



Recurrent Neural Networks

CS 759/859 Natural Language Processing Lecture 13

Samuel Carton, University of New Hampshire

Last lecture

Naïve bayes: application of Bayes Rule + unigram language modeling to classification

Bayes nets: General term for directed probabilistic graphical models (like Naïve Bayes, HMMs)

Hidden Markov models

- **Key idea: latent, hidden states** that are responsible for generating the text
 - HMMs most direct implementation of this idea
- Need fancy algorithms for learning and inference
- Not incredibly well-supported in Python
 - <https://hmmlearn.readthedocs.io/en/latest/>
- Largely superseded by RNNs these days
 - BUT... neurosymbolic and Bayesian neural networks are a hot research area:
https://www.cs.toronto.edu/~duvenaud/distill_bayes_net/public/



Word vectors & composition

Word vectors are pretty cool

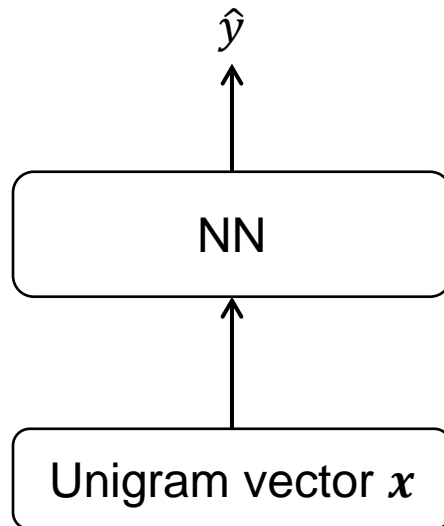
- Semantic similarity
- Analogies

But ultimately, NNs need **fixed-length** input, and it's not obvious how to **compose** a variable-length sequence of word vectors into a single **document vector**

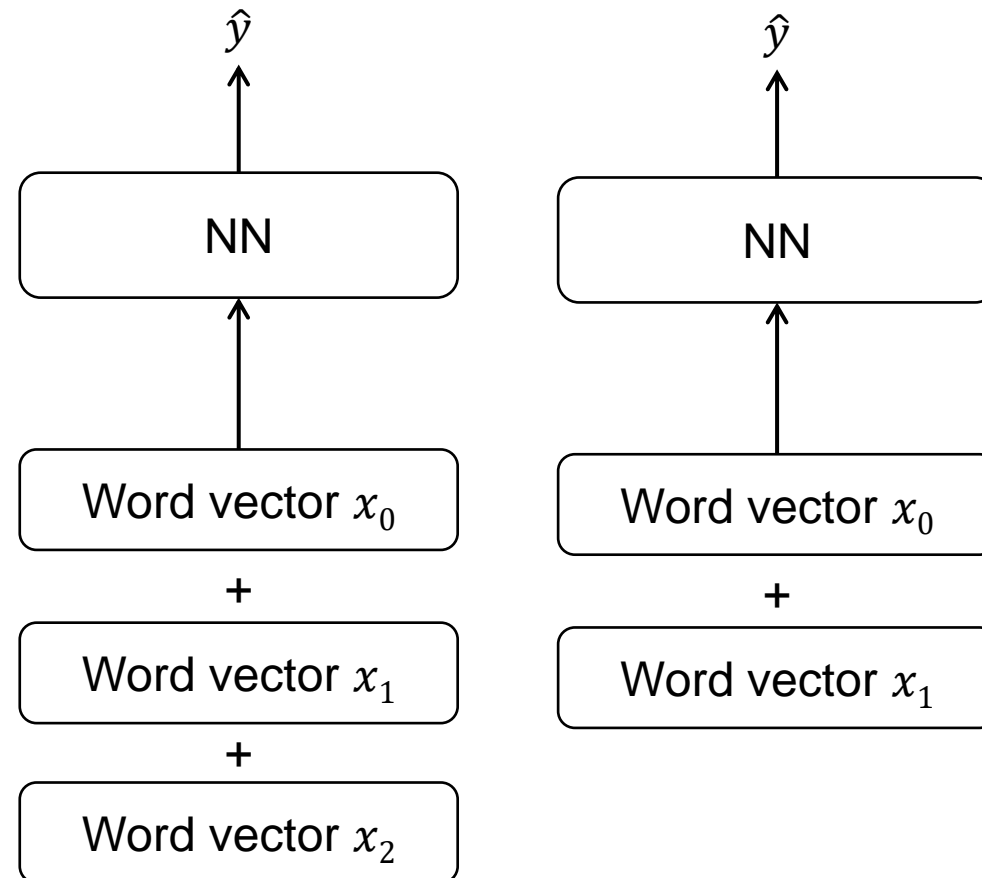
Just taking the centroid netted us some disappointing results

Neural text classification so far

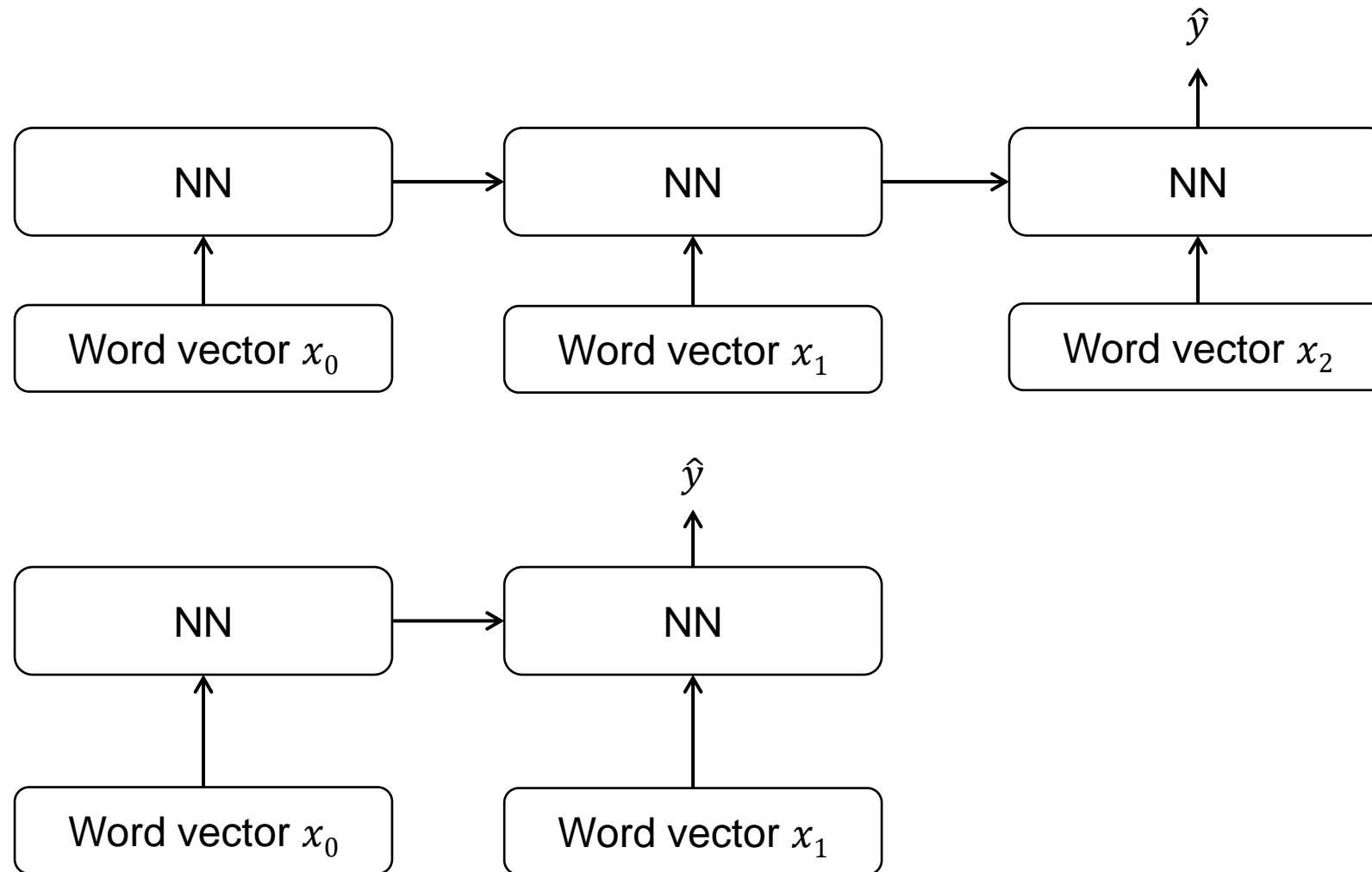
With unigram/TF-IDF vector



With word vector centroids



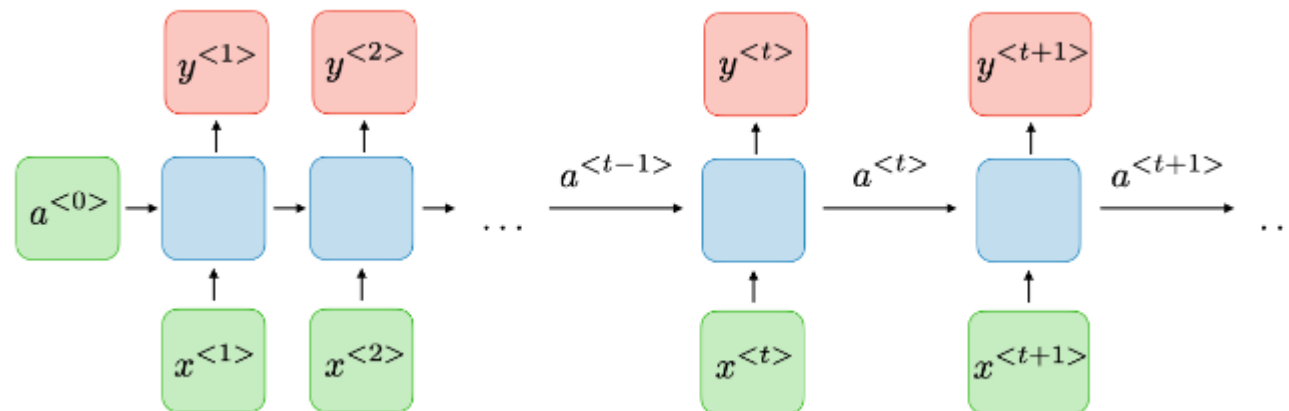
How to properly use word vectors?



Recurrent Neural Nets (RNNs)

Basic idea: the model runs over one word at a time, producing one or more intermediate **hidden state vectors** (aka activation vector) which it passes to itself when it looks at the next word.

Analogous to humans: read one word at a time and remember **whatever you need to remember** from word to word, to understand the meaning of the whole text.



Diagrams from <https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks>

- Very nice cheat-sheet for RNNs

Recurrent Neural Nets (RNNs)

More generally:

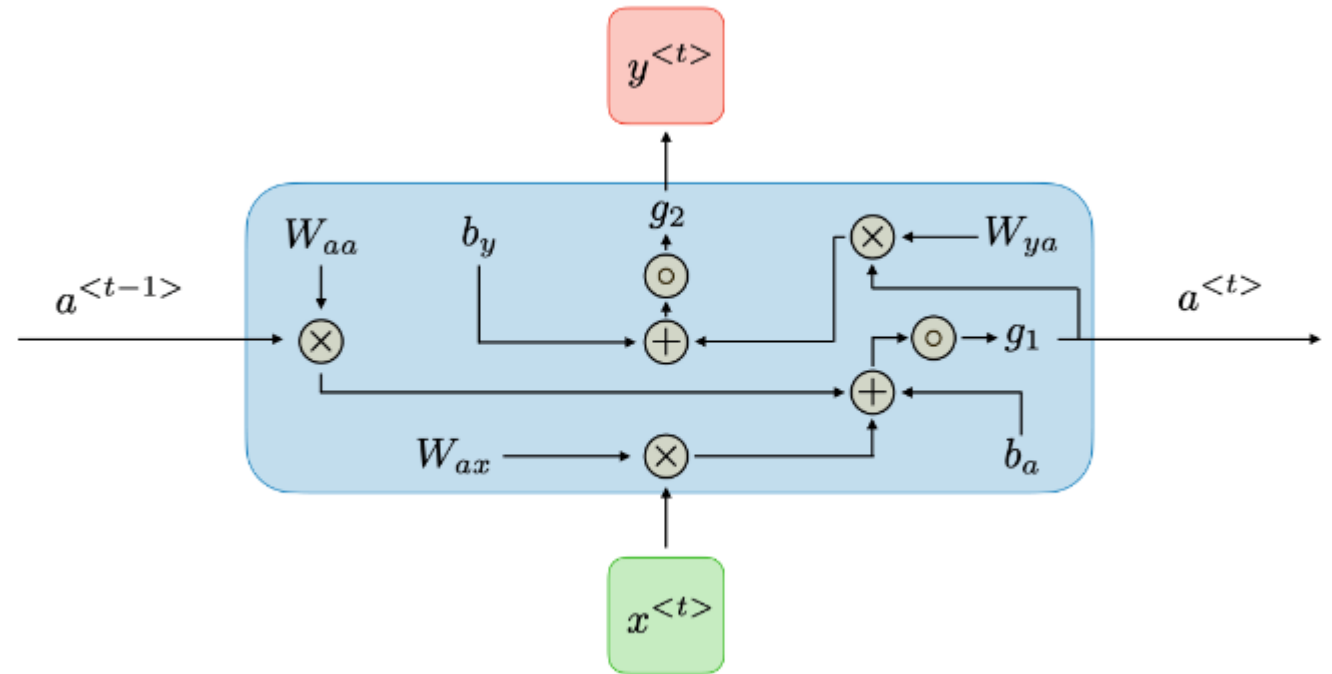
$$a^t = f(a^{t-1}, x^t)$$

$$\hat{y}^t = g(a^{t-1} \text{ or } a^t, x^t)$$

So a^t is what gets **remembered** from word to word, and \hat{y}^t is what gets **outputted** from word to word.

And models learn to remember what they need to remember, via objective functions on \hat{y}^t

$$a^{<t>} = g_1(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a) \quad \text{and} \quad y^{<t>} = g_2(W_{ya}a^{<t>} + b_y)$$



(a tad over-specific, IMHO)



Example: “dumb” insult detector

Say you are trying to train an RNN to read a whole text and predict “yes” if the text has the word “dumb” (or a synonym like “moronic”) in it, and “no” if not

Then, a^t can just be a 1 or a 0, indicating “has one of these words been found before?”

$$a^t = f(a^{t-1}, x^t)$$

$$\hat{y}^t = g(a^{t-1} \text{ or } a^t, x^t)$$

And $a^t = f(a^{t-1}, x^t)$ can be defined as ($a^{t-1} = 1$ or $x^t = \text{"dumb"}$)

And then finally \hat{y}^t could just be equal to a^t , and we would put an objective on just the final \hat{y}^t (\hat{y}^N), encouraging it to be 1 if there is a “dumb” somewhere in the text.

Challenge: how could we detect whether a given $x^t = \text{"dumb"}$ or some similar word?



Example: “dumb” insult detector

Say you are trying to train an RNN to read a whole text and predict “yes” if the text has the word “dumb” (or a synonym like “moronic”) in it, and “no” if not

Then, a^t can just be a 1 or a 0, indicating “has one of these words been found before?”

$$a^t = f(a^{t-1}, x^t)$$

$$\hat{y}^t = g(a^{t-1} \text{ or } a^t, x^t)$$

And $a^t = f(a^{t-1}, x^t)$ can be defined as ($a^{t-1} = 1$ or $x^t = \text{"dumb"}$)

And then finally \hat{y}^t could just be equal to a^t , and we would put an objective on just the final \hat{y}^t (\hat{y}^N), encouraging it to be 1 if there is a “dumb” somewhere in the text.

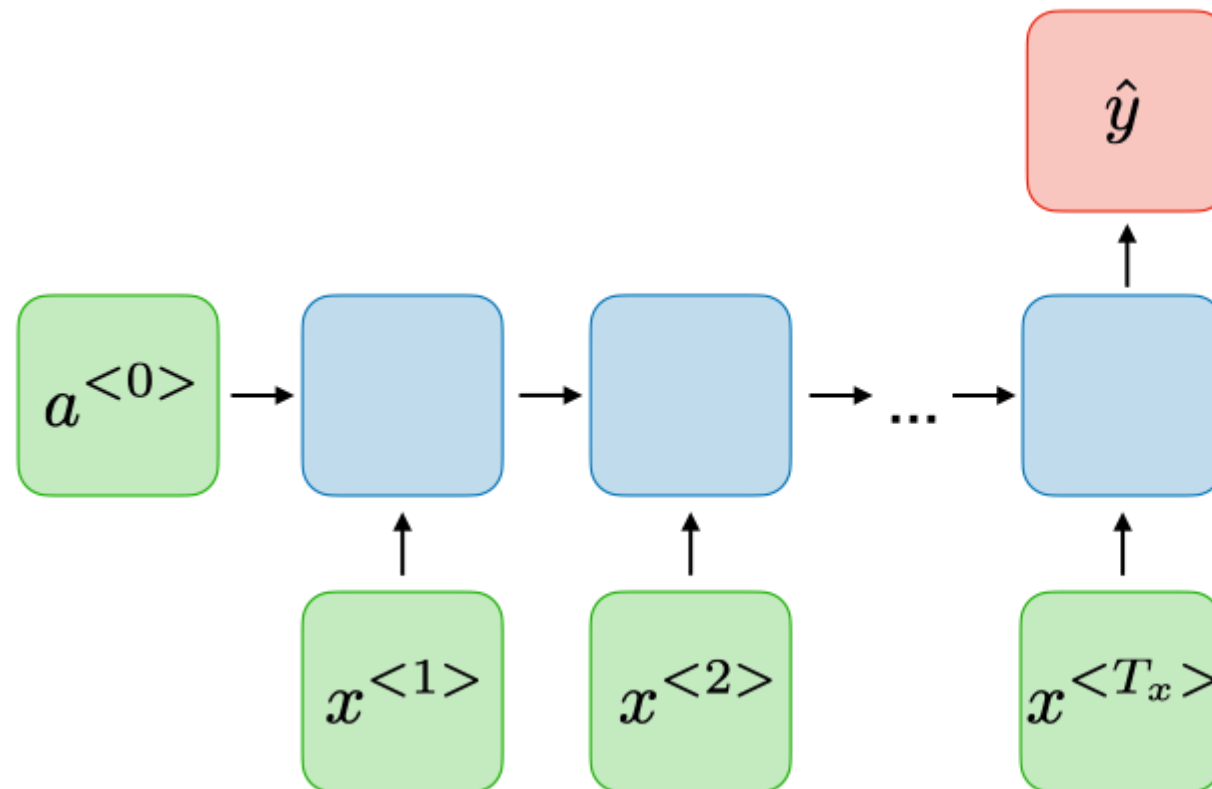
Challenge: how could we detect whether a given $x^t = \text{"dumb"}$ or some similar word?

- Word vectors!

Different RNN types

Many-to-one

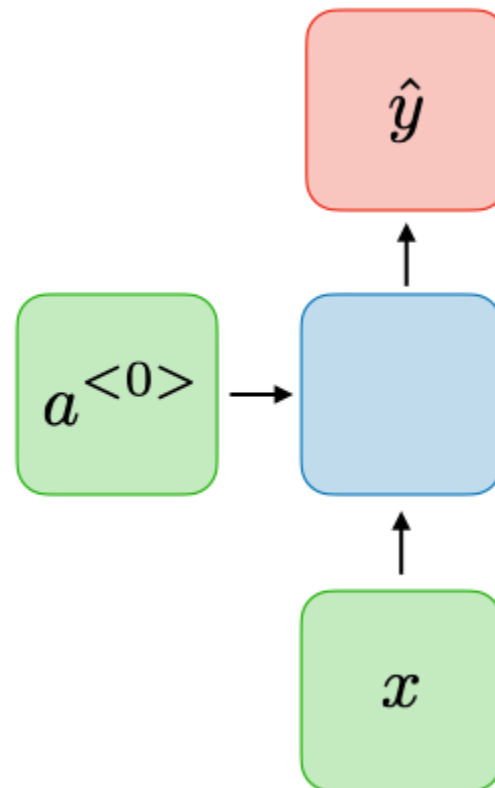
- Most text classification is this



Different RNN types

One-to-one

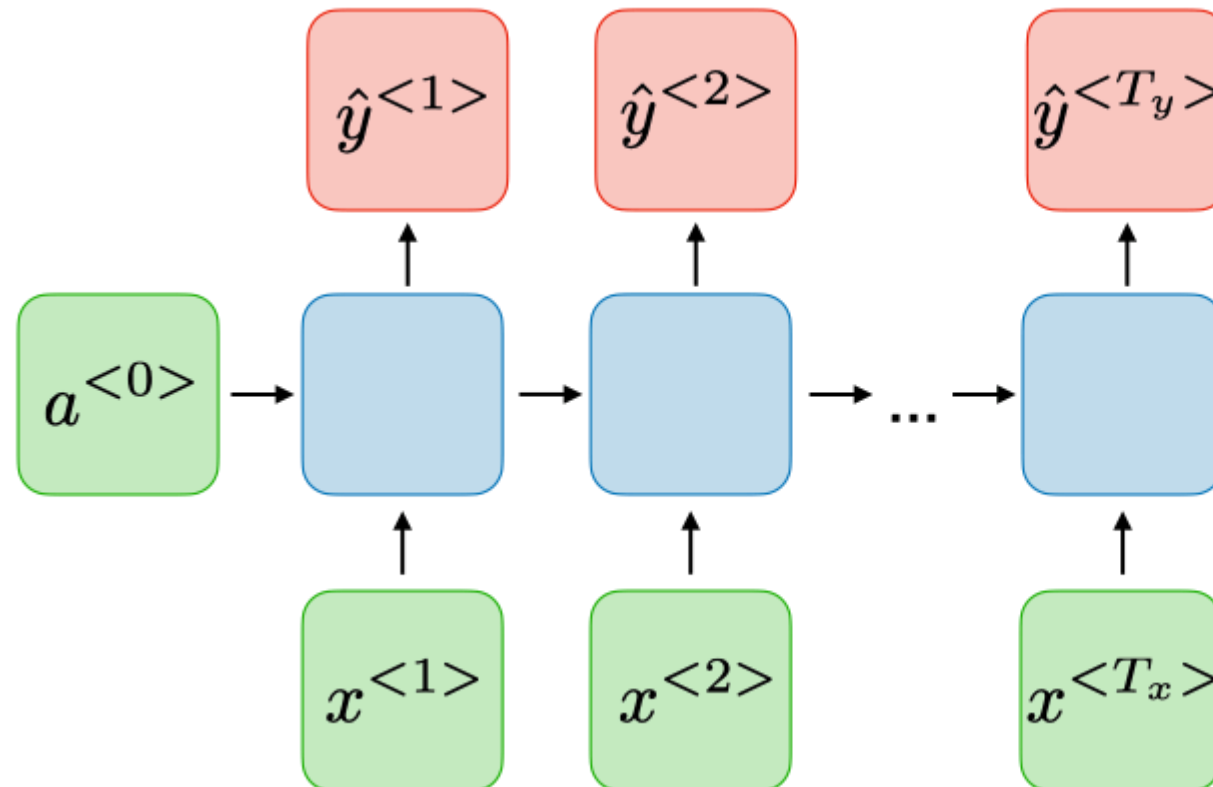
- A conventional (feedforward) neural net could be described as this



Different RNN types

Many-to-many

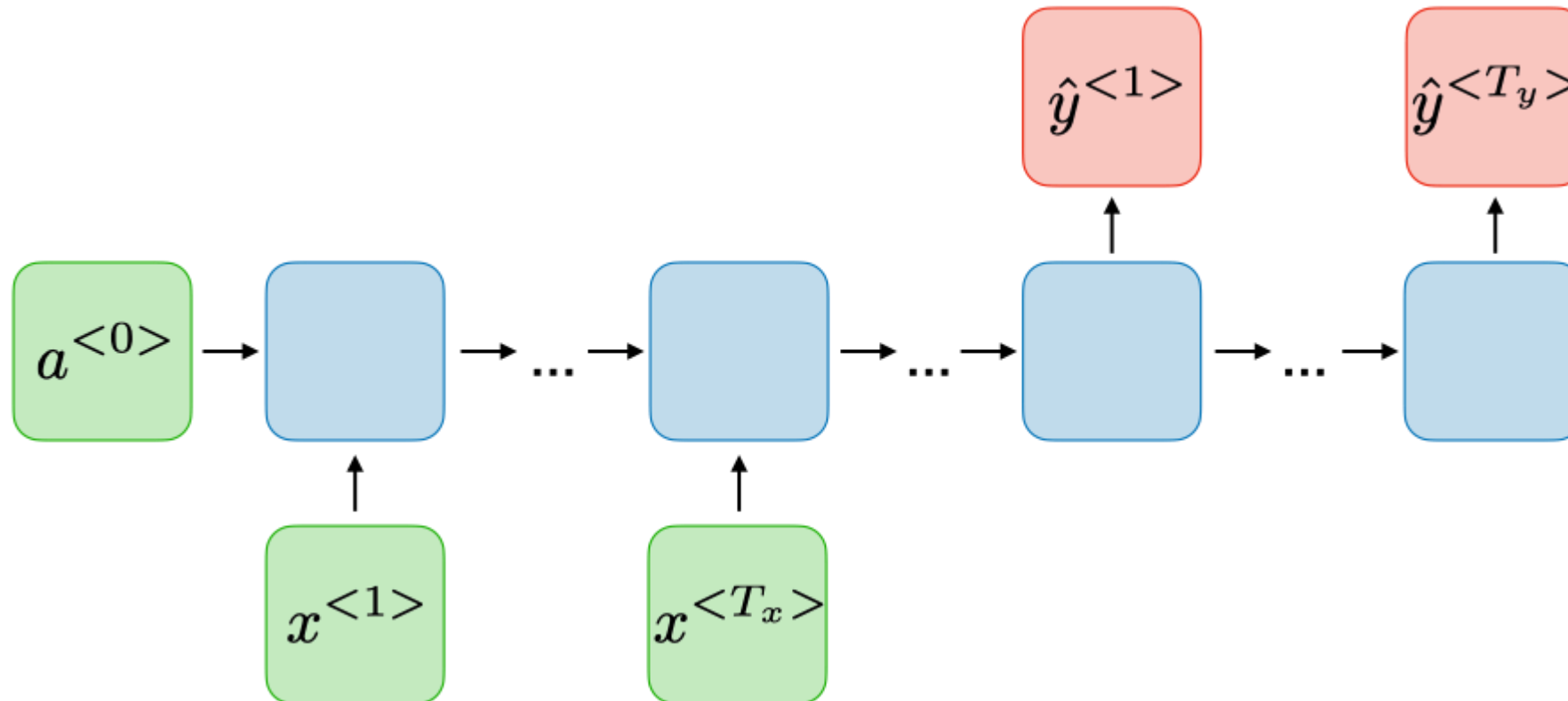
- POS tagging would be an example of this



Different RNN types

Many-to-many ($T_x \neq T_y$)

- Variant of many-to-many where there are inputs and outputs on different cells
- Machine translation is the main example of this



Vanishing gradients

RNNs are like a feedforward neural net being applied **horizontally** across each word of the text, rather than **vertically** across a flat representation of the text

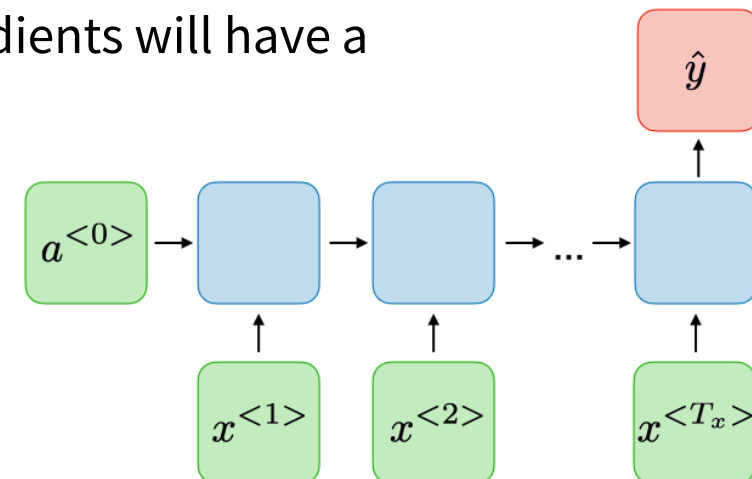
- Such as the centroid of the word vectors in the text, which is what we tried last lecture
- But same parameters at each layer, rather than different weight tensor

Like FFNNs, RNNs have problems with **vanishing gradients**

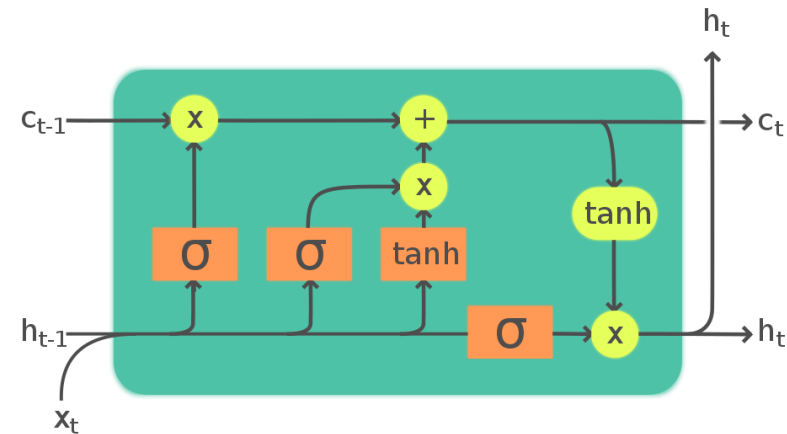
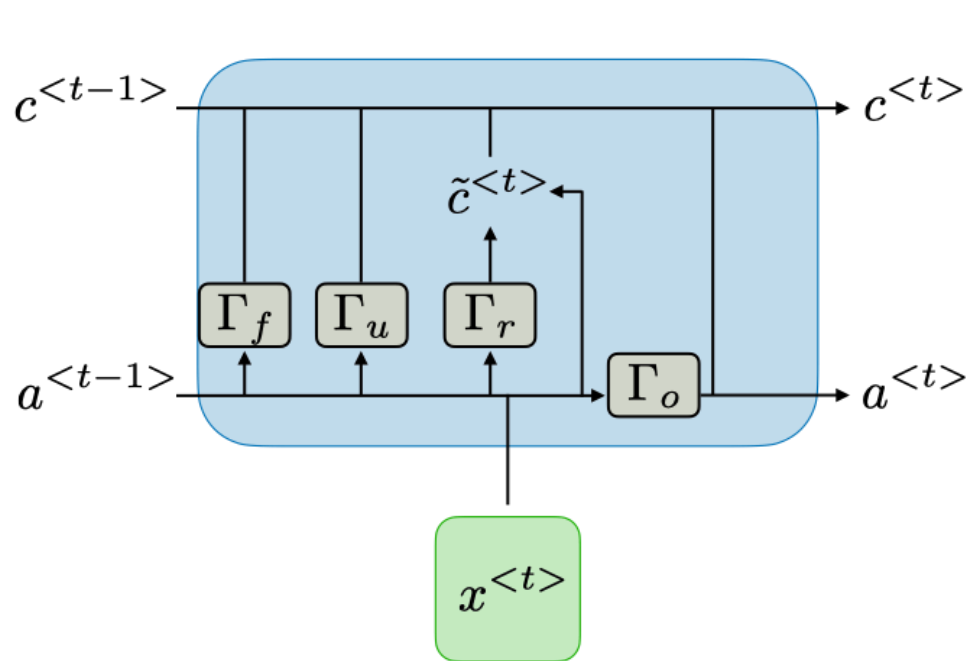
If you apply an objective only to \hat{y} at the end, the gradients will have a tough time training the cells toward the beginning

Called **catastrophic forgetting**





- Like losing focus on a sentence before you're done reading it



Long Short-Term Memory (LSTM)



Legend: Layer ComponentwiseCopy Concatenate

https://en.wikipedia.org/wiki/Long_short-term_memory

Long Short-Term Memory (LSTM)

$$f_t = \sigma_g(W_f x_t + U_f h_{t-1} + b_f)$$

$$i_t = \sigma_g(W_i x_t + U_i h_{t-1} + b_i)$$

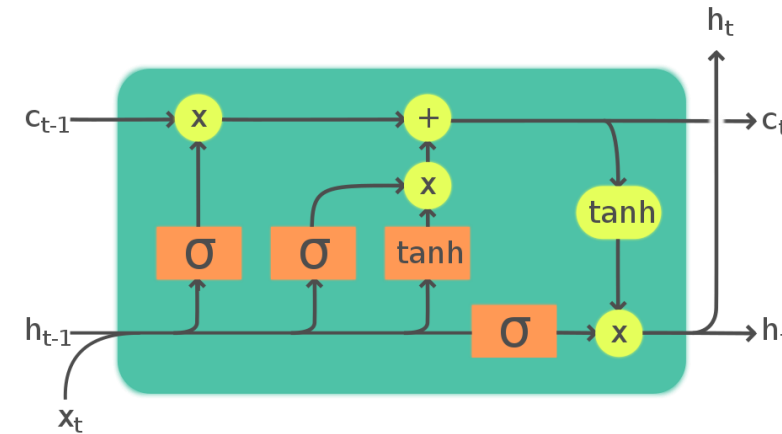
$$o_t = \sigma_g(W_o x_t + U_o h_{t-1} + b_o)$$

$$\tilde{c}_t = \sigma_c(W_c x_t + U_c h_{t-1} + b_c)$$

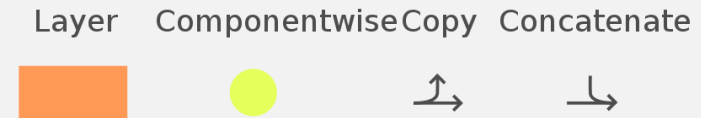
$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$$

$$h_t = o_t \odot \sigma_h(c_t)$$

- $x_t \in \mathbb{R}^d$: input vector to the LSTM unit
- $f_t \in (0, 1)^h$: forget gate's activation vector
- $i_t \in (0, 1)^h$: input/update gate's activation vector
- $o_t \in (0, 1)^h$: output gate's activation vector
- $h_t \in (-1, 1)^h$: hidden state vector also known as output vector of the LSTM unit
- $\tilde{c}_t \in (-1, 1)^h$: cell input activation vector
- $c_t \in \mathbb{R}^h$: cell state vector
- $W \in \mathbb{R}^{h \times d}$, $U \in \mathbb{R}^{h \times h}$ and $b \in \mathbb{R}^h$: weight matrices and bias vector parameters which need to be learned during training



Legend:





Long Short-Term Memory (LSTM)

Seem arbitrary? It kind of is.

Valaee et al. (2017) shows that different kinds of RNNs (GRUs, etc) have similar performance

- <https://arxiv.org/pdf/1801.01078.pdf>

So the exact internal equations aren't that important, more the idea of a persistent memory vector (or vectors) that can be added to or subtracted from based on new x^t 's, in a way that **can be learned** from the objective function.



Basic LSTM in Pytorch

Code description

- Reading and preprocessing SST-2 dataset
- Loading in GloVe vectors with Gensim
- Building a custom DataSet and collate function for a DataLoader
- Building, training and evaluating a basic LSTM RNN using Pytorch and Pytorch-lightning

Notebook headings

Loading GloVe vectors with Gensim

Reading and preprocessing SST-2 dataset

Dataset and DataLoader

Basic LSTM classification model

Model

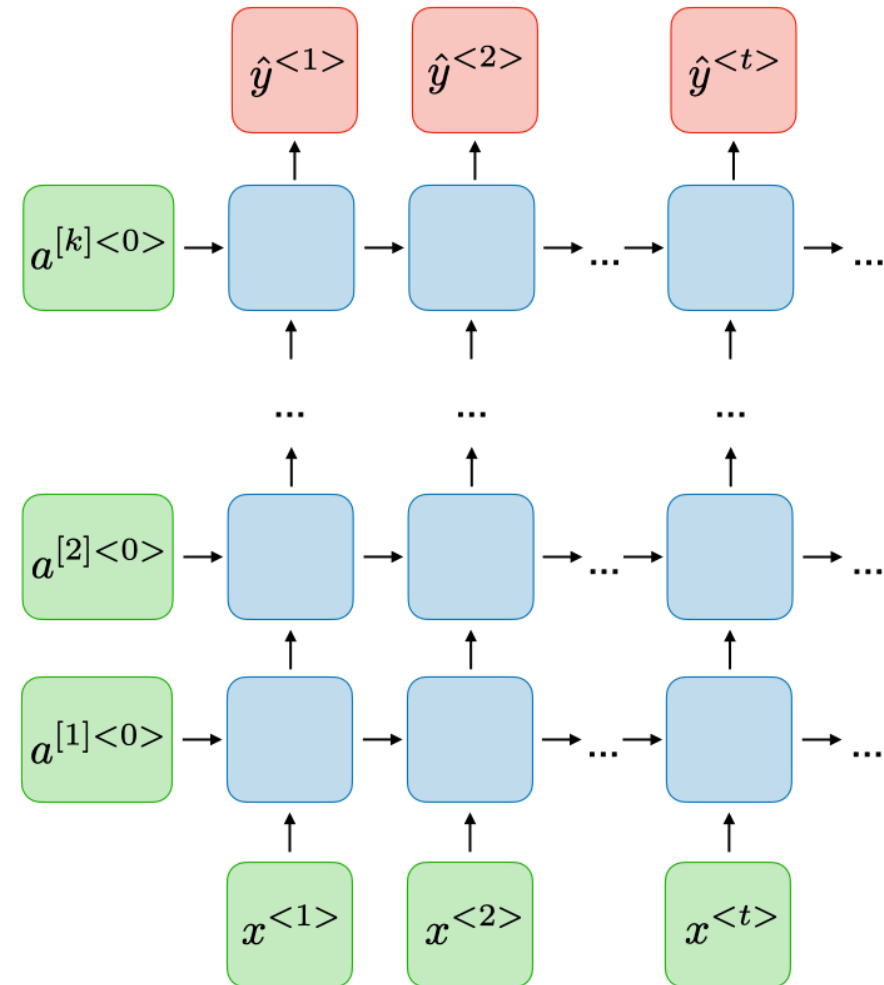
Training

Deep RNNs



Basic idea: Have multiple RNNs in a “stack”, with the bottom one running over the text, but the upper ones running over the output from the lower ones

Adds more learning capacity to the model, just like feedforward nets versus logistic regression

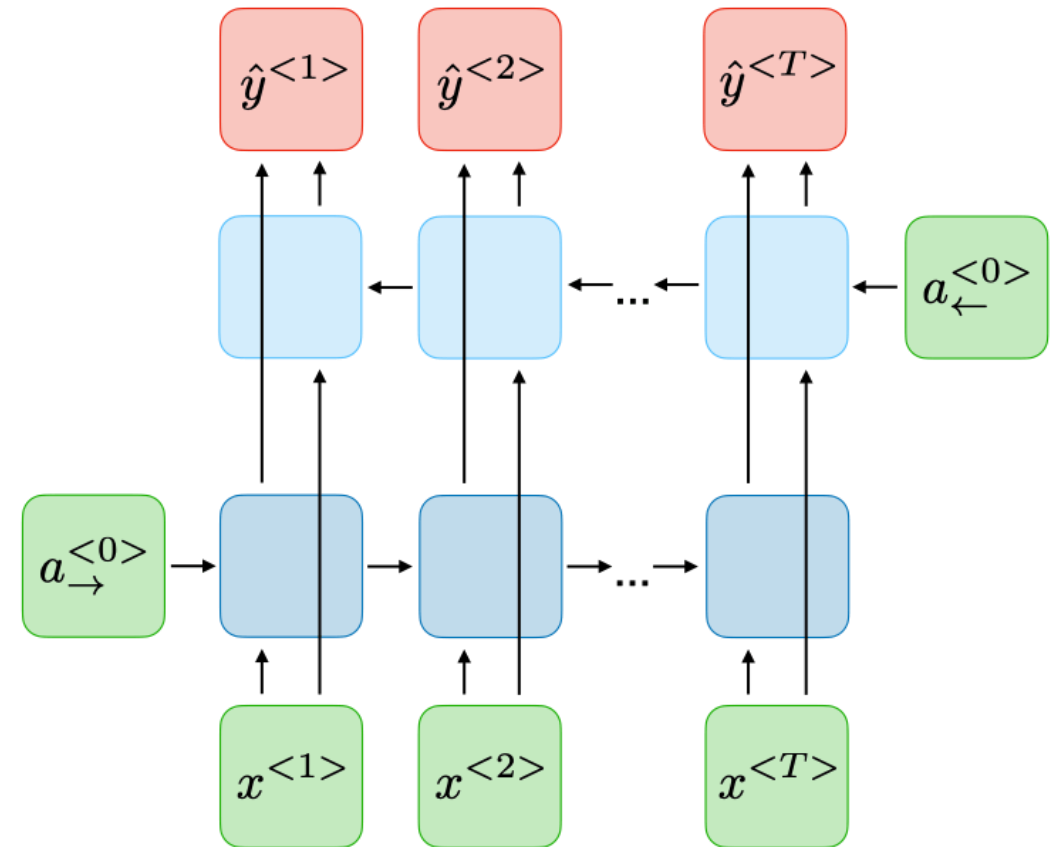


Bidirectional RNNs



Basic idea: run the model separately both forward and backward on the text, and then concatenate the final vectors from both passes

Fights catastrophic forgetting by having a gradient that gets applied at both the beginning and end of the text.



Dropout



Basic idea: with some percentage chance, randomly zero intermediate values within the model during training

Another form of regularization, like L1 or L2 regularization

Discourages overfitting by discouraging the model from relying too much on individual parameter values (which may be dropped).



Multilayer BiLSTM in Python

Code description

- Creating, training and evaluating a Multilayer BiLSTM using Pytorch and Pytorch-Lightning
 - Also using dropout during training

Notebook headings

Multilayer BiLSTM classification model

Model

Training

```

1 bilstm_trainer.fit(model=bilstm_model,
2 | | | | | train_dataloaders=train_dataloader,
3 | | | | | val_dataloaders=dev_dataloader)

```

INFO:pytorch_lightning.accelerators.cuda:LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]

INFO:pytorch_lightning.callbacks.model_summary:

	Name	Type	Params
0	word_embeddings	Embedding	40.0 M
1	lstm	LSTM	403 K
2	output_layer	Linear	402
3	train_accuracy	MulticlassAccuracy	0
4	val_accuracy	MulticlassAccuracy	0

403 K Trainable params

40.0 M Non-trainable params

40.4 M Total params

161.615 Total estimated model params size (MB)

Validation accuracy: tensor(0.5000, device='cuda:0')

Epoch 4: 100%  6911/6911 [01:12<00:00, 94.68it/s, loss=0.101, v_num=8]

Validation accuracy: tensor(0.8073, device='cuda:0')

Validation accuracy: tensor(0.8131, device='cuda:0')

Training accuracy: tensor(0.8396, device='cuda:0')

Validation accuracy: tensor(0.8177, device='cuda:0')

Validation accuracy: tensor(0.8291, device='cuda:0')

Training accuracy: tensor(0.9023, device='cuda:0')

Validation accuracy: tensor(0.8337, device='cuda:0')

Validation accuracy: tensor(0.8394, device='cuda:0')

Training accuracy: tensor(0.9331, device='cuda:0')

Validation accuracy: tensor(0.8486, device='cuda:0')

Validation accuracy: tensor(0.8417, device='cuda:0')

Training accuracy: tensor(0.9474, device='cuda:0')

Validation accuracy: tensor(0.8440, device='cuda:0')

Validation accuracy: tensor(0.8612, device='cuda:0')

INFO:pytorch_lightning.utilities.rank_zero:Trainer.fit` stopped: `max_epochs=5` reached.

Training accuracy: tensor(0.9569, device='cuda:0')

Yay

Concluding thoughts

RNNs

- One-to-one
- **Many-to-one**
- Many-to-many

LSTMS

Increasing RNN capacity

- Depth
- Bidirectionality

Dropout