BST 261: Data Science II
Lecture 11
Recurrent Neural Networks (RNNs) Continued

Heather Mattie
Harvard T.H. Chan School of Public Health
Spring 2 2020

Recipe of the Day!

Pink Champagne Cake

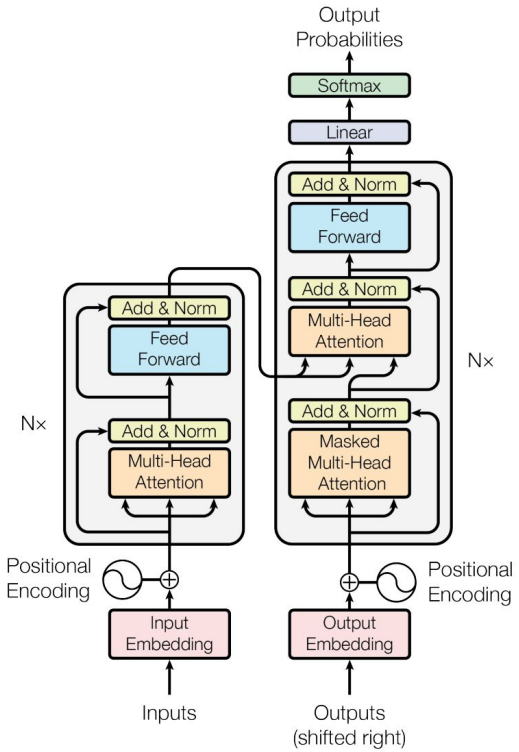
Paper Presentations

Attention is all you need

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin

Presentation by Beau Coker

Goal: explain the transformer



Why do we care?

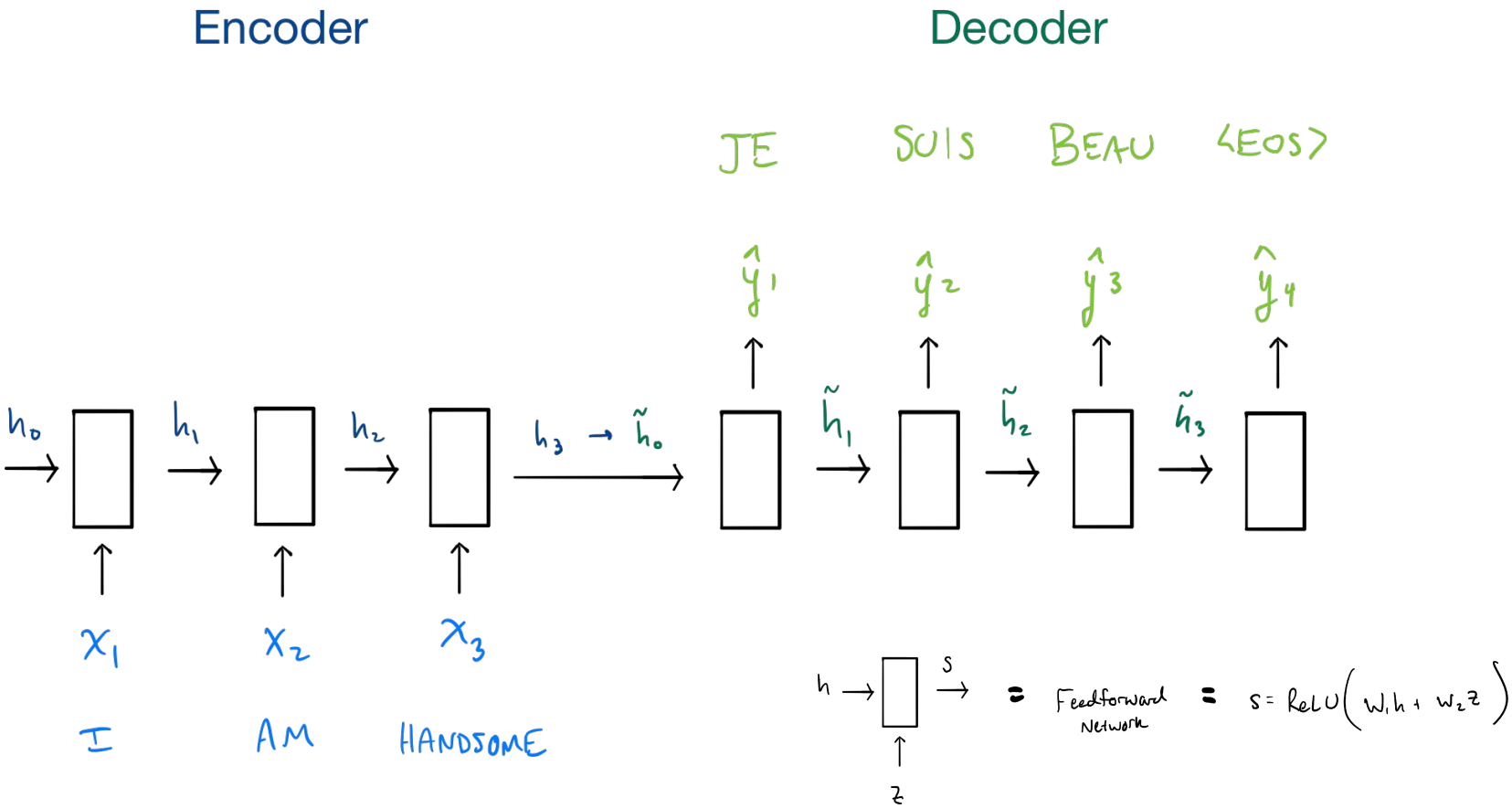
- Computational: recurrent structure fundamentally can’t be parallelized but the transformer can
- Results: higher BLEU scores on English-German and English-French

Outline

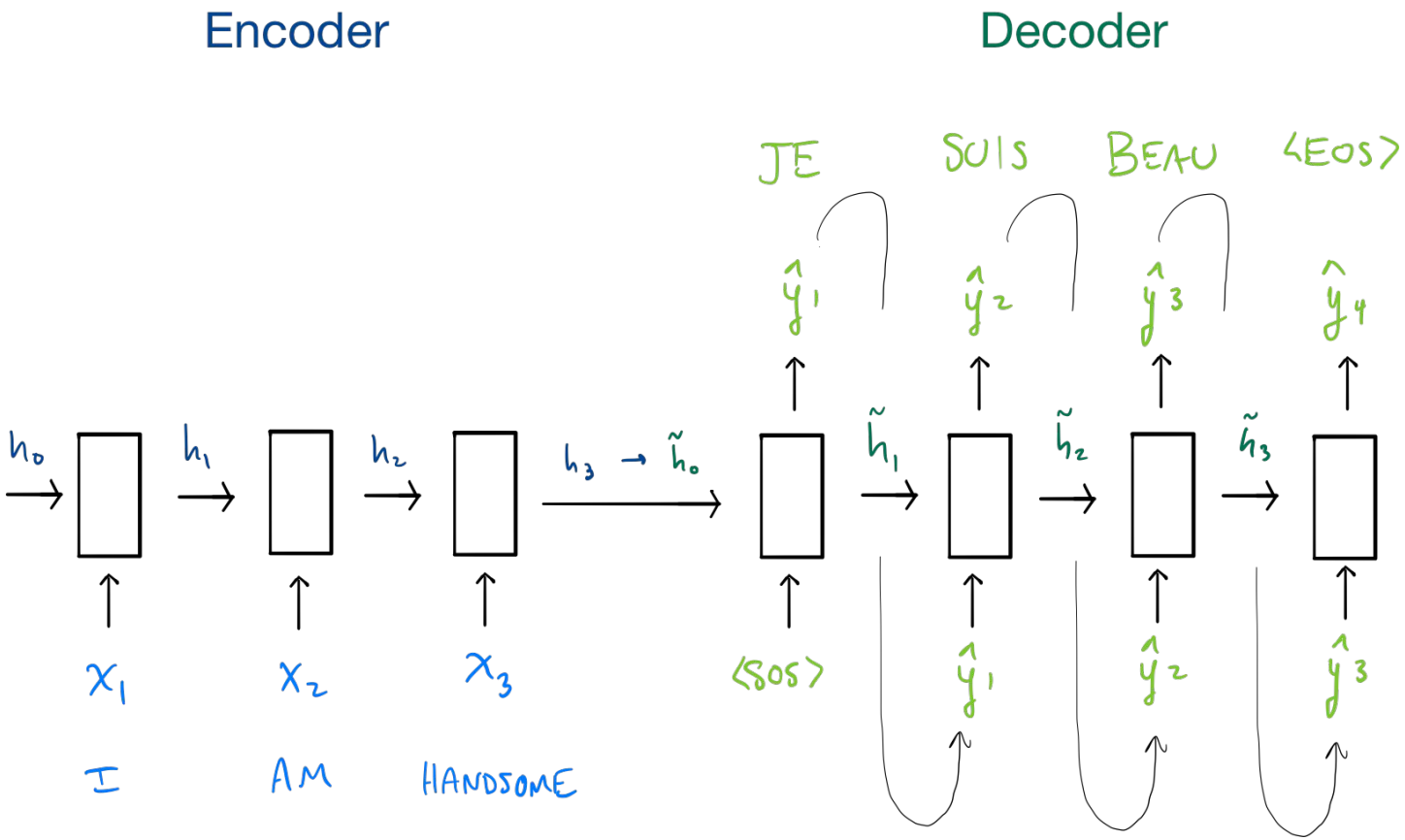
A few approaches to neural machine translation:

- RNN encoder-decoder ← last lecture
- RNN encoder-decoder with attention
- Transformer (no RNN, attention only) ← “Attention is all you need”

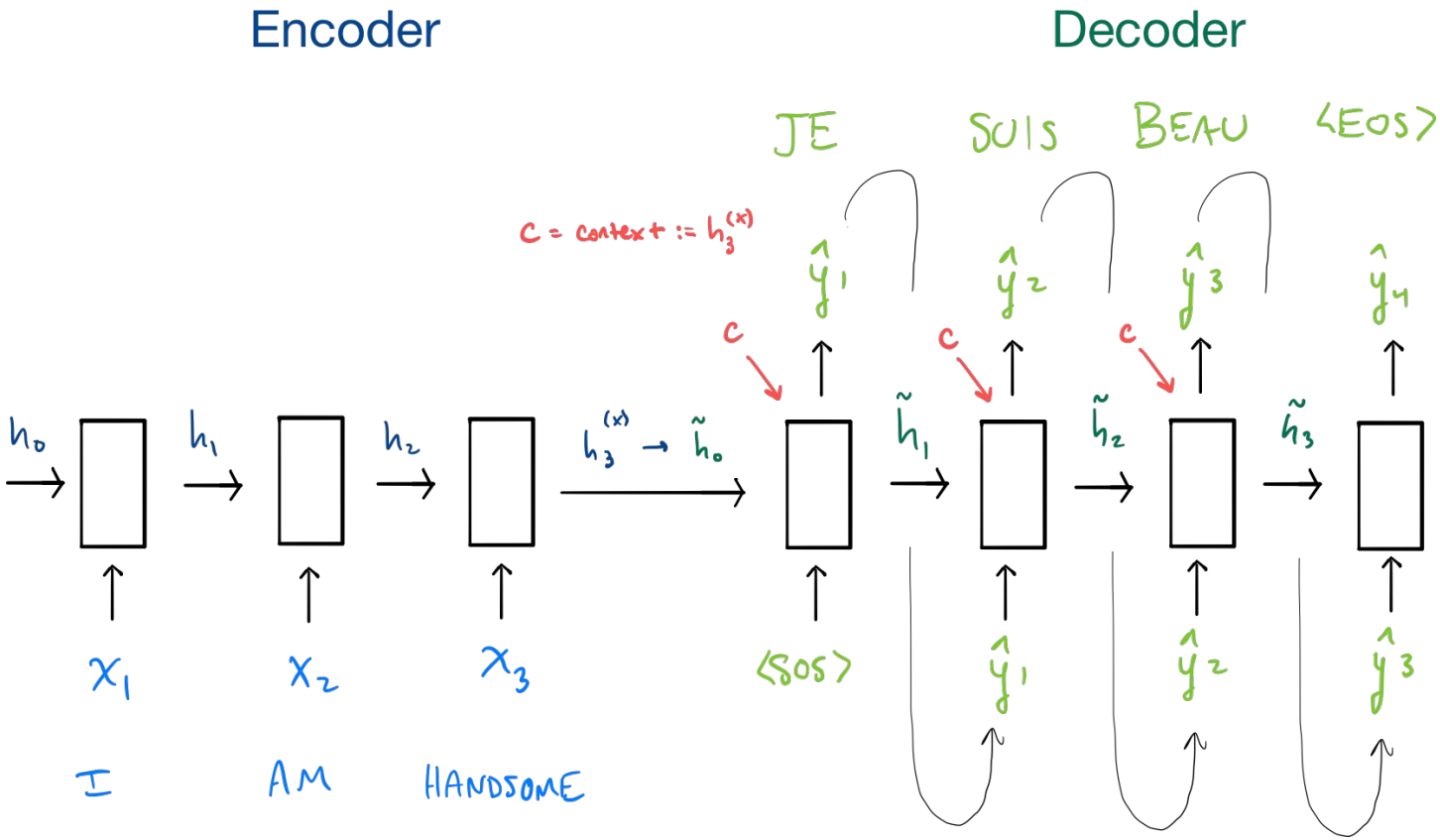
RNN encoder-decoder



RNN encoder-decoder



RNN encoder-decoder



RNN encoder-decoder with attention

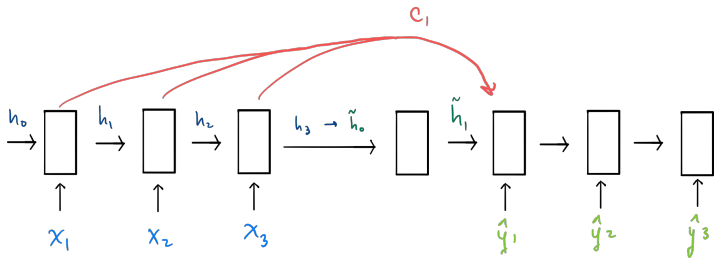
Idea of attention: replace **global** context with **local** context

$$c_{t_y} = \sum_{t_x=1}^{3} s(t_y, t_x) \underbrace{h_{t_x}}_{\text{"value"}}$$

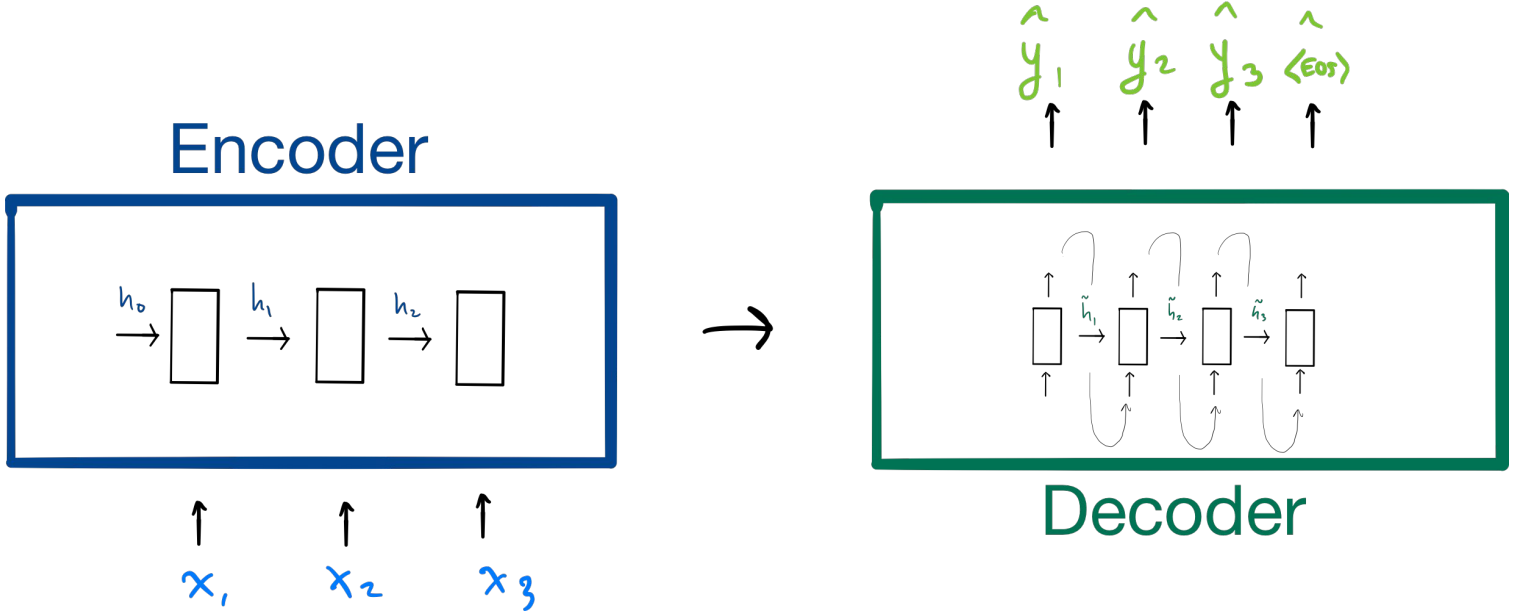
$$= f_{\text{similarity}}\left(\tilde{h}_{t_{y-1}}, h_{t_x}\right)$$

$$:= \text{Softmax}\left(\tilde{h}_{t_{y-1}}^{T} h_{t_x}\right)$$

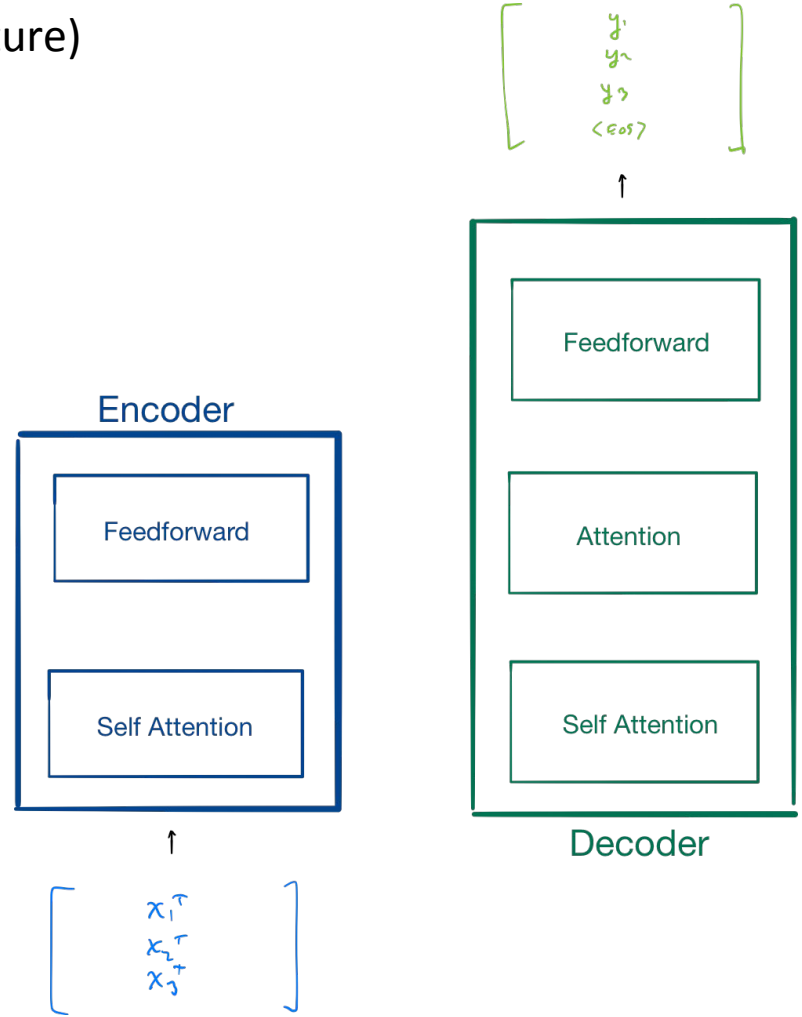
"query" "key"



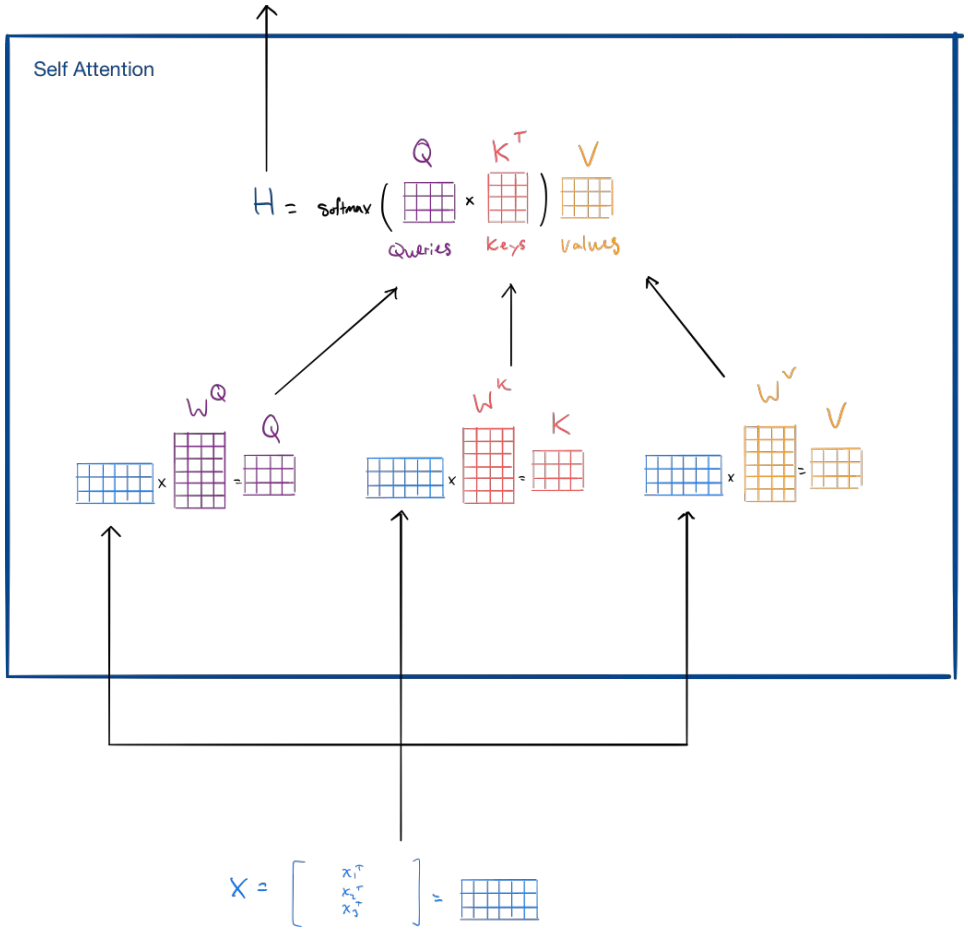
RNN encoder-decoder with attention (basic structure)



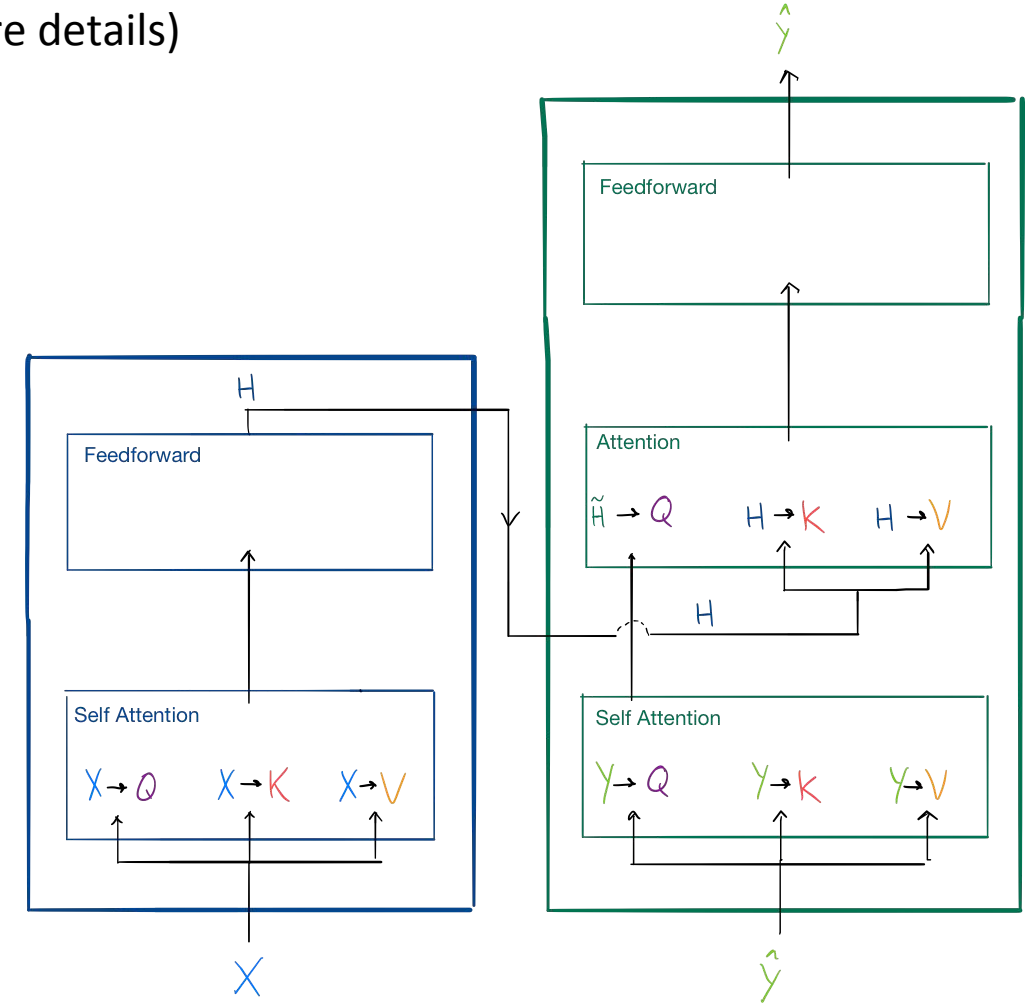
Transformer (basic structure)



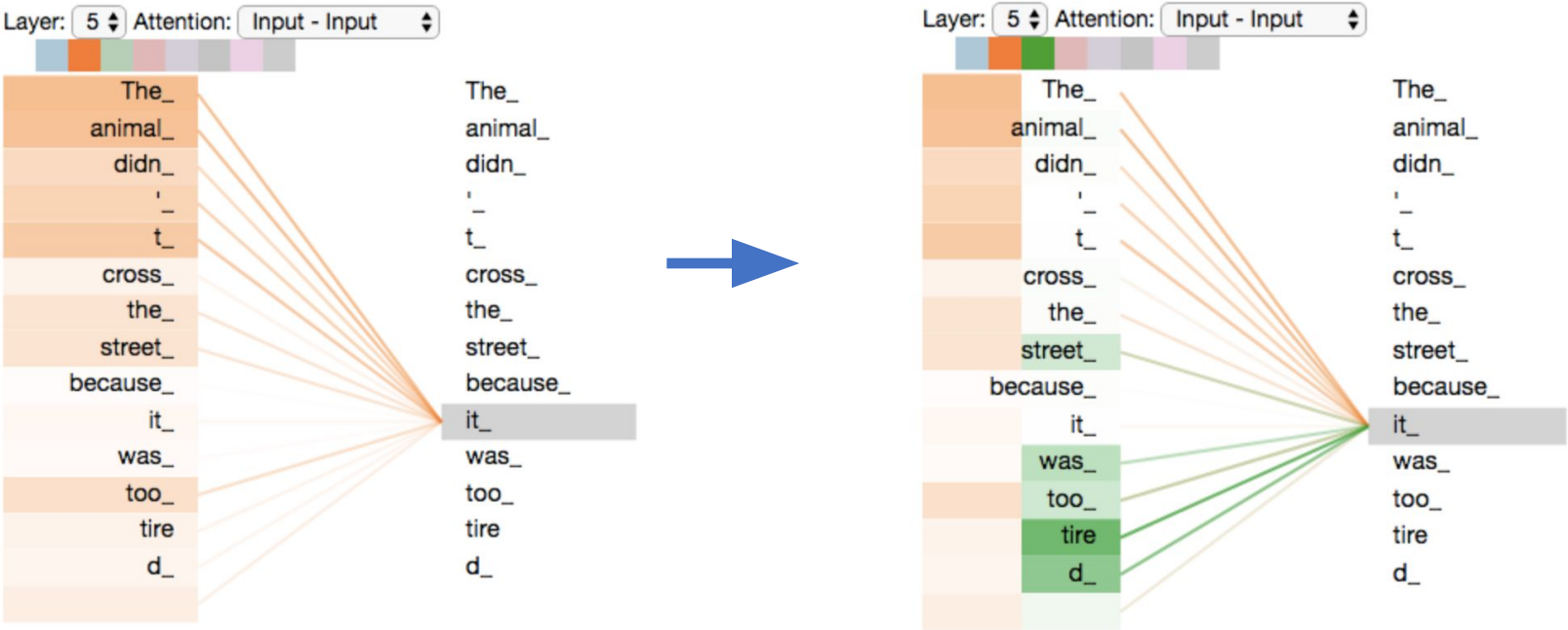
Attention block



Transformer (more details)

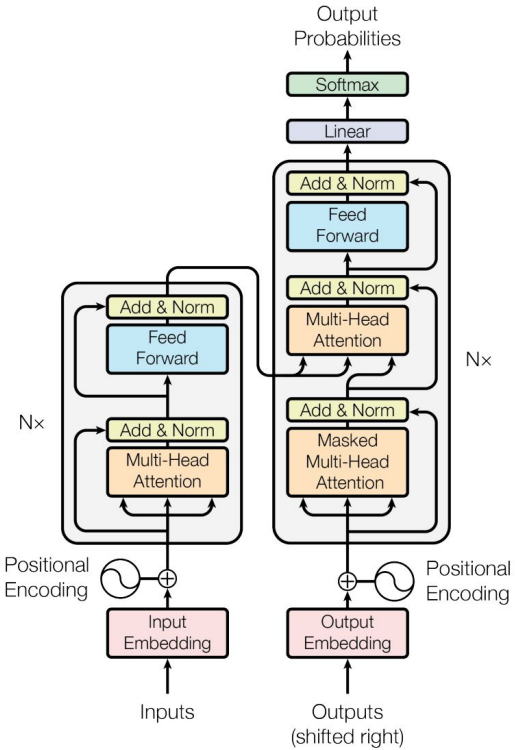


Attention □ multi-headed attention



http://jalammar.github.io/illustrated-transformer/

Here's the full transformer



Things I simplified

- There are 6 encoders and decoders stacked on top of each other
- Each attention block has 8 “heads” (sets of Q, K, V, and H matrices)
- Sinusoidal “positional encodings” are added before attention block to encode order of words

DEEP CONTEXTUALIZED WORD REPRESENTATIONS

Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, Luke Zettlemoyer

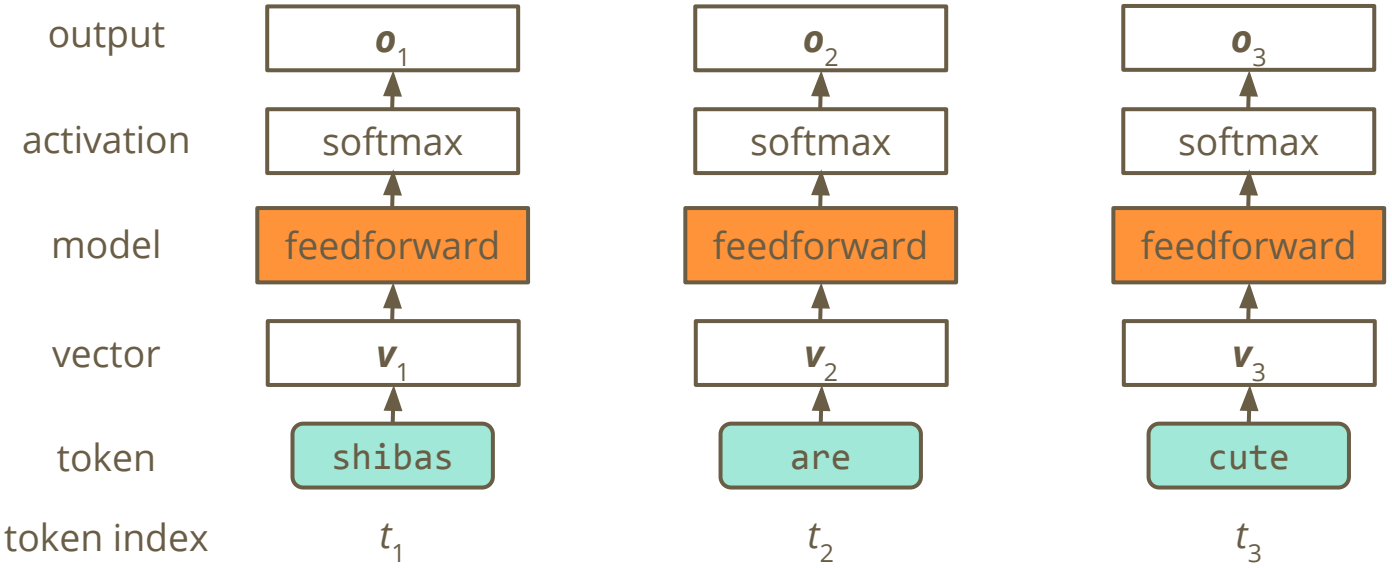
BST 261 PRESENTATION

Motivation

- Context has always been a challenging concept in NLP
  - Syntax
  - Polysemy

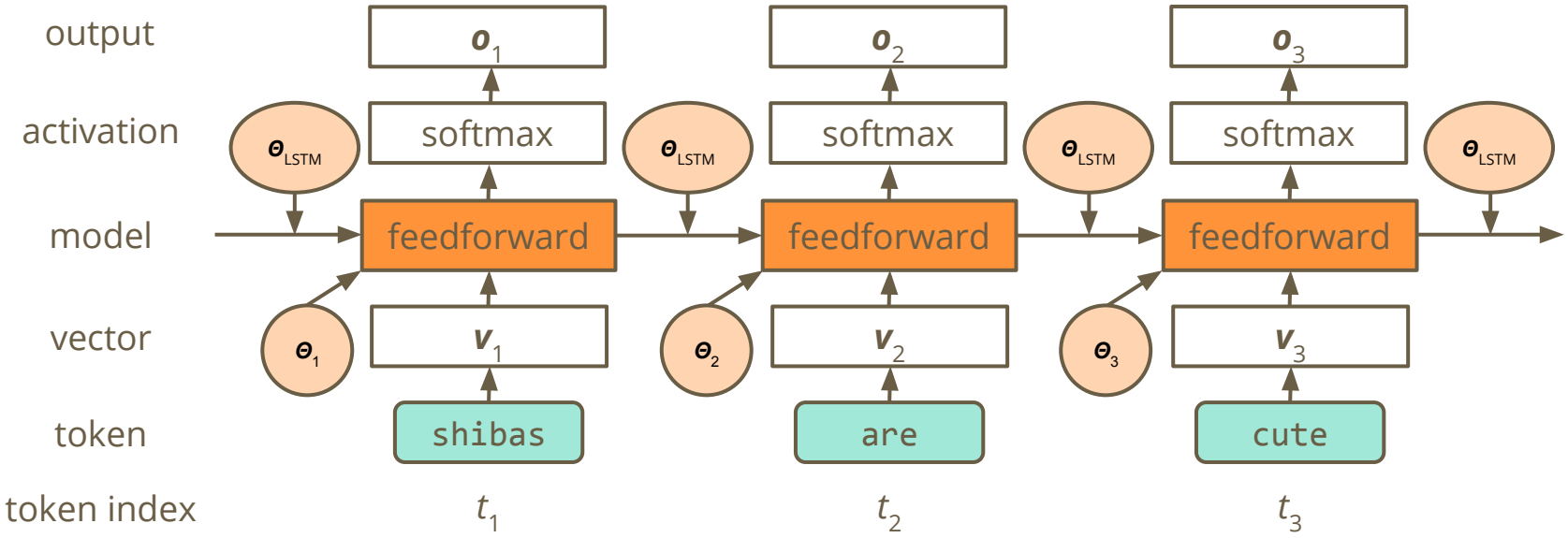
Motivation

- Simple feedforward models do not work well with deciphering context because they do not model information from sequence data



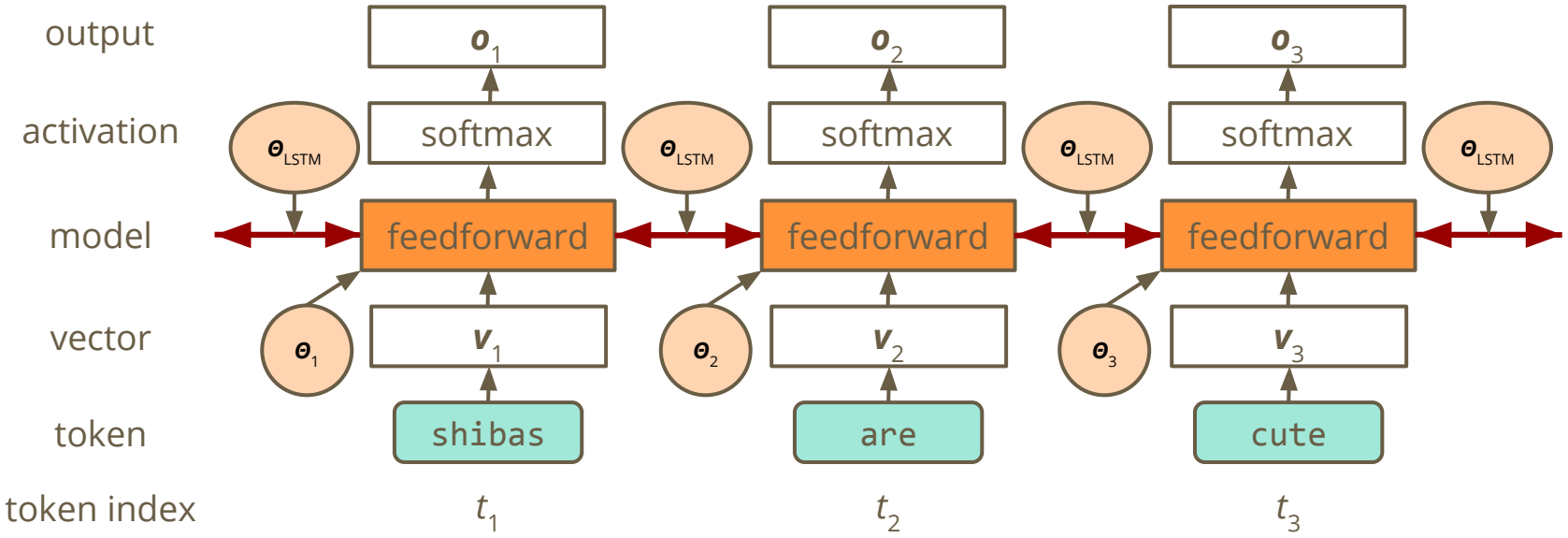
LSTM: long short-term memory

- Type of RNN briefly covered in Friday's class
- Models information through depth and time

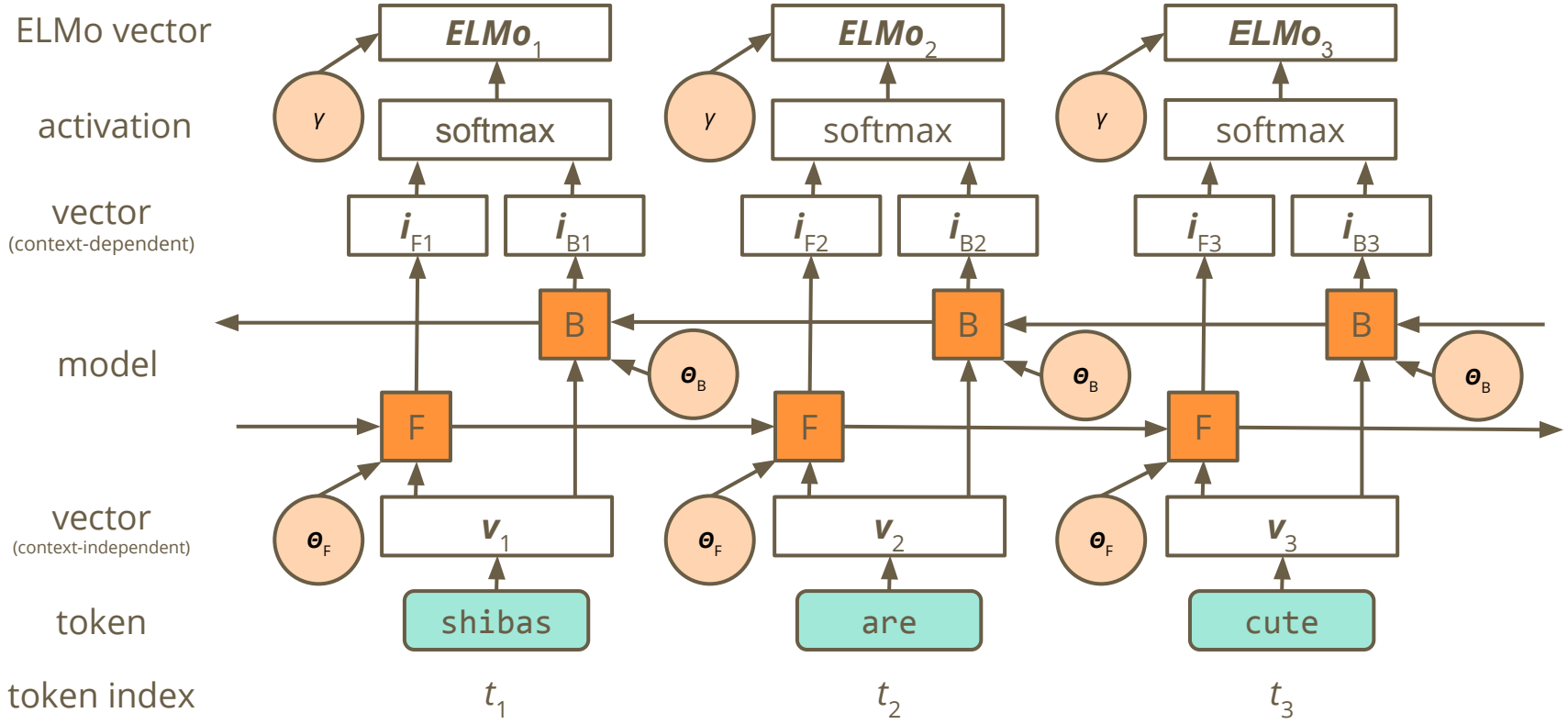


LSTM: long short-term memory

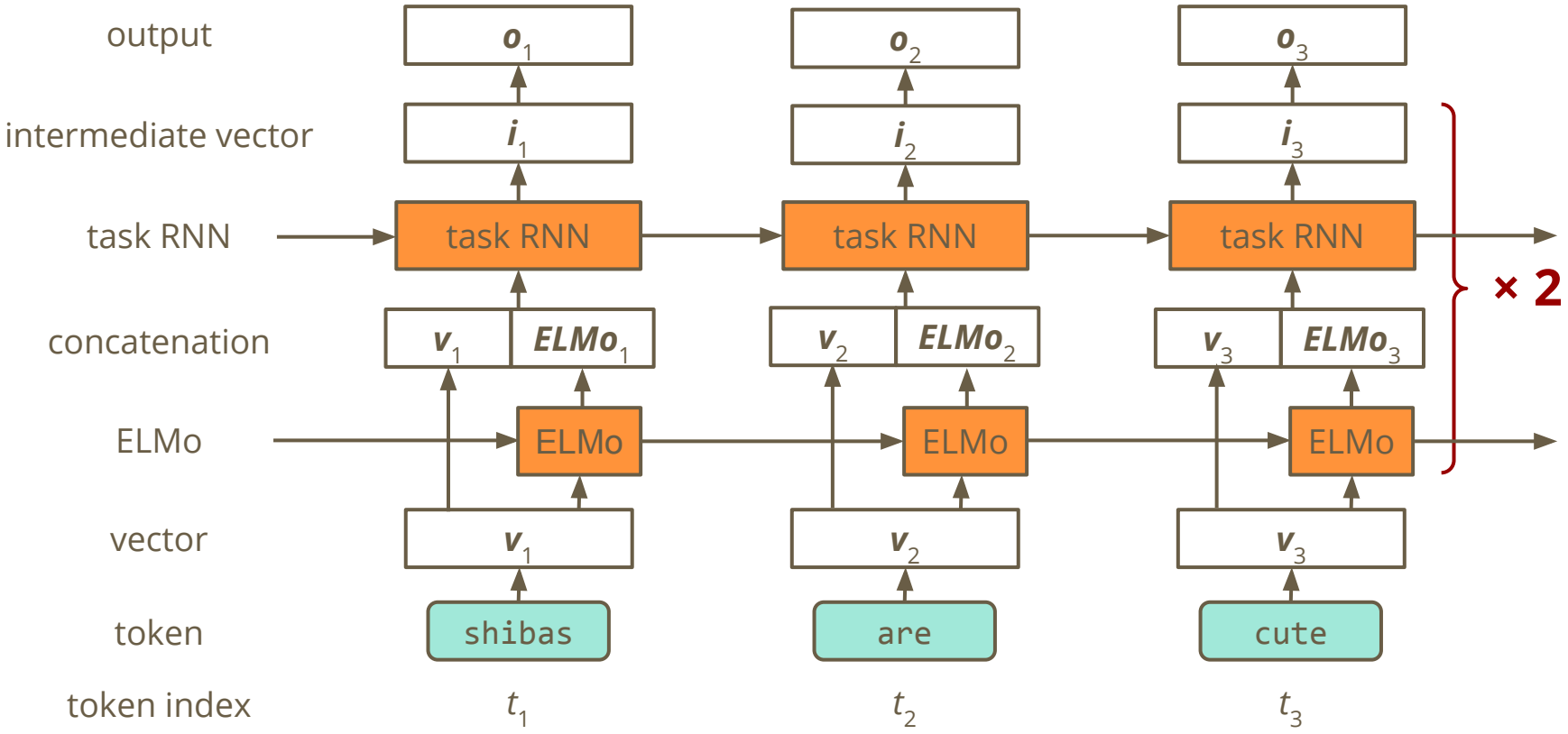
- Can go both forwards and backwards
- Bidirectional LSTM (biLSTM): commonly used at the time



ELMo: embeddings from language models



Deep contextualized word representations



Advantages of ELMo

- Instead of representing each word only once, ELMo does it three times, preserving contextual information that may otherwise be lost
- Each ELMo representation is a function of the entire input sentence
- Character convolutions in the biLSTMs capture even more contextual information

Performance on test-set

Task	Baseline performance from SOTA model	Baseline + ELMo performance	Increase (absolute / relative)
Question answering	81.1	85.8	4.7 / 24.9%
Textual entailment	88.0	88.7 ± 0.17	0.7 / 5.8%
Semantic role labeling	81.4	84.6	3.2 / 17.2%
Coreference resolution	67.2	70.4	3.2 / 9.8%
Named entity extraction	90.15	92.22 ± 0.10	2.06 / 21%
Sentiment analysis	51.4	54.7 ± 0.5	3.3 / 6.8%

References

- Peters, M. E., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., Zettlemoyer, L. (2018). Deep contextualized word representations. *Proceedings of NAACL-HLT*. 2227-2237.

That's all, folks!

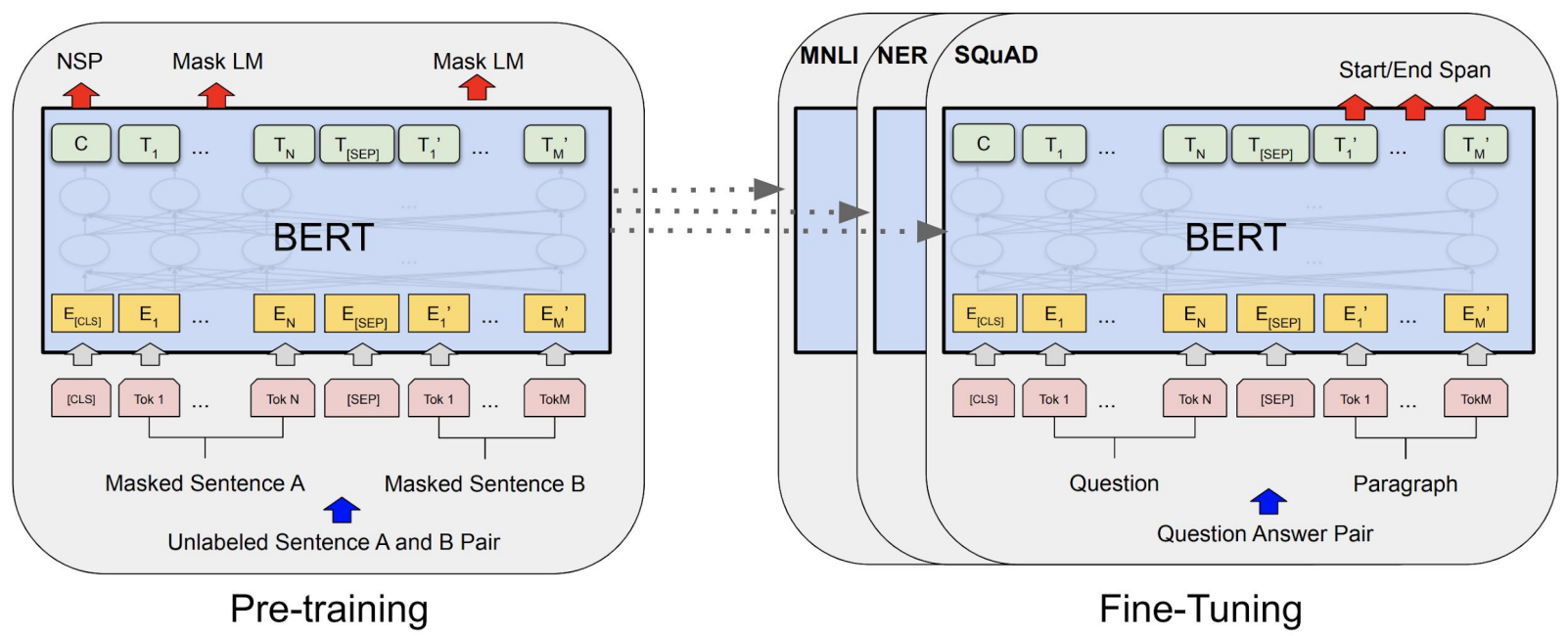
BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding

Jacob Devlin
Google AI Language

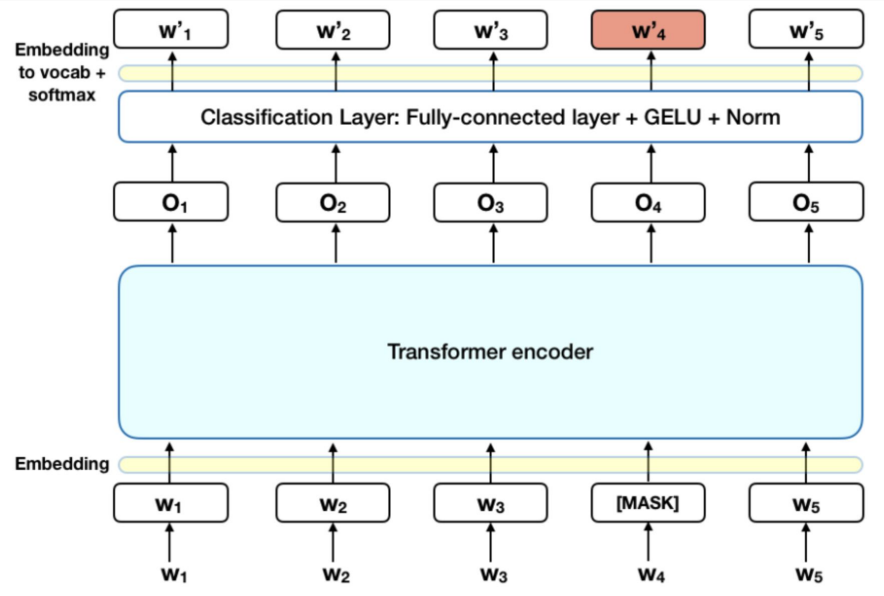
Jerri Zhang
BST 261

BERT:

Bidirectional **E**ncoder **R**epresentations from **T**ransformers



Pre-Training: Masked Language Modeling



- Randomly select 15% of the words in each sequence are replaced with a [MASK] token.
- The model then attempts to predict the original value of the masked words, based on the context provided by the other, non-masked, words in the sequence.

Pre-Training: Masked Language Modeling

Training the language model in BERT is done by predicting 15% of the tokens in the input, that were randomly picked.

Out of the 15% of the tokens:

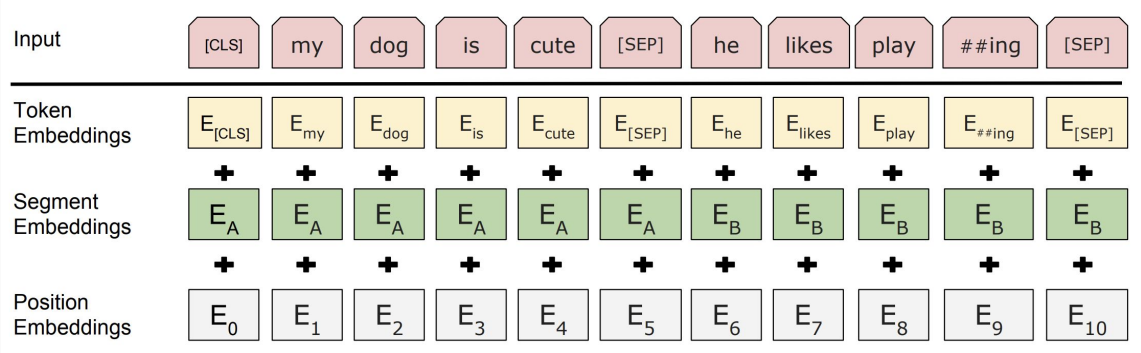
- 80% are replaced with a “[MASK]” token — e.g. my dog is hairy → my dog is [MASK]
- 10% with a random word — e.g. my dog is hairy → my dog is apple
- 10% use the original word — e.g. my dog is hairy → my dog is hairy

Pre-Training: Next Sentence Prediction (NSP)

- Many important downstream tasks such as Question Answering (QA) and Natural Language Inference (NLI) are based on understanding the relationship between two sentences.

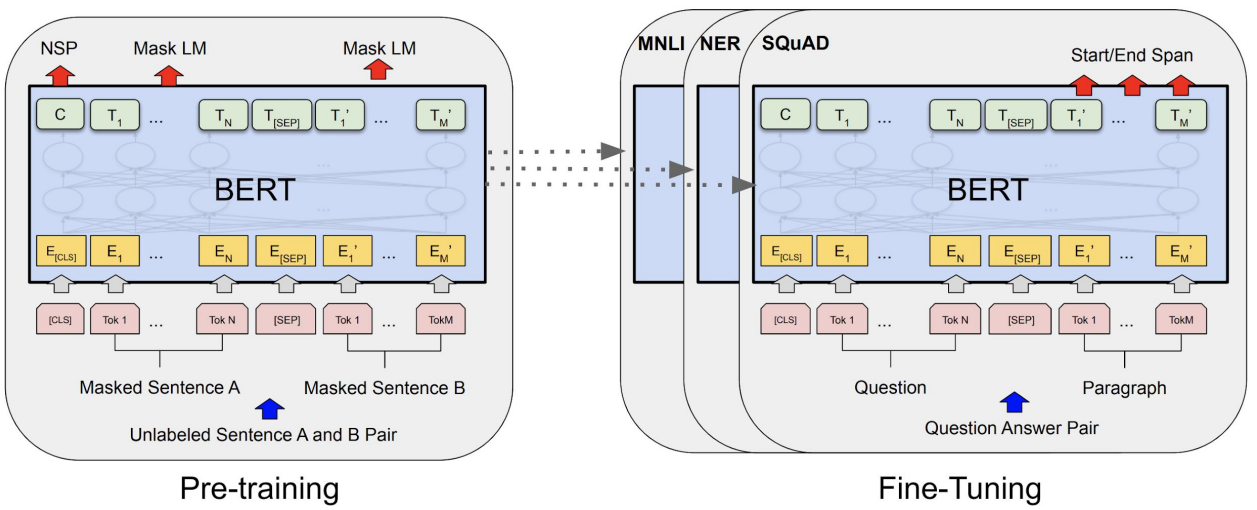
- In order to train a model that understands sentence relationships, we pre-train for a binarized next sentence prediction task.

Pre-Training: Next Sentence Prediction (NSP)



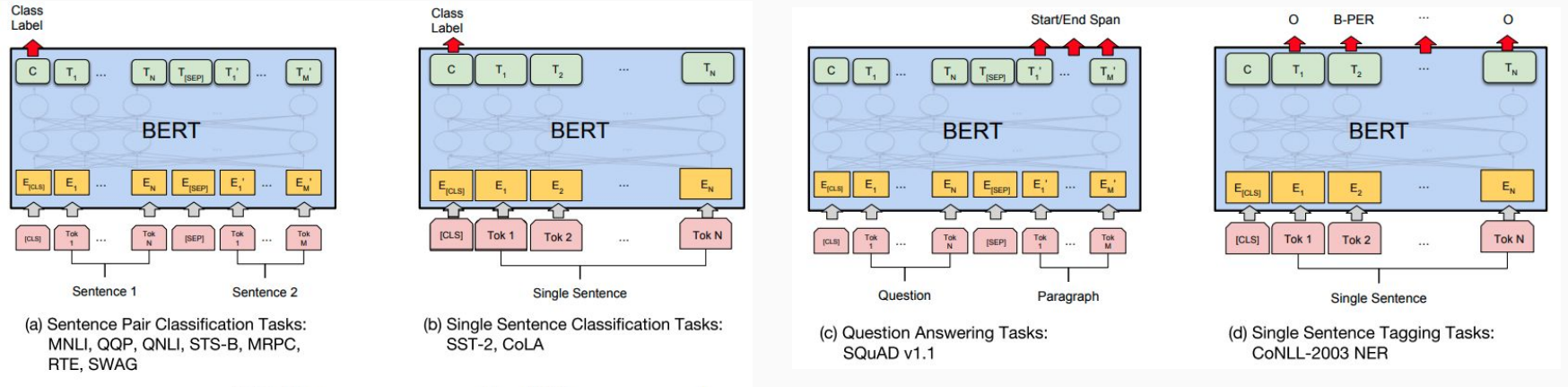
- 50% of the sentences were paired with actual adjacent sentences in the corpus and 50% of them were paired with sentences picked randomly from the corpus.
- A [CLS] token is inserted at the beginning of the first sentence and a [SEP] token is inserted at the end of each sentence.
- A sentence embedding indicating Sentence A or Sentence B is added to each token.

RECAP:
Bidirectional **E**ncoder **R**epresentations from **T**ransformers



- When training the BERT model, Masked Language Modeling and Next Sentence Prediction are trained together, with the goal of minimizing the combined loss function of the two strategies.

Fine-Tuning



- For each task, simply plug in the task specific inputs and outputs into BERT and fine tune all the parameters end-to-end.

Experiments

Present BERT fine-tuning results on 11 NLP tasks:

- General Language Understanding Evaluation (GLUE) benchmark
    - MNLI, QQP, QNLI, SST-2, CoLA, STS-B, MRPC, RTE
- Stanford Question Answering Dataset (SQuAD) v1.1, v2.0
- Situations With Adversarial Generations (SWAG)

BERT advances the **state of the art** for 11 NLP tasks.

Effect of Model Size

	Total parameters	Performance
BERT_base	110M (similar to OpenAI)	Better than OpenAI
BERT_large	340M	Better than OpenAI and BERT_base

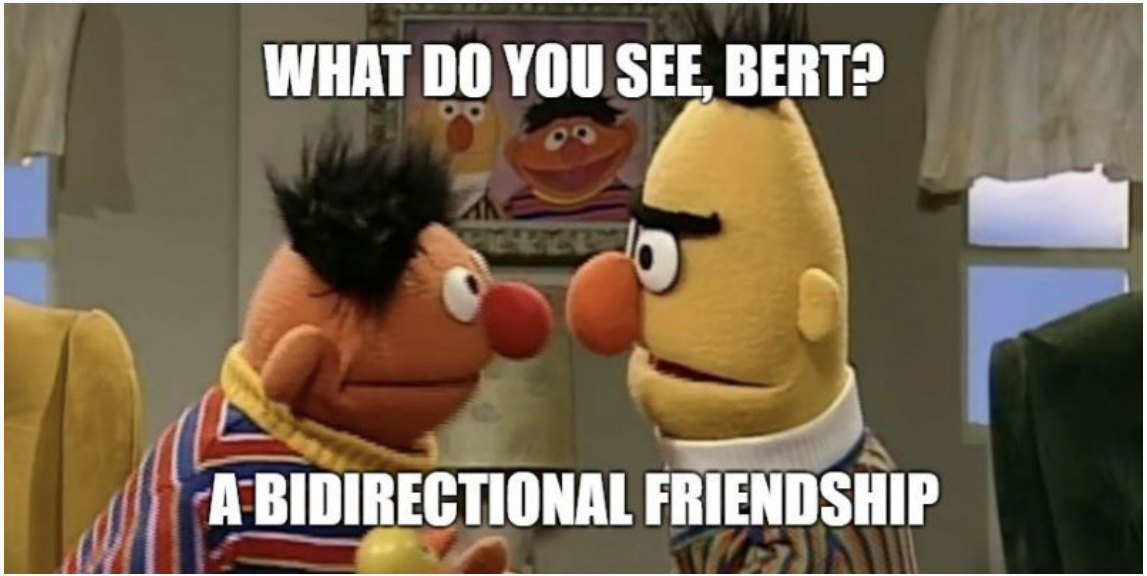
- Both BERT_base and BERT_large outperform ALL systems on ALL tasks by a substantial margin.
- BERT_large significantly outperforms BERT_base across all tasks, especially those with very little training data.

Takeaways

- BERT outperforms previous NLP models on many language benchmark tasks.
- BERT's fast fine-tuning allows a wide range of downstream tasks and practical applications in the future.
- BERT's bidirectional approach outperforms left-to-right approach.
- Model size matters. Larger model leads to large improvements on very small scale tasks, provided that the model has been sufficiently pre-trained.

REFERENCES:
BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. Jacob Devlin et al., Google AI Language. Oct 2018.
BERT Explained: State of the art language model for NLP

Thank you!



RNNs Continued

Recap from last week

- 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)

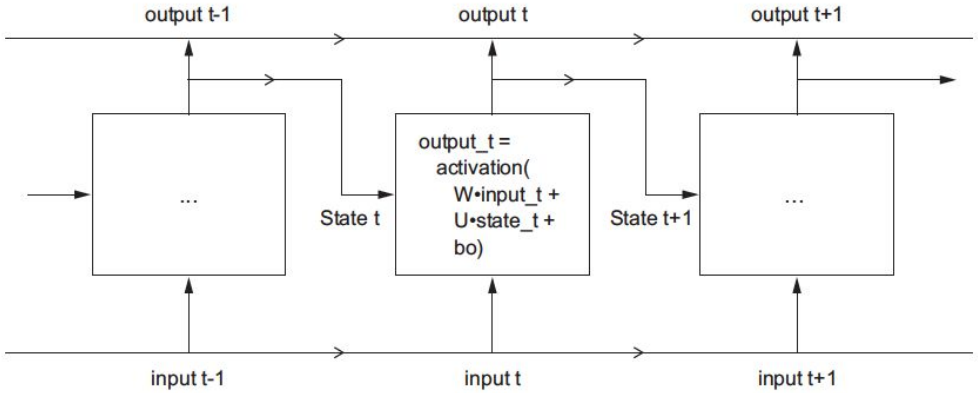
MLPs ⟹ RNNs

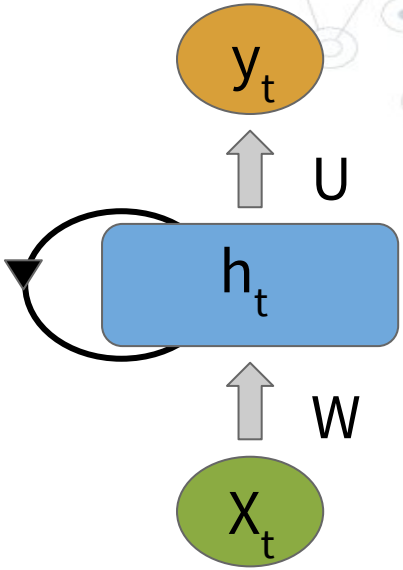
- RNNs are a natural extension of MLPs
- MLPs are “memoryless”, but often we need knowledge of the past sequence of events to predict the future

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

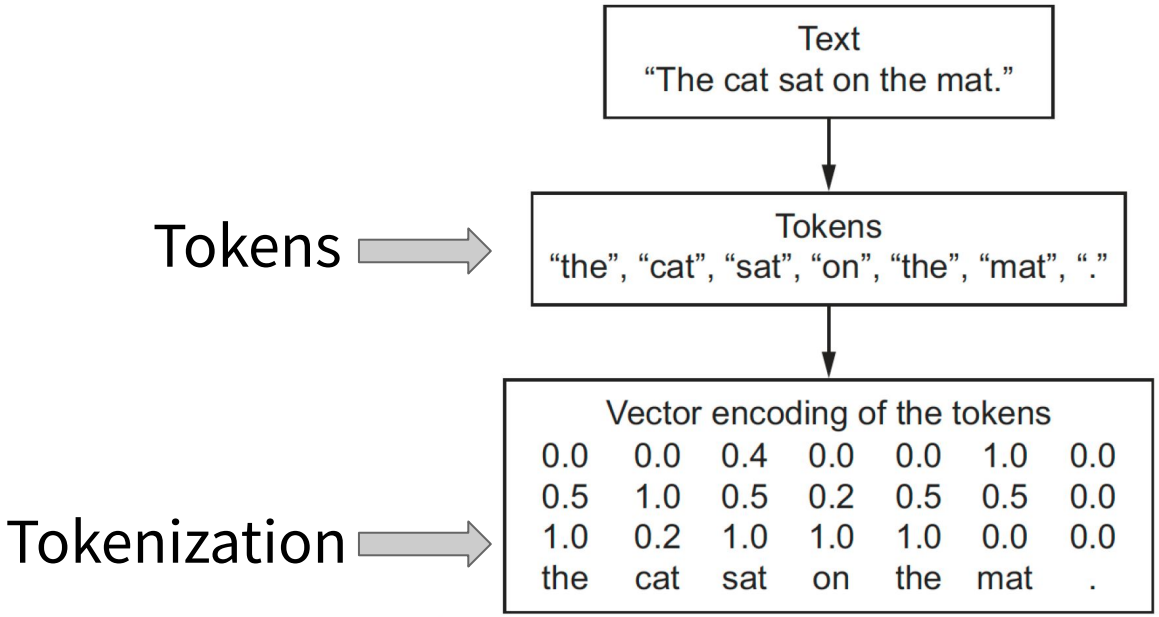
RNNs

“Unrolled” RNN





Tokenization



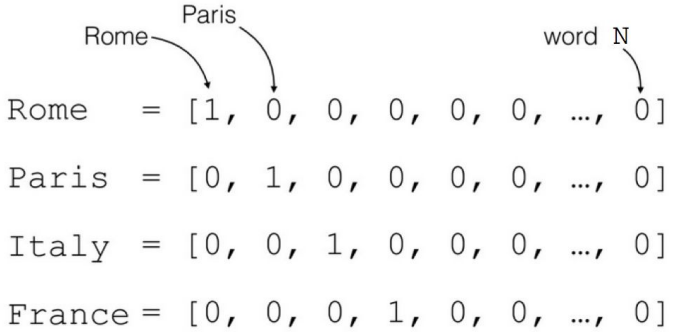
One-hot Encoding

- 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 ith entry, which is 1



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

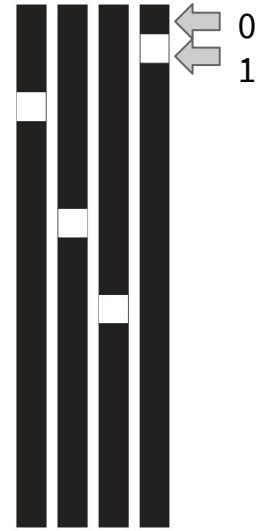
One-hot Hashing

- Main advantage: saves memory and allows generation of tokens before all of the data has been seen

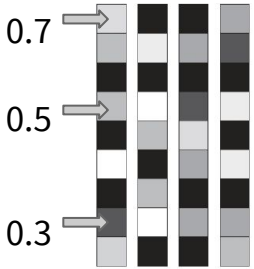
- Main drawback: **hash collisions**
  - Two different words end up with the same hash
  - The likelihood of this decreases when the dimensionality of the hashing space is much larger than the total number of unique tokens being hashed

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

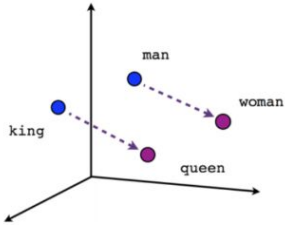
- It’s easy to simply associate a vector with a word randomly - but this results in an embedding space without structure, and things like synonyms that could be interchangeable will have completely different embeddings

- This makes it difficult for a deep neural network to make sense of these representations

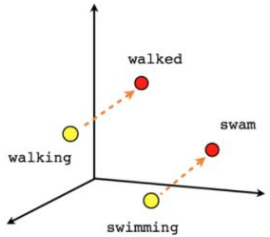
Learning Word Embeddings

- It is better for similar words to have similar embeddings, and dissimilar words to have dissimilar embeddings

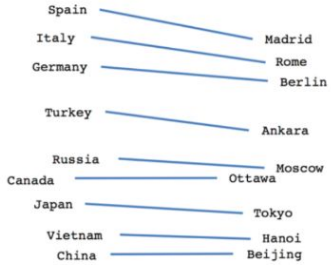
- We can, for example, relate the L2 distance to the similarity of the words with a smaller distance meaning the words are similar and bigger distances indicating very different words



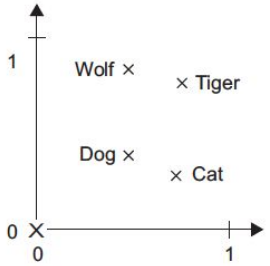
Male-Female



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

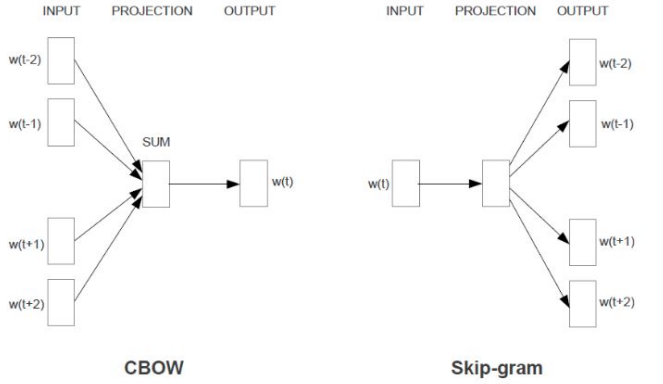
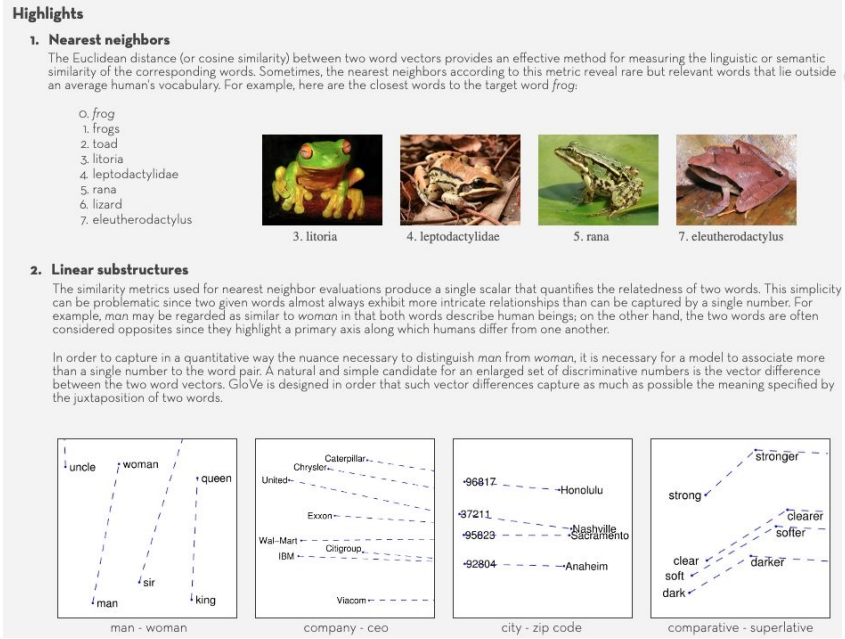


Figure 1: New model architectures. The CBOW architecture predicts the current word based on the context, and the Skip-gram predicts surrounding words given the current word.

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



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

*2vec

Distributed Representations of Sentences and Documents

Quoc Le QVL@GOOGLE.COM
Tomas Mikolov TMIKOLOV@GOOGLE.COM
Google Inc, 1600 Amphitheatre Parkway, Mountain View, CA 94043

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 tages. 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."

*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

Skip-Thought Vectors

Ryan Kiros [1], **Yukun Zhu** [1], **Ruslan Salakhutdinov** [1,2], **Richard S. Zemel** [1,2]
Antonio Torralba [3], **Raquel Urtasun** [1], **Sanja Fidler** [1]
University of Toronto [1]
Canadian Institute for Advanced Research [2]
Massachusetts Institute of Technology [3]

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.

*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

Skip-Thought Vectors

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

Patrick Ng
ppn3@cs.cornell.edu

Abstract

One of the ubiquitous representation 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

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

dna2vec: ıtions of

One of the ubiquito ter 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

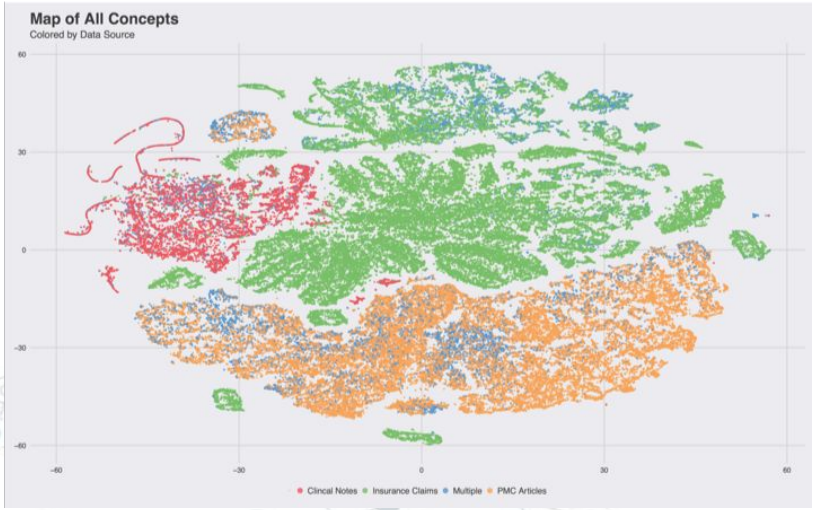
Inbar Fried
University of North Carolina

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



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

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

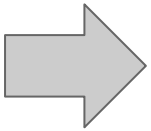
IMDb Example

Tokenization

Review 1: “This movie was great!” [5, 6, 11, 32]

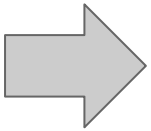
Review 2: “This movie was so bad I quit after ten minutes.” [5, 6, 11, 14, 66, 3, 49 , 55, 98, 121]

Review 3: “The setting is enchanting and captivating.” [31, 12, 2, 77, 33, 78]



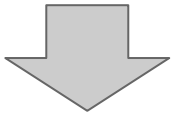
IMDb Example

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.”

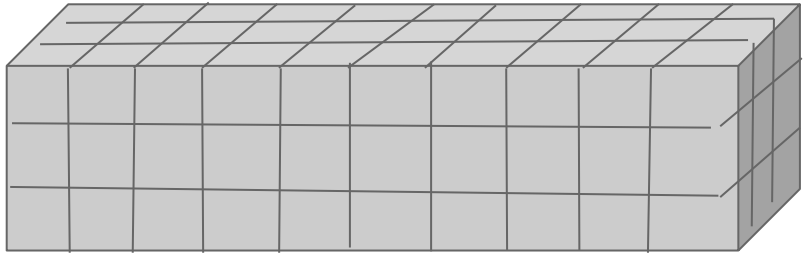
Tokenization

[5, 6, 11, 32]
[5, 6, 11, 14, 66, 3, 49 , 55, 98, 121]
[31, 12, 2, 77, 33, 78]

Padding

[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]

Embedding



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.”

Tokenization

[5, 6, 11, 32]
[5, 6, 11, 14, 66, 3, 49 , 55, 98, 121]
[31, 12, 2, 77, 33, 78]

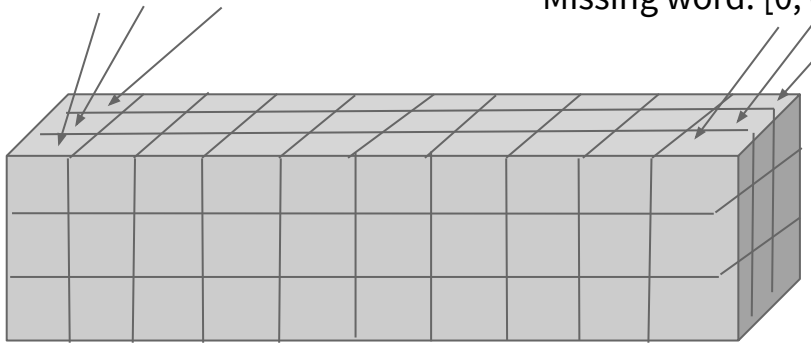
Padding

[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]

Embedding

“This” = [0.1, 0.4, 0.6]

Missing word: [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

```
# Number of words to consider as features
max_features = 10000

# Cut texts after this number of words
# (among top max_features most common words)
maxlen = 20

# Load the data as lists of integers.
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=max_features)

# This turns our lists of integers into a 2D integer tensor
# of shape (samples, maxlen)
x_train = preprocessing.sequence.pad_sequences(x_train, maxlen=maxlen)
x_test = preprocessing.sequence.pad_sequences(x_test, maxlen=maxlen)
```

Colab notebook

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

```
model = tf.keras.models.Sequential([
  # We specify the maximum input length to our Embedding layer
  # so we can later flatten the embedded inputs
  tf.keras.layers.Embedding(10000, 8, input_length = maxlen),

  # After the Embedding layer,
  # our activations have shape (samples, maxlen, 8).

  # We flatten the 3D tensor of embeddings
  # into a 2D tensor of shape (samples, maxlen * 8)
  tf.keras.layers.Flatten(),

  # We add the classifier on top
  tf.keras.layers.Dense(1, activation='sigmoid')
])

model.compile(loss = 'binary_crossentropy',
              optimizer = tf.keras.optimizers.RMSprop(),
              metrics = ['accuracy'])


history = model.fit(x_train, y_train,
                    epochs = 10,
                    batch_size = 32,
                    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

```
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
import numpy as np

maxlen = 100                 # We will cut reviews after 100 words
training_samples = 200       # We will be training on 200 samples
validation_samples = 10000   # We will be validating on 10000 samples
max_words = 10000            # We will only consider the top 10,000 words in the dataset

tokenizer = Tokenizer(num_words=max_words)
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)

word_index = tokenizer.word_index
print('Found %s unique tokens.' % len(word_index))

data = pad_sequences(sequences, maxlen=maxlen)

labels = np.asarray(labels)
print('Shape of data tensor:', data.shape)
print('Shape of label tensor:', labels.shape)

# Split the data into a training set and a validation set
# But first, shuffle the data, since we started from data
# where sample are ordered (all negative first, then all positive).
indices = np.arange(data.shape[0])
np.random.shuffle(indices)
data = data[indices]
labels = labels[indices]

x_train = data[:training_samples]
y_train = labels[:training_samples]
x_val = data[training_samples: training_samples + validation_samples]
y_val = labels[training_samples: training_samples + validation_samples]
```

```
Found 88582 unique tokens.
Shape of data tensor: (25030, 100)
Shape of label tensor: (25030,)
```

```
glove_dir = 'drive/My Drive/Teaching/BST 261/2020/In progress notebooks/glove/'

embeddings_index = {}
f = open(os.path.join(glove_dir, 'glove.6B.100d.txt'))
for line in f:
    values = line.split()
    word = values[0]
    coefs = np.asarray(values[1:], dtype='float32')
    embeddings_index[word] = coefs
f.close()

print('Found %s word vectors.' % len(embeddings_index))
```

```
embedding_dim = 100

embedding_matrix = np.zeros((max_words, embedding_dim))
for word, i in word_index.items():
    embedding_vector = embeddings_index.get(word)
    if i < max_words:
        if embedding_vector is not None:
            # Words not found in embedding index will be all-zeros.
            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.

```
model = tf.keras.models.Sequential([
  tf.keras.layers.Embedding(10000, 32),

  tf.keras.layers.SimpleRNN(32),
])

model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_4 (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 = tf.keras.models.Sequential([
  tf.keras.layers.Embedding(10000, 32),

  tf.keras.layers.SimpleRNN(32, return_sequences=True),
])
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_5 (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
```

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

```
Layer (type)                 Output Shape              Param #
=================================================================
embedding_4 (Embedding)      (None, None, 32)          320000
_________________________________________________________________
simple_rnn_1 (SimpleRNN)     (None, 32)                2080
=================================================================
Total params: 322,080
Trainable params: 322,080
Non-trainable params: 0
```

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

```
Layer (type)                 Output Shape              Param #
=================================================================
embedding_5 (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
```

- 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.

```
model = tf.keras.models.Sequential([
  tf.keras.layers.Embedding(10000, 32),

  tf.keras.layers.SimpleRNN(32),
])

model.summary()
```

```
Layer (type)                 Output Shape              Param #
=================================================================
embedding_4 (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 = tf.keras.models.Sequential([
  tf.keras.layers.Embedding(10000, 32),

  tf.keras.layers.SimpleRNN(32, return_sequences=True),
])
```

```
Layer (type)                 Output Shape              Param #
=================================================================
embedding_5 (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

```
model = tf.keras.models.Sequential([
  tf.keras.layers.Embedding(10000, 32),

  tf.keras.layers.SimpleRNN(32, return_sequences=True),
  tf.keras.layers.SimpleRNN(32, return_sequences=True),
  tf.keras.layers.SimpleRNN(32, return_sequences=True),
  tf.keras.layers.SimpleRNN(32), # This last layer only returns the last outputs.
])

model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_6 (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
```

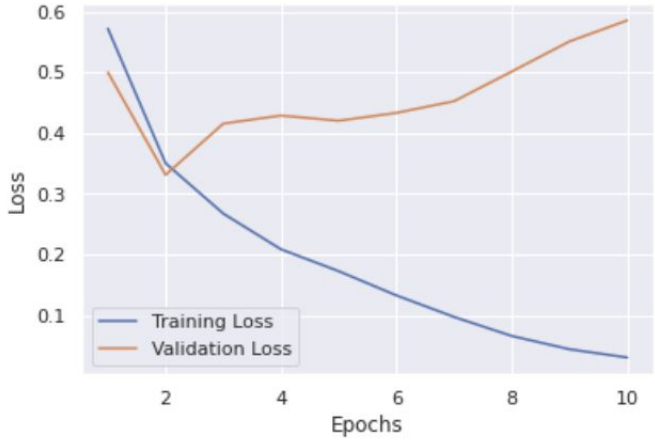
It is sometimes useful to stack several recurrent layers one after the other in order to increase the representational power of a network. In such a setup, you have to get **all intermediate layers to return full sequences**.

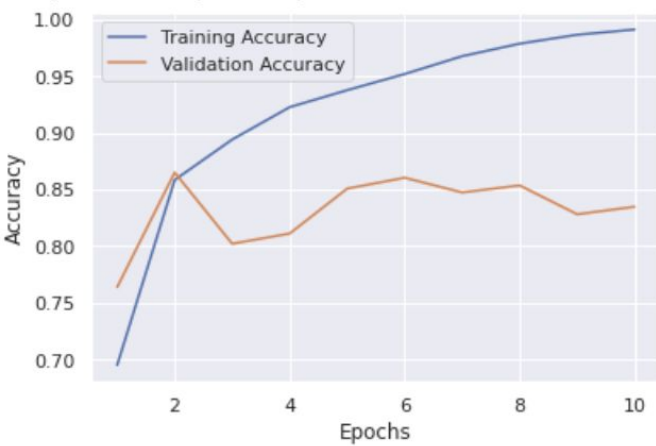
IMDb Example - Simple RNN

```
model = tf.keras.models.Sequential([
  tf.keras.layers.Embedding(max_features, 32),

  tf.keras.layers.SimpleRNN(32),

  tf.keras.layers.Dense(1, activation='sigmoid')
])

model.compile(optimizer = tf.keras.optimizers.RMSprop(),
              loss='binary_crossentropy',
              metrics=['accuracy'])

history = model.fit(input_train, y_train,
                    epochs=10,
                    batch_size=128,
                    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.





Problems with RNNs

Problems with RNNs

- Recall the formula for a generic RNN:

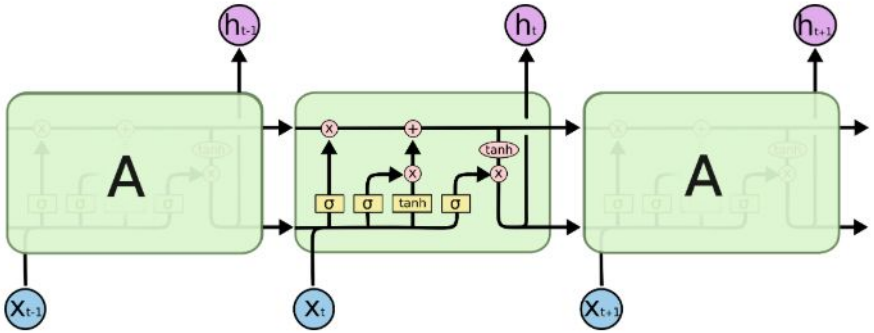
$$h_t = f(X_t W + h_{t-1} U + b)$$

- What happens for really long sequences during backprop?
  - You multiply by the matrix U repeatedly
  - Largest eigenvalue > 1, gradient $\longrightarrow$ ∞ (explodes)
  - Largest eigenvalue < 1, gradient $\longrightarrow$ 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 lecture