EE 182 Scribe notes: RNN & LSTMs, Self-supervision

Zipeng Lin, Kuba Grudzien

December 8, 2022

1 Review for last lectures

So far, we have covered convolutional networks for images and generalized versions of convolutional network, graph neural nets.

We have the following are features for convolutional network for images.

Features of convolutional nets for images:

1. Weight-sharing across space
2. Residual Modules to support depth. Recall how we use the skip-connection to make sure the gradient is not vanishing.
3. Pooling to more quickly support long-range dependency. Pooling can process the convolutional result quickly.

Features of graph neural network (topology of image $\rightarrow$ topology of graphs) :

1. Weight-sharing across graph nodes. This is a more generalized version of CNN sharing since instead of grids, this shares weights across graph nodes.
2. modules in each layer can reference self, global state, and neighbor. This is because of the structure of graphs.
3. But need to aggregate symmetrically over neighbours. This is because we still need to maintain the invariance of the model.

2 Filters

We want to understand RNNs, and it turns out we want to first recall the filters in signal processing. They give the ideas of recurrent neural networks.

Definition 2.1 (FIR: Finite Impulse Response: generalized moving average)**.** ***Moving average*** *is taking the average while moving across different inputs. Recall in the previous lecture that momentum is just an exponentially weighted average.*

An example is output $y[t]$

$$y[t] = \sum_{i=-k}^{+k} x[t-i]h[i]$$

where x and h are the inputs and impulse response respectively.

*This is the definition of **convolution** (discrete). Notice that this is the same thing behind convolutional neural networks. Indeed, when we generalize this to 2D dimension we get convolutional neural networks.*

What if instead of having finite signals, we have infinite signals? This motivates us to look into IIR: infinite impulse response. Before going to the definition, we first realize that we can not really express the infinite operation by just writing them out. **We want a compact way** to write out infinite operations. The **key idea** to do this is to use hidden states.

Example 2.2 (Infinite impulse)

Consider the recurrence relationship

$$y[t] = a * y[t-1] + b * x[t]$$

we have both $x[t]$ and $y[t]$ are current state, while $y[t-1]$ is for past state. We get a sense that IIR is strongly related to time. Recall momentum discussion in lectures: results in exponentially weighted average.

The IIR filter has the key property that it processes the input sequentially. When there are inputs needed to be processed sequentially, IIR can be useful.

Similarly, we can process inputs sequentially:

$$\vec{y_t} = A\vec{y_{t-1}} + B\vec{x_t}$$

the vector version (still linear). The key idea is that the current state depends on the past state, and it is like momentum.

Let us consider a linear example: Kalman filter. It is a way to track the system.

Example 2.3 (Learned Kalman Filter)

Textbook definition of Kalman Filter (K.F): given known dynamic for some linear system driven by Gaussian Noise with known covariance. These dynamics have the state $\vec{h}$ hidden (think about this like the intermediate weights in MLP). Instead, we observe

$$\vec{x_t} = c\vec{h_t} + \vec{v_t}$$

where c and $\vec{v_t}$ are known, and $\vec{v_t}$ is known statistics.

K.F is a linear example. Therefore, we compute the K.F Dynamics (From known dynamics and observation structure).

$$\vec{h_{t+1}} = A\vec{h_t} + B\vec{x_t}$$

In a learned Kalman Filter, we do not know the dynamics. We want to learn A, B weights from data. Also, we know A, B matrices are constant across time. We want to solve the coefficients in recurrent relationships.

Let $W = A, B$ and h, x be the hidden state and input. W is the unknown weight to be learned. Assume we have traces of train data $\left(\vec{h_{t,j}}, \vec{x_{t,j}}\right)_{j=0}^{n_j}$ for the js being $1, 2, \ldots, m$. Notice that n_j is the length.

Setup of the RNN: see the figure 1, we input x from the bottom and input initial hidden state $\vec{o}$, the above are loss layer to calculate loss function and do gradient descent.

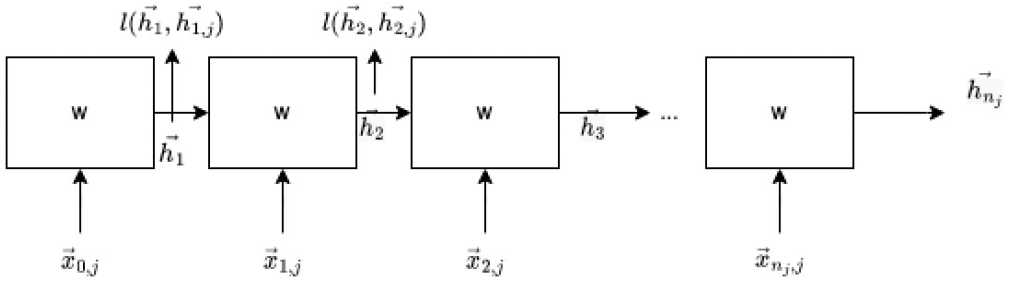

Figure 1: Simple-RNN

Now consider the inputs and outputs.

We have the inputs including the real world system, real-world states h_t, real random inputs u_t to this system, and real measurements/outputs x_t

We want to build a system (Kalman filters), a computation system specifically, to estimate the $\vec{h_t}$. In the textbook definition of Kalman filter, we just compute the system. The output of real-world-system is input for our computational system. The diagram from lecture below could improve your understanding:

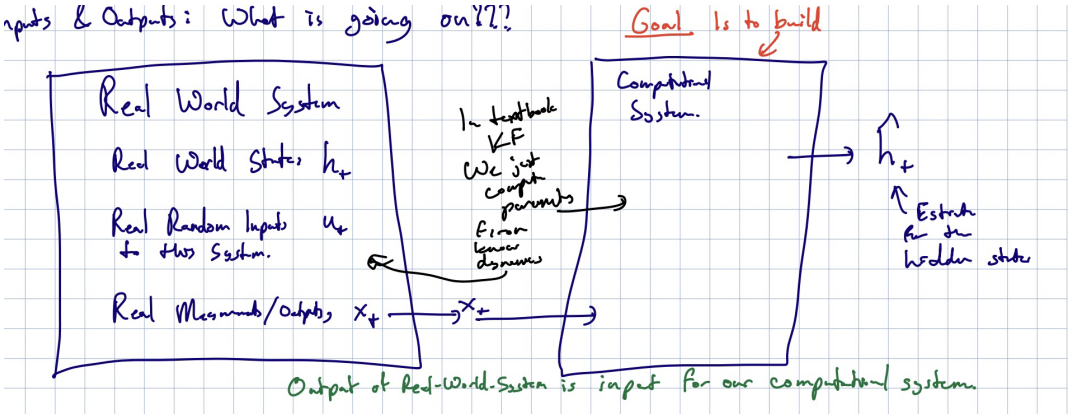

Figure 2: Kalman filter system

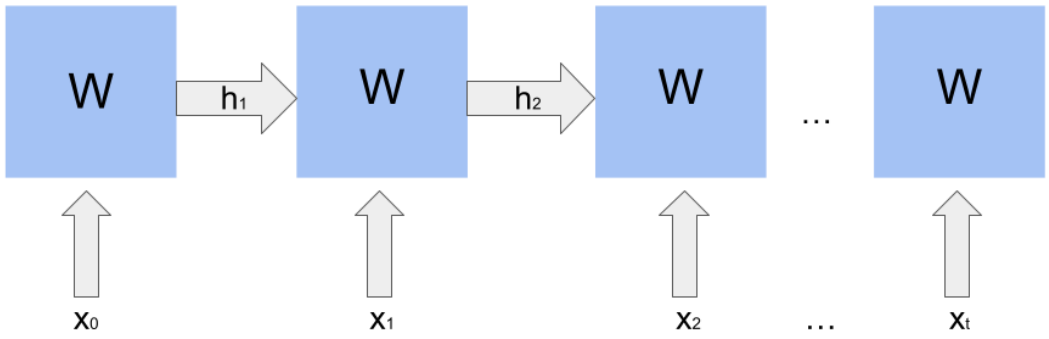

Figure 3: Simple RNN with input x and hidden state h

3 Recurrent neural network

Question: why do we take the loss like that? We compare our estimated value to the real value.

Question: why don't we compute h from x? This is because we want h to be dependent on time and want our system to reflect that. In the real world, we do not know what real data and we want to learn the filter dynamics.

We generalize the system described above to include non-linearities. (Aside, for the linear features, recall the neural network lectures, we can replace linear features with MLP).

In MLP, we can add expressiveness to the model by making the layers wider and the model deeper. Now the question is how to make complicated RNN models expand expressiveness. Recall the figure of RNN from above (figure 3). We can approach the problem from two kinds of perspectives: either we change the internal structure of RNN by a little bit, or like doing with other deep learning models, we stack over layers.

3.1 Choice one: stick to the picture but make things wider/deeper

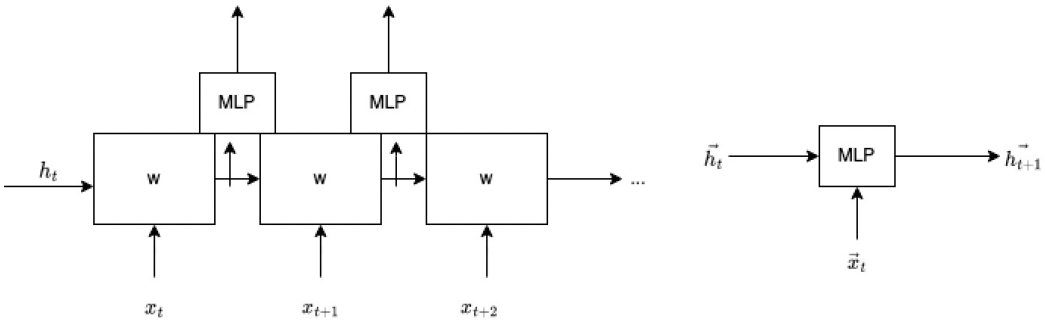

Figure 4: Option 1

We change two things here: the MLP and the h_t. We make both wider (MLP can be deeper too) to make it more expressive.

3.2 Choice two: Use layers of simpler RNNs

We use layers of simpler RNNs. Treat each RNN as a filter, we compose filters together.

It is the same way that convolution network gets deeper. We can put an output layer above, and the inputs are on the ground. The gradient could flow in two directions.

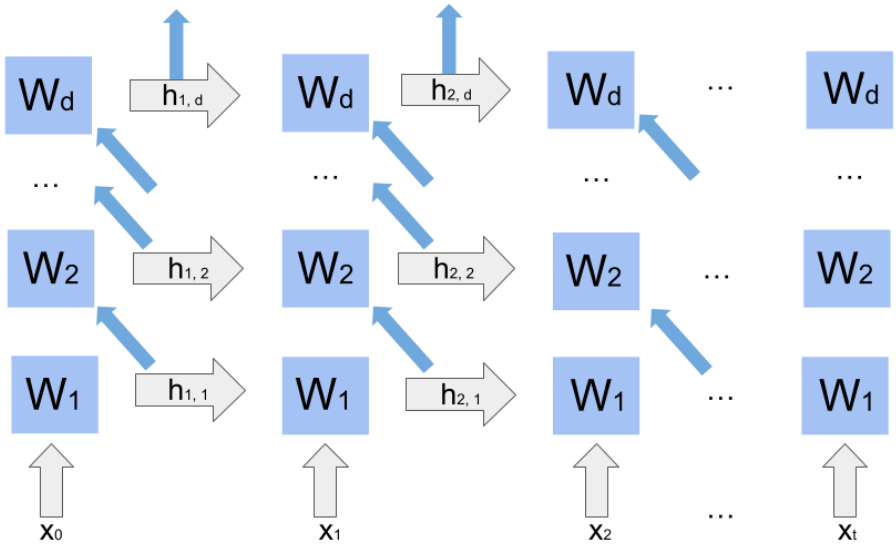

Figure 5: Option 2

3.3 RNN challenges

Similar to the challenges we face while using layers of CNN, we also need to deal with **dying gradients** and **exploding gradients**. In order to do that, we want to use a saturating non-linearity.

Definition 3.1 (non-saturating)**.** *A function f is non-saturating if and only if either its limit at ∞ is $+\infty$ or its limit at $-\infty$ is $+\infty$.*

Definition 3.2 (Saturating)**.** *A function is saturating if and only if f is not saturating*

Example 3.3 (ReLU is non-saturating)

The function ReLU reaches ∞ when $x \to \infty$ so it is non-saturating, which is why it is not often used in RNN. However, in the PyTorch version of RNN, you can still use ReLU to explore the details of computing.

Thinking 3.4 (Why not to use non-saturating nonlinearity?)
Since RNN are recurrent, the gradient would involve a lot of multiplication. Therefore, in order to prevent the gradient from exploding, we want to make sure the output **value** of the activation function would be smaller than one, otherwise, the result would diverge if we have too many RNN time steps.

Examples of saturating non-linearity, tanh and sigmoid.

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \in (-1, 1)$$

and

$$\text{sigmoid}(x) = \frac{1}{1 + e^{-x}} \in (0, 1)$$

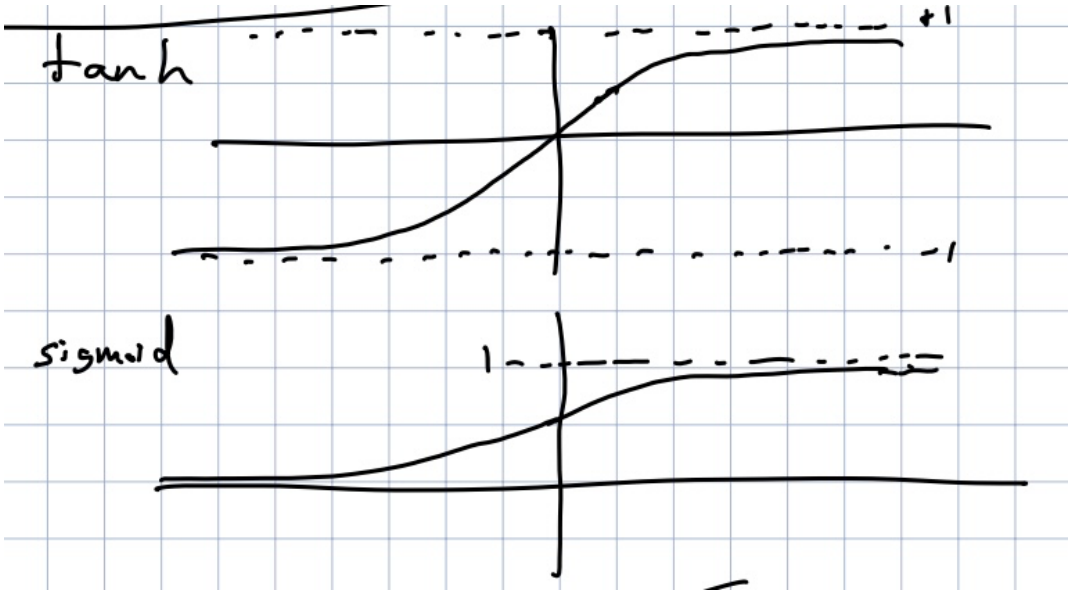

Figure 6: Graphs of two saturating non-linearities

From another perspective, we can use **Layer Normalization** on the data to prevent exploding gradient. We can apply layer normalization in two ways in the context of RNN: either we normalize the hs above, or, we can do the layer norm in the x direction. We can also put layer norm inside w_is. The reason why **we do not usually use Batch Normalization** is because it would not consider the recurrent part of the network, as in each recurrence calculation the statistics about the data would change. If you want to explore further, you can check paper `https://arxiv.org/abs/1603.09025` and see how reparametrization to get Batch Normalization work.

On the other hand, how do we combat dying gradients? Can we use "skip connection" like the one in ResNet? Yes, we can, but there should be discussions about which direction we should use

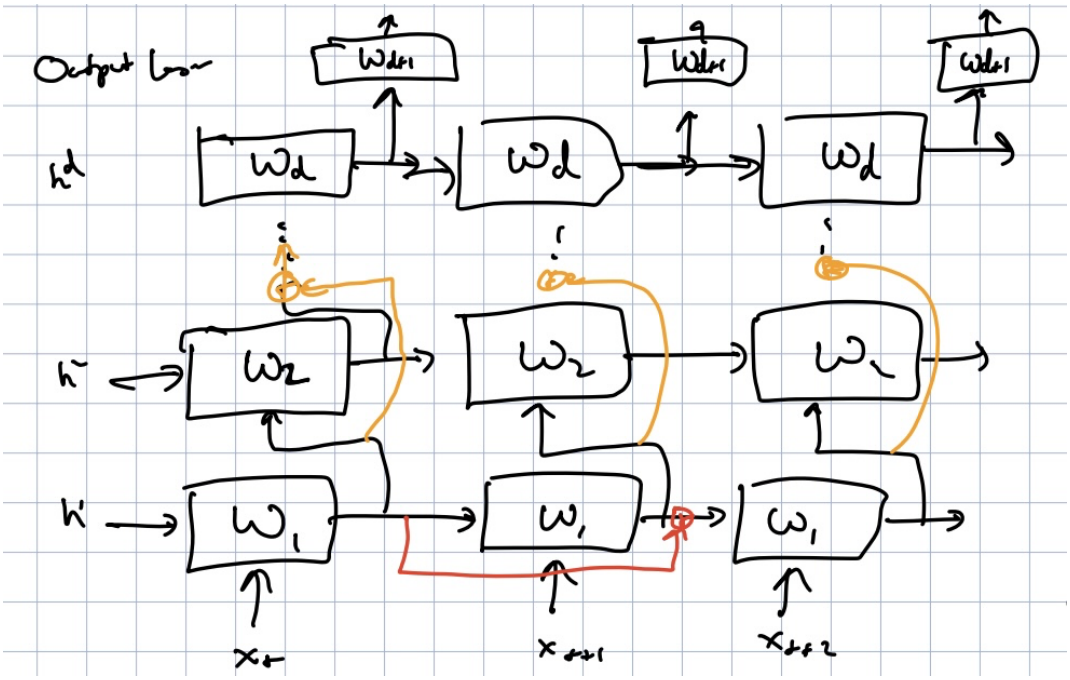

Figure 7: Several RNN layers

As we can see, the input is from the bottom, the hidden state are from the left. We have two options:

- Do ResNet skip connection in the vertical direction (orange lines)
- Do ResNet skip connection in the horizontal direction (red line)

We get the vertical skip connection could work, but the **horizontal one might not work**. The reason is that the horizontal direction is for the hidden state, so we ignore the current input slightly. In many applications, adding horizontal skips changes inductive bias in a bad way (since it can not go back without knowing the ignored inputs).

In order to address the horizontal direction of the skip connection, we introduce the following idea.

Key Idea: add a memory cell: make horizontal paths have a way for gradients to flow backward when those gradient values make sense, but which can learn to block gradients as well.

Context is represented by the symbol c_t and it should change but at a slower rate most of the time. However, when the input changes a lot, the context should change too. Most of the time, we want to multiply that with f_t. Usually, f_t is 1. However, sometimes f_t is not 1, we thus multiply the input by $1 - f_t$ and we have new context c_{t+1}. When $f_t = 1$, $(1 - f_t) * \text{input}$ has no affects on the context. f_t is the forget gate (1 being remembering). An example of f_t is

$$f_t = \text{sigmoid}(w_1 x_t + w_2 h + w_3 c_t + \text{ bias})$$

with $w_3 c_t$ sometimes being ignored. The input here would be

$$\text{input} = \tanh(w_1 x_t + w_2 h + w_3 c_t + \text{ bias})$$

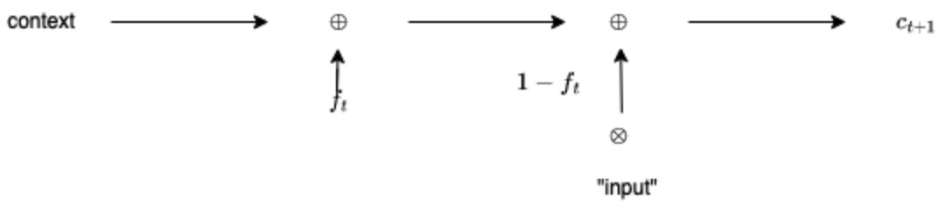

Figure 8: Flow in LSTM

The approach that follows the idea is adding memory to recurrent unit in addition to hidden state h_t.

Twist in practice (how it differs): usually, the current state needs to include the context, so we have the hidden state at time $t+1$, $\vec{h_{t+1}}$ is equal to

$$\vec{h_{t+1}} = \vec{o} \odot \tanh(c_{t+1}),$$
$$\odot : \text{element wise product},$$
$$\vec{o} : \text{computed with non-linearity from} x, h, \text{ and probably } c$$

this implies LSTM, which is in the discussion worksheet.