Neural network programming: recurrent neural networks

Erik Spence

SciNet HPC Consortium

10 May 2018

Today's model, code, data and slides

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

https://support.scinet.utoronto.ca/education

Click on the link for the class, and look under "File Storage".

The code, model and data for today's class are in lecture8_code.tar.gz.

Today's class

Today's class will cover the following topics:

- Recurrent neural networks (RNNs).
- LSTMs.
- Example.
- Other RNNs.

Please ask questions if something isn't clear.

Dealing with sequential data

So far we've focussed on solving problems that involve getting the input data all at once, such as images.

But suppose we are given information that is sequential instead?

- Timeseries data, predicting future trends.
- Natural Language Processing (NLP), voice recognition, language translation.
- Next-word predictions, question answering.
- Handwriting generation.

Generally these data are processed as the data arrives, or generate an output based on a sequence of inputs, rather than getting the data all at once. This requires a different sort of network.

Dealing with sequential data, continued

Sequential data is further complicated by the long-term relationships that exist between data points.

Consider the following sentence:

I live in Canada. I speak English and ...

We can all guess what the next word in the sentence probably is. But the information which we use to determine that word is given in the sentence before.

For a neural network to be able to predict the next word, it must remember that we're talking about Canada. This information must stay in the network somehow. The network needs to 'remember'.

Recurrent neural networks

In most applications dealing with sequential data, the network needs a means of "remembering" previous data.

To this end, the output of a node is fed back into the network, as part of the input. These are called 'recurrent' neural networks. (Not to be confused with 'recursive' neural networks.)
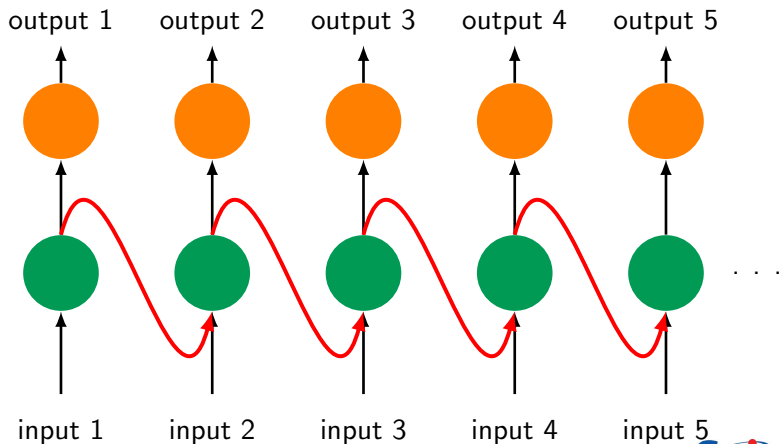
This allows the network to have 'memory', in a sense.

output

input
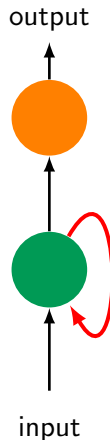
Recurrent neural networks, continued

This is what the previous network looks like when it's 'unrolled'. All the orange (and green, respectively) nodes are the same node.

The chain-like structure of these networks naturally lends itself to dealing with sequences and lists.

Backpropagation through time

How do you perform backpropagation on such a network?

- Recall that when we use Stochastic Gradient Descent to train our network, we need the derivatives of the cost function with respect to the weights and biases.
- The obvious problem is that the hidden layer references itself, and thus the references in the partial derivatives go backward forever in time, as seen in the last slide.
- While this is true, one must observe that, if my input sequence is of length n, then the unrolled version of the network only needs $n + 1$ layers to calculate the gradient.
- So while it may look scary at first, this is actually fairly straightforward.

As with all such problems, backpropagation through time is done automatically in Keras.

RNNs, continued more

Simple recurrent neural networks suffer from an instability.

- Because of the feedback on itself, it naturally acts like an amplifier.
- Depending on the activation function, you either get the vanishing gradient problem (sigmoid), or the exploding gradient problem (rectifier linear units).
- Regularization can help in these cases.
- However, the more-common approach is to switch to a different recurrent network architecture, one which is capable of actively suppressing the instability.
- The most common of these is known as Long Short Term Memory networks (LSTMs), though others are also used.
- LSTMs have been trained to do some amazing things.

output

input
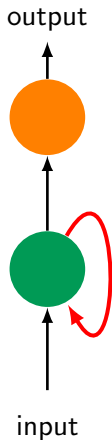
Recall the sigmoid function
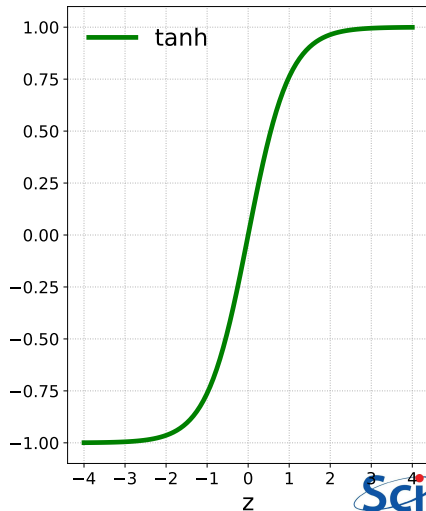
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

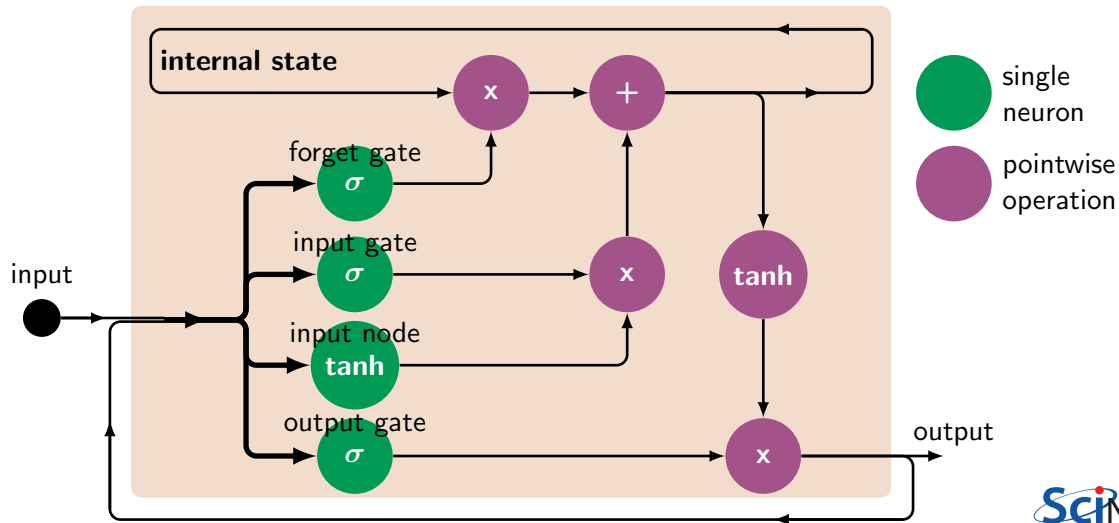This function is ideal for 'gates'.
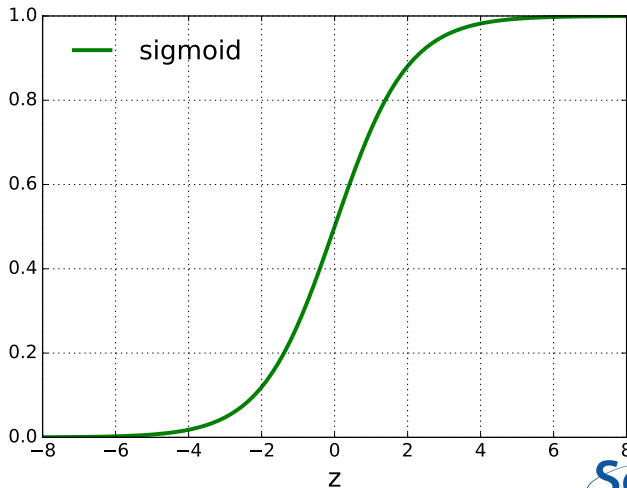
Recall the tanh function

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

In LSTMs these are used to scale the output to between -1 and 1.

Long Short Term Memory networks, memory cells

Notes about LSTM memory cells

Some notes about these memory cells.

- The 'input node' is a standard input node. These typically use a tanh activation function, though others can be used.
- How much of the input is added to the 'internal state' is controlled by the 'input gate.'
- The 'forget gate' controls how much of the internal state we're keeping, based on the input.
- The 'output gate' controls how much of the internal state is output.
- The internal state is put through a tanh function before output. This is optional, and is only done to put the output in the same range (-1 to 1) as a typical hidden layer. Some implementations use other functions such as rectifier linear units.

Notes about LSTMs

Some notes about LSTMs in networks.

- Each 'memory cell' is treated like a single neuron in a hidden layer. Typically there are many such cells in such a layer.
- In the Keras implementation of LSTMs, not only is the output of a single LSTM cell concatenated to its input, the output of all the LSTM cells in the layer are concatenated to the input.
- These networks are trained in the usual way, using Stochastic Gradient Descent and Backpropagation, as with other neural networks.
- These have been used in language translation, voice recognition, handwriting analysis, next-letter prediction, and many many other applications.

LSTM example

One common application of LSTMs is text prediction. Let's use an LSTM network to create a recipe.

- We will use the recipe data set, which is a text file containing 4869 recipes.
- We take the recipe data set, as a single file, and analyse it to find all unique words.
- We then one-hot-encode the words in the data set using our word list.
- We then break the data set into 50-word one-hot-encoded chunks ("sentences").
- We will then train the network:
  - the input will be the 50-word-encoded chunks.
  - the target will be the next word in the data set.
- Once the network is trained we can feed the network a random sentence as a seed, and it will use that sentence to generate new words, until we have a new recipe.

LSTM example, the data

```
ejspence@mycomp ~> head -24 allrecipes.txt
                    Almond Liqueur

  Amount   Measure        Ingredient -- Preparation Method
  -------  ------------   -------------------------------
        3           cup   sugar
   2 1/4           cup   water
        3                 lemons; the rind -- finely grated
        1         quart   vodka
        3     tablespoon  almond extract
        2     tablespoon  vanilla extract

Combine first 3 ingredients in a Dutch oven; bring to a boil. Reduce
heat and simmer 5 minutes, stirring occasionally; cool
completely. Stir in remaining ingredients; store in airtight
containers.

Yield: about 6 1/2 cups.

---------------------------------------------------------

                    Cafe Mexicano
  Amount   Measure        Ingredient -- Preparation Method
  -------  ------------   -------------------------------
```

One-hot encoding

One way of portraying sentences is one-hot encoding. In this representation, all words are given an index in a vector of length num_words. 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).

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

Our LSTM network

The network is simple:

- The input has dimensions (sentence_length, n_words)

- sentence_length = 50

- n_words = number of unique words in the data.

- The LSTM layer has 256 nodes.

- The output layer is fully-connected, of length n_words.

input layer

output
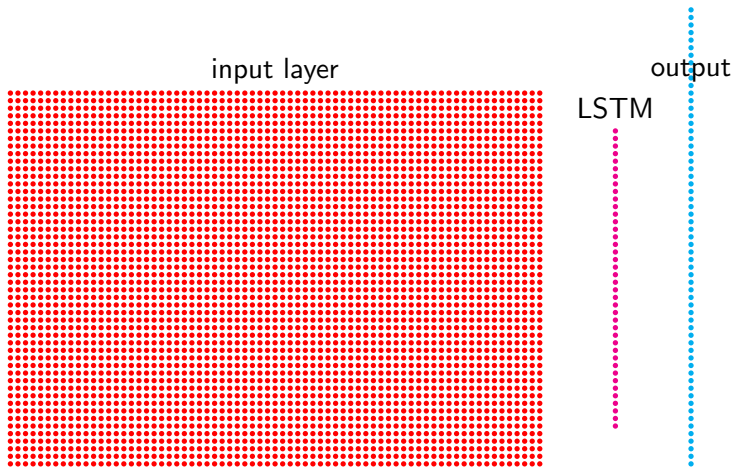
LSTM

LSTM example, data preprocessing

Before being used, the data needs to be preprocessed so that the network has an easier time learning. How is it preprocessed?

- Put spaces around the punctuation, so that "word!" becomes "word !" This is done so that "word" and "word!" are not counted as two distinct words.
- Do the same with new line symbols.
- Treat multiple dash combinations as words, put spaces around single dashes.
- Change all entries to lower case.
- Split on spaces.
- Separate all new line characters from words, so that "word" and "word\n" are not considered distinct words.
- Remove all spaces from the data (this was key).

LSTM example, learning code

```python
# Learn_Recipes.py
import numpy as np; import shelve
import keras.models as km, keras.layers as kl

# Read the data set, preprocess (not shown).
f = open('allrecipes.txt')
corpus = f.read();       f.close()

# Create the list of words.
words = sorted(list(set(corpus)))
n_words = len(words)

# Create word-index encodings.
encoding = {w: i for i, w in enumerate(words)}
decoding = {i: w for i, w in enumerate(words)}

# Initialize some parameters.
sentence_len = 50;   xdata = [];   ydata = []
```

```python
# Break up the corpus into sentences.
for i in range(len(corpus) - sentence_len):
  sentence = corpus[i: i + sentence_len]
  next_word = corpus[i + sentence_len]
  xdata.append([encoding[w] for w in sentence])
  ydata.append(encoding[next_word])

# The one-hot-encoded variables.
n_sentences = len(xdata)
x = np.zeros((n_sentences, sentence_len, \
  n_words), dtype = bool)
y = np.zeros((n_sentences, n_words), dtype = bool)

# Populate the variables.
for i, sentence in enumerate(xdata):
  for t, encoded_word in enumerate(sentence):
    x[i, t, encoded_word] = 1
  y[i, ydata[i]] = 1
```

LSTM example, learning code, continued

```python
# Learn_Recipes.py, continued

# Save the data.
g = shelve.open("data/recipes.shelve")
g["sentence_len"] = sentence_len
g["n_words"] = n_words
g["encoding"] = encoding
g["decoding"] = decoding
g.close()

# Create the NN.
model = km.Sequential()

# A layer of LSTMs.
model.add(kl.LSTM(256,
  input_shape = (sentence_len, n_words)))
```

```python
# Add a fully-connected output layer.
model.add(kl.Dense(n_words, activation = 'softmax'))

# The usual compilation.
model.compile(loss = 'categorical_crossentropy',
  optimizer = 'sgd', metrics = ['accuracy'])

# Run the fit.
fit = model.fit(x, y, epochs = 200,
  batch_size = 128, verbose = 2)

# Save the model.
model.save('data/recipes.model.h5')
```

LSTM example, running

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

```
ejspence@mycomp ~>

ejspence@mycomp ~> python Learn_Recipes.py
Using Tensorflow backend.
Epoch 1/200
332s - loss:  6.8757 - acc:  0.1458
Epoch 2/200
327s - loss:  5.5554 - acc:  0.1462
.
.
.
Epoch 198/200
328s - loss:  3.7448 - acc:  0.3332
Epoch 199/200
325s - loss:  3.7415 - acc:  0.3333
Epoch 200/200
326s - loss:  3.7388 - acc:  0.3335

ejspence@mycomp ~>
```

LSTM example, generating code

```python
# Generate_Recipe.py
import shelve, numpy as np
import keras.models as km
import random


# Read the parameters.
g = shelve.open("data/recipes.shelve")
sentence_len = g["sentence_len"]
n_words = g["n_words"]
encoding = g["encoding"]
decoding = g["decoding"];   g.close()


# Create a random seed sentence.
seed = []
for i in range(sentence_len):
  seed.append(decoding[random.randint(0,
    n_words - 1)])
```

```python
# Get the model.
model = km.load_model(data/recipes.model.h5')


# Create and populate the x data.
x = np.zeros((1, sentence_len, n_words), dtype = bool)
for i, w in enumerate(seed): x[0, i, encoding[w]] = 1

text = ""


for i in range(1000):
  pred = np.argmax(model.predict(x, verbose = 0))
  text += decoding[pred] + " "
  next_word = np.zeros((1, 1, n_words), dtype = np.bool)
  next_word[0, 0, pred] = 1
  x = np.concatenate((x[:, 1:, :], next_word), axis = 1)

print "Our recipe:"
print text
```

LSTM example, prediction

```
ejspence@mycomp ~> python Generate_Recipe.py

apple - carrot and blend mold

amount measure ingredient -- preparation method
-------- ------------ --------------------------------
1 large red small onion
1 package frozen broccoli -- chopped
1/4 cup sour cream
1 teaspoon vanilla extract
1 teaspoon instant coffee grounds
1/2 cup half ; finely shredded

place jello , cool and mix with all ingredients except remaining
1/4 cup almonds , spread topping with cinnamon juice , coconut , sunflower ,
baking powder and cut in the coconut .  place in almonds , and
top .  combine all ingredients .  bake at 350 degrees for 20 to 35
minutes to a time , until crisp .  serve and frost with your frosting .
```

LSTM example, notes

Some notes about this example.

- The model only gets 86% training accuracy so far, hence some of the 'features' of the generated recipe.
- Getting it this far took over 400 hours of training (16.5 straight days) on a GPU.
- Note the things that it gets correct:
  - ▶ It creates a title.
  - ▶ It correctly lays out the amount/measure/ingredients table.
  - ▶ It lays out ingredients with sensible amounts.
  - ▶ The instructions are more-or-less sensible.
- The things it gets wrong:
  - ▶ The instructions reference ingredients which are not in the ingredients list.

Further training should improve this. A larger dataset would improve it even more.

LSTM example, notes continued

Some more notes about this example.

- Examine the code if you want to see how the data was preprocessed.
- In particular, the many blank spaces, which created the nice formatting of the recipes, were removed, since they were dominating the fit.
- You need a large amount of data to train this model (there are a large number of parameters). Consequently it takes a long time. Use a GPU!
- The entire field of Natural Language Processing is based on techniques similar to this. However, much more efficient techniques are used.
- In particular, if you need to go down this road, you should explore the 'word embedding' packages word2vec (`https://code.google.com/p/word2vec`) and GloVe (`http://nlp.stanford.edu/projects/glove`). These create low-dimensional vectors with semantic meaning; similar words have similar vectors.

Other types of RNNs

There are many types of RNNs out there.

- Gated Recurrent Units (GRUs). These are similar to LSTMs, and are the leading competitor to them. We'll look at these in more detail.
- Bidirectional RNNs. Sometimes your output depends not just on data points that came previously, but also on future data points. These networks are just two RNNs glued together, reading the input from opposite directions.
- Fully-recurrent neural networks. In these, every node is connected to every node in the network, including itself.
- Variations on LSTMs. There are many: LSTMs with 'peep holes', LSTMs with combined input and forget gates, and many others.

Gated Recurrent Units
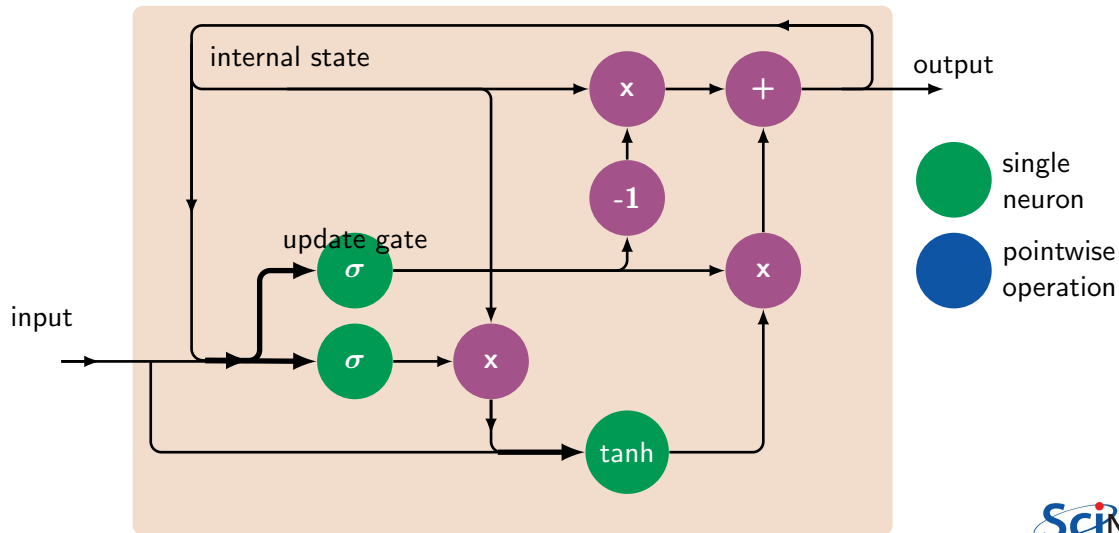
Gated Recurrent Units, notes

Some notes about gated recurrent units (GRUs).

- Are GRUs better than LSTMs? In many situations they give similar results.
- But because there are fewer trainable neurons in a GRU it will train faster than an LSTM.
- Side note: only LSTMs, GRUs and SimpleRNNs are implemented in Keras. SimpleRNNs are essentially the nodes described at the beginning of class. They are supplied for educational purposes only; do not use them.

Linky goodness

RNNs and LSTMs:

- https://karpathy.github.io/2015/05/21/rnn-effectiveness
- http://colah.github.io/posts/2015-08-Understanding-LSTMs
- https://deeplearning4j.org/lstm.html
- http://www.deeplearningbook.org/contents/rnn.html
- http://www.wildml.com/2015/09/
  recurrent-neural-networks-tutorial-part-1-introduction-to-rnns