

EECS 182  
Fall 2022Deep Neural Networks  
Anant Sahai

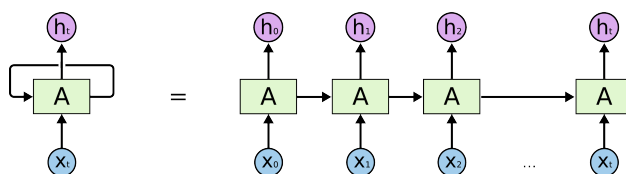
Midterm Review: Seq2Seq

## 1. RNN Recap (in Homework 4)

A vanilla RNN layer implements the function

$$h_t = \sigma(W^h h_{t-1} + W^x x_t + b)$$

where  $W^h$ ,  $W^x$ , and  $b$  are learned parameter matrices,  $x$  is the input sequence, and  $\sigma$  is a nonlinearity such as tanh. The RNN layer “unrolls” across a sequence, passing a hidden state between timesteps and returning an array of hidden states at all timesteps.

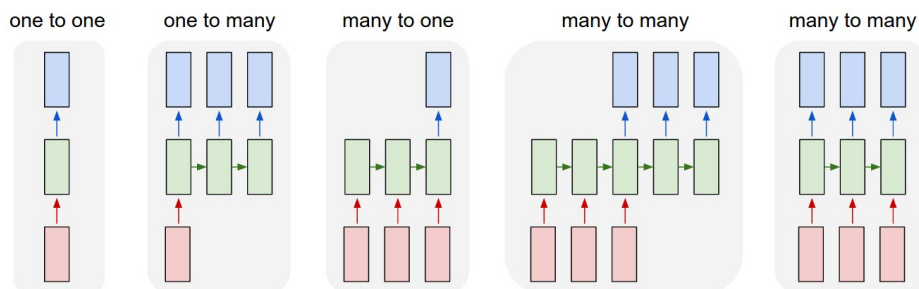


**Figure 1:** Source: <https://colah.github.io/posts/2015-08-Understanding-LSTMs/>

For output, you will use this RNN layer in a regression model by adding a final linear layer on top of the RNN outputs.

$$\hat{y}_t = W^f h_t + b^f$$

We’ll compute one prediction for each timestep. RNNs can be used for many kinds of prediction problems, as shown below.

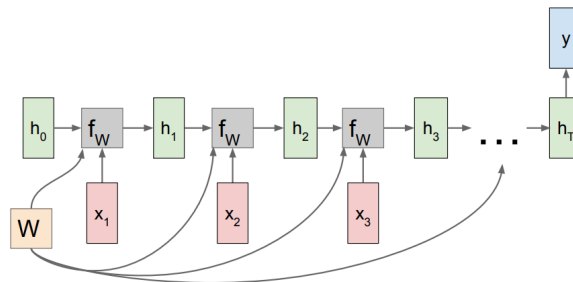


We here consider a simple averaging task. The input  $X$  consists of a sequence of numbers, and the label  $y$  is a running average of all numbers seen so far.

We will consider two tasks with this dataset:

- Task 1: predict the running average at all timesteps
- Task 2: predict the average at the last timestep only

## RNN: Computational Graph: Many to One



**Figure 2:** Image source: [https://calvinfeng.gitbook.io/machine-learning-notebook/supervised-learning/recurrent-neural-network/recurrent\\_neural\\_networks](https://calvinfeng.gitbook.io/machine-learning-notebook/supervised-learning/recurrent-neural-network/recurrent_neural_networks)

- (a) Consider an RNN which outputs a single prediction at timestep  $T$ . As shown in Figure 2, each weight matrix  $W$  influences the loss by multiple paths. As a result, the gradient is also summed over multiple paths:

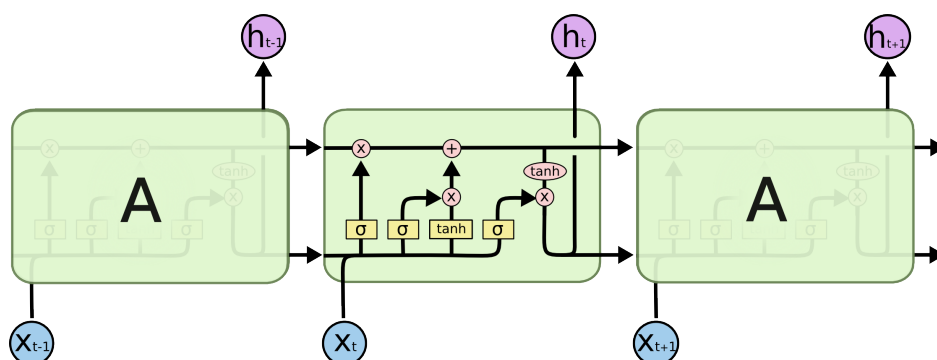
$$\frac{\partial \mathcal{L}}{\partial W} = \frac{\partial \mathcal{L}}{\partial h_T} \frac{\partial h_T}{\partial W} + \frac{\partial \mathcal{L}}{\partial h_{T-1}} \frac{\partial h_{T-1}}{\partial W} + \dots + \frac{\partial \mathcal{L}}{\partial h_1} \frac{\partial h_1}{\partial W} \quad (1)$$

When you backpropagate a loss through many timesteps, the later terms in this sum often end up with either very small or very large magnitude - called vanishing or exploding gradients respectively. Either problem can make learning with long sequences difficult.

**In the original Notebook Section 1.D**, it plots the magnitude at each timestep of  $\frac{\partial \mathcal{L}}{\partial h_t}$ . Play around with this visualization tool and try to generate exploding and vanishing gradients.

**If the network has no nonlinearities, under what conditions would you expect the exploding or vanishing gradients with for long sequences? Why?** (Hint: it might be helpful to write out the formula for  $\frac{\partial \mathcal{L}}{\partial h_t}$  and analyze how this changes with different  $t$ ). **Do you see this pattern empirically using the visualization tool in Section 1.D in the notebook with last\_step\_only=True?**

- (b) Compare the magnitude of hidden states and gradients when using ReLU and tanh nonlinearities in Section 1.D in the notebook. **Which activation results in more vanishing and exploding gradients? Why?** (This does not have to be a rigorous mathematical explanation.)
- (c) **What happens if you set last\_target\_only = False in Section 1.D in the notebook? Explain why this change affects vanishing gradients. Does it help the network's ability to learn dependencies across long sequences?** (The explanation can be intuitive, not mathematically rigorous.)
- (d) **In Section 1.8 of the notebook** consists of a LSTM layer. LSTMs pass a cell state between timesteps as well as a hidden state. **Explore gradient magnitudes using the visualization tool you implemented earlier and report on the results.**



**Figure 3:** Image source: <https://colah.github.io/posts/2015-08-Understanding-LSTMs/>

The LSTM forward pass is shown below:

$$\begin{aligned}
 f_t &= \sigma(x_t U^f + h_{t-1} W^f + b^f) \\
 i_t &= \sigma(x_t U^i + h_{t-1} W^i + b^i) \\
 o_t &= \sigma(x_t U^o + h_{t-1} W^o + b^o) \\
 \tilde{C}_t &= \tanh(x_t U^g + h_{t-1} W^g + b^g) \\
 C_t &= f_t \circ C_{t-1} + i_t \circ \tilde{C}_t \\
 h_t &= \tanh(C_t) \circ o_t
 \end{aligned}$$

where  $\circ$  represents the Hadamard Product (elementwise multiplication) and  $\sigma$  is the sigmoid function.

- (e) When using an LSTM, you should still see vanishing gradients, but the gradients should vanish less quickly. **Interpret why this might happen by considering gradients of the loss with respect to the cell state.** (Hint: consider computing  $\frac{\partial \mathcal{L}}{\partial C_{T-1}}$  using the terms  $\partial \mathcal{L}, \partial C_T, \partial C_{T-1}, \partial h_T, \partial h_{T-1}$ ).
- (f) Consider a ResNet with simple resblocks defined by  $h_{t+1} = \sigma(W_t h_t + b_t) + h_t$ . **Draw a connection between the role of a ResNet's skip connections and the LSTM's cell state in facilitating gradient propagation through the network.**

## 2. Beam Search Recap (in Homework 6)

When making predictions with an autoregressive sequence model, it can be intractable to decode the true most likely sequence of the model, as doing so would require exhaustively searching the tree of all  $O(M^T)$  possible sequences, where  $M$  is the size of our vocabulary, and  $T$  is the max length of a sequence. We could decode our sequence by greedily decoding the most likely token each timestep, and this can work to some extent, but there are no guarantees that this sequence is the actual most likely sequence of our model.

Instead, we can use beam search to limit our search to only candidate sequences that are the most likely so far. In beam search, we keep track of the  $k$  most likely predictions of our model so far. At each timestep, we expand our predictions to all of the possible expansions of these sequences after one token, and then we keep only the top  $k$  of the most likely sequences out of these. In the end, we return the most likely sequence out of our final candidate sequences. This is also not guaranteed to be the true most likely sequence, but it is usually better than the result of just greedy decoding.

The beam search procedure can be written as the following pseudocode ( $y_{t,i,k}$  is the  $k$ th token at timestep  $t$  generated from hypothesis  $i$ .  $y_{1:t-1}$ ,  $i$  means hypothesis  $i$  is a string of tokens for timesteps 1 through  $t-1$ .)

**Algorithm 1** Beam Search

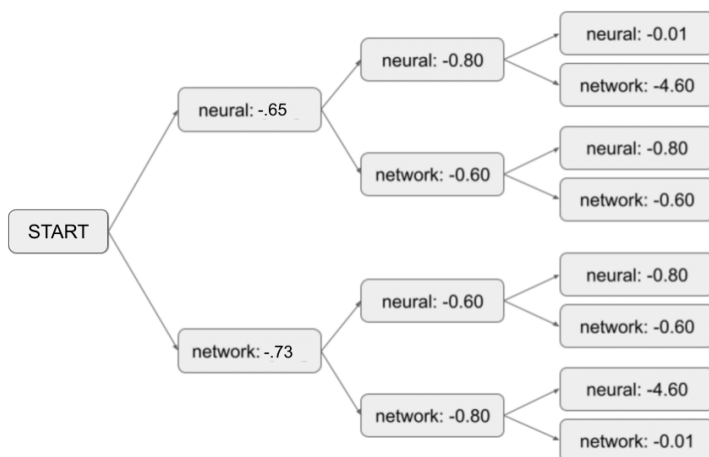
---

```

for each time step  $t$  do
  for each hypothesis  $y_{1:t-1,i}$  that we are tracking do
    find the top  $k$  tokens  $y_{t,i,1}, \dots, y_{t,i,k}$ 
  end for
  sort the resulting  $k^2$  length  $t$  sequences by their total log-probability
  store the top  $k$ 
  advance each hypothesis to time  $t + 1$ 
end for

```

---



**Figure 4:** The numbers shown are the decoder’s log probability prediction of the current token given previous tokens.

We are running beam search to decode a sequence of length 3 with  $k = 2$ . Consider predictions of a decoder in Figure 4, where each node in the tree represents the next token **log probability** prediction of one step of the decoder conditioned on previous tokens. The vocab consists of two words: “neural” and “network”.

- At timestep 1, which sequences is beam search storing?
- At timestep 2, which sequences is beam search storing?
- At timestep 3, which sequences is beam search storing?
- Does beam search return the overall most-likely sequence? Explain why or why not.
- What information needs to be stored for each of the  $k$  hypotheses in beam search with an RNN?