t)}^{(\mathbf{i})}$, the loss due to the t^{th} output $\mathbf{y}_{(t)}^{(\mathbf{i})}$ of example i
- For each $1 \leq t \leq T$

Less obvious is the *space* requirement

- As we saw in the module "How a Neural Network Toolkit works"
- We may store information in each layer of the Forward pass (so storage for T layers)
- To facilitate computation of analytical derivatives on the Backward pass
    - For example: the Multiply layer stored the multiplicands in the forward pass
    - Because they are needed for the derivatives

Moreover, as we shall shortly see

- Derivatives may vanish or explode as we proceed further backwards from the Loss layer to the Input layer

So, in theory, the weights $\mathbf{W}_{(t)}$ for small t (close to the input) may not get updated.

- This is certainly a problem in a non-recurrent network
- But is **fatal** in a recurrent layer
- Since there is a **single** weight matrix $\mathbf{W}$ that is shared across *all time steps*

$$\mathbf{W}_{(t)} = \mathbf{W} \; \text{ for all } 1 \le t \le T$$

The solution to these difficulties

- Is to *truncate* the unrolled RNN
- To a fixed number of time steps
- From the loss layer backwards
- The truncated graph is a suffix of the fully unrolled graph

This process is known as *Truncated Back Propagation Through Time* (TBPTT).

Note that *truncation only occurs in the backward pass.*

There is *no truncation* of the forward pass of the RNN !

Because the unrolled graph is less than T steps

- Gradient computation takes fewer steps
- So weight updates can occur more often

The obvious downside to truncation is that

- Gradients are only approximate

But there is a subtle and more impactful difference

- The RNN layer *cannot capture long-term dependencies*

Suppose we unrolled the layer for only τ time steps (the "window" size)

- The loss for the t^{th} time step ($\mathcal{L}^{(\mathbf{i})}_{(t)}$)
- Flows backwards only to steps
$$(t - \tau + 1), \ldots, t$$

So the "error signal" from time t does not affect time steps $t' < (t - \tau + 1)$

Consider a long sentence or document (sequence of words)

- If the gender of the subject is defined by the early words in the sentence
- An incorrect "prediction" late in the sentence
- May not be able to be corrected

"Z was the first woman who ... **he** said ..."

In other words

- Truncation may affect the ability of an RNN to encode *long-term* dependencies
- Vanishing gradients may cause a similar impact

TBPTT variants

There are several common ways to decide on how many unrolled time steps to keep.

Let t'' denote the index of the *smallest* time step in the unrolled layer for step t.

- $t'' = (t - \tau + 1)$

Plain, untruncated BPTT defines

- $t'' = 0$
- Unroll all the way to the Input Layer

k-truncated BPTT defines window size $\tau = k$

- $t'' = \max(0, t - k)$

Subsequence truncated BPTT defines

- $t'' = k * \lfloor t/k \rfloor$

That is, it breaks the sequence into "chunks" of size k

$$
\begin{array}{l}
\mathbf{x}^{(\mathbf{i})}_{(1)}, \ldots, \mathbf{x}^{(\mathbf{i})}_{(k)} \\
\mathbf{x}^{(\mathbf{i})}_{(k+1)}, \ldots, \mathbf{x}^{(\mathbf{i})}_{(2*k)} \\
\vdots \\
\mathbf{x}^{(\mathbf{i})}_{((i'*k)+1)}, \ldots, \mathbf{x}^{(\mathbf{i})}_{((i'+1)*k)} \\
\vdots
\end{array}
$$

- Gradients flow *within* chunks
- But *not between* chunks

Subsequence TBPTT is very common as it fits well into the design of current toolkits

See the Deep Dive on [How to deal with long sequences (RNN_Long_Sequences.ipynb)] for how to arrange your training examples.

Calculating gradients in an RNN

There is an important subtlety we have ignored regarding Back Propagation in an unrolled RNN

- There is a **single** weight matrix $\mathbf{W}$ that is shared across *all time steps*
$$\mathbf{W}_{(t)} = \mathbf{W} \;\; \text{for all } 1 \le t \le T$$

This

- Makes the derivative computation slightly more complex
- Creates an *additional* exposure to the problem of vanishing/exploding gradients

A simple picture will illustrate.

Consider the loss at time step t of example i

- $\mathcal{L}_{(t)}^{(\mathbf{i})} = L(\hat{\mathbf{y}}_{(t)}^{(\mathbf{i})}, \mathbf{y}_{(t)}^{(\mathbf{i})}; \mathbf{W})$
- The loss is a function of
    - $\hat{\mathbf{y}}_{(t)}^{(\mathbf{i})}$: The t^{th} element of the output sequence $\hat{\mathbf{y}}^{(\mathbf{i})} = \mathbf{y}_{(T)}$ for example i
    - The $\mathbf{y}_{(t)}^{(\mathbf{i})}$: The t^{th} element of the **target** sequence $\mathbf{y}^{(\mathbf{i})}$ for example i

The derivative

$$\frac{\partial \mathcal{L}_{(t)}}{\partial \mathbf{W}}$$

- Which is used in Gradient Descent to update the estimate for weights $\mathbf{W}$
- Is the sensitivity of $\mathcal{L}_{(t)}$ to a change in $\mathbf{W}$

RNN Time step

RNN multiple dependence on W

The red lines show **two** different ways that $\mathbf{W}$ affects $\mathbf{h}_{(t)}$

- And thus $\hat{\mathbf{y}}_{(t)} = \mathbf{W}_{hy}\mathbf{h}_{(t)} + \mathbf{b}_y$
- By its indirect effect on $\mathbf{h}_{(t)}$ **through** $\mathbf{h}_{(t-1)}$ (lower line)
- By its direct effect on $\mathbf{h}_{(t)}$ (upper line)
- Both using the part of $\mathbf{W}$ denoted by $\mathbf{W}_{hh}$

So

$$
\begin{aligned}
\frac{\partial \mathbf{h}_{(t)}^{(\mathbf{i})}}{\partial \mathbf{W}_{hh}} &= \frac{d\mathbf{h}_{(t)}^{(\mathbf{i})}}{d\mathbf{W}_{hh}} + \frac{\partial \mathbf{h}_{(t)}^{(\mathbf{i})}}{\partial \mathbf{h}_{(t-1)}^{(\mathbf{i})}} \frac{\partial \mathbf{h}_{(t-1)}^{(\mathbf{i})}}{\partial \mathbf{W}_{hh}} \\
&= \frac{d(\mathbf{W}_{hh}\mathbf{h}_{(t-1)}^{(\mathbf{i})})}{d\mathbf{W}_{hh}} + \frac{\partial \mathbf{h}_{(t)}^{(\mathbf{i})}}{\partial \mathbf{h}_{(t-1)}^{(\mathbf{i})}} \frac{\partial \mathbf{h}_{(t-1)}^{(\mathbf{i})}}{\partial \mathbf{W}_{hh}}
\end{aligned}
$$

(Each addend reflect a different path through which $\mathbf{W}_{hh}$ affects $\mathbf{h}_{(t)}$)

So

$$\frac{\partial \mathcal{L}_{(t)}^{\mathbf{(i)}}}{\partial \mathbf{W}} = \mathcal{L}'_{(t)} \frac{\partial \mathbf{y}_{(t)}^{\mathbf{(i)}}}{\partial W}$$

and

$$\frac{\partial \mathbf{y}_{(t)}^{\mathbf{(i)}}}{\partial W}$$

depends on all time steps from 1 to t.

Thus, the derivative update for $\mathbf{W}$ cannot be computed without the gradient (for each time step t) flowing all the way back to time step 0.

Conclusion

Updating the weights of a Recurrent layer appears, at first glance, to be straight forward

- Unroll the loop
- Use ordinary Back Propagation

We have discovered some complexity

- Full unrolling is expensive
- Gradient computation is complicated by shared weights

Fortunately, we have solutions to these complexities.

In [3]:

```python
print("Done")
```

```
Done
```