Sequential Models for Text Data

Word Embeddings + Conv 1D + other LSTM Extensions

Today:

- Word Embeddings
- Conv 1D
- Stacked LSTMs
- Bidirectional LSTMS
- Sequential Dropout (code examples)

Sequential Models for Text Data

- You are familiar with term document frequencies

- Easy to fit models to this kind of tabular data

  - E.g.- Spam/Ham prediction, Twitter favorite count predictions, etc.

- Problem: Leaves out sequential context of language

- Solution: Use LSTM layer!

  - Every word is a step in the sequence

Sequential Models for Text Data

- Need to preprocess text to build a sequential model, How?

1. Read documents and labels into Python

2. Extract Dictionary of Terms using _Keras Tokenizer_ (same idea as Sklearn)

3. (New) For each document:

   a. Limit words in every document to sequence length, so that all docs are same length.

      i. E.g.-every document has same number of words in input sequence

   b. Convert words to their numeric index values as labelled in tokenizer dictionary

      i. Note: not strictly necessary, but this is how Keras stores words in sequence

4. Each word is read into Keras sequential model as column of a one hot encoded in a matrix

   a. Sequence steps are fed into LSTM model in same way we have seen before.

Sequential Models for Text Data

- Need to preprocess text to build a sequential model, How?

Some example code: *lmdb reviews = texts, labels are 1=positive and 0=negative*

```python
# Tokenize the data into one hot vectors
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) # converts words in each text to each word's numeric index in tokenizer dictionary.

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

data = pad_sequences(sequences, maxlen=maxlen)

labels = np.asarray(labels)
```

Sequential Models for Text Data

Data preprocessing is a starting point for building sequential models with text

But we also want to add some new techniques:

- Word Embeddings & Conv 1D Layers

Word Embeddings

- What are they? What value do they add?

    - They allow us to add continuous features to text data.

    - Every word is <u>not</u> a discrete, one hot encoded value as it is read into the sequence.

    - Instead we extract multiple meaningful features tied to each word into a vector of continuous numerical values.

    - This feature vector can be learned as a layer in a Neural Network.

        - Or it can be borrowed via transfer learning

- Let's unpack this idea.

Word Embeddings

- What are they? What value do they add?

Vocabulary (terms from tokenizer)

Features (that might break down meaningful info about words)

	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	0.70	0.69	0.03	-0.02
Food	0.09	0.01	0.02	0.01	0.95	0.97

- Words in a language have tons of meaning we miss even beyond the sequential context we haven't captured
- "Man" and "Woman" are both similar words, but we don't know this when we simply count them up in a TDF matrix
- Word embeddings help us learn features that extract this kind of meaning.

Word Embeddings

- What are they? What value do they add?

Vocabulary (terms from tokenizer)

Features (that might break down meaningful info about words)

	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	0.70	0.69	0.03	-0.02
Food	0.09	0.01	0.02	0.01	0.95	0.97

- Let's think of words as vectors we might plot geometrically
- "Man" and "Woman" would be closer to one another than they would be to "Apple" or "Orange"
- Word embeddings let us learn a vector of continuous numeric feature values for every word in our tokenized vocabulary.
  - We can learn a feature vector for each one hot encoded word as our network optimizes to a cost function.

Word Embeddings

- What are they? What value do they add?

Vocabulary (terms from tokenizer)

Features (that might break down meaningful info about words)

	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	0.70	0.69	0.03	-0.02
Food	0.09	0.01	0.02	0.01	0.95	0.97

- Using a feature vector of continuous numeric values allows us to meaningfully place similar words into a geometric space.
  - It can lead us to the extraction of meaningful data within language we could not capture otherwise!

Word Embeddings

- Workflow in practice for sequential models?
    - New document preprocessed into sequence
    - Embedding layer added as first layer of Keras sequential model.
        - We choose # of features Embedding layer will have for each word
        - Random values instantiated to start, then learned via optimization to cost function
    - For sequential model layer input, first sequential step now has same number of inputs per word in sequence as embedding layer (i.e.- new # of feature inputs like a multivariate time series model)
        - Embedding output = length of preprocessed doc sequence by number of embedding features

Word Embeddings

- Example code with no sequential layer:

```python
# Let's start with a model that ignores the sequential steps that make up each observation
from keras.layers import Dense, Embedding
from keras.models import Sequential
from keras.layers import Flatten, Dense

model = Sequential()
# Specify the size of your vocabulary (i.e.-10,000 terms)
# Specify the number of features you want to extract via fitting weights to your embedding matrix.
# We also 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)
```

Word Embeddings

- Example code with sequential layer:

```python
# Example 1: simple RNN
from keras.layers import SimpleRNN, LSTM
from keras.models import Sequential
from keras.layers import Embedding, SimpleRNN

model = Sequential()
model.add(Embedding(max_words, embedding_dim, input_length=maxlen))
model.add(SimpleRNN(32))
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_split=0.2)
```
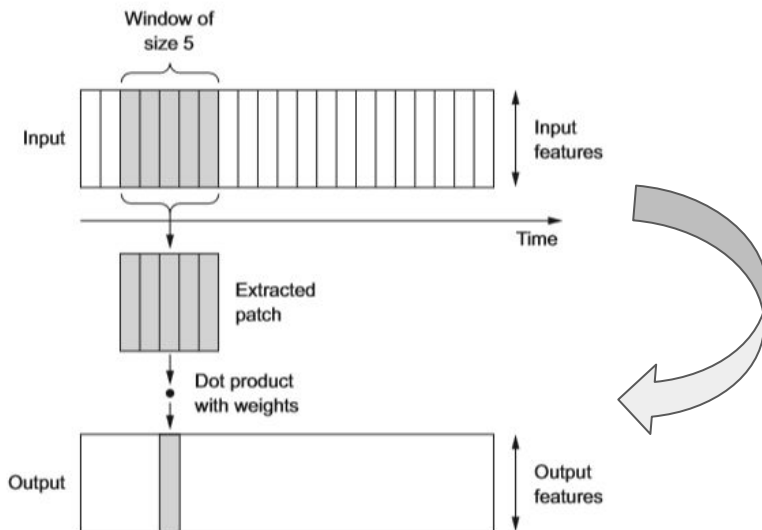
Conv1d Explained

Why not build NLP sequential models using conv layers instead?

We can! Use Conv1d with 1d pooling layers.

- Each output timestep is calculated by taking input sequence window of values and multiplying it by weights.
- Start from first window of values calculate first output timestep, slide right, repeat
- Conv1d(nmber_of_filters, window_size, input_length)

Conv1d Explained

Why not build NLP sequential models using conv layers instead?

We can! Use Conv1d with 1d pooling layers.
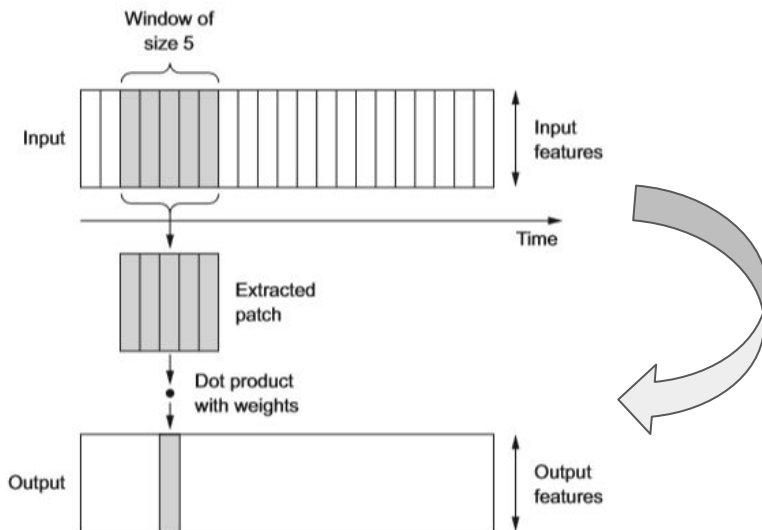
Add activation, then use <u>max pool layer</u> to complete steps for one layer.

Extracts maximum values from conv1d output within a window of values.

```
Conv1D(32, 7, activation='relu')
MaxPooling1D(5)
```

Conv1d Explained

Conv1d layers offer another approach to fitting sequential NLP models

Models train much faster than LSTMs!!

Also a way to downsample longer sequences into shorter sequences before running them through LSTM layer(s).

Stacked (i.e. "Deep") LSTMs (RNN example, but same for LSTMs)

We've seen these briefly, so here is a quick code review:

```
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))
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_split=0.2)
```
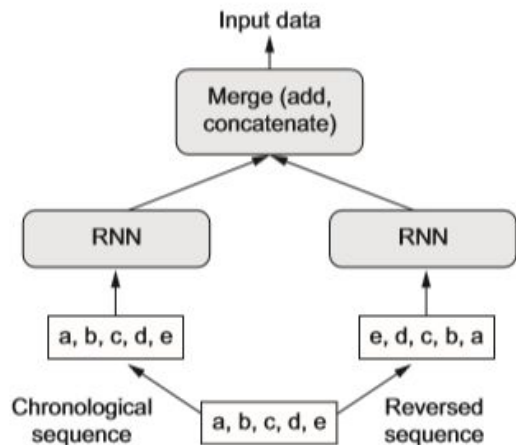
Why consider this architecture? Might improve predictive power of your model!

Common to try three stacked layers and then to add dense layers final LSTM layer above before sending to output layer.

Bidirectional RNNs or LSTMs (or GRUs)

- Why not reverse the sequence to predict outputs?
- Might find meaningful sequential context be reversing calculation
  - Recall that sequential models take data from first step to pass forward, so this would start from end of sequence to pass data to beginning!

Input data

Merge (add, concatenate)

RNN

RNN

a, b, c, d, e

e, d, c, b, a

Chronological sequence

a, b, c, d, e

Reversed sequence

Calculating same formulas for RNNs as before, but now we work through formulas on reverse of sequential inputs too:

$$a^{\langle t \rangle} = \tanh(W_{ax}x^{\langle t \rangle} + W_{aa}a^{\langle t-1 \rangle} + b_a)$$

Outputs from RNN layers per timestep act as inputs for yhat predictions in following formula:

$$\hat{y}^{\langle t \rangle} = soft\max(W_{ya}a^{\langle t \rangle} + b_y)$$

Bidirectional RNNs or LSTMs (or GRUs)

- Code example:

```python
model = Sequential()
model.add(layers.Embedding(max_features, 32))
model.add(layers.Bidirectional(layers.LSTM(32)))
model.add(layers.Dense(1, activation='sigmoid'))

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