



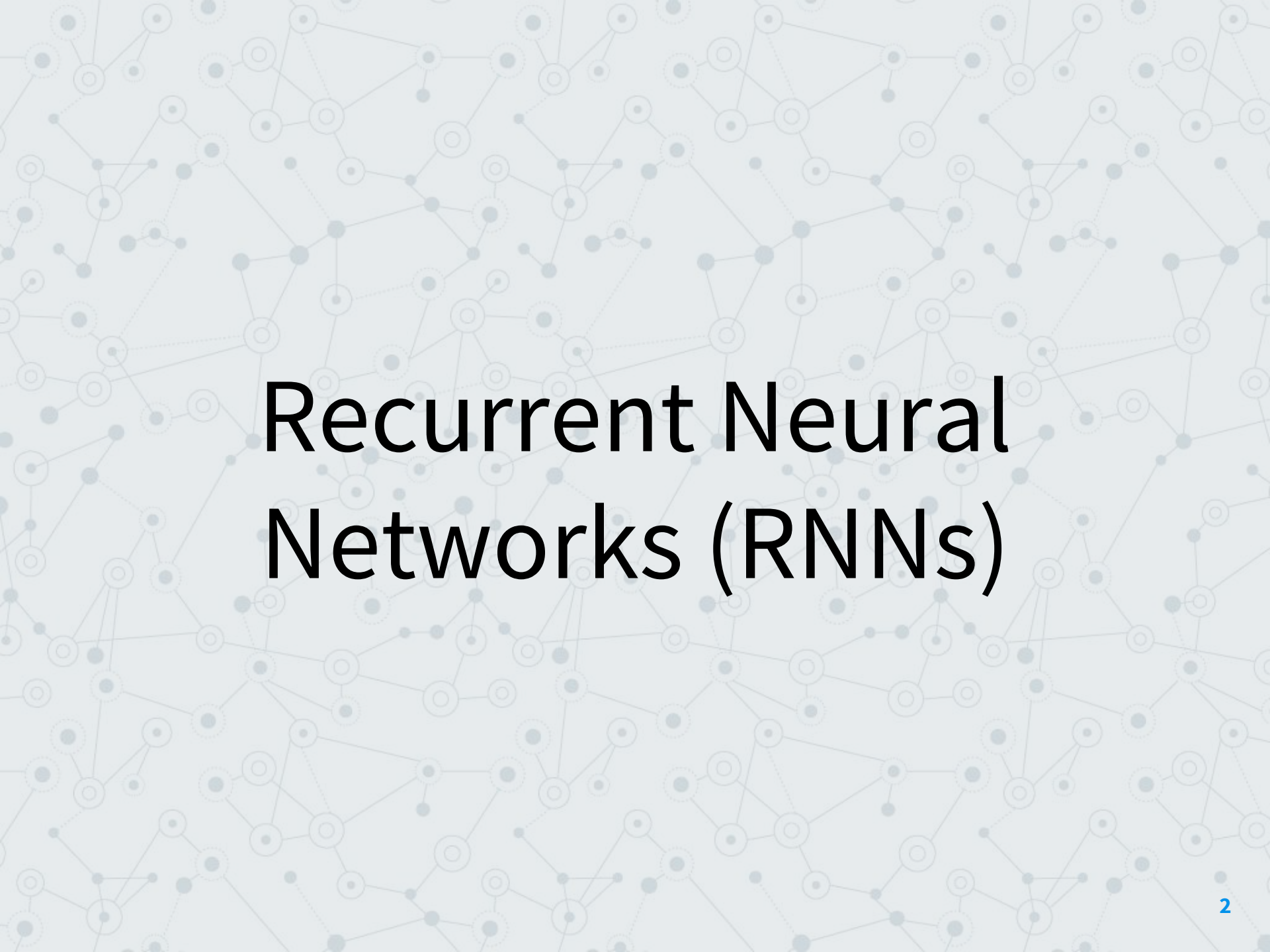
BST 261: Data Science II

Lecture 8

Introduction to Recurrent Neural Networks (RNNs)

Heather Mattie
Harvard T.H. Chan School of Public Health
Spring 2019





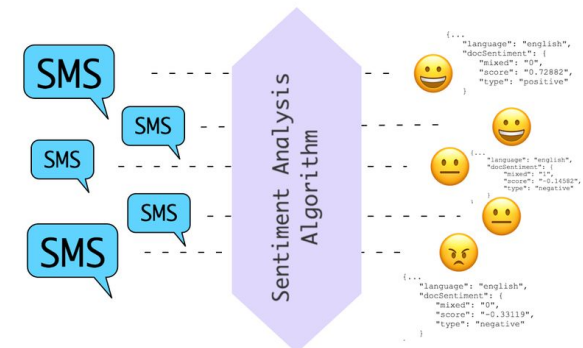
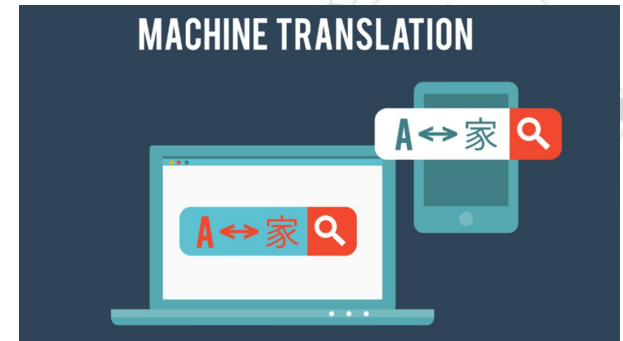
Recurrent Neural Networks (RNNs)

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



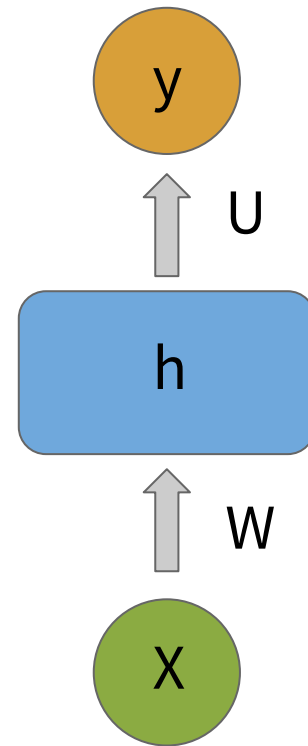
MLPs \longrightarrow RNNs

- ⊙ 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	Outputs	Probability
MLP	X	y	$P(y X)$
RNN	$[x_1, x_2, x_3, \dots, x_t]$	y	$P(y x_1, x_2, x_3, \dots, x_t)$

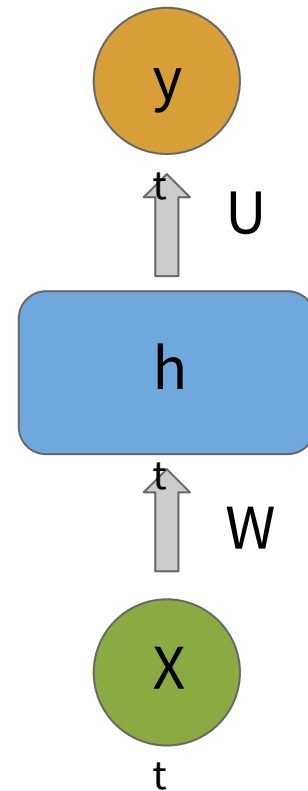
MLPs \rightarrow RNNs

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



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



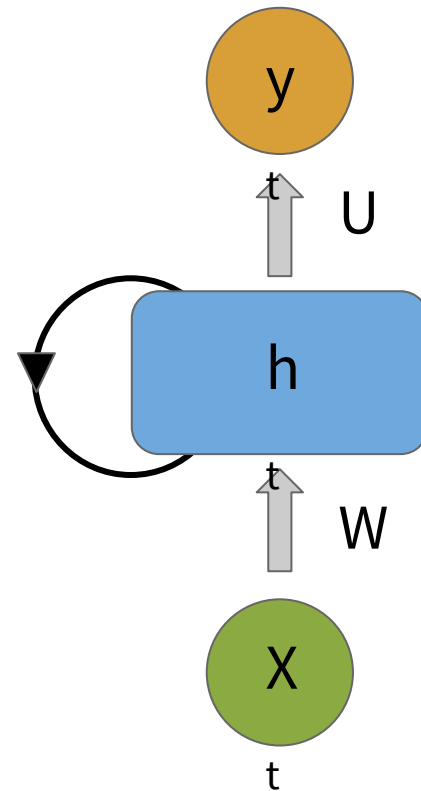
MLPs \longrightarrow RNNs

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

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

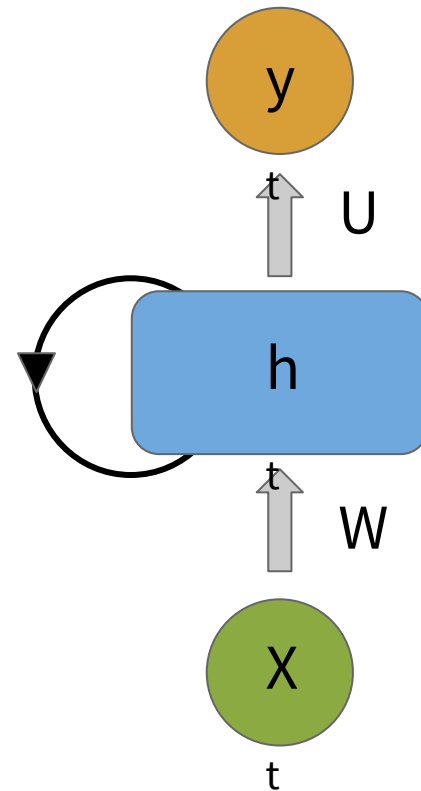
\uparrow \uparrow
Input at Hidden state
time t from previous
 time point

- ◎ This is equivalent to connecting the hidden state to itself



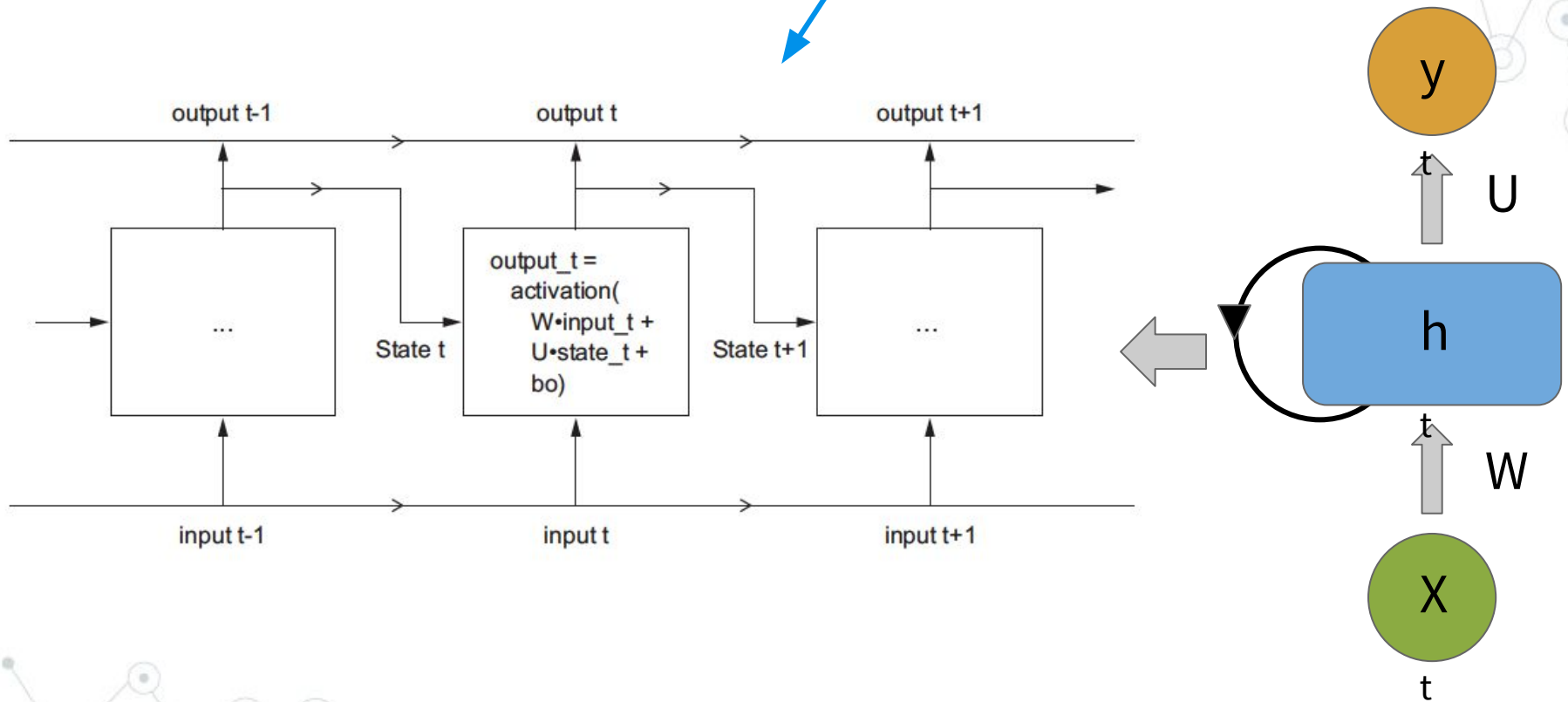
RNN Backprop

- ⦿ 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



RNN Backprop

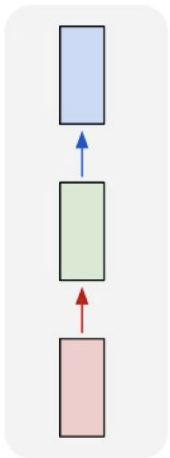
“Unrolled” RNN



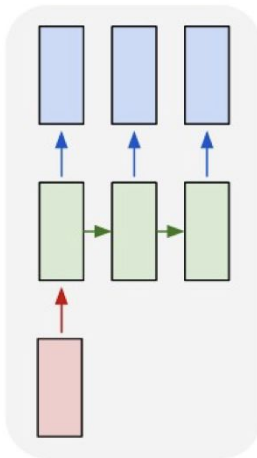
RNNs

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

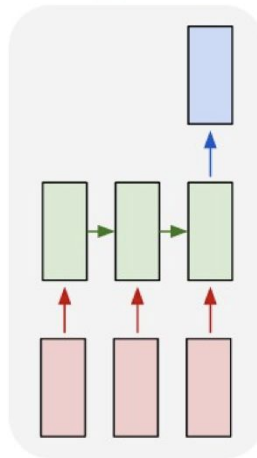
one to one



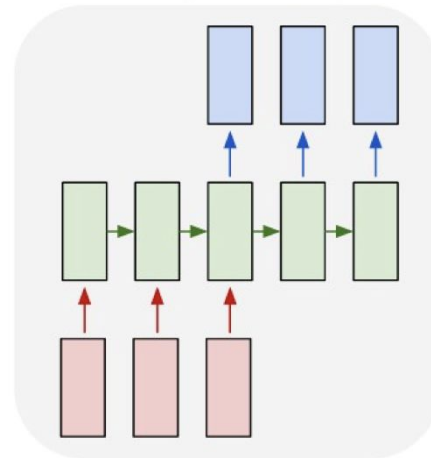
one to many



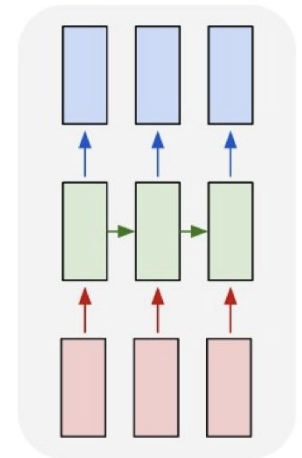
many to one



many to many

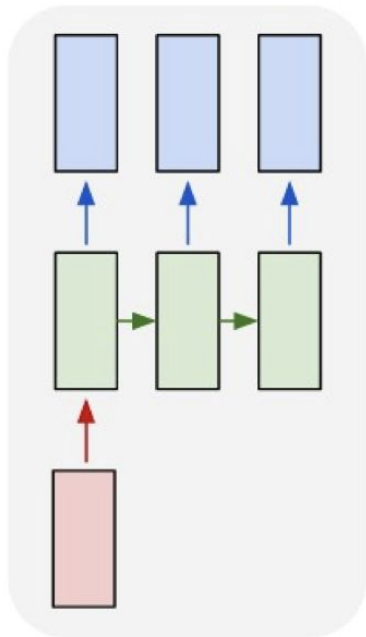


many to many

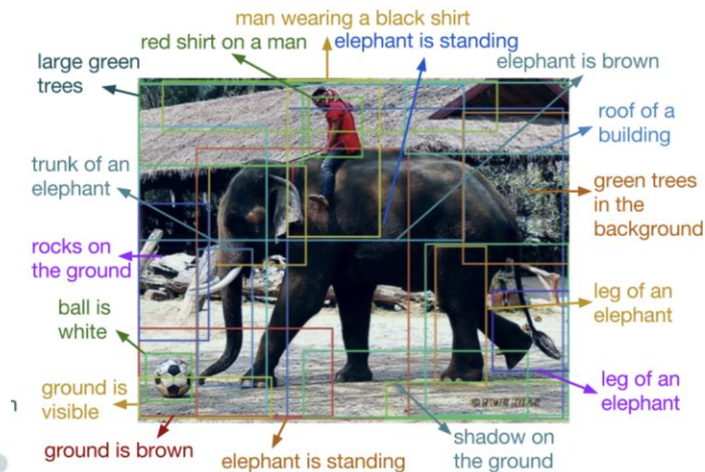
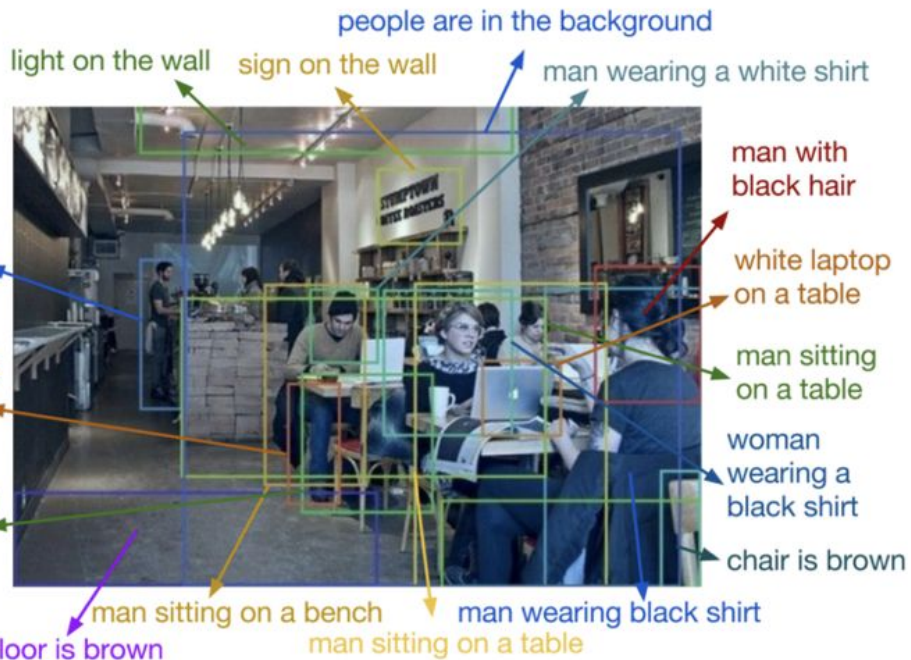
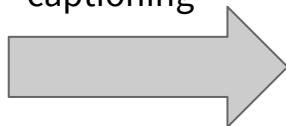


RNNs

one to many



Ex: image captioning



RCNN

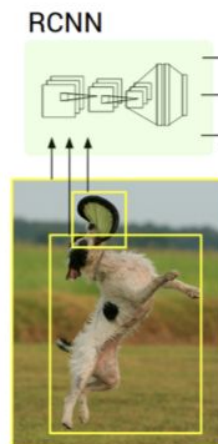
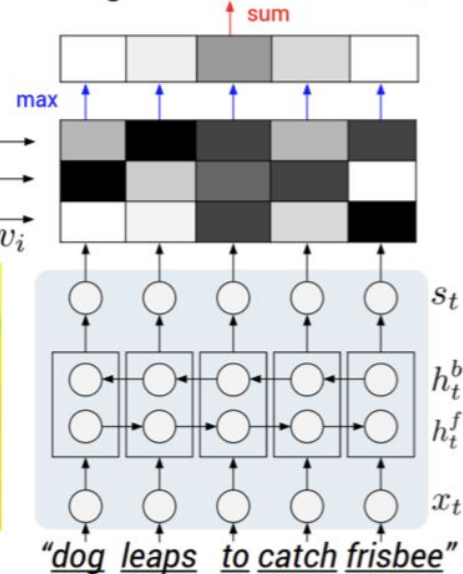
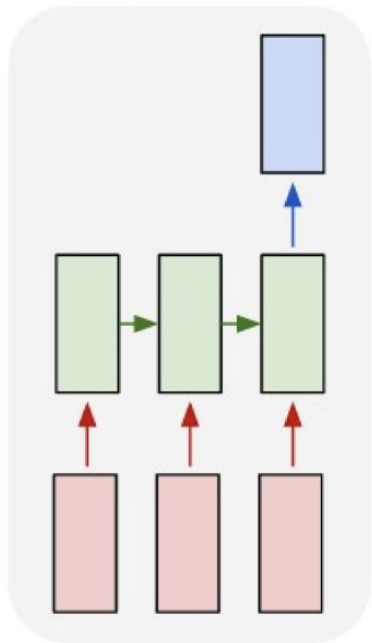


image - sentence score S_{kl}

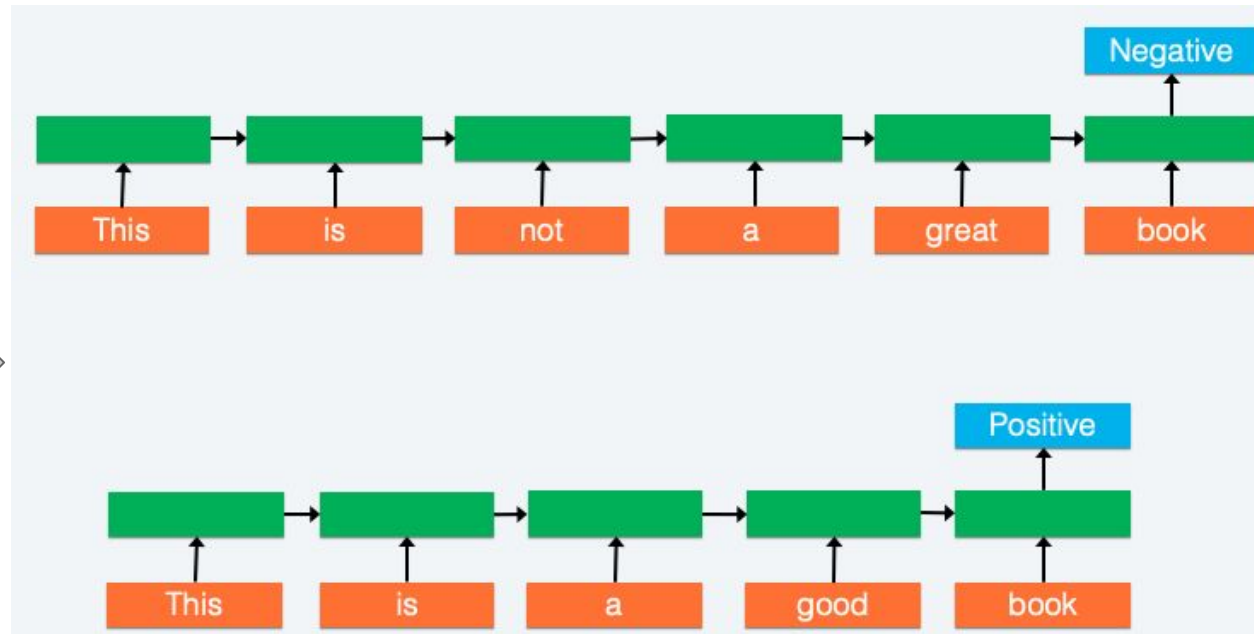


RNNs

many to one

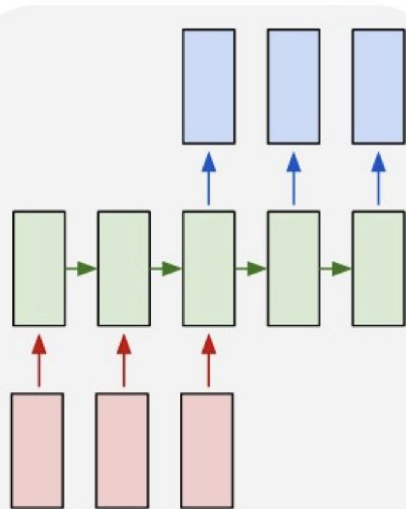


Ex:
Sentiment
Analysis

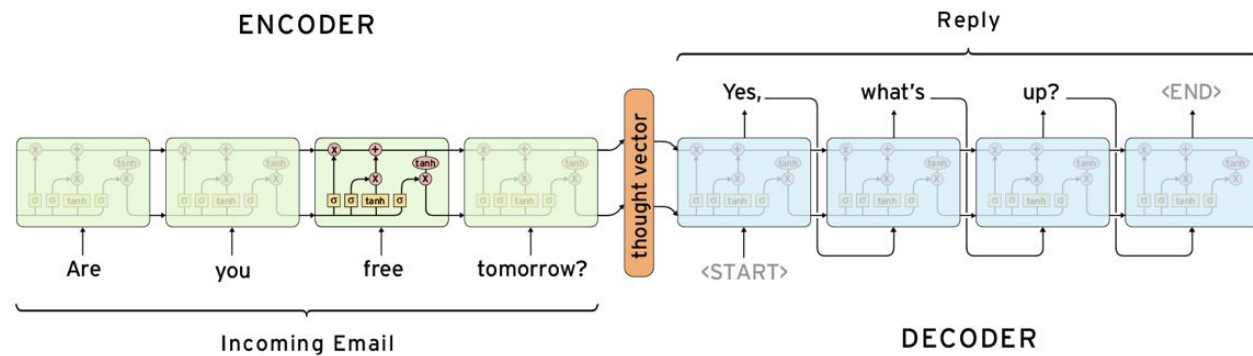
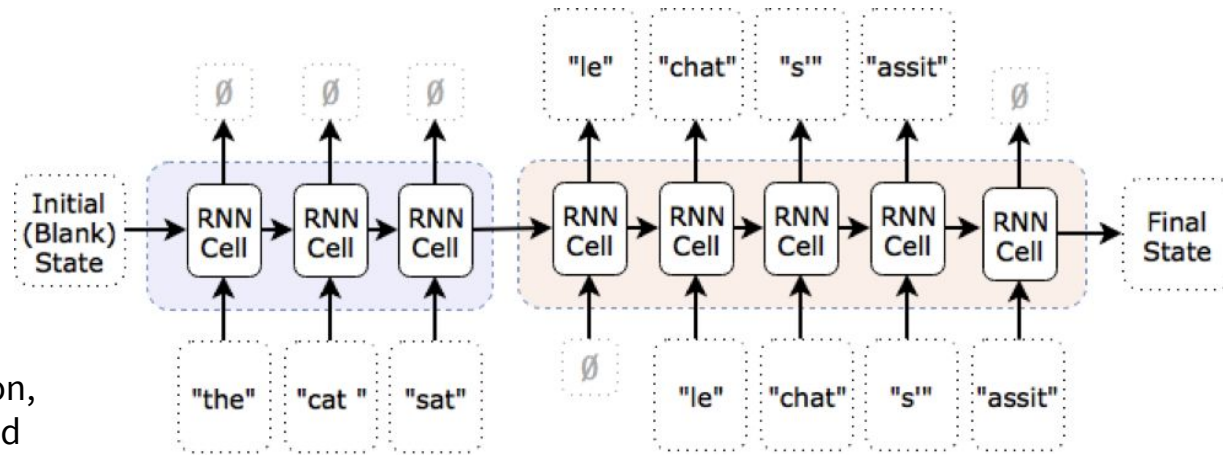


RNNs

many to many

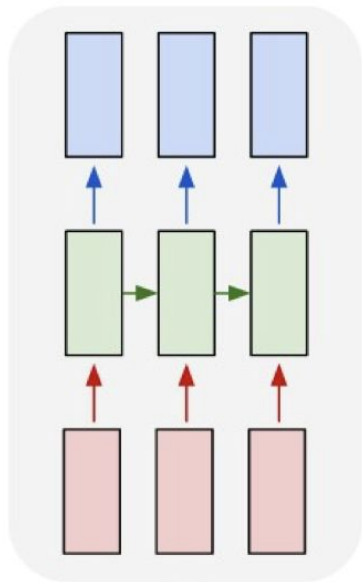


Ex:
Translation,
automated
response

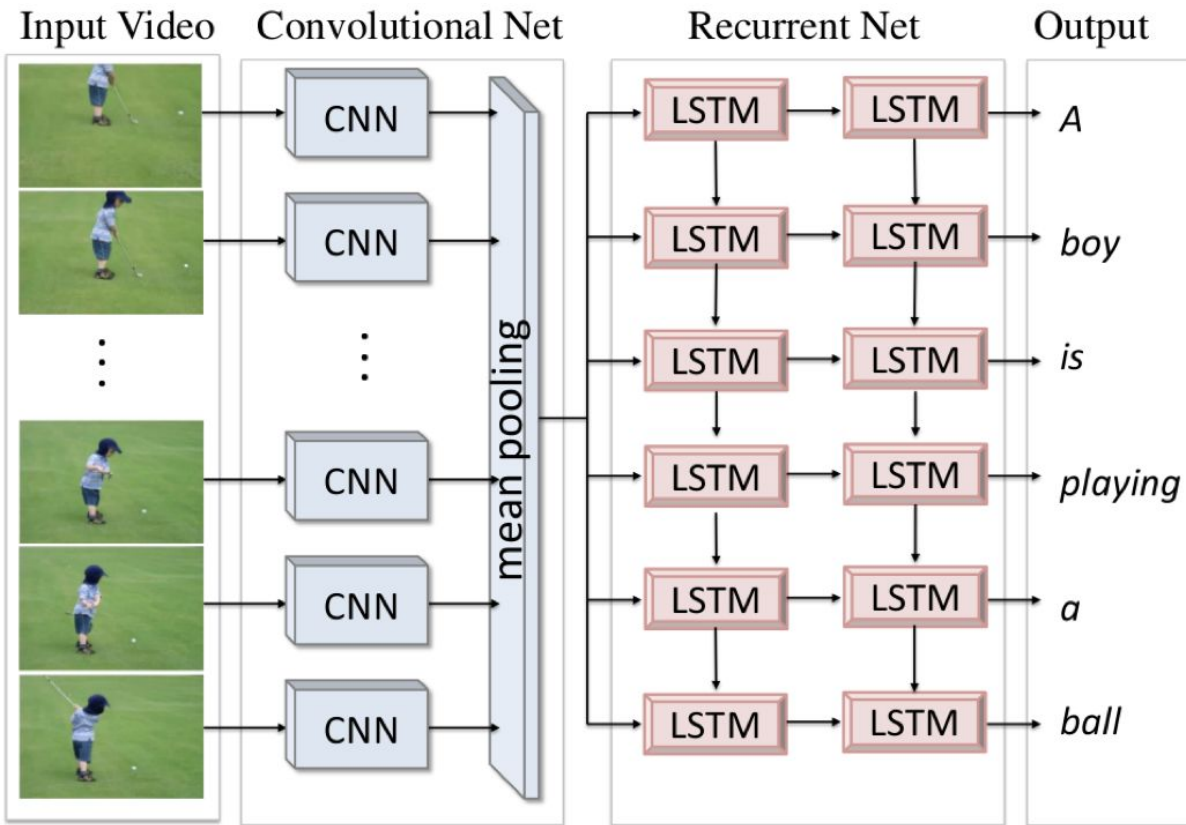
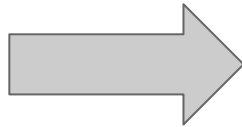


RNNs

many to many



Ex: frame by
frame image
captioning



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

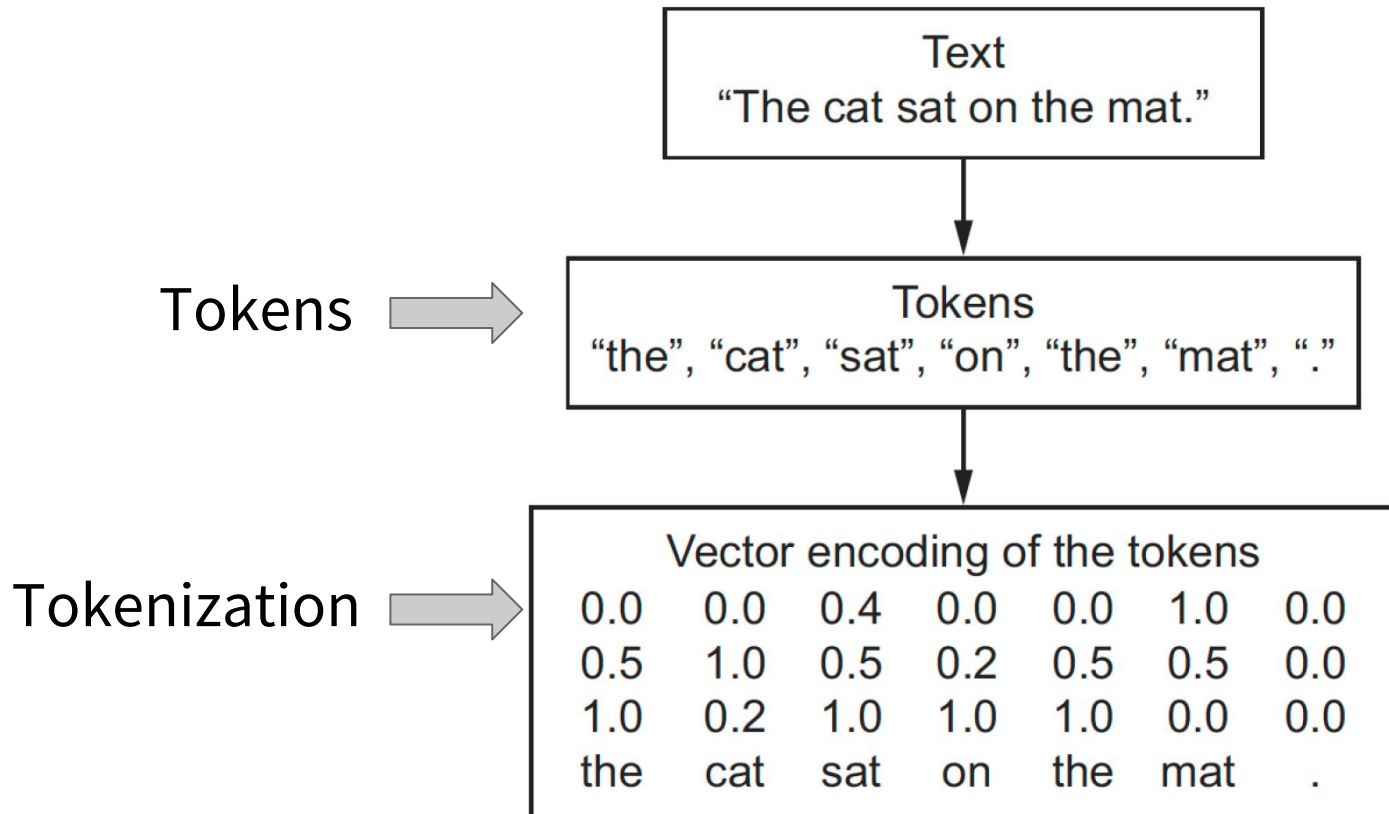
The background of the slide is a light gray network pattern. It consists of numerous small circles, some of which are solid gray and others are hollow with a gray outline. These circles are interconnected by a web of thin, light gray lines, creating a complex, organic structure that resembles a neural network or a data graph.

Working with text data

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
- ◎ 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

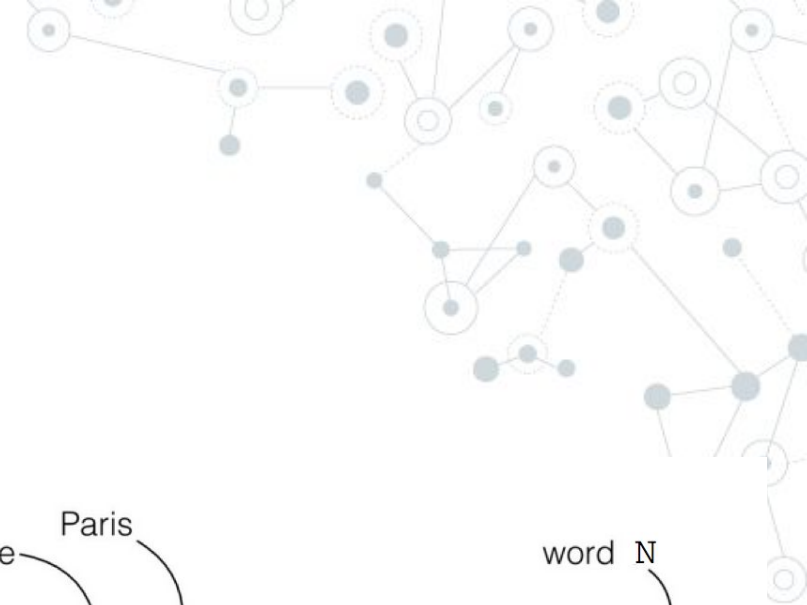
- ◎ Word **n-grams** are groups of N (or fewer) consecutive words that you can extract from a sentence. The same concept may also be applied to characters instead of words.
- ◎ For example, the sentence "**Data science rocks my socks off!**" can be decomposed into a set of 3-grams:
 - {"**Data**", "**Data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

Rome = [1, 0, 0, 0, 0, 0, ..., 0]

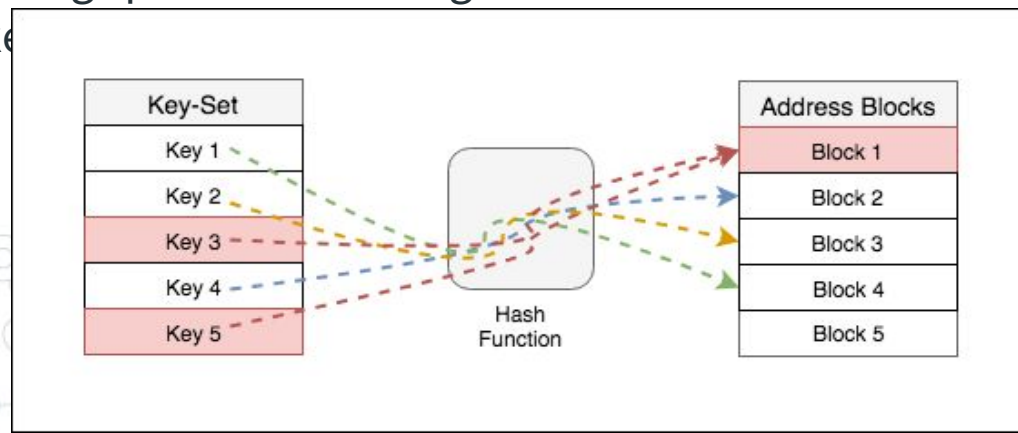
Paris = [0, 1, 0, 0, 0, 0, ..., 0]

Italy = [0, 0, 1, 0, 0, 0, ..., 0]

France = [0, 0, 0, 1, 0, 0, ..., 0]

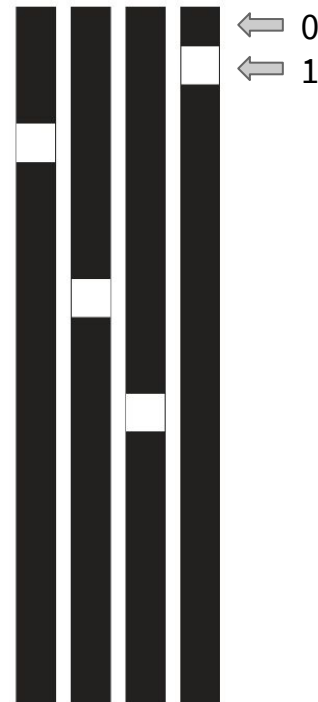
One-hot Hashing

- ◎ 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
- ◎ 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



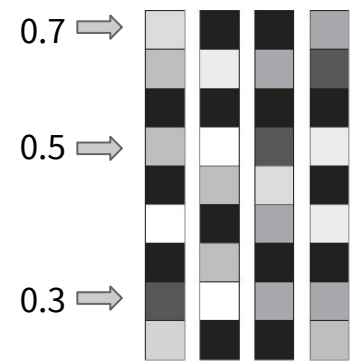
Word Embeddings

- 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

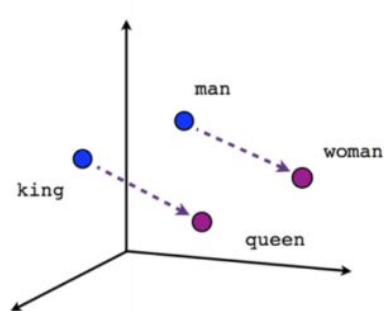
Word Embeddings

There are 2 ways to obtain word embeddings:

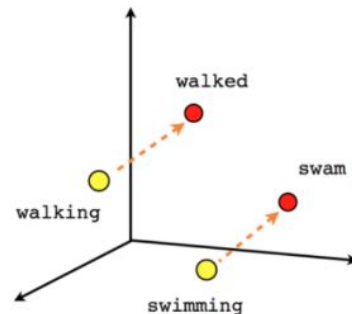
1. Learn word embeddings jointly with the main task you care about
Start with random word vectors and then learn word vectors in the same way you learn the weights of the network
2. Use pre-trained word embeddings
Load into your model word embeddings that were precomputed using a different machine-learning task than the one you're trying to solve

Learning Word Embeddings

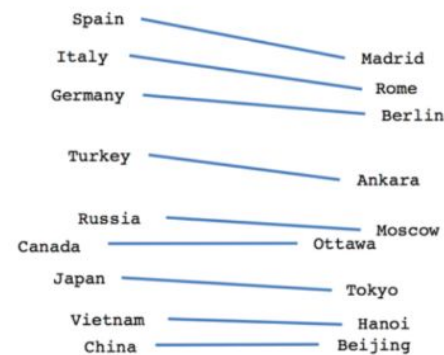
- It's easy to simply associate a vector with a word randomly - but this results in an embedding space without structure, and things like synonyms that could be interchangeable will have completely different embeddings
- This makes it difficult for a deep neural network to make sense of these representations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



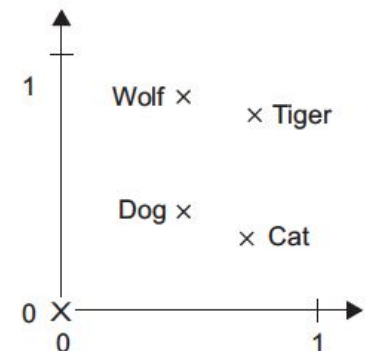
Male-Female



Verb tense



Country-Capital



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 contexts
 - A “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

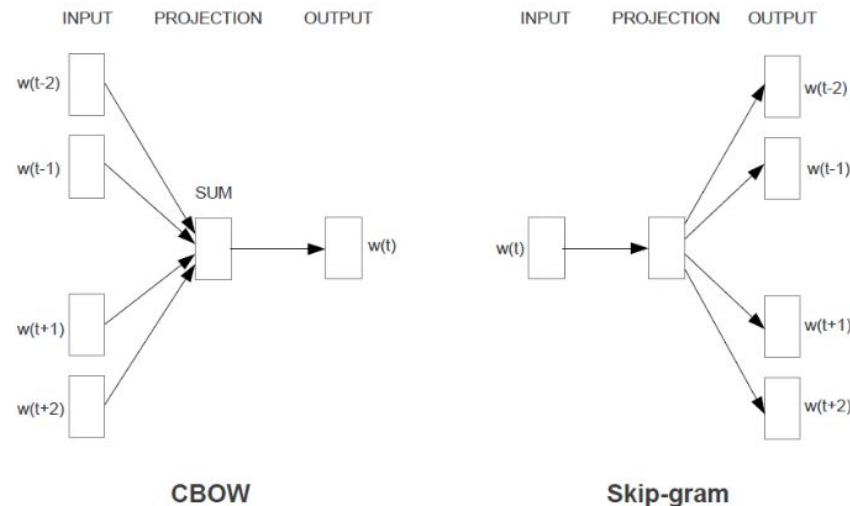


Figure 1: New model architectures. The CBOW architecture predicts the current word based on the context, and the Skip-gram predicts surrounding words given the current word.

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

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*:

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



3. *litoria*



4. *leptodactylidae*



5. *rana*

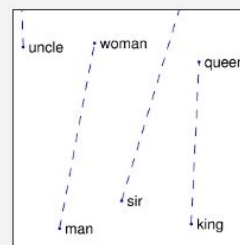


7. *eleutherodactylus*

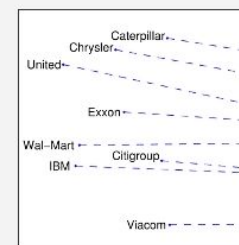
2. Linear substructures

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.

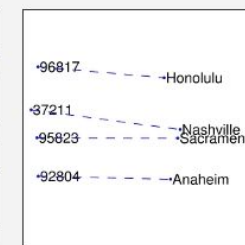
In order to capture in a quantitative way the nuance necessary to distinguish *man* from *woman*, it is necessary for a model to associate more than a single number to the word pair. A natural and simple candidate for an enlarged set of discriminative numbers is the vector difference between the two 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.



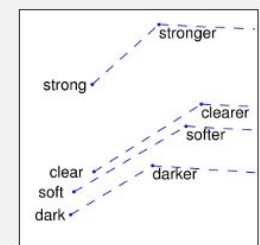
man - woman



company - ceo



city - zip code



comparative - superlative

Distributed Representations of Sentences and Documents

Quoc Le
Tomas Mikolov

Google Inc, 1600 Amphitheatre Parkway, Mountain View, CA 94043

QVL@GOOGLE.COM
TMIKOLOV@GOOGLE.COM

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

tags. The word order is lost, and thus different sentences can have exactly the same representation, as long as the same words are used. Even though bag-of-n-grams considers the word order in short context, it suffers from data sparsity and high dimensionality. Bag-of-words and bag-of-n-grams have very little sense about the semantics of the words or more formally the distances between the words. This means that words “powerful,” “strong” and “Paris” are equally distant despite the fact that semantically, “powerful” should be closer to “strong” than “Paris.”

Distributed Representations of Sentences and Documents

Quoc Le
Tomas Mikolov
Google Inc, 1600 Amphitheatre Parkway

Skip-Thought Vectors

Abstract

Many machine learning algorithms require input to be represented as a fixed-length vector. When it comes to texts, one common fixed-length feature is the bag-of-words. Despite their popularity, bag-of-words have two major weaknesses: they ignore the order of the words and they also ignore the meaning of the words. For example, “power” and “Paris” are equally distant. In

Ryan Kiros¹, Yukun Zhu¹, Ruslan Salakhutdinov^{1,2}, Richard S. Zemel^{1,2},
Antonio Torralba³, Raquel Urtasun¹, Sanja Fidler¹
University of Toronto¹
Canadian Institute for Advanced Research²
Massachusetts Institute of Technology³

Abstract

We describe an approach for unsupervised learning of a generic, distributed sentence encoder. Using the continuity of text from books, we train an encoder-decoder model that tries to reconstruct the surrounding sentences of an encoded passage. Sentences that share semantic and syntactic properties are thus mapped to similar vector representations. We next introduce a simple vocabulary expansion method to encode words that were not seen as part of training, allowing us to expand our vocabulary to a million words. After training our model, we extract and evaluate our vectors with linear models on 8 tasks: semantic relatedness, paraphrase detection, image-sentence ranking, question-type classification and 4 benchmark sentiment and subjectivity datasets. The end result is an off-the-shelf encoder that can produce highly generic sentence representations that are robust and perform well in practice.

Distributed Representations of Sentences and Documents

Quoc Le
Tomas Mikolov
Google Inc, 1600 Amphitheatre Parkway

Skip-Thought Vectors

Abstract

Many machine learning algorithms require input to be represented as a fixed-length vector. When it comes to texts, a common fixed-length feature is the bag-of-words. Despite their popularity, bag-of-words have two major weaknesses: they ignore the ordering of the words and they also ignore the meaning of the words. For example, "power" and "Paris" are equally distant. In

dna2vec: Consistent vector representations of variable-length k-mers

Patrick Ng
ppn3@cs.cornell.edu

Abstract

One of the ubiquitous representations of long DNA sequence is dividing it into shorter 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.

*2vec

Distributed Representations of Sentences and Documents

EMBED ALL THE THINGS

Quoc Le
Tomas Mikolov
Google Inc, 1600 Amphitheatre Parkway

Abstract

Many machine learning algorithms require input to be represented as a fixed-length vector. When it comes to texts, a common fixed-length feature is the bag-of-words. Despite their popularity, bag-of-words have two major weaknesses: they ignore the order of the words and they also ignore the meaning of the words. For example, "power" and "Paris" are equally distant. In



representations of
 k -mers

edu

One of the ubiquitous representations of long DNA sequences is dividing it into shorter 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

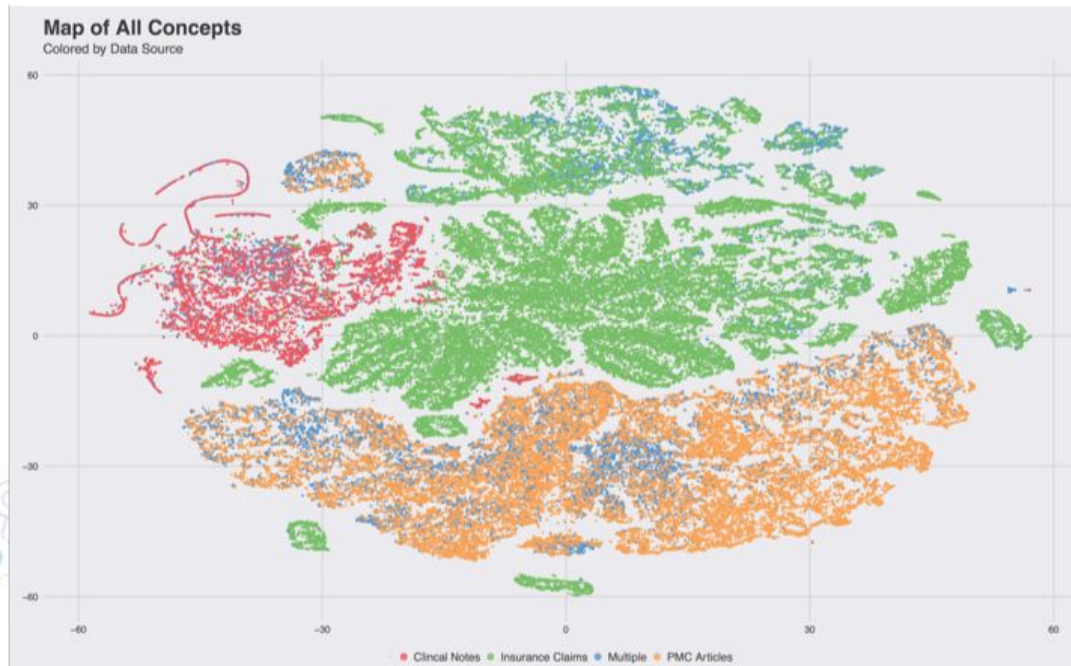
Inbar Fried
University of North Carolina

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



A background network diagram consisting of numerous nodes and edges. The nodes are represented by small circles, some of which are solid grey and others are hollow with a grey outline. The edges are thin grey lines connecting the nodes, forming a complex, interconnected web. The overall pattern is dense and covers the entire slide area.

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 vector

- ◎ 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

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



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

8-dimensional embeddings - one for each word

Size of vocabulary

Length of sequence

Note that we aren't fitting a RNN yet - this is a 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
- ⦿ 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: (25000, 100)
Shape of label tensor: (25000,)

```

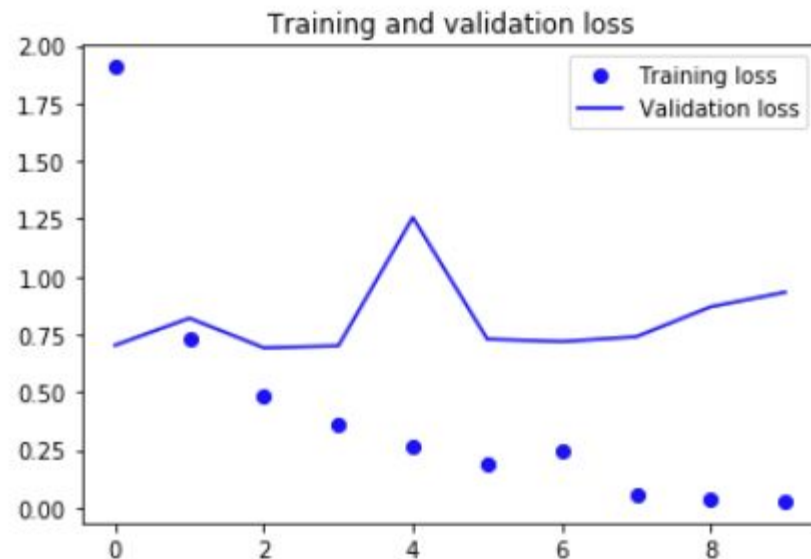
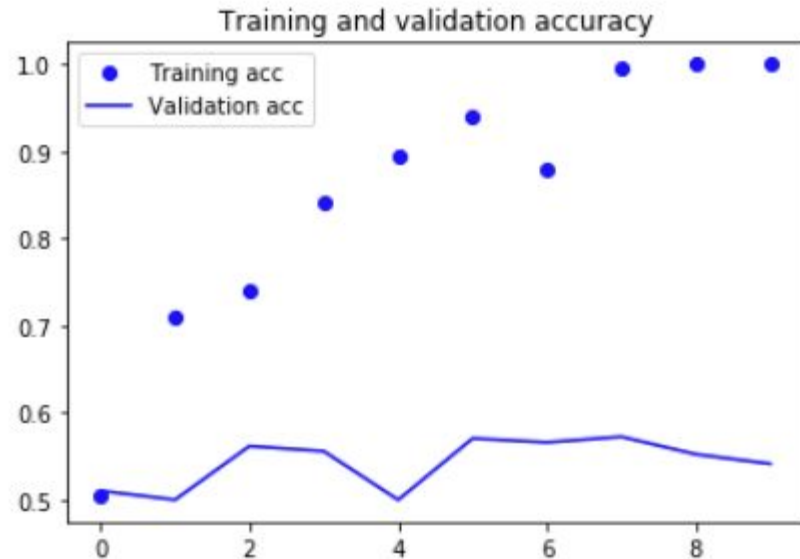
```
1 glove_dir = '/Users/heathermattie/Desktop/glove/'
2
3 embeddings_index = {}
4 f = open(os.path.join(glove_dir, 'glove.6B.100d.txt'))
5 for line in f:
6     values = line.split()
7     word = values[0]
8     coefs = np.asarray(values[1:], dtype='float32')
9     embeddings_index[word] = coefs
10 f.close()
11
12 print('Found %s word vectors.' % len(embeddings_index))
```

Found 400000 word vectors.

```
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%.



IMDb Example - Simple RNN

- 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.

```
from keras.models import Sequential
from keras.layers import Embedding, SimpleRNN

model = Sequential()
model.add(Embedding(10000, 32))
model.add(SimpleRNN(32))
model.summary()
```

Layer (type)	Output Shape	Param #
embedding_1 (Embedding)	(None, None, 32)	320000
simple_rnn_1 (SimpleRNN)	(None, 32)	2080
Total params: 322,080		
Trainable params: 322,080		
Non-trainable params: 0		

```
model = Sequential()
model.add(Embedding(10000, 32))
model.add(SimpleRNN(32, return_sequences=True))
model.summary()
```

Layer (type)	Output Shape	Param #
embedding_2 (Embedding)	(None, None, 32)	320000
simple_rnn_2 (SimpleRNN)	(None, None, 32)	2080
Total params: 322,080		
Trainable params: 322,080		
Non-trainable params: 0		

IMDb Example - Simple RNN

- 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.

```
from keras.models import Sequential
from keras.layers import Embedding, SimpleRNN

model = Sequential()
model.add(Embedding(10000, 32))
model.add(SimpleRNN(32))
model.summary()
```

Layer (type)	Output Shape	Param #
embedding_1 (Embedding)	(None, None, 32)	320000
simple_rnn_1 (SimpleRNN)	(None, 32)	2080
Total params: 322,080		
Trainable params: 322,080		
Non-trainable params: 0		

```
model = Sequential()
model.add(Embedding(10000, 32))
model.add(SimpleRNN(32, return_sequences=True))
model.summary()
```

Layer (type)	Output Shape	Param #
embedding_2 (Embedding)	(None, None, 32)	320000
simple_rnn_2 (SimpleRNN)	(None, None, 32)	2080
Total params: 322,080		
Trainable params: 322,080		
Non-trainable params: 0		

IMDb Example - Simple RNN

- 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.

```
from keras.models import Sequential
from keras.layers import Embedding, SimpleRNN
```

```
model = Sequential()
model.add(Embedding(10000, 32))
model.add(SimpleRNN(32))
model.summary()
```

Layer (type)	Output Shape	Param #
embedding_1 (Embedding)	(None, None, 32)	320000
simple_rnn_1 (SimpleRNN)	(None, 32)	2080
Total params: 322,080		
Trainable params: 322,080		
Non-trainable params: 0		

```
model = Sequential()
model.add(Embedding(10000, 32))
model.add(SimpleRNN(32, return_sequences=True))
model.summary()
```

Layer (type)	Output Shape	Param #
embedding_2 (Embedding)	(None, None, 32)	320000
simple_rnn_2 (SimpleRNN)	(None, None, 32)	2080
Total params: 322,080		
Trainable params: 322,080		
Non-trainable params: 0		

IMDb Example - Simple RNN

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**:

```
model = Sequential()
model.add(Embedding(10000, 32))
model.add(SimpleRNN(32, return_sequences=True))
model.add(SimpleRNN(32, return_sequences=True))
model.add(SimpleRNN(32, return_sequences=True))
model.add(SimpleRNN(32)) # This last layer only returns the last outputs.
model.summary()
```

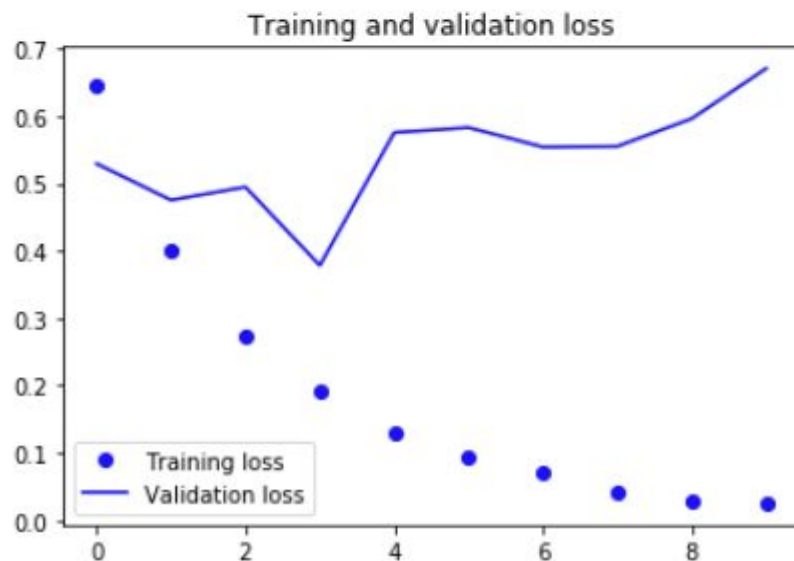
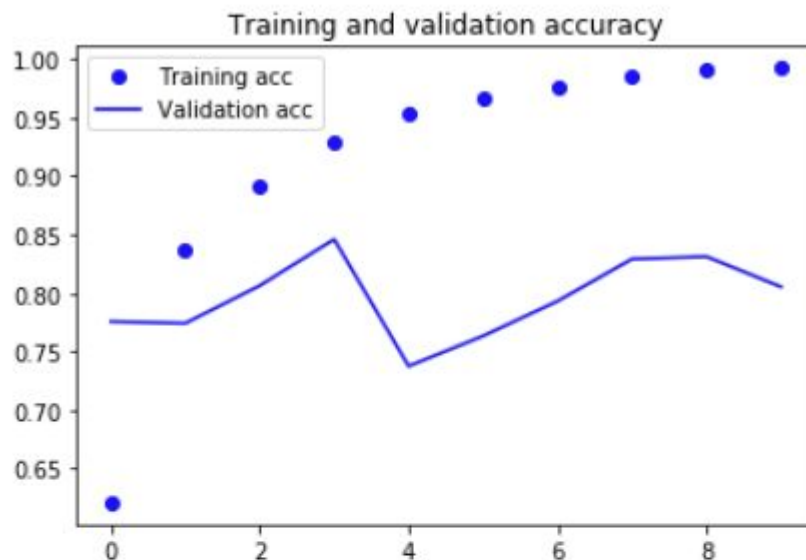
Layer (type)	Output Shape	Param #
embedding_3 (Embedding)	(None, None, 32)	320000
simple_rnn_3 (SimpleRNN)	(None, None, 32)	2080
simple_rnn_4 (SimpleRNN)	(None, None, 32)	2080
simple_rnn_5 (SimpleRNN)	(None, None, 32)	2080
simple_rnn_6 (SimpleRNN)	(None, 32)	2080
Total params: 328,320		
Trainable params: 328,320		
Non-trainable params: 0		

IMDb Example - Simple RNN

```
1 from keras.layers import Dense
2 from keras.models import Sequential
3 from keras.layers import Embedding, SimpleRNN
4
5 model = Sequential()
6 model.add(Embedding(max_features, 32))
7 model.add(SimpleRNN(32))
8 model.add(Dense(1, activation='sigmoid'))
9
10 model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])
11 history = model.fit(input_train, y_train,
12                     epochs=10,
13                     batch_size=128,
14                     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 500 words rather than 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 week.





Problems with RNNs

Problems with RNNs

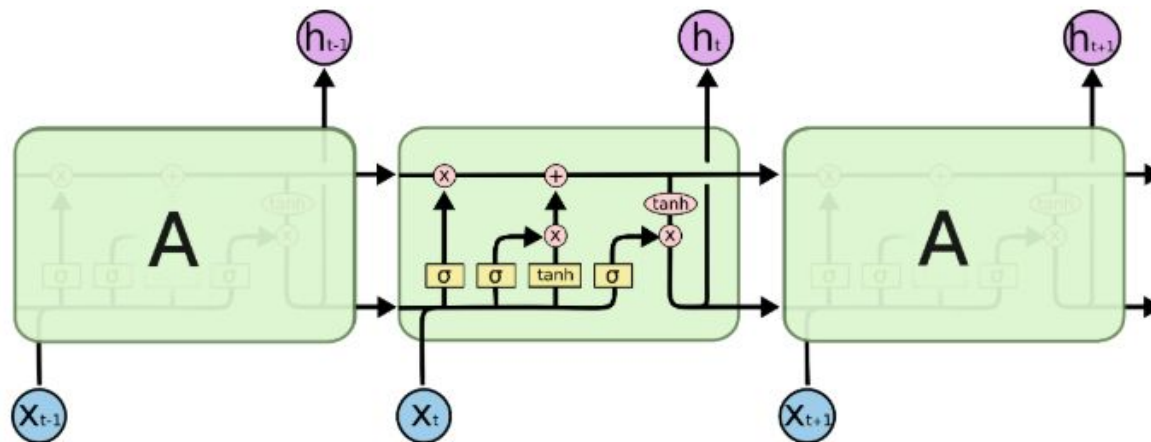
- ◎ Recall the formula for a generic RNN:

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

- ◎ What happens for really long sequences during backprop?
 - You multiply by the matrix U repeatedly
 - Largest eigenvalue > 1 , gradient \rightarrow infinity (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 next week

