Neural network programming: sequence-to-sequence networks

Erik Spence

SciNet HPC Consortium

6 June 2023

Today's code and slides

You can get the slides and code for today's class at the SciNet Education web page.

`https://scinet.courses/1271`

Click on the link for the class, look under "Lectures", click on "Sequence-to-sequence networks".

Today's class

This class will cover the following topics:

- Classes of recurrent-neural-network problems.
- Word representation frameworks.
- Sequence-to-sequence networks.
- Example.

Please ask questions if something isn't clear.

Other classes of RNNs

The example recurrent neural network which we dealt with last class was good, but it represented only one of the several types of recurrent neural networks we can describe. These are:

- one-to-one (one input, one output),
- many-to-one (many inputs, one output),
- one-to-many (one input, many outputs),
- many-to-many (many inputs, many outputs).

Obviously, the type we did last class was of the second type, many-to-one. Today we'd like to examine a more-complicated type, the many-to-many recurrent neural network, also called a sequence-to-sequence network.

One-hot encoding of sentences

Recall the previous way we represented our sentences. In this representation, all words are given an index in a vector of length num_words (the size of the vocabulary). The word gets a '1' when the word occurs and a '0' when it doesn't. The sentence then consists of an array of sentence_length rows and num_words columns.

Consider the sentence "The dog is in the dog crate."

The number of unique words is 5. Each word gets its own index: {the: 0, dog: 1, is: 2, in: 3, crate: 4}.

The sentence above can then be represented by the matrix to the right, with dimensions (sentence_length, num_words).

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Dealing with text sequences

Before we get to the sequence-to-sequence network we first need to deal with the shortcoming we identified with the approach we used last class:

- the input data was way too sparse,
- as a result, the information density of the input was extremely low, and the dimensionality of the problem was way too high.
- This is extremely inefficient, both from a training and a processing point of view.
- It would be better if we could come up with a technique which could condense the information associated with each word into a format with higher information density.

Various techniques to address this problem have been explored over the last decade.

Word embeddings

The first major attempt at associating words with the words found around them were the word-embedding algorithms.

- word2vec (2013): a neural network model developed by Google, it doesn't need to be trained with the network that uses it. It uses the words around the word to produce an output vector, which represents the word.
- Global Vectors for Word Representation (GLoVe, 2014): combines matrix factorization techniques with word2vec.

Word embeddings have a major problem. Consider word *class* in these two sentences:

- I was in my neural network programming *class*.
- That woman has *class*.

The word-embedding algorithms output the same word vector for both uses of the word *class*. The word vectors are context independent.

Characters and word pieces

A different approach to word-representation is to break up the words into characters or "word pieces".

- Early attempts to simplify things broke words up into individual characters. This resulted in a small vocabulary.
- Later attempts have used "word pieces". In this approach, words are broken up into common subwords.

Once the input sentence has been "tokenized" (broken up into the representation of choice), the input was often passed through an embedding layer. Note that this approach does not solve the lack-of-context problem which plagues word embeddings.

The use of word pieces has been more successfuly than individual characters. This is the approach which we will use in this class.

Embedding layers

What are embedding layers, and how do they work?

- Embedding layers were developed for text analysis.
- For each input index, the layer returns a vector of length embedding_dimension which corresponds to a word's "word embedding".
- This vector is simply a row from a matrix of weights, of shape (vocabulary_size, embedding_dimension). These weights are learned as part of the NN training.
- The embedding layer transforms each word-index i into the ith row of the embedding weights matrix, resulting in a matrix of shape (vocabulary_size, embedding_dimension).

This allows the word tokens to be represented by indices without one-hot-encoding them all.

Word representations, the latest

As an aside, note that context-dependent word representations now exist. As you might expect, context-dependent word representations significantly out-perform word representations without context.

- ELMo (2018): Embeddings from Language Models. A bi-directional LSTM is trained; the internal states of the LSTMs are used to represent the word in question.
- ULMFiT(2018): Universal Language Model Fine-tuning. A collection techniques to apply transfer learning to language models.
- BERT (2018): Bidirectional Encoder Representations from Transformers, introduced by Google. A combination of the above two papers, and transformers, and several other approaches. It was a huge step forward.

If you start doing any sort of language processing work you will need to familiarize yourself with these.

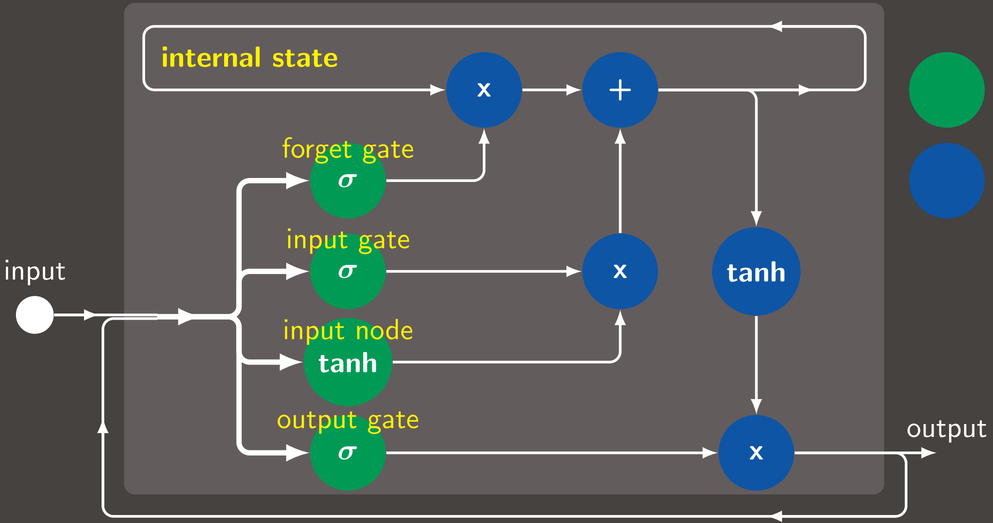
Sequence-to-sequence networks (2014)

Now that we've got a strategy for representing our words (sub-word tokenization + embedding layer) we are ready to address sequence-to-sequence networks.

- Sequence-to-sequence networks deal with the more-generic case of variable-length input and output.
- Applications of such networks include language translation, automatic text generation.
- Unlike the many-to-one network which we examined before, in this case the entire input must be processed before the output can be generated.
- The requirement of processing the entire input before proceeding to the output requires a different approach to our network.

We can leverage our previous work with LSTMs to build such a network.

Long Short Term Memory networks, memory cells



🟢 single neuron

🔵 pointwise operation

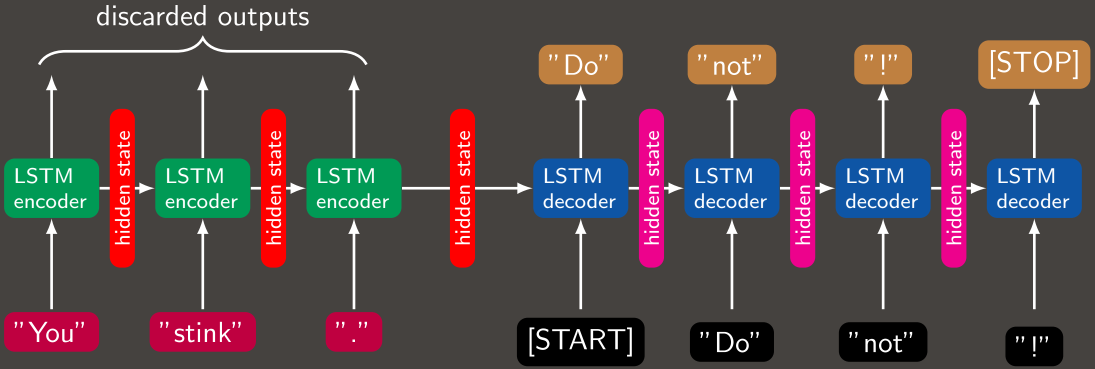
Sequence-to-sequence networks, continued

How do we build such a network?

- We use an LSTM layer (or several) as an "encoder".
- This will process the input sequence and then *return the LSTM layer's hidden states* (the internal states of the LSTM memory cells), rather than return the output of the layer. The output of the layer is discarded.
- The hidden states acts as "context" for the next step, the decoder.
- A second LSTM layer (or several) is then used as a "decoder".
- The decoder is trained to do next-word prediction on the target data.
- The decoder takes the hidden states of the encoder as its initial internal state.

In a sense you can think of this as an RNN autoencoder (though not quite).

Sequence-to-sequence networks, continued more



Recall that the words are first all tokenized, and the output of the decoder is fed back into itself (not pictured). Only the encoder's *final* hidden state is passed to the decoder.

Sequence-to-sequence example

Let's create a question-answer chatbot using a sequence-to-sequence neural network. We will use the Cornell Movie Dialog data set.

The data set consists of conversations extracted from movie scripts.

- 220,579 conversational exchanges,
- 10,292 pairs of movie characters.
- the data set is full of metadata:
  - genres of movies
  - release year
  - character metadata

We will use these talk-response pairs to train a sequence-to-sequence network. Once the network is trained, given an input text sequence we will get a response text sequence.

Sequence-to-sequence example, continued

How do we prepare the data to train such a network? We need to craft two input data sets, and a single target:

- the encoder input data set, consisting of the input data,
- the decoder input data set, consisting of the target data, used as inputs for next-word-prediction training of the decoder.
- the decoder target data set, which is the next word target for the decoder training.

We will also tokenize both the input and target data. We will treat the output of the decoder as a categorization problem, one-hot-encoding the vocabulary.

Sequence-to-sequence example, code

```
# Learn_Movies.py
import numpy as np; import shelve
import tensorflow.keras.models as km
import tensorflow.keras.layers as kl
import tensorflow_datasets.deprecated.text as tfds
import tensorflow.keras.preprocessing as tfkp

# Max sentence length, LSTM and embedding size.
max_len = 40;   latent_size = lstm_size = 64

# Preprocess (not shown).  Convert the data into
# 2 lists of cleaned questions and answers.
tokens = tfds.SubwordTextEncoder.build_from_corpus(
  questions + answers, target_vocab_size = 2**13)

START = [tokens.vocab_size]
END = [tokens.vocab_size + 1]
VOCAB_SIZE = tokens.vocab_size + 2
```

```
t_q, t_a = [], []
for q, a in zip(questions, answers):
  t_q.append(START + tokens.encode(q) + END)
  t_a.append(START + tokens.encode(a) + END)

input_size = len(t_q)

encoder_data = tfkp.sequence.pad_sequences(t_q,
  maxlen = max_len, padding = "post")
decoder_data = tfkp.sequence.pad_sequences(t_a,
  maxlen = max_len, padding = "post")

# Encode the decoder target.
decoder_target = np.zeros((input_size, max_len,
  VOCAB_SIZE), dtype = np.bool)
for i in range(input_size):
  for j in range(1,max_len):
    decoder_target[i, j-1, decoder_data[i,j]] = 1
```

Sequence-to-sequence example, code, continued

```
# LearnMovies.py, continued
encoder_input = kl.Input(shape = None)

encoder_embed = kl.Embedding(VOCAB_SIZE,
  latent_size, input_length = max_len,
  name = "encoder_embed")
encoder_embed_output = \
  encoder_embed(encoder_input)

encoder_lstm = kl.LSTM(lstm_size,
  return_state = True, name = "e_lstm",
  input_shape = (max_len, latent_size))

# Throw away the encoder output.
_, state_h, state_c = \
  encoder_lstm(encoder_embed_output)

encoder_states = [state_h, state_c]
```

```
decoder_input = kl.Input(shape = None)

decoder_embed = kl.Embedding(VOCAB_SIZE, latent_size,
  input_length = max_len, name = "decoder_embed")
decoder_embed_output = decoder_embed(decoder_input)

decoder_lstm = kl.LSTM(lstm_size, name = "d_lstm",
  return_sequences = True,
  input_shape = (max_len, latent_size))
decoder_lstm_output = decoder_lstm(decoder_embed_output,
   initial_state = encoder_states)

decoder_dense = kl.Dense(VOCAB_SIZE,
  activation = "softmax", name = "output")
decoder_output = decoder_dense(decoder_lstm_output)

model = km.Model([encoder_input, decoder_input],
  decoder_output)
```

Sequence-to-sequence example, code, continued more

```
# Learn_Movies.py, continued

model.compile( optimizer = "adam",
  loss = "categorical_crossentropy",
  metrics = ["accuracy"])

fit = model.fit([encoder_data, decoder_data],
  decoder_target, epochs = 500,
  batch_size = 128, verbose = 2)

model.save("movies.S2S.h5")
```

Do not run this. And don't even think of running it without a GPU.

```
ejspence@mycomp ~>
ejspence@mycomp ~> python Learn_Movies.py
Epoch 1/250
81450/81450 - 255s - loss:  1.6200 - acc:  0.8208
- 117s - loss:  1.4271 - acc:  0.7963
Epoch 2/250
81450/81450 - 255s - loss:  0.9436 - acc:  0.8562
:
Epoch 248/250
81450/81450 - 254s - loss:  0.5058 - acc:  0.9009
Epoch 249/250
81450/81450 - 255s - loss:  0.5055 - acc:  0.9010
Epoch 250/250
81450/81450 - 253s - loss:  0.5052 - acc:  0.9010
ejspence@mycomp ~>
```

Inference with sequence-to-sequence networks

Great! Now that the network is trained we're ready to test our chatbot. As you might expect, to perform inference (run new data forward through the network) we

- feed the new input sequence into the encoder,
- grab the internal state of the encoder,
- We then iterate on the decoder:
    - pass the internal state of the encoder to the decoder,
    - starting with the "[START]" symbol, perform next word prediction to get the first predicted word,
    - append the first word to the "[START]" symbol, and then use that as the input to the decoder,
    - repeat until we generate the "[STOP]" symbol.

Let's build such a neural network.

Sequence-to-sequence example, generating code

```
# Generate_Movies.py
import shelve, numpy as np
import tensorflow.keras.models as km
import tensorflow.keras.layers as kl

model = km.load_model("movies.S2S.h5")

layers = {layer.name: layer
  for layer in model.layers}

g = shelve.open("movies.metadata.shelve")
tokens = g["tokens"]
max_len = g["max_len"]
g.close()
```

```
# Encoder input layer.
encoder_input = kl.Input(shape = (None,))

# Encoder embedding layer.
encoder_embed = layers["encoder_embed"]
encoder_embed_output = \
  encoder_embed(encoder_input)

# Encoder LSTM layer.
encoder_lstm = layers["e_lstm"]
_, e_state_h, e_state_c = \
  encoder_lstm(encoder_embed_output)
encoder_states = [e_state_h, e_state_c]

# Encoder gets its own model.
encoder_model = km.Model(encoder_input,
  encoder_states)
```

S2S example, generating code, continued

```
# Generate_Movies.py, continued

# Decoder input layer.
decoder_input = kl.Input(shape = (None,))

# Decoder embedding layer.
decoder_embed = layers["decoder_embed"]
decoder_embed_output = decoder_embed(decoder_input)

# Decoder encoder-hidden-state input.
decoder_state_input_h = kl.Input(shape = (None,))
decoder_state_input_c = kl.Input(shape = (None,))
decoder_state_inputs = [decoder_state_input_h,
  decoder_state_input_c]
```

```
# Decoder LSTM.
decoder_lstm = layers["d_lstm"]
decoder_lstm_output = \
  decoder_lstm(decoder_embed_output,
    initial_state = decoder_state_inputs)

decoder_dense = layers["output"]
decoder_output = decoder_dense(decoder_lstm_output)

decoder_model = km.Model(
  [decoder_input] + decoder_state_inputs,
  decoder_output)
```

S2S example, generating code, continued more

```
# Generate_Movies.py, continued
input_text = "It's hot out today."

START = [token.vocab_size]
END = [token.vocab_size + 1]

# The clean_text function preps the input.
t_input = START + \
  tokens.encode(clean_text(input_text)) +
  END

# Create and populate the x data.
encoder_data = np.zeros((1, max_len))
encoder_data[0, 0:len(t_input)] = t_input

encoder_states_value = \
  encoder_model.predict(encoder_data)
```

```
decoder_data = np.zeros((1, max_len))
decoder_data[0, 0] = START[0]

done = False;     response = "";     i = 1

while (not done):
  d_output = decoder_model.predict([decoder_data]
    + encoder_states_value)
  token_index = np.argmax(d_output[0, i - 1, :])

  if token_index != END[0]:
    response += tokens.decode([token_index]) + " "
  else: done = True

  decoder_data[0, i] = token_index

print("The input is", input_text)
print("The response is", response)
```

Sequence-to-sequence example, result

Once you have the model and the metadata you can run this on your laptop.

```
ejspence@mycomp ~>
ejspence@mycomp ~> python Generate_Movies.py
The input is
It's hot out today.
The response is
what are you ?
ejspence@mycomp ~>
```

Notes about our model

Some thoughts about our model.

- The model is complicated, has many many free parameters, and takes a while to train, even with a GPU.
- The data set is too small, there is some overfitting seen with the validation data.
- A modification option would be to add another LSTM layer to both the encoder and decoder, but even more data would be needed.

This is moving in the correct direction, but it does suffer from some shortcomings. More-advanced techniques are yet available. We'll look at those next class.

Linky goodness

word2vec:

- `http://adventuresinmachinelearning.com/word2vec-keras-tutorial`
- `http://www.claudiobellei.com/2018/01/07/backprop-word2vec-python`
- `https://machinelearningmastery.com/develop-word-embeddings-python-gensim`
- `https://rare-technologies.com/deep-learning-with-word2vec-and-gensim`

Dynamic word representations:

- ELMo: `https://arxiv.org/abs/1802.05365`
- ULMFiT: `https://arxiv.org/abs/1801.06146`
- BERT: `https://arxiv.org/abs/1810.04805`

Linky goodness, continued

Cornell Movie-Dialogs Corpus:

- https://www.cs.cornell.edu/~cristian/Cornell_Movie-Dialogs_Corpus.html

Sequence-to-sequence neural networks:

- https://arxiv.org/abs/1409.3215
- https://arxiv.org/abs/1406.1078
- https://blog.keras.io/a-ten-minute-introduction-to-sequence-to-sequence-learning-in-keras.html