CS 182 Lecture Notes, Spring 2022

Deep Learning

Professor: Marvin Zhang
Vishal Raman

Contents

§1 02/28/2022 - Recurrent Neural Networks

§1.1 Problem Setup

We consider settings in which the features $\mathbf{x}$ represent *sequential data* which may be **variable length**. Some examples include

- Text of different lengths
- Audio recordings/waves
- Video recordings

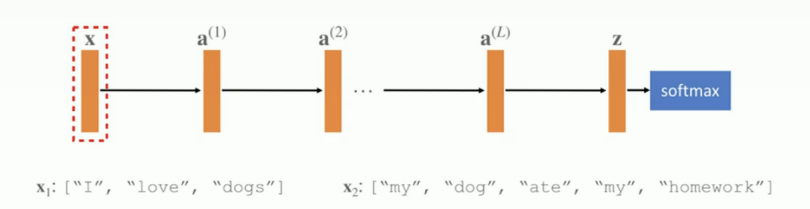
The labels could be scalars y(sentiment analysis, identification), or they could be sequences $\mathbf{y}$(translation, transcription, captioning). There could also be no label at all! This is true in unsupervised learning/generating modeling.

Some models we use include:

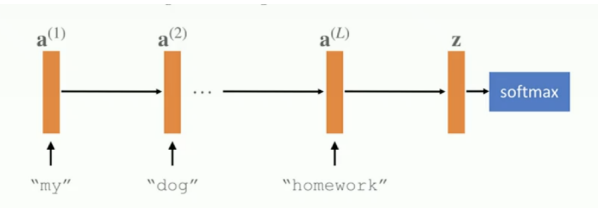
- Markov/n-gram models, hidden Markov models(HMMs)
- Embedding/clustering-based methods
- Convolutions("temporal" convolutions which could be a 1-d convolution of a sequence, or a 3-d convolution over a video).
- Recurrent Neural Networks(RNNs)
- Long short-term memory(LSTM)
- Gated Recurrent Units(GRUs)
- Transformers

§1.2 Dealing with Variable Length Inputs

Before, when dealing with images, we could reasonably assume fixed size inputs. However, with sequential data, it can be assumed that the input lengths will almost always vary.



To deal with this, we feed one piece of input called a **token** per layer.



Now, the input to layer $\ell + 1$ is $[a^{(\ell)}; x[\ell]]$.

The issues with this approach are the following:

- We need as many layers as the max number of tokens. For very large inputs, we would have extremely deep networks which are difficult to train.
- We have a different $W^{(\ell)}$ and $b^{(\ell)}$ for each layer, which is a massive number of items to optimize.
- The later layers get trained very rarely, so it's hard to generalize to longer sequences at test time.
- Another issue is that $a^{(1)}$ is missing the previous layer output.

§1.3 Recurrent Neural Networks

To solve some of these issues, we use **weight sharing**: use the same parameters in every layer.

$$\text{Before: } a^{(l+1)} = \sigma(W^{(l+1)}[a^{(l)}; x[l]] + b^{(l+1)})$$
$$\text{Now: } a^{(l+1)} = \sigma(W[a^{(l)}; x[l]] + b)$$

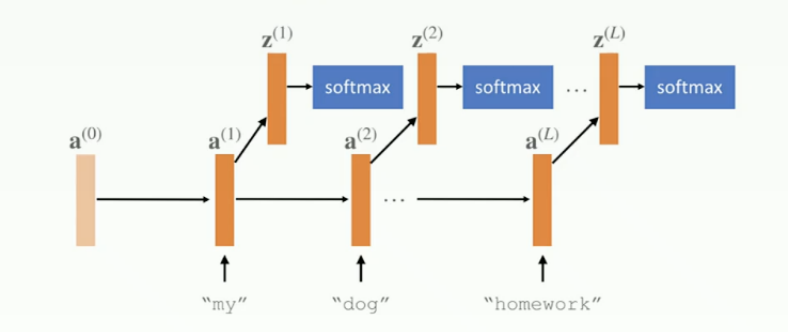
Remark 1.1. In backpropagation, W and b get a gradient signal from every single time they are applied in the network, and we sum them to get the final gradient for W and b.

To address the problem with $a^{(1)}$, initialize some $a^{(0)}$ independently from the input x and feed it into $a^{(1)}$.

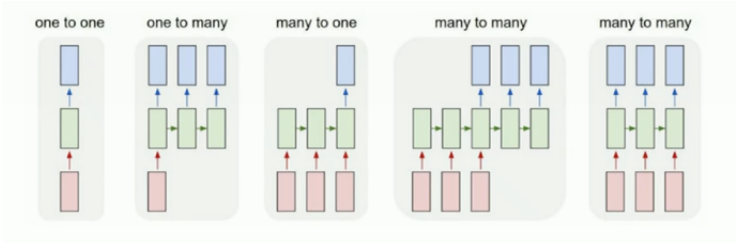
Putting these two ideas together gives us the Recurrent Neural Network(RNN).

Remark 1.2. The values $x[l]$ are encoded from some word2vec algorithm and the end vector is the same length for each word.

The above applies for a sequence input, single output. If we wanted sequence input, sequence output, we could output a scalar at each layer to get a sequence output.



In general, different applications give rise to different ways we use RNNs.

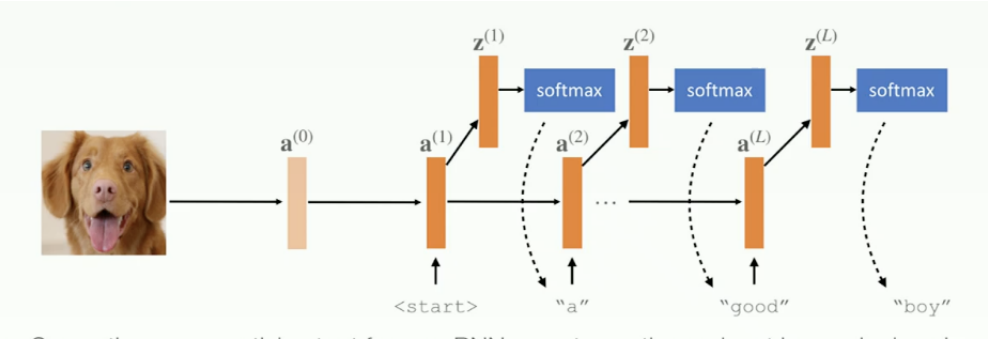


- One-to-one is not an RNN. It is like the ConvNet or Fully-Connected Net we have worked with.

- For translation, we might do something like the 4th diagram, since we need to see the whole sentence before we can translate.
- Image Captioning is single input, sequence output.
- Sentiment Analysis is many to one, sequence input single output.
- Text generation would be many to many like the last diagram.
- The many-to-many can have many different styles depending on the application.

§1.4 Generating Outputs from RNNs

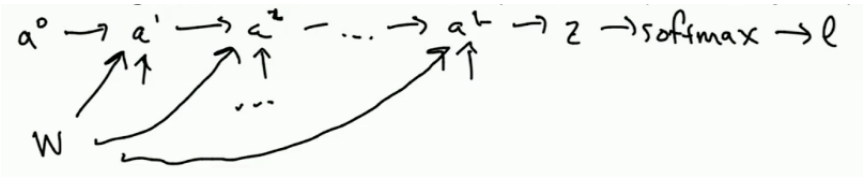
Generating a sequential output is done in an **autoregressive** manner: condition the model on what you have seen before to generate the next thing.



Remark 1.3. In the above, we assume that the RNN is already trained, and then we use it to generate text using the above scheme.

§1.5 Vanishing/Exploding Gradients

What is the gradient of the final loss with respect to W/b?



We have

$$\frac{d\ell}{dW} = \sum_{k=1}^{L} \frac{d\ell}{da^k} \frac{da^k}{dW},$$

$$\frac{d\ell}{da^1} = \frac{d\ell}{da^L} \frac{da^L}{da^{L-1}} \cdots \frac{da^2}{da^1},$$

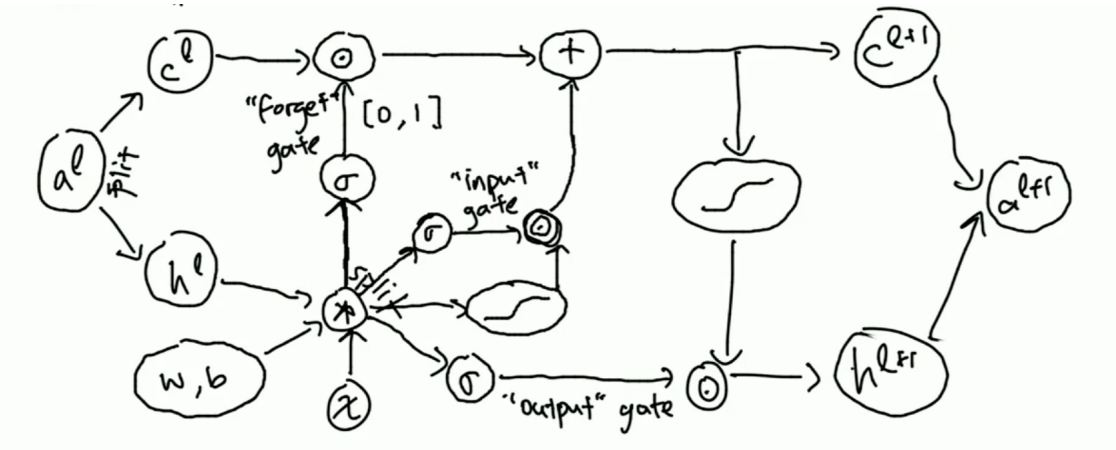
so it is very easy for gradients to vanish or explode.

We can deal with this as follows:

- For exploding gradients, we can just **clip** the gradients - divide by the magnitude of the large gradients to make them smaller.
- Vanishing gradients seem to require clever architecture choices: one idea we can use to address this is a **skip connection**!

- One architecture that does this is the **LSTM**: it is much older than skip-connections but employs the same basic principle that allow better gradient flow by making smarter connection choices.

§2 Long Short-Term Memory(LSTM)



1. Start with a state a^ℓ and split into two states, c^ℓ the cell state, and h^ℓ the hidden state. We have $d_c = d_h$, and $d_a = d_c + d_h$, since we exactly split a into two parts.
   - c^ℓ is the part that acts like a skip so we have better gradient flow with long sequences.
   - h^ℓ deals with the nonlinearities which we need for expressivity.
2. For the cell state c^ℓ, we element-wise multiply by scalars f^ℓ in $[0, 1]$(we can do this by passing some number into the sigmoid σ). This is the **forget gate** - when the scalar is close to 0, we forget these dimensions, and when the scalars are close to 1, we retain the information and gradient flow. Then, we add a bias i^ℓ, which is called the **input** to the cell state. This gives us $c^{\ell+1}$. Overall, we have
$$c^{\ell+1} = c^\ell \odot f^\ell + i^\ell.$$
3. For the hidden state h^ℓ, we use an affine transformation using parameters W, b and the input x. This is split into 4 equal parts each with the same dimension as h:
   - The first piece determines f^ℓ, the element sent into the **forget gate**.
   - The next piece is sent to a nonlinearity(doesn't matter which one) and a piece sent to the sigmoid which is elementwise-multiplied to the nonlinearity output. This is called the **input gate**.
   - The last piece is sent to a sigmoid to be used for the output gate.
4. We also take c^{l+1} and use a nonlinearity(doesn't matter which one) and element-wise multiplied by the component from the output gate to determine h^{l+1}.

Remark 2.1. Some elements are important but others are pretty arbitrary. For example, the GRU is very similar but it doesn't have an output gate. However, it still performs pretty well.