Ontario Summer School
Artificial Neural Networks:
recurrent neural networks and transformers

Erik Spence

SciNet HPC Consortium

10 June 2025

SciNet

Today's code and slides

You can get the slides and code for today's class at the 2025 Compute Ontario Summer School web site:

`https://training.computeontario.ca/courses/course/view.php?id=130`

Scroll down to this session. You can find the slides and code there.

The code for today is in "nn3_code.tar.gz".

SciNet

Today's class

This morning's class will cover the following topics:

- Recurrent neural networks (RNNs),
- LSTMs,
- Example,
- Sequence-to-sequence networks.
- Attention networks.
- Transformers.
- Different classes of Transformers.
- Example.

Please ask questions if something isn't clear.

SciNet

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.

SciNet

Dealing with sequential data, continued

Sequential data is 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'.

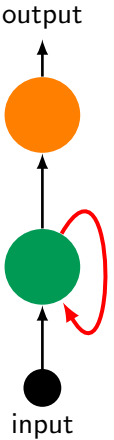
SciNet

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.)

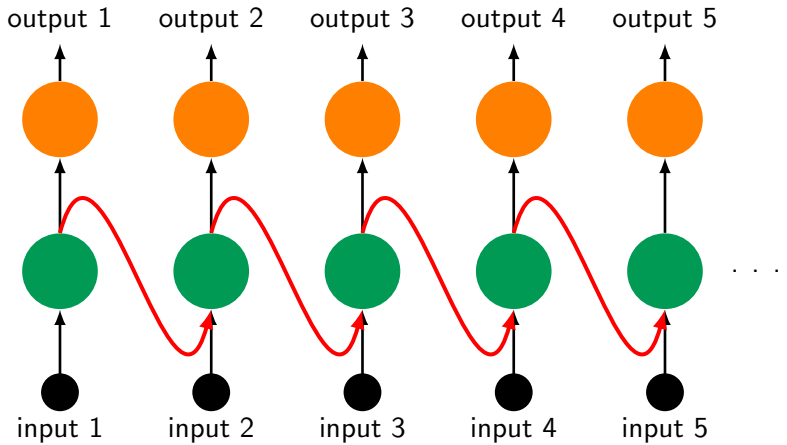
This allows the network to have 'memory', in a sense.



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 $\boldsymbol{n}$, then the unrolled version of the network only needs $\boldsymbol{n+1}$ steps to calculate the gradient.
- So while it may look scary at first, this is actually fairly straightforward.

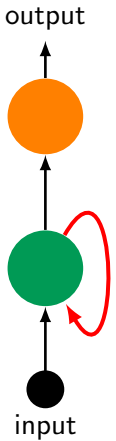
As with all such problems, backpropagation through time is done automatically in Keras.

SciNet

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.

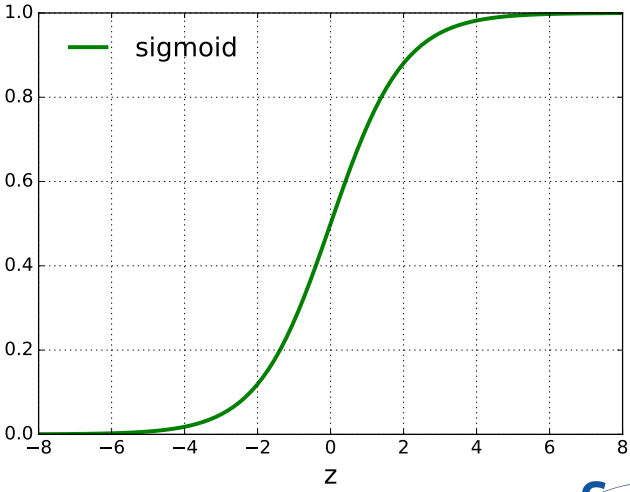


SciNet

Recall the sigmoid function

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

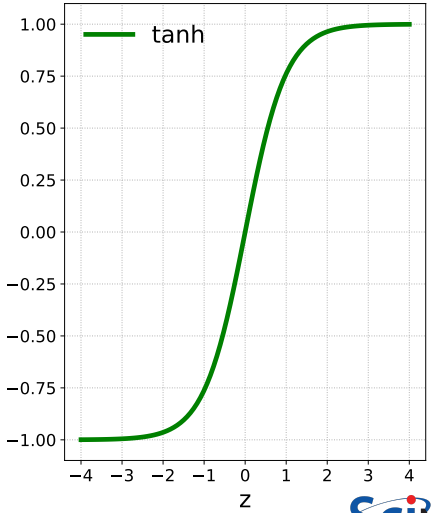
This function is ideal for 'gates'.



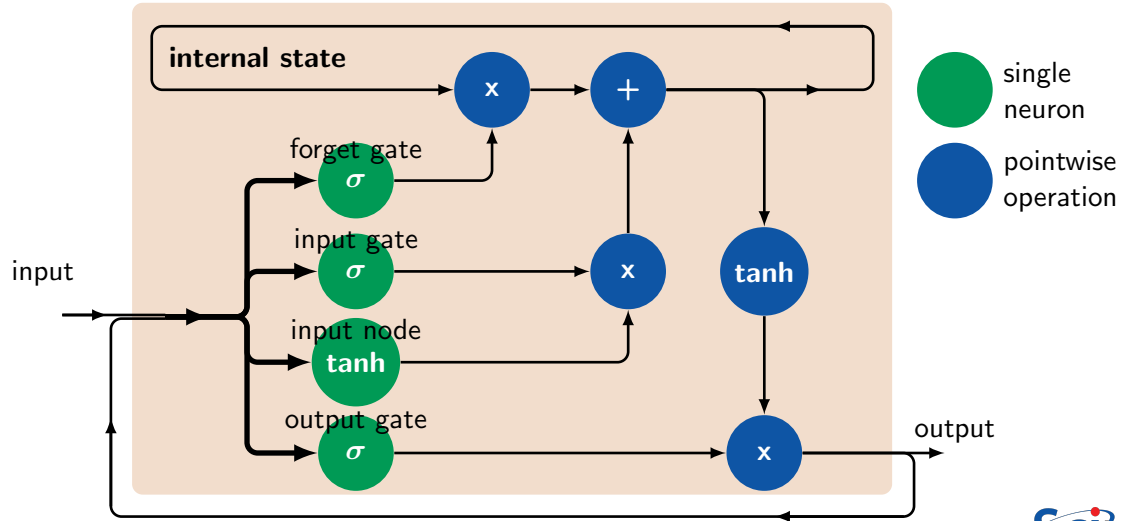
Recall the tanh function

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

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' (also called 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.

SciNet

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.

SciNet

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.

SciNet

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).

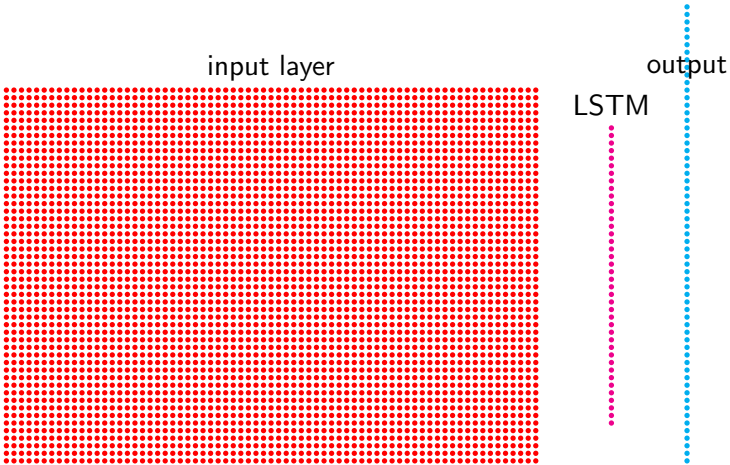
$$\begin{bmatrix} \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ \mathbf{1} & \mathbf{0}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

SciNet

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.



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).
- Remove all words that show up less than 5 times.

SciNet

LSTM example, learning code

```
# LSTM_Learn_Recipes.py
import numpy as np; import shelve
import tensorflow.keras.models as km
import tensorflow.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 = []
```

```
# 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 = np.bool)
y = np.zeros((n_sentences, n_words))

# 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

```
# LSTM_Learn_Recipes.py, continued

# Save the metadata.
g = shelve.open("lstm_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)))
```

```
# 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('lstm_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 LSTM_Learn_Recipes.py
Epoch 1/200
- 208s - loss:  4.3052 - acc:  0.2560
Epoch 2/200
- 201s - loss:  3.3269 - acc:  0.3635
⋮
Epoch 198/200
- 209s - loss:  0.0727 - acc:  0.9787
Epoch 199/200
- 206s - loss:  0.0732 - acc:  0.9784
Epoch 200/200
- 204s - loss:  0.0722 - acc:  0.9789
ejspence@mycomp ~>
```

LSTM example, generating code

```
# LSTM_Generate_Recipe.py
import shelve, numpy as np
import tensorflow.keras.models as km
import random

# Read the parameters.
g = shelve.open('lstm_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)])
```

```
# Get the model.
model = km.load_model('lstm_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 LSTM_Generate_Recipe.py

sour cream and horseradish whip squares

amount measure ingredient -- preparation method
-------- ------------ --------------------------------
1 3/4 pounds top flour -- frozen
1/2 cup olive oil
1/4 cup lemon juice
1 teaspoon garlic -- finely minced
1 teaspoon lemon rind -- finely grated
1 teaspoon vanilla extract

prepare the baking dish in a bowl , make crust , with the topping .  set aside .  add all
dry ingredients , blend well with an electric mixer .  beat the egg whites with a mixer until
blended and bake at 350f , for 15 minutes .  remove from firm , and carefully pour over margarine
.  bake until tester is well blended , 8 to 10 minutes with small spatula , ; sprinkle with
confectioner's sugar .  sprinkle chopped pecans over peaches .  spread immediately .
```

LSTM example, notes

Some notes about this example.

- The model gets 97% training accuracy so far (overfitting?).
- Getting it this far took over 12 hours of training 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 might improve this. A larger dataset would improve it even more.

SciNet

Other classes of RNNs

The example recurrent neural network which we just dealt with 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 previously was of the second type, many-to-one. Now we'd like to examine a more-complicated type, the many-to-many recurrent neural network, also called a sequence-to-sequence network.

SciNet

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 previously:

- 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. We will outline the approach which currently most-commonly used.

Sub-word tokenization

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

- Early attempts to simplify things broke words up into individual characters. This resulted in a very small vocabulary.
- Current attempts use "word pieces" (tokens). 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 is usually 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 successful than individual characters. This is the approach which we will use today.

Embedding layers

What are embedding layers, and how do they work?

- Embedding layers were developed for text analysis.
- For each input token 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 token-index i into the ith row of the embedding weights matrix.

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

SciNet

Sequence-to-sequence networks (2014)

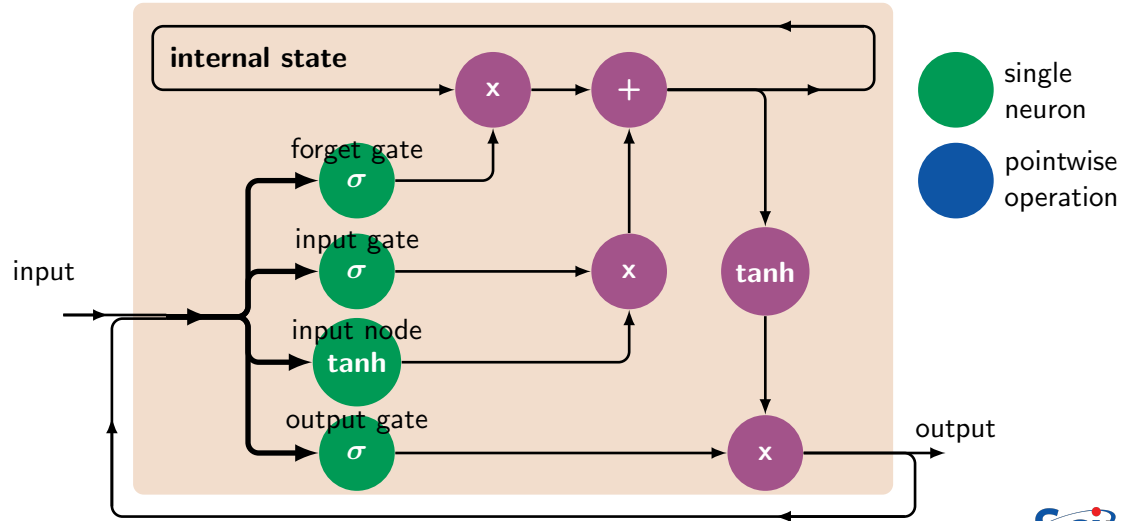
Now that we've got a strategy for representing our words (sub-word tokenization + embedding layer) we are ready to address sequence-to-sequence networks.

- 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.

SciNet

Long Short Term Memory networks, memory cells



Sequence-to-sequence networks, continued

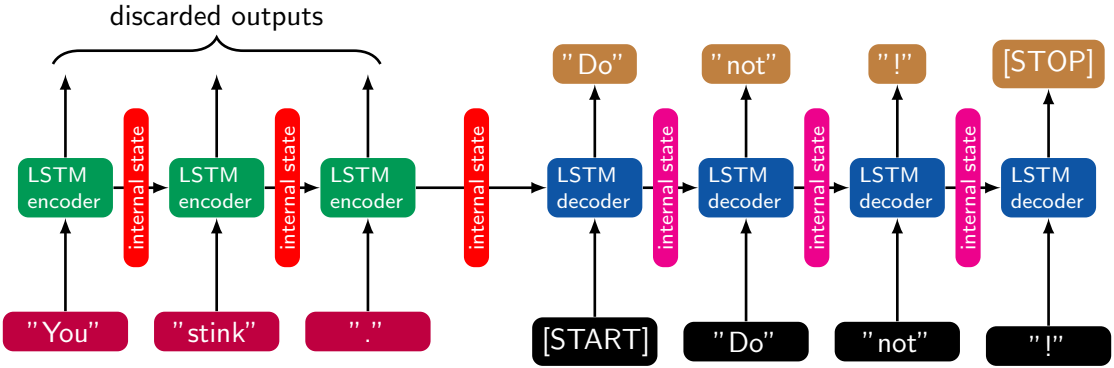
How do we build such a network?

- We use an LSTM layer (or several) as an "encoder".
- This will process the input sequence and then *return the LSTM layer's internal 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 internal states act 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 internal states of the encoder as its initial internal state.

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

SciNet

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. Only the encoder's *final* internal state is passed to the decoder.

Sequence-to-sequence model shortcomings

It turns out that there were some serious shortcomings with our previous approach to sequence-to-sequence networks.

- The internal (hidden) state of the encoder, which was passed to the decoder, was a bottleneck to the network.
- Why? Because this vector was always the same length, regardless of the length of the input sentence. This made it difficult to deal with long sentences.
- To get around this problem, a new concept called "Attention" was introduced.
- Attention allows the network to "pay attention" to the parts of the input sequence that are most important, so that it can better understand the nature of the different parts of the input.

This was a significant step forward in sequence-to-sequence networks.

SciNet

Attention models (2014, 2015)

There are two major differences between Attention models and the sequence-to-sequence models.

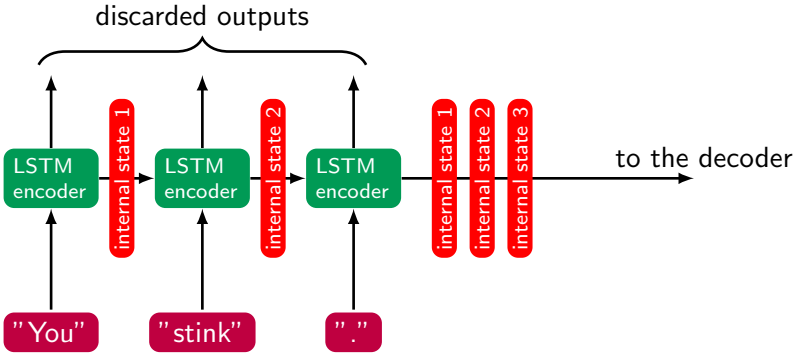
The first difference is that the encoder passes a lot more information to the decoder.

- In the sequence-to-sequence model, the encoder only passed the decoder its internal state after the whole input sequence had been processed.
- In Attention models, instead of just passing the encoder's final internal state to the decoder, *all* of the encoder's internal states are passed to the decoder.
- Thus, one internal state is passed per input step.

This extra information allows the decoder to give "attention" to specific parts of the input sequence.

SciNet

Attention networks, first change



Rather than pass just the final internal state of the encoder, Attention models pass the internal states of all the encoder steps to the decoder.

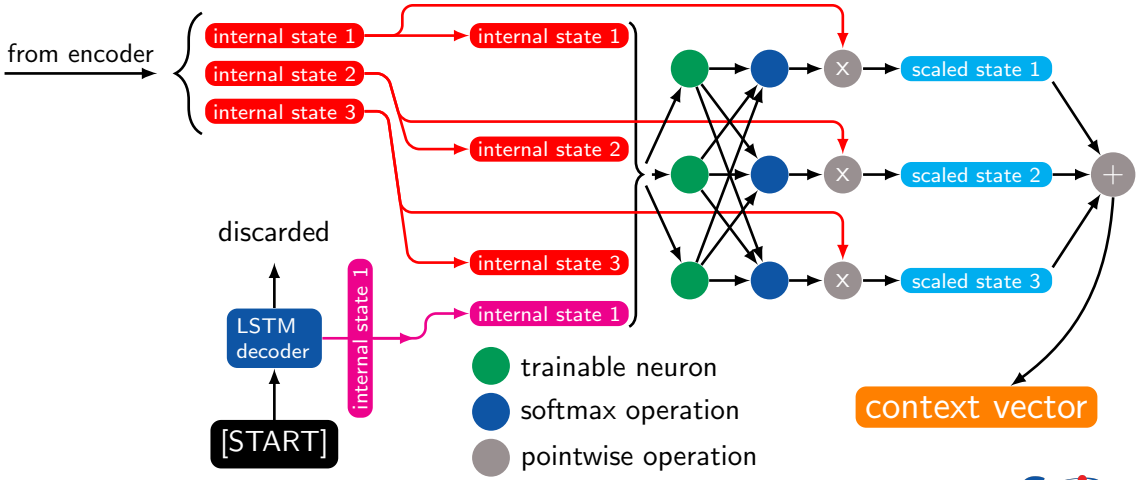
Attention models, continued

The second major difference between our previous sequence-to-sequence models and Attention models involves an extra decoder preprocessing step. For each output (decoder) step:

- The encoder's internal states and the current internal state from the decoder are run through a trainable neural network.
- The neural network has a softmax output, and outputs one value for each input (encoder) time step.
- The encoder internal states are each multiplied by their respective output from the above neural network. This amplifies the internal states that get high scores, and suppresses those that get low scores. The result is then summed, creating a "context" vector.
- This context vector gives "attention" to those internal states from the encoder which correspond to the words which are associated with the current word being processed by the decoder.

The context vector is then used in the next decoder step.

Attention networks, second change



Context vectors

Context vectors were an interesting innovation.

- Because the little neural network uses softmax as its output, you can examine its output to see which internal state gets the highest score.
- This can give you some insight as to which input word the network thinks the current output word corresponds to, or is associated with.
- In translation applications you can often see the order of the corresponding words moving around, for example, in cases where there are adjectives before versus after nouns.

The same can be done with visualization applications (which part of the image is the network focussing on when this caption word being generated?).

SciNet

Attention models, second change, continued

Ok, we've got the context vector. Now what?

The context vector is used for two purposes:

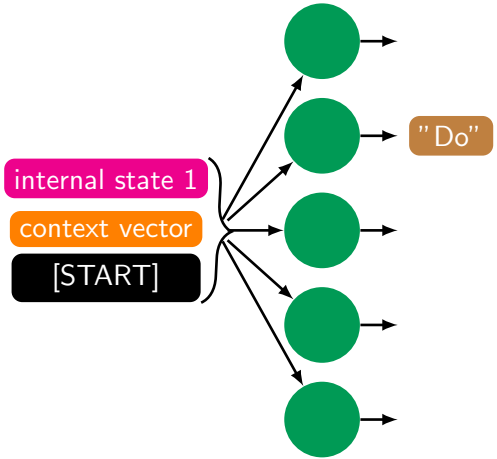
- We use it to predict the next word:
  - The context vector is concatenated with the decoder's hidden state.
  - We pass this concatenated vector, along with the decoder's input data, through yet another single-layer neural network.
  - The output of this neural network is softmax, indicating the next word in the sequence.
- We use it to update the decoder's internal state:
  - The context vector and the previous internal state are passed into the decoder and are used to calculate the next internal state (our previous implementation of the LSTM only used the input data to determine the next internal state).

SciNet

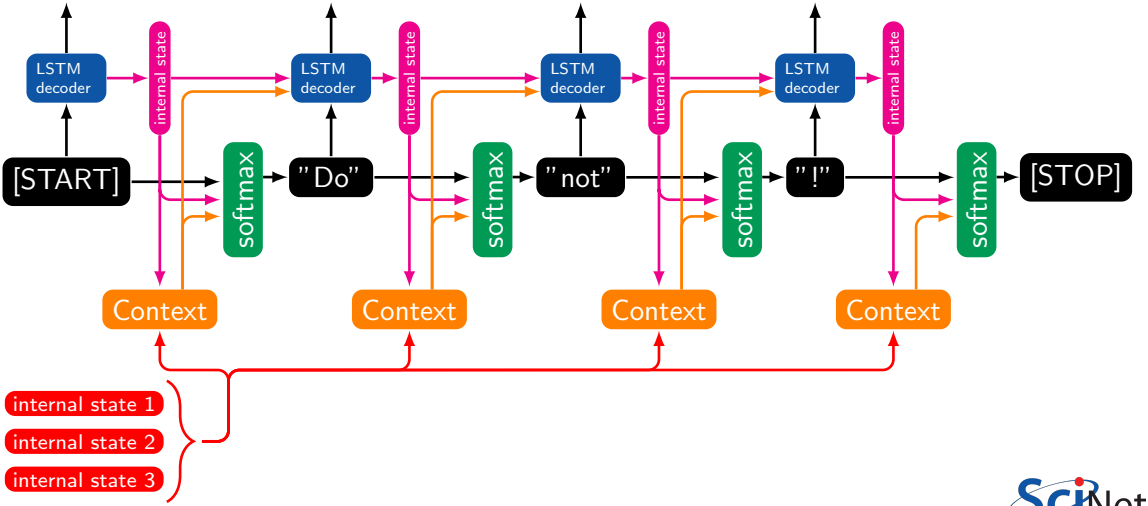
Attention networks, second change, continued more

The final next-word prediction is not done by the decoder itself, but rather by a separate network.

The network consists of a single layer; the output is softmax.



Attention networks, decoder, the whole picture



Transformers (2017)

Attention networks were a significant step forward. The next major step was the introduction of "Transformers". These are like Attention networks on steroids.

- Like regular Attention networks, the outputs of all the layers of the encoder are passed to the decoding layer.
- Unlike Attention networks, LSTMs are no longer used.
- Transformers apply a previously-used technique, "self-attention", to the sequence-to-sequence problem.
- As you might expect, this applies attention-like processing to either the encoder or decoder's own data.
- Like our previous attention mechanism, which determined which part of a previous sentence the current sentence's word should focus on, self-attention determines which words in a given sentence a given word is associated with.

SciNet

Self-attention

Self-attention is a multi-step process, not surprisingly. Recall that the input data starts as a set of embedded word vectors, one vector for each word in the input sentence.

- For each word in the sentence, take our (embedded) word vector and multiply it by three different, trainable, arrays. This creates three output vectors: "query", "key" and "value" vectors.
- We now take the dot product of the query vector of each word with the key vector of other words:
  - if we are using a unidirectional model (left-to-right), we only take the dot product with those words which are to the left of the word in question.
  - Otherwise, for bidirectional models, take the dot product with every word in the sentence.
- We divide the results by the square root of the length of these vectors, and pass the results through a softmax layer.
- As with our previous Attention method, we multiply the results by the value vector for each word, and sum up the result.

SciNet

Multi-headed self-attention

The original Transformer paper also introduced multi-headed self-attention.

- This involves doing multiple self-attention calculations in parallel, on the same input, but with different trainable matrices to create the query, key and value vectors.
- The resulting outputs of the various self-attention heads are then concatenated together.
- Like feature maps in CNNs, the different heads end up focussing on different aspects of the input.
- This concatenated result is then multiplied by yet another trainable matrix, which reduces the dimensionality to the one the network is expecting.
- The purpose of multiple self-attention heads is to allow the network to focus on multiple associated words at the same time.

SciNet

Positional encoding

Self-attention is all well and good, but everything I've described thus far is just matrix multiplication using trainable matrices. There's a problem with this: matrices have no sense of word order.

- To overcome this lack of information, the paper introduced "positional encoding" to the input data sequences.
- This consists of creating a vector for each word in the input, of the same length as the word vectors.
- Each vector is then added to each word vector in the sequence.
- Each particular vector corresponds to the position of the word in the sequence. The vector is calculated using an algorithm, not learned as part of the network.

The algorithm for calculating the positional encoding vectors is such that inputs of unseen lengths can be handled by the network.

SciNet

Transformers: the encoder

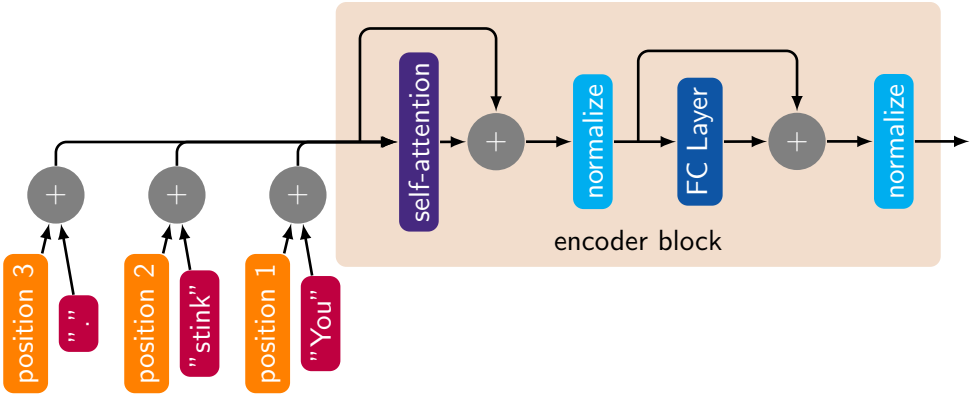
So what does the encoder look like?

- The entire sentence comes in as input. The words are tokenized into vectors, and passed through an embedding layer.
- The positional encoding is added to each word vector.
- The whole sentence is run through the self-attention layer.
- The output of the layer is added to a residual connection, and normalized.
- This is then fed into a fully-connected layer.
- The output is then added to a residual connection, and normalized.
- If it's the last layer of the encoder, the output is transformed, using a pair of trainable matrices, into a pair of "key" and "value" attention vectors. These will be used by the decoder, if there is one.

The above may be repeated several times, to create an encoder of several layers.

SciNet

Transformers: the encoder, continued



The inputs are built into a matrix, not concatenated. The output is then either passed to another encoder block, or into a into some trainable matrices which create the "key" and "value" attention vectors.

Transformers: a aside

The "transformer", as such, does not actually have a formal definition. Many different architectures are now called "transformers".

- When the original transformer was introduced, the conventional thinking was still "encoder-decoder" (like our sequence-to-sequence model), so decoders were also introduced into the model.
- Since then, many models have dispensed with either the decoder half (BERT) of the model, or the encoding half (GPT).
- The minimum requirement to be a "transformer" appears to be the presence of self-attention.

Transformers: the decoder

So what does the decoder look like? It's actually very similar to the encoder.

- First, run the encoder's input data through the encoder. Take the output and transform it into "key" and "value" attention vectors.
- Take the current output sentence as the input to the decoder (which is just [START], when we start). Tokenize this and convert into a set of vectors, using an embedding layer.
- Add positional encoding to each word vector.
- Run the current output sentence through a masked self-attention layer.
- The self-attention layer is "masked", to make sure that each word only interacts ("pays attention to") words to its left.
- The output of the layer is added to a residual connection and normalized.

So far, this is the same as the encoder, except that the self-attention layer is masked.

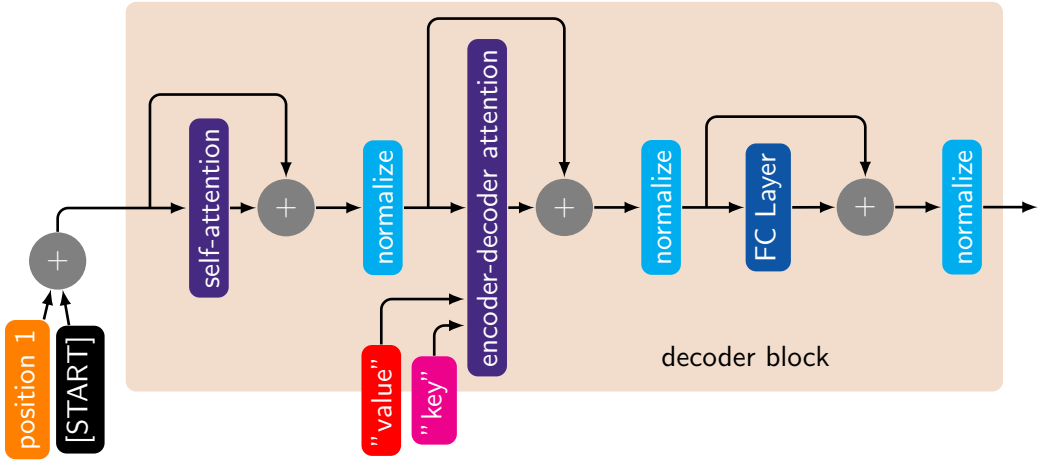
Transformers: the decoder, continued

But there's more...

- Next, feed the output into an encoder-decoder attention layer, which uses the "key" and "value" attention vectors from the encoder.
- The output is then added to a residual connection and normalized.
- This is then fed into a fully-connected layer.
- The output is then added to a residual connection and normalized.
- This is then run through a softmax layer, to pick the next word.
- This word is used to update the current output sentence, and the above sequence is repeated until the [STOP] symbol is produced.

In this case the encoder-decoder attention layer operates in a manner very similar to the sequence-to-sequence Attention models we saw earlier. Instead of using the internal states of an LSTM, we use the "key" and "value" attention vectors created by the encoder.

Transformers: the decoder, continued more



The output is then either passed to another decoder block, or into a fully-connected layer with a softmax output, to predict the next word.

Classes of models

Broadly speaking Transformer models fall into one of three categories:

- Autoregressive models:
  - Attempt to solve the classic NLP task: next word prediction, reading from the left.
  - Only use the Transformer's decoder. Usually are unidirectional.
  - Examples include: GPT, GPT-2, GPT-3, ChatGPT, XLNet, and others.
- Autoencoding models:
  - Are still solving the missing-word problem, but the word is often in the middle of a sentence. Training involves removing words from full sentences.
  - Only use the Transformer's encoder. Are usually bidirectional.
  - Examples include: BERT, ALBERT, RoBERTa, and others.
- Sequence-to-sequence models:
  - Usually used for full-sentence responses (translation, question-answering, summarization).
  - Uses the full Transformer (encoder and decoder).
  - Examples include BART, T5, and others.

SciNet

BERT (2018)

Bidirectional Encoder Representations from Transformers (BERT) was an early application of Transformers.

- The main body consists of a bidirectional encoder Transformer. No decoder is used.
  - 12 encoder-only Transformer blocks,
  - 1024 nodes in the fully-connected layers,
  - 12 self-attention heads. The self-attention is bidirectional.
  - About 340M free parameters.
- The two unsupervised tasks are used to pretrain BERT:
  - Masking: a sentence is given to the model as input, with 15% of the tokens "masked" out. The target is then the masked tokens.
  - Next sentence prediction: two sentences are input, categorize whether the second sentence actually follows the first.
- Once pretrained, BERT was fine-tuned for a number of tasks.
- Was state-of-the-art on 11 language processing tasks.

SciNet

GPT

The Generative Pretrained Transformer (GPT) models are a series of autoregressive models (only built of decoders) developed by OpenAI.

Model	year	Decoder blocks	Number of parameters
GPT	2018	12	117M
GPT-2	2019	48	1.5B
GPT-3	2020	96	175B
GPT-4	2023	unknown	unknown

ChatGPT is currently based on GPT-4o, which we also don't know the sizeo of.

We will use a version of GPT-2 for our example.

SciNet

Example, an aside

As you have probably ascertained, I'm not a big fan of black-box code.

- I prefer to show you how to build models, so that you can build them and play with them yourself.
- With modern transformer models this is too difficult. They're just too complex, and too big, and you don't want to train them from scratch anyway.
- The usual recommended way to get started with such models is the use the prebuilt models available from Hugging Face (`https://huggingface.co`), through the "transformers" package ("pip install transformers").
- We will use such a transformer for our example.

SciNet

Example

As with the RNN example, I'd like to build a tranformer that generates recipes.

- Rather than use my own data set, which is far too small, we will use the Epicurious data set.
  - Over 20,000 recipes (still too small?).
  - Includes ratings, nutritional information, and other goodies.
  - `https://www.kaggle.com/hugodarwood/epirecipes`.
- We'll use a pretrained version of GPT-2 (124M parameters).
- We will then fine-tune the model on the Epicurious recipes data set.

If you want a copy of this data set, in the manner in which I preprocessed it, let me know.

Example, continued

```
# Train_Recipes.py
from transformers import AutoTokenizer,
  AutoConfig, TFAutoModelForCausalLM
import datasets

MODEL_NAME = 'gpt2'
batch_size = 128
num_epochs = 200

base = 'epicurious.recipes.',
data_files = {'test': base + 'testing',
  'train': base + 'training',
  'validation': base + 'validation'}

data = datasets.load_dataset('text',
  data_files = data_files)
```

```
tokenizer = AutoTokenizer.from_pretrained(MODEL_NAME)
config = AutoConfig.from_pretrained(MODEL_NAME)
model = TFAutoModelForCausalLM.from_pretrained(MODEL_NAME)

t_data = data.map(tokenizer, batched = True)

train_data = t_data['train']
val_data = t_data['validation']

tf_train_data = train_data.to_tf_dataset(
  batch_size = batch_size, shuffle = True
  ).with_options(options)
tf_val_data = val_data.to_tf_dataset(
  batch_size = batch_size, shuffle = False
  ).with_options(options)
```

Example, continued more

Because we are using the Tensorflow implementation of the model, the model is compiled and fit using the Keras commands.

Note that I've simplified the code quite a bit for the slides. For the full gory details please see the code.

```
# Train_Recipes.py, continued

model.compile(optimizer = 'adam')

hist = model.fit(
  tf_train_data, validation_data = tf_val_data,
  epochs = num_epochs, verbose = 2)
```

Example, continued more

```
ejspence@mycomp ~> python Train_Recipes.py
Epoch 1/200
314/314 - 490s - loss:  1.6342 - val_loss:  1.7831
Epoch 2/200
314/314 - 464s - loss:  1.4900 - val_loss:  1.7759
Epoch 2/200
314/314 - 464s - loss:  1.3753 - val_loss:  1.7824
⋮
Epoch 198/200
314/314 - 490s - loss:  0.2412 - val_loss:  3.0661
Epoch 199/200
314/314 - 490s - loss:  0.2282 - val_loss:  3.1107
Epoch 200/200
314/314 - 464s - loss:  0.2168 - val_loss:  3.1658
ejspence@mycomp ~>
```

Example, generating recipes

```
# Generate_Recipes.py
from transformers import AutoTokenizer, TFAutoModelForCausalLM

# The 'output' directory is where the model is stored.
model = TFAutoModelForCausalLM.from_pretrained('output')

# Just use the standard gpt2 encoder.
tokenizer = AutoTokenizer.from_pretrained('gpt2')

encoded = tokenizer.encode("Spinach Salad with Warm Feta Dressing\n1 9-ounce bag fresh spinach
leaves\n5 tablespoons olive oil, divided\n1 medium red onion, halved, cut into 1/3-inch-thick
wedges with some core attached\n1 7-ounce package feta cheese, coarsely crumbled",
  return_tensors = 'tf')

generated = model.generate(encoded, max_length = 256)
print(tokenizer.decode(generated[0]))
```

Example, generating recipes

```
ejspence@mycomp ~> python Generate_Recipes.py
Spinach Salad with Warm Feta Dressing
1 9-ounce bag fresh spinach leaves
5 tablespoons olive oil, divided
1 medium red onion, halved, cut into 1/3-inch-thick wedges with some core attached
1 7-ounce package feta cheese, coarsely crumbled
1/2 cup chopped fresh basil
1/4 cup chopped fresh Italian parsley
1/4 cup chopped drained oil-packed sun-dried tomatoes
2 tablespoons Dijon mustard
$$
Preheat oven to 375°F. Sprinkle chicken with salt and pepper.  Heat 3 tablespoons oil in large
nonstick skillet over medium nonstick skillet over medium-high heat.  Add onion.  Sauté 5
minutes.  Add onion and sauté until golden, stirring frequently.  Add spinach; sauté until
tender, about 3 minutes.  Season with salt and pepper.  Stir in pepper.  Reduce heat to taste
with pepper.  Remove from heat.  Stir in pepper.  Stir in beans; cool.  Transfer to medium bowl.
Heat remaining 3 minutes.  Heat remaining 3 tablespoons oil.  Stir in sun-dough breadcracker
mixture to bowl.  Whisk vinegar; cool completely.  Stir in sun-dough breadcracker mixture to
```

Notes about the example

Some thoughts on our model:

- Very little data modification needed to be done prior to using. I took the liberty of adding separations between directions and ingredients, but that is optional.
- This model took over 1 day to train on a single GPU.
- The model has 124M parameters; there are only 3.6M words in the training data set. A bigger data set is needed.
- Careful examination of the output of the model indicates some memorization of the text (copied recipe names, lists of ingredients). This is also a symptom of overfitting.
- Nonetheless, the model gets much correct, fixing problems with our RNN model:
  - ingredients in the list are often referenced in the instructions, in order of appearance,
  - the title makes sense, given the ingredients,
  - the grammar is improved.

Overall, the model is better than our RNN model, but could be improved further.

SciNet

More-recent fine-tuning techniques

The technique used here is out of date.

- This approach attempted to fine-tune the entire model. Meaning that all the training parameters were available for updating.
- More-recent approaches for fine-tuning models on particular data sets involve a technique called LoRA (Low Rank Adaptation).
- This technique involves leaving the original 'base' model alone, but training 'side' weights and biases that modify the output of each attention block.
- This results in significantly fewer trainable parameters, as few as $<1\%$ of the base model.
- This allows successful fine-tuning on much smaller data sets.

If you need to fine-tune a model, look into implementing LoRA (or even QLoRA).

SciNet

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

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

SciNet

Linky goodness

Attention:

- https://arxiv.org/abs/1409.0473
- https://arxiv.org/abs/1508.04025
- https://arxiv.org/abs/1509.00685
- https://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention

Transformers:

- https://arxiv.org/abs/1706.03762
- https://arxiv.org/abs/1807.03819

SciNet

Linky goodness, continued

Models:

- BERT: https://arxiv.org/abs/1810.04805
- GPT: https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf
- XLNet: https://arxiv.org/abs/1906.08237
- RoBERTa: https://arxiv.org/abs/1907.11692
- GPT-2: https://d4mucfpksywv.cloudfront.net/better-language-models/language_models_are_unsupervised_multitask_learners.pdf
- ALBERT: https://arxiv.org/abs/1909.11942
- DistilBERT: https://arxiv.org/abs/1910.01108
- XLM: https://arxiv.org/abs/1901.07291
- BART: https://arxiv.org/abs/1910.13461
- T5: https://arxiv.org/abs/1910.13461
- GPT-3: https://arxiv.org/abs/2005.14165

SciNet