

BST 261: Data Science II

Lecture 12

Word embeddings
Recurrent Neural Networks (RNNs), and LSTMs

Santiago Romero Brufau
Harvard T.H. Chan School of Public Health
Spring 2

Administrivia

- Last lab, Friday, May 5th
- May 1st, Transformers
- Guest lecture next Wednesday, May 3rd: AI Safety
- Guest lecture May 8th:



COGNITIVE PSYCHOLOGY



Research Article

Consequences of erudite vernacular utilized irrespective of necessity: problems with using long words needlessly

Daniel M. Oppenheimer

First published: 31 October 2005 | <https://doi.org/10.1002/acp.1178> | Citations: 200

Abstract

Most texts on writing style encourage authors to avoid overly-complex words. However, a majority of undergraduates admit to deliberately increasing the complexity of their vocabulary so as to give the impression of intelligence. This paper explores the extent to which this strategy is effective. Experiments 1–3 manipulate complexity of texts and find a negative relationship between complexity and judged intelligence. This relationship held regardless of the quality of the original essay, and irrespective of the participants' prior expectations of essay quality. The negative impact of complexity was mediated by processing fluency. Experiment 4 directly manipulated fluency and found that texts in hard to read fonts are judged to come from less intelligent authors. Experiment 5 investigated discounting of fluency. When obvious causes for low fluency exist that are not relevant to the judgement at hand, people reduce their reliance on fluency as a cue; in fact, in an effort not to be influenced by the irrelevant source of fluency, they over-compensate and are biased in the opposite direction. Implications and applications are discussed. Copyright © 2005 John Wiley & Sons, Ltd.



That's the magic of deep learning: turning meaning into vectors, then into geometric spaces, and then incrementally learning complex geometric transformations that map one space to another. All you need are spaces of sufficiently high dimensionality in order to capture the full scope of the relationships found in the original data.

The whole process hinges on a single core idea: *that meaning is derived from the pairwise relationship between things* (between words in a language, between pixels in an image, and so on) and that *these relationships can be captured by a distance function*. But note that





Bojan Tunguz
@tunguz

...



Should we tell him?



From your Digest



Nathan Kellert

X

Knows English · 1y

I have been working as a fake software developer by copying and pasting for 9 years. I want to learn programming properly and become a real developer. What should I do?



2:57 PM · Apr 5, 2022 · Twitter Web App

Processing text data (better)

- Word embeddings
- RNNs to LSTMs

Working with text data

A Sequence Modeling Problem: Predict the Next Word

“This morning I took my cat for a walk.”

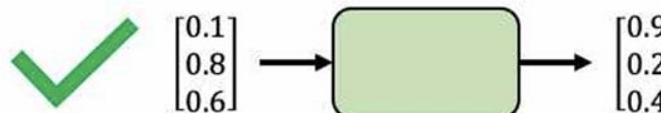
given these words

predict the
next word

Representing Language to a Neural Network



Neural networks cannot interpret words



Neural networks require numerical inputs

Text Data

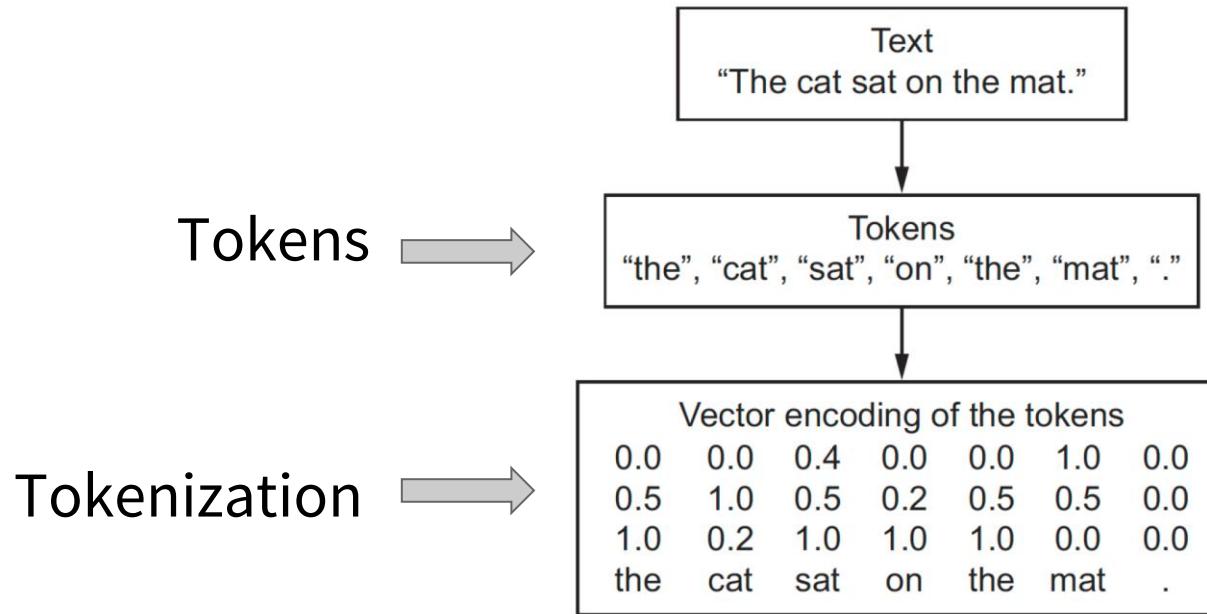
- ◎ Text data can be understood as either a sequence of characters or a sequence of words
 - Most common to work at the level of **words**
- ◎ Like all other neural networks, we can't simply input raw text - we must **vectorize the text**: transform it into numeric tensors
- ◎ We can do this in multiple ways:
 - Segment text into words, and transform each word into a vector
 - Segment text into characters and transform each character into a vector
 - Extract n-grams (overlapping groups of multiple consecutive words or characters) of words or characters, and transform each n-gram into a vector

Text Data

- ◎ The different units into which you break down text (words, characters, n-grams) are called **tokens**, and the action of breaking text into tokens is **tokenization**

- ◎ There are multiple ways to associate a vector with a token
 - One-hot encoding
 - Token embedding (or word embedding)

Tokenization



N-grams

- ◎ Word **n-grams** are groups of N (or fewer) consecutive words that you can extract from a sentence. The same concept may also be applied to characters instead of words.
- ◎ For example, the sentence "**Data science rocks my socks off!**" can be decomposed into a set of 3-grams:
 - {"**Data**", "**Data science**", "science", "science rocks", "**Data science rocks**", "rocks", "rocks my", "science rocks my", "my", "my socks", "socks", "rocks my socks", "off", "socks off", "my socks off"}

N-grams

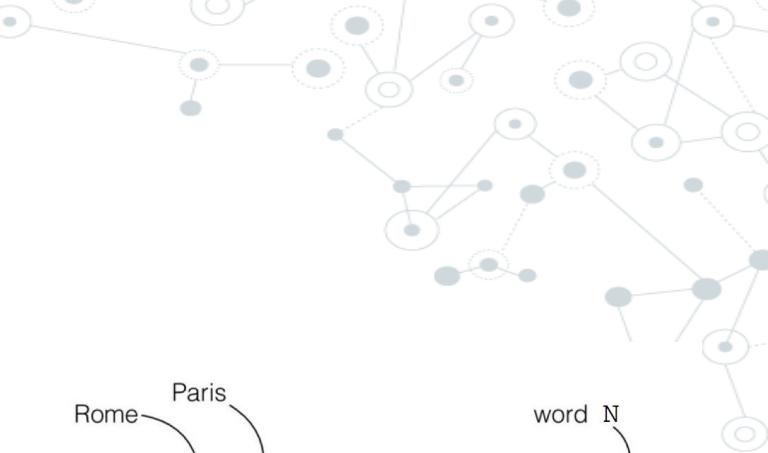
- ◎ This set is called a **bag of 3-grams**, which refers to the fact that it is a set of tokens, rather than a list or sequence: the tokens have no specific order
- ◎ This family of tokenization methods is called **bag-of-words**
- ◎ Order is not preserved, so the general structure of the sentence is lost
- ◎ Typically only used in shallow language-processing models

Extracting n-grams is a form of feature engineering that deep learning models do automatically in another way

One-hot Encoding

- ◎ Most common and most basic way to turn a token into a vector
- ◎ We used this with the IMDB data set

1. Associate a unique integer index with every word
 2. Then, turn the integer index i into a binary vector of size N (the size of the vocabulary, or number of words in the set)
- ◎ The vector is all 0s except for the i th entry, which is 1



Rome = [1, 0, 0, 0, 0, 0, ..., 0]
Paris = [0, 1, 0, 0, 0, 0, ..., 0]
Italy = [0, 0, 1, 0, 0, 0, ..., 0]
France = [0, 0, 0, 1, 0, 0, ..., 0]

word N



One-hot Hashing

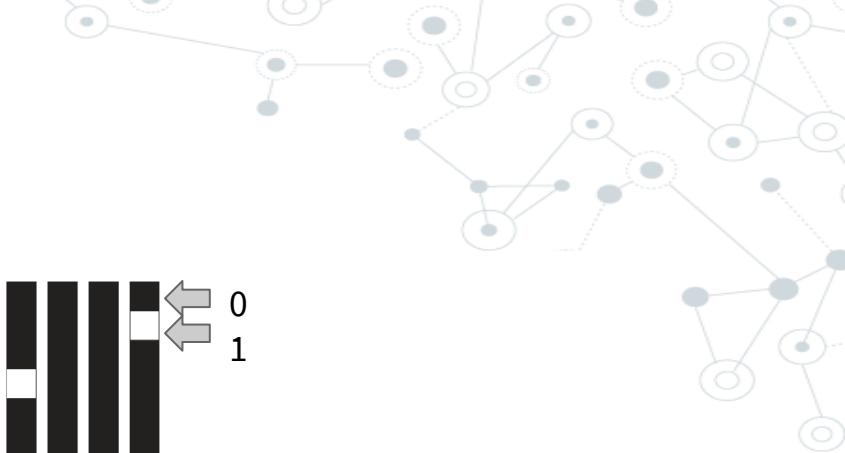
- ◎ A variant of one-hot encoding is the **one-hot hashing** trick
- ◎ Useful when the number of unique tokens is too large to handle explicitly
- ◎ Instead of explicitly assigning an index to each word and keeping a reference of these indices in a dictionary, you can hash words into vectors of fixed size

One-hot Hashing

- ◎ Main advantage: saves memory and allows generation of tokens before all of the data has been seen
- ◎ Main drawback: **hash collisions**
 - Two different words end up with the same hash
 - The likelihood of this decreases when the dimensionality of the hashing space is much larger than the total number of unique tokens being hashed

Word Embeddings

- Another common and powerful way to associate a vector with a word is the use of **dense word vectors** or **word embeddings**
- Word embeddings are dense, low-dimensional floating-point vectors
- Are learned from the data rather than hard coded
- 256, 512 and 1024-dimensional word embeddings are common



One-hot word vectors:

- Sparse
- High-dimensional
- Hardcoded

Word embeddings:

- Dense
- Lower-dimensional
- Learned from data

Word Embeddings

There are 2 ways to obtain word embeddings:

1. Learn word embeddings jointly with the main task you care about

Start with random word vectors and then learn word vectors in the same way you learn the weights of the network
2. Use pre-trained word embeddings

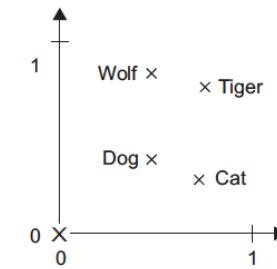
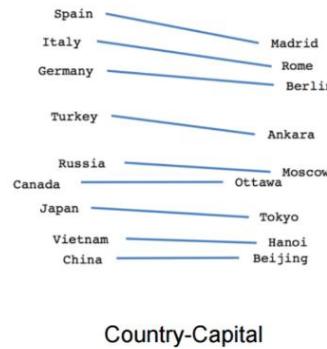
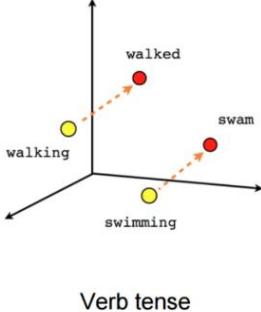
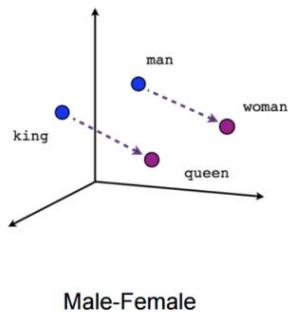
Load into your model word embeddings that were precomputed using a different machine-learning task than the one you're trying to solve

Learning Word Embeddings

- ◎ It's easy to simply associate a vector with a word randomly - but this results in an embedding space without structure, and things like synonyms that could be interchangeable will have completely different embeddings
- ◎ This makes it difficult for a deep neural network to make sense of these representations

Learning Word Embeddings

- It is better for similar words to have similar embeddings, and dissimilar words to have dissimilar embeddings
- We can, for example, relate the L2 distance to the similarity of the words with a smaller distance meaning the words are similar and bigger distances indicating very different words



Word Embeddings

- ◎ Common examples of useful geometric transformations are “gender” and “plural” vectors:
 - Adding a “female” vector to the vector “king” will result in the vector “queen”
 - Adding the “plural” vector to the vector “elephant” will result in the vector “elephants”

- ◎ Is there a word-embedding space that would perfectly map human language and be used in any natural-language processing task?
 - Maybe, but we haven’t discovered it yet
 - Very complicated - many different languages that are not isomorphic due to specific cultures and contextsA “good” word-embedding space depends on the task

Pre-trained Word Embeddings

- ◎ Similar to using pre-trained convolutional bases, we can use pre-trained word embeddings
 - ◎ Particularly useful when your sample size is small
 - ◎ Load embedding vectors from a precomputed embedding space that is highly structured with useful properties
 - Captures generic aspects of language structure
 - ◎ These embeddings are typically computed using **word-occurrence statistics**:
 - Observations about what words co-occur in sentences or documents
 - ◎ Various word-embedding methods exist:
 - **Word2vec** algorithm (developed by Tomas Mikolov at Google in 2013)
 - **GloVe**: Global Vectors for Word Representation (developed by researchers at Stanford in 2014)
- Both embeddings can be used in Keras

Word2vec

- ◎ Mikolov et al. introduce the word2vec algorithm which is actually a collection of different models
 - Continuous bag of words (CBOW)
 - Skip-gram with negative sample (SGNS)
 - Key insight: simple linear model trained on tons of data works much better than fancy nonlinear model that was difficult to train

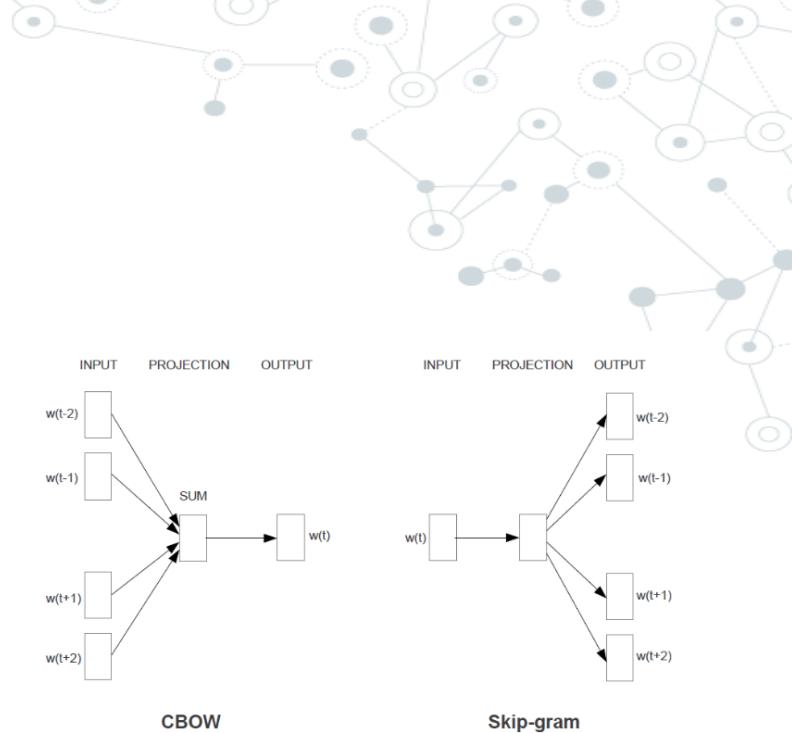


Figure 1: New model architectures. The CBOW architecture predicts the current word based on the context, and the Skip-gram predicts surrounding words given the current word.

GloVe

- ◎ GloVe: Global vectors for word representation
- ◎ Developed by researchers at Stanford in 2014
- ◎ Open-source project at Stanford
- ◎ Has similarities to other word embedding methods
 - Word2vec is a “predictive” model whereas GloVe is a “count-based” model

<https://nlp.stanford.edu/projects/glove/>

Highlights

1. Nearest neighbors

The Euclidean distance (or cosine similarity) between two word vectors provides an effective method for measuring the linguistic or semantic similarity of the corresponding words. Sometimes, the nearest neighbors according to this metric reveal rare but relevant words that lie outside an average human's vocabulary. For example, here are the closest words to the target word *frog*:

- o *frog*
- 1. *frogs*
- 2. *toad*
- 3. *litoria*
- 4. *leptodactylidae*
- 5. *rana*
- 6. *lizard*
- 7. *eleutherodactylus*



3. *litoria*

4. *leptodactylidae*

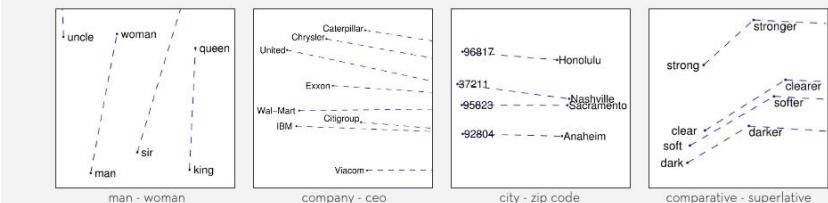
5. *rana*

7. *eleutherodactylus*

2. Linear substructures

The similarity metrics used for nearest neighbor evaluations produce a single scalar that quantifies the relatedness of two words. This simplicity can be problematic since two given words almost always exhibit more intricate relationships than can be captured by a single number. For example, *man* may be regarded as similar to *woman* in that both words describe human beings; on the other hand, the two words are often considered opposites since they highlight a primary axis along which humans differ from one another.

In order to capture in a quantitative way the nuance necessary to distinguish *man* from *woman*, it is necessary for a model to associate more than a single number to the word pair. A natural and simple candidate for an enlarged set of discriminative numbers is the vector difference between the two word vectors. GloVe is designed in order that such vector differences capture as much as possible the meaning specified by the juxtaposition of two words.





Arindam Paul, Ph.D.

Researcher in Machine Learning, Northwestern University · Author has 385 answers and 961.4K answer views · 3y

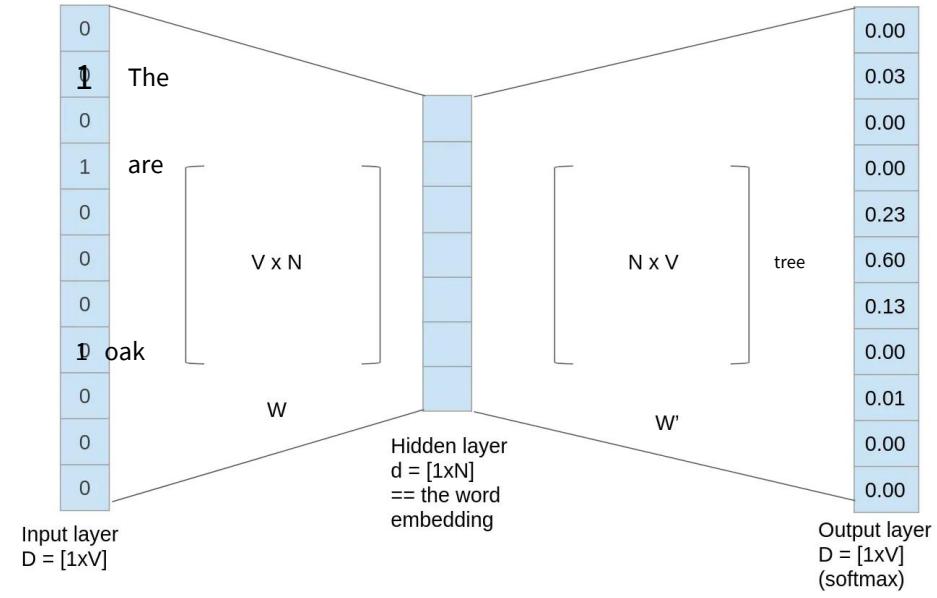
X



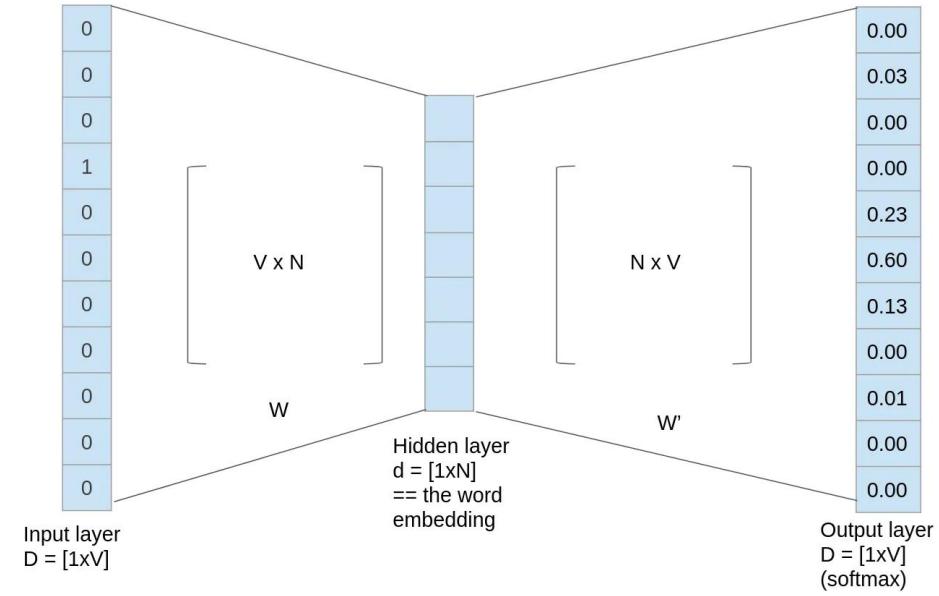
Differences:

1. Presence of Neural Networks: GloVe does not use neural networks while word2vec does. In GloVe, the loss function is the difference between the product of word embeddings and the log of the probability of co-occurrence. We try to reduce that and use SGD but solve it as we would solve a linear regression. While in the case of word2vec, we either train the word on its context (skip-gram) or train the context on the word (continuous bag of words) using a 1-hidden layer neural network.
2. Global information: word2vec does not have any explicit global information embedded in it by default. GloVe creates a global co-occurrence matrix by estimating the probability a given word will co-occur with other words. This presence of global information makes GloVe ideally work better. Although in a practical sense, they work almost similar and people have found similar performance with both.

How to train embeddings



How to train embeddings



*2vec

Distributed Representations of Sentences and Documents

Quoc Le
Tomas Mikolov

Google Inc, 1600 Amphitheatre Parkway, Mountain V

Abstract

Many machine learning algorithms require the input to be represented as a fixed-length feature vector. When it comes to texts, one of the most common fixed-length features is bag-of-words. Despite their popularity, bag-of-words features have two major weaknesses: they lose the ordering of the words and they also ignore semantics of the words. For example, “powerful,” “strong” and “Paris” are equally distant. In this paper, we

Ski

EMBED ALL THE THINGS

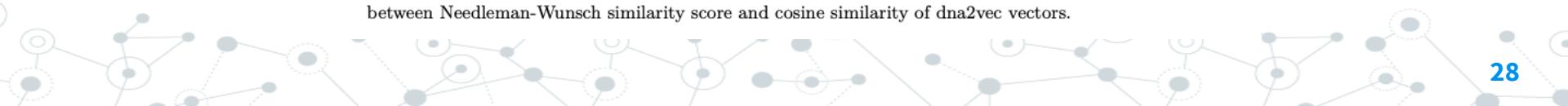
dna2vec:

itions of



memegenerator.net One of the ubiquitous k-mer components.

Unfortunately, the straightforward vector encoding of k-mer as a one-hot vector is vulnerable to the curse of dimensionality. Worse yet, the distance between any pair of one-hot vectors is equidistant. This is particularly problematic when applying the latest machine learning algorithms to solve problems in biological sequence analysis. In this paper, we propose a novel method to train distributed representations of variable-length k-mers. Our method is based on the popular word embedding model *word2vec*, which is trained on a shallow two-layer neural network. Our experiments provide evidence that the summing of dna2vec vectors is akin to nucleotides concatenation. We also demonstrate that there is correlation between Needleman-Wunsch similarity score and cosine similarity of dna2vec vectors.



cui2vec: embeddings for medical concepts

Clinical Concept Embeddings Learned from Massive Sources of Multimodal Medical Data

Andrew L. Beam
Harvard Medical School

Benjamin Kompa
University of North Carolina

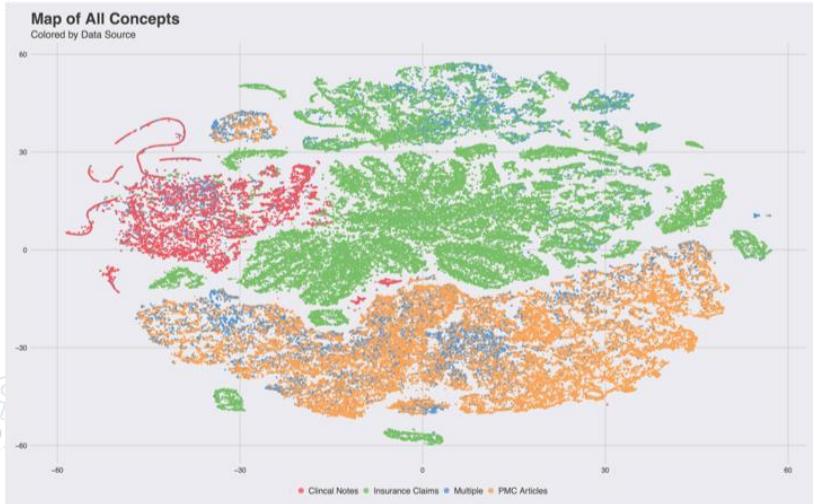
Inbar Fried
University of North Carolina

Nathan Palmer
Harvard Medical School

Xu Shi
Harvard School of Public Health

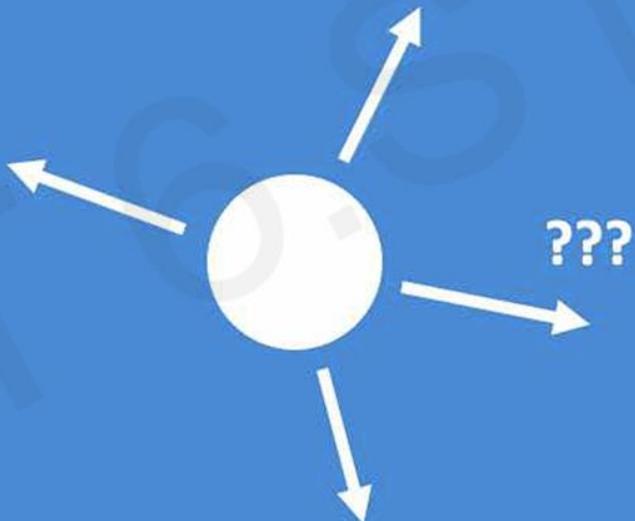
Tianxi Cai
Harvard School of Public Health

Isaac S. Kohane
Harvard Medical School



Recurrent Neural Networks (RNNs)

Given an image of a ball,
can you predict where it will go next?



Given an image of a ball,
can you predict where it will go next?



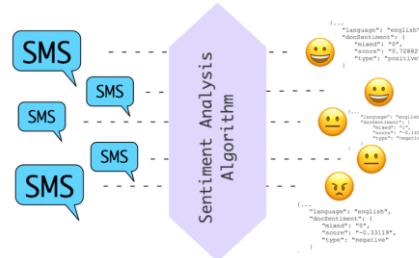
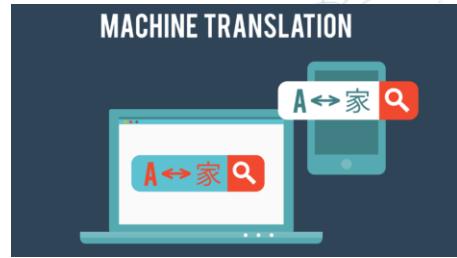
Neural Networks

- ◎ So far we have seen:
 - Deep feedforward networks (MLPs)
 - Map a fixed length **vector** to a fixed length **scalar/vector**
 - Use case: classical machine learning
 - CNNs
 - Map a fixed length **matrix/tensor** to a fixed length **scalar/vector**
 - Use case: image recognition
- ◎ RNNs
 - Map a **sequence of matrices/tensors** to a **scalar/vector**
 - Map a **sequence** to a **sequence**
 - Use case: natural language processing (NLP)

NLP

- ◎ The challenge of language for computers:
 - Computers are built to process numbers
 - Language isn't easily represented by numbers
 - How can we represent human language in a computable fashion?
 - Applications: machine translation, text classification, information retrieval, sentiment analysis and many more

You already saw one example: classifying IMDb movie reviews as either positive or negative



MLPs



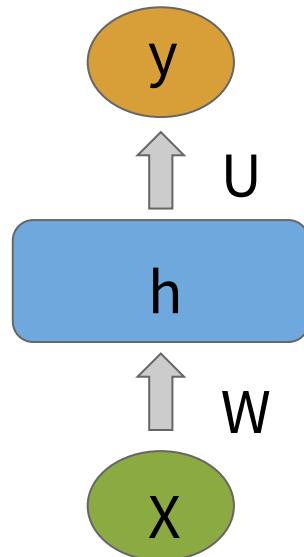
RNNs

- RNNs are a natural extension of MLPs
- MLPs are “memoryless”, but often we need knowledge of the past sequence of events to predict the future

	Inputs	Output	Probability
MLP	X	y	$P(y X)$
RNN	$[x_1, x_2, x_3, \dots, x_t]$	y	$P(y x_1, x_2, x_3, \dots, x_t)$

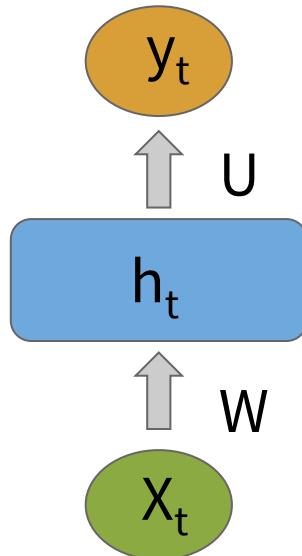
MLPs \rightarrow RNNs

- Recall that the first hidden layer for an MLP is given by $h = f(XW + b)$ where $f()$ is the activation function and W is the weight matrix in the hidden layer, b is the bias term, and U is the weight matrix in the output layer



MLPs \rightarrow RNNs

- ◎ RNNs add the concept of “state” to traditional neural networks
- ◎ To incorporate the notion of time we will index the hidden layer with t and feed it X_t :
$$h_t = f(X_t W + b)$$



MLPs \rightarrow RNNs

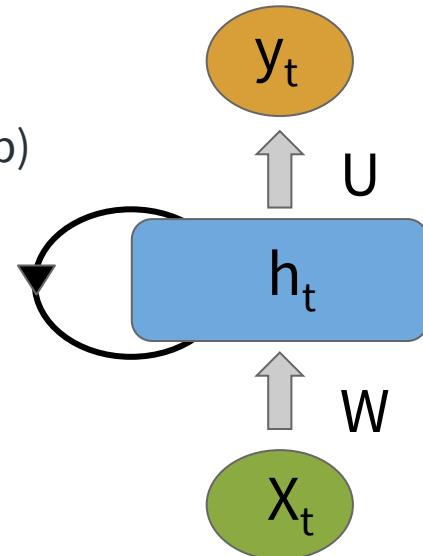
- To incorporate information from the previous state we will make the following modification:

$$h_t = f(X_t W + b) \longrightarrow$$

$$h_t = f(X_t W + h_{t-1} U + b)$$

Input at
time t

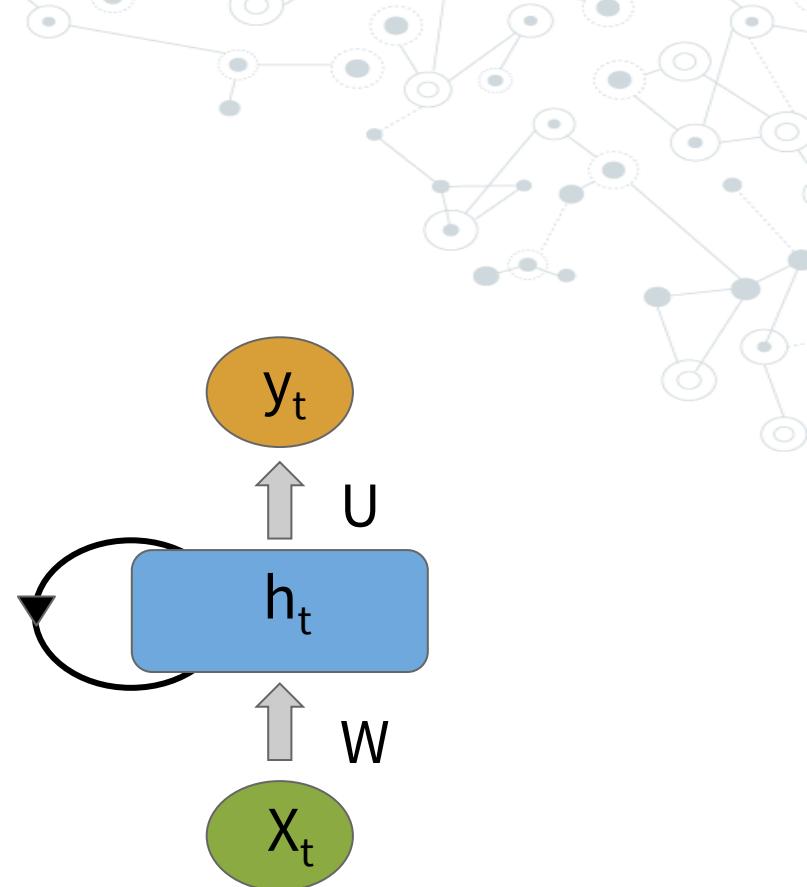
Hidden state
from previous
time point



- This is equivalent to connecting the hidden state to itself

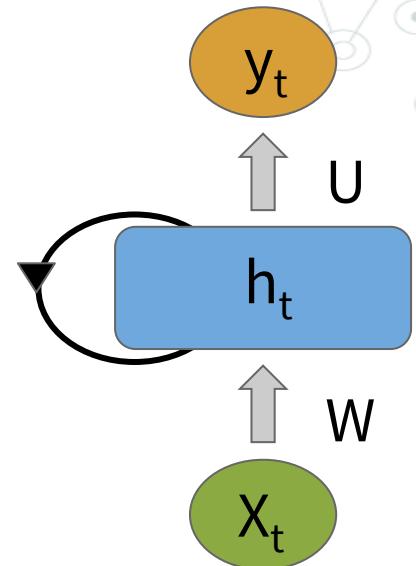
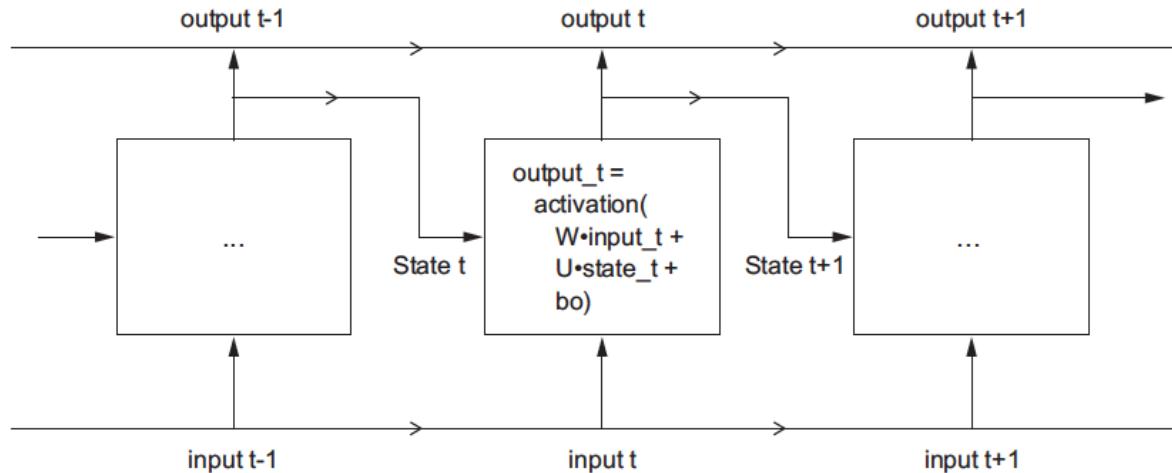
RNN Backprop

- ◎ How do we backprop through something with a loop?
- ◎ Have to backprop through depth and time
- ◎ This is similar to what we saw with MLPs, but we aren't going to go through it here

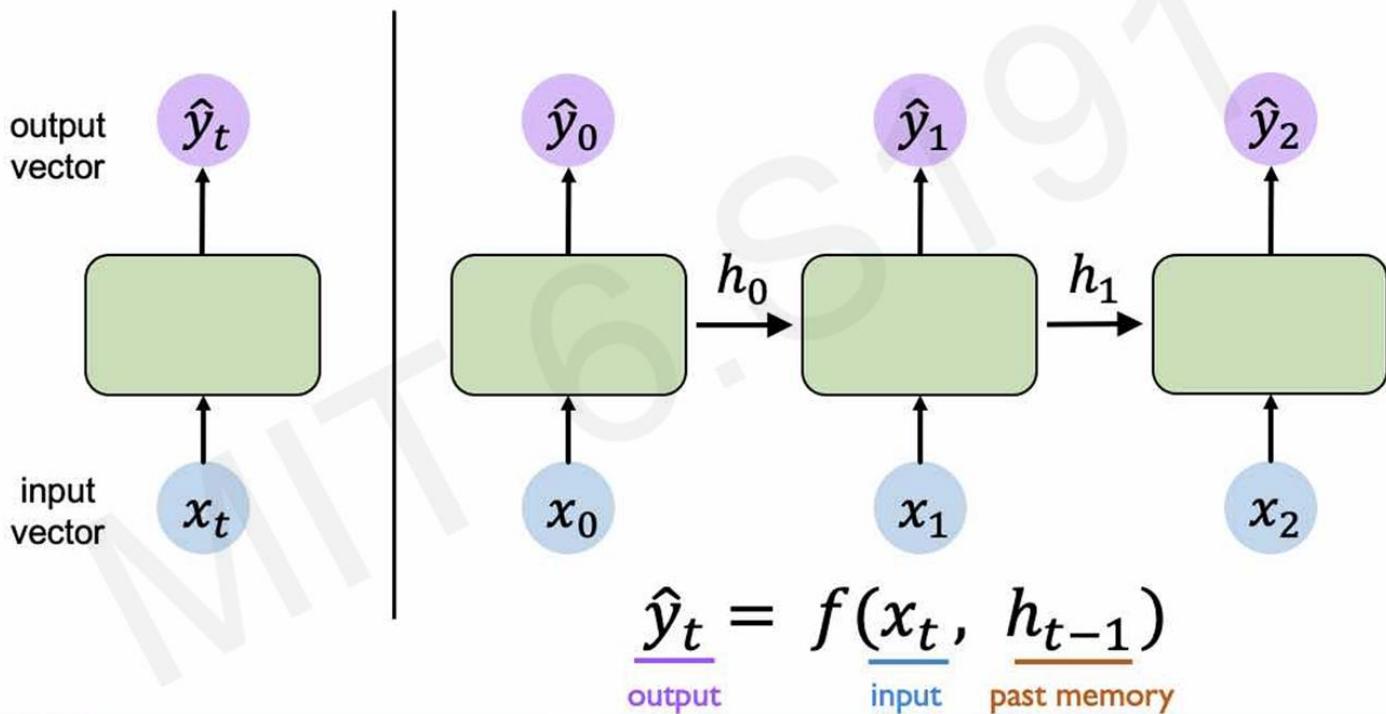


RNNs

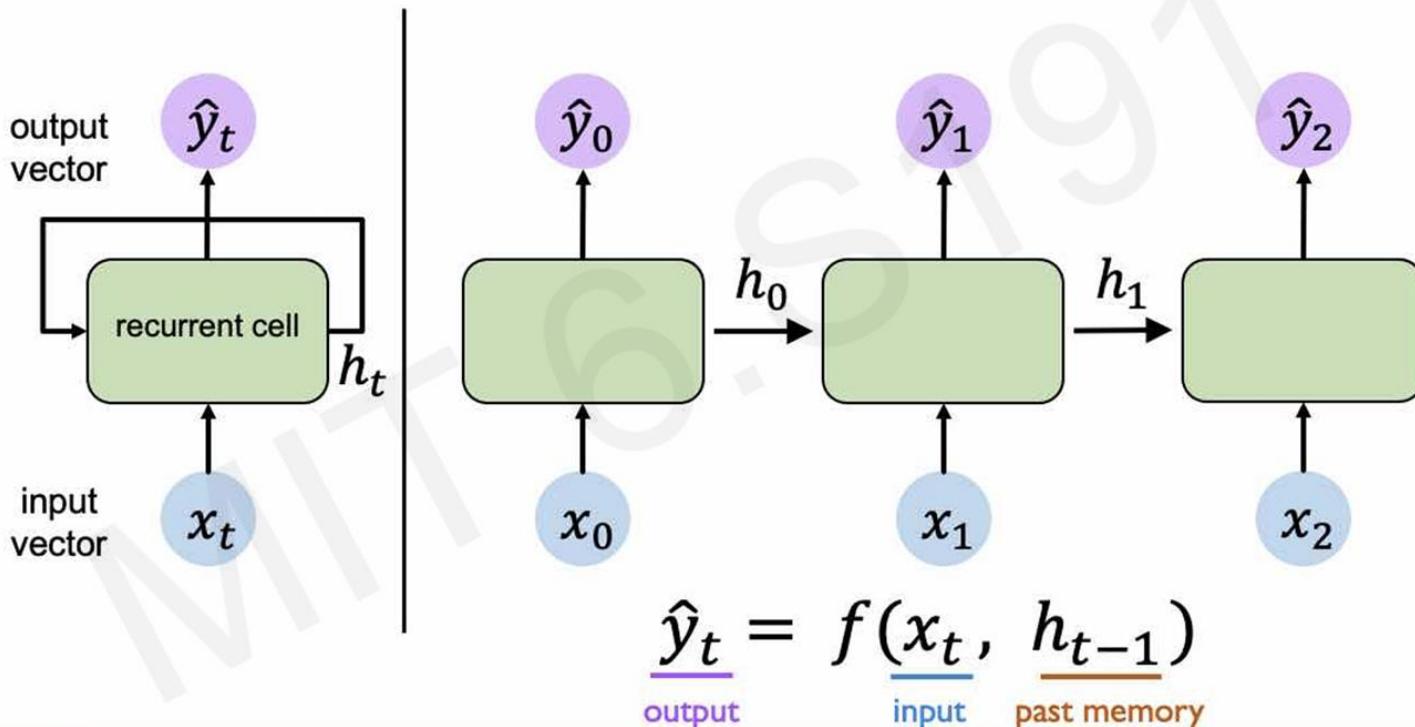
“Unrolled” RNN



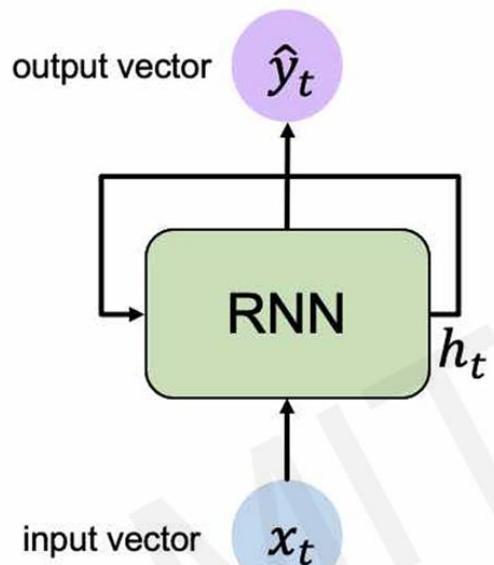
Neurons with Recurrence



Neurons with Recurrence



RNN State Update and Output



Output Vector

$$\hat{y}_t = \mathbf{W}_{hy}^T h_t$$

Update Hidden State

$$h_t = \tanh(\mathbf{W}_{hh}^T h_{t-1} + \mathbf{W}_{xh}^T x_t)$$

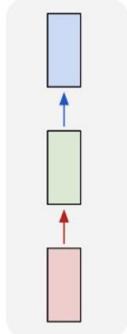
Input Vector

$$x_t$$

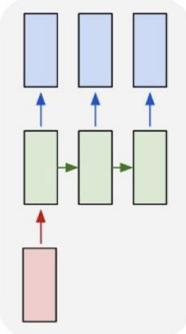
RNNs

- ◎ There are many ways to configure the input \Rightarrow output mapping

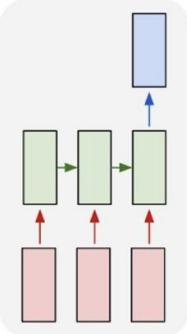
one to one



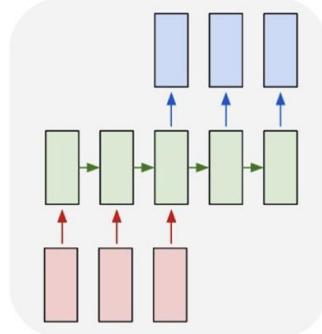
one to many



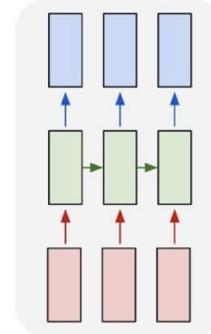
many to one



many to many

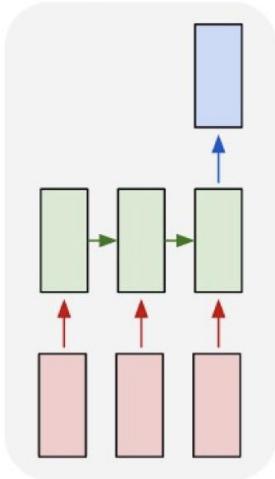


many to many

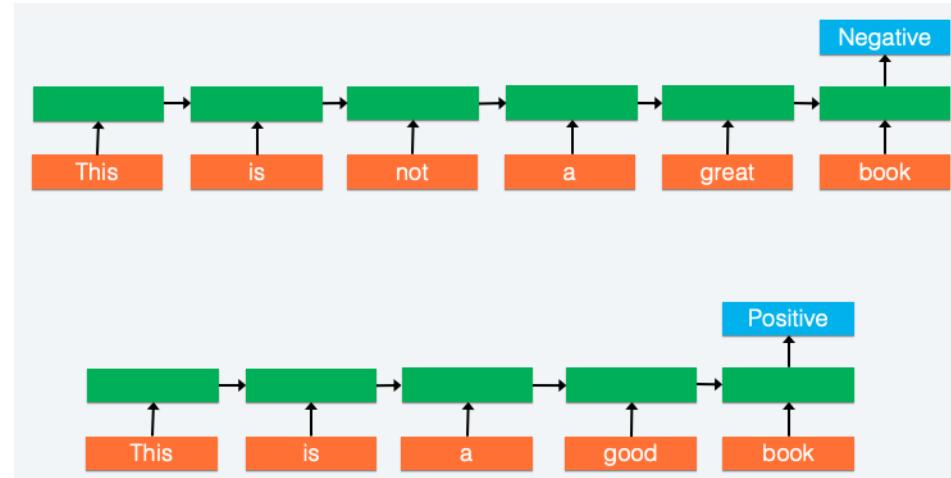
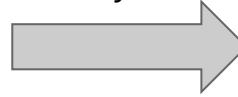


RNNs

many to one

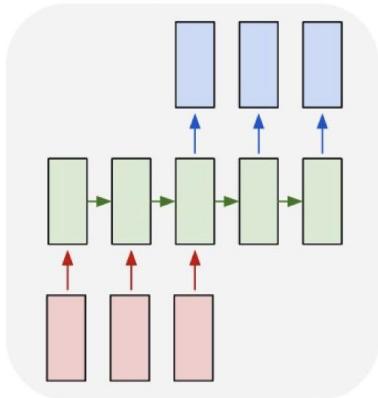


Ex:
Sentiment
Analysis

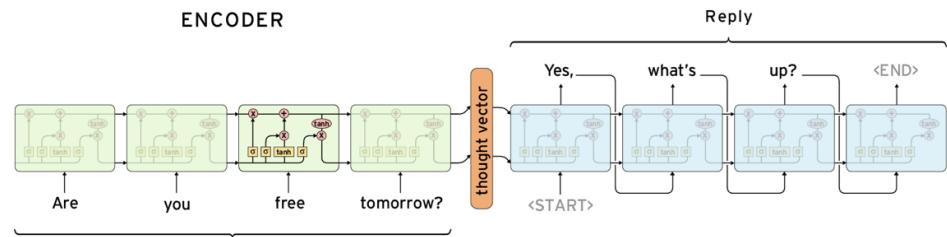
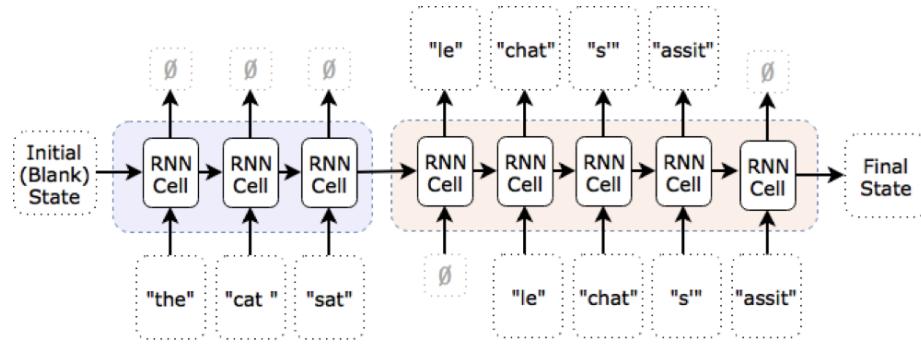


RNNs

many to many



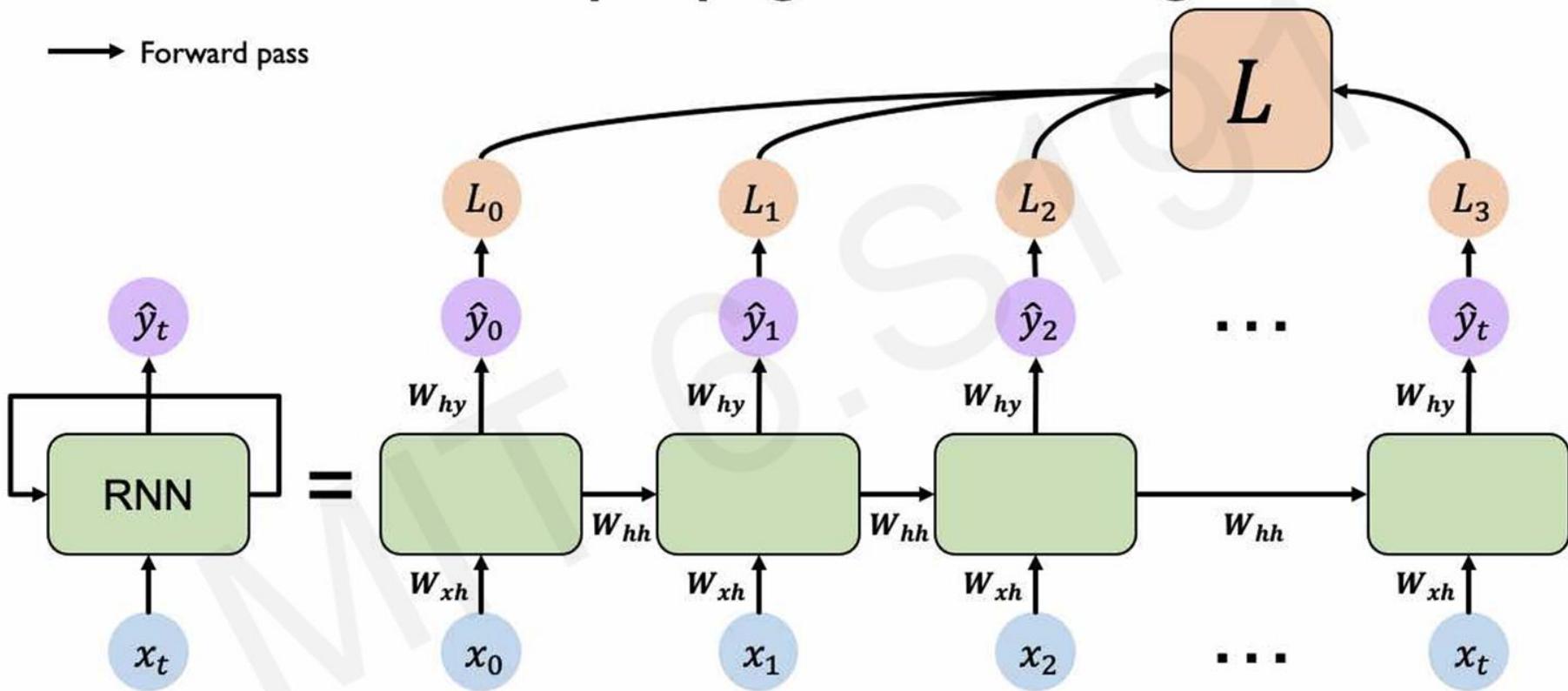
Ex:
Translation,
automated
response



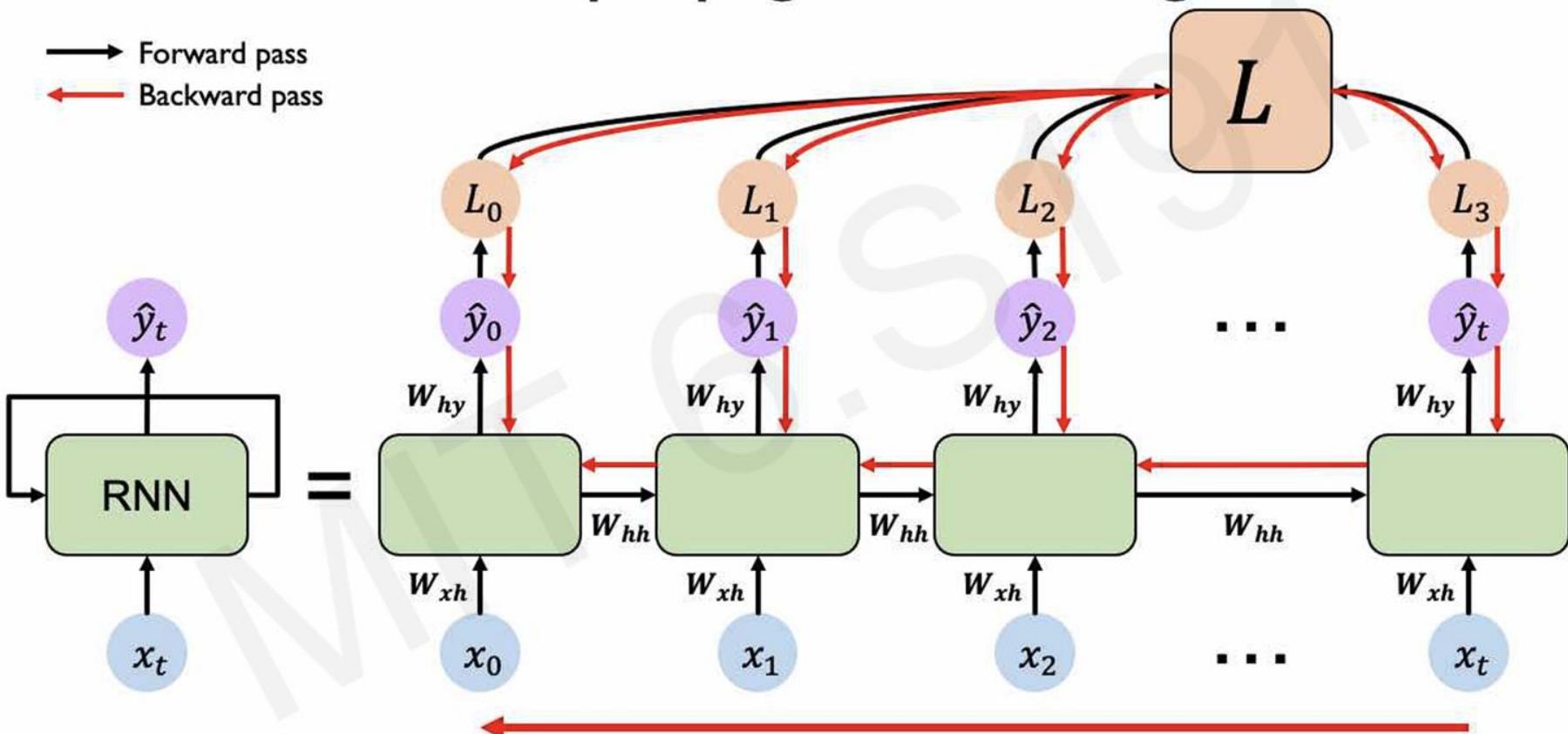
Incoming Email

DECODER

RNNs: Backpropagation Through Time



RNNs: Backpropagation Through Time



RNNs

- © High-level takeaways:

- RNNs provide a way to handle **sequence** data where the order of events is important
- “Simple” modification to MLP model
- RNNs maintain a “**state**” that reflects current configuration of the “world”

RNNs

- © High-level takeaways:

- RNNs provide a natural way to “update” your beliefs about the world as new information arrives
- Really **flexible** and can model many different scenarios that get weird/complicated quickly
- CNNs = hard to understand but easy to implement; RNNs = easy to understand but hard to implement

Applications

- ◎ Document and time series classification e.g. identifying the topic of an article or the author of a book
- ◎ Time series comparisons e.g. estimating how closely related two documents are
- ◎ Sentiment analysis
- ◎ Time series forecasting e.g. predicting weather (something that needs major improvement for Boston...)
- ◎ Sequence-to-sequence learning e.g. decoding an English sentence into Turkish

Problems with RNNs

Problems with RNNs

- ◎ Recall the formula for a generic RNN:

$$h_t = f(X_t W + h_{t-1} U + b)$$

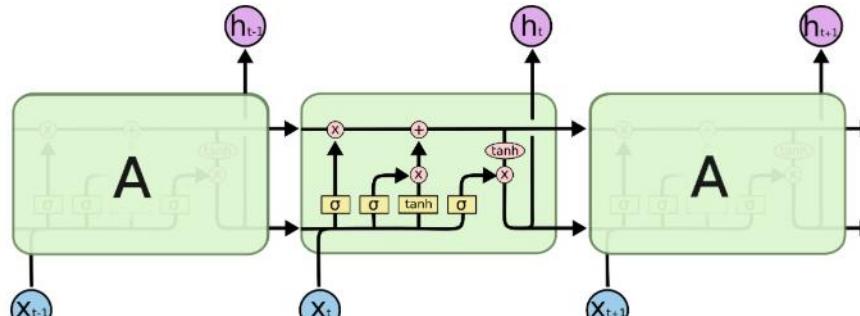
- ◎ What happens for really long sequences during backprop?

- You multiply by the matrix U repeatedly
- Largest eigenvalue > 1 , gradient $\rightarrow \infty$ (explodes)
- Largest eigenvalue < 1 , gradient $\rightarrow 0$ (vanishes)

- ◎ This is known as the **vanishing or exploding gradient problem**

Fixing RNNs

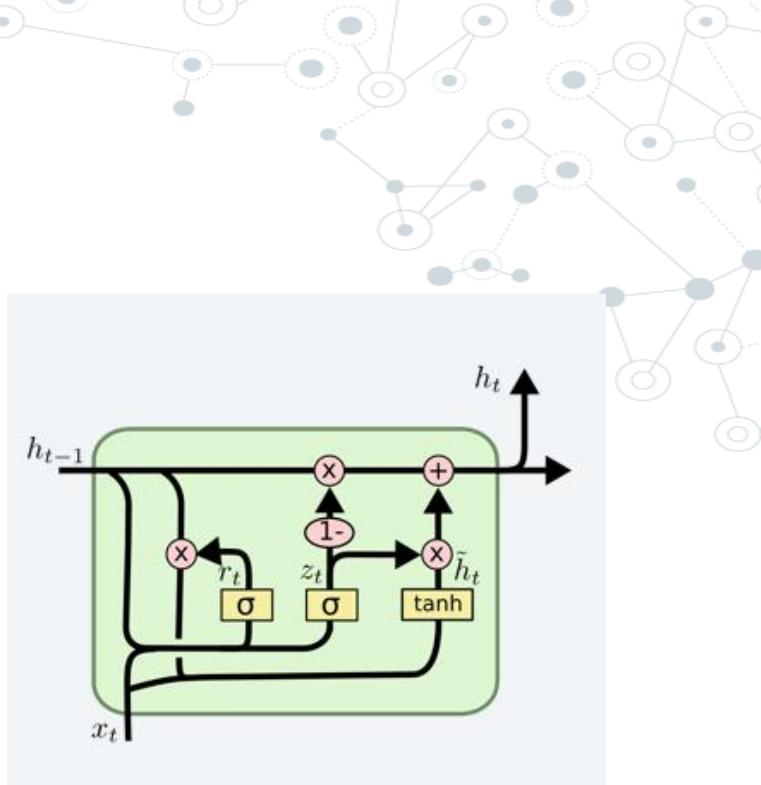
- ◎ Sepp Hochreiter and Jürgen Schmidhuber proposed the long short term memory (LSTM) hidden unit in 1997
- ◎ LSTMs selectively modify the inputs to produce “well-behaved” outputs, fixing the gradient issues
- ◎ Can model very long sequences without having the gradients vanish or explode



The repeating module in an LSTM contains four interacting layers.

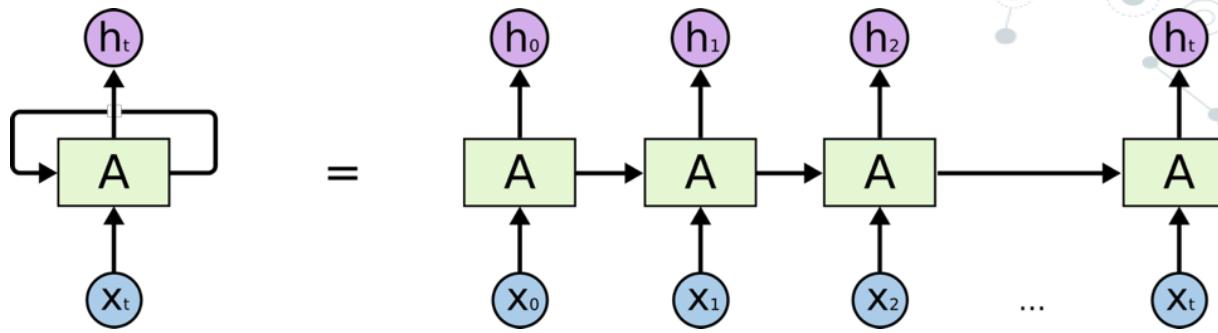
Fixing RNNs

- ◎ Gated Recurrent Network (GRU)
- ◎ Relatively new (2014), introduced by Cho et al.
- ◎ Combined aspects of the LSTM hidden unit
- ◎ Performance is on par with LSTM but computationally more efficient
- ◎ We'll dig into the details of these two new units



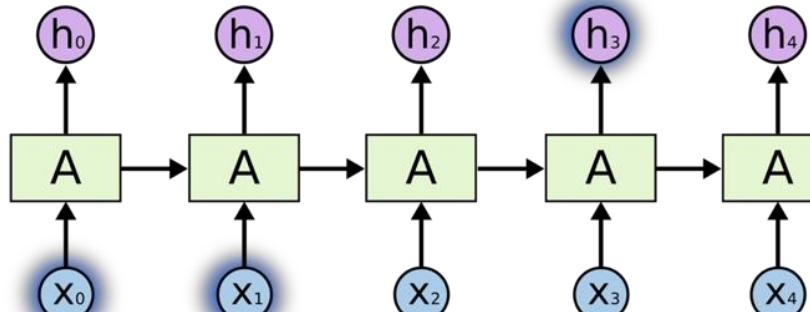
Long Short Term Memory (LSTM)

An “unrolled” RNN



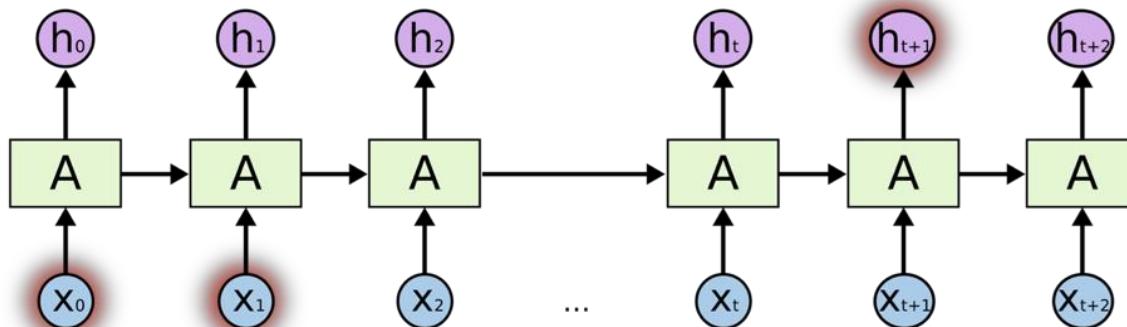
RNN where the output h_3 only depends on the input from X_0 and X_1
(The relevant information needed at h_3 comes from X_0 and X_1)

The gap between relevant information and the place it is needed is small

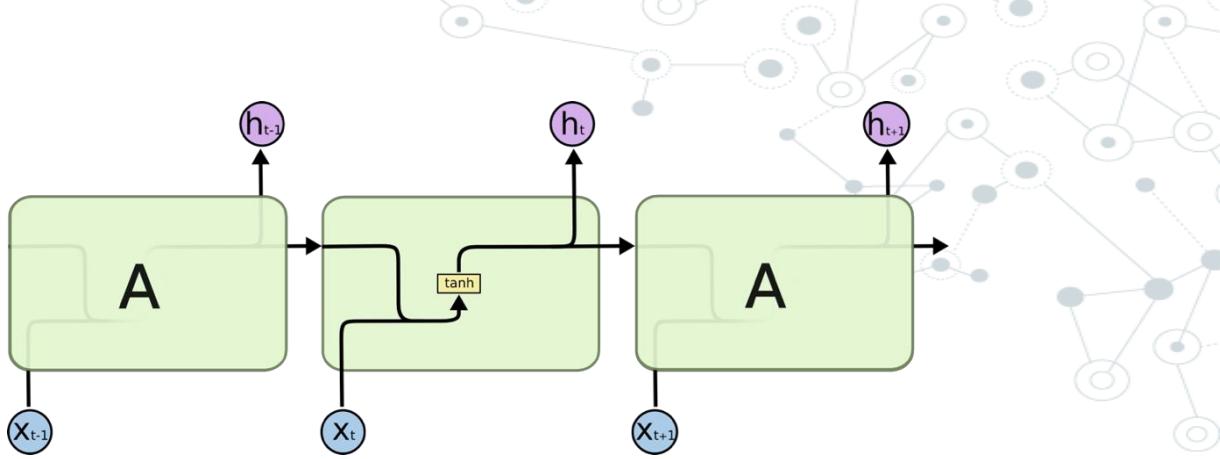


RNN where the output h_{t+1} is dependent on data inputs X_0 and X_1 that are too far for the gradient to carry

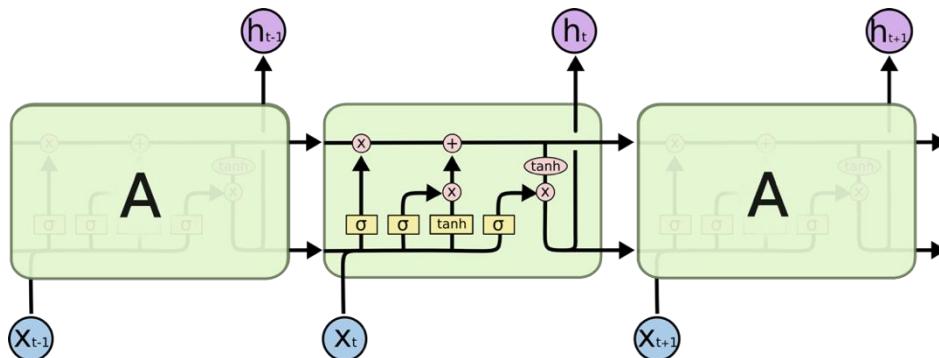
This is an example of a **long-term dependency** - RNNs struggle to learn to make connections when there are large gaps between the relevant information and where it is needed



Simple, “vanilla” RNN:



RNN with LSTM units:



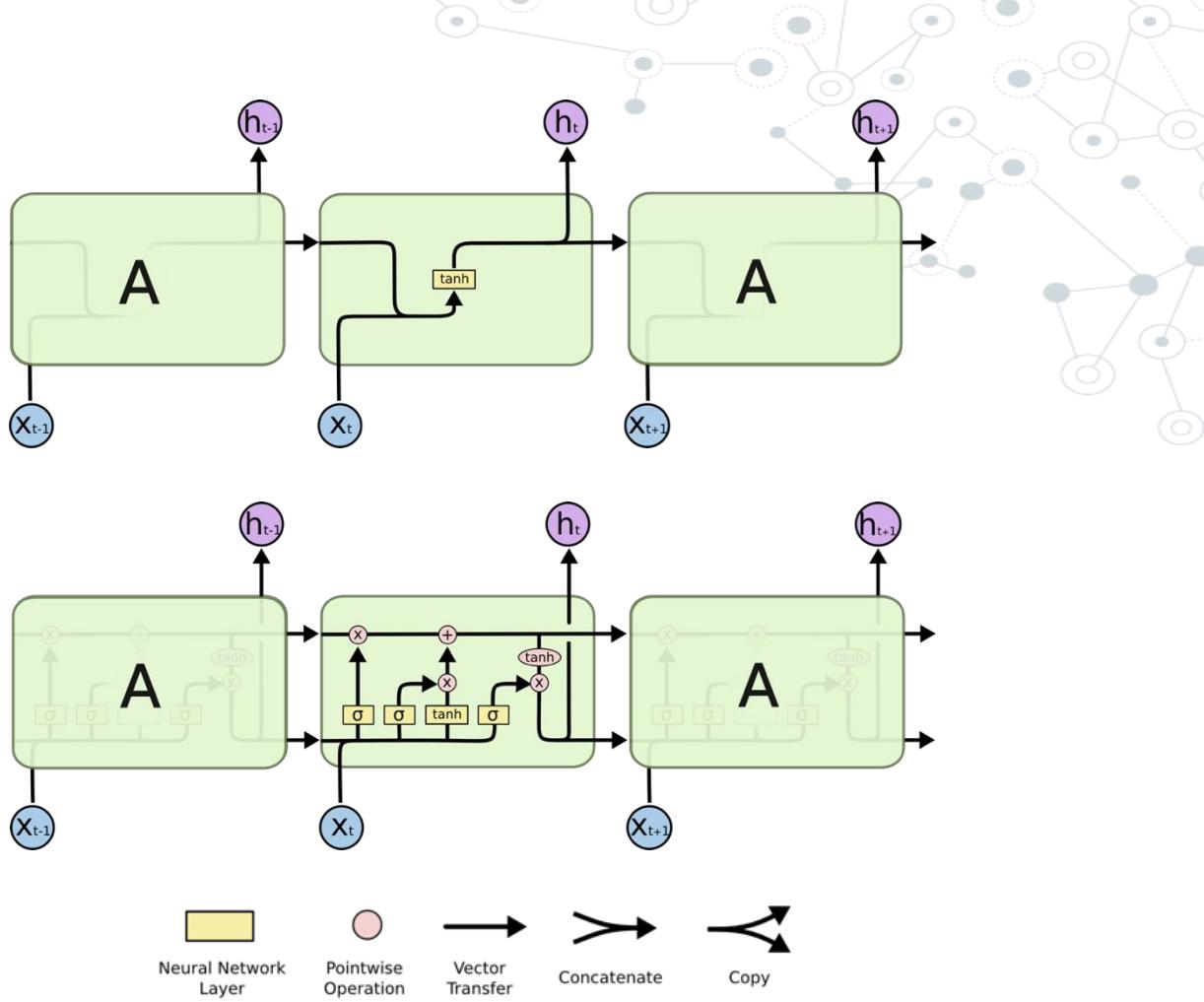
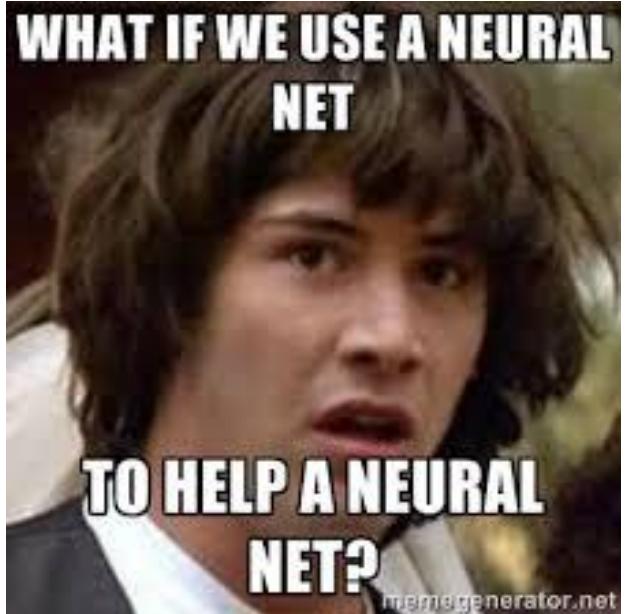
Neural Network Layer

Pointwise Operation

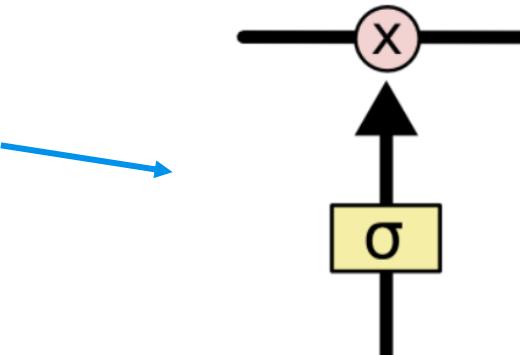
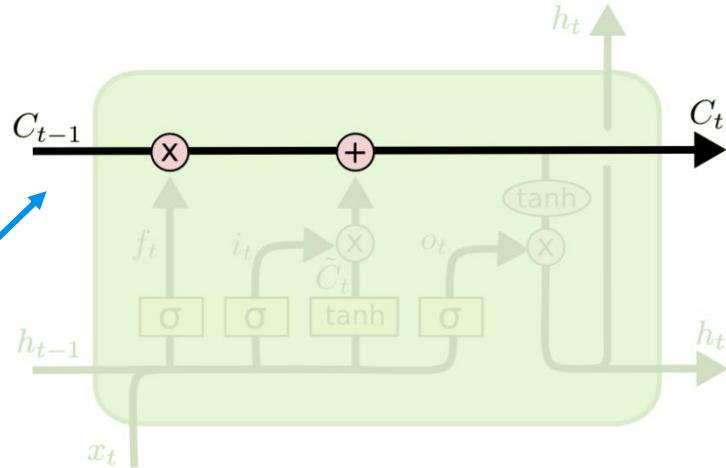
Vector Transfer

Concatenate

Copy

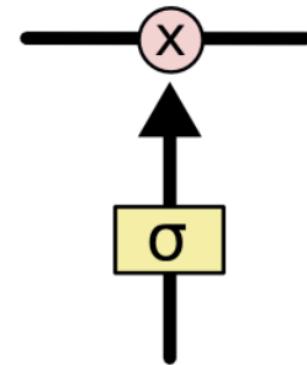


- ◎ LSTMs were explicitly designed to avoid the long-term dependency problem
- ◎ The key to LSTMs is the ability to let certain information through and carry it until it is deemed no longer useful (which may not happen)
- ◎ Information is carried through the sequence in the **cell state**, which acts as a conveyor belt or highway of information (memory of the network)
- ◎ Information is kept or forgotten by passing through **gates** (neural nets that regulate the flow of information from one time step to the next)

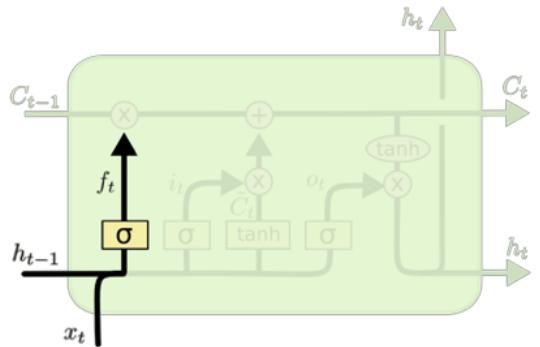


Gates

- ◎ Gates control which information is let through
 - ◎ They are composed of a sigmoid neural net layer and a pointwise multiplication operation
 - ◎ The sigmoid layer outputs numbers between 0 and 1, representing how much information should be let through
- ◎ $1 = \text{all information}, 0 = \text{no information}$

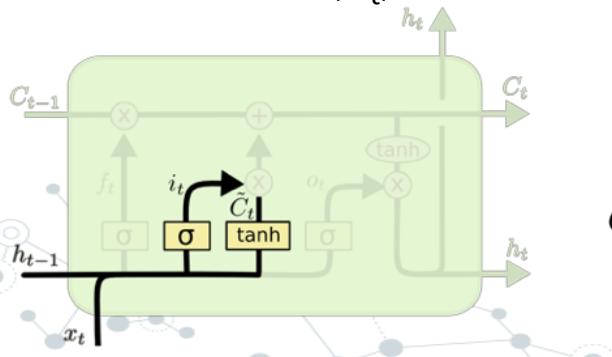


Step 1: Forget Gate - Determine how much of the previous state should affect the current state based on the current observed input x_t



$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

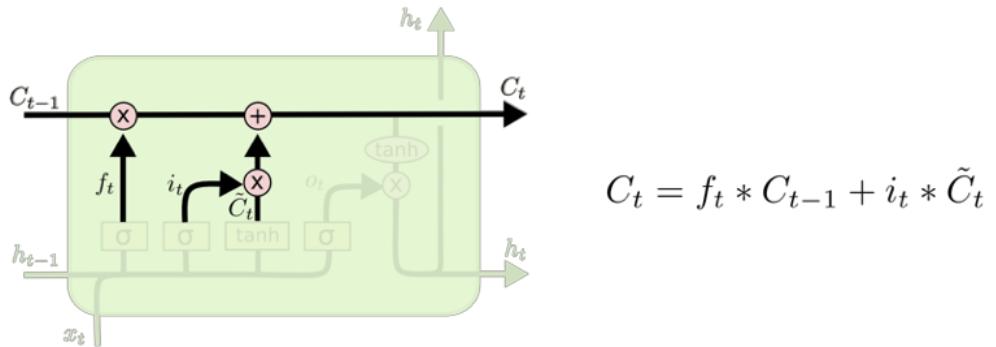
Step 2: Update Cell State - First determine which values we will update and by how much (gate i_t), then create a list of candidate values that we will add to the current state (C_t) based on the current input (x_t) and the previous output (h_{t-1}).



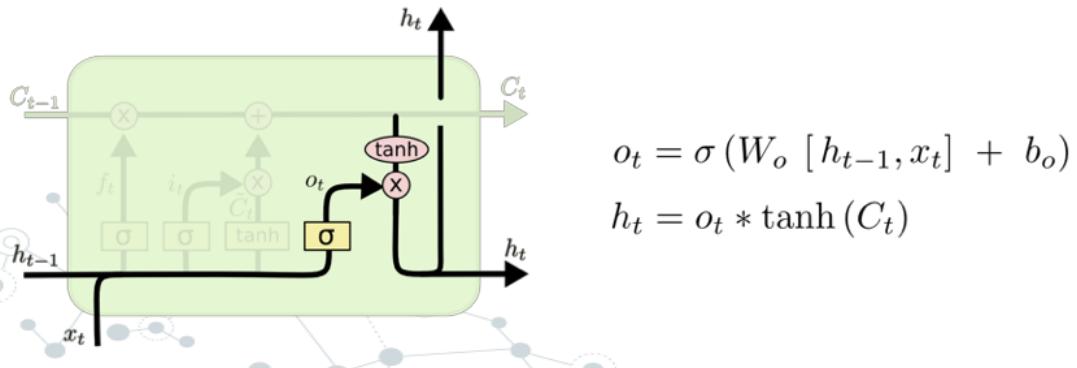
$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

Step 3: Execute the Update - update the cell state C_{t-1} to C_t .



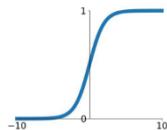
Step 4: Compute Unit Output - determine which parts of the cell state will be used as unit output. Output is a filtered version of the cell state.



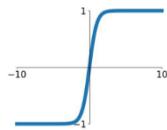
Why tanh?

- ◎ To overcome the vanishing/exploding gradient problem
- ◎ Force

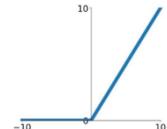
Sigmoid
 $\sigma(x) = \frac{1}{1+e^{-x}}$



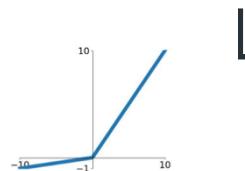
tanh
 $\tanh(x)$



ReLU
 $\max(0, x)$

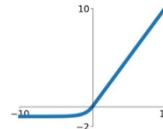


Leaky ReLU
 $\max(0.1x, x)$



Maxout
 $\max(w_1^T x + b_1, w_2^T x + b_2)$

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



LSTM Variants

- ◎ The steps we went through are for the standard, “normal” LSTM
- ◎ There are several variations - see blog post link from previous slide
- ◎ Encoder-decoder LSTMs led to the emergence of the

Attention Mechanism

- Selectively concentrates on a few relevant things while ignoring others
- ◎ Think of an encoder as part of a neural net that reads in a sequence, tries to summarize it (encode a context vector), and passes it to the decoder
- ◎ The decoder translates the input from the encoder
- ◎ The Attention Mechanism overcame shortcomings of encoder-decoder LSTMs and led to huge breakthroughs in NLP

IMDb Example

IMDb Example

Recall from a previous lecture:

The [IMDb data set](#) is a set of movie reviews that have been labeled as either positive or negative, based on the text content of the reviews

- ◎ **Training set:** 25,000 either positive or negative movie reviews that have each been turned into a vector of integers
 - We'll see how to actually do this later in the course
 - Each review can be of any length
 - Only the top 10,000 most frequently occurring words are kept i.e. rare words are discarded
 - Each review includes a label: 0 = negative review and 1 = positive review

Testing set: 25,000 either positive or negative movie reviews, similar to the training set

IMDb Example - Word Embeddings

- ◎ Keras has a function that enables learning word-embeddings: the **embedding** layer
- ◎ Basically a dictionary that maps integer indices (that represent words) to dense vectors
- ◎ It takes integers as input, looks up the integers in an internal dictionary, and returns the associated vectors

Word index →

Embedding layer →

Corresponding word

IMDb Example - Word Embeddings

- ◎ Input: 2D tensor of integers of shape (samples, sequence_length)
- ◎ Note that you need to select a sequence length that is the same for all sequences
- ◎ If a sequence is shorter than the set sequence length, pad the remaining entries with 0s
- ◎ If a sequence is longer than the set sequence length, truncate the sequence

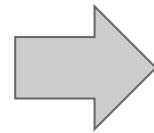
IMDb Example

Review 1: "This movie was great!"

Review 2: "This movie was so bad I quit after ten minutes."

Review 3: "The setting is enchanting and captivating."

Tokenization



[5, 6, 11, 32]

[5, 6, 11, 14, 66, 3, 49, 55, 98, 121]

[31, 12, 2, 77, 33, 78]

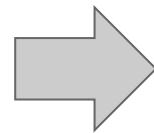
IMDb Example

Review 1: "This movie was great!"

Review 2: "This movie was so bad I quit after ten minutes."

Review 3: "The setting is enchanting and captivating."

Tokenization

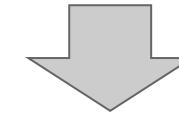


[5, 6, 11, 32]

[5, 6, 11, 14, 66, 3, 49 , 55, 98, 121]

[31, 12, 2, 77, 33, 78]

Padding



[5, 6, 11, 32, 0, 0, 0, 0, 0, 0]

[5, 6, 11, 14, 66, 3, 49 , 55, 98, 121]

[31, 12, 2, 77, 33, 78, 0, 0, 0, 0]

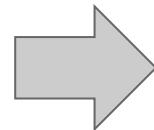
IMDb Example

Review 1: "This movie was great!"

Review 2: "This movie was so bad I quit after ten minutes."

Review 3: "The setting is enchanting and captivating."

Tokenization

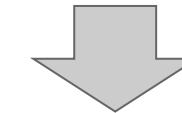


[5, 6, 11, 32]

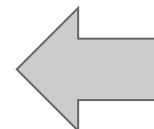
[5, 6, 11, 14, 66, 3, 49 , 55, 98, 121]

[31, 12, 2, 77, 33, 78]

Padding



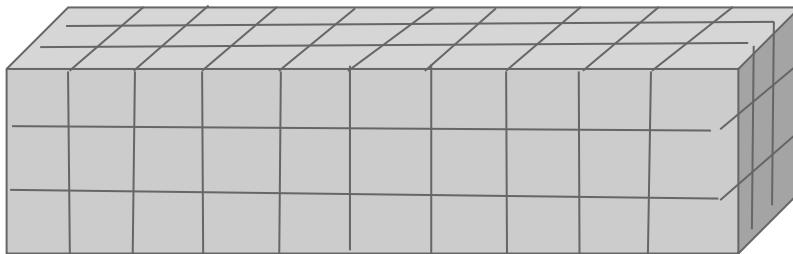
Embedding



[5, 6, 11, 32, 0, 0, 0, 0, 0, 0]

[5, 6, 11, 14, 66, 3, 49 , 55, 98, 121]

[31, 12, 2, 77, 33, 78, 0, 0, 0, 0]



Each word is represented by a vector with 3 elements



IMDb Example

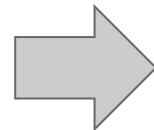
Review 1: "This movie was great!"

Review 2: "This movie was so bad I quit after ten minutes."

Review 3: "The setting is enchanting and captivating."

"This" = [0.1, 0.4, 0.6]

Tokenization

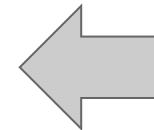


[5, 6, 11, 32]

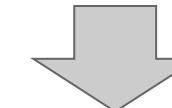
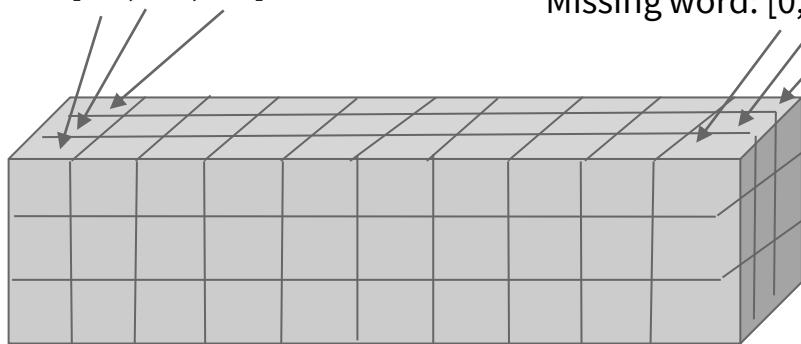
[5, 6, 11, 14, 66, 3, 49, 55, 98, 121]

[31, 12, 2, 77, 33, 78]

Embedding



Missing word: [0, 0, 0]



Padding

[5, 6, 11, 32, 0, 0, 0, 0, 0, 0]

[5, 6, 11, 14, 66, 3, 49, 55, 98, 121]

[31, 12, 2, 77, 33, 78, 0, 0, 0, 0]

Each word is represented by a vector with 3 elements. The input is now a 3D tensor of shape (3, 10, 3)

Number of reviews

Length of each review

Depth of word embedding: how many numbers represent a word

```
1 # Number of words to consider as features
2 max_features = 10000
3
4 # Cut texts after this number of words
5 # (among top max_features most common words)
6 maxlen = 20
7
8 # Load the data as lists of integers.
9 (x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=max_features)
10
11 # This turns our lists of integers into a 2D integer tensor
12 # of shape (samples, maxlen)
13 x_train = preprocessing.sequence.pad_sequences(x_train, maxlen=maxlen)
14 x_test = preprocessing.sequence.pad_sequences(x_test, maxlen=maxlen)
```



[Colab notebook](#)

We need each review to be the same length to feed into the network. This either “pads” reviews less than 20 words in length with zeros, or truncates reviews longer than 20 words to the first 20 words.

Here we will use the pre-tokenized IMDB data packaged in Keras

```
1 model = keras.Sequential([
2     # We specify the maximum input length to our Embedding layer
3     # so we can later flatten the embedded inputs
4     layers.Embedding(10000, 8, input_length = maxlen),
5
6     # After the Embedding layer,
7     # our activations have shape (samples, maxlen, 8).
8
9     # We flatten the 3D tensor of embeddings
10    # into a 2D tensor of shape (samples, maxlen * 8)
11    layers.Flatten(),
12
13    # We add the classifier on top
14    layers.Dense(1, activation='sigmoid')
15 ])
16
17 model.compile(loss = 'binary_crossentropy',
18                 optimizer = tf.keras.optimizers.RMSprop(),
19                 metrics = ['accuracy'])
20
21
22 history = model.fit(x_train, y_train,
23                       epochs = 10,
24                       batch_size = 32,
25                       validation_split = 0.2)
```

8-dimensional embeddings - one for each word

Length of sequence

Size of vocabulary

Note that we aren't fitting an RNN yet - this is an MLP network. We are first focusing on how to use word embeddings.

IMDb Example - Word Embeddings

- ◎ We get an accuracy of about 75%
 - Not bad for only using the first 20 words of a review
- ◎ Here we are merely flattening the embedded sequences and training a single dense layer on top
 - This treats each word in the input sequence separately, without considering inter-word relationships and structure sentence (e.g. it would likely treat both "this movie is shit" and "this movie is the shit" as being “negative” reviews).
 - It would be much better to add recurrent layers or 1D convolutional layers on top of the embedded sequences to learn features that take into account each sequence as a whole. That's what we will focus on next.

IMDb Example - Word Embeddings

- ◎ Now we'll do the same thing but with pre-trained word embeddings
 - We'll use GloVe embeddings
- ◎ We have to download both the [raw IMDb reviews](#) and [GloVe embeddings](#) before running the code
 - I have also imported them into the Google Drive Data folder
 - [IMDb reviews](#)
 - [GloVe embeddings](#)

Pre-trained embeddings are meant to perform well on small data sets - let's see how well our model does if we only train on 200 reviews

```
1 from keras.preprocessing.text import Tokenizer
2 from keras.preprocessing.sequence import pad_sequences
3 import numpy as np
4
5 maxlen = 100          # We will cut reviews after 100 words
6 training_samples = 200    # We will be training on 200 samples
7 validation_samples = 10000 # We will be validating on 10000 samples
8 max_words = 10000       # We will only consider the top 10,000 words in the dataset
9
10 tokenizer = Tokenizer(num_words=max_words)
11 tokenizer.fit_on_texts(texts)
12 sequences = tokenizer.texts_to_sequences(texts)
13
14 word_index = tokenizer.word_index
15 print('Found %s unique tokens.' % len(word_index))
16
17 data = pad_sequences(sequences, maxlen=maxlen)
18
19 labels = np.asarray(labels)
20 print('Shape of data tensor:', data.shape)
21 print('Shape of label tensor:', labels.shape)
22
23 # Split the data into a training set and a validation set
24 # But first, shuffle the data, since we started from data
25 # where sample are ordered (all negative first, then all positive).
26 indices = np.arange(data.shape[0])
27 np.random.shuffle(indices)
28 data = data[indices]
29 labels = labels[indices]
30
31 x_train = data[:training_samples]
32 y_train = labels[:training_samples]
33 x_val = data[training_samples: training_samples + validation_samples]
34 y_val = labels[training_samples: training_samples + validation_samples]
```

Found 88582 unique tokens.

Shape of data tensor: (25030, 100)

Shape of label tensor: (25030,)

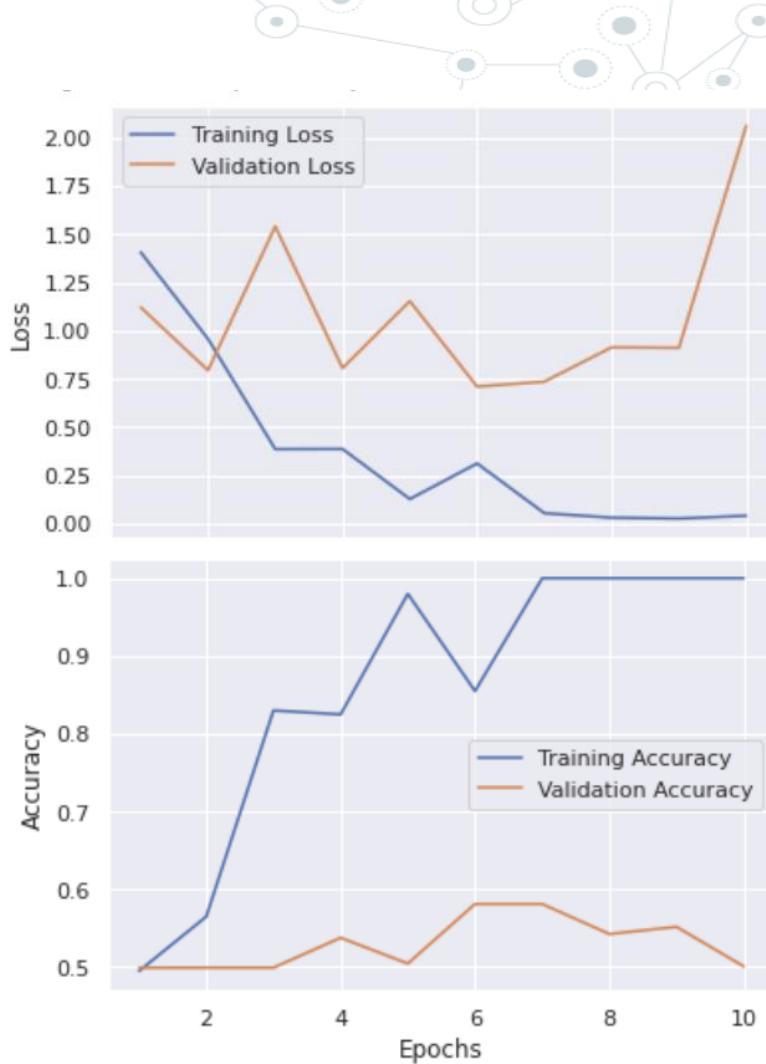
```
1 # Heather's directory - change to your path
2 glove_dir = 'drive/My Drive/Teaching/BST 261/2021/261StudentFolder/In-class examples/Data/glove/'
3
4 embeddings_index = {}
5 f = open(os.path.join(glove_dir, 'glove.6B.100d.txt'))
6 for line in f:
7     values = line.split()
8     word = values[0]
9     coefs = np.asarray(values[1:], dtype='float32')
10    embeddings_index[word] = coefs
11 f.close()
12
13 print('Found %s word vectors.' % len(embeddings_index))
```

```
1 embedding_dim = 100
2
3 embedding_matrix = np.zeros((max_words, embedding_dim))
4 for word, i in word_index.items():
5     embedding_vector = embeddings_index.get(word)
6     if i < max_words:
7         if embedding_vector is not None:
8             # Words not found in embedding index will be all-zeros.
9             embedding_matrix[i] = embedding_vector
```

The model quickly starts overfitting, unsurprisingly given the small number of training samples.

Validation accuracy has high variance for the same reason, but seems to reach high 50s.

The test set accuracy is a terrible 56%.



- We'll get into more complicated RNNs, but for now let's build a simple RNN and run it on the IMDb data set

- SimpleRNN is a layer that can be run in two different modes

- It can return either the full sequences of successive outputs for each timestep (a 3D tensor of shape (batch_size, timesteps, output_features)),
- Or it can return only the last output for each input sequence (a 2D tensor of shape (batch_size, output_features)).

These two modes are controlled by the **return_sequences** constructor argument.

```
1 model = keras.Sequential([
2   layers.Embedding(10000, 32),
3
4   layers.SimpleRNN(32),
5 ])
6
7 model.summary()
```

Model: "sequential_2"

Layer (type)	Output Shape	Param #
embedding_2 (Embedding)	(None, None, 32)	320000
simple_rnn (SimpleRNN)	(None, 32)	2080
=====		
Total params: 322,080		
Trainable params: 322,080		
Non-trainable params: 0		

```
1 model = keras.Sequential([
2   layers.Embedding(10000, 32),
3
4   layers.SimpleRNN(32, return_sequences=True),
5 ])
6
7 model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
embedding (Embedding)	(None, None, 32)	320000
simple_rnn (SimpleRNN)	(None, None, 32)	2080
=====		
Total params: 322,080		
Trainable params: 322,080		
Non-trainable params: 0		

- We'll get into more complicated RNNs, but for now let's build a simple RNN and run it on the IMDb data set

- SimpleRNN is a layer that can be run in two different modes

- It can return either the full sequences of successive outputs for each timestep (a 3D tensor of shape (batch_size, timesteps, output_features)),
- Or it can return only the last output for each input sequence (a 2D tensor of shape (batch_size, output_features)).

These two modes are controlled by the **return_sequences** constructor argument.

```
1 model = keras.Sequential([
2   layers.Embedding(10000, 32),
3
4   layers.SimpleRNN(32),
5 ])
6
7 model.summary()
```

Model: "sequential_2"

Layer (type)	Output Shape	Param #
embedding_2 (Embedding)	(None, None, 32)	320000
simple_rnn (SimpleRNN)	(None, 32)	2080

Total params: 322,080
Trainable params: 322,080
Non-trainable params: 0

```
1 model = keras.Sequential([
2   layers.Embedding(10000, 32),
3
4   layers.SimpleRNN(32, return_sequences=True),
5 ])
6
7 model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
embedding (Embedding)	(None, None, 32)	320000
simple_rnn (SimpleRNN)	(None, None, 32)	2080

Total params: 322,080
Trainable params: 322,080
Non-trainable params: 0

- We'll get into more complicated RNNs, but for now let's build a simple RNN and run it on the IMDb data set

- SimpleRNN is a layer that can be run in two different modes

- It can return either the full sequences of successive outputs for each timestep (a 3D tensor of shape (batch_size, timesteps, output_features)),
- Or it can return only the last output for each input sequence (a 2D tensor of shape (batch_size, output_features)).

These two modes are controlled by the **return_sequences** constructor argument.

```
1 model = keras.Sequential([
2   layers.Embedding(10000, 32),
3
4   layers.SimpleRNN(32),
5 ])
6
7 model.summary()
```

Model: "sequential_2"

Layer (type)	Output Shape	Param #
embedding_2 (Embedding)	(None, None, 32)	320000
simple_rnn (SimpleRNN)	(None, 32)	2080
=====		
Total params: 322,080 Trainable params: 322,080 Non-trainable params: 0		

```
1 model = keras.Sequential([
2   layers.Embedding(10000, 32),
3
4   layers.SimpleRNN(32, return_sequences=True),
5 ])
6
7 model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
embedding (Embedding)	(None, None, 32)	320000
simple_rnn (SimpleRNN)	(None, None, 32)	2080
=====		
Total params: 322,080 Trainable params: 322,080 Non-trainable params: 0		

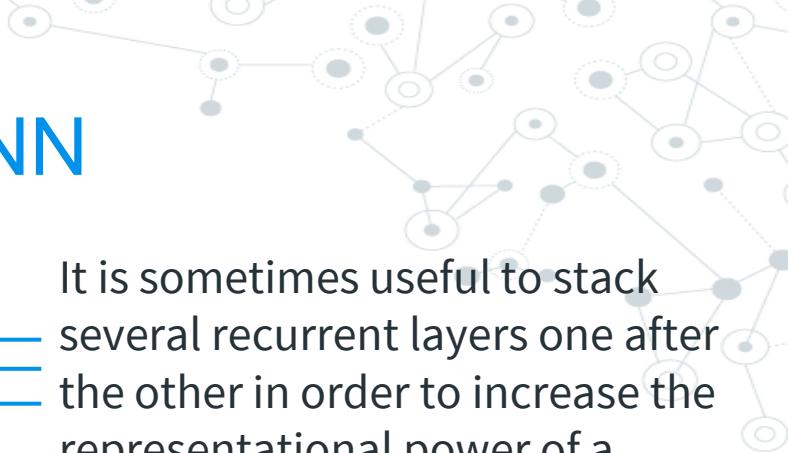
IMDb Example - Simple RNN

```
1 model = keras.Sequential([
2     layers.Embedding(10000, 32),
3
4     layers.SimpleRNN(32, return_sequences=True), ←
5     layers.SimpleRNN(32, return_sequences=True), ←
6     layers.SimpleRNN(32, return_sequences=True), ←
7     layers.SimpleRNN(32), # This last layer only returns the last outputs. ←
8 ])
9
10 model.summary()
```

Model: "sequential_4"

Layer (type)	Output Shape	Param #
<hr/>		
embedding_4 (Embedding)	(None, None, 32)	320000
simple_rnn_2 (SimpleRNN)	(None, None, 32)	2080
simple_rnn_3 (SimpleRNN)	(None, None, 32)	2080
simple_rnn_4 (SimpleRNN)	(None, None, 32)	2080
simple_rnn_5 (SimpleRNN)	(None, 32)	2080
<hr/>		

Total params: 328,320
Trainable params: 328,320
Non-trainable params: 0



It is sometimes useful to stack several recurrent layers one after the other in order to increase the representational power of a network. In such a setup, you have to get **all intermediate layers to return full sequences**.

IMDb Example - Simple RNN

```
1 model = keras.Sequential([
2     layers.Embedding(max_features, 32),
3
4     layers.SimpleRNN(32),
5
6     layers.Dense(1, activation='sigmoid')
7 ])
8
9 model.compile(optimizer = tf.keras.optimizers.RMSprop(),
10                 loss='binary_crossentropy',
11                 metrics=['accuracy'])
12
13 history = model.fit(input_train, y_train,
14                       epochs=10,
15                       batch_size=128,
16                       validation_split=0.2)
```

As a reminder, in lecture 3, our very first naive approach to this very dataset got us to 88% test accuracy. Our small recurrent network doesn't perform very well at all compared to this baseline (only up to 85% validation accuracy). Part of the problem is that our inputs only consider the first 100 words rather the full sequences -- hence our RNN has access to less information than our earlier baseline model.

The remainder of the problem is simply that SimpleRNN isn't very good at processing long sequences, like text.

Other types of recurrent layers perform much better. We'll talk about these next.

