Neural network programming: attention and transformers

Erik Spence

SciNet HPC Consortium

14 May 2026

SciNet

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/1400`

Click on the link for the class, under "Lectures", click on "Attention networks".

Today's class

This class will cover the following topics:

- Attention networks.
- Transformers.
- Different classes of Transformers.
- Example.

Please ask questions if something isn't clear.

SciNet

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 we dealt with last class.

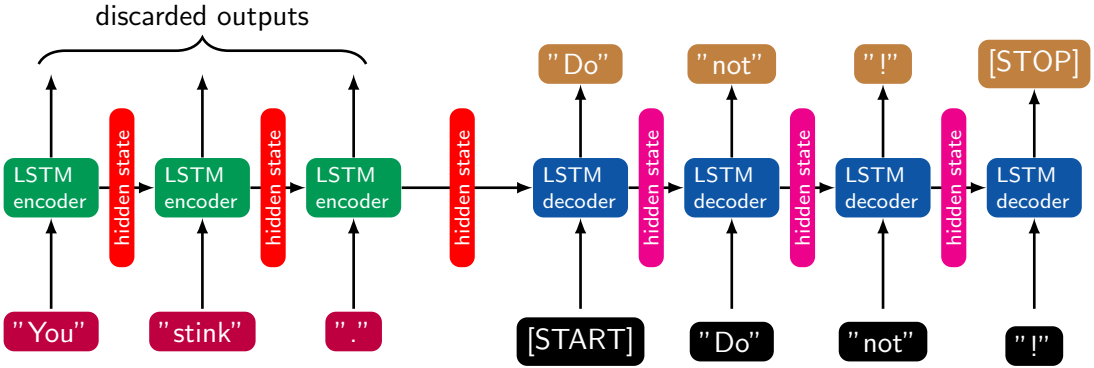
The first difference is that the encoder passes a lot more information to the decoder.

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

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

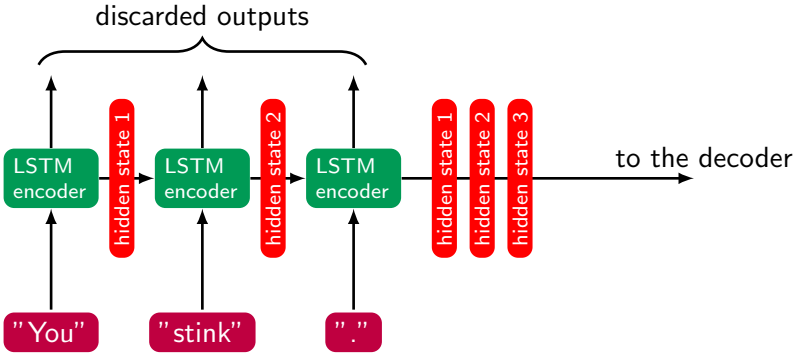
SciNet

Sequence-to-sequence networks



In our previous sequence-to-sequence model, only the final hidden state of the encoder was passed to the decoder.

Attention networks, first change



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

Attention models, continued

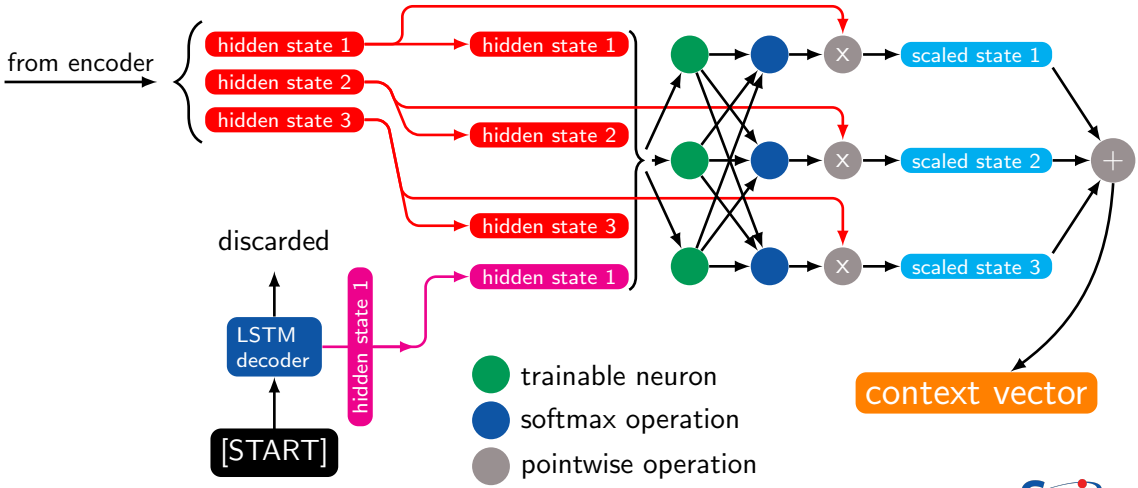
The second major difference between our previous sequence-to-sequence models and Attention models involves an extra decoder preprocessing step. For each output (decoder) step:

- The encoder's hidden states and the current hidden 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 hidden states are each multiplied by their respective output from the above neural network. This amplifies the hidden 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 hidden 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.

SciNet

Attention networks, second change



Context vectors

Context vectors are an interesting innovation.

- Because the little neural network uses softmax as its output, you can examine its output to see which hidden 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:

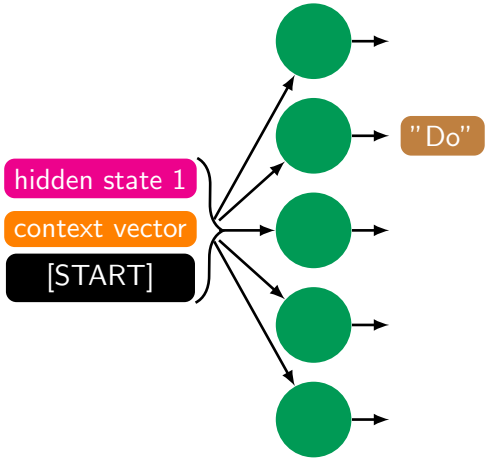
- We use it to predict the next word:
  - The context vector is concatenated with the decoder's hidden state.
  - We pass this concatenated vector, along with the decoder's input data, through yet another single-layer neural network.
  - The output of this neural network is softmax, indicating the next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

SciNet

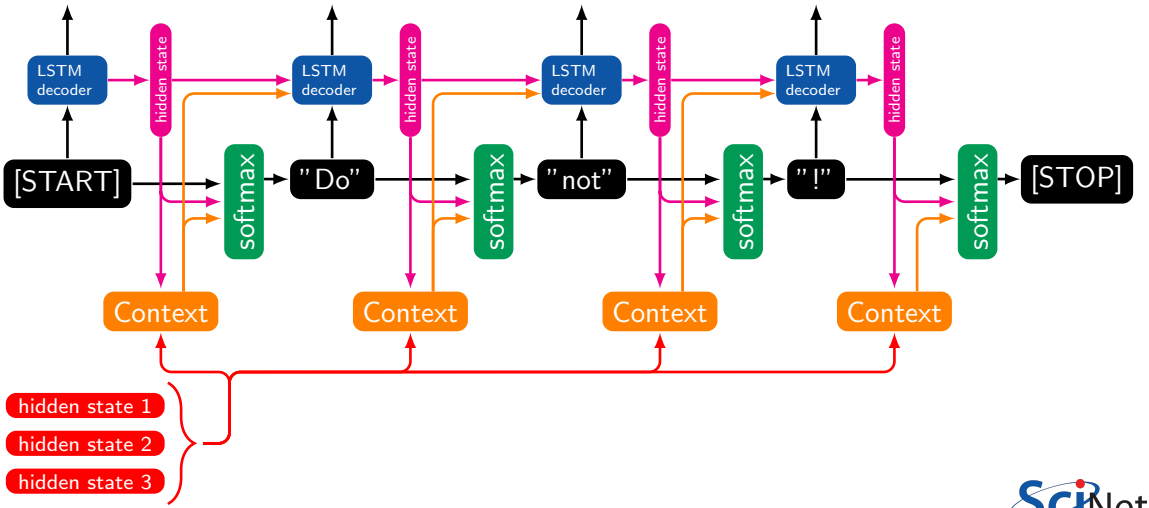
Attention networks, second change, continued more

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.

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

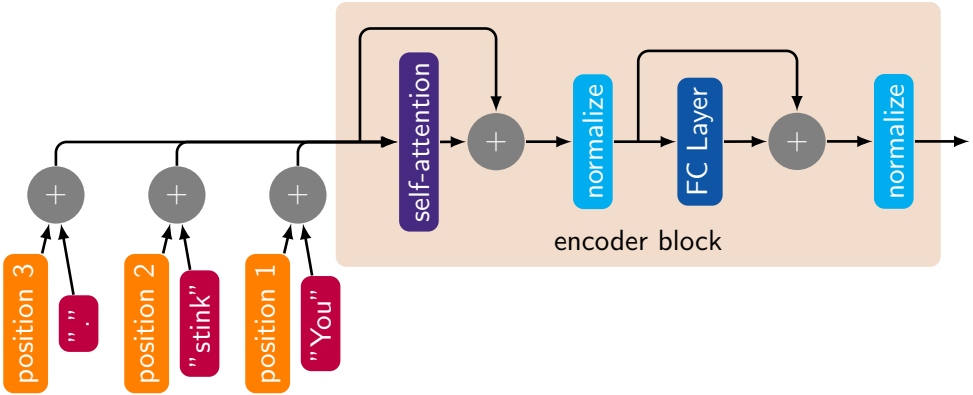
So what does the encoder look like?

- 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.

SciNet

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.

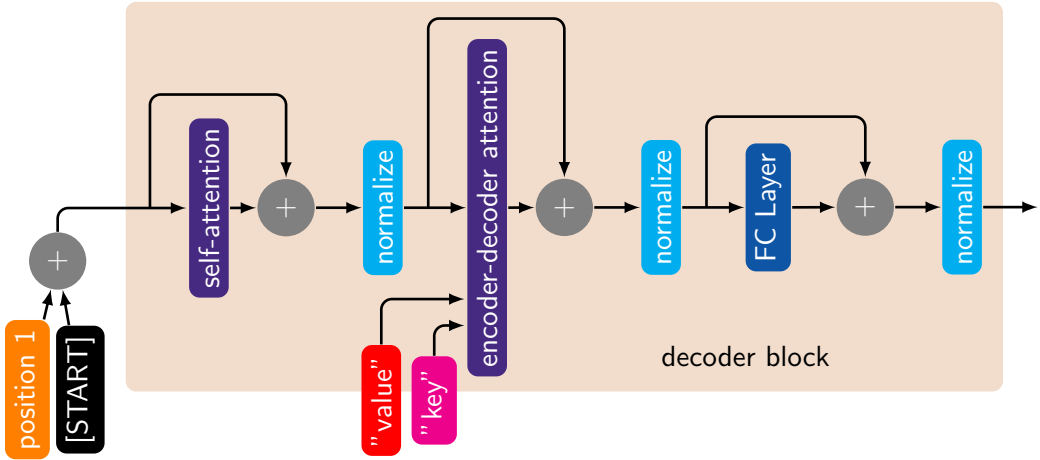
Transformers: the decoder, continued

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
GPT-4.5	2025	unknown	unknown

ChatGPT is currently based on GPT-5.5, which we also don't know the size 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").
- The transformers package no longer supports TensorFlow, so you'll also need to install torch for this to work.
- We will use such a transformer for our example.

SciNet

Example

As with the example in the RNNs class, 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, AutoModelForCausalLM,
  TrainingArguments, Trainer
import datasets, evaluate, diffusers

MODEL_NAME = 'gpt2'
batch_size = 8
num_epochs = 100

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 = AutoModelForCausalLM.from_pretrained(MODEL_NAME)

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

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

Example, continued more

Because we are using the transformers package the model is compiled and fit using a different set of 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

training_args = TrainingArguments(
  output_dir = "./results",
  num_train_epochs = num_epochs,
  per_device_train_batch_size = batch_size)

trainer = Trainer(model = model,
  args = training_args,
  train_dataset = train_dataset,
  processing_class = tokenizer,
  data_collator = data_collator,
  compute_metrics = compute_metrics)

result = trainer.train()
```

SciNet

Example, continued more

```
ejspence@mycomp ~> python Train_Recipes.py
{'loss': '2.795', 'grad_norm': '6.408', 'learning_rate': '4.999e-05', 'epoch': '0.02403'}
{'loss': '2.541', 'grad_norm': '4.603', 'learning_rate': '4.998e-05', 'epoch': '0.04805'}
{'loss': '2.472', 'grad_norm': '4.085', 'learning_rate': '4.996e-05', 'epoch': '0.07208'}
⋮
{'loss': '0.3174', 'grad_norm': '2.196', 'learning_rate': '2.645e-08', 'epoch': '99.95'}
{'loss': '0.3168', 'grad_norm': '2.921', 'learning_rate': '1.444e-08', 'epoch': '99.97'}
{'loss': '0.3165', 'grad_norm': '2.696', 'learning_rate': '2.427e-09', 'epoch': '100'}
ejspence@mycomp ~>
```

Example, generating recipes

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

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

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

encoded = tokenizer("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 = 'pt')

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 min-
utes.  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 pep-
per.  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

Further notes

Some further thoughts on model finetuning.

- Modern models are often too big to fit onto a single GPU.
- Techniques have been developed to squish large models onto single GPUs. Quantization is the most-commonly used technique in this space.
- Techniques for speeding up the fine-tuning of models have also been developed. The most common of these is LoRA (Low-Rank Adaptation). LoRA does "side-training" of small numbers of weights beside the original model's weights, so there's less training to be done.
- These two techniques are often combined: QLoRA.

Using QLoRA, large models can be trained fairly quickly.

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