CS 182/282A Designing, Visualizing and Understanding Deep Neural Networks
Spring 2021 Sergey Levine **Discussion 6**

This discussion covers recurrent neural networks and LSTMs. Your TA will also walk through some midterm preparation for the midterm.

1 Recurrent Neural Network

The world is full of sequential information, from video to language modelling to time series data. In particular, we would like to model these sequences using neural networks, and solve some major types of tasks that we would like to solve with sequence models.

1.1 Types of Problems

- **One-to-one** problems take a single input x and produce a single output y. Problems like classification (takes an image as input, and produces a class label as output) and semantic segmentation (image as input, segmentation mask as output) fall under this category.
- **One-to-many** problems take a single input, and produce a sequence of output. Problems like image captioning (takes a single image as input, and produces a caption (a sequence of words) as output) fall under this category.
- **Many-to-many** problems take sequences of inputs and produce sequences of outputs. Problems like language translation (sequence of words in one language to sequence of words in another) fall under this category

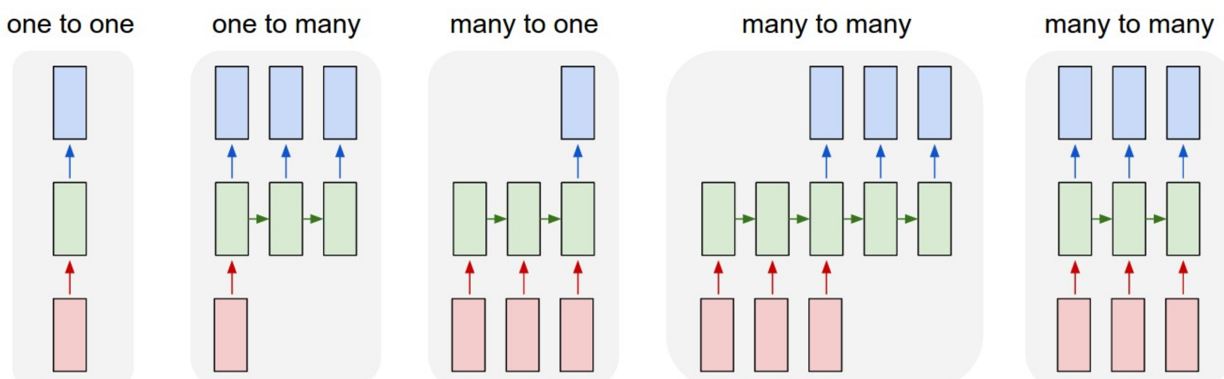

Figure 1: Types of problems we would like to solve using sequential models

1.2 Why the Recurrence?

As you read through this discussion worksheet, you don't process each word entirely on its own, but instead use your understanding from the previous words as well. Traditional neural networks do not have the capability to use its reasoning about previous events to infer later ones. For example, if we would like to classify what is happening at every frame in a movie, this can be framed as an image classification task where the network is provided the current image. However, it is unclear how a traditional neural network should incorporate knowledge from the previous frames in the film to inform later ones.

Recurrent neural networks (RNNs) address this issue, by using the idea of "recurrent connections." RNNs are networks with loops in them that allow information from previous inputs to persist as the network processes the future inputs. These recurrent connections allow information to propagate from "the past" (earlier in the sequence) to the future (later in the sequence).

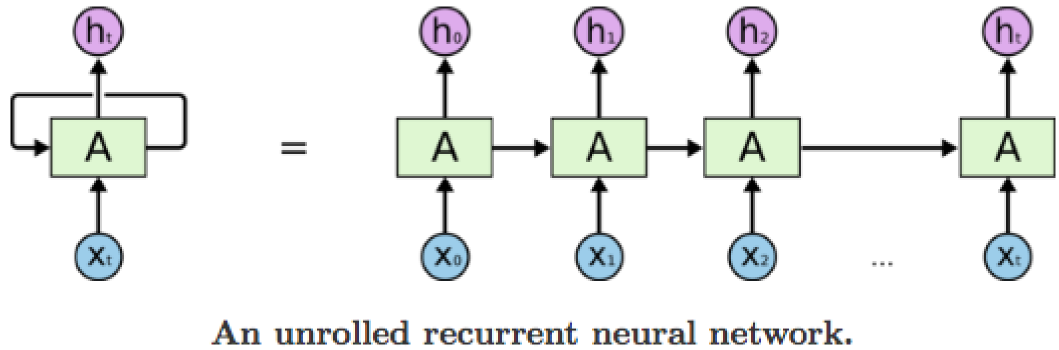

Figure 2: An example of a generic recurrent neural network. This shows how to "unroll" a network through time - instead of thinking about sequence modeling as a single network with shared weights

In Figure 2, we illustrate the RNN computation as it is unrolled through time. Each $i \in \{0, \ldots, t\}$ represents a new timestep in the network. By feeding in a state computed from earlier timesteps as an input together with the current input, information can persist throughout the time as the network "remembers" the past inputs it processed.

1.3 Vanilla RNN

In the following section, we will use the following notation. Denote the input sequence as $x_t \in \mathbb{R}^k$ for $t \in \{1, \ldots, T\}$, and output of the network be $y_t \in \mathbb{R}^m$ for $t \in \{1, \ldots, T\}$. In the following example, we construct a"vanilla" many-to-many RNN, consisting of a node that updates the hidden state h_t and produces an output y_t at each timestep with the following equations:

$$h_t = \mathsf{tanh}(W_{h,h}h_{t-1} + W_{x,h}x_t + B_h)$$
$$y_t = W_{h,y}h_t + B_y$$

where h_t is the time step of a hidden state (one can think of h_{t-1} as the previous hidden state), $W_{\cdot,\cdot}$ be the set of weights (for example, $W_{x,h}$ represents weight matrix that accepts an input vector and produce a new hidden state), y_t be the output at timestep t and B_h and B_y be the bias terms. We can also represent it as the diagram below,

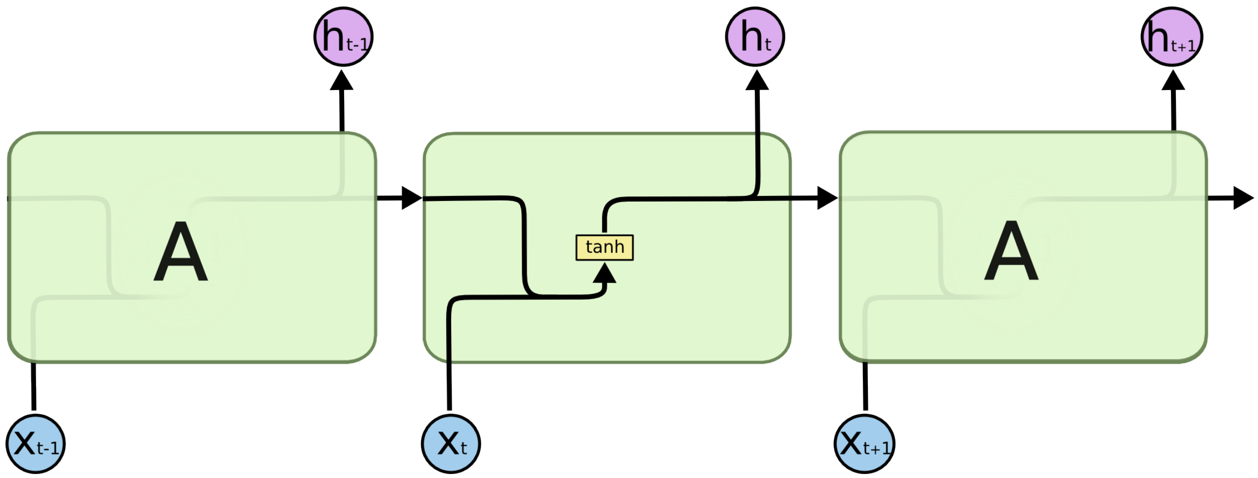

Figure 3: A simple RNN cell. As we can see by the arrows, we only pass a single hidden state from time $t-1$ to time t

In this vanilla RNN, we update to a hidden state "h_t" based on the previous hidden state h_{t-1} and input at the current time x_t, and produce an output which that is a simple affine function of the hidden state. To compute the forward (and backward) passes of the network, we have to "unroll" the network, as shown in Figure 2. This "unrolling" process creates something that resembles a very deep feed forward network (with depth corresponding to the length of the input sequence), with shared affine parameters at each layer. Our gradient is computed by summing the losses from each time-step of the output.

Problem: Gradients in Vanilla RNN

Why are vanishing or exploding gradients an issue for RNNs?

Problem: Coding RNNs Up!

Complete the class definition, started for you below,

```python
import numpy as np

class VanillaRNN:
    def __init__(self):
        self.hidden_state = np.zeros((3, 3))
        self.W_hh = np.random.randn(3, 3)
        self.W_xh = np.random.randn(3, 3)
        self.W_hy = np.random.randn(3, 3)
        self.Bh = np.random.randn(3)
        self.By = np.random.randn(3)
        self.hidden_state = np.zeros(3)

    def forward(self, x):
        # Processes the input at a single timestep and updates the hidden state
        self.hidden_state = np.tanh(...)
        self.output = np.dot(...) + ...
        return self.output
```

2 Long Short Term Memory (LSTM)

To address the problem of vanishing and exploding gradients, we can use a different kind of recurrent cell - the LSTM cell (standing for "long short term memory"). The layout of the cell is shown in Figure 4. The LSTM has two states which are passed between timesteps: a "cell memory" C and the hidden state h. The LSTM update is given as follows:

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

where $\circ$ represents the Hadamard Product (elementwise multiplication).

The update function is rather complex, but it makes a lot of sense when looking at it in the context of the cell state C as a "memory". First, we compute the value f_t, which we call the "forget" gate, as it controls how we retain information through time. Because of the sigmoid activation function, f_t is bounded between 0 and 1, and the first thing we do is multiply the previous memory by f_t. Intuitively, if f_1 is close to 1, we "remember" the previous state, and if f_t is close to 0, we forget it. Next, we compute i_t, which we consider as the "input/update" gate, which controls how much we update the cell memory at the current timestep. The update gate gets added to the memory cell, so it takes information from the current input x_t and adds it to the memory. Finally, the output gate o_t controls the output of the network, the value that gets passed on to the next cell.

We can compare the LSTM to a vanilla RNN. Since a vanilla RNN had to use the hidden state h_t both to produce outputs as well as store memories, h_t gets updated with an affine map and a tanh activation at every timestep, which can easily lead to vanishing or exploding gradients. On the other hand, the LSTM can use the cell state C_t as its "long-term memory," and backpropagation through the long term memory is much easier since the cell states change fairly slowly in a simple way (simply being a moving average of the compute $\tilde{C}_t$'s.). On the other hand, the hidden state in the LSTM can serve as a "short-term memory" and change quickly through time.

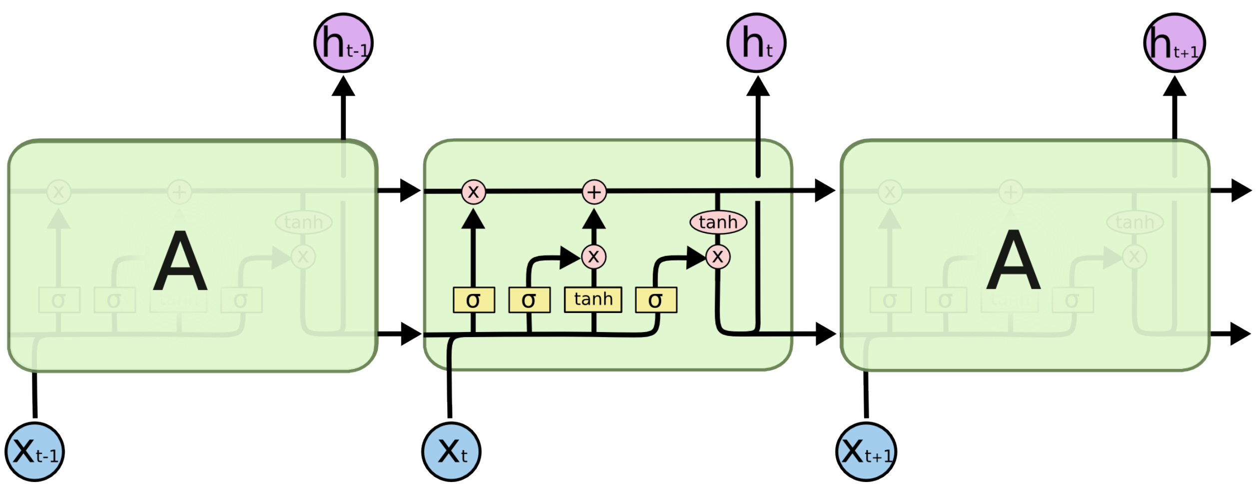

Figure 4: An overview of the LSTM cell

Problem: Backpropagation Through LSTM

Denote the final cost function as J. Compute the gradient $\frac{\partial J}{\partial W^g}$ using a combination of the following gradients,

$$\frac{\partial h_t}{\partial h_{t-1}}, \frac{\partial h_{t-1}}{\partial W^g}, \frac{\partial J}{\partial h_t}, \frac{\partial C_t}{\partial W^g}, \frac{\partial C_{t-1}}{\partial W^g}, \frac{\partial C_t}{\partial C_{t-1}}, \frac{\partial h_t}{\partial C_t}, \frac{\partial h_t}{\partial o_t}$$

Problem: Vanishing Gradient in LSTMs

Using the previously derived gradient, which part of $\frac{\partial J}{\partial W^g}$ allows LSTMs to mitigate the vanishing gradient problem?