karpathy / min-char-rnn.py
Last active 13 hours ago

Minimal character-level language model with a Vanilla Recurrent Neural Network, in Python/numpy

min-char-rnn.py

```python
"""
Minimal character-level Vanilla RNN model. Written by Andrej Karpathy (@karpathy)
BSD License
"""
import numpy as np

# data I/O
data = open('input.txt', 'r').read() # should be simple plain text file
chars = list(set(data))
data_size, vocab_size = len(data), len(chars)
print 'data has %d characters, %d unique.' % (data_size, vocab_size)
char_to_ix = { ch:i for i,ch in enumerate(chars) }
ix_to_char = { i:ch for i,ch in enumerate(chars) }

# hyperparameters
hidden_size = 100 # size of hidden layer of neurons
seq_length = 25 # number of steps to unroll the RNN for
learning_rate = 1e-1

# model parameters
Wxh = np.random.randn(hidden_size, vocab_size)*0.01 # input to hidden
Whh = np.random.randn(hidden_size, hidden_size)*0.01 # hidden to hidden
Why = np.random.randn(vocab_size, hidden_size)*0.01 # hidden to output
bh = np.zeros((hidden_size, 1)) # hidden bias
by = np.zeros((vocab_size, 1)) # output bias

def lossFun(inputs, targets, hprev):
  """
  inputs,targets are both list of integers.
  hprev is Hx1 array of initial hidden state
  returns the loss, gradients on model parameters, and last hidden state
  """
  xs, hs, ys, ps = {}, {}, {}, {}
  hs[-1] = np.copy(hprev)
  loss = 0
  # forward pass
  for t in xrange(len(inputs)):
    xs[t] = np.zeros((vocab_size,1)) # encode in 1-of-k representation
    xs[t][inputs[t]] = 1
    hs[t] = np.tanh(np.dot(Wxh, xs[t]) + np.dot(Whh, hs[t-1]) + bh) # hidden state
    ys[t] = np.dot(Why, hs[t]) + by # unnormalized log probabilities for next chars
    ps[t] = np.exp(ys[t]) / np.sum(np.exp(ys[t])) # probabilities for next chars
    loss += -np.log(ps[t][targets[t],0]) # softmax (cross-entropy loss)
  # backward pass: compute gradients going backwards
  dWxh, dWhh, dWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
  dbh, dby = np.zeros_like(bh), np.zeros_like(by)
  dhnext = np.zeros_like(hs[0])
  for t in reversed(xrange(len(inputs))):
    dy = np.copy(ps[t])
    dy[targets[t]] -= 1 # backprop into y. see http://cs231n.github.io/neural-networks-case-study/#grad if confused here
    dWhy += np.dot(dy, hs[t].T)
    dby += dy
    dh = np.dot(Why.T, dy) + dhnext # backprop into h
    dhraw = (1 - hs[t] * hs[t]) * dh # backprop through tanh nonlinearity
    dbh += dhraw
    dWxh += np.dot(dhraw, xs[t].T)
    dWhh += np.dot(dhraw, hs[t-1].T)
    dhnext = np.dot(Whh.T, dhraw)
  for dparam in [dWxh, dWhh, dWhy, dbh, dby]:
    np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
  return loss, dWxh, dWhh, dWhy, dbh, dby, hs[len(inputs)-1]

```

```
def sample(h, seed_ix, n):
  """
  sample a sequence of integers from the model
  h is memory state, seed_ix is seed letter for first time step
  """
  x = np.zeros((vocab_size, 1))
  x[seed_ix] = 1
  ixes = []
  for t in xrange(n):
    h = np.tanh(np.dot(Wxh, x) + np.dot(Whh, h) + bh)
    y = np.dot(Why, h) + by
    p = np.exp(y) / np.sum(np.exp(y))
    ix = np.random.choice(range(vocab_size), p=p.ravel())
    x = np.zeros((vocab_size, 1))
    x[ix] = 1
    ixes.append(ix)
  return ixes

n, p = 0, 0
mWxh, mWhh, mWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
mbh, mby = np.zeros_like(bh), np.zeros_like(by) # memory variables for Adagrad
smooth_loss = -np.log(1.0/vocab_size)*seq_length # loss at iteration 0
while True:
  # prepare inputs (we're sweeping from left to right in steps seq_length long)
  if p+seq_length+1 >= len(data) or n == 0:
    hprev = np.zeros((hidden_size,1)) # reset RNN memory
    p = 0 # go from start of data
  inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
  targets = [char_to_ix[ch] for ch in data[p+1:p+seq_length+1]]

  # sample from the model now and then
  if n % 100 == 0:
    sample_ix = sample(hprev, inputs[0], 200)
    txt = ''.join(ix_to_char[ix] for ix in sample_ix)
    print '----\n %s \n----' % (txt, )

  # forward seq_length characters through the net and fetch gradient
  loss, dWxh, dWhh, dWhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
  smooth_loss = smooth_loss * 0.999 + loss * 0.001
  if n % 100 == 0: print 'iter %d, loss: %f' % (n, smooth_loss) # print progress

  # perform parameter update with Adagrad
  for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
                                [dWxh, dWhh, dWhy, dbh, dby],
                                [mWxh, mWhh, mWhy, mbh, mby]):
    mem += dparam * dparam
    param += -learning_rate * dparam / np.sqrt(mem + 1e-8) # adagrad update

  p += seq_length # move data pointer
  n += 1 # iteration counter
```

karpathy commented on 27 Jul 2015 Owner

Also here is the gradient check code as well. It's ugly but works:

```
# gradient checking
from random import uniform
def gradCheck(inputs, target, hprev):
  global Wxh, Whh, Why, bh, by
  num_checks, delta = 10, 1e-5
  _, dWxh, dWhh, dWhy, dbh, dby, _ = lossFun(inputs, targets, hprev)
  for param,dparam,name in zip([Wxh, Whh, Why, bh, by], [dWxh, dWhh, dWhy, dbh, dby], ['Wxh', 'Whh', 'Why', 'bh', 'by']):
    s0 = dparam.shape
    s1 = param.shape
    assert s0 == s1, 'Error dims dont match: %s and %s.' % (`s0`, `s1`)
    print name
    for i in xrange(num_checks):
      ri = int(uniform(0,param.size))
      # evaluate cost at [x + delta] and [x - delta]
      old_val = param.flat[ri]
      param.flat[ri] = old_val + delta
      cg0, _, _, _, _, _, _ = lossFun(inputs, targets, hprev)
      param.flat[ri] = old_val - delta
      cg1, _, _, _, _, _, _ = lossFun(inputs, targets, hprev)
      param.flat[ri] = old_val # reset old value for this parameter
```

```
        # fetch both numerical and analytic gradient
        grad_analytic = dparam.flat[ri]
        grad_numerical = (cg0 - cg1) / ( 2 * delta )
        rel_error = abs(grad_analytic - grad_numerical) / abs(grad_numerical + grad_analytic)
        print '%f, %f => %e ' % (grad_numerical, grad_analytic, rel_error)
        # rel_error should be on order of 1e-7 or less
```

denis-bz commented on 1 Aug 2015

Nice. Could you add a few words describing the problem being solved, or links ?
Is there a straw man e.g. naive Bayes, for which RNN is much better ?
Bytheway, the clip line should be

```
  np.clip( dparam, -1, 1, out=dparam )  # clip to mitigate exploding gradients
```

(Any ideas on ways to plot gradients before / after smoothing ?)

voho commented on 12 Aug 2015

Wonderful. For a beginner, could you please add usage description with some example? Would be very grateful!

farizrahman4u commented on 15 Aug 2015

Why is the loss going up sometimes during training?

r03ert0 commented on 16 Aug 2015

Yes! more comments please (it will not count for the number of lines ;D)

suhaspillai commented on 16 Aug 2015

I think it goes up for first 100 iterations but reduces for all other iterations. I think the reason it goes up is because initially some letters might have different output letters.
Like for example:
Winter is harsh in Rochester,USA
Summer is harsh in India
now for one sentence n-> R and for another sentence you have n->I. So, for first few iterations the weights are trying to learn this features, I think they might be capturing some information about weather (Summer and Winter, in this eg). Thus, after few hundred iterations your weights have learned that information and then predicts the correct letter based on some conditional information of the past(like weather in this case), thereby increasing the class score for that letter and -log(score) decreases, thus reducing the loss.

daquang commented on 29 Aug 2015

Does this code implement mini-batch truncated bptt?

ozancaglayan commented on 18 Sep 2015

The original blog post referring to this code is:
http://karpathy.github.io/2015/05/21/rnn-effectiveness/

popwin commented on 9 Nov 2015

Thank you~I learned a lot from your code

GriffinLiang commented on 20 Nov 2015

Thanks for sharing~

kkunte commented on 4 Dec 2015

Thanks for sharing an excellent article and the code.
I am bit confused about the [targets[t],0] array reference in following line:
loss += -np.log(ps[t][targets[t],0]) # softmax (cross-entropy loss)

I tried searching the documentation for numpy but with no luck.

bshillingford commented on 14 Dec 2015

@kkunte Numpy lets you do: `array_of_floats[array_of_indices]` to select out the elements of an array, so that syntax computes `result[i] = array_of_floats[array_of_indices[i]]` for `i=0,...,len(array_of_indices)-1` .(more quickly, conveniently, and without a loop)

ijkilchenko commented on 9 Jan 2016

If you want a word-level language model, insert `data = data.split()` after reading the input file (after line 8 at the time of writing this comment). Leave everything else as is.

jayanthkoushik commented on 10 Jan 2016

What's the purpose of `smooth_loss` here?

to0ms commented on 17 Jan 2016

@bshillingford IMO not the good answer.
@kkunte Targets is a list of integers (so targets[t] is an integer which plays index) and ps[t] a column matrix, so ps[t][targets[t], 0] -> ps[t][targets[t]][0]

More generally with x, a numpy matrix with (2,4) shape, x[1, 3] == x[1][3]

"Unlike lists and tuples, numpy arrays support multidimensional indexing for multidimensional arrays. That means that it is not necessary to separate each dimension’s index into its own set of square brackets."

rajarsheem commented on 31 Jan 2016

While performing word level modelling, isn't it better to use word2vec representation for each word instead of onehot encoding ?

ShuaiW commented on 10 Feb 2016

Thanks for this mini RNN (which I also find easier to read than text).

There is one thing I don't quite understand: what's the intuition of dhnext (defined on line 47) and then adding it to the gradient dh (line 53)? I turned '+ dhnext' (line 53) off and found that without it the model enjoys a faster convergence rate and a lower loss. Here are my experiment results.

Without '+ dhnext' on line 53: iter 10000, loss: 4.696478; iter 40000, loss: 0.763520

With '+ dhnext' on line 53: iter 10000, loss: 5.893804; iter 40000, loss: 1.647147

karpathy commented on 11 Feb 2016 Owner

@ShuaiW the hidden state variable `h` is used twice: one going vertically up to the prediction (`y`), and one going horizontally to the next hidden state `h` at the next time step. During backpropagation these two "branches" of computation both contribute gradients to `h` , and these gradients have to add up. The variable `dhnext` is the gradient contributed by the horizontal branch. It's strange to see you get better performance without it, but my guess is that if you ran it longer the *proper* way would eventually win out. It's computing the correct gradient.

xiaoyu32123 commented on 16 Feb 2016

I think the line 51 should be: dWhy += np.dot(dy, (1/hs[t]).T), also line 53, 56, 57. Am I wrong?

HanuManuOm commented on 10 Mar 2016

When I am running this python code, min_char_rnn.py with a text file called input.txt having content as "Hello World. Best Wishes." Then it is un-ending. Its taking more than 24 hours to run. Iterations and Loss are going on but, never ending. Please help me out.

pmichel31415 commented on 19 Mar 2016

@HanuManuOm as you can see the last part of the code is a `while True:` loop so it is supposed not to end. It's just a toy script, you should check out his char-nn on github for a more complete version. This is just to see how it works. Run it on fancy text, look at the random babbling it produces every second and, when you're bored, just `Ctrl+C` your way out of it

CamJohnson26 commented on 3 Apr 2016

So I can't get results as good as the article even after 1.5 million iterations. What parameters were you using for the Paul Graham quotes? Mine seems to learn structure but can't consistently make actual words

0708andreas commented on 13 Apr 2016

@Mostlyharmless26 In the article, he links to this GitHub repo: https://github.com/karpathy/char-rnn. That code is implemented using Torch and defaults to slightly larger models. You should probably use that if you're trying to replicate his results

laie commented on 19 Apr 2016

@ShuaiW, actually, as like @karpathy's calculation, it's correct to add two terms to calculate exact derivative. In that case you are treating previous hidden state as input like they are not influenced by the network's weights. But I think that exact derivative's harming the first-order optimizer more than your wrong assumption. Nowadays most researchers don't fully trust GD's weight update proposition. So they preprocess gradients by clipping, or using element-wise methods like RMSPROP, ADADELTA, ...

ChiZhangRIT commented on 21 Apr 2016

@Karpathy Thanks very much for providing the gradient check. When I run the gradient checking, I found that all the relative errors are very small except for some of them in Wxh. They are shown as NaN:

```
Wxh
0.000000, 0.000000 => nan
0.000000, 0.000000 => nan
-0.025170, -0.025170 => 1.155768e-08
0.000000, 0.000000 => nan
0.000000, 0.000000 => nan
0.000000, 0.000000 => nan
0.000000, 0.000000 => nan
0.010142, 0.010142 => 3.613506e-09
-0.002687, -0.002687 => 2.578197e-08
0.000000, 0.000000 => nan
```

I tried to change the dtype to np.float64 but it did not go away. Do you have any idea what is going on here?

I appreciate if you could provide help of any kind.

BenMacKenzie commented on 21 May 2016

@ShuaiW @Karpathy does adding in dhnext on line 53 really give you the exact gradient? Wouldn't the full gradient for a particular output include gradients from all outputs the occur later in the letter sequence? It looks like this is an approximation that limits influence of an output to the next letter in sequence.

rongjiecomputer commented on 4 Jun 2016

@Karpathy Thank you so much for the code, it is really helpful for learning!

For those who don't fancy Shakespeare much, Complete Sherlock Holmes in raw text might be more interesting to play with!

alihassan1 commented on 6 Jun 2016 • edited

@Karpathy Thank you so much for this awesome code.

I'm new to Python and I was wondering if you can explain the following line (75)

```
ix = np.random.choice(range(vocab_size), p=p.ravel())
```

Shouldn't we be taking the index of max value of 'p' here instead?

rohitsaluja22 commented on 10 Jun 2016 • edited

Hi Karpathy, thanks a lot for sharing this code and article on this. It helped me a lot growing my understanding about RNN.

@allhassan1, line 75 is doing the same thing you said, i.e. it is giving maximum value index of p. I do not know exactly how, but if i check on python with some random vocab_size and p, its giving the maximum value index of item in p.
range(vocab_size) will give a normal python list - [0 1 2 3 ..... (vocab_size-1)]
p.ravel() just readjust m by n matrix to mn by 1 array.

Check these references and let me know if you figure it out why the line 75 gives max value index in p:-
http://docs.scipy.org/doc/numpy-1.10.1/reference/generated/numpy.ravel.html
http://docs.scipy.org/doc/numpy-dev/reference/generated/numpy.random.choice.html

alihassan1 commented on 12 Jun 2016

@rohitsaluja22 Thanks a lot for your response above. I still think it doesn't give maximum value index of p because by definition the function np.random.choice generates a non-uniform random sample when called with p. I wonder what would be the equivalent function in Matlab?

sunshineatnoon commented on 12 Jun 2016

@jayanthkoushik, did you figure out the purpose of smooth_loss? I have the same question.

DvHuang commented on 22 Jun 2016 • edited

@alihassan1 @rohitsaluja22
the code in line (75),it doesn't return the index of max value of 'p'.
As the code down here,when you try some times it return different value
p=np.zeros((4,1))
p[:,0]=[0.3,0.2,0.4,0.1]
print p,p.ravel(),p.shape,p.ravel().shape
ix = np.random.choice(4, p=p.ravel())
print ix,"ix"

the index of max value of 'p' : p.argmax()

profPlum commented on 3 Jul 2016 • edited

@karpathy
I'm trying to understand the math behind this... On line 41 you say that ys[] contains the "unnormalized log probabilities for next chars" and then you use the exp() function to get the real probabilities. At what point did those become "log probabilities"? Is it an effect of the tanh() activation function? Any insight would be appreciated.

EDIT: Ok I figured out that you're computing the softmax by doing that, but now I'm curious why you use a different activation function in the hidden layers than in the output layer?

liuzhi136 commented on 6 Jul 2016

Thanks very much for your code. It is the code that I can understand the RNN more deeply. I wander that what dose the code of "# prepare inputs (we're sweeping from left to right in steps seq_length long)" mean. I have read your blog http://karpathy.github.io/2015/05/21/rnn-effectiveness/. and test the very simple example "hello". I would be very appreiciate if I could receive your anwser.

eliben commented on 23 Jul 2016

@alihassan1 -- correct, this doesn't use argmax to select the one char with highest probability, but rather uses *sampling* to select from all chars weighted by their probabilities (so the maximal prob char still has the highest chance of being selected, but now it's a probability distribution). Read the documentation of numpy's `random.choice` for the full insight.

IMHO in @karpathy's https://github.com/karpathy/char-rnn/ repository this is configurable with the `sample` option which you set to 1 if you want sampling and 0 if you want argmax. In case of sampling you can also use `temperature` to scale down all probabilities a bit.

I hope this makes sense :)

mohamaddanesh commented on 26 Jul 2016 • edited

Hi, could you explain or give a link describing about the usage of `dhraw` and what's it for in line 54 and 56? I got a little confused about it.
thanks

rincerwind commented on 31 Jul 2016 • edited

Hi, thanks for the code.

@karpathy, Is it correct to say that this network is Fully Recurrent and that the relationship between the neurons in one layer is a Soft-Winner-Takes-All? It seems like that from the hidden-weight matrix.

Thanks

uvarovann commented on 7 Aug 2016

something error...
inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
targets = [char_to_ix[ch] for ch in data[p+1:p+seq_length]]

length inputs one less targets, for example:
inputs = [5, 6, 2, 1, 0, 3, 4]
targets = [6, 2, 1, 0, 3, 4]

12digimon commented on 15 Aug 2016

Hi Is it possible to speak with you

12digimon commented on 15 Aug 2016

Hi Is it possible to speak with you

guotong1988 commented on 17 Aug 2016 • edited

@uvarovann same question. Have you fixed it ?

rukshn commented on 19 Aug 2016

@profPlum it's better to use a softmax function as the output's activation function over tanh or sigmoid function. Softmax function also doesn't have the problem of values going to extreme levels, and the vanishing gradient problem. Usually what I heard was that even for the hidden nodes it's better to use the ReLU function over the sigmoid of tanh functions because they also then doesn't suffer the vanishing gradient problem, however it's far more difficult to train the network oppsed to tanh or sigmoid.

rukshn commented on 20 Aug 2016

@uvarovann usually this happens at the end of the corpus because there is no character after the last character of the corpus meaning that the target is always one short from the input, what i did was append a space to the end of it

cyrilfurtado commented on 1 Sep 2016

When does the training complete?
Also after training how does it output any learnt data like 'code' or 'Shakespeare'?

eduOS commented on 5 Sep 2016 • edited

@sunshineatnoon @jayanthkoushik I thought the smooth_loss has nothing to do with the algorithm, since it is only used as a friendly(smooth) dashboard to check the decreasing value of the true loss.

pavelkomarov commented on 17 Sep 2016 • edited

I think I've managed to reimplement the above in a slightly more sensible way. I couldn't understand it very well before this exercise. Maybe this will help some others, give you a different jumping-off point.

```
#implemented as I read Andrej Karpathy's post on RNNs.
import numpy as np
import matplotlib.pyplot as plt

class RNN(object):

    def __init__(self, insize, outsize, hidsize, learning_rate):
        self.insize = insize

        self.h = np.zeros((hidsize , 1))#a [h x 1] hidden state stored from last batch of inputs

        #parameters
        self.W_hh = np.random.randn(hidsize, hidsize)*0.01#[h x h]
        self.W_xh = np.random.randn(hidsize, insize)*0.01#[h x x]
        self.W_hy = np.random.randn(outsize, hidsize)*0.01#[y x h]
        self.b_h = np.zeros((hidsize, 1))#biases
        self.b_y = np.zeros((outsize, 1))

        #the Adagrad gradient update relies upon having a memory of the sum of squares of dparams
        self.adaW_hh = np.zeros((hidsize, hidsize))
        self.adaW_xh = np.zeros((hidsize, insize))
        self.adaW_hy = np.zeros((outsize, hidsize))
        self.adab_h = np.zeros((hidsize, 1))
        self.adab_y = np.zeros((outsize, 1))

        self.learning_rate = learning_rate

    #give the RNN a sequence of inputs and outputs (seq_length long), and use
    #them to adjust the internal state
    def train(self, x, y):
        #=====initialize=====
        xhat = {}#holds 1-of-k representations of x
        yhat = {}#holds 1-of-k representations of predicted y (unnormalized log probs)
        p = {}#the normalized probabilities of each output through time
        h = {}#holds state vectors through time
        h[-1] = np.copy(self.h)#we will need to access the previous state to calculate the current state

        dW_xh = np.zeros_like(self.W_xh)
        dW_hh = np.zeros_like(self.W_hh)
        dW_hy = np.zeros_like(self.W_hy)
        db_h = np.zeros_like(self.b_h)
        db_y = np.zeros_like(self.b_y)
        dh_next = np.zeros_like(self.h)

        #=====forward pass=====
        loss = 0
        for t in range(len(x)):
            xhat[t] = np.zeros((self.insize, 1))
            xhat[t][x[t]] = 1#xhat[t] = 1-of-k representation of x[t]

            h[t] = np.tanh(np.dot(self.W_xh, xhat[t]) + np.dot(self.W_hh, h[t-1]) + self.b_h)#find new hidden state
            yhat[t] = np.dot(self.W_hy, h[t]) + self.b_y#find unnormalized log probabilities for next chars

            p[t] = np.exp(yhat[t]) / np.sum(np.exp(yhat[t]))#find probabilities for next chars

            loss += -np.log(p[t][y[t],0])#softmax (cross-entropy loss)

        #=====backward pass: compute gradients going backwards=====
        for t in reversed(range(len(x))):
            #backprop into y. see http://cs231n.github.io/neural-networks-case-study/#grad if confused here
            dy = np.copy(p[t])
            dy[y[t]] -= 1

            #find updates for y
            dW_hy += np.dot(dy, h[t].T)
            db_y += dy

            #backprop into h and through tanh nonlinearity
            dh = np.dot(self.W_hy.T, dy) + dh_next
            dh_raw = (1 - h[t]**2) * dh

            #find updates for h
            dW_xh += np.dot(dh_raw, xhat[t].T)
            dW_hh += np.dot(dh_raw, h[t-1].T)
            db_h += dh_raw

            #save dh_next for subsequent iteration
            dh_next = np.dot(self.W_hh.T, dh_raw)

        for dparam in [dW_xh, dW_hh, dW_hy, db_h, db_y]:
            np.clip(dparam, -5, 5, out=dparam)#clip to mitigate exploding gradients

        #update RNN parameters according to Adagrad
        for param, dparam, adaparam in zip([self.W_hh, self.W_xh, self.W_hy, self.b_h, self.b_y], \
                                [dW_hh, dW_xh, dW_hy, db_h, db_y], \
                                [self.adaW_hh, self.adaW_xh, self.adaW_hy, self.adab_h, self.adab_y]):
            adaparam += dparam*dparam
            param += -self.learning_rate*dparam/np.sqrt(adaparam+1e-8)

        self.h = h[len(x)-1]

        return loss

    #let the RNN generate text
    def sample(self, seed, n):
        ndxs = []
        h = self.h

        xhat = np.zeros((self.insize, 1))
        xhat[seed] = 1#transform to 1-of-k

        for t in range(n):
            h = np.tanh(np.dot(self.W_xh, xhat) + np.dot(self.W_hh, h) + self.b_h)#update the state
            y = np.dot(self.W_hy, h) + self.b_y
            p = np.exp(y) / np.sum(np.exp(y))
            ndx = np.random.choice(range(self.insize), p=p.ravel())

            xhat = np.zeros((self.insize, 1))
            xhat[ndx] = 1

            ndxs.append(ndx)

        return ndxs


def test():
    #open a text file
    data = open('shakespeare.txt', 'r').read() # should be simple plain text file
    chars = list(set(data))
    data_size, vocab_size = len(data), len(chars)
    print 'data has %d characters, %d unique.' % (data_size, vocab_size)

    #make some dictionaries for encoding and decoding from 1-of-k
    char_to_ix = { ch:i for i,ch in enumerate(chars) }
    ix_to_char = { i:ch for i,ch in enumerate(chars) }

    #insize and outsize are len(chars). hidsize is 100. seq_length is 25. learning_rate is 0.1.
    rnn = RNN(len(chars), len(chars), 100, 0.1)

    #iterate over batches of input and target output
    seq_length = 25
    losses = []
    smooth_loss = -np.log(1.0/len(chars))*seq_length#loss at iteration 0
    losses.append(smooth_loss)

    for i in range(len(data)/seq_length):
        x = [char_to_ix[c] for c in data[i*seq_length:(i+1)*seq_length]]#inputs to the RNN
        y = [char_to_ix[c] for c in data[i*seq_length+1:(i+1)*seq_length+1]]#the targets it should be outputting
```

```
            if i%1000==0:
                sample_ix = rnn.sample(x[0], 200)
                txt = ''.join([ix_to_char[n] for n in sample_ix])
                print txt

            loss = rnn.train(x, y)
            smooth_loss = smooth_loss*0.999 + loss*0.001

            if i%1000==0:
                print 'iteration %d, smooth_loss = %f' % (i, smooth_loss)
                losses.append(smooth_loss)

        plt.plot(range(len(losses)), losses, 'b', label='smooth loss')
        plt.xlabel('time in thousands of iterations')
        plt.ylabel('loss')
        plt.legend()
        plt.show()

    if __name__ == "__main__":
        test()
```

pavelkomarov commented on 22 Sep 2016 • edited

@karpathy How can we extend this to multiple layers? It's irritating to me that all the implementations I can easily Google use libraries like tensorflow. I want to know how to do this at a rawer level.

pavelkomarov commented on 22 Sep 2016 • edited

@karpathy I also would like a more granular explanation of how to backprop through matrix multiplications like this. These are great http://cs231n.github.io/optimization-2/, but it is unclear how that scales up to more dimensions.

```
#                  [b_h]                                              [b_y]
#                    v                                                  v
#   x -> [W_xh] -> [sum] -> h_raw -> [nonlinearity] -> h -> [W_hy] -> [sum] -> y -> [exp(y[k])/sum(exp(y))] -> p
#                    ^                                 |
#                    '----h_next------[W_hh]-----------'
#
```

I can follow the notes and understand how to get from p to dy, and I can see your expressions for propagating through the rest of this, but I do not understand how they are derived analytically. If I want to understand what gradient I should be passing from one layer of a network to the previous one in backpropagation, I need to be able to get through all of this.

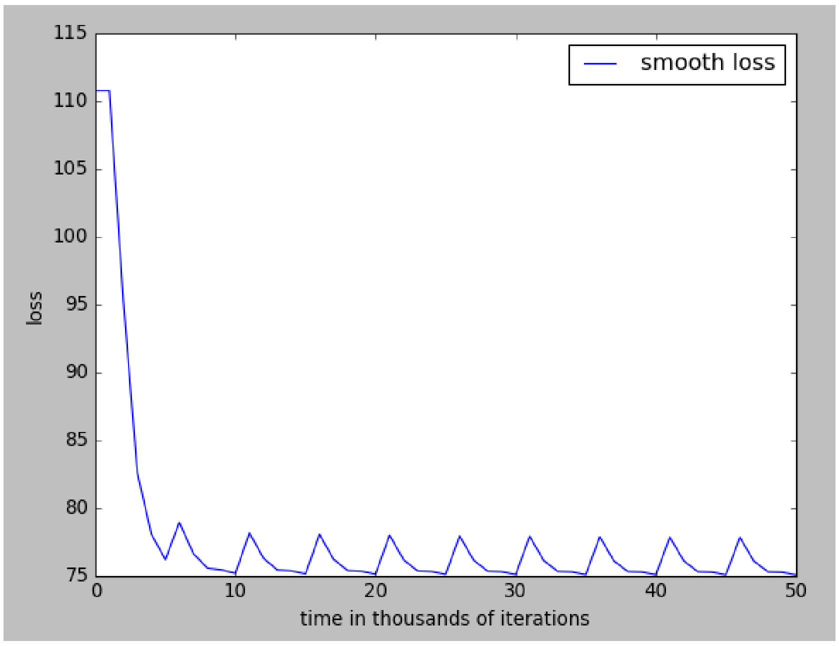
pavelkomarov commented on 23 Sep 2016 • edited

I managed to find something that sort of works, but I am still having issues.

If I calculate dy for the output layer as before, let dx = np.dot(self.W_xh.T, dh_raw) in backprop steps, and use dx as dy for the next layers, I see my loss function decrease. But it only does so to some point, and I know that my 3-layer RNN should have more characterizing power than this.

I implemented a smooth_error like smooth_loss and ran it over the first 50th of my shakespeare.txt training set 10 times. I should see that the network is getting more and more overfit to these inputs, but the error rate remains at about 0.77 through the 10 iterations. Why should it get stuck? I am using a small update and Adagrad.

Here is a plot of the loss during that process:



Here is the complete but not-cleaned-up code in case you want to run or comb through it:

```
#An attempt at a batched RNNs
#
#I don't think this is an LSTM. What is the difference, exactly? I want to
#know the more complicated functional forms, how to backprop them, and what
#the advantage is.
import numpy as np
import matplotlib.pyplot as plt

class RNNlayer(object):

    def __init__(self, x_size, h_size, y_size, learning_rate):
        self.h_size = h_size
        self.learning_rate = learning_rate#ugh, nightmares

        #inputs and internal states for each layer, used during backpropagation
        self.x = {}
        self.h = {}
        self.h_last = np.zeros((h_size, 1))

        #x is the input. h is the internal hidden stuff. y is the output.
        self.W_xh = np.random.randn(h_size, x_size)*0.01#x -> h
        self.W_hh = np.random.randn(h_size, h_size)*0.01#h -> h
        self.W_hy = np.random.randn(y_size, h_size)*0.01#h -> y
        self.b_h = np.zeros((h_size, 1))#biases
        self.b_y = np.zeros((y_size, 1))

        #the Adagrad gradient update relies upon having a memory of the sum of squares of dparams
        self.adaW_xh = np.zeros((h_size, x_size))#start sums at 0
        self.adaW_hh = np.zeros((h_size, h_size))
        self.adaW_hy = np.zeros((y_size, h_size))
        self.adab_h = np.zeros((h_size, 1))
        self.adab_y = np.zeros((y_size, 1))

    #given an input, step the internal state and return the output of the network
    #Because the whole network is together in one object, I can make it easy and just
    #take a list of input ints, transform them to 1-of-k once, and prop everywhere.
    #
    #   Here is a diagram of what's happening. Useful to understand backprop too.
    #
    #                 [b_h]                                              [b_y]
    #                   v                                                  v
    #   x -> [W_xh] -> [sum] -> h_raw -> [nonlinearity] -> h -> [W_hy] -> [sum] -> y ... -> [e] -> p
    #                   ^                                 |
    #                   '----h_next------[W_hh]-----------'
    #
    def step(self, x):
        #load the last state from the last batch in to the beginning of h
        #it is necessary to save it outside of h because h is used in backprop
        self.h[-1] = self.h_last
        self.x = x

        y = {}
        p = {}#p[t] = the probabilities of next chars given chars passed in at times <=t
        for t in range(len(self.x)):#for each moment in time

            #self.h[t] = np.maximum(0, np.dot(self.W_xh, self.xhat[t]) + \
            #   np.dot(self.W_hh, self.h[t-1]) + self.b_h)#ReLU

            #find new hidden state in this layer at this time
            self.h[t] = np.tanh(np.dot(self.W_xh, self.x[t]) + \
                np.dot(self.W_hh, self.h[t-1]) + self.b_h)#tanh

            #find unnormalized log probabilities for next chars
            y[t] = np.dot(self.W_hy, self.h[t]) + self.b_y#output from this layer is input to the next
            p[t] = np.exp(y[t]) / np.sum(np.exp(y[t]))#find probabilities for next chars

        #save the last state from this batch for next batch
        self.h_last = self.h[len(x)-1]

        return y, p

    #given the RNN a sequence of correct outputs (seq_length long), use
    #them and the internal state to adjust weights
    def backprop(self, dy):

        #we will need some place to store gradients
        dW_xh = np.zeros_like(self.W_xh)
        dW_hh = np.zeros_like(self.W_hh)
        dW_hy = np.zeros_like(self.W_hy)
        db_h = np.zeros_like(self.b_h)
        db_y = np.zeros_like(self.b_y)

        dh_next = np.zeros((self.h_size, 1))#I think this is the right dimension
        dx = {}

        for t in reversed(range(len(dy))):
            #find updates for y stuff
            dW_hy += np.dot(dy[t], self.h[t].T)
            db_y += dy[t]

            #backprop into h and through nonlinearity
            dh = np.dot(self.W_hy.T, dy[t]) + dh_next
            dh_raw = (1 - self.h[t]**2)*dh#tanh
            #dh_raw = self.h[t][self.h[t] <= 0] = 0#ReLU

            #find updates for h stuff
            dW_xh += np.dot(dh_raw, self.x[t].T)
            dW_hh += np.dot(dh_raw, self.h[t-1].T)
            db_h += dh_raw

            #save dh_next for subsequent iteration
            dh_next = np.dot(self.W_hh.T, dh_raw)

            #save the error to propagate to the next layer. Am I doing this correctly?
            dx[t] = np.dot(self.W_xh.T, dh_raw)

        #clip to mitigate exploding gradients
        for dparam in [dW_xh, dW_hh, dW_hy, db_h, db_y]:
            dparam = np.clip(dparam, -5, 5)
        for t in range(len(dx)):
            dx[t] = np.clip(dx[t], -5, 5)

        #update RNN parameters according to Adagrad
        #yes, it calls by reference, so the actual things do get updated
        for param, dparam, adaparam in zip([self.W_hh, self.W_xh, self.W_hy, self.b_h, self.b_y], \
                    [dW_hh, dW_xh, dW_hy, db_h, db_y], \
                    [self.adaW_hh, self.adaW_xh, self.adaW_hy, self.adab_h, self.adab_y]):
            adaparam += dparam*dparam
            param += -self.learning_rate*dparam/np.sqrt(adaparam+1e-8)

        return dx

def test():
    #open a text file
    data = open('shakespeare.txt', 'r').read() # should be simple plain text file
    chars = list(set(data))
    data_size, vocab_size = len(data), len(chars)
    print 'data has %d characters, %d unique.' % (data_size, vocab_size)

    #make some dictionaries for encoding and decoding from 1-of-k
    char_to_ix = { ch:i for i,ch in enumerate(chars) }
    ix_to_char = { i:ch for i,ch in enumerate(chars) }

    #num_hid_layers = 3, insize and outsize are len(chars). hidsize is 512 for all layers. learning_rate is 0.1.
    rnn1 = RNNlayer(len(chars), 50, 50, 0.001)
    rnn2 = RNNlayer(50, 50, 50, 0.001)
    rnn3 = RNNlayer(50, 50, len(chars), 0.001)

    #iterate over batches of input and target output
    seq_length = 25
    losses = []
    smooth_loss = -np.log(1.0/len(chars))*seq_length#loss at iteration 0
    losses.append(smooth_loss)
    smooth_error = seq_length

    for j in range(10):
        print "============= j = ",j," ================="
        for i in range(len(data)/(seq_length*50)):
            inputs = [char_to_ix[c] for c in data[i*seq_length:(i+1)*seq_length]]#inputs to the RNN
            targets = [char_to_ix[c] for c in data[i*seq_length+1:(i+1)*seq_length+1]]#the targets it should be outputting

            if i%1000==0:
                sample_ix = sample([rnn1, rnn2, rnn3], inputs[0], 200, len(chars))
                txt = ''.join([ix_to_char[n] for n in sample_ix])
                print txt
                losses.append(smooth_loss)

            #forward pass
            x = oneofk(inputs, len(chars))
            y1, p1 = rnn1.step(x)
            y2, p2 = rnn2.step(y1)
            y3, p3 = rnn3.step(y2)

            #calculate loss and error rate
            loss = 0
            error = 0
            for t in range(len(targets)):
                loss += -np.log(p3[t][targets[t],0])
                if np.argmax(p3[t]) != targets[t]:
                    error += 1
            smooth_loss = smooth_loss*0.999 + loss*0.001
            smooth_error = smooth_error*0.999 + error*0.001

            if i%10==0:
                print i,"\tsmooth loss =",smooth_loss,"\tsmooth error rate =",float(smooth_error)/len(targets)

            #backward pass
            dy = logprobs(p3, targets)
            dx3 = rnn3.backprop(dy)
            dx2 = rnn2.backprop(dx3)
            dx1 = rnn1.backprop(dx2)

    plt.plot(range(len(losses)), losses, 'b', label='smooth loss')
    plt.xlabel('time in thousands of iterations')
    plt.ylabel('loss')
    plt.legend()
    plt.show()


#let the RNN generate text
def sample(rnns, seed, n, k):

    ndxs = []
    ndx = seed

    for t in range(n):
        x = oneofk([ndx], k)
        for i in range(len(rnns)):
            x, p = rnns[i].step(x)

        ndx = np.random.choice(range(len(p[0])), p=p[0].ravel())
        ndxs.append(ndx)

    return ndxs

#I have these out here because it's not really the RNN's concern how you transform
#things to a form it can understand

#get the initial dy to pass back through the first layer
def logprobs(p, targets):
    dy = {}
    for t in range(len(targets)):
        #see http://cs231n.github.io/neural-networks-case-study/#grad if confused here
        dy[t] = np.copy(p[t])
        dy[t][targets[t]] -= 1
    return dy

#encode inputs in 1-of-k so they match inputs between layers
def oneofk(inputs, k):
    x = {}
    for t in range(len(inputs)):
        x[t] = np.zeros((k, 1))#initialize x input to 1st hidden layer
        x[t][inputs[t]] = 1#it's encoded in 1-of-k representation
    return x

if __name__ == "__main__":
    test()
```

mfagerlund commented on 25 Sep 2016

@pavelkomarov, for an analytical treatment of this very code, have a look here: http://www.existor.com/en/ml-rnn.html

caverac commented on 10 Oct 2016

@karpathy thanks a lot for this posting! I tried to run it with your hello example ... and this is what I get

```
Traceback (most recent call last):
File "min-char-rnn.py", line 100, in
loss, dWxh, dWhh, dWhy, dbh, dby, hprev = lossFun(inputs, targets, hprev)
File "min-char-rnn.py", line 43, in lossFun
loss += -np.log(ps[t][targets[t],0]) # softmax (cross-entropy loss)
IndexError: list index out of range
```

Any ideas?
Thanks!

pavelkomarov commented on 13 Oct 2016 • edited

@mfagerlund Damn, that's some crazy calculus, but thanks. Karpathy should have referenced something like this in his post for those of us who want to do it ourselves. How to backpropagate through to get dx (the thing you would want to pass back to the next layer in the network) is still unclear, so I have no idea whether I did it correctly. I also still have no idea why my loss stops decreasing as it does.

Also, is my diagram/flowchart correct?

delijati commented on 1 Nov 2016 • edited

Explanation of the rnn code: https://youtu.be/cO0a0QYmFm8?list=PLlJy-eBtNFt6EuMxFYRiNRS07MCWN5UIA&t=836

georgeblck commented on 29 Nov 2016

@pavelkomarov and all others
If you want to check the math more thoroughly, you can do so with the recently published Deep Learning Book by Goodfellow/Bengio/Courville. Check Chapter 10.1 & 10.2 or more specifically look on page 385-386 for the Backprop equations. The architecture used in that example is exactly the same as the one used here.

It does take some time to connect the notation in the book with the notation in the code, but it is worth it.

100ZeroGravity commented on 18 Dec 2016 • edited

Just discovered github and this is my favorite gist for the moment wow. Helped me with my post thanks.

taosiqin1991 commented on 18 Dec 2016

wonderful

Zoson commented on 24 Dec 2016

The loss divided by the length of inputs before backprop will be better.

somah1411 commented on 12 Jan

how can i use this code for translation where shall i put the input and target langauge

ppaquette commented on 17 Jan

@caverac The variable `seq_length` must be smaller than the size of the data in your input.txt, otherwise there are not enough targets to calculate the loss function. Decreasing `seq_length` or adding more text in your input.txt should fix the issue.

georgeblck commented on 7 Feb

For anyone interested, I rewrote the entire code in R.
You can find it here.

bhomass commented on 9 Feb

is seq_length the same as the the number of hidden nodes? I think it is, just want to be sure.

georgeblck commented on 9 Feb

@bhomass `seq_length` is not the number of hidden nodes. `seq_length` determines for how many time steps you want to unravel your RNN. In this case one time step is a letter, so you train your network based on the 25 previous time steps/letters.

bhomass commented on 10 Feb • edited

@georgeblck That agrees with my understanding what seq_length is. I see now there is another variable for hidden_size of 100. I understand the difference now. Thanks!

inexxt commented on 22 Feb • edited

@ChiZhangRIT The reason is because if two of the derivatives are exactly zero, you're dividing by zero - it's a special case not handled by the code.
@karpathy Worse situation is when exactly one of them is equal to zero - then dividing by the abs(sum) yields just that value, which can be correctly greater than the treshold.

shaktisd commented 22 days ago

@karpathy can you share similar implementation using Keras ? It is much easier to understand the code using Keras.

hkxIron commented 20 days ago

According to 《Supervised Sequence Labelling with Recurrent Neural Networks》,Alex Graves,2012,we have no gradient for dh and dhnext,can you explain it ?

coolBoyGym commented 15 days ago

A nice material. Thanks a lot !

wilderfield commented 13 days ago

I want to apply this concept to a different problem without using one-hot encoding. My input vector has 2^48 possibilities. I am afraid to one-hot encode that. If I strip out the one-hot encoding, can I use a different cost function such as 1/2 L2Norm^2 ?? I feel like I can't use softmax since I am not expecting the log probabilities to add up to 1... My input could be 0,0,1,1... and my output could be 1,0,1,0

wilderfield commented 12 days ago

@karapathy Why the need for dictionaries to hold various y[t] vectors when you could just create a matrix Y, whose columns are time steps, and rows are dimensions?