CS 182/282A Deep Neural Networks **Spring 2023**

Lecture 13: RNN Designs, LSTM

Professor: Anant Sahai *Scribe: Alex Wong, Bill Zheng*

1. Recap: Recurrent Neural Networks

1.1. RNN Structure

Last lecture, we discussed how Recurrent Neural Networks (RNNs) are inspired by Kalman Filters to learn the values of the hidden states, and we use this to filter and process sequential inputs to fit our desired output.

Recall the structure of an RNN layer that takes in the input of the given time step and the hidden state:

$$\vec{\hat{h}}_t = \sigma(\boldsymbol{W}_h \vec{\hat{h}}_{t-1} + \boldsymbol{W}_x \vec{x}_{t-1} + \vec{b}), \vec{\hat{h}}_0 = 0, \vec{\hat{h}} \in \mathbb{R}^{d_h} \tag{13.1}$$

Figure 13.1: An unrolled one-layer RNN. $\hat{h}_t$ denotes that it is the inferred output at timestep t. Image made by authors, all figures are based on illustrations in lecture **?**.

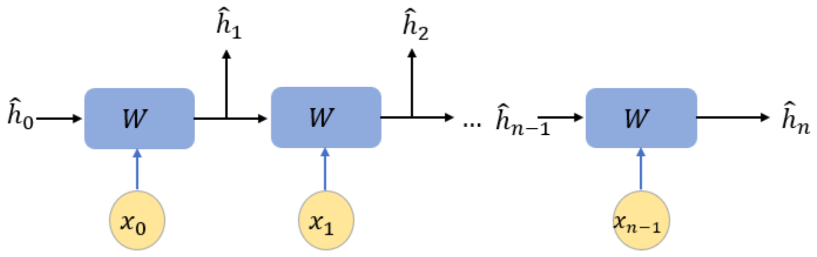
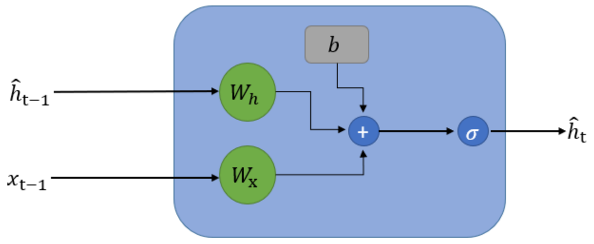


Figure 13.2: The computational graph of W



Where σ represents the chosen activation function, $\boldsymbol{W}_h$ and $\boldsymbol{W}_x$ represent the learnable "filters" that transforms both the input and the hidden states, d_h represents the dimension of the hidden state, and $\vec{b}$ represents the shared biases. Note that these "filters" can take on many forms, such as matrices or MLPs (if one were to include non-linearity).

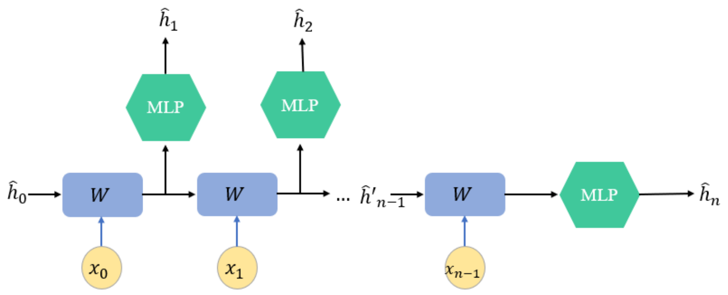
Our objective is to have our computed hidden states to match as close to the desired hidden states as possible by learning the filter weights, and we do that by using a loss function, $L(\vec{\hat{h}}_i, \vec{h}_i)$ for each hidden state output.

Thanks to weight-sharing, the transformation imposed on the hidden states and the inputs are invariant across all inputs.

1.2. Weight-sharing and workaround

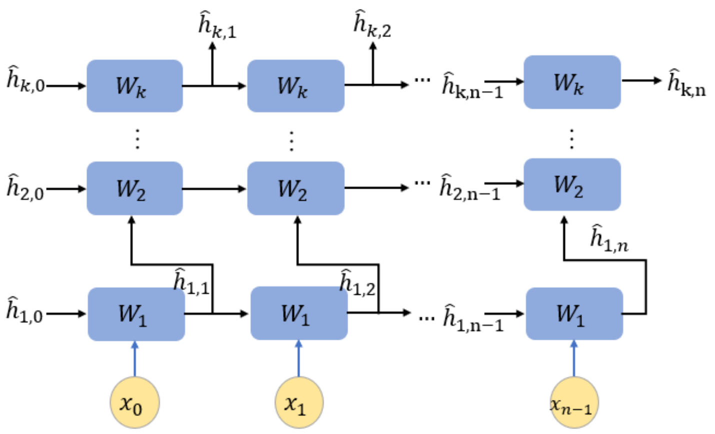
One significant obstacle these networks need to overcome comes from the need of increasing the expressiveness of this "filtering" process, given that there are only 3 learnable weights in a simple system: $\boldsymbol{W}_h, \boldsymbol{W}_x$, and $\vec{b}$. There are two ways to increase such expressiveness. The first method would be to modify the transformation W, by either appending an MLP before returning each hidden state, or if W were an MLP, by adding the number of dimensions in the output. Below is an illustration on such process, and due to weight sharing, all MLPs will share the same weights and biases.

Figure 13.3: one trick to augment the expressiveness of RNNs is through appending MLPs in this fashion. $\vec{\hat{h}}'$ means that it did not go through the MLP. Image made by authors



However, the more widely implemented method today is to compose the hidden states into more RNNs, thus providing both a better way of processing data as well as giving out a more intuitive explanation for the weights (here the inductive bias is that an RNN is better at processing sequential data compared to MLPs).

Figure 13.4: a composed RNN, where each horizontal layer (what we would consider a "layer" in a CNN) uses the same weights and biases, and each vertical layer process information of the same time frame in its own way. However, we are only taking $\vec{\hat{h}}_k$ vectors as our final return values for hidden states. Image made by authors



1.3. Questions

1. How do we interpret weights learned in RNNs?

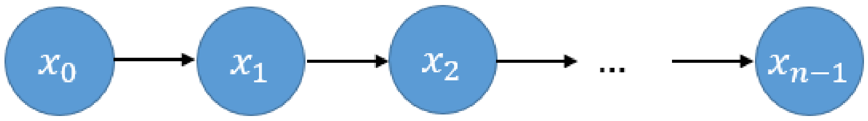
One can interpret the weights learned in the same way as we interpret weights learned in ConvNets, which in the highest, most abstract level, means that the weights are a way to process each piece of data in the sequential input. However, the interpretability of such processes are not as clear-cut as that of the CNNs, where each filter can be directly interpreted as something that either blurs the images or sharpens it.

2. What are some techniques that we have learned not applicable in RNNs?

One unique drawback (or feature, depending on how you want to look at it) of RNNs is its inability to process downsampling. While it is a great application if one were to use it in CNNs and GNNs, downsampling simply isn't really applicable in a sequential manner.

One mathematical way to approach this understanding is to consider all of the data structure that we take in as graphs (whether it is an image, graph, or a sequential input). Sequential inputs, unlike images or graphs, are not strongly connected at all regardless of the scenario (a strongly connected component within a graph means that for every node in that component, there exists a path between that node and every other node in the component), which made downsampling a rather costly operation that can damage the overall property of the graph.

Figure 13.5: a length-n input to an RNN can be transalted to a linked list, which has no strongly connected components. Image made by authors



3. How do we implement back propagation in RNNs?

It follows the same rule as any other neural networks, and since we use the same weights throughout all layers, we can use the help of chain rule to help us.

$$\frac{\partial L}{\partial W_i} = \sum_j \frac{\partial L}{\partial h_{i,j}} \frac{\partial h_{i,}}{\partial W_i} \tag{13.2}$$

In most of real life applications, we do not usually derive them by hand, instead we use computers to help us solve this task.

2. Main Challenge: Exploding/Dying Gradients

A core problem with RNNs is that by design, there are always long chains to backpropagate through. With most neural network architectures, we just have to backprop through layers (e.g. CNNs, fully connected nets), but RNNs add the sequence dimension (which is often larger than the number of layers). So, long backprop chains are created because there are connections in two directions: through the sequence and through layers.

Since long backprop chains mean more multiplication of gradients, we must take special care to manage the magnitude of gradients.

If the gradient were to explode, the RNN would return extremely large predictions (which would be displayed as NaN) very early on at iteration 1 or 2, rendering the RNN unusable.

If the gradient were to die, the RNN would have a strong inductive bias against learning long-range dependencies in the input, since the incoming gradient at each sequence step would be very small and the RNN would rely almost exclusively on the input given at that step.

We expand on solutions to each of these issues below.

2.1. Dealing with Exploding Gradients

2.1.1. Saturating Activation Functions

The most common way to deal with exploding gradients is to use saturating activation functions, which have a small gradient when the input is large. In other words, we are replacing ReLU with a function such as tanh or sigmoid.

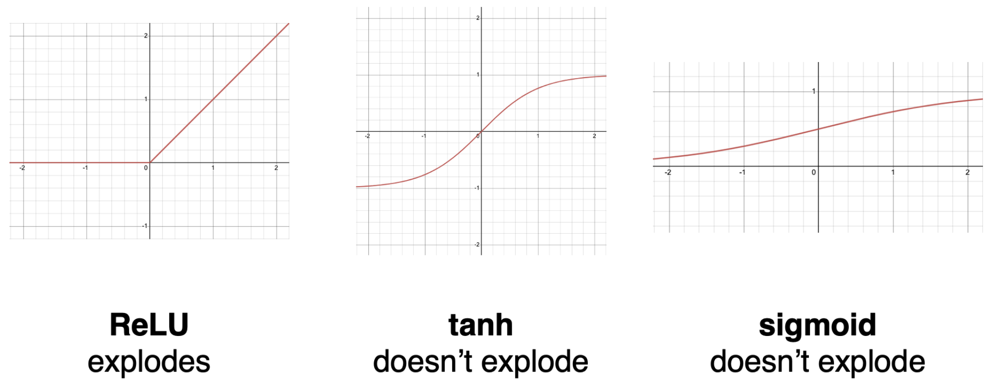


Figure 13.6: Activation functions **?**

This has the effect of "applying a brake" on the gradient; if the magnitude of the weight/output is large, the gradient of the activation function is small, meaning the weight/output cannot explode.

As an aside, saturating activation functions help overcome a shortcoming in the view of RNNs as generalizations of a Kalman filter. While Kalman filters only model linear dynamics, we believe RNNs should model real-world nonlinear physical systems. One characteristic these nonlinear systems generally have is that they do not run off to infinity, so we use tanh and sigmoid with RNNs because they have similar asymptotic behavior (unlike ReLU).

2.1.2. Gradient Clipping

Gradient clipping is a standard technique to deal with exploding gradients in any kind of neural network. This requires simply clipping the gradients if they get above a certain value.

In the case of RNNs, this will stop the gradient from exploding, but it may have unintended consequences because it will distort the "shape" on which gradient descent moves. Therefore, historically, people use tanh/sigmoid for RNNs, and gradient clipping can be seen as a last resort.

2.1.3. Normalizations

Normalization, which is a common practice in many types of neural networks, helps deal with exploding gradients (and somewhat helps with dying gradients as well). For RNNs, there are normalization variations we can perform, but we must account for some subtleties due to the two-dimensional architecture of RNNs.

Layer Norm

Similar to a CNN, you can place a normalization layer between layers of an RNN (just imagine replacing the image of a CNN with a sequence). Provided in Figure **??** is a visualization of layer norm, where x is the inputs, h is the hidden states, and W is the learned weights.

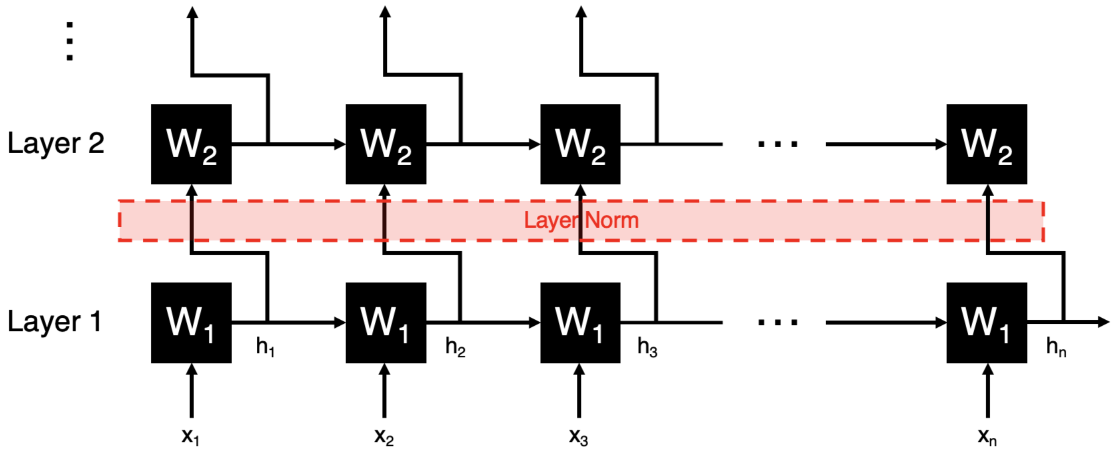


Figure 13.7: Layer norm visualization (figure made by authors)

This has no issues with training: start at the first layer, train the whole sequence from left to right, then train the normalization layer between layers 1 and 2, and repeat for all layers.

However, there are potential issues at test time. If you have receive all test data at once and generate predictions after, there are also no issues, as it works exactly the same as training. But, issues arise when you want to run the RNN on test data one step at a time (in other words, receive one input, apply all the layers at that step, receive a prediction, then repeat for the rest of the inputs). An example of this paradigm is using an RNN to generate real-time predictions on streaming data, such as with live translation.

This is an issue because without the entire sequence available, normalization is impossible, since we don't know all the values over which we want to normalize. The solution to this problem (which is the same as the solution we use to make batch normalization work at test time) is to substitute the predicted effects of all other inputs we haven't seen yet at test time to perform the normalization. This can be done, for instance, by normalizing test data with the mean and variance of the training data.

Mini Layer Norm

Mini layer norm circumvents the issue of layer norm (don't know the whole sequence at test time) by normalizing each hidden state independently as it goes to the next layer.

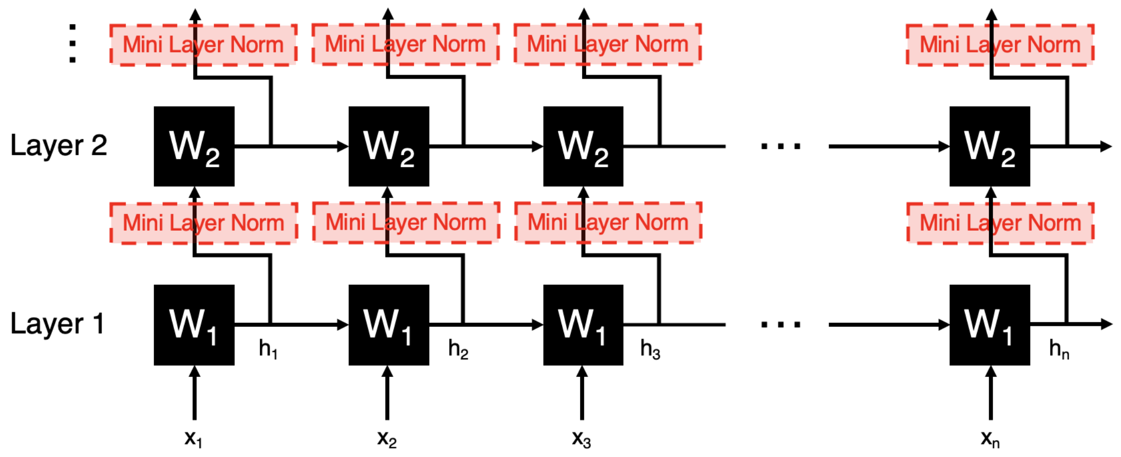


Figure 13.8: Mini layer norm visualization (figure made by authors)

Visualized, this has the effect of placing each hidden state onto a sphere or ellipsoid (norm ball) with learnable means/covariances on the individual axes.

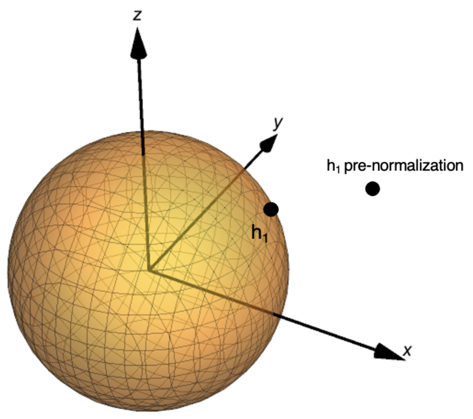


Figure 13.9: Normalization onto a sphere **?**

This works, with the caveat that the hidden states must be high-dimensional enough for the normalization action to be meaningful. As an extreme example, if the hidden state were one-dimensional, normalization would make all inputs to the next layer -1 or 1, and this loss of precision would almost certainly not be desired.

Mini Vertical Norm

Mini vertical norm is the same as mini layer norm, except instead of normalizing a hidden state's path to the next layer, it normalizes its path to the next sequence step.

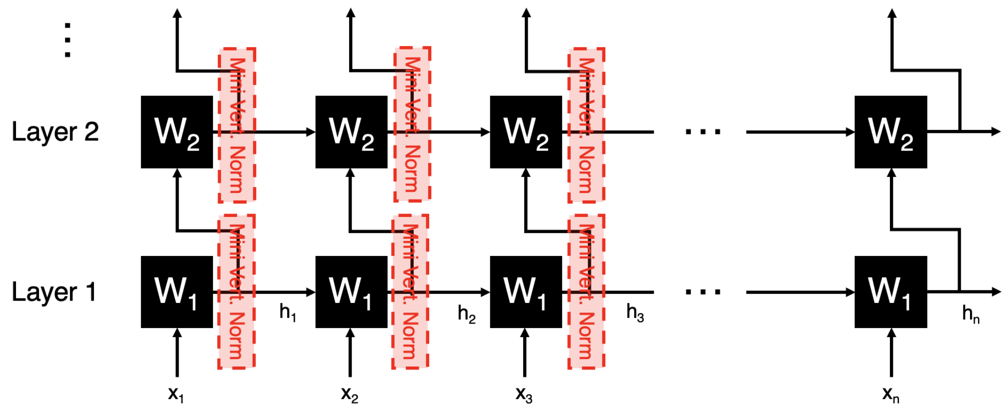


Figure 13.10: Mini vertical norm visualization (figure made by authors)

This works as well, with the same high-dimensional caveat as mini layer norm.

Vertical Norm

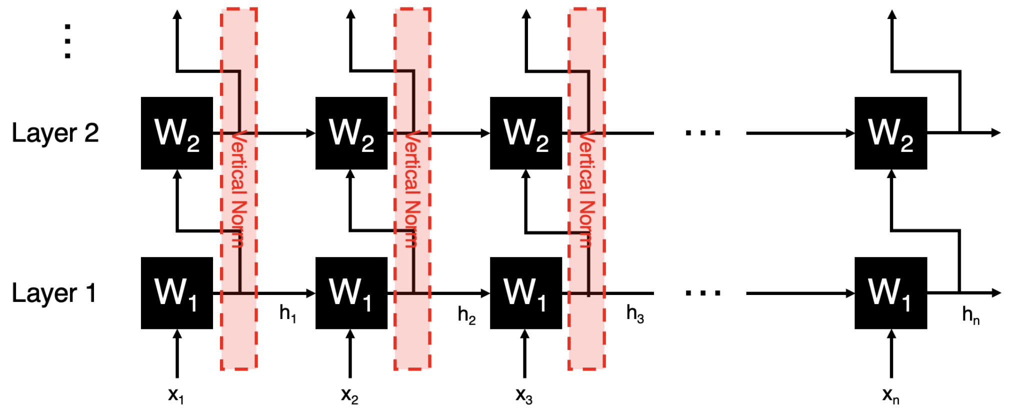


Figure 13.11: Vertical norm visualization (figure made by authors)

This is not done in practice. There is no issue with computation, it is just not practical or desired to mix the effect of different learned filters.

Batch Norm

This is possible, but not usually done. It is more difficult to implement than other forms of normalization because sequences from different samples could have different lengths.

2.2. Dealing with Dying Gradients

The basic approach we learned for dealing with dying gradients is to use residual/skip connections, where the output from one layer is passed to later layers in addition to the layer directly following it. We originally saw this with ResNets, but we can apply it to RNNs as well.

2.2.1. Vertical Skips

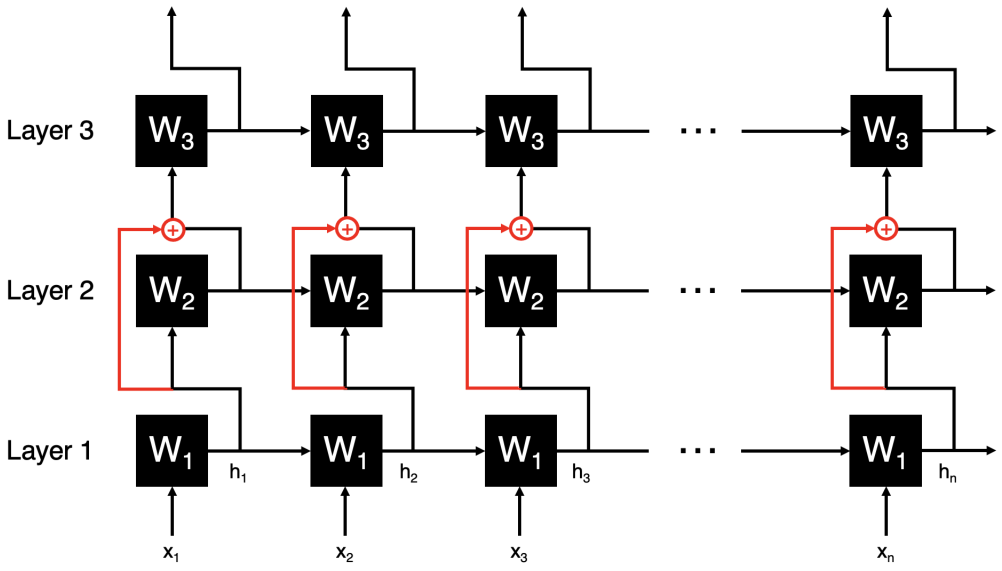


Figure 13.12: Vertical skips shown in red (figure made by authors)

With vertical skips, the gradient flows downward through the skips, so all layers "feel" the gradient more. This can be done with no issues, though it may be useful to make the output of each layer more complex than an MLP and a nonlinearity.

2.2.2. Horizontal Skips

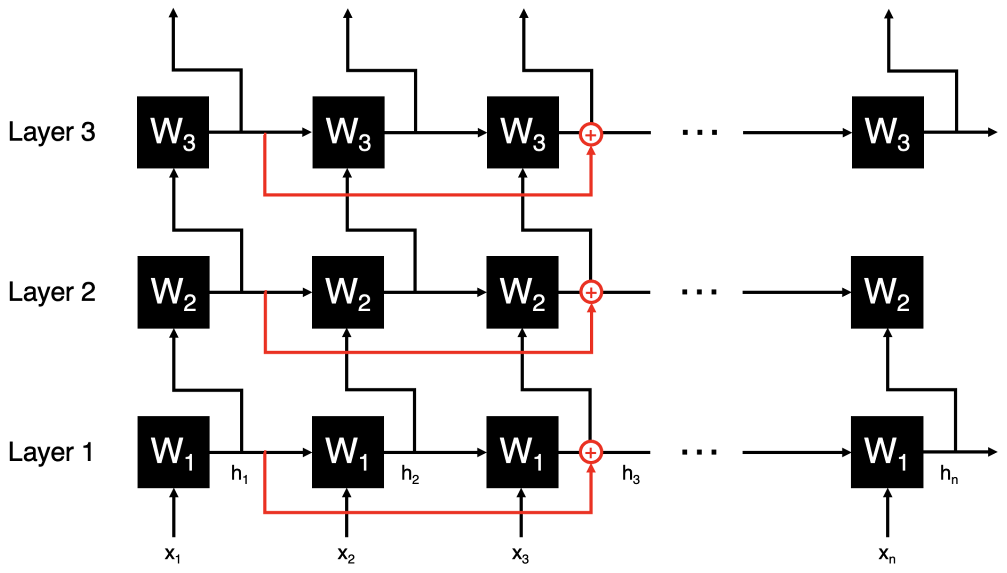


Figure 13.13: Horizontal skips shown in red (figure made by authors)

Mathematically, this is not an issue, and helps with dying gradients. However, we must be careful when doing this because it changes the inductive bias of the RNN.

With RNNs, we typically want the learned weights to act primarily locally (in other words, learn mostly based on the input they directly receive at that sequence step). However, horizontal skips weaken this locality by bringing in an input from earlier in the sequence that will be less "dead" than the gradient coming from the sequence step directly previous. Put concisely, horizontal skips mean less favoring of local information (inputs nearby in sequence position).

Is this a bad thing? It depends on the application. If it relies heavily on local data (short-term dependence), we would need a lot more data to force the RNN to look locally instead of using gradients from the past. If it requires more long-term dependence, then we are okay.

2.2.3. Dying Gradients Note

Overall, dying gradients aren't a huge problem because gradients are biased toward short-term dependence, which is often what we want. But, in some domains, there's definitely a weird balance between wanting to solve dying gradients but also wanting some gradients to die. As an example, RNNs are frequently used in linguistics domains, and many linguistics problems are difficult because they require both long-term and short-term dependence.

One possible solution to this is offering the neural network the choice to either learn short- or long-term dependence and have it learn what to do. This is the idea behind long short-term memory (LSTM) neural networks, covered later.

3. Another Challenge: Multiscale Information

The data that is typically used with RNNs, such as linguistics data, has the characteristic of being multiscale, which means that important information can be gleaned at different levels of precision (e.g. sentence level, phrase level). This presents some challenges:

1. Context in the data is long-lived (e.g. a full sentence), but can shift suddenly (e.g. that sentence ends and a new one starts).
2. Details that the RNN is looking for evolve and mix at different points in the sequence.

These challenges do exist in other data; for instance, an image could have a drastic cutoff between the interior and exterior of a building. Previously, we have used the U-net architecture as a solution, which has downsampling followed by upsampling. Downsampling helps capture global context at different scales, while upsampling before the final output allows details to still be accessible. However, this doesn't tend to work for RNNs because shifts are more sudden and can happen anywhere in the sequence.

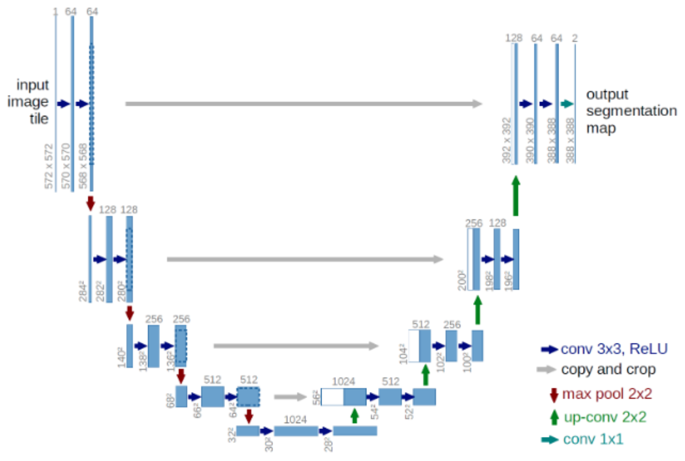


Figure 13.14: U-Net architecture **?**

Historically, LSTMs, which are covered in Section 4, have been used to solve the problem of multiscale information. However, transformer architectures based on attention have mostly taken their place because they are easier to implement.

4. LSTM (Long-Short Term Memory)

Note: this topic is not covered in the midterm, however, it is recommended that you get familiar with this concept, as this is a good gateway into topics such as attention and transformers.

4.1. Motivations

Since adding horizontal skip connections decreases the importance of the current input in the system, it must be done selectively. One good implementation of such horizontal skip connection is indication of context, since adding the 2 hidden states could represent an analog of considering these 2 inputs together. As a result, we would like to make it a learnable feature that dictates how far the gradient can flow.

Consider a sequential input such as

"Napoleon Bonaparte, a Corsican native, was one of the great military commanders in history."

English grammar would translate to a skip connection between [*Napoleon, Bonaparte*] and [*great, military, commander*] in the context of RNN, since "a Corsican native" is a clause describing Napoleon and can be omitted. However, we should not establish a skip connection between words such as *Bonaparte* and *considered.* This process of understanding contexts led to the creation of LSTM (long-short term memory) and GRU (gated recurrent unit), which both have rather similar structures.

4.2. Implementation

We introduce a new mechanism called the "forget-gate", which consists of a learned weight, $\vec{c_t} \in \mathbb{R}^h$, which is the "context" measure of the system that will get processed simultaneously as the inputs/hidden states, and a forgetting function $\vec{f_t} \in [0,1]^h$, which will affect the passforward algorithm for such context by modulating the weights of the input and the context.

The convention is that a value of 0 means the forget gate will ignore all previous contexts of this element and 1 means the gate will only consider that of the previous element. To ensure that such relationship is preserved, we can use the following computational graph to represent the context passing through a single layer (remember, this is more of a well-chosen design without a complete proof of optimality):

$$c_t = c_{t-1} \odot f_{t-1} + i_{t-1} \odot (1 - f_{t-1}) \tag{13.3}$$

$$f_t = \sigma(W_1 x_t + W_2 h_t + W_3 c_t{}^* + b) \tag{13.4}$$

$$i_t = \tanh(W_1 x_t + W_2 h_t + W_3 c_t{}^* + b) \tag{13.5}$$

*: in many cases, this term is ignored since we do not wish back propagation to change the values of c_t

To incorporate contexts in practice, we can use elementwise multiplication with respect to the hidden state h_t, thus giving us this relationship:

$$h_t = o_t \odot \tanh c_t \tag{13.6}$$

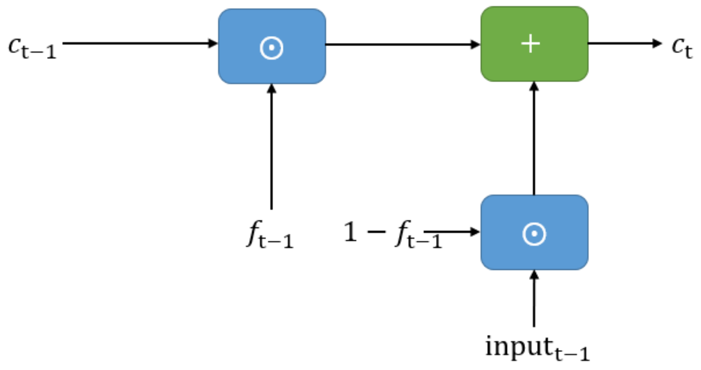
Where o_t is a combination of x_t, $h_t^{vanilla}$, and c_t, applied with nonlinearity, depending on implementation.

4.3. Questions

4.3.1. What are the advantages/disadvantages of LSTM?

Although transformer architecture has taken over most of the industry as it does not require a strict format like RNN, LSTM is still applicable when the amount of data present is insufficient to train a good implementation of what transformer also achieves. Moreover, LSTM is also applicable in terms of generating key/value pairings for attention, especially in Seq2Seq implementation, and we will cover it next lecture.

Figure 13.15: This is the computational graph for forward pass process for context, $\odot$ is elementwise multiplication



However, besides having a delicate relationship between learned weights and contexts, it is also quite computationally intensive compared to vanilla RNN, and just like all RNNs, it is not a good processor of data that are arbitrarily long, something that transformers excel at. Another way to think about it is that LSTM is somewhere in the middle between vanilla RNN and attention/transformer in terms of effectiveness and cost.

A good rule of thumb is to determine:

a. Does this problem have a well-defined input structure?

b. Are there enough training data to support a transformer architecture?

If the answer to these 2 questions are Yes and No, in order, then LSTM is still very much applicable today.

4.3.2. What type of problems can LSTM fix?

While LSTM doesn't fix the issue of exploding gradients (however other activation functions can mitigate that), a reasonably good implementation of LSTM fixes the issue of dying gradients *where it needs to be fixed*. Overall, given some of the restrictions that RNNs impose, LSTM is a good implementation of ideas that approximate contexts/attention in naive RNNs.

5. Sequential Problem Examples

RNNs are generally used in problems with sequential data, so it is worth briefly discussing applications that have this kind of data.

1. Filtering: In these problems, we feed in a sequential input and receive an output in real time. The idea is that we need to produce an output on that sequential data without the ability to look ahead. Filtering problems were the original motivation for RNNs.

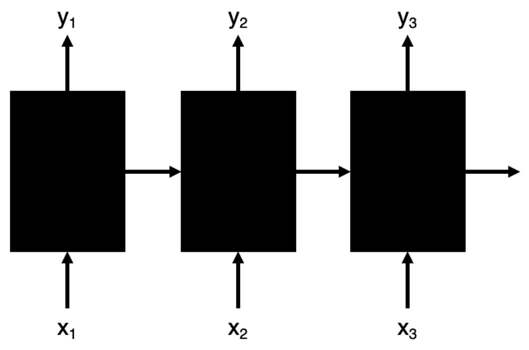


Figure 13.16: Filtering problem (figure made by authors)

2. Generation: Here, we generate a sequence where the input at each step includes the output that we've already produced (e.g. if generating the fifth word of a sentence, the input includes the first four words of that sentence). This is also called the autoregressive use of a network. Like filtering, generation is a real-time problem, so we cannot look ahead at future outputs when determining the current output.

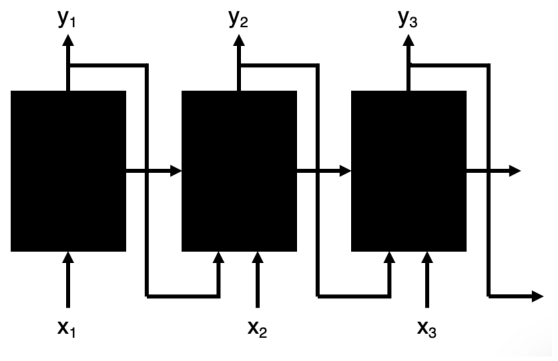


Figure 13.17: Generation problem (figure made by authors)

6. What we wish this lecture also had to make things clearer

We wish this lecture contained more mathematical proofs/notation when describing the different normalization methods, time permitting.