Lecture 10: Recurrent networks

CS 182/282A ("Deep Learning")

2022/02/23

Today's lecture

- The bulk of today's lecture will cover a class of models known as **recurrent neural networks (RNNs)**, which were designed to process sequential data
- However, we will first wrap up our discussion of image data with one final cool application: **style transfer**
- Together, this material should be sufficient for working on HW2
- HW2 also covers some *network visualization* topics that we won't discuss in lecture, though these topics are well explained by the assignment itself

Style transfer

(some images borrowed from Stanford CS231n)
(some images borrowed from the original paper, Gatys et al 2016)

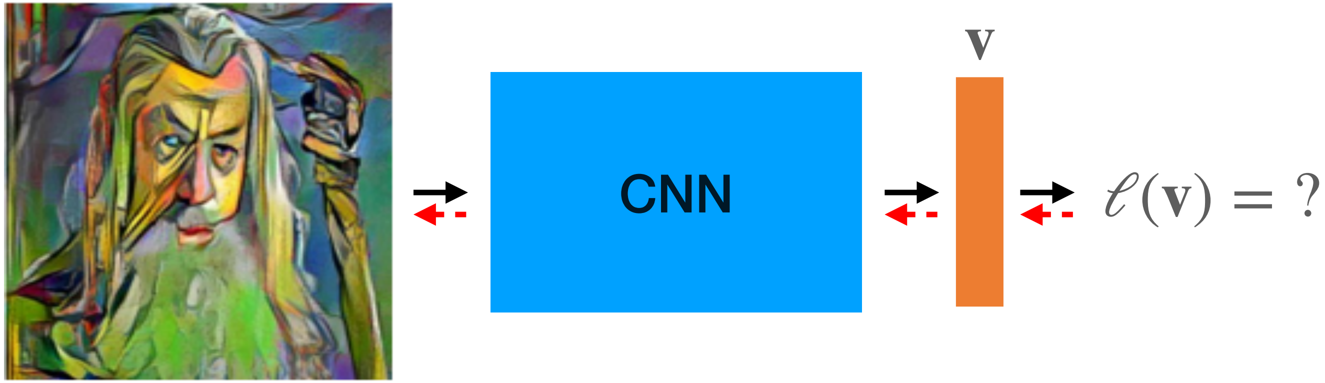
Generating images from CNNs

- Suppose you have a CNN trained to do image classification, and you wish to use the CNN to do image *generation* instead
  - This may serve a number of purposes, e.g., inspecting the model to better understand it, or just to have pretty/weird pictures to look at
- One general way to do this is to perform generation by optimization
  - Define a loss function that quantifies what image we wish to generate
  - Keep the network parameters fixed! I.e., **freeze** the network weights
  - Backpropagate the loss *to the input image* in order to update it

Style transfer, illustrated

"style" image

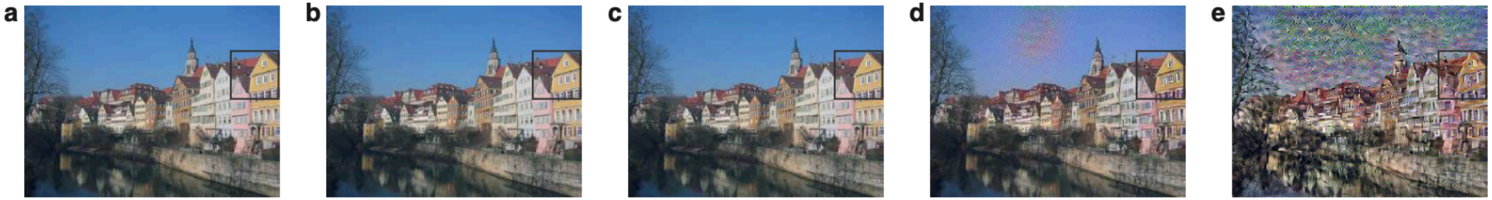
"content" image



- We will actually use a "two-dimensional" $\mathbf{v}$ which is the output of an intermediate conv layer
- The first dimension is channels, the second dimension is height and width combined

The content of an image

- If we have a content image and we only wish to generate an image with the same content from our CNN, how might we do this?
- The idea is to define a loss function that measures the difference between intermediate CNN activations when inputting the generated vs. content image
- $\ell_{\mathrm{C}}(\mathbf{v}) = \frac{1}{2}\sum_{ij}(v_{ij} - c_{ij})^2$, where $\mathbf{c}$ represents the activations from inputting the content image



The style of an image?

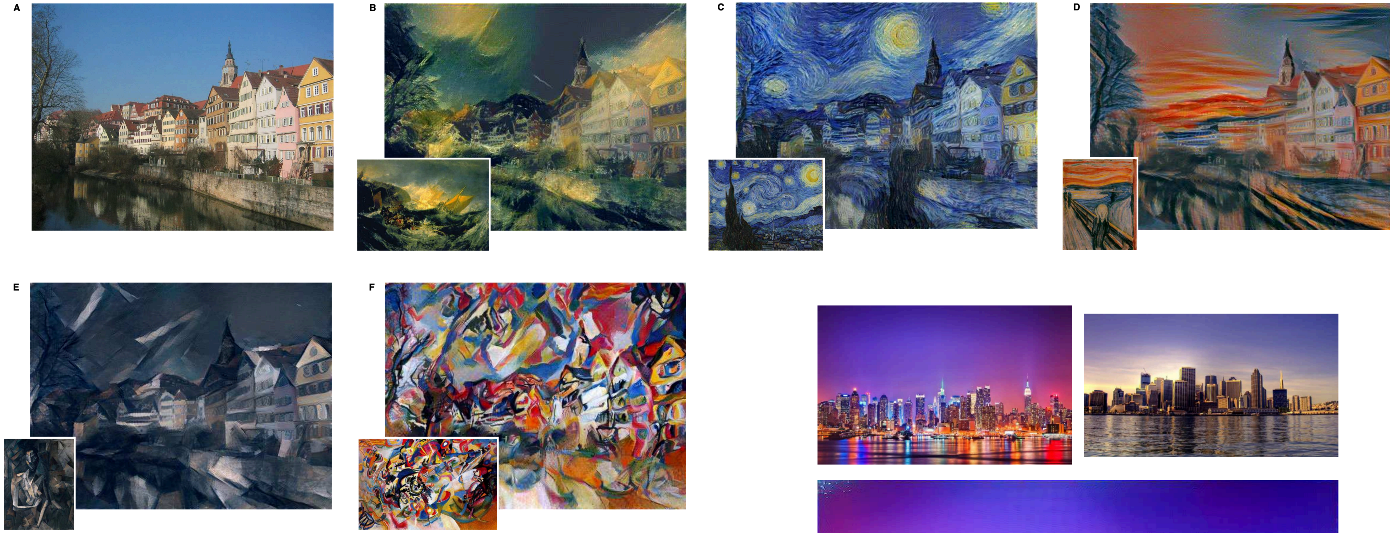
- To quantify style or “texture”, we compute correlations between the different channels of $\mathbf{v}$ via the *Gram matrix* $\mathbf{G}$
  - $\mathbf{G}_{ij} = \sum_k v_{ik} v_{jk}$ — like an unnormalized covariance estimate
- Surprising, but true: matching this Gram matrix matches styles, roughly speaking
- $\ell_{\mathrm{S}}(\mathbf{v}) \propto \sum_{ij} (\mathbf{G}_{ij} - \mathbf{G}^{\mathbf{s}}_{ij})^2$, where $\mathbf{G}^{\mathbf{s}}$ represents the Gram matrix computed from the activations resulting from inputting the style image

The style of an image?

- Different activations of the network can capture different “levels of abstraction” for the style of the image
- Therefore, unlike content matching, the style loss component operates on *multiple intermediate activations* of the CNN with different weights
- The final loss function is $\ell = \alpha\ell_{\mathrm{c}} + \beta\ell_{\mathrm{s}}$ – the authors use $\alpha/\beta = 10^{-3}$

Merging content with style

Try it yourself: https://deepart.io/



Recurrent neural networks

Problem setup

- We now consider settings in which our features $\mathbf{x}$ represent *sequential data* which may be *variable length*

It was the best of times, it was the worst of times, it was the age of wisdom, it was the age of foolishness...

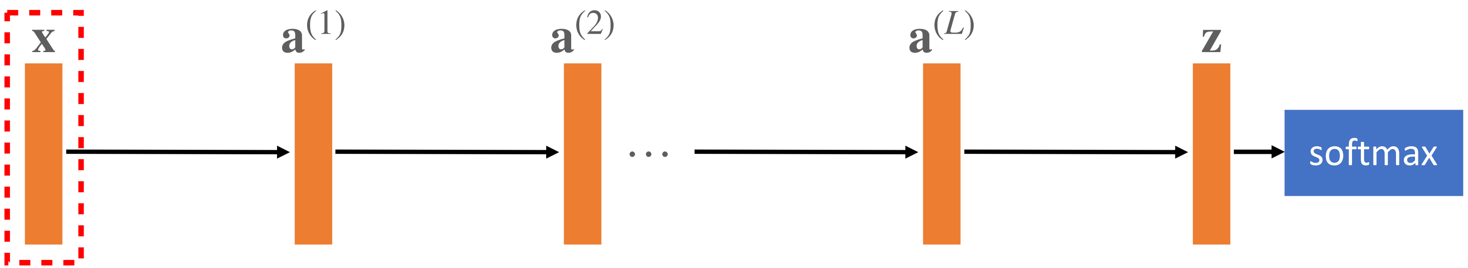
- Our labels could be scalars y, e.g., sentiment analysis, identification, …
- Or the labels could be sequences $\mathbf{y}$! E.g., translation, transcription, captioning, …
- Or there could be no label at all! I.e., **unsupervised learning / generative modeling**

Models for sequential data

- Markov / n-gram models, hidden Markov models (HMMs)
- Embedding / clustering based methods
- Convolutions (sometimes called “temporal” convolutions)
- Recurrent neural networks (RNNs) — today
    - Long short-term memory (LSTMs), gated recurrent units (GRUs)
- Transformers — in a couple of weeks

Dealing with variable size (length) inputs

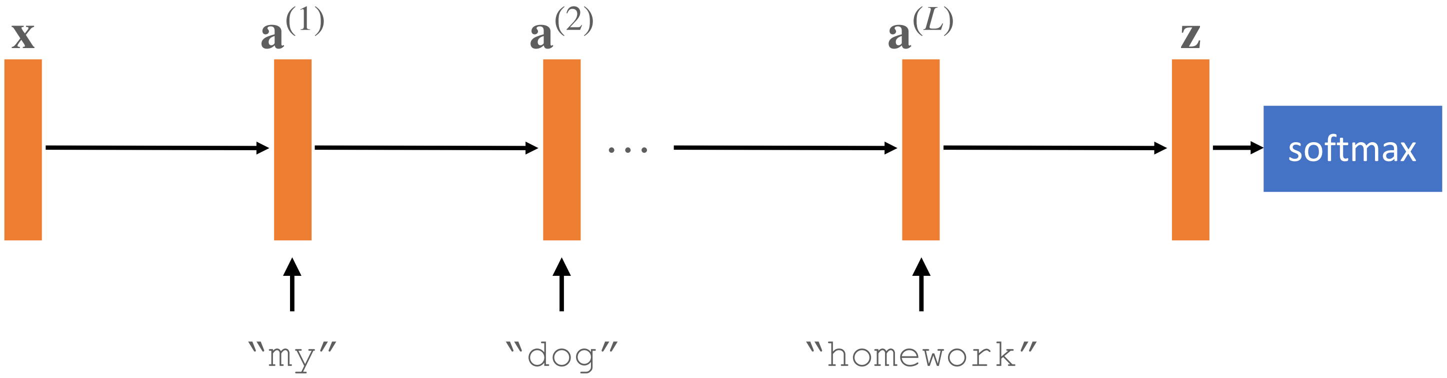
- Before, when dealing with images, we could reasonably assume fixed size inputs
- Now, with sequential data, it is often the case that input lengths vary



$\mathbf{x}_1$: `["I", "love", "dogs"]` $\mathbf{x}_2$: `["my", "dog", "ate", "my", "homework"]`

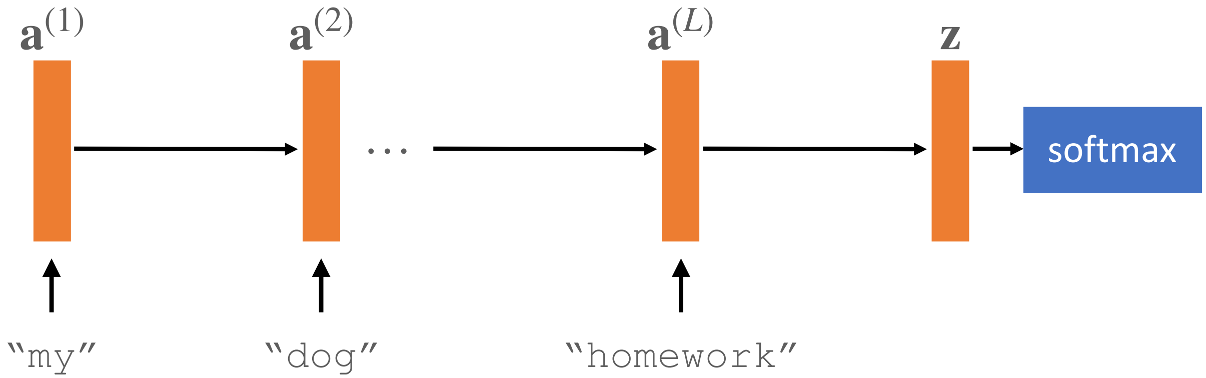
One input piece per layer?

- An idea: let's feed in one piece of the input (sometimes called a **token**) per layer
- The input to layer $l+1$ is now $\left[\mathbf{a}^{(l)}; \mathbf{x}[l]\right]$



Recurrent networks: attempt #1

What are some problems with this approach?



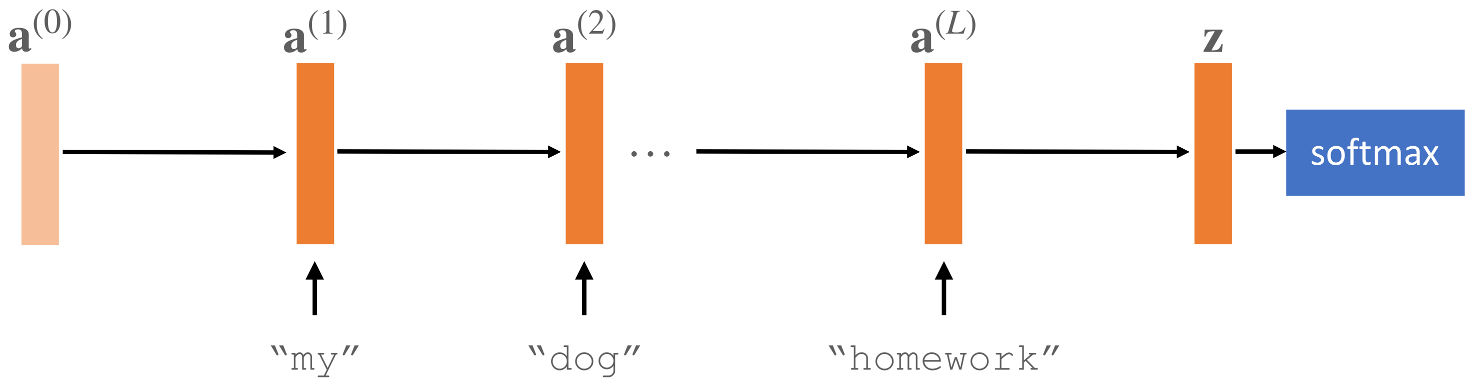
- Problem #1: we need as many layers as the max number of tokens
    - Later layers hardly get trained, and we can't generalize to longer sequences
- Problem #2: $\mathbf{a}^{(1)}$ is missing the "previous layer output"

Weight sharing

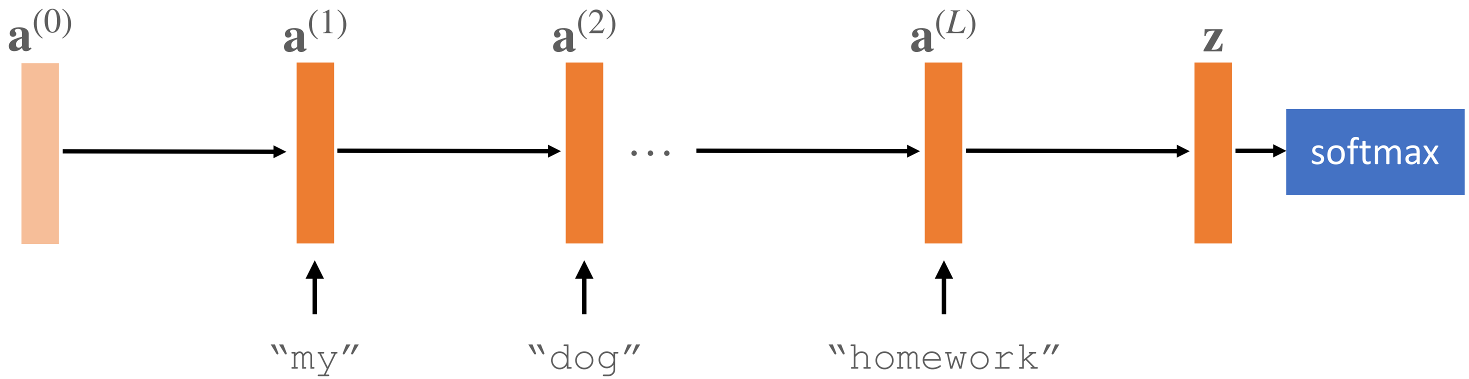
- Problem #1: we need as many layers as the max number of tokens
  - Later layers hardly get trained, and we can't generalize to longer sequences
- Solution: use the same parameters (weights) in every layer
  - This is an example of *weight sharing*
- Before: $\mathbf{a}^{(l+1)} = \sigma\left(\mathbf{W}^{(l+1)}\left[\mathbf{a}^{(l)}; \mathbf{x}[l]\right] + \mathbf{b}^{(l+1)}\right)$
- Now: $\mathbf{a}^{(l+1)} = \sigma\left(\mathbf{W}\left[\mathbf{a}^{(l)}; \mathbf{x}[l]\right] + \mathbf{b}\right)$ for all l

RNNs: the first input

- Problem #2: $\mathbf{a}^{(1)}$ is missing the "previous layer output"
- Solution: initialize some $\mathbf{a}^{(0)}$ independently from the input $\mathbf{x}$ to feed into $\mathbf{a}^{(1)}$

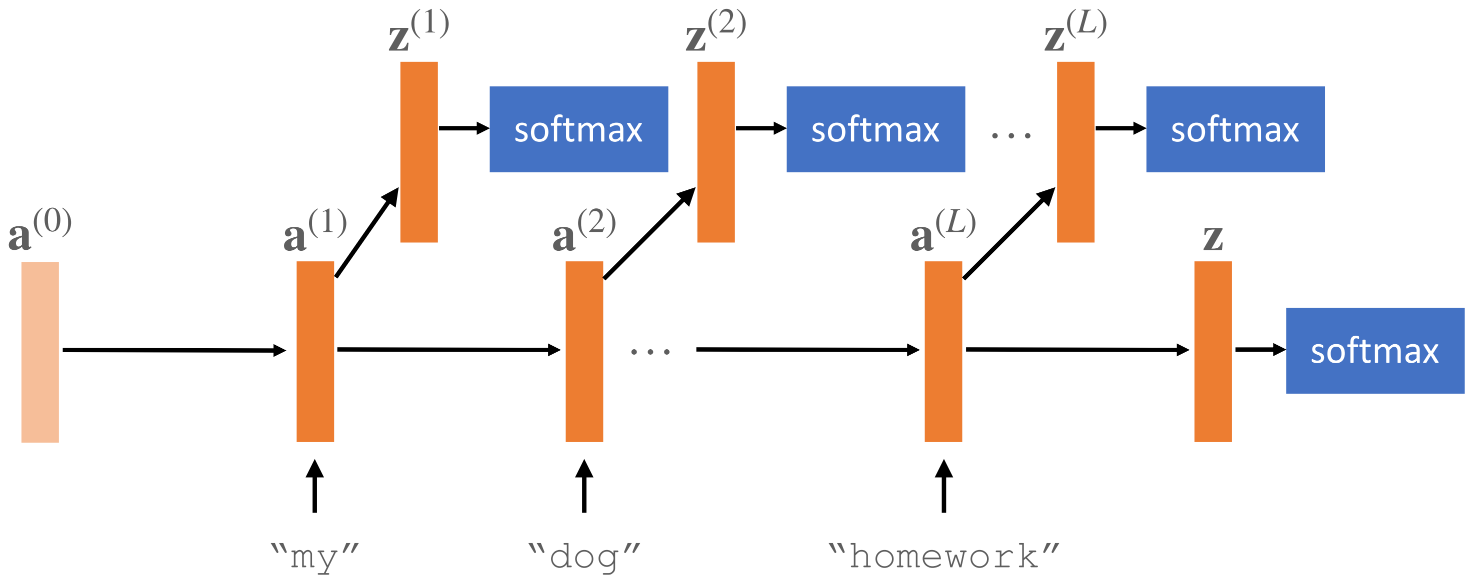


Recurrent networks: attempt #2



- Important, and not visualized here: $\mathbf{a}^{(l+1)} = \sigma\left(\mathbf{W}\left[\mathbf{a}^{(l)}; \mathbf{x}[l]\right] + \mathbf{b}\right)$ for all l
- In many applications, we think of each l as a “time step” (denoted t instead) and each $\mathbf{a}^{(l)}$ as the “state” (or *hidden state*) at time step l (denoted $\mathbf{h}^{(t)}$ instead)

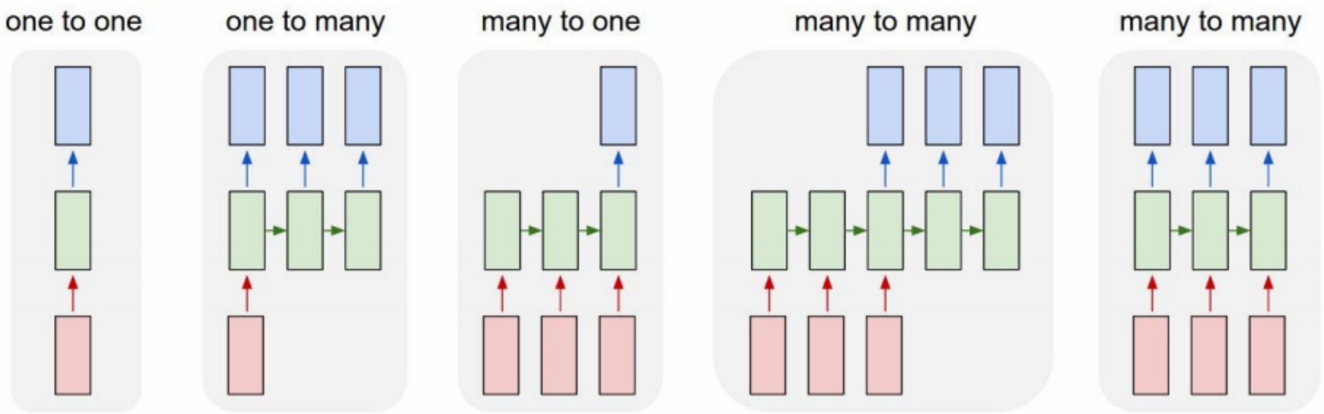
Sequential outputs



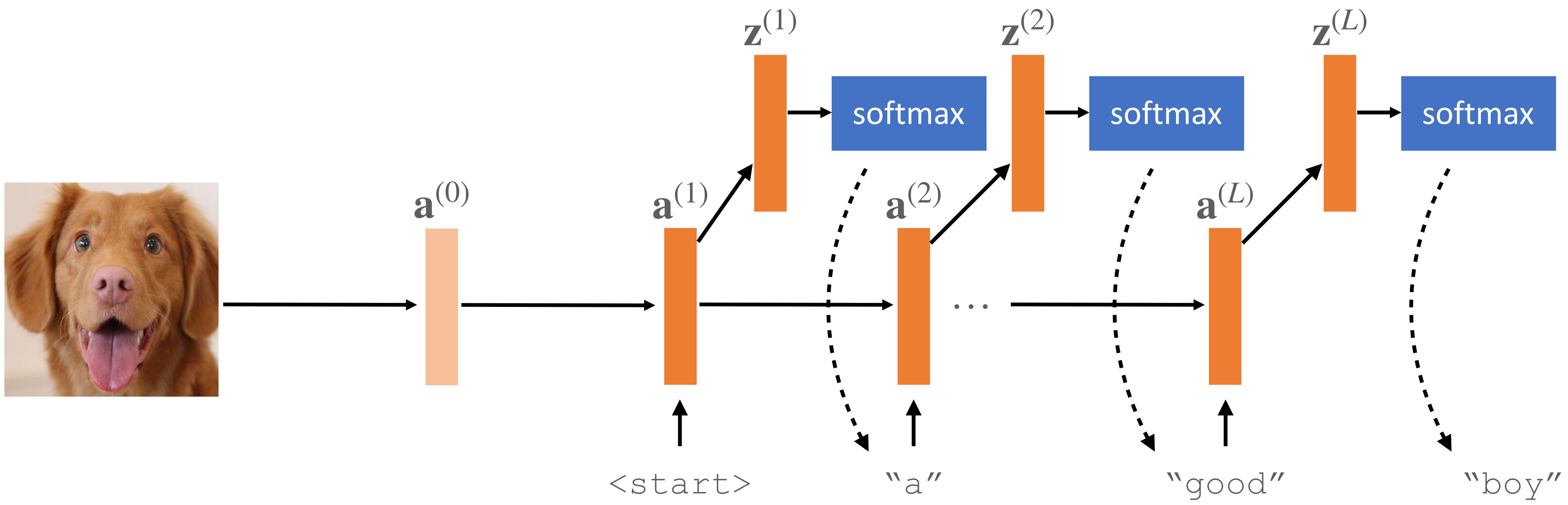
- This is what our RNN will look like for "sequence input, single output"
  - What about sequence output? Just have an output at every layer

Different combinations of (non)sequential data

- Different applications will give rise to different ways in which we use RNNs
- Match the following applications to the diagrams below that they correspond to: image captioning, text sentiment analysis, language translation, text generation



Generating outputs from RNNs



- Generating a sequential output from an RNN, e.g., to caption an input image, is done in an **autoregressive** manner
  - This makes it possible for the RNN to condition on what it has already generated

The problem with training RNNs

what is the gradient of the final loss with respect to $\mathbf{W}$? (similar story for $\mathbf{b}$)

$$a^0 \to a^1 \to a^2 \to \cdots \to a^L \to z \to \text{softmax} \to \ell$$

W $\cdots$

$$\boxed{\frac{dL}{da^L}\frac{da^L}{da^{L-1}} \cdots \frac{da^2}{da^1}}$$

$$\frac{d\ell}{dW} = \frac{d\ell}{da^L}\frac{da^L}{dW} + \frac{d\ell}{da^{L-1}}\frac{da^{L-1}}{dW} + \cdots + \underbrace{\overbrace{\frac{d\ell}{da^1}}}\frac{da^1}{dW}$$

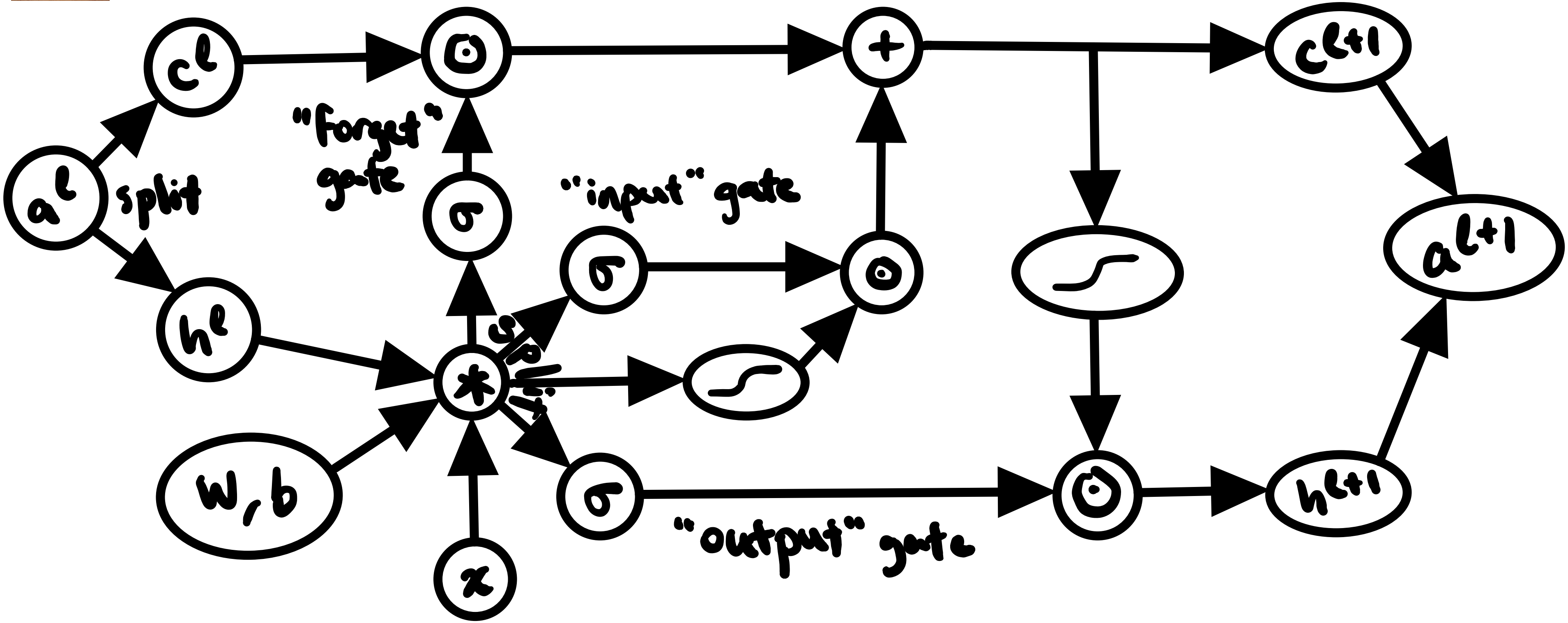
very easy for this term to be too small / large!

Fixing exploding and vanishing gradients

- We want to avoid gradients that **explode** or **vanish** as they travel backwards through the network
  - Exploding gradients are an easier problem: we can just **clip** the gradients
  - Vanishing gradients seem to require clever architecture choices
- We have already seen the basic idea behind fixing this issue: skip connections!
- Let’s detail one RNN architecture that employs the same basic principle
  - It’s not quite skip connections, but the intuition is similar — this architecture, known as the **LSTM**, far precedes the modern popularity of skip connections

Long short-term memory (LSTM)

$(d_h = d_c)$



(W is $4d_h \times (d_h + d_x)$, b is $4d_h$)

Bidirectional RNN models

- Often, it can be useful to incorporate information from “the future”, if available
    - E.g., speech transcription, *contextual* word representations, …
- For these applications, one option is to essentially learn two RNNs! One which processes the sequence forwards, and the other which processes in reverse
    - But, the RNNs are learned jointly to produce a single prediction/representation
- For a while, bidirectional LSTMs were the best model for learning language representations that could be fine tuned for a variety of downstream tasks
    - Nowadays, the best model is the transformer — stay tuned for that