jupyter lesson6-rnn (autosaved)

Logout

Not Connected | Not Trusted | Python 2

File Edit View Insert Cell Kernel Widgets Help

Code

In [1]:

```
%reload_ext autoreload
%autoreload 2
%matplotlib inline
â€‹
from fastai.io import *
from fastai.conv_learner import *
â€‹
from fastai.column_data import *
```

Setup

We're going to download the collected works of Nietzsche to use as our data for this class.

In [2]:

```
PATH='data/nietzsche/'
```

In [3]:

```
get_data("https://s3.amazonaws.com/text-datasets/nietzsche.txt", f'{PATH}nietzsche.txt')
text = open(f'{PATH}nietzsche.txt').read()
print('corpus length:', len(text))
```

No of characters in the whole text

```
corpus length: 600893
```

show the first 400 characters of the text

In [4]:

```
text[:400]
```

Out[4]:

```
'PREFACE\n\n\nSUPPOSING that Truth is a woman--what then? Is there not gro
und\nfor suspecting that all philosophers, in so far as they have been\ndo
gmatists, have failed to understand women--that the terrible\nseriousness
and clumsy importunity with which they have usually paid\ntheir addresses
to Truth, have been unskilled and unseemly methods for\nwinning a woman? C
ertainly she has never allowed herself '
```

In [5]:

```
chars = sorted(list(set(text)))
vocab_size = len(chars)+1
print('total chars:', vocab_size)
```

set creates a list of unique characters, in this case 85 unique characters

```
total chars: 85
```

Sometimes it's useful to have a zero value in the dataset, e.g. for padding

In [6]:

```
chars.insert(0, "\0")
```

insert \ padding

In [7]:

```
''.join(chars[1:-6])
```

Out[7]:
```
'\n !"\'(),-.0123456789:;=?ABCDEFGHIJKLMNOPQRSTUVWXYZ[]_abcdefghijklmnopqr
stuvwxy'
```

Map from chars to indices and back again

map every character to a unique ID and every unique ID to every character

In [8]:
```
char_indices = dict((c, i) for i, c in enumerate(chars))
indices_char = dict((i, c) for i, c in enumerate(chars))
```

idx will be the data we use from now on - it simply converts all the characters to their index (based on the mapping above)

converts each character into a corresponding index

In [9]:
```
idx = [char_indices[c] for c in text]
```

collected works of Nietzsche

In [10]:
```
idx[:10]
```

Out[10]:
```
[40, 42, 29, 30, 25, 27, 29, 1, 1, 1]
```

P R E F A C E n n n

In [11]:
```
''.join(indices_char[i] for i in idx[:70])
```

Out[11]:
```
'PREFACE\n\n\nSUPPOSING that Truth is a woman--what then? Is there not gro
'
```

Three char model

Create inputs

Create a list of every 4th character, starting at the 0th, 1st, 2nd, then 3rd characters

0th character

1st character

2nd character

3rd character

skip every 3 characters

```
cs=3
c1_dat = [idx[i]   for i in range(0, len(idx)-1-cs, cs)]
c2_dat = [idx[i+1] for i in range(0, len(idx)-1-cs, cs)]
c3_dat = [idx[i+2] for i in range(0, len(idx)-1-cs, cs)]
c4_dat = [idx[i+3] for i in range(0, len(idx)-1-cs, cs)]
```

Our inputs

In [13]:
```
x1 = np.stack(c1_dat[:-2])
x2 = np.stack(c2_dat[:-2])
x3 = np.stack(c3_dat[:-2])
```

0th, 1st and 2nd character as inputs

Our output

In [14]:
```
y = np.stack(c4_dat[:-2])
```

predict the 3rd character as the output

The first 4 inputs and outputs

In [14]:
```
x1[:4], x2[:4], x3[:4]
```
Out[14]:
```
(array([40, 30, 29,  1]), array([42, 25,  1, 43]), array([29, 27,  1, 45])
)
```
0th P 1st R 2nd E

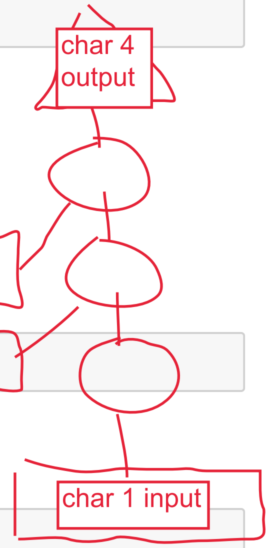
In [15]:
```
y[:4]
```
Out[15]:
```
array([30, 29,  1, 40])
```
F

used to predict the 3rd character

In [16]:
```
x1.shape, y.shape
```
Out[16]:
```
((200295,), (200295,))
```

Create and train model

3 layer network



Pick a size for our hidden state

In [15]:
```
n_hidden = 256
```

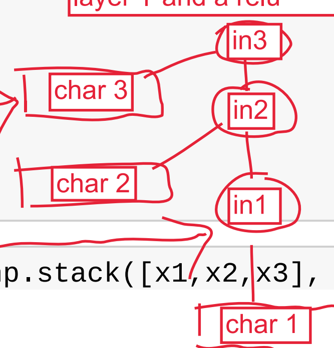
The number of latent factors to create (i.e. the size of the embedding matrix)

In [16]:
```
n_fac = 42
```
1/2 the number of characters

In [30]:
```
class Char3Model(nn.Module):
    def __init__(self, vocab_size, n_fac):
        super().__init__()
        self.e = nn.Embedding(vocab_size, n_fac)
â€‹
        # The 'green arrow' from our diagram - the layer operation from
 hidden
        self.l_in = nn.Linear(n_fac, n_hidden)
â€‹
        # The 'orange arrow' from our diagram - the layer operation fro
to hidden
        self.l_hidden = nn.Linear(n_hidden, n_hidden)

        # The 'blue arrow' from our diagram - the layer operation from
 output
        self.l_out = nn.Linear(n_hidden, vocab_size)

    def forward(self, c1, c2, c3):
        in1 = F.relu(self.l_in(self.e(c1)))
        in2 = F.relu(self.l_in(self.e(c2)))
        in3 = F.relu(self.l_in(self.e(c3)))

        h = V(torch.zeros(in1.size()).cuda())
        h = F.tanh(self.l_hidden(h+in1))
        h = F.tanh(self.l_hidden(h+in2))
        h = F.tanh(self.l_hidden(h+in3))

        return F.log_softmax(self.l_out(h))
```
square matrix

when calling forward we will pass 3 characters c1, c2, and c3

each character will pass through an embedding 'e', linear layer 'l' and a relu



In [20]:
```
md = ColumnarModelData.from_arrays('.', [-1], np.stack([x1,x2,x3], axis=
```

```
, bs=512)
```

hyperbolic tanh

In [21]:
```
m = Char3Model(vocab_size, n_fac).cuda()
```

In [22]:
```
it = iter(md.trn_dl)
*xs,yt = next(it)
t = m(*V(xs))
```

inputs of 3 and size 512

Variablized versions of the inputs

In [191]:
```
opt = optim.Adam(m.parameters(), 1e-2)
```

In [192]:
```
fit(m, md, 1, opt, F.nll_loss)
```

```
A Jupyter Widget

[ 0.         2.09627   6.52849]
```

In [193]:
```
set_lrs(opt, 0.001)
```

In [194]:
```
fit(m, md, 1, opt, F.nll_loss)
```

```
A Jupyter Widget

[ 0.         1.84525   6.52312]
```

Test model

covert to tensor T

In [195]:
```
def get_next(inp):
    idxs = T(np.array([char_indices[c] for c in inp]))
    p = m(*V(idxs))
    i = np.argmax(to_np(p))
    return chars[i]
```

In [196]:
```
get_next('y. ')
```

3 char model hence can pass in 3 things

Out[196]: 'T'

In [197]:
```
get_next('ppl')
```

Out[197]: 'e'

In [198]:
```
get_next(' th')
```

Out[198]: 'e'

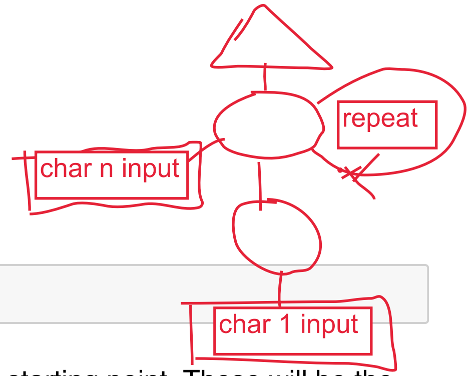
In [199]:
```
get_next('and')
```

Out[199]: ' '

Our first RNN!

Create inputs

8 layer network = 8 time steps in RNN



This is the size of our unrolled RNN.

we are now going to predict every 8th character

In [19]:
```
cs=8
```

For each of 0 through 7, create a list of every 8th character with that starting point. These will be the 8 inputs to out model.

In [20]:
```
c_in_dat = [[idx[i+j] for i in range(cs)] for j in range(len(idx)-cs-1)]
```

Then create a list of the next character in each of these series. This will be the labels for our model.

In [21]:
```
c_out_dat = [idx[j+cs] for j in range(len(idx)-cs-1)]
```

In [22]:
```
xs = np.stack(c_in_dat, axis=0)
```

In [23]:
```
xs.shape
```

corpus length

Out[23]:
```
(600884, 8)
```

char size

In [24]:
```
y = np.stack(c_out_dat)
```

So each column below is one series of 8 characters from the text.

input

In [25]:
```
xs[:cs,:cs]
```

Out[25]:
```
array([[40, 42, 29, 30, 25, 27, 29,  1],
       [42, 29, 30, 25, 27, 29,  1,  1],
       [29, 30, 25, 27, 29,  1,  1,  1],
       [30, 25, 27, 29,  1,  1,  1, 43],
       [25, 27, 29,  1,  1,  1, 43, 45],
       [27, 29,  1,  1,  1, 43, 45, 40],
       [29,  1,  1,  1, 43, 45, 40, 40],
       [ 1,  1,  1, 43, 45, 40, 40, 39]])
```

characters 0 to 7

characters 1 to 8 etc

output

...and this is the next character after each sequence.

In [26]:
```
y[:cs]
```

Out[26]:
```
array([ 1,  1, 43, 45, 40, 40, 39, 43])
```

the y prediction is the 8th character

Create and train model

In [27]:
```
val_idx = get_cv_idxs(len(idx)-cs-1)
```

In [28]:

```python
md = ColumnarModelData.from_arrays('.', val_idx, xs, y, bs=512)
```

In [158]:

```python
class CharLoopModel(nn.Module):
    def __init__(self, vocab_size, n_fac):
        super().__init__()
        self.e = nn.Embedding(vocab_size, n_fac)
        self.l_in = nn.Linear(n_fac, n_hidden)
        self.l_hidden = nn.Linear(n_hidden, n_hidden)
        self.l_out = nn.Linear(n_hidden, vocab_size)

    def forward(self, *cs):
        bs = cs[0].size(0)
        h = V(torch.zeros(bs, n_hidden).cuda())
        for c in cs:
            inp = F.relu(self.l_in(self.e(c)))
            h = F.tanh(self.l_hidden(h+inp))

        return F.log_softmax(self.l_out(h))
```

same as 3 char model above except now in a loop or RNN

In [159]:

```python
m = CharLoopModel(vocab_size, n_fac).cuda()
opt = optim.Adam(m.parameters(), 1e-2)
```

In [160]:

```python
fit(m, md, 1, opt, F.nll_loss)
```

A Jupyter Widget

[0.   2.01881 2.0294]

input state inp and hidden state h are qualitatively different hence adding them together may result in loss of information hence better to concatenate them instead of adding them

In [161]:

```python
set_lrs(opt, 0.001)
```

In [162]:

```python
fit(m, md, 1, opt, F.nll_loss)
```

A Jupyter Widget

[0.   1.7161 1.7246]

loss

because concatenating now linear layer is n_fac+n_hidden to n_hidden instead of n_fac to n_hidden as before

In [163]:

```python
class CharLoopConcatModel(nn.Module):
    def __init__(self, vocab_size, n_fac):
        super().__init__()
        self.e = nn.Embedding(vocab_size, n_fac)
        self.l_in = nn.Linear(n_fac+n_hidden, n_hidden)
        self.l_hidden = nn.Linear(n_hidden, n_hidden)
        self.l_out = nn.Linear(n_hidden, vocab_size)

    def forward(self, *cs):
        bs = cs[0].size(0)
        h = V(torch.zeros(bs, n_hidden).cuda())
        for c in cs:
            inp = torch.cat((h, self.e(c)), 1)
            inp = F.relu(self.l_in(inp))
            h = F.tanh(self.l_hidden(inp))

        return F.log_softmax(self.l_out(h))
```

concatenating is a good design heuristic if you have different types of information you want to combine.

Adding information even if same shape generally means losing information

starting point for 'for' loop

this is now size n_fac + n_hidden

makes it back to size n_hidden

same square matrix as before so still size n_hidden

h = hidden layer here is a rank 2 tensor = 2 dimensions

In [164]:
```
m = CharLoopConcatModel(vocab_size, n_fac).cuda()
opt = optim.Adam(m.parameters(), 1e-3)
```

In [165]:
```
it = iter(md.trn_dl)
*xs,yt = next(it)
t = m(*V(xs))
```

In [166]:
```
fit(m, md, 1, opt, F.nll_loss)
```

```
A Jupyter Widget
[ 0.         1.80699   1.7688 ]
```

In [167]:
```
set_lrs(opt, 1e-4)
```

In [168]:
```
fit(m, md, 1, opt, F.nll_loss)
```

```
A Jupyter Widget
[ 0.         1.69154   1.68602]
```

concatenating reduces loss

Test model

In [46]:
```
def get_next(inp):
    idxs = T(np.array([char_indices[c] for c in inp]))
    p = m(*VV(idxs))
    i = np.argmax(to_np(p))
    return chars[i]
```

In [47]:
```
get_next('for thos')
```

can now pass in 8 chars

Out[47]: 'e'

In [48]:
```
get_next('part of ')
```

Out[48]: 't'

In [49]:
```
get_next('queens a')
```

Out[49]: 'n'

RNN with pytorch

same as before but using pytorch and less code

In [169]:
```
class CharRnn(nn.Module):
    def __init__(self, vocab_size, n_fac):
        super().__init__()
        self.e = nn.Embedding(vocab_size, n_fac)
        self.rnn = nn.RNN(n_fac, n_hidden)
        self.l_out = nn.Linear(n_hidden, vocab_size)

```

```
    def forward(self, *cs):
        bs = cs[0].size(0)
        h = V(torch.zeros(1, bs, n_hidden))
        inp = self.e(torch.stack(cs))
        outp,h = self.rnn(inp, h)

        return F.log_softmax(self.l_out(outp[-1]))
```

h - hidden layer here is a rank 3 tensor = 3 dimensions

starting point for 'for' loop

In [170]:
```
m = CharRnn(vocab_size, n_fac).cuda()
opt = optim.Adam(m.parameters(), 1e-3)
```

In [171]:
```
it = iter(md.trn_dl)
*xs,yt = next(it)
```

In [172]:
```
t = m.e(V(torch.stack(xs)))
t.size()
```

Out[172]:
```
torch.Size([8, 512, 42])
```

char size

batch size

size of the embedding matrix n_fac

In [173]:
```
ht = V(torch.zeros(1, 512,n_hidden))
outp, hn = m.rnn(t, ht)
outp.size(), hn.size()
```

Out[173]:
```
(torch.Size([8, 512, 256]), torch.Size([1, 512, 256]))
```

unit access = allows RNNs to be bi-directional in this case it is a 1 hence the reason the output is 1

n_hidden

In [175]:
```
t = m(*V(xs)); t.size()
```

Out[175]:
```
torch.Size([512, 85])
```

In [176]:
```
fit(m, md, 4, opt, F.nll_loss)
```

```
A Jupyter Widget

[ 0.        1.86162   1.84379]
[ 1.        1.66857   1.66926]
[ 2.        1.5879    1.60001]
[ 3.        1.55005   1.55722]
```

In [177]:
```
set_lrs(opt, 1e-4)
```

In [178]:
```
fit(m, md, 2, opt, F.nll_loss)
```

```
A Jupyter Widget

[ 0.        1.45798   1.5094 ]
[ 1.        1.45333   1.50419]
```

loss improves

Test model

In [179]:
```
def get_next(inp):
    idxs = T(np.array([char_indices[c] for c in inp]))
    p = m(*VV(idxs))
```

```python
    i = np.argmax(to_np(p))
    return chars[i]
```

In [180]:
```python
get_next('for thos')
```

Out[180]: 'e'

In [181]:
```python
def get_next_n(inp, n):
    res = inp
    for i in range(n):
        c = get_next(inp)
        res += c
        inp = inp[1:]+c
    return res
```

In [182]:
```python
get_next_n('for thos', 40)
```

Out[182]: 'for those of the same to the same to the same to'

Multi-output model

results should be the same as before but multi-output model is more efficient

Setup

Let's take non-overlapping sets of characters this time

In [29]:
```python
c_in_dat = [[idx[i+j] for i in range(cs)] for j in range(0, len(idx)-cs-1, cs)]
```

Then create the exact same thing, offset by 1, as our labels

In [30]:
```python
c_out_dat = [[idx[i+j] for i in range(cs)] for j in range(1, len(idx)-cs, cs)]
```

In [31]:
```python
xs = np.stack(c_in_dat)
xs.shape
```

shape now 75111 long instead of 600884 as we are looking every 8 non-overlapping characters (600884/8 = 75111)

Out[31]: (75111, 8)

In [32]:
```python
ys = np.stack(c_out_dat)
ys.shape
```

Out[32]: (75111, 8)

In this case using non-overlapping characters

In [33]:
```python
xs[:cs,:cs]
```

Out[33]:
```
array([[40, 42, 29, 30, 25, 27, 29,  1],
       [ 1,  1, 43, 45, 40, 40, 39, 43],
       [33, 38, 31,  2, 73, 61, 54, 73],
       [ 2, 44, 71, 74, 73, 61,  2, 62],
       [72,  2, 54,  2, 76, 68, 66, 54],
       [67,  9,  9, 76, 61, 54, 73,  2],
```

first 8 characters

next 8 characters

```
        [73, 61, 58, 67, 24,  2, 33, 72],
        [ 2, 73, 61, 58, 71, 58,  2, 67]])
```

In [34]:
```
ys[:cs,:cs]
```

Out[34]:
```
array([[42, 29, 30, 25, 27, 29,  1,  1],
       [ 1, 43, 45, 40, 40, 39, 43, 33],
       [38, 31,  2, 73, 61, 54, 73,  2],
       [44, 71, 74, 73, 61,  2, 62, 72],
       [ 2, 54,  2, 76, 68, 66, 54, 67],
       [ 9,  9, 76, 61, 54, 73,  2, 73],
       [61, 58, 67, 24,  2, 33, 72,  2],
       [73, 61, 58, 71, 58,  2, 67, 68]])
```

Create and train model

In [35]:
```
val_idx = get_cv_idxs(len(xs)-cs-1)
```

In [36]:
```
md = ColumnarModelData.from_arrays('.', val_idx, xs, ys, bs=512)
```

In [136]:
```
class CharSeqRnn(nn.Module):
    def __init__(self, vocab_size, n_fac):
        super().__init__()
        self.e = nn.Embedding(vocab_size, n_fac)
        self.rnn = nn.RNN(n_fac, n_hidden)
        self.l_out = nn.Linear(n_hidden, vocab_size)

    def forward(self, *cs):
        bs = cs[0].size(0)
        h = V(torch.zeros(1, bs, n_hidden))
        inp = self.e(torch.stack(cs))
        outp,h = self.rnn(inp, h)
        return F.log_softmax(self.l_out(outp))
```

In [137]:
```
m = CharSeqRnn(vocab_size, n_fac).cuda()
opt = optim.Adam(m.parameters(), 1e-3)
```

2nd axis = batch size = 512

```
it = iter(md.trn_dl)
*xst,yt = next(it)
```

have to write a custom loss function because pytorch takes into account tensor ranks

In [139]:
```
def nll_loss_seq(inp, targ):
    sl,bs,nh = inp.size()
    targ = targ.transpose(0,1).contiguous().view(-1)
    return F.nll_loss(inp.view(-1,nh), targ)
```

3rd axis = hidden state = 256

flatten targets

1st axis = sl = sequence length of RNN in this case 8 characters = time steps

flatten inputs

```
fit(m, md, 4, opt, nll_loss_seq)
```

```
A Jupyter Widget

[ 0.       1.1164   0.9535]
[ 1.       0.83636  0.76285]
[ 2.       0.71097  0.68187]
[ 3.       0.64594  0.63435]
```

In [141]:
```
set_lrs(opt, 1e-4)
```

In [142]:
```
fit(m, md, 1, opt, nll_loss_seq)
```

```
A Jupyter Wi
[ 0.        0
```

example size 3 identity matrix trick about identity matrixes is that something times an identity matrix produces itself preventing gradient explosion

$$9 \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = 9$$

Identity init!

In [145]:
```
m = CharSeqRnn(vocab_size, n_fac).cuda()
opt = optim.Adam(m.parameters(), 1e-2)
```

In [146]:
```
m.rnn.weight_hh_l0.data.copy_(torch.eye(n_hidden))
```

eye for identity

Out[146]:

identity matrix of size n-hidden

```
 1     0     0  ...     0     0     0
 0     1     0  ...     0     0     0
 0     0     1  ...     0     0     0
    ...          â‹±          ...
 0     0     0  ...     1     0     0
 0     0     0  ...     0     1     0
 0     0     0  ...     0     0     1
[torch.cuda.FloatTensor of size 256x256 (GPU 0)]
```

use ?m.rnn to get info on a matrix; inputs, outputs, atributes

In [147]:
```
fit(m, md, 4, opt, nll_loss_seq)
```

```
A Jupyter Widget

[ 0.        0.80381  0.7284 ]
[ 1.        0.66663  0.63739]
[ 2.        0.61271  0.6188 ]
[ 3.        0.59271  0.60541]
```

In [148]:
```
set_lrs(opt, 1e-3)
```

In [149]:
```
fit(m, md, 4, opt, nll_loss_seq)
```

```
A Jupyter Widget

[ 0.        0.5156   0.53407]
[ 1.        0.49554  0.52318]
[ 2.        0.48124  0.51721]
[ 3.        0.47085  0.51224]
```

In [150]:
```
set_lrs(opt, 1e-4)
```

In [151]:
```
fit(m, md, 4, opt, nll_loss_seq)
```

```
A Jupyter Widget
```

```
[ 0.         0.4549     0.50927]
[ 1.         0.45277    0.50849]
[ 2.         0.45234    0.50804]
[ 3.         0.45165    0.50758]
```

Stateful model

Setup

In [156]:
```
from torchtext import vocab, data
â€‹
from fastai.nlp import *
from fastai.lm_rnn import *
```

In [157]:
```
PATH='data/nietzsche/'
â€‹
TRN_PATH = 'trn/'
VAL_PATH = 'val/'
TRN = f'{PATH}{TRN_PATH}'
VAL = f'{PATH}{VAL_PATH}'
â€‹
%ls {PATH}
```

```
nietzsche.txt  trn/  val/
```

In [158]:
```
TEXT = data.Field(lower=True, tokenize=list)
```

In [159]:
```
bs=64; bptt=8; n_fac=42; n_hidden=256
```

In [160]:
```
FILES = dict(train=TRN_PATH, validation=VAL_PATH, test=VAL_PATH)
md = LanguageModelData.from_text_files(PATH, TEXT, **FILES, bs=bs, bptt=
n_freq=3)
```

In [161]:
```
len(md.trn_dl), md.nt, len(md.trn_ds), len(md.trn_ds[0].text)
```

Out[161]:
```
(963, 56, 1, 493747)
```

In [162]:
```
class CharSeqStatefulRnn(nn.Module):
    def __init__(self, vocab_size, n_fac, bs):
        super().__init__()
        self.e = nn.Embedding(vocab_size, n_fac)
        self.rnn = nn.RNN(n_fac, n_hidden)
        self.l_out = nn.Linear(n_hidden, vocab_size)
        self.h = V(torch.zeros(1, bs, n_hidden))

    def forward(self, cs):
        bs = cs[0].size(0)
        if self.h.size(1) != bs: self.h = V(torch.zeros(1, bs, n_hidden))
        inp = self.e(cs)
        outp,h = self.rnn(inp, self.h)
```

```
        self.h = repackage_var(h)
        return F.log_softmax(self.l_out(outp)).view(-1,vocab_size)
```

In [172]:
```
m = CharSeqStatefulRnn(vocab_size, n_fac, 512).cuda()
opt = optim.Adam(m.parameters(), 1e-3)
```

In [173]:
```
m.rnn.weight_hh_l0.data.copy_(torch.eye(n_hidden));
```

In [167]:
```
fit(m, md, 1, opt, F.nll_loss)
```

```
A Jupyter Widget

[ 0.       1.13513  1.08475]
```

In [174]:
```
set_lrs(opt, 1e-3)
```

In [175]:
```
fit(m, md, 2, opt, F.nll_loss)
```

```
A Jupyter Widget

[ 0.       0.96742  0.96103]
[ 1.       0.88555  0.8695 ]
```

In [176]:
```
fit(m, md, 2, opt, F.nll_loss)
```

```
A Jupyter Widget

[ 0.       0.87285  0.881  ]
[ 1.       0.87575  0.76261]
```

In [177]:
```
fit(m, md, 4, opt, F.nll_loss)
```

```
A Jupyter Widget

[ 0.       0.80812  0.82194]
[ 1.       0.85778  0.80804]
[ 2.       0.79613  0.73514]
[ 3.       0.827    0.78421]
```

In [178]:
```
fit(m, md, 4, opt, F.nll_loss)
```

```
A Jupyter Widget

[ 0.       0.73793  0.76126]
[ 1.       0.74701  0.7797 ]
[ 2.       0.69897  0.75077]
[ 3.       0.69903  0.73123]
```

In [179]:
```
set_lrs(opt, 1e-4)
```

In [180]:
```
fit(m, md, 4, opt, F.nll_loss)
```

```
A Jupyter Widget

[ 0.        0.65736  0.72496]
[ 1.        0.66048  0.74494]
[ 2.        0.68257  0.75462]
[ 3.        0.69551  0.76302]
```

Test

In [181]:

```python
def get_next(inp):
    idxs = TEXT.numericalize(inp)
    p = m(VV(idxs.transpose(0,1)))
#     print(p)
    r = np.argmax(to_np(p), axis=1)
#     print(r.shape)
    return TEXT.vocab.itos[r[-1]]
```

In [182]:

```python
get_next('for thos')
```

Out[182]: 'k'

In [181]:

```python
def get_next_n(inp, n):
    res = inp
    for i in range(n):
        c = get_next(inp)
        res += c
        inp = inp[1:]+c
    return res
```

In [182]:

```python
get_next_n('for thos', 40)
```

Out[182]: 'for those of the same to the same to the same to'

In [ ]:

```
â€‹
```