Text Classification with Recurrent Neural Networks and Word Embeddings

CSE 4311 – Neural Networks and Deep Learning
Vassilis Athitsos
Computer Science and Engineering Department
University of Texas at Arlington

Learning Sequence-Based Features

- Bigrams are manually-crafted features that preserve some information about the order of words.
- Can we have the model learn to construct its own features that contain information about word order?
- This is what recurrent models are designed to do:
  - They process a sequence one step at a time.
  - The units of a recurrent layer receive information both from previous steps and from the current step, and combine that information in computing their output.
  - Compared to SimpleRNN units, LSTM units have even more capacity to preserve information from previous steps, and from longer ago in the past.

Preprocessing Text for an RNN

- A text document should be converted to a time series before it is given as an input to an RNN.
  - We first tokenize the document.
  - Then, each token is mapped to a number or vector.
- What would each element of this time series be?
  - What should each token map to?
- We have already seen two options:
  - An integer, indicating the position of the token in the vocabulary.
  - A one-hot vector, whose dimensions equal the size of the vocabulary.
- We have discussed why one-hot vectors are a better idea.
  - Integer representations of tokens can map tokens with very different meanings to integers close to each other.
  - With one-hot vector, each token is mapped to a vector equally different from all other vectors.

Preprocessing Text for an RNN

```
train_ds = keras.utils.text_dataset_from_directory("aclImdb/train", batch_size=32)
val_ds = keras.utils.text_dataset_from_directory("aclImdb/val", batch_size=32)
test_ds = keras.utils.text_dataset_from_directory("aclImdb/test", batch_size=32)

text_vectorization = TextVectorization(max_tokens=20000, output_mode="int")

text_only_train_ds = train_ds.map(lambda x, y: x)
text_vectorization.adapt(text_only_train_ds)
int_train_ds = train_ds.map(lambda x, y: (text_vectorization(x), y))
int_val_ds = val_ds.map(lambda x, y: (text_vectorization(x), y))
int_test_ds = test_ds.map(lambda x, y: (text_vectorization(x), y))
```

- This code maps each document into a sequence of integers.
- We have used every part of this code before, but not all together.

From Integers to One-Hot Vectors

```
text_vectorization = TextVectorization(max_tokens=20000, output_mode="int")

text_only_train_ds = train_ds.map(lambda x, y: x)
text_vectorization.adapt(text_only_train_ds)
int_train_ds = train_ds.map(lambda x, y: (text_vectorization(x), y))
int_val_ds = val_ds.map(lambda x, y: (text_vectorization(x), y))
int_test_ds = test_ds.map(lambda x, y: (text_vectorization(x), y))
```

- Our preprocessing code converts each document into a sequence of integers.
- As we have discussed several times before, eventually we want to map each integer to a one-hot vector.
- Why don’t we do that as part of preprocessing?

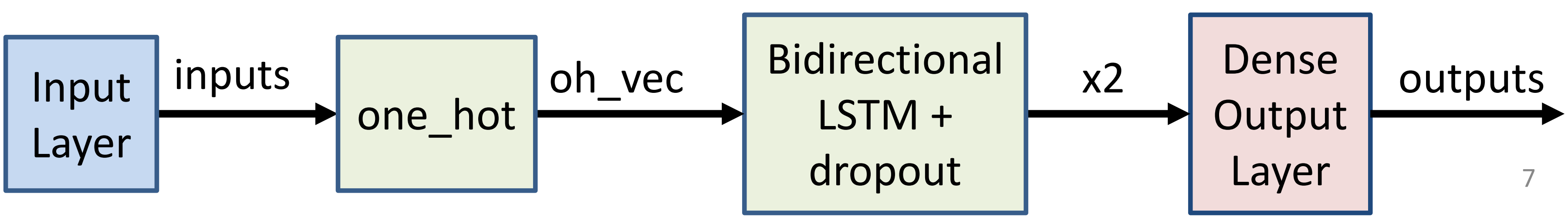
Preprocessing Text for an RNN

- If we map each document to a sequence of one-hot vectors, and we store the results, we hit a problem: memory.
- We have:
  - 50,000 documents (20,000 training, 5,000 validation, 25,000 test).
  - 230 words per document on average.
  - 20,000 dimensions per one-hot vector (since we have set our vocabulary to be 20,000 tokens).
- The resulting one-hot vectors consist of 230 billion ones and zeros.
- Even if we save them as bits, it requires about 28 gigabytes.
- This may or may not fit in a modern computer's main memory.
- A choice that reduces memory requirements dramatically is to:
  - Preprocess the documents to sequences of integers (<50MB needed).
  - Convert each document to a one-hot vector on the fly as needed.

An RNN Model for Our Dataset

```
inputs = keras.Input(shape=(None,), dtype="int64")
oh_vec = tf.one_hot(inputs, depth=max_tokens)
x1 = layers.Bidirectional(layers.LSTM(32))(oh_vec)
x2 = layers.Dropout(0.5)(x1)
outputs = layers.Dense(1, activation="sigmoid")(x2)
model = keras.Model(inputs, outputs)
```

- This code creates an RNN model, using the **Functional API**.
- See slides on the Functional API for reference.
- The main steps of the model are shown below.



Why Use the Functional API

```
inputs = keras.Input(shape=(None,), dtype="int64")
oh_vec = tf.one_hot(inputs, depth=max_tokens)
x1 = layers.Bidirectional(layers.LSTM(32))(oh_vec)
x2 = layers.Dropout(0.5)(x1)
outputs = layers.Dense(1, activation="sigmoid")(x2)
model = keras.Model(inputs, outputs)
```

- In this model, we have these layers:
  - Input layer: outputs sequence of integers
  - A layer converting the input to a sequence of one-hot vectors.
  - A bidirectional LSTM layer.
  - A fully connected output layer, with a 50% dropout rate.
- Why not use the **Sequential()** method to create this model?
  - Because there is no predefined Keras layer to produce one-hot vectors.

Why Use the Functional API

```
inputs = keras.Input(shape=(None,), dtype="int64")
oh_vec = tf.one_hot(inputs, depth=max_tokens)
x1 = layers.Bidirectional(layers.LSTM(32))(oh_vec)
x2 = layers.Dropout(0.5)(x1)
outputs = layers.Dense(1, activation="sigmoid")(x2)
model = keras.Model(inputs, outputs)
```

- With the Functional API, we can convert each input, which is a sequence of integers, to a sequence of one-hot vectors using the **tf.one_hot()** method.

RNN with One-Hot Vectors: Results

- Training this model is much slower than what we are used to.
- On my computer:
  - About 1.5 hours per epoch.
  - 15 hours for 10 epochs.
- Accuracy: about 87%.
  - Bigrams with bag-of-words vectors gave us about 90% on average.
- Why is it so slow?
- The average document is represented using 230 one-hot vectors.
- Each one-hot vector is 20,000-dimensional.
- So, the average document is represented by 4.6 million numbers.
- The model itself has about 5 million trainable parameters.
  - 64 LSTM units, each with about 80,000 weights.

Representing Words as Vectors

- If we map each word to a one-hot vector, then all resulting vectors are equally far from each other.
  - The Euclidean distance between any two such vectors is $\sqrt{2}$.
- Mapping words to vectors that are equally far from each other has its own conceptual disadvantages.
- Suppose that M is the function mapping words to vectors.
- Some words have meanings very similar to each other.
  - For example, “excellent” and “outstanding”.
- We would like M to capture that relationship, so that M(“excellent”) is very close to M(“outstanding”).
- That would simplify the learning problem.
  - If the model learns that “excellent movie” is associated with a positive review, then it automatically treats “outstanding movie” the same way.

Representing Words as Vectors

- It would also be useful if the differences between word vectors had meaning in themselves.
- For example, consider these pairs:
  - "boy" and "girl".
  - "man" and "woman".
  - "male" and "female".
- The difference between these pairs is the gender, going from male in the first element of each pair to female in the second element.
- So, intuitively, we would like a mapping M such that:

M("boy") – M("girl") = M("man") – M("woman") = M("male") – M("female")

Word Embeddings

- To recap, we would like a mapping M such that:

M(“boy”) – M(“girl”) = M(“man”) – M(“woman”) = M(“male”) – M(“female”)

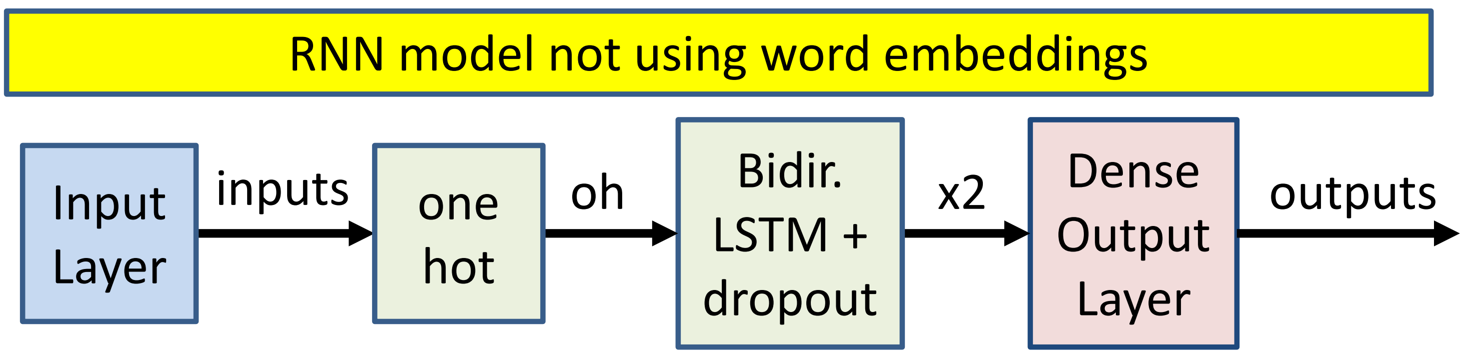
M(“large”) is similar to M(“big”)

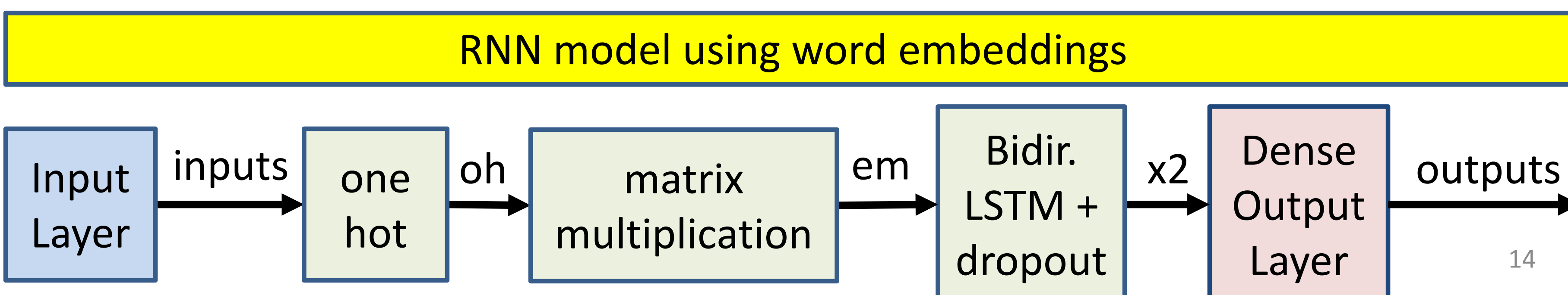
M(“buy”) is similar to M(“purchase”)

- One-hot vectors are, by definition, incapable of such behavior.
  - They do not depend in any way on the meaning of each word.
- A word embedding is a function mapping words to vectors, that aims to capture semantic relationships like the ones above.
- We can learn such a function as part of training our model.

Learning a Word Embedding

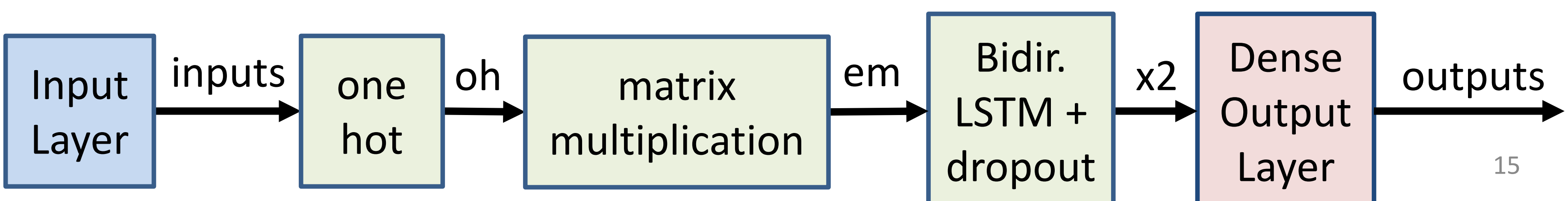
- The word embedding can be implemented as a multiplication of one-hot vector $\boldsymbol{v}$ by a matrix $\boldsymbol{W}$:
  - $\boldsymbol{v}$ = one_hot(token)
  - M(token) = $\boldsymbol{W} \times \boldsymbol{v}$.





Learning a Word Embedding

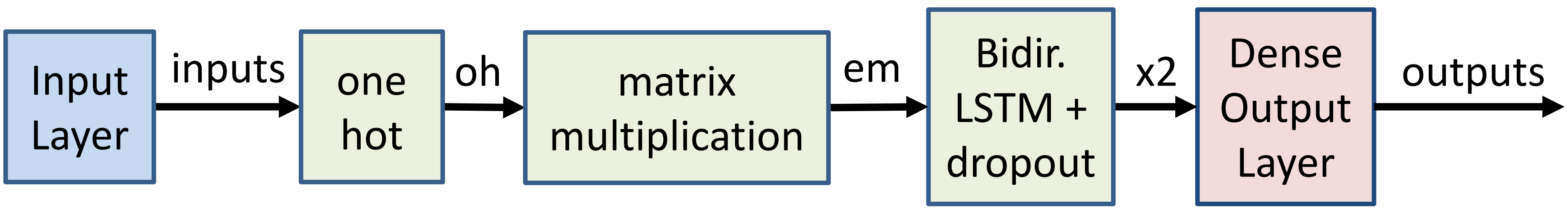
- The word embedding can be implemented as a multiplication of one-hot vector $\boldsymbol{v}$ by a matrix $\boldsymbol{W}$:
  - $\boldsymbol{v}$ = one_hot(token)
  - M(token) = $\boldsymbol{W} \times \boldsymbol{v}$.
- If the one-hot vector $\boldsymbol{v}$ is K-dimensional, and the word embedding is L-dimensional, then matrix $\boldsymbol{W}$ is of size $K \times L$.
  - The model learns those K*L values of matrix $\boldsymbol{W}$ during training.



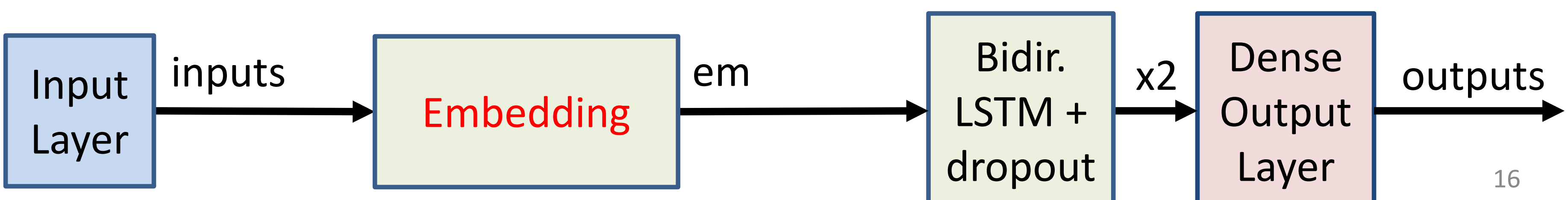
Word Embeddings in Keras

- The **keras.layers.Embedding** layer can be used directly for word embeddings.
  - It directly maps each integer to a word embedding.

RNN model using word embeddings, NOT using the Keras Embedding layer



RNN model using word embeddings, using the Keras Embedding layer

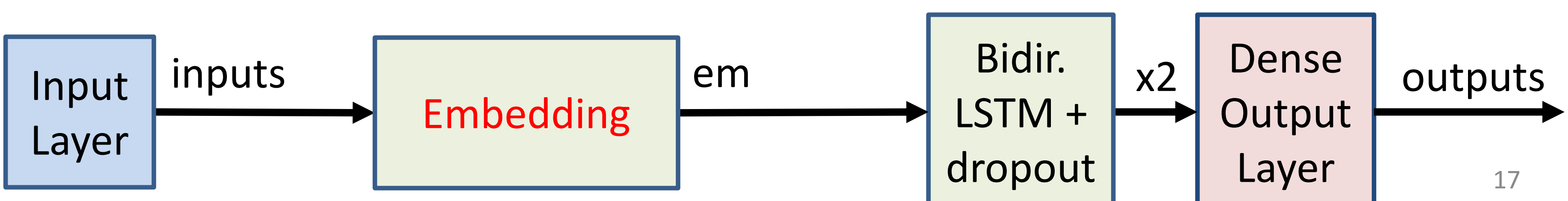


Word Embeddings in Keras

```
inputs = keras.Input(shape=(None,), dtype="int64")
em = layers.Embedding(input_dim=max_tokens, output_dim=256)(inputs)
x1 = layers.Bidirectional(layers.LSTM(32))(em)
x2 = layers.Dropout(0.5)(x1)
outputs = layers.Dense(1, activation="sigmoid")(x2)
model = keras.Model(inputs, outputs)
```

- This code creates an RNN model that uses word embeddings.
  - Setting output_dim=256 specifies that each embedding is 256-dimensional.

RNN model using word embeddings, using the Keras Embedding layer



Results for Movie Reviews

- For movie review classification, the results do not improve much.
- We still get around 87% accuracy, same as with the previous RNN model that did not use word embeddings.
  - As a reminder, bag-of-words with bigrams gave us around 90% accuracy.
- Nonetheless, word embeddings are very commonly used in text processing models.
  - We will use them again for our English-to-Spanish translation system.

Playing with Word Embeddings

- We can get the distance of the vectors corresponding to two words, using this code:

```
def we_diff(model, tv_layer, s1, s2):
    em_model = keras.Sequential(model.layers[0:2])
    v1 = em_model(tv_layer([s1]))
    v2 = em_model(tv_layer([s2]))
    diff = v2[0,0,:] - v1[0,0,:]
    return diff

def we_distance(model, tv_layer, s1, s2):
    diff = we_diff(model, tv_layer, s1, s2)
    dist = np.linalg.norm(diff)
    return dist
```

Key idea: em_model contains only the first two layers of our RNN model (input layer, embedding layer), and thus maps a sequence of words to a sequence of the corresponding vectors.

Playing with Word Embeddings

- Using the code before, we try out various pairs of words:

```
we_distance(model, text_vectorization, "great", "excellent")
we_distance(model, text_vectorization, "great", "awful")
```

Output:

```
distance from "great" to "excellent" = 1.90
distance from "great" to "awful" = 3.63
```

- Reasonable result:
  - In the word embedding space, “great” is mapped closer to “excellent” than to “awful”.

Playing with Word Embeddings

- Using the code before, we try out various pairs of words:

```
we_distance(model, text_vectorization, "big", "large")
we_distance(model, text_vectorization, "big", "small")
```

Output:

```
distance from "big" to "large" = 0.91
distance from "big" to "small" = 0.79
```

- Unexpected result:
  - In the word embedding space, “big” is mapped closer to “small” than to “large”.
- Perhaps for the purposes of separating positive and negative reviews, distinguishing these three words is not important.

Using Pretrained Word Embeddings

- Instead of learning word embeddings from our training data, we can use pre-trained embeddings.
- This is another form of transfer learning:
  - Learn word embeddings from a larger dataset.
  - Use those pre-learned embeddings in a smaller dataset.
- Some popular pre-trained word embeddings include:
  - GloVe:

    Paper: “Global Vectors for Word Representation.” J. Pennington, R. Socher, C. D. Manning. EMNLP 2014.
    Link: https://nlp.stanford.edu/projects/glove/
  - word2vec:

    Paper: “Distributed Representations of Words and Phrases and their Compositionality.” T. Mikolov, I. Sutskever, K. Chen, G. Corrado, J. Dean. NeurIPS 2013.

GloVe Embeddings

- You can download pre-trained GloVe embeddings from here:

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

- On my computer, using Anaconda, I got errors running the textbook code with those files.
- The problem was that some characters (both in the GloVe embedding files and in the movie reviews dataset) had ASCII codes greater than 127.
  - Some functions complained when encountering these characters.
- I wrote code that replaces all those problematic characters with SPACE (ASCII code 32).
- The code is posted as glove.py on the lectures web page.

Results with Glove Embeddings

- On the movie review dataset, test accuracy using pre-trained GloVe embeddings drops to 80.5%.
  – We got about 87% using word embeddings that were learned together with the rest of the model.
- Likely reasons that accuracy drops:
  – The embeddings that were learned together with the rest of the model focused on words that correspond to a review positive or negative.
  – It looks like the movie review dataset had enough training data to learn word embeddings that were more useful than the pre-trained ones.

Comparing the Two Embeddings

Output using word embeddings learned from the movie reviews:

```
distance from "buy" to "purchase" = 0.85
distance from "buy" to "shop" = 0.73
distance from "buy" to "study" = 0.77
distance from "buy" to "swim" = 1.05
```

Output using pre-trained GloVe embeddings:

```
distance from "buy" to "purchase" = 3.31
distance from "buy" to "shop" = 5.86
distance from "buy" to "study" = 6.83
distance from "buy" to "swim" = 7.16
```

- Words “buy”, “purchase”, “shop”, “study”, “swim” are not relevant for classifying movie reviews.
- GloVe embeddings capture that buy is closer to “purchase”, and to “shop”.

Comparing the Two Embeddings

Output using word embeddings learned from the movie reviews:

```
distance from "big" to "large" = 0.91
distance from "big" to "small" = 0.79
```

Output using pre-trained GloVe embeddings:

```
distance from "big" to "large" = 4.37
distance from "big" to "small" = 4.25
```

- Surprisingly, “big” is mapped closer to “small” than “large” with both approaches.
- Once again, we have models that give reasonably good results in end-to-end systems, but do not exhibit a level of understanding that resembles human intelligence.