CSC321: Assignment 3

Due on Monday, March 21, 2016

Davi Frossard

April 2, 2016

Part 1

In order to implement sampling from the RNN with temperature, all we must do is modify the softmax equation in the provided sampling function to account for the temperature. We do so in Code 1.

Code 1: Implementation of Text Sampling Function With Temperature.

```python
def sample_rnn(self, seed, sample_length, temperature=1):
  '''
  Samples a text from the RNN using seed as the initial exciter.
  :param seed: Initial exciter
  :param sample_length: Length of the sample to be produced
  :param temperature: Softmax temperature
  :return: Produced sample
  '''
  self.temperature = temperature
  x = self.one_hot(self.char_to_ix[seed])
  sample = ''
  for t in xrange(sample_length):
    self.hprev = np.tanh(np.dot(self.Wxh, x) + np.dot(self.Whh, self.hprev) + self.bh)
    y = np.dot(self.Why, self.hprev) + self.by
    p = np.exp(y/self.temperature) / np.sum(np.exp(y/self.temperature))
    ix = np.random.choice(range(self.vocab_size), p=p.ravel())
    sample += self.ix_to_char[ix]
    x = self.one_hot(ix)

  return sample
```

With this implementation, we draw samples with temperatures in [0.1, 0.5, 0.7, 1.0, 1.5], obtaining the results shown in Code 2-6.

Code 2: Samples With Temperature 0.100000.

```
ve the the the the the the the the the the the the the consuse the the the the the the
     come the the the the the the the the the the the the the the the the the the the
    the the the the the the the the
-----------------------------------------

e the the the the the the the the the the the the the the the the the the the the the
    the the the const the the the the the the the the the the the the the the the the
    the the the the the the the the
-----------------------------------------

the the the the the the the the the the the the the the the the the the the the the the
    the the the the the the the the the the the consuse the the the the the the the the
     the the the the the the the the
```

Code 3: Samples With Temperature 0.500000.

```
not the are the con the say the not kingme my and the did the cound what not the the
    son the consizen:
But to and the shander say the good all the with peake peake when the were of he his
    he to when,
-----------------------------------------
```

```

er of the pats the that the with the to in to and the with the than he the not would
    death he say the pray a wOR:
And you the comon to for of be prens the the cheathous whet these to as the the consur
-----------------------------------------

n the the to have the not sencingtherer he the the the of my more your enven dine the
    me it to so and the stand the atcher as me the titt to have of you cond he he agery
    ,
And this he he the not so cay
```

Code 4: Samples With Temperature 0.700000.

```
now your the s to done with sene you wounst. und deant furting the bentuged he to to
    thet in not bones.

CORIOLANUS:
Brear you renome coull here the stars or lutsing the headn, his the were let for gi
-----------------------------------------

ath house fill the to of cortunces:
It on king have some is him the the us the the in like but in not aster me?

First in our all and not on the goost with hathering bithing prout are falr marsies
    dow
-----------------------------------------

penter more gods, this the time my not to barped to sey and and if one to will heavin
    the hearthing you prode anole, the we me that he to not the man he the thou not in
    to more son, and bloogh the wou
```

Code 5: Samples With Temperature 1.000000.

```
ABEThous.
Pramertherply.

Firlo asperseving:
A rees werain,
Youe?

SAESad, winout-me eveokn wit he his tive flegers tak bepersere one ceencespar the
    inste to with yet’s, if hear as digom:
Rerves.
Whou
-----------------------------------------

, you thet jusk you goospus feed bedterd mathyousings say be’s hison Put of vill, I
    thy men te kinst my ciuch desie to not him with mings.
To pladice a wo’st.
Whet upbethyever?

BUCKINGHES:
To kinger
-----------------------------------------
```

```

rey foratizuck at strires. Go,
The sinerle in for man gake: nobined to enownteriteran trame doth you.
No thy not entwe

Firdiun the you, ch nafoifieng,
Be, Thou be his, this of fors,
Are to I world.
```

Code 6: Samples With Temperature 1.500000.

```
taok
ped knk ruther if;
No bereme diss?.

CORGORIWSSTER:
Gwak!

Ferstacchers'
wey'tles you hewelereiseir
Gyolonougst for kome plot h: Cassoud
sust:
Whi'd.
Teres ge, po of shseinesnar lete.
fe?

Sezile
----------------------------------------

plakprind, now winl you jidvingly?

GLOUCECETRS:
Ss ampoly's this
Lrorl azeYouse freo:.

GLOULANATAS:
Ye helcuxt s

her 'ppusent she,
Gepay,
Citutireragn.

Ceref to hos.
Addeltl ot tour neikigst qulid
----------------------------------------

me,
A prits:
W.s
Cinsan bettuns wrorks, wo.

IR:
Migizetur.
And' or in offorn: weTit. ibciths
```

```
Evakisl.s.

Thirno!
werchrenf Cceizesto that obBirower: ance, fan Noble, genghe dup.

Savether y forr'd
M
```

We notice that smaller values of temperature produces only the most likely outputs, which gives us essentially just a chain of "the", not an interesting result at all.

As we increase the temperature, the softmax function returns a distribution less concentrated around the most likely output, which in turn produces samples more interesting and that closely resemble the original text. Values from 0.5 to 1.0 seem to produce the best results, as seen in Code 3-5.

However, if we increase the temperature too much, the distribution starts to converge to a uniform, with outputs that are very unlikely being generated often. Code 6 shows samples in which most words are not valid and the general structure of the text does not resemble the original data.

Part 2

To complete a phrase, we must first excite the RNN with each letter in it in order to have the state of the RNN at the end reflect the encoding of the phrase. We do so by changing the function implemented in Part 1 so that the seed is a string (not just a single character) and make the RNN generates a minimum of *minlength* characters and stop after generating a character that symbolizes an ending (".", "!", "? " or a newline). With these specifications we get the implementation listed in Code 7.

Code 7: Implementation of Phrase Completion Function.

```
def complete_phrase(self, phrase, temperature=1, minlenght=100):
  '''
  Completes a given phrase using RNN samples.
  :param phrase: Phrase to be completed
  :return: Phrase with completion
  '''
  self.temperature = temperature
  phrase_ix = [self.char_to_ix[ch] for ch in phrase]
  #Compute states for sequence
  for chr in phrase_ix:
    self.excite_rnn(chr)

  # Compute first letter after phrase
  y = np.dot(self.Why, self.hprev) + self.by
  p = np.exp(y/self.temperature) / np.sum(np.exp(y/self.temperature))
  ix = np.random.choice(range(self.vocab_size), p=p.ravel())

  # Keep generating letters:
  sample = self.ix_to_char[ix]
  len = 0
  while True:
    p = self.get_prob(ix)[0]
    ix = np.random.choice(range(self.vocab_size), p=p.ravel())
    char = self.ix_to_char[ix]
    len += 1
    if char not in ['.', '\n', '!', '?'] or len < minlenght:
      sample += char
    else:
      sample += char
      break
  return phrase+sample
```

We do not consistently get meaningful sentences running this function in generic phrases. However, playing with the temperature and seeds we get somewhat interesting results:

The answer to life the universe and everything is hell. And to this hear blood you to the counterst behertererter to to not have.

Interesting apocalyptic approach to the subject, not the 42 we were expecting.

To be or not to be peanut. Second thishing heavence, we dont'd farsite like me have fre?

CORIOLANUS:

He the com thene the the bood the so the sun be all this a

This one is specially interesting considering that the word "peanut" does not appear in the training set. It then proceeds to use structures known from the text.

CORIOLANUS:

And let is they with that ther the not is ap hoar,

I dode to the the him Cay.

I that man, who go dearther our this and a have had,

and he more no the revenster to he wonds I thim in they me.

We notice that using character names as the starting string consistently produces text that closely resemble the structure of the original text (albeit they might not really make sense).

Neural networks writing Shakespeare is him wath he held the made of to a him the noble

The ancity in pare all the had more he porst 'tben lith: prood

---

Strive not to be a success, but rather to be the greater half

But O'lredagher?

MENENIUS:

I vers combane at he more he he format; we thretredistrr atsion lood, wit?

It's interesting that it actually got the "to be" completion to the quote right, although the rest makes little sense.

Part 3

To find the most relevant weights in determining that a certain sequence is likely to be generated, in this case a newline *"\n"* after a colon *":"*, we first find the behaviour of the RNN's state by averaging out multiple samples of it after feeding a colon. We then compute the product between the State-to-Output weights (*Why*) connected to the newline output and select the entries corresponding to the 10 biggest values. Afterwards we check the Input-to-State weights (*Wxh*) at the entries found beforehand and remove the ones that point to negative or below the average weights.

With this method, we're left with weights that mostly contribute to the probability of generating a newline after a colon. The implementation is shown in Code 8.

Code 8: Implementation of Text Sampling Function With Temperature.

```python
def test_sequence(self, init, next, samples=500):
  init_ix = self.char_to_ix[init]
  next_ix = self.char_to_ix[next]

  self.excite_rnn(init_ix)
  hprev_avg = self.hprev
  p = self.get_prob(init_ix)[0]

  chars = np.array(self.ix_to_char.values())
  for i in range(samples):
    # "Reshuffle" RNN state
    self.sample_rnn(chars[np.random.randint(0,len(chars))], 10)

    # Compute state after feeding init_ix
    self.excite_rnn(init_ix)
    hprev_avg = (hprev_avg + self.hprev)/2

  pred = hprev_avg.ravel()*self.Why[next_ix,:]
  ibprev = np.argsort(pred)[::-1][:10]
  avg_wxh = np.mean(self.Wxh[:, init_ix])
  best_weights = [i for i in ibprev if self.Wxh[i, init_ix] > 0
                                    and self.Wxh[i, init_ix] > avg_wxh]


  print_info("Hypothesis with %.2f%% probability" % (p[next_ix]*100))
  nexchar = np.argmax(p)
  genchar = self.ix_to_char[nexchar]
  print_info("\t%s with highest probability (%.2f%%) after %s"
           %(repr(genchar), p[nexchar]*100, repr(init)))
  if(genchar != next):
    print_info("\t%s with %.2f%% probability"
             %(repr(next), p[next_ix]*100))
  print "Weights Involved:"
  print "\tWxh at [:,%d]" % init_ix
  print "\tWhy at [%d,:]" % next_ix

  print "Most relevant weighs:"
  print "\tWxh at [[%s], %d]" %(', '.join((str(s) for s in best_weights)), init_ix)
  print "\tWhy at [%d, [%s]]" %(next_ix, ', '.join((str(s) for s in best_weights)))

  return best_weights
```

Using the function proposed, we get the results listed in Code 4. If we check the pre-softmax output at these coordinates, it indeed mostly corresponds to the biggest values, reinforcing our belief in the method.

Code 9: Most relevant weights in generating a newline after a colon

```
Input to State Weights: [[100, 187, 108, 114, 185], 9]
  4.82918986837, 4.54751884983, 0.740072143331, 4.26500286018, 4.17333934191

State to Output Weights: [0, [100, 187, 108, 114, 185]]
  2.67271236039, 2.03015431333, 1.42691148871, 1.34150860805, 1.39779537745
```

Part 4

To find other character associations, we generate the output probabilities after a character over a large amount of samples, and then return the one that appears most frequently, following the implementation shown in Code 10.

Code 10: Implementation of Text Sampling Function With Temperature.

```python
def find_association(self, chr, temperature=1, numsamples=1000):
  a = []
  ix = self.char_to_ix[chr]
  chars = self.ix_to_char
  for i in range(numsamples):
    p = self.get_prob(ix)[0]
    # Reshuffle state
    self.sample_rnn(chars[np.random.randint(0,len(chars))], 10)
    ipmax, pmax = np.argmax(p), np.max(p)
    genchar = self.ix_to_char[ipmax]
    a.append(genchar)
  return max(set(a), key=a.count)
```

Iterating over all the characters known to the RNN, we get the following results:

```
'\n [ 0] -> E [14]'
'   [ 2] -> t [56]'
'! [ 1] -> \n [ 0]'
'& [ 4] -> C [12]'
"' [ 3] ->    [ 2]"
', [ 6] ->    [ 2]'
'- [ 5] -> w [57]'
'. [ 7] -> \n [ 0]'
': [ 9] -> \n [ 0]'
'; [ 8] -> \n [ 0]'
'? [10] -> \n [ 0]'
'A [11] -> R [29]'
'B [13] -> Y [34]'
'C [12] -> E [14]'
'D [15] ->    [ 2]'
'E [14] -> Y [34]'
'F [17] -> : [ 9]'
'G [16] -> L [23]'
'H [19] -> : [ 9]'
'I [18] -> U [30]'
'J [21] -> e [39]'
'K [20] -> : [ 9]'
'L [23] -> I [18]'
'M [22] -> A [11]'
'N [25] -> I [18]'
'O [24] -> L [23]'
'P [27] -> : [ 9]'
'Q [26] -> : [ 9]'
'R [29] -> : [ 9]'
'S [28] -> : [ 9]'
'T [31] -> : [ 9]'
'U [30] -> S [28]'
'V [33] -> : [ 9]'
'W [32] -> o [49]'
'Y [34] -> : [ 9]'
'Z [35] -> u [55]'
'a [36] -> n [50]'
'b [38] -> e [39]'
'c [37] -> e [39]'
'd [40] -> s [53]'
'e [39] ->    [ 2]'
'f [42] -> e [39]'
'g [41] -> i [43]'
'h [44] -> e [39]'
'i [43] -> n [50]'
'j [46] -> e [39]'
'k [45] ->    [ 2]'
'l [48] ->    [ 2]'
'm [47] ->    [ 2]'
'n [50] -> e [39]'
'o [49] -> t [56]'
'p [52] -> o [49]'
'q [51] -> u [55]'
'r [54] -> e [39]'
's [53] ->    [ 2]'
't [56] -> e [39]'
'u [55] -> s [53]'
'v [58] -> e [39]'
'w [57] -> e [39]'
'x [60] -> a [36]'
'y [59] ->    [ 2]'
'z [61] -> e [39]'
```

Its particularly interesting the chain formed between $U \to S \to :$ considering that most character names end with "US" (Brutus, Menenius, Sicinius, Coriolanus - in fact the chain can be extended to $N \to I \to U \to S \to :$ with these name patterns) and it is usually followed by a colon in the text. So we apply the same method used in Part 3 in the sequence $S \to :$ and obtain the results shown in Code 11.

Code 11: Most relevant weights in generating a newline after a colon

```
Input to State Weights: [[187, 176, 106], 28]
  2.50890337864, 4.52003972612, 0.474918435699

State to Output Weights: [9, [187, 176, 106]]
  1.32354504958, 1.21579921647, -0.969110747683
```

Checking the outputs, we can ascertain the same probabilities observed in Part 3.