BST 261: Data Science II

Lecture 11

Heather Mattie

Department of Biostatistics
Harvard T.H. Chan School of Public Health
Harvard University

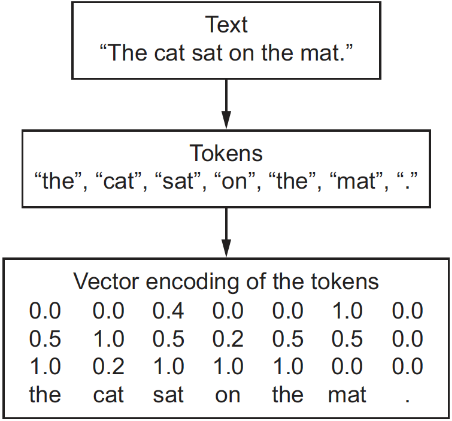
April 23, 2018

Recurrent Neural Networks

- We will now learn about deep learning models that can process text, timeseries, and sequence data in general
- The two algorithms we will cover are **recurrent neural networks** and **1D CNNs**
- 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
  - Speech recognition
  - DNA sequence analysis
  - Name entity recognition
  - etc.

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)


Text
“The cat sat on the mat.”
Tokens
“the”, “cat”, “sat”, “on”, “the”, “mat”, “.”
Vector encoding of the tokens
0.0 0.0 0.4 0.0 0.0 1.0 0.0
0.5 1.0 0.5 0.2 0.5 0.5 0.0
1.0 0.2 1.0 1.0 1.0 0.0 0.0
the cat sat on the mat .

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 does 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 and Reuters data sets
- First, associate a unique integer index with every word
- 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

Paris
Rome
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]
```

Terminology and One-hot Encoding

Suppose we want to recognize names in a sentence:
Example sentence (x) : Marcello Pagano and Heather Mattie are writing a book.
Example output (y): [1, 1, 0, 1, 1, 0, 0, 0, 0]

One-hot Encoding in Keras

```
from keras.preprocessing.text import Tokenizer

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

# We create a tokenizer, configured to only take
# into account the top-1000 most common words
tokenizer = Tokenizer(num_words=1000)
# This builds the word index
tokenizer.fit_on_texts(samples)

# This turns strings into lists of integer indices.
sequences = tokenizer.texts_to_sequences(samples)

# You could also directly get the one-hot binary representations.
# Note that other vectorization modes than one-hot encoding are supported!
one_hot_results = tokenizer.texts_to_matrix(samples, mode='binary')

# This is how you can recover the word index that was computed
word_index = tokenizer.word_index
print('Found %s unique tokens.' % len(word_index))
```

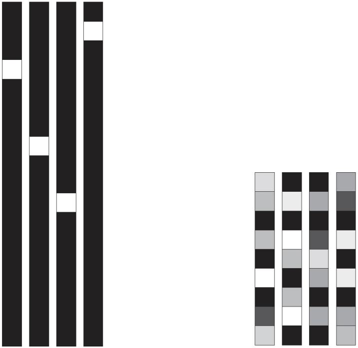
One-hot Hashing

- 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 being hashed

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

# We will store our words as vectors of size 1000.
# Note that if you have close to 1000 words (or more)
# you will start seeing many hash collisions, which
# will decrease the accuracy of this encoding method.
dimensionality = 1000
max_length = 10

results = np.zeros((len(samples), max_length, dimensionality))
for i, sample in enumerate(samples):
    for j, word in list(enumerate(sample.split()))[:max_length]:
        # Hash the word into a "random" integer index
        # that is between 0 and 1000
        index = abs(hash(word)) % dimensionality
        results[i, j, index] = 1.
```

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

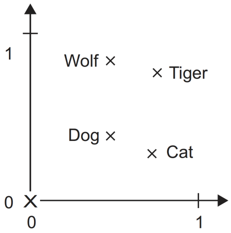
Word embeddings:
- Dense
- Lower-dimensional
- Learned from data

Word Embeddings

- There are 2 ways to obtain word embeddings:
  - 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
  - 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



Word Embeddings

- Common examples of useful geometric transformations are “sex" 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

The Embedding Layer

- 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 $\rightarrow$ Embedding layer $\rightarrow$ 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

```
from keras.layers import Embedding
embedding_layer = Embedding(1000, 64)
```

- The `embedding` layer returns a 3D tensor of shape (`samples`, `sequence_length`, `embedding_dimensionality`)

IMDB Example

- Recall the IMDB movie review sentiment prediction task from an earlier lecture
- We have thousands of reviews and we want to classify them as either positive or negative

```
from keras.datasets import imdb
from keras import preprocessing

# 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)
```

```
from keras.models import Sequential
from keras.layers import Flatten, Dense

model = Sequential()
# We specify the maximum input length to our Embedding layer
# so we can later flatten the embedded inputs
model.add(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)`
model.add(Flatten())

# We add the classifier on top
model.add(Dense(1, activation='sigmoid'))
model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])
model.summary()

history = model.fit(x_train, y_train,
                    epochs=10,
                    batch_size=32,
                    validation_split=0.2)
```

IMDB Example

- This simple model reaches an accuracy of approximately 76%
- Note that this model treats each word in the input sequence separately, without considering inter-word relationships and sentence structure
- For example, the following two reviews might both be classified as positive:
  - “The cast was absolutely spectacular. "
  - “This was a spectacular waste of my time."
- We need to add recurrent layers or 1D convolutional layers on top of the embedded sequences in order to take into account the sequence as a whole

Using 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

- This time we will download the original text data and use pre-trained word embeddings
- You can download the data from
  `http://ai.stanford.edu/~amaas/data/sentiment/`

```
import os

imdb_dir = '/home/ubuntu/data/aclImdb'
train_dir = os.path.join(imdb_dir, 'train')

labels = []
texts = []

for label_type in ['neg', 'pos']:
    dir_name = os.path.join(train_dir, label_type)
    for fname in os.listdir(dir_name):
        if fname[-4:] == '.txt':
            f = open(os.path.join(dir_name, fname))
            texts.append(f.read())
            f.close()
            if label_type == 'neg':
                labels.append(0)
            else:
                labels.append(1)
```

IMDB Example

```
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)

Found 88582 unique tokens.
Shape of data tensor: (25000, 100)
Shape of label tensor: (25000,)
```

IMDB Example

```
# 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]
```

IMDB Example

- Download the GloVe pre-computed word embeddings from 2014 English Wikipedia from `https://nlp.stanford.edu/projects/glove/`
- It contains 100-dimensional embedding vectors for 400,000 words (or non-word tokens)

```
glove_dir = '/home/ubuntu/data/'

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))
Found 400000 word vectors.
```

- Now build an embedding matrix that we will be able to load into an `Embedding` layer
- Needs to be of shape (`max_words`, `embedding_dim`), where each entry i contains the `embedding_dim`-dimensional vector for the word of index i in the reference word 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
```

IMDB Example

- Now build a model with the same architecture as before

```
from keras.models import Sequential
from keras.layers import Embedding, Flatten, Dense

model = Sequential()
model.add(Embedding(max_words, embedding_dim, input_length=maxlen))
model.add(Flatten())
model.add(Dense(32, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
```

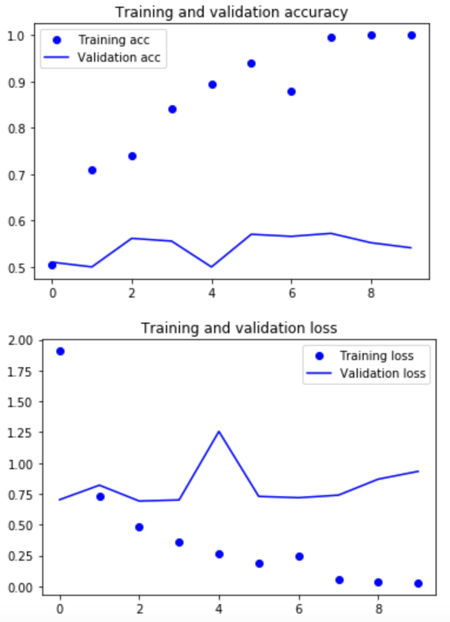
- Load the GloVe embeddings in the model

```
model.layers[0].set_weights([embedding_matrix])
model.layers[0].trainable = False
```

- Train and evaluate

```

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['acc'])
history = model.fit(x_train, y_train,
                    epochs=10,
                    batch_size=32,
                    validation_data=(x_val, y_val))
model.save_weights('pre_trained_glove_model.h5')
```

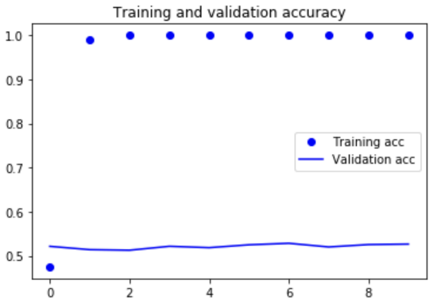
IMDB Example

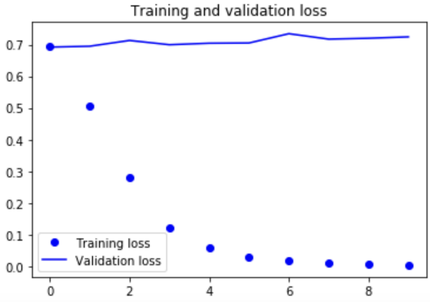


- Now let's see what happens when we train the same model without loading the pre-trained word embeddings and without freezing the embedding layer
- Here, we are learning a task-specific embedding of our input tokens
- This is generally more powerful than pre-trained word embeddings, but only when a lot of data is available

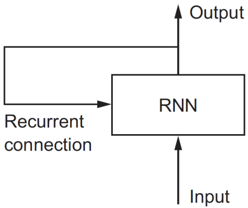
```
from keras.models import Sequential
from keras.layers import Embedding, Flatten, Dense

model = Sequential()
model.add(Embedding(max_words, embedding_dim, input_length=maxlen))
model.add(Flatten())
model.add(Dense(32, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['acc'])
history = model.fit(x_train, y_train,
                    epochs=10,
                    batch_size=32,
                    validation_data=(x_val, y_val))
```

IMDB Example





- When we were studying deep feedforward networks and CNNs, each input was processed independently and no "state" was kept between examples
- When reading text data, we process it word by word, keeping memories of what came before and eventually discovering the meaning of the whole sentence
- This "memory" property is the main difference between feedforward networks and CNNs, which are memoryless
- RNNs process sequences by iterating through the sequence elements and maintaining a **state** containing the information relative to what it has seen so far
    - Some like to think of it as a network with an internal loop



- The state of the RNN is reset between processing two different, independent sequences (examples), so you can still think of each sequence as a single data point, but that each data point is processed with an internal loop rather than in a single step

RNN Models: “Many-to-many" with $T_x = T_y$

RNN Models: “Many-to-one"

RNN Models: “One-to-many"

RNN Models: “Many-to-many" with $T_x \neq T_y$

RNNs

- Note that the time step index need not literally refer to the passage of time in the real world - it refers only to the position in the sequence.
- Recurrent networks share parameters in a different way than what we saw with CNNs
- Each member of the output is a function of the previous members of the output
- Each member of the output is produced using the same update rule applied to the previous outputs
- This recurrent formulation results in the sharing of parameters through a very deep computational graph

$$\boldsymbol{h}^{<t>} = f(\boldsymbol{h}^{(<t-1>}, \boldsymbol{x}^{<t>}; \boldsymbol{w}) \tag{1}$$

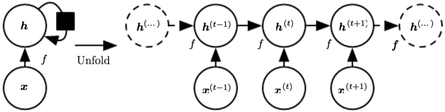
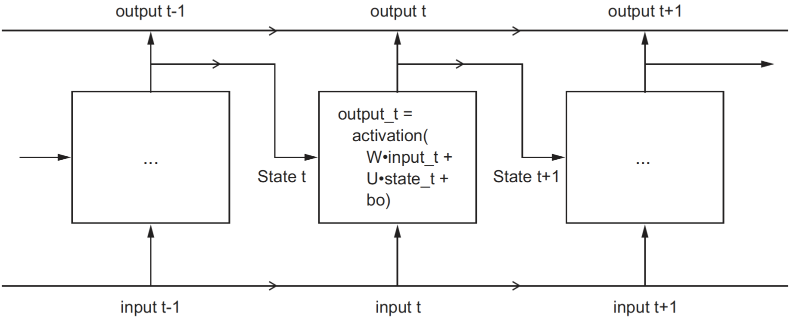


Figure 10.2: A recurrent network with no outputs. This recurrent network just processes information from the input $\boldsymbol{x}$ by incorporating it into the state $\boldsymbol{h}$ that is passed forward through time. *(Left)* Circuit diagram. The black square indicates a delay of a single time step. *(Right)* The same network seen as an unfolded computational graph, where each node is now associated with one particular time instance.

Unfolded RNN



RNN Forward Pass

- For better intuition, we can write how a training example is fed through an RNN with the following code

```
import numpy as np
timesteps = 100          # Number of timesteps in the input sequence
input_features = 32      # Dimensionality of the input feature shape
output_features = 64     # Dimensionality of the output feature shape

# Input random data just as an example
inputs = np.random.random((timesteps, input_features))
# Initial state: a zero vector
state_t = np.zeros((output_features, ))

# Create random weight matrices
W = U = np.random.random((output_features, input_features))
U = np.random.random((output_features, output_features))
b = np.random.random((output_features, ))

successive_outputs = []
for input_t in inputs:
    # Combines the input with the current state (the previous output) to obtain
        the current output
    output_t = np.tanh(np.dot(W, input_t) + np.dot(U, state_t) + b)
    # Stores the output in a list
    sucessive_outputs.append(output_t)
    # Updates the state of the network for the next time step
    state_t = output_t

# The final output is a 2D tensor of shape (timesteps, output_features)
final_output_sequence = np.concatenate(successive_outputs, axis = 0)
```

- The `SimpleRNN` processes batches of of sequences
- Takes as input a 3D tensor of shape (`batch_size`, `timesteps`, `input_features`)
- The output can return either the full sequence of successive outputs for each timestep, or only the last output for each input sequence
  - This is controlled by the `return_sequences` argument
- Only the last timestep:

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

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

- The full state sequence:

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

- It can be useful to stack several recurrent layers in order to increase the representational power of the network
- All of the intermediate layers must return full sequence outputs:

```
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.
```

- Preprocess the data

```
from keras.datasets import imdb
from keras.preprocessing import sequence

max_features = 10000  # number of words to consider as features
maxlen = 500  # cut texts after this number of words (among top max_features
      most common words)
batch_size = 32

print('Loading data...')
(input_train, y_train), (input_test, y_test) = imdb.load_data(num_words=
      max_features)
print(len(input_train), 'train sequences')
print(len(input_test), 'test sequences')

print('Pad sequences (samples x time)')
input_train = sequence.pad_sequences(input_train, maxlen=maxlen)
input_test = sequence.pad_sequences(input_test, maxlen=maxlen)
print('input_train shape:', input_train.shape)
print('input_test shape:', input_test.shape)

Loading data...
25000 train sequences
25000 test sequences
Pad sequences (samples x time)
input_train shape: (25000, 500)
input_test shape: (25000, 500)
```

IMDB Example with SimpleRNN

- Train the model

```
from keras.layers import Dense

model = Sequential()
model.add(Embedding(max_features, 32))
model.add(SimpleRNN(32))
model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])
history = model.fit(input_train, y_train,
                    epochs=10,
                    batch_size=128,
                    validation_split=0.2)
```

IMDB Example with SimpleRNN

- The simple RNN performs worse than the feedforward network we used earlier in the course
  - Here we only consider the first 500 words and not the entire sequences
  - `SimpleRNN` isn't great at processing long sequences such as text