Assignment 3: RNN Language Model

SOLUTIONS - do not distribute!

Overview

- **(a)** RNNLM Inputs and Parameters (written questions)
- **(b)** Implementing the RNNLM
- **(c)** Training your RNNLM
- **(d)** Sampling Sentences
- **(e)** Linguistic Properties

This notebook contains solutions to **part (a)** and **part (e)**, as well as a few observations about the coding sections **(b), (c), and (d)**.

RNNLM Model Structure

RNNLM

Here's the basic spec for our model. We'll use the following notation:

- $w^{(i)}$ for the i^{th} word of the sequence (as an integer index)
- $x^{(i)}$ for the vector representation of $w^{(i)}$
- $h^{(i)}$ for the i^{th} hidden state, with indices as in Section 4.8 of the async
- $o^{(i)}$ for the i^{th} output state, which may or may not be the same as the hidden state
- $y^{(i)}$ for the i^{th} target word, which for a language model is always equal to $w^{(i+1)}$

Let $h^{(-1)} = h^{init}$ be an initial state. For an input sequence of n words and $i = 0, \ldots, n-1$, we have:

- **Embedding layer:** $x^{(i)} = W_{in}[w^{(i)}]$
- **Recurrent layer:** $(o^{(i)}, h^{(i)}) = \mathrm{CellFunc}(x^{(i)}, h^{(i-1)})$
- **Output layer:** $\hat{P}(y^{(i)}) = \hat{P}(w^{(i+1)}) = \mathrm{softmax}(o^{(i)} W_{out} + b_{out})$

$\mathrm{CellFunc}$ can be an arbitrary function representing our recurrent cell - it can be a simple RNN cell, or something more complicated like an LSTM, or even a stacked multi-layer cell. *Note that the cell has its own internal, trainable parameters.*

It may be convenient to deal with the logits of the output layer, which are the un-normalized inputs to the softmax:

$$\mathrm{logits}^{(i)} = o^{(i)} W_{out} + b_{out}$$

We'll use these as shorthand for important dimensions:

- `V` : vocabulary size
- `H` : hidden state size = embedding size = per-cell output size

(a) RNNLM Inputs and Parameters

SOLUTIONS - do not distribute!

(written - no code) Write your answers in the markdown cell below.

You should use big-O notation when appropriate (i.e. computing $\exp(\mathbf{v})$ for a vector $\mathbf{v}$ of length n is $O(n)$ operations).

Note: for $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{n\times l}$, computing the matrix product AB takes $O(mnl)$ time.

Note: We've given detailed solutions below, but your answers need not be so long.

1.

Let CellFunc be a simple RNN cell (see Section 4.8). Write the functional form in terms of nonlinearities and matrix multiplication. How many parameters (matrix or vector elements) are there for this cell, in terms of `V` and `H`?

Solution: For a simple RNN,

$$o^{(i)} = h^{(i)} = \tanh(W_{cell}[x^{(i)}, h^{(i-1)}] + b_{cell})$$

There are a number of equivalent ways to write this; the notation from the async uses $\tanh(WM + B)$, where $M = [x^{(i)}, h^{(i-1)}]$. Note that if you write this, you need to be careful to distinguish the cell W from the weight matricies in the embedding and output layers.

In terms of parameters, the matrix W_{cell} maps a concatenated pair of H-length vectors into a vector of length H, so it must have shape $H \times 2H$. The bias term b_{cell} is a vector of length H, which gives us $2H^2 + H = O(H^2)$ parameters.

2.

How many parameters are in the embedding layer? In the output layer? (By parameters, we mean total number of matrix elements across all train-able tensors.)

Solution: In the embedding layer, we have a single embedding table W_{in} that maps each word in our vocabulary to a vector of length H; thus it has $H \times V = O(HV)$ parameters.

In the output layer, we're effectively performing logistic regression on V classes (the words of the vocabulary) using a feature vector of length H. Thus we need a $H \times V$ weight matrix (or the transpose), and a V-length bias vector (one bias per class), for $HV + V = O(HV)$ parameters.

3.

How many floating point operations are required to compute $\hat{P}(w^{(i+1)})$ for a given target word $w^{(i+1)}$, assuming $w^{(i)}$ given and $h^{(i-1)}$ already computed? How about for all target words?

Solution: Big-O notation is important here; otherwise it's easy to get lost in the details of the individual operations. Let's look at each layer:

- **Embedding layer:** $x^{(i)} = W_{in}[w^{(i)}]$ is a lookup of an H-length vector, so $O(H)$
- **Recurrent layer:** $(h^{(i)}, o^{(i)}) = \text{CellFunc}(x^{(i)}, h^{(i-1)})$ takes inputs of size H and emits outputs of size H. If we do this with matrix multiplication, it's some (small) number of $H \times H$ matricies, plus bias terms and element-wise nonlinearities on vectors of length H. The matrix multiplication dominates, so we get $O(H^2)$.
- **Output layer:** $\hat{P}(y^{(i)}) = \hat{P}(w^{(i+1)}) = \text{softmax}(o^{(i)} W_{out} + b_{out})$. We can compute the un-normalized logit for a particular word j here as $o^{(i)} w_{out,j} + b_j$, which takes $O(H)$ time for a dot product. However, in order to compute the probability, we need to take the softmax over *all* words - for a total of V dot-products that constitute the matrix-vector product $o^{(i)} W_{out}$. This takes $O(HV)$ time. This is followed by an $O(V)$ sum and $O(1)$ division, but the $O(HV)$ term dominates.

In general, $V >> H$, and so the $O(HV)$ term dominates. However, it's important not to forget about the H^2 term, as we'll see in 5. So our final answer should properly be $O(HV + H^2)$.

If we want to predict probabilities for all target words, we only have to do $O(V)$ additional work over the single-target case, since the bulk of the work was already done to compute the softmax normalization. So, our answer is still $O(HV + H^2)$.

4.

How does your answer to 3. change if we approximate $\hat{P}(w^{(i+1)})$ with a sampled softmax with k samples? How about if we use a hierarchical softmax? (*Recall that hierarchical softmax makes a series of left/right decisions using a binary classifier* $P_s(\text{right}) = \sigma(u_s \cdot o^{(i)} + b_s) \geq 0.5$ *at each split* s *in the tree.*)

Solution: If we use a sampled softmax, then instead of normalizing over the entire vocabulary we just sample k words at random, compute the logits, and normalize. Since we're only computing k dot products, the work is only $O(Hk) << O(HV)$.

Note: strictly speaking, there's an $O(V)$ *step in performing the sampling - but this is very fast and can be done in parallel with the other operations. For* $V \approx 10^4$ *and typical* $H \approx 10^2 - 10^3$ *and* $k \approx 10^2$, $O(Hk)$ *is similar to* $O(V)$.

If we use a hierarchical softmax (with a fixed tree), then for each word we need to follow a path from the root to the leaf of that tree and compute a logistic regression with our H-vector $o^{(i)}$ at each node. Unless the tree is extremely unbalanced, this path will be of length $O(\log V)$, and so our total computation will be $O(H \log V) << O(HV)$.

5.

If you have an LSTM with $H = 200$ and use sampled softmax with $k = 100$, what part of the network takes up the most computation time during training? (*Choose "embedding layer", "recurrent layer", or "output layer"*)

Solution: We can read this off from our answer to 3.

- **Embedding layer:** $O(H) = O(200)$ operations
- **Recurrent layer:** $O(H^2) = O(200^2) = O(40,000)$ operations
- **Output layer:** $O(Hk) = O(200 \times 100) = O(20,000)$ operations

Furthermore, we know that the LSTM has four internal matrix multiplications of size $2H \times H$ each, so we end up with around $O(8 \times 200^2) = 320,000$ operations in the recurrent layer at each timestep - far more than the output layer!

In fact, it's very common to operate in this regime, such that the recurrent layer is the bottleneck. Even without approximating the softmax, for small vocabularies (10-30k) the recurrent layer can still be slower because the operations (even within an LSTM) must be run sequentially - whereas a large softmax can be efficiently parallelized.

(b) Implementing the RNNLM

In order to better manage the model parameters, we'll implement our RNNLM in the `RNNLM` class in `rnnlm.py`. We've given you a skeleton of starter code for this, but the bulk of the implementation is left to you.

Particularly, you'll need to implement three functions:

- `BuildCoreGraph()` : the main RNN itself
- `BuildTrainGraph()` : the training operations, including `train_loss_`, and `train_step_`
- `BuildSamplerGraph()` : operations to generate output samples (`pred_samples_`)

See `rnnlm.py` for more documentation.

Notes and Observations

`BuildCoreGraph`

- Generally, you want to initialize bias terms to 0-vectors, and weight matricies (including embeddings) to some type of random noise.
- Don't forget the bias term b_{out} on the output layer!
- A few people had extraneous calls to `tf.sigmoid` in the output layer. Remember that the softmax function handles normalization internally, so you only need to feed it the un-normalized outputs of the matrix multiplication (+ bias).

`BuildTrainGraph`

- Per the initial instructions, we gave full credit if you just used the normal softmax loss for training. (Although, we apologize for how long it may take to train...)
- `tf.nn.sampled_softmax_loss` (https://www.tensorflow.org/versions/r0.10/api_docs/python/nn.html#sampled_softmax_loss) has a somewhat strange API, and you do need to transpose your weight matrix and flatten your inputs to 2D to get it to work.

`BuildSamplerGraph`

- The output shape of `[batch_size, max_time, 1]` translates to producing one sample for each predicted timestep, for each batch entry. You don't need to do any extra slicing in TensorFlow to extract the last one - the starter code in `sample_step` does this for you.

(c) Training your RNNLM

Notes and Observations

`run_epoch`:

- Don't forget to pass the learning rate to your model!
- The `if i == 0:` clause in the starter code fetches a fresh initial state at the beginning of an epoch - however, that doesn't mean you shouldn't train on that first batch! A number of implementations had their training code inside an `else:` block here, which isn't necessary.
- Don't forget to handle `final_h_` and pass the state forward, even if `train=False`.

(d) Sampling Sentences

Notes and Observations

Most people did pretty well here. This part is a good sanity check if your model trained well - you should get reasonably-fluent output, at least on par with the n-gram models we built in Week 2.

If your model outputs large numbers of rare words, it's possible there's a bug in your training code.

(e) Linguistic Properties

1. Number agreement

Compare **"the boy and the girl [are/is]"**. Which is more plausible according to your model?

If your model doesn't order them correctly (*this is OK*), why do you think that might be? (answer in cell below)

Solution: There's no right answer to this, and we awarded full-credit to answers that were plausible and insightful about your model.

It's hard to get a model to pick the right answer here, and it's quite reasonable to suspect that our models - which aren't trained for very long - will only pick up on the short-term dependency "the girl is". If this is a more likely trigram, then that might take precedence over the long-range agreement.

In [15]:
```
#### YOUR CODE HERE ####


load_and_score([s.split() for s in sents], sort=True)

#### END(YOUR CODE) ####
```

```
WARNING:tensorflow:<tensorflow.python.ops.rnn_cell.BasicLSTMCell object at 0x
7fd4362594d0>: Using a concatenated state is slower and will soon be deprecat
ed.  Use state_is_tuple=True.
```

```
"the boy and the girl is" : -24.58
"the boy and the girl are" : -26.37
```

2. Type/semantic agreement

Compare:

- **"peanuts are my favorite kind of [nut/vegetable]"**
- **"when I'm hungry I really prefer to [eat/drink]"**

Of each pair, which is more plausible according to your model?

How would you expect a 3-gram language model to perform at this example? How about a 5-gram model? (answer in cell below)

Solution: Again, no single right answer here. Our reference model was able to get these right, but only by a small margin - not enough to conclusively say that it's capturing the long-range dependency.

The local n-grams in these examples are more ambiguous, and you could argue either way for which is more common in the corpus: maybe "favorite kind of vegatable" is a more likely 4-gram, but "kind of nut" is a more common 3-gram - and so which wins would depend on your model. Regardless, neither a 3- or a 5-gram model would be able to capture the dependency between "peanuts ... nut". We'd need a 7-gram model, which suffers from horrible sparsity issues - it would only learn this dependency by memorizing the entire 7-word input.

In the second case, a 6-gram model would work - although because I originally miscounted myself, we gave credit for saying a 5-gram model would capture the "hungry ... eat" dependency.

In [16]:

```
#### YOUR CODE HERE ####



#### END(YOUR CODE) ####
```

```
WARNING:tensorflow:<tensorflow.python.ops.rnn_cell.BasicLSTMCell object at 0x
7fd46c0f3f50>: Using a concatenated state is slower and will soon be deprecat
ed.  Use state_is_tuple=True.
WARNING:tensorflow:<tensorflow.python.ops.rnn_cell.BasicLSTMCell object at 0x
7fd49c382050>: Using a concatenated state is slower and will soon be deprecat
ed.  Use state_is_tuple=True.
```

```
"peanuts are my favorite kind of nut" : -44.19
"peanuts are my favorite kind of vegetable" : -45.14
"when I'm hungry I really prefer to eat" : -54.83
"when I'm hungry I really prefer to drink" : -56.49
```