EECS 182 Deep Neural Networks
Fall 2022 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 σ 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.

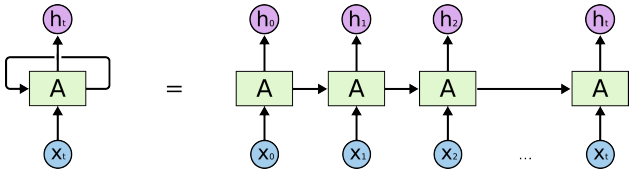
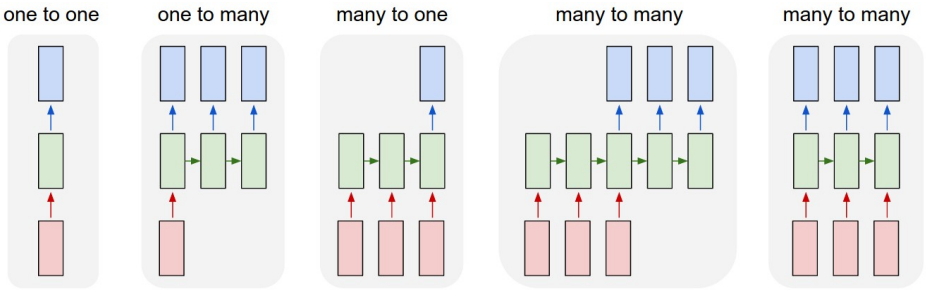


Figure 1: Source: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

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

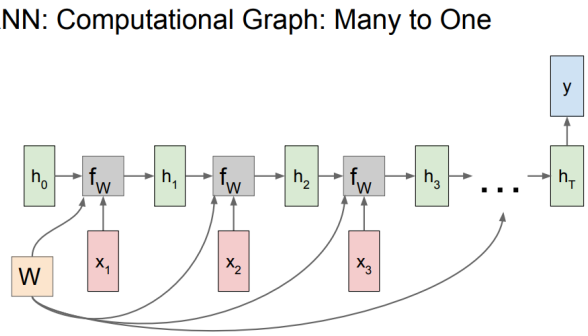


Figure 2: Image source: https://calvinfeng.gitbook.io/machine-learning-notebook/supervised-learning/recurrent-neural-network/recurrent_neural_networks

(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:

$$\frac{\partial \mathcal{L}}{\partial W} = \frac{\partial \mathcal{L}}{\partial h_T}\frac{\partial h_T}{\partial W} + \frac{\partial \mathcal{L}}{\partial h_{T-1}}\frac{\partial h_{T-1}}{\partial W} + \ldots + \frac{\partial \mathcal{L}}{\partial h_1}\frac{\partial h_1}{\partial W} \tag{1}$$

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?

Solution: If we use the MSE loss on a single example (x, y), the gradient $\frac{\partial \mathcal{L}}{\partial h_t} = 2(\hat{y}-y)W^f(W^h)^{T-t}$. (To clarify, the exponents f and h are matrix indicators, but $t-i$ is an exponent.) If the magnitude of the largest eigenvalue of W^h is much greater than 1, the gradient will explode, and if it's much less than 1, the gradient will start to vanish. Gradients are only stable when the largest eigenvalue magnitude is close to 1.

We see the expected pattern empirically.

(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.)

Solution: Hidden states: tanh restricts hidden state values to (-1, 1), so hidden state magnitudes remain small. With ReLU, hidden state values can easily explode.

Gradients: When tanh inputs are large, gradients are close to zero. This results in fewer exploding gradients but more vanishing gradients. Exploding gradients are still possible, however, when the largest eigenvalue of W_h has magnitude > 1, but the hidden states remain close to zero. ReLU activations, in contrast, result in frequent exploding hidden state sizes and gradients, similar to in the no-activation RNN.

(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.)

Solution: For every timestep k, the model's prediction produces loss $\mathcal{L}_k$. The gradient $\partial L_k/\partial h_k$ is high-magnitude, resulting in high-magnitude logged gradients in the visualization tool, but for any timestep $t << k$, $\partial L_k/\partial h_t$ will still be small. This means the network will still struggle to pass gradients over long sequences, making it hard to learn long-range dependencies.

(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. Solution:** Hidden state outputs remain between +/- 1. Gradients don't explode, but they still vanish. Typically, they don't vanish as fast as vanilla RNNs.

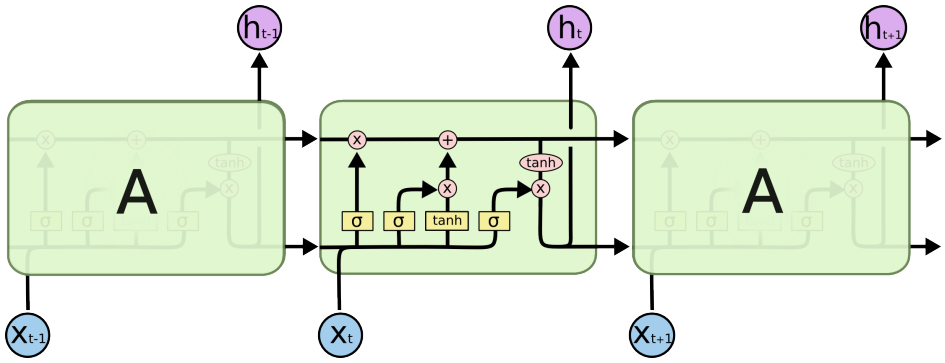


Figure 3: Image source: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

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 σ 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 $\partial \mathcal{L}, \partial C_T, \partial C_{T-1}, \partial h_T, \partial h_{T-1}$).

Solution: In an LSTM, information can be stored in the cell state without needing to persist through a chain of matrix multiplications.

$$
\frac{\partial \mathcal{L}}{\partial C_{T-1}} = \frac{\partial \mathcal{L}}{\partial h_T}\left(\frac{\partial h_T}{\partial C_T}\frac{\partial C_T}{\partial C_{T-1}} + \frac{\partial h_T}{\partial h_{T-1}}\frac{\partial h_{T-1}}{\partial C_{T-1}}\right)
$$

Unlike $\frac{\partial h_T}{\partial h_{T-1}}$, which results in decaying gradients due to matrix multiplication, $\frac{\partial C_T}{\partial C_{T-1}}$ is a diagonal matrix with f_t on the diagonal. If the network sets f_t close to 1, then $\frac{\partial C_T}{\partial C_{T-1}}$ will be close to the identity, allowing information stored in the cell state to "pass through" without decay. In practice, however, many elements of f_t are not close to 1, which results in some gradient decay.

In addition, LSTMs also contain a hidden state in which to store information, and this "pathway" through the network will decay like in a vanilla RNN, potentially resulting in some overall gradient decay.

(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.**

Solution: ResNet skip connections and LSTM cell states both allow information to pass through the network without requiring a matrix multiplication at each layer. As a result, they both mitigate vanishing gradients.

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 kth 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 $y_{t,i,1}$,...,$y_{t,i,k}$
  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

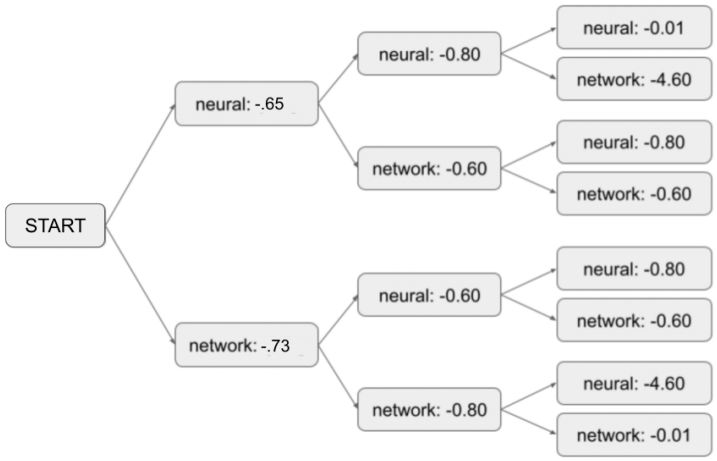


Figure 4: The numbers shown are the decoder's log probability prediction of the current token given previous tokens.

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".

(a) **At timestep 1, which sequences is beam search storing?**

Solution: There are only two options, so our beam search keeps them both: "neural" (log prob = -.65) and "network" (log prob = -.73).

(b) **At timestep 2, which sequences is beam search storing?**

Solution: We consider all possible two word sequences, but we then keep only the top two, "neural network" (with log prob = -.65 - .6 = -1.25) and "network neural" (with log prob = -.73 - .6 = -1.33).

(c) **At timestep 3, which sequences is beam search storing?**

Solution: We consider three word sequences that start with "neural network" and "network neural", and the top two are "neural network network" (with log prob = -.65 - .6 - .6 = -1.85) and "network neural network" (with log prob = -.73 - .6 - .6= -1.93).

(d) **Does beam search return the overall most-likely sequence? Explain why or why not.**

Solution: No, the overall most-likely sequence is "neural neural neural" with log prob = -.65 - .8 - .01= -1.46). These sequences don't get returned since they get eliminated from consideration in step 2, since "neural neural" is not in the $k = 2$ most likely length-2 sequences.

(e) **What information needs to be stored for each of the k hypotheses in beam search with an RNN?**

Solution: We need to store the sequence of tokens $y_{1:t}$ and the final hidden state (and if applicable, cell state) so that we can continue unrolling the RNN.

(If you do not store the hidden states, then rolling out a hypothesis one more timestep involves re-running the RNN from the start, which increases the runtime. For the new runtime, the first timestep unrolls 1 timestep, the second unrolls 2, the third 3, etc. so the average unroll length is $T/2$. Therefore, the average RNN forward pass at each iteration of the algorithm switches from $O(1)$ to $O(T)$, giving $O(T^2KM \log M)$ overall.

