بسم الله الرحمن الرحیم

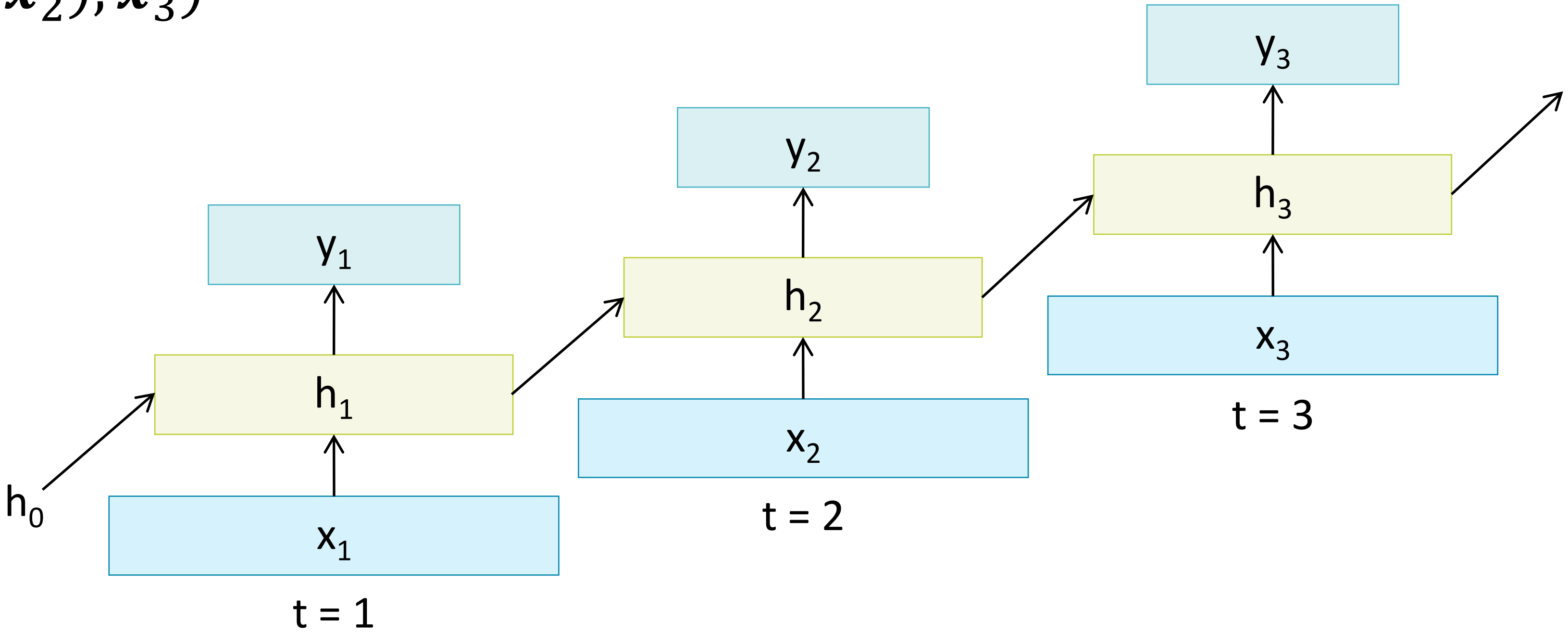
Deep Learning

Mohammad Reza Mohammadi

2021

Recurrent Neural Networks

$$
\begin{aligned}
\boldsymbol{h}_3 &= f_W(\boldsymbol{h}_2, \boldsymbol{x}_3) \\
&= f_W(f_W(\boldsymbol{h}_1, \boldsymbol{x}_2), \boldsymbol{x}_3) \\
&= f_W(f_W(f_W(\boldsymbol{h}_0, \boldsymbol{x}_1), \boldsymbol{x}_2), \boldsymbol{x}_3) \\
&= g^{(3)}(\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3)
\end{aligned}
$$



Working with text data

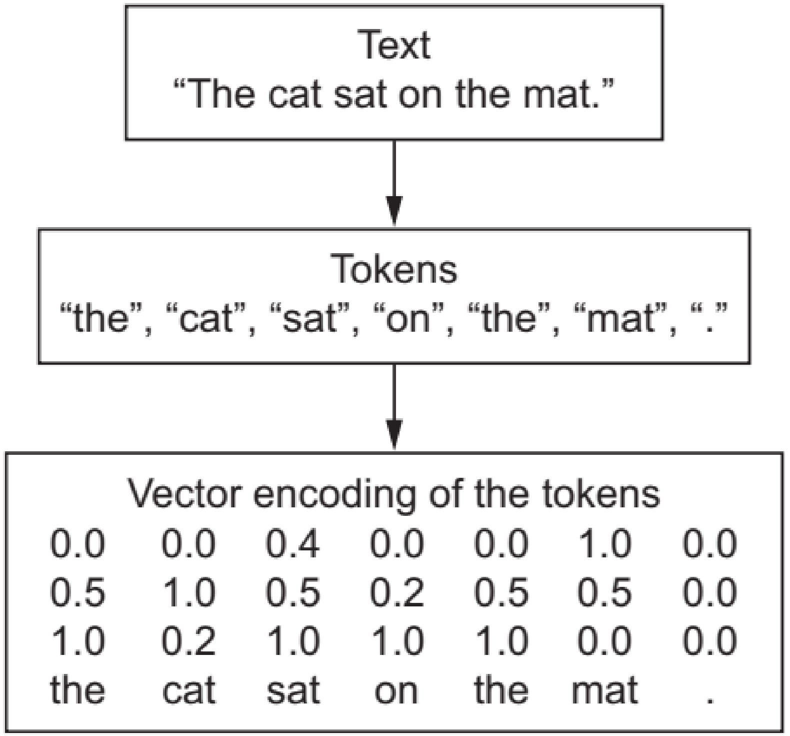
- Text can be understood as either a sequence of characters or a sequence of words
- Deep learning for natural-language processing is pattern recognition applied to words, sentences, and paragraphs, in much the same way that deep learning for computer vision is pattern recognition applied to pixels
- Applications including document classification, sentiment analysis, author identification, and even question-answering (QA)

Text processing

- Like all other neural networks, deep-learning models don’t take as input raw text: they only work with numeric tensors
- Vectorizing text:
  - 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 of words or characters, and transform each n-gram into a vector
    - N-grams are overlapping groups of multiple consecutive words or characters

Tokenization

- The different units into which you can break down text (words, characters, or n-grams) are called tokens, and breaking text into such tokens is called tokenization
- All text-vectorization processes consist of applying some tokenization scheme and then associating numeric vectors with the generated tokens



One-hot encoding

- Most basic way to turn a token into a vector
- Associating a unique integer index with every word and then turning this integer index i into a binary vector of size N (size of vocabulary)
- The vector is all zeros except for the i^{th} entry, which is 1
- One-hot encoding can be done at the character level

One-hot encoding

Listing 6.3 Using Keras for word-level one-hot encoding

Creates a tokenizer, configured to only take into account the 1,000 most common words

```
from keras.preprocessing.text import Tokenizer

samples = ['The cat sat on the mat.', 'The dog ate my homework.']

tokenizer = Tokenizer(num_words=1000)
tokenizer.fit_on_texts(samples)

sequences = tokenizer.texts_to_sequences(samples)

one_hot_results = tokenizer.texts_to_matrix(samples, mode='binary')

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

Builds the word index

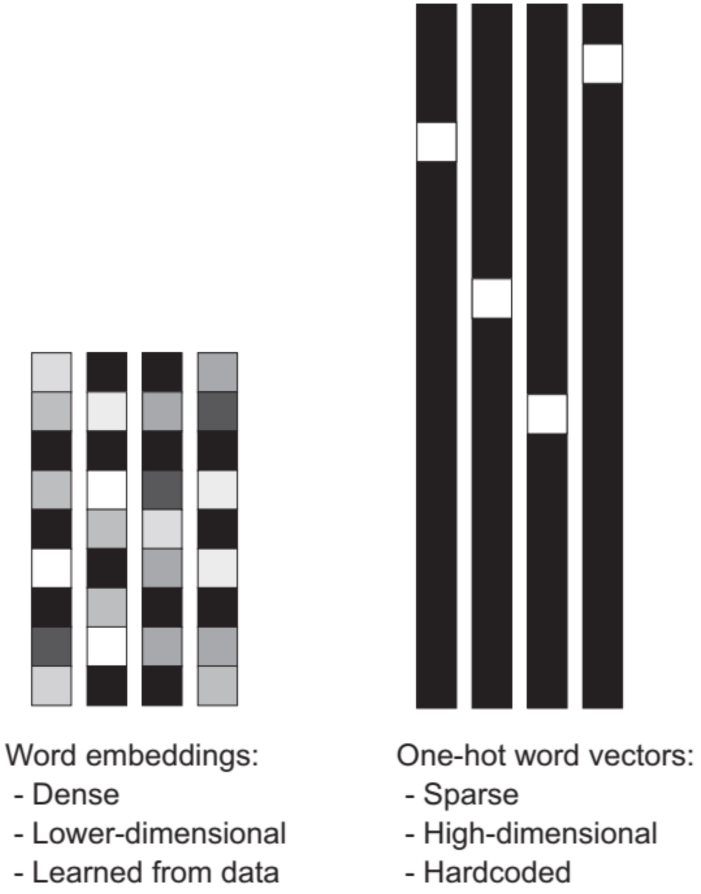
Turns strings into lists of integer indices

How you can recover the word index that was computed

You could also directly get the one-hot binary representations. Vectorization modes other than one-hot encoding are supported by this tokenizer.

Word embeddings

- Word embeddings pack more information into far fewer dimensions
- They can be pre-trained on large amounts of text training data



Word embeddings

	Man (5391)	Woman (9853)	King (4914)	Queen (7157)	Apple (456)	Orange (6257)
Gender	-1	1	-0.95	0.97	0.00	0.01
Royal	0.01	0.02	0.93	0.95	-0.01	0.00
Age	0.03	0.02	0.7	0.69	0.03	-0.02
Food	0.04	0.01	0.02	0.01	0.95	0.97
⋮	⋮	⋮				
size						
cost						
alive						
verb						

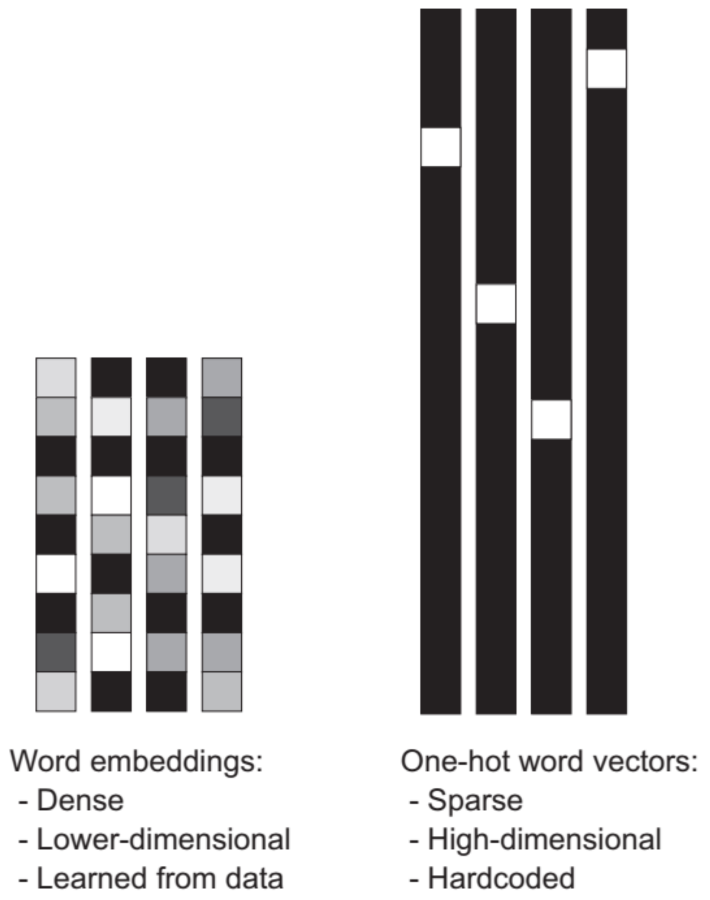
300

e_{5391} e_{9853}

Andrew Ng

Word embeddings

- There are two ways to obtain word embeddings:
  - Learn word embeddings jointly with the main task
    - Start with random word vectors
  - Load word embeddings that were precomputed using a different machine-learning task
    - Called pretrained word embeddings

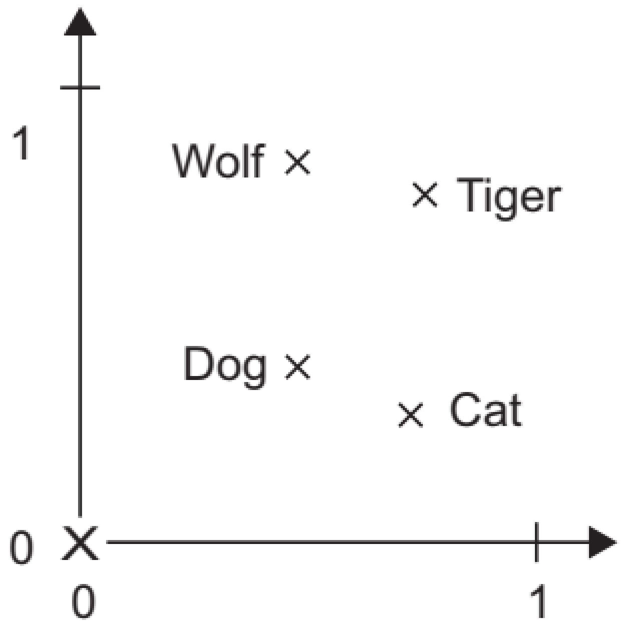


Learning word embeddings

- Associate a random vector to each word
- The problem with this approach is that the resulting embedding space has no structure
  - For instance, the words accurate and exact may end up with completely different embeddings, even though they're interchangeable in most sentences
- The geometric relationships between word vectors should reflect the semantic relationships between these words
- Word embeddings are meant to map human language into a geometric space
- In a reasonable embedding space, you would expect synonyms to be embedded into similar word vectors

Learning word embeddings

- We expect the geometric distance between any two word vectors to relate to the semantic distance between the associated words
- We may want specific directions in the embedding space to be meaningful
- The same vector allows us to go from cat to tiger and from dog to wolf
  - from pet to wild animal
- The same vector allows us to go from dog to cat and from wolf to tiger
  - from canine to feline



Learning word embeddings

- A good word-embedding space depends heavily on your task
- The perfect word-embedding space for an English-language movie-review sentiment analysis model may look different from the perfect embedding space for an English language legal-document-classification model, because the importance of certain semantic relationships varies from task to task
- Reasonable to learn a new embedding space with every new task

Embedding layer

- A dictionary that maps integer indices (which stand for specific words) to dense vectors
- Takes as input a 2D tensor of integers, of shape (samples, sequence_length)
  - (32, 10): batch of 32 sequences of length 10
- Returns a 3D floating-point tensor
  - (samples, sequence_length, embedding_dimensionality)

Listing 6.5 Instantiating an `Embedding` layer

```
from keras.layers import Embedding

embedding_layer = Embedding(1000, 64)
```

The Embedding layer takes at least two arguments: the number of possible tokens (here, 1,000: 1 + maximum word index) and the dimensionality of the embeddings (here, 64).

Embedding layer

- When we instantiate an Embedding layer, its weights (its internal dictionary of token vectors) are initially random, just as with any other layer
- During training, these word vectors are gradually adjusted via backpropagation, structuring the space into something the downstream model can exploit

Listing 6.5 Instantiating an `Embedding` layer

```
from keras.layers import Embedding

embedding_layer = Embedding(1000, 64)
```

The Embedding layer takes at least two arguments: the number of possible tokens (here, 1,000: 1 + maximum word index) and the dimensionality of the embeddings (here, 64).

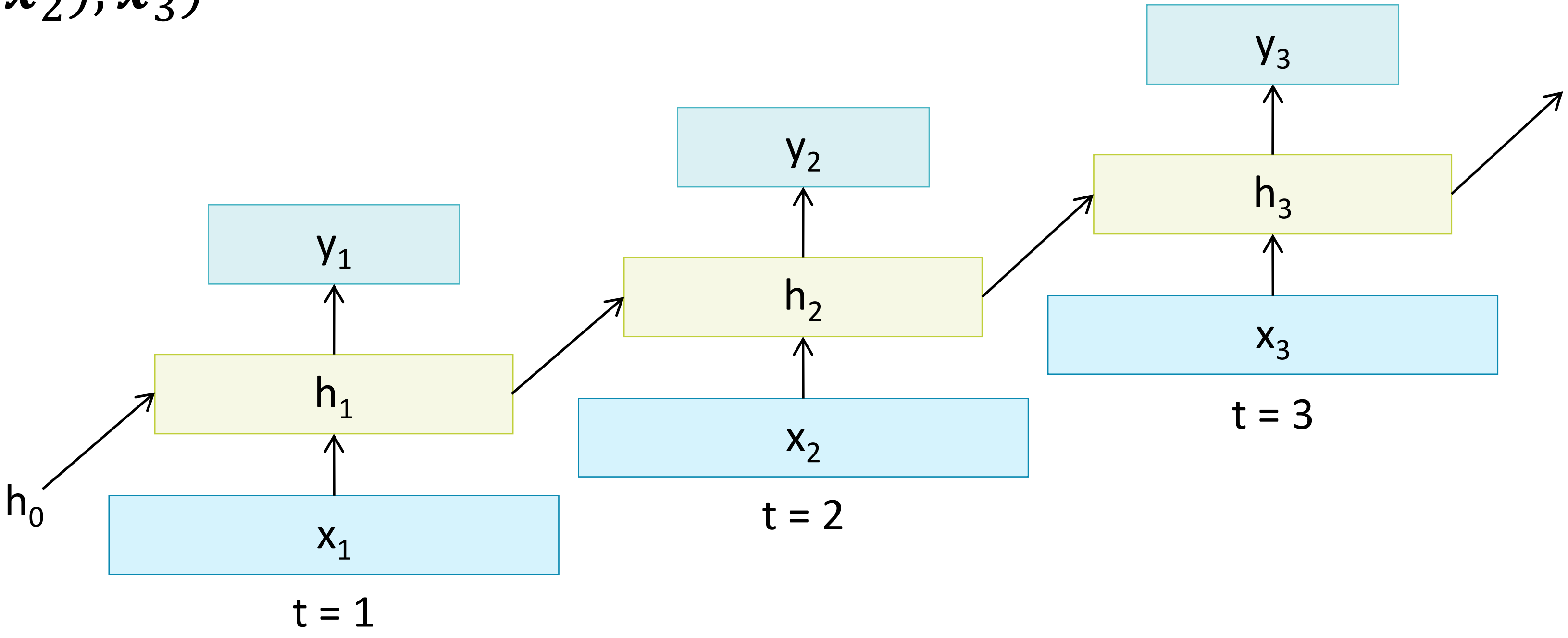
https://colab.research.google.com/drive/13B31wC8BbIQIu8frcPlR0_JBlzK1XT8V?usp=sharing

Pretrained word embeddings

- Similar in concept to pretrained ConvNets
  - We don’t have enough data available to learn truly powerful features on our own, but we expect the features that we need to be fairly generic
- Instead of learning word embeddings jointly with our problem, we can load embedding vectors from a precomputed embedding space
- Usually computed using word-occurrence statistics using a variety of techniques, some involving neural networks, others not
- Word2vec and GloVe are two of the most famous and successful word-embedding schemes

Recurrent Neural Networks

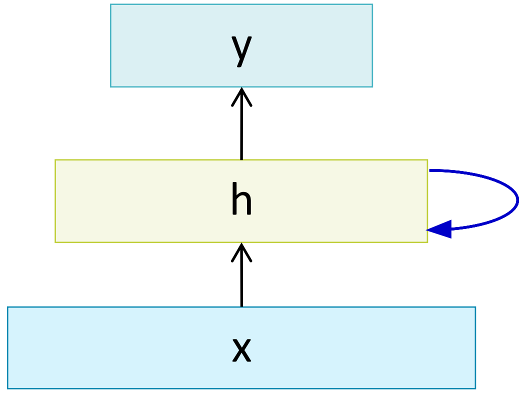
$$
\begin{aligned}
\boldsymbol{h}_3 &= f_W(\boldsymbol{h}_2, \boldsymbol{x}_3) \\
&= f_W(f_W(\boldsymbol{h}_1, \boldsymbol{x}_2), \boldsymbol{x}_3) \\
&= f_W(f_W(f_W(\boldsymbol{h}_0, \boldsymbol{x}_1), \boldsymbol{x}_2), \boldsymbol{x}_3) \\
&= g^{(3)}(\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3)
\end{aligned}
$$



Recurrent Neural Networks

- We can process a sequence of vectors x by applying a recurrence formula at every time step
- The same function and the same set of parameters are used at every time step

$$\underbrace{\boxed{\boldsymbol{h}_t}}_{\text{new state}} = \overbrace{\boxed{f_W}}^{\text{some function with parameters } W}(\underbrace{\boxed{\boldsymbol{h}_{t-1}}}_{\text{old state}}, \overbrace{\boxed{\boldsymbol{x}_t}}^{\text{input vector at time } t})$$



(Simple) RNN

- The state consists of a single "hidden" vector h
- Sometimes called a "Vanilla RNN"

$$h_t = f_W(h_{t-1}, x_t)$$

$$h_t = \tanh(W_{hh} h_{t-1} + W_{xh} x_t)$$

$$y_t = W_{hy} h_t$$