Notebook

April 28, 2023

1 Bigrams - MakeMore pt.1 (12/04/2023)

We want to learn something from a dataset of words. First of all, we need a machine which generates these data (the so-called words).

[22]:
```
words = open('data/nomi_italiani.txt').read().splitlines()
```

Let's figure out the length of each word in our dataset.

[23]:
```
L = [len(w) for w in words]
print(words[L.index(max(L))])
```

```
marie-odette-rose-gabrielle
```

[24]:
```
import numpy as np

print(np.mean(L))
```

```
7.088193300384404
```

[25]:
```
import random
random.seed(154)
random.shuffle(words)
print(words[:10])
```

```
['castorino', 'ella', 'irmo', 'leoluca', 'sankhare', 'galvano', 'faleria',
'germando', 'illo', 'romilde']
```

[26]:
```
for w in words[:1]:
    for ch1, ch2 in zip(w, w[1:]):
        print(ch1, ch2)
```

```
c a
a s
s t
t o
o r
r i
```

```
i n
n o
```

[27]:

```python
print(w)
list(w)
print(w[1:])
```

```
castorino
astorino
```

To get all bigrams of, for example, the last three words one can do something like that

[28]:

```python
for w in words[:3]:
    chs = ['<S>'] + list(w) + ['<E>']
    for ch1, ch2 in zip(chs, chs[1:]):
        print(ch1, ch2)
```

```
<S> c
c a
a s
s t
t o
o r
r i
i n
n o
o <E>
<S> e
e l
l l
l a
a <E>
<S> i
i r
r m
m o
o <E>
```

How many often does a bigram happen?

[29]:

```python
b = {}
for w in words[:3]:
    chs = ['<S>'] + list(w) + ['<E>']
    for ch1, ch2 in zip(chs, chs[1:]):
        b[(ch1, ch2)] = b.get((ch1, ch2), 0) + 1
print(b)
```

```
{('<S>', 'c'): 1, ('c', 'a'): 1, ('a', 's'): 1, ('s', 't'): 1, ('t', 'o'): 1,
('o', 'r'): 1, ('r', 'i'): 1, ('i', 'n'): 1, ('n', 'o'): 1, ('o', '<E>'): 2,
```

```
('<S>', 'e'): 1, ('e', 'l'): 1, ('l', 'l'): 1, ('l', 'a'): 1, ('a', '<E>'): 1,
('<S>', 'i'): 1, ('i', 'r'): 1, ('r', 'm'): 1, ('m', 'o'): 1}
```

We can do it for all words

[30]:
```python
b = {}
for w in words:
    chs = ['<S>'] + list(w) + ['<E>']
    for ch1, ch2 in zip(chs, chs[1:]):
        b[(ch1, ch2)] = b.get((ch1, ch2), 0) + 1
```

We now want to construct a machine which tells us the probability of the next character. Let's sort all by frequency

[31]:
```python
print(sorted(b.items(), key=lambda z: -z[1]))
```

```
[(('o', '<E>'), 4235), (('a', '<E>'), 3831), (('i', 'n'), 1935), (('a', 'n'),
1497), (('r', 'i'), 1483), (('n', 'a'), 1429), (('i', 'o'), 1380), (('i', 'a'),
1344), (('n', 'o'), 1269), (('l', 'i'), 1261), (('e', 'r'), 1149), (('<S>',
'a'), 1095), (('e', 'l'), 1070), (('a', 'r'), 957), (('o', 'r'), 920), (('<S>',
'f'), 885), (('a', 'l'), 856), (('<S>', 'o'), 789), (('d', 'o'), 754), (('i',
'l'), 712), (('r', 'o'), 705), (('e', 'n'), 699), (('r', 'a'), 686), (('l',
'a'), 682), (('m', 'a'), 663), (('<S>', 'g'), 657), (('n', 'i'), 649), (('e',
'<E>'), 630), (('<S>', 'e'), 629), (('<S>', 'r'), 626), (('<S>', 'm'), 595),
(('o', 'n'), 591), (('r', 'e'), 584), (('t', 'a'), 580), (('d', 'a'), 565),
(('t', 'o'), 559), (('d', 'i'), 552), (('o', 'l'), 545), (('d', 'e'), 540),
(('l', 'l'), 537), (('m', 'i'), 532), (('s', 'i'), 531), (('<S>', 'c'), 520),
(('l', 'e'), 511), (('g', 'i'), 496), (('i', 's'), 482), (('l', 'o'), 474),
(('<S>', 'l'), 446), (('n', 'e'), 437), (('n', 'd'), 433), (('t', 'i'), 430),
(('l', 'd'), 405), (('i', 'd'), 403), (('v', 'i'), 398), (('t', 't'), 392),
(('<S>', 's'), 391), (('f', 'i'), 387), (('e', 't'), 386), (('t', 'e'), 378),
(('c', 'o'), 364), (('<S>', 'd'), 351), (('z', 'i'), 350), (('<S>', 'p'), 345),
(('i', 'c'), 342), (('m', 'e'), 335), (('n', 't'), 334), (('c', 'a'), 333),
(('s', 'a'), 333), (('e', 's'), 331), (('c', 'i'), 326), (('o', 's'), 322),
(('<S>', 'i'), 319), (('<S>', 'n'), 313), (('i', 'e'), 311), (('s', 't'), 307),
(('<S>', 'v'), 297), (('<S>', 'b'), 293), (('f', 'e'), 283), (('v', 'a'), 279),
(('g', 'e'), 272), (('i', 'r'), 268), (('m', 'o'), 263), (('a', 't'), 255),
(('b', 'e'), 255), (('a', 'm'), 254), (('a', 'd'), 251), (('v', 'e'), 245),
(('e', 'd'), 245), (('r', 'd'), 237), (('n', 'z'), 237), (('a', 's'), 236),
(('c', 'e'), 235), (('r', 't'), 234), (('<S>', 't'), 226), (('i', 'm'), 225),
(('s', 'e'), 220), (('e', 'o'), 219), (('n', 'n'), 217), (('e', 'm'), 215),
(('i', 't'), 206), (('i', 'g'), 195), (('e', 'a'), 189), (('f', 'a'), 187),
(('r', 'm'), 182), (('o', 'd'), 181), (('o', 'm'), 177), (('c', 'c'), 175),
(('s', 'o'), 167), (('f', 'r'), 166), (('p', 'i'), 164), (('u', 'r'), 163),
(('l', 'm'), 160), (('g', 'a'), 156), (('b', 'a'), 155), (('u', 'c'), 154),
(('o', 'v'), 145), (('i', '<E>'), 143), (('p', 'a'), 142), (('l', 'v'), 140),
(('p', 'e'), 139), (('l', 'f'), 139), (('a', 'u'), 138), (('c', 'h'), 137),
(('i', 'v'), 132), (('b', 'i'), 126), (('a', 'c'), 123), (('f', 'o'), 121),
(('g', 'o'), 121), (('u', 'i'), 117), (('s', 's'), 116), (('a', 'v'), 115),
```

(('b', 'r'), 113), (('s', 'c'), 113), (('u', 'l'), 113), (('r', 'u'), 112),
(('l', 'u'), 110), (('z', 'a'), 109), (('c', 'l'), 109), (('o', 't'), 107),
(('a', 'b'), 106), (('f', 'l'), 105), (('h', 'i'), 103), (('u', 's'), 103),
(('n', 'u'), 102), (('t', 'r'), 102), (('n', 'c'), 100), (('e', 'g'), 100),
(('a', 'z'), 99), (('d', 'r'), 97), (('<S>', 'z'), 97), (('g', 'l'), 94), (('a',
'g'), 93), (('r', 'n'), 93), (('n', 'g'), 91), (('s', '<E>'), 90), (('p', 'r'),
90), (('p', 'o'), 90), (('i', 'z'), 90), (('u', 'n'), 89), (('z', 'o'), 86),
(('t', 'u'), 86), (('e', 'v'), 82), (('e', 'u'), 81), (('b', 'o'), 80), (('<S>',
'u'), 80), (('z', 'e'), 79), (('r', 'c'), 77), (('r', 'r'), 75), (('z', 'z'),
75), (('r', 'g'), 74), (('a', 'i'), 74), (('q', 'u'), 73), (('i', 'b'), 72),
(('o', 'b'), 71), (('r', 's'), 70), (('h', 'e'), 68), (('l', 'b'), 68), (('u',
'd'), 68), (('s', 'p'), 67), (('o', 'c'), 67), (('g', 'u'), 65), (('a', 'f'),
61), (('n', 's'), 61), (('i', 'u'), 59), (('c', 'r'), 59), (('l', 'c'), 58),
(('l', 't'), 57), (('r', 'l'), 57), (('s', 'm'), 57), (('v', 'o'), 56), (('f',
'f'), 56), (('g', 'r'), 56), (('m', 'b'), 55), (('<S>', 'w'), 55), (('u', 'e'),
53), (('u', 'a'), 53), (('r', '<E>'), 52), (('m', 'm'), 50), (('d', 'u'), 50),
(('p', 'p'), 49), (('s', 'u'), 48), (('u', 't'), 47), (('e', 'f'), 47), (('<S>',
'q'), 47), (('e', 'z'), 47), (('r', 'v'), 46), (('e', 'p'), 46), (('a', '-'),
45), (('o', 'p'), 45), (('i', 'p'), 44), (('o', 'f'), 44), (('l', 'g'), 44),
(('a', 'o'), 42), (('e', 'c'), 41), (('r', 'z'), 40), (('u', 'g'), 39), (('a',
'e'), 37), (('n', '<E>'), 37), (('m', 'p'), 35), (('f', 'u'), 34), (('u', 'b'),
33), (('d', 'd'), 33), (('g', 'h'), 31), (('<S>', 'j'), 31), (('o', 'i'), 31),
(('w', 'a'), 30), (('g', 'n'), 29), (('c', 'u'), 29), (('g', 'g'), 29), (('b',
'b'), 28), (('m', 'u'), 28), (('n', 'f'), 28), (('e', 'b'), 28), (('u', 'm'),
27), (('a', 'p'), 26), (('i', 'f'), 26), (('h', 'a'), 25), (('r', 'f'), 25),
(('o', 'g'), 25), (('o', 'a'), 24), (('s', 'l'), 24), (('j', 'a'), 24), (('s',
'v'), 23), (('l', 's'), 23), (('u', 'f'), 22), (('r', 'b'), 22), (('e', 'i'),
21), (('p', 'l'), 19), (('o', 'e'), 19), (('w', 'i'), 17), (('u', 'p'), 17),
(('b', 'u'), 16), (('l', '<E>'), 16), (('o', '-'), 16), (('u', 'z'), 16), (('k',
'a'), 15), (('d', '<E>'), 15), (('j', 'o'), 14), (('e', '-'), 14), (('u', 'o'),
13), (('-', 'm'), 13), (('-', 'a'), 13), (('p', 'u'), 13), (('l', 'p'), 12),
(('-', 'r'), 12), (('u', '<E>'), 12), (('s', 'q'), 12), (('r', 'p'), 11),
(('<S>', 'k'), 11), (('b', 'l'), 11), (('n', 'r'), 10), (('n', '-'), 10), (('w',
'e'), 9), (('-', 'g'), 9), (('d', 'm'), 9), (('t', 'h'), 9), (('l', 'z'), 8),
(('m', '<E>'), 8), (('a', 'h'), 8), (('n', 'm'), 8), (('s', 'd'), 7), (('z',
'u'), 7), (('n', 'l'), 7), (('-', 'e'), 7), (('e', 'e'), 6), (('t', '<E>'), 6),
(('s', 'f'), 6), (('y', 'n'), 6), (('-', 'j'), 6), (('r', 'k'), 6), (('v', 'v'),
6), (('v', 'r'), 6), (('s', 'z'), 6), (('n', 'p'), 6), (('g', 'd'), 5), (('s',
'b'), 5), (('a', 'q'), 5), (('o', 'h'), 5), (('i', 'k'), 5), (('b', 'd'), 5),
(('k', 'o'), 5), (('o', 'z'), 5), (('g', 'f'), 5), (('-', 'p'), 5), (('s', 'h'),
5), (('o', 'u'), 5), (('r', 'q'), 5), (('k', 'h'), 4), (('-', 'd'), 4), (('-',
'i'), 4), (('<S>', 'y'), 4), (('z', 'b'), 4), (('x', 'a'), 4), (('y', '<E>'),
4), (('h', 'r'), 4), (('k', 'r'), 4), (('t', 'y'), 4), (('l', 'r'), 4), (('a',
'w'), 4), (('j', 'u'), 4), (('z', '<E>'), 4), (('d', 'v'), 4), (('h', '<E>'),
4), (('-', 'l'), 4), (('-', 'c'), 4), (('-', 'o'), 3), (('y', 'o'), 3), (('l',
'y'), 3), (('v', 'l'), 3), (('r', 'y'), 3), (('y', 's'), 3), (('-', 's'), 3),
(('f', 'n'), 3), (('l', 'k'), 3), (('w', 'o'), 3), (('k', 'i'), 3), (('e', 'x'),
3), (('n', 'v'), 3), (('j', 'e'), 3), (('x', 'i'), 3), (('w', '<E>'), 3), (('p',

```
'h'), 3), (('i', 'j'), 3), (('h', 'o'), 3), (('e', 'w'), 3), (('<S>', 'h'), 3),
(('r', 'x'), 3), (('k', '<E>'), 3), (('s', 'r'), 3), (('n', 'q'), 3), (('d',
'g'), 3), (('z', 'y'), 3), (('n', 'k'), 2), (('c', 'y'), 2), (('y', 'u'), 2),
(('v', '<E>'), 2), (('e', 'j'), 2), (('j', 'i'), 2), (('-', 'b'), 2), (('-',
'v'), 2), (('-', 'h'), 2), (('w', 'l'), 2), (('z', 'k'), 2), (('y', 'a'), 2),
(('s', '-'), 2), (('d', '-'), 2), (('-', 'k'), 2), (('v', 'u'), 2), (('t', '-'),
2), (('a', 'y'), 2), (('l', 'n'), 2), (('s', 'k'), 2), (('n', 'b'), 2), (('e',
'k'), 2), (('m', 'l'), 2), (('-', 'n'), 2), (('f', '<E>'), 2), (('n', 'j'), 2),
(('y', 'l'), 2), (('u', 'j'), 2), (('c', 't'), 2), (('t', 's'), 2), (('u', 'k'),
2), (('r', '-'), 2), (('c', '<E>'), 2), (('x', '<E>'), 2), (('c', 'm'), 1),
(('o', 'x'), 1), (('b', '-'), 1), (('y', 'v'), 1), (('m', 'y'), 1), (('y', 'r'),
1), (('-', 'y'), 1), (('s', 'n'), 1), (('i', 'x'), 1), (('-', 'f'), 1), (('t',
'l'), 1), (('r', 'j'), 1), (('m', '-'), 1), (('t', 'b'), 1), (('d', 'j'), 1),
(('k', 'e'), 1), (('k', 'b'), 1), (('a', 'j'), 1), (('t', 'g'), 1), (('-', 'w'),
1), (('k', '-'), 1), (('-', 'z'), 1), (('n', 'w'), 1), (('b', '<E>'), 1), (('o',
'k'), 1), (('k', 's'), 1), (('j', '<E>'), 1), (('y', 'k'), 1), (('m', 'n'), 1),
(('l', '-'), 1), (('i', '-'), 1), (('m', 's'), 1), (('t', 'p'), 1), (('o', 'y'),
1), (('y', 'c'), 1), (('w', 'u'), 1), (('d', 'y'), 1), (('z', 't'), 1), (('f',
'-'), 1), (('z', '-'), 1), (('n', 'y'), 1), (('a', 'a'), 1), (('d', 'h'), 1),
(('-', '<E>'), 1), (('u', '-'), 1), (('d', 'w'), 1), (('u', 'v'), 1), (('j',
'n'), 1), (('h', '-'), 1), (('t', 'm'), 1), (('j', 'j'), 1), (('a', 'x'), 1),
(('c', 'q'), 1), (('l', 'h'), 1), (('h', 'm'), 1), (('k', 't'), 1), (('g',
'<E>'), 1)]
```

With the set function built-in python one can get the unique elements of a list

[32]:
```python
w = set(list(words[1]))
print(w)
```

```
{'e', 'a', 'l'}
```

Now we want to code this information Let's take only the unique elements of a word, then of all words, i.e. our finite alphabet

[33]:
```python
chars = sorted(list(set(''.join(words))))
chars.append('<S>')
chars.append('<E>')
print(chars)
```

```
['-', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z', '<S>', '<E>']
```

At this point we should have 27 characters, 26 letters of the alphabet and the dash from composite names

[34]:
```python
print(len(chars))
```

```
29
```

Actually we need 29 characters, i.e. the 27 previously discussed plus the initial and final of a word

[35]:
```
import torch

N = torch.zeros(29, 29)
```

Let's now build an encoder, which assigns an integer value to each character of our alphabet. This will be a dictionary.

[36]:
```
stoi = {s:i for i, s in enumerate(chars)}
stoi['<S>'] = 27
stoi['<E>'] = 28
print(stoi)
```

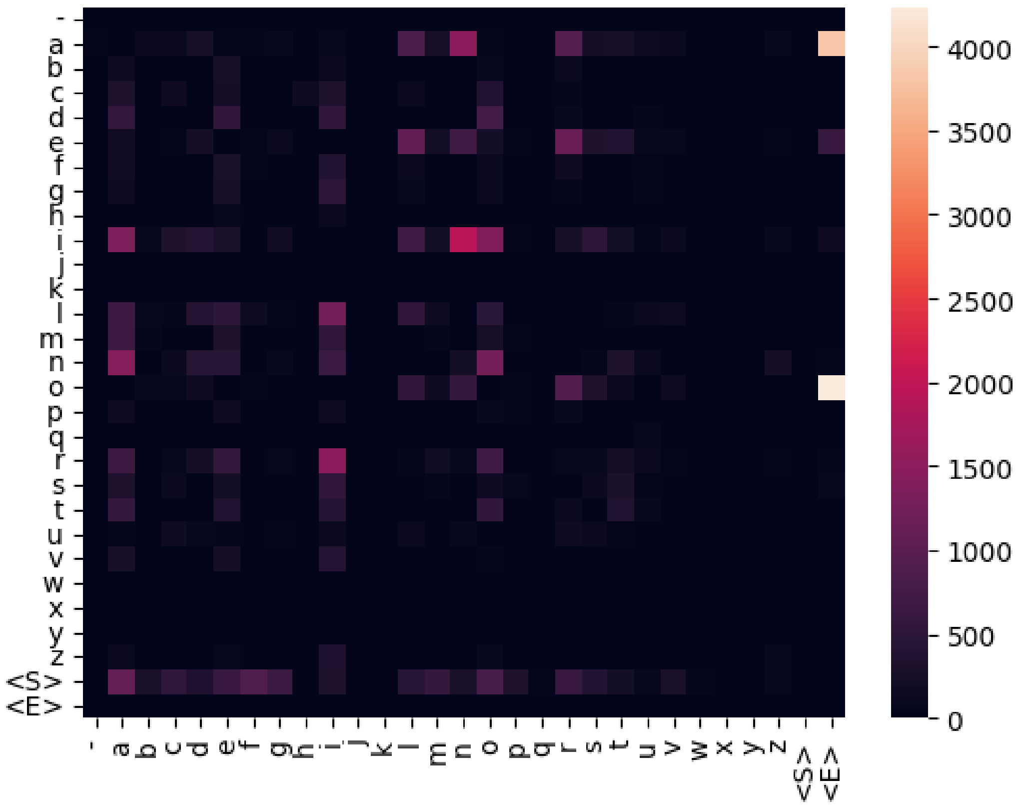
```
{'-': 0, 'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5, 'f': 6, 'g': 7, 'h': 8, 'i': 9,
'j': 10, 'k': 11, 'l': 12, 'm': 13, 'n': 14, 'o': 15, 'p': 16, 'q': 17, 'r': 18,
's': 19, 't': 20, 'u': 21, 'v': 22, 'w': 23, 'x': 24, 'y': 25, 'z': 26, '<S>':
27, '<E>': 28}
```

Now let's count the frequency of each bigram and put it in our tensor.

[37]:
```
for w in words:
    chs = ['<S>'] + list(w) + ['<E>']
    for ch1, ch2 in zip(chs, chs[1:]):
        N[stoi[ch1], stoi[ch2]] += 1
N[28,27] = 1 # <E> <S>
```

[38]:
```
import seaborn as sns
sns.heatmap(N, xticklabels=chars, yticklabels=chars)
```

[38]:
```
<Axes: >
```



Now one can also build a decoder

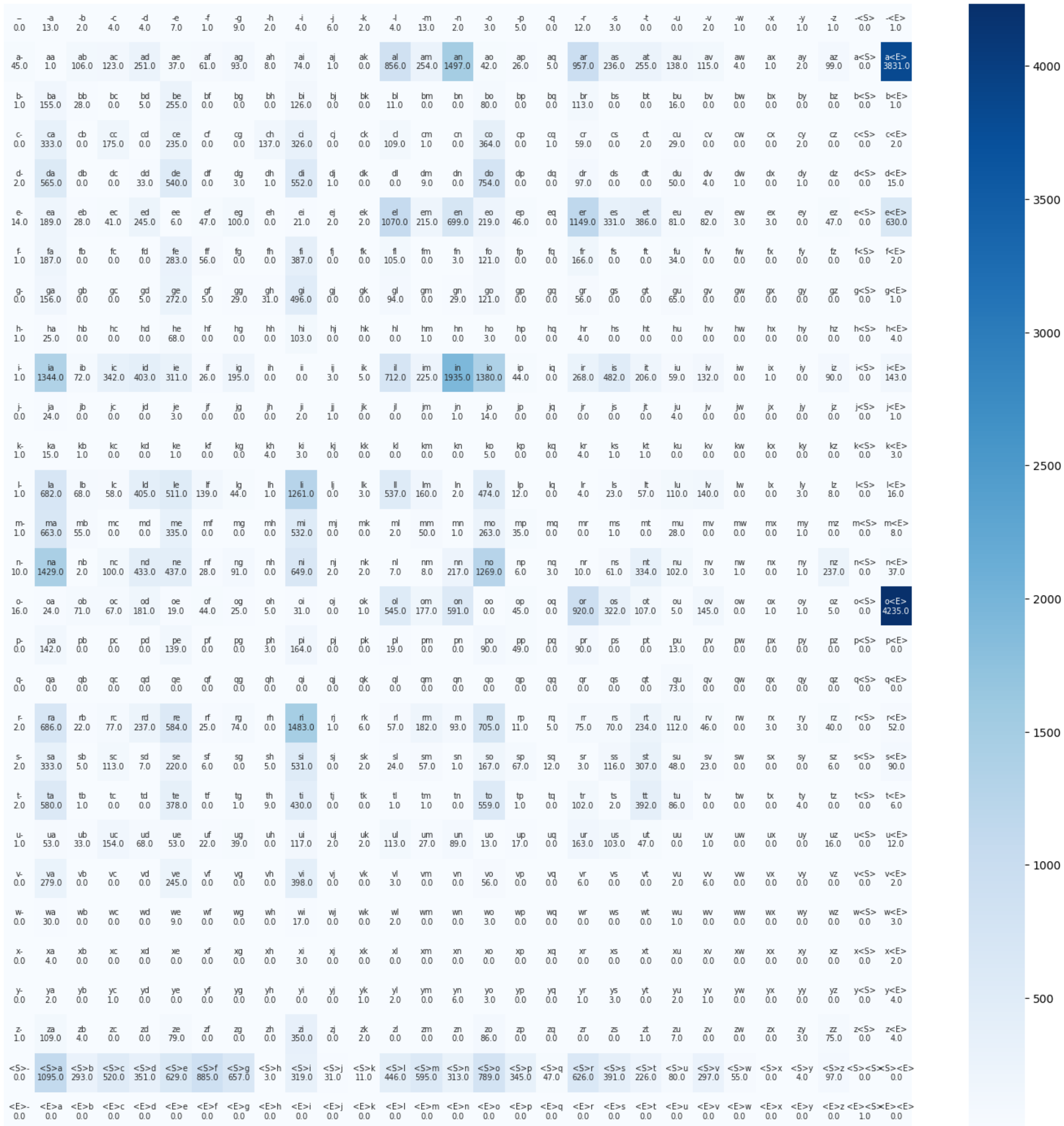
[39]:
```
itos = {i:s for s, i in stoi.items()}
print(itos)
```

```
{0: '-', 1: 'a', 2: 'b', 3: 'c', 4: 'd', 5: 'e', 6: 'f', 7: 'g', 8: 'h', 9: 'i',
10: 'j', 11: 'k', 12: 'l', 13: 'm', 14: 'n', 15: 'o', 16: 'p', 17: 'q', 18: 'r',
19: 's', 20: 't', 21: 'u', 22: 'v', 23: 'w', 24: 'x', 25: 'y', 26: 'z', 27:
'<S>', 28: '<E>'}
```

For a better visualization we can plot the whole matrix together with the bigrams and their frequencies

[40]:
```
from matplotlib import pyplot as plt
fig, ax = plt.subplots(figsize=(18, 18))
labels = np.array([[itos[i] + itos[j] + '\n' + str(N[i,j].item()) for j in range(29)] for i in range(29)])
sns.heatmap(N, xticklabels=False, yticklabels=False, fmt='', cmap='Blues', annot=labels, annot_kws={"fontsize":6.9}) # nice font size
```

[40]:
```
<Axes: >
```



How can we use counting to infer our probability? How can we reproduce the probability distribution of these numbers?

Let's start by computing the probability of the first character. First, normalize all the rows of the tensor, then sample using frequencies. It is possible, with PyTorch, to normalize all the rows of a matrix (stochastic on the row) by doing M/M.sum(...). Assume that $\vec{p}$ is now our 27th row normalized, i.e. '<S>' + '%c' string, which represents the frequency of starting letters. Using the conditional probability known by the dataset one can start to generate words basing on bigrams, actually in a Markov chain approximation. However, the result is not properly good (is very, very bad ngl). Words must be extracted with repetition from our $\vec{p}$ vector. Actually, an integer is generated, not a word.

TRIVIA ChatGPT has a vocabulary of words (chunks - word pieces), not characters.

[41]:
```
p = N / N.sum(axis=1, keepdims=True)
g=torch.Generator().manual_seed(123450)

for i in range(10):
    out=[]
    ix=27
    while True:
        pix = p[ix]
        ix=torch.multinomial(pix,num_samples=1,replacement=True,generator=g).
 ↪item()
        out.append(itos[ix])
        if ix==28:
            break
    print(''.join(out))
```

```
epucia<E>
o<E>
lgomenzano<E>
a<E>
ginttio<E>
s<E>
ciniglclalirermo<E>
feonedo<E>
mola<E>
ndo<E>
```

What if the probability distribution was uniform?

[42]:
```
for i in range(10):
    out=[]
    ix=27
    while True:
        pix=torch.ones(29)/29.0
        ix=torch.multinomial(pix,num_samples=1,replacement=True,generator=g).
 ↪item()
        out.append(itos[ix])
        if ix==28:
            break
    print(''.join(out))
```

```
aaognqusx<E>
sx<S>ukksqzwca<S><E>
zvksvkbglkndtjxwfra<S>fotxorg<S>x<S>mytxlnvdrqordxkclczhn-ynrqjp-l<E>
xwcsruntl-rdkfbzexahxcxoxa-ucfnoae<E>
natlmee<S>yxjtpynrn-zsps<E>
zmpaaxp<E>
```

```
cidlwhsmmllkndpzawmdxkhnnyghb<E>
cchgmhzv<S>lf<S>sfwrwwolpb<S>-itrmv-j-lld<S>gouh<E>
mmuykixwsdddbaopvxqldjwwy<S>my-trkjwbjdoqkkxepmw<S>cjv<E>
lzyywd<S>kwu<E>
```

Words generated with uniform distribution are way worse, so just using the probability of the dataset (simple information - just counting) one can achieve a quite good result with respect to the total randomness.

How to evaluate an algorithm like this one? (just one char memory) How can we do better?

2 Trigrams - MakeMore2 (19/04/2023)

[443]:
```
# https://youtu.be/TCH_1BHY58I
# https://github.com/karpathy/makemore
```

Now we'll try to build a multilayer perceptron (MLP). Each character is going to be embedded in a 2D space. We've three vectors 30D, so 90D as total dimension. (28 chars + 2) Training the network the embedding will change. We'll have a linear transformation which transpose in an intermediate layer we can see as 100D vector. Transforming this non-linearly (with a hyperbolic tangent) it will construct the derivatives (back propagation). With another linear transform we'll connect all. Exponentiating and normalizing we'll get the desired probability distribution. Hyperparameters are the a priori defined parameters. To do things in a good way one needs to know how to tune the hyperparameters.

[444]:
```
# we now go to MLP (multilayer perceptron)....(using NLP (natural language␣
↪processing))
# 'a neural probabilistic language model' (2003) chrome-extension://
↪efaidnbmnnnibpcajpcglclefindmkaj/
# https://www.jmlr.org/papers/volume3/bengio03a/bengio03a.pdf
# fig 1: 4th word predicted after the three....
import random
import torch
import torch.nn.functional as  F
import matplotlib.pyplot as plt
%matplotlib inline
```

[445]:
```
# read in all the words
random.seed(158)
words=open('data/nomi_italiani.txt','r').read().splitlines()
random.shuffle(words)
print(words[0:10])
print(len(words))
```

```
['argento', 'giovannino', 'licurga', 'elvira', 'marena', 'sirio', 'emilia',
'bisio', 'preziosa', 'perpetua']
9105
```

[446]:
```python
# build the vocabulary of characters and mapping to/from integers
chars=sorted(list(set(''.join(words))))

stoi={s:i+1 for i,s in enumerate(chars)}
stoi['.']=0
itos ={i:s for s,i in stoi.items()}
print(itos)
print(stoi)
```

```
{1: '-', 2: 'a', 3: 'b', 4: 'c', 5: 'd', 6: 'e', 7: 'f', 8: 'g', 9: 'h', 10:
'i', 11: 'j', 12: 'k', 13: 'l', 14: 'm', 15: 'n', 16: 'o', 17: 'p', 18: 'q', 19:
'r', 20: 's', 21: 't', 22: 'u', 23: 'v', 24: 'w', 25: 'x', 26: 'y', 27: 'z', 0:
'.'}
{'-': 1, 'a': 2, 'b': 3, 'c': 4, 'd': 5, 'e': 6, 'f': 7, 'g': 8, 'h': 9, 'i':
10, 'j': 11, 'k': 12, 'l': 13, 'm': 14, 'n': 15, 'o': 16, 'p': 17, 'q': 18, 'r':
19, 's': 20, 't': 21, 'u': 22, 'v': 23, 'w': 24, 'x': 25, 'y': 26, 'z': 27, '.':
0}
```

Previous example - Markov chain, so the block size was 1. Updating the contest means shift over the string and add the last character. X contains the samples (what I'm looking at). Each row of X is a trigram. Y contains the correct answers.

[447]:
```python
# build the dataset

block_size = 3 #context length: how many characters do we take to predict the
 ↪next one ... change it !!
# try: block_size=1 ...Markov Chain, then try = 2 and =10
X,Y = [],[]  # input & label

for w in words[0:5]:
    print(w)
    context=[0]*block_size # 000 corresponds to the character '...'
    for ch in w +'.':
        ix=stoi[ch]
        X.append(context)
        Y.append(ix)
        print(''.join(itos[i] for i in context), '--->', itos[ix])
        context=context[1:]+[ix]  # shift: crop and append
X=torch.tensor(X)
Y=torch.tensor(Y)
```

```
argento
… ---> a
..a ---> r
.ar ---> g
arg ---> e
rge ---> n
gen ---> t
```

```
ent ---> o
nto ---> .
giovannino
… ---> g
..g ---> i
.gi ---> o
gio ---> v
iov ---> a
ova ---> n
van ---> n
ann ---> i
nni ---> n
nin ---> o
ino ---> .
licurga
… ---> l
..l ---> i
.li ---> c
lic ---> u
icu ---> r
cur ---> g
urg ---> a
rga ---> .
elvira
… ---> e
..e ---> l
.el ---> v
elv ---> i
lvi ---> r
vir ---> a
ira ---> .
marena
… ---> m
..m ---> a
.ma ---> r
mar ---> e
are ---> n
ren ---> a
ena ---> .
```

[448]:
```python
print(X)
print(Y)
```

```
tensor([[ 0,  0,  0],
        [ 0,  0,  2],
        [ 0,  2, 19],
        [ 2, 19,  8],
        [19,  8,  6],
```

```
        [ 8,  6, 15],
        [ 6, 15, 21],
        [15, 21, 16],
        [ 0,  0,  0],
        [ 0,  0,  8],
        [ 0,  8, 10],
        [ 8, 10, 16],
        [10, 16, 23],
        [16, 23,  2],
        [23,  2, 15],
        [ 2, 15, 15],
        [15, 15, 10],
        [15, 10, 15],
        [10, 15, 16],
        [ 0,  0,  0],
        [ 0,  0, 13],
        [ 0, 13, 10],
        [13, 10,  4],
        [10,  4, 22],
        [ 4, 22, 19],
        [22, 19,  8],
        [19,  8,  2],
        [ 0,  0,  0],
        [ 0,  0,  6],
        [ 0,  6, 13],
        [ 6, 13, 23],
        [13, 23, 10],
        [23, 10, 19],
        [10, 19,  2],
        [ 0,  0,  0],
        [ 0,  0, 14],
        [ 0, 14,  2],
        [14,  2, 19],
        [ 2, 19,  6],
        [19,  6, 15],
        [ 6, 15,  2]])
tensor([ 2, 19,  8,  6, 15, 21, 16,  0,  8, 10, 16, 23,  2, 15, 15, 10, 15, 16,
         0, 13, 10,  4, 22, 19,  8,  2,  0,  6, 13, 23, 10, 19,  2,  0, 14,  2,
        19,  6, 15,  2,  0])
```

[449]:
```
print(X.shape, X.dtype, Y.shape, Y.dtype)
```

```
torch.Size([41, 3]) torch.int64 torch.Size([41]) torch.int64
```

Now we want to predict the next character starting from trigrams. We're going to take a 2D embedding of the 28 characters. There are many pre-calculated embeddings in the world.

We can generate a random (normal) matrix 28x2. In deep data analysis (what we're doing) the world is going really fast. There is a lot of material, 99% of which are bullshits.

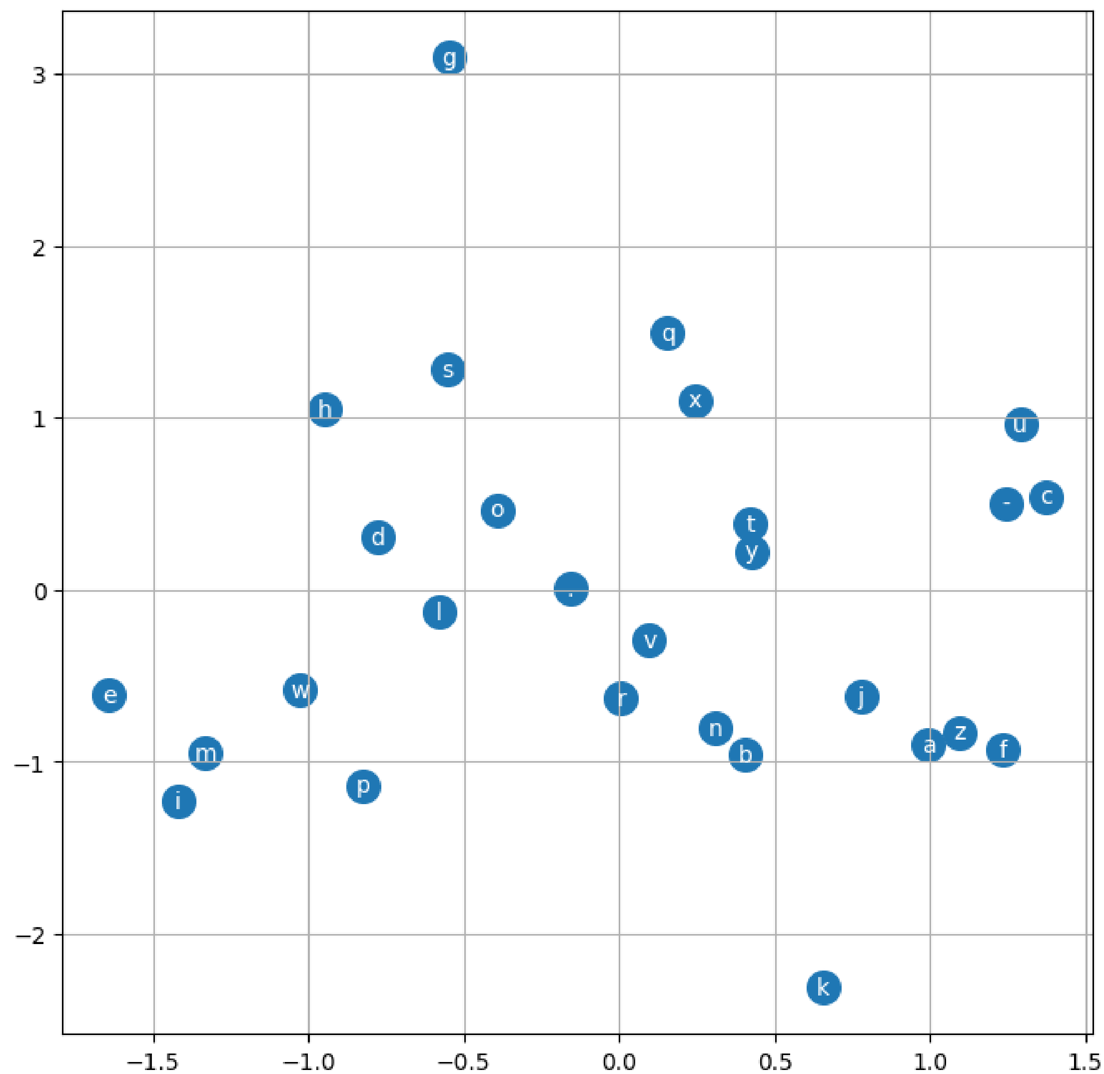
[450]:
```
C = torch.randn((28,2))  #https://pytorch.org/docs/stable/generated/torch.randn.
↪html
```

[451]:
```
print(C[5])
print(C.shape)
```

```
tensor([-0.7776,  0.3077])
torch.Size([28, 2])
```

Here is a plot of the embedding. Letters are random, after the training this picture is going to change. From this plot we can learn how characters are related each other.

[452]:
```
plt.figure(figsize=(8,8))
plt.scatter(C[:,0].data, C[:,1].data,s=200)
for i in range(C.shape[0]):
    plt.text(C[i,0].item(), C[i,1].item(),itos[i], ha="center",va="center",␣
↪color="white")
    plt.grid('minor')
```



Another way to embed characters is the one-hot embedding discussed last lecture. We can see this embedding as the first layer of our network, even if there's no linearity in it. In fact, they're completely equivalent. One can also see all the embeddings.

[453]:
```
print(F.one_hot(torch.tensor(5),num_classes=28))
print(F.one_hot(torch.tensor(5),num_classes=28).float()@C)
print(C[5])
print(C[Y])
```

```
tensor([0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0])
tensor([-0.7776,  0.3077])
tensor([-0.7776,  0.3077])
tensor([[ 0.9940, -0.8991],
```

```
        [ 0.0061, -0.6300],
        [-0.5460,  3.0955],
        [-1.6397, -0.6131],
        [ 0.3087, -0.8025],
        [ 0.4220,  0.3822],
        [-0.3936,  0.4641],
        [-0.1584,  0.0036],
        [-0.5460,  3.0955],
        [-1.4187, -1.2333],
        [-0.3936,  0.4641],
        [ 0.0970, -0.2923],
        [ 0.9940, -0.8991],
        [ 0.3087, -0.8025],
        [ 0.3087, -0.8025],
        [-1.4187, -1.2333],
        [ 0.3087, -0.8025],
        [-0.3936,  0.4641],
        [-0.1584,  0.0036],
        [-0.5780, -0.1284],
        [-1.4187, -1.2333],
        [ 1.3726,  0.5412],
        [ 1.2897,  0.9605],
        [ 0.0061, -0.6300],
        [-0.5460,  3.0955],
        [ 0.9940, -0.8991],
        [-0.1584,  0.0036],
        [-1.6397, -0.6131],
        [-0.5780, -0.1284],
        [ 0.0970, -0.2923],
        [-1.4187, -1.2333],
        [ 0.0061, -0.6300],
        [ 0.9940, -0.8991],
        [-0.1584,  0.0036],
        [-1.3297, -0.9474],
        [ 0.9940, -0.8991],
        [ 0.0061, -0.6300],
        [-1.6397, -0.6131],
        [ 0.3087, -0.8025],
        [ 0.9940, -0.8991],
        [-0.1584,  0.0036]])
```

How to embed the 41 trigrams we have?

[454]:
```
emb = C[X]
print(emb.shape)
```

```
torch.Size([41, 3, 2])
```

Moreover, we can *differentiate* C! Input has dimension 6 = 3 * 2

[455]:
```
# construct the Layer.... x.W+ b ... so the input has dimension 6=3*2 for (say)␣
↪100 neurons...
W1 = torch.randn(6,100)
b1 = torch.randn(100)
```

We want to concatenate tensors. And maybe unbind them.

[456]:
```
# https://pytorch.org/docs/stable/torch.html search for concatenate...

print(torch.cat([emb[:,0,:],emb[:,1,:],emb[:,2,:]],1)[1])
print(emb[1])
#torch.cat([emb[:,0,:],emb[:,1,:],emb[:,2,:]],1).shape
```

```
tensor([-0.1584,  0.0036, -0.1584,  0.0036,  0.9940, -0.8991])
tensor([[-0.1584,  0.0036],
        [-0.1584,  0.0036],
        [ 0.9940, -0.8991]])
```

[457]:
```
# we want a code for general n-grams.....
# use 'unbind' https://pytorch.org/docs/stable/generated/torch.unbind.
↪html#torch.unbind
len(torch.unbind(emb,1))
```

[457]:
```
3
```

[458]:
```
# and this work fore any context length.......

torch.cat(torch.unbind(emb,1),1)
```

[458]:
```
tensor([[-0.1584,  0.0036, -0.1584,  0.0036, -0.1584,  0.0036],
        [-0.1584,  0.0036, -0.1584,  0.0036,  0.9940, -0.8991],
        [-0.1584,  0.0036,  0.9940, -0.8991,  0.0061, -0.6300],
        [ 0.9940, -0.8991,  0.0061, -0.6300, -0.5460,  3.0955],
        [ 0.0061, -0.6300, -0.5460,  3.0955, -1.6397, -0.6131],
        [-0.5460,  3.0955, -1.6397, -0.6131,  0.3087, -0.8025],
        [-1.6397, -0.6131,  0.3087, -0.8025,  0.4220,  0.3822],
        [ 0.3087, -0.8025,  0.4220,  0.3822, -0.3936,  0.4641],
        [-0.1584,  0.0036, -0.1584,  0.0036, -0.1584,  0.0036],
        [-0.1584,  0.0036, -0.1584,  0.0036, -0.5460,  3.0955],
        [-0.1584,  0.0036, -0.5460,  3.0955, -1.4187, -1.2333],
        [-0.5460,  3.0955, -1.4187, -1.2333, -0.3936,  0.4641],
        [-1.4187, -1.2333, -0.3936,  0.4641,  0.0970, -0.2923],
        [-0.3936,  0.4641,  0.0970, -0.2923,  0.9940, -0.8991],
        [ 0.0970, -0.2923,  0.9940, -0.8991,  0.3087, -0.8025],
        [ 0.9940, -0.8991,  0.3087, -0.8025,  0.3087, -0.8025],
        [ 0.3087, -0.8025,  0.3087, -0.8025, -1.4187, -1.2333],
        [ 0.3087, -0.8025, -1.4187, -1.2333,  0.3087, -0.8025],
```

```
        [-1.4187, -1.2333,  0.3087, -0.8025, -0.3936,  0.4641],
        [-0.1584,  0.0036, -0.1584,  0.0036, -0.1584,  0.0036],
        [-0.1584,  0.0036, -0.1584,  0.0036, -0.5780, -0.1284],
        [-0.1584,  0.0036, -0.5780, -0.1284, -1.4187, -1.2333],
        [-0.5780, -0.1284, -1.4187, -1.2333,  1.3726,  0.5412],
        [-1.4187, -1.2333,  1.3726,  0.5412,  1.2897,  0.9605],
        [ 1.3726,  0.5412,  1.2897,  0.9605,  0.0061, -0.6300],
        [ 1.2897,  0.9605,  0.0061, -0.6300, -0.5460,  3.0955],
        [ 0.0061, -0.6300, -0.5460,  3.0955,  0.9940, -0.8991],
        [-0.1584,  0.0036, -0.1584,  0.0036, -0.1584,  0.0036],
        [-0.1584,  0.0036, -0.1584,  0.0036, -1.6397, -0.6131],
        [-0.1584,  0.0036, -1.6397, -0.6131, -0.5780, -0.1284],
        [-1.6397, -0.6131, -0.5780, -0.1284,  0.0970, -0.2923],
        [-0.5780, -0.1284,  0.0970, -0.2923, -1.4187, -1.2333],
        [ 0.0970, -0.2923, -1.4187, -1.2333,  0.0061, -0.6300],
        [-1.4187, -1.2333,  0.0061, -0.6300,  0.9940, -0.8991],
        [-0.1584,  0.0036, -0.1584,  0.0036, -0.1584,  0.0036],
        [-0.1584,  0.0036, -0.1584,  0.0036, -1.3297, -0.9474],
        [-0.1584,  0.0036, -1.3297, -0.9474,  0.9940, -0.8991],
        [-1.3297, -0.9474,  0.9940, -0.8991,  0.0061, -0.6300],
        [ 0.9940, -0.8991,  0.0061, -0.6300, -1.6397, -0.6131],
        [ 0.0061, -0.6300, -1.6397, -0.6131,  0.3087, -0.8025],
        [-1.6397, -0.6131,  0.3087, -0.8025,  0.9940, -0.8991]])
```

Let's see a better way.

[459]:
```
# https://pytorch.org/docs/stable/generated/torch.Tensor.view.html

# https://pytorch.org/docs/stable/generated/torch.Tensor.stride.html

# use google image and discuss in class....

a = torch.arange(18)
print(a)
print(a.shape)
print(a.view(9, 2))
print(a.view(2, 9))
print(a.storage()) # very efficient in torch
```

```
tensor([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17])
torch.Size([18])
tensor([[ 0,  1],
        [ 2,  3],
        [ 4,  5],
        [ 6,  7],
        [ 8,  9],
        [10, 11],
        [12, 13],
```

```
        [14, 15],
        [16, 17]])
tensor([[ 0,  1,  2,  3,  4,  5,  6,  7,  8],
        [ 9, 10, 11, 12, 13, 14, 15, 16, 17]])
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
 10
 11
 12
 13
 14
 15
 16
 17
[torch.storage.TypedStorage(dtype=torch.int64, device=cpu) of size 18]
```

```
/tmp/ipykernel_494/1622729900.py:12: UserWarning: TypedStorage is deprecated. It
will be removed in the future and UntypedStorage will be the only storage class.
This should only matter to you if you are using storages directly.  To access
UntypedStorage directly, use tensor.untyped_storage() instead of
tensor.storage()
  print(a.storage()) # very efficient in torch
```

[460]:

```python
print(emb.view(41,6) == torch.cat(torch.unbind(emb,1),1))
```

```
tensor([[True, True, True, True, True, True],
        [True, True, True, True, True, True],
        [True, True, True, True, True, True],
        [True, True, True, True, True, True],
        [True, True, True, True, True, True],
        [True, True, True, True, True, True],
        [True, True, True, True, True, True],
        [True, True, True, True, True, True],
        [True, True, True, True, True, True],
        [True, True, True, True, True, True],
        [True, True, True, True, True, True],
        [True, True, True, True, True, True],
        [True, True, True, True, True, True],
        [True, True, True, True, True, True],
        [True, True, True, True, True, True],
```

```
[True, True, True, True, True, True],
[True, True, True, True, True, True],
[True, True, True, True, True, True],
[True, True, True, True, True, True],
[True, True, True, True, True, True],
[True, True, True, True, True, True],
[True, True, True, True, True, True],
[True, True, True, True, True, True],
[True, True, True, True, True, True],
[True, True, True, True, True, True],
[True, True, True, True, True, True],
[True, True, True, True, True, True],
[True, True, True, True, True, True],
[True, True, True, True, True, True],
[True, True, True, True, True, True],
[True, True, True, True, True, True],
[True, True, True, True, True, True],
[True, True, True, True, True, True],
[True, True, True, True, True, True],
[True, True, True, True, True, True],
[True, True, True, True, True, True],
[True, True, True, True, True, True],
[True, True, True, True, True, True],
[True, True, True, True, True, True],
[True, True, True, True, True, True],
[True, True, True, True, True, True]])
```

So we can use

[461]:
```
h = emb.view(41,6) @ W1 + b1
```

[462]:
```
print(h)
print(h.shape)
print(emb.view(-1,6) @ W1 + b1) # -1 means 'infer' the dimension from the other
 ↪dimensions (sort-of auto)
```

```
tensor([[-0.0127, -1.5797,  0.3777,  …,  0.8798,  0.8424,  0.3150],
        [ 0.4762, -1.6867,  2.4601,  …, -1.1835,  1.1385,  1.0527],
        [-2.4161, -1.6783,  2.7843,  …, -1.8970, -1.9951,  0.8419],
        …,
        [ 0.0383, -0.0525,  3.2570,  …,  2.5062, -3.9853, -0.9087],
        [ 1.8422, -0.6354,  2.8349,  …,  1.2162, -0.9053, -1.1034],
        [-0.7111, -4.3180,  3.8939,  …, -3.6111,  0.0440,  2.0282]])
torch.Size([41, 100])
tensor([[-0.0127, -1.5797,  0.3777,  …,  0.8798,  0.8424,  0.3150],
        [ 0.4762, -1.6867,  2.4601,  …, -1.1835,  1.1385,  1.0527],
        [-2.4161, -1.6783,  2.7843,  …, -1.8970, -1.9951,  0.8419],
        …,
```

```
        [ 0.0383, -0.0525,  3.2570,  …,  2.5062, -3.9853, -0.9087],
        [ 1.8422, -0.6354,  2.8349,  …,  1.2162, -0.9053, -1.1034],
        [-0.7111, -4.3180,  3.8939,  …, -3.6111,  0.0440,  2.0282]])
```

Embed (glue) + apply matrix + add b1. Now apply a non-linear transformation like hyperbolic tangent.

[463]:
```python
# first layer

# https://pytorch.org/docs/stable/generated/torch.tanh.html

h = torch.tanh(emb.view(-1,6) @ W1 + b1)
```

[464]:
```python
print(h)
```

```
tensor([[-0.0127, -0.9185,  0.3607,  …,  0.7063,  0.6871,  0.3050],
        [ 0.4432, -0.9337,  0.9855,  …, -0.8285,  0.8139,  0.7829],
        [-0.9842, -0.9326,  0.9924,  …, -0.9560, -0.9637,  0.6868],
        …,
        [ 0.0383, -0.0525,  0.9970,  …,  0.9868, -0.9993, -0.7205],
        [ 0.9510, -0.5618,  0.9931,  …,  0.8385, -0.7189, -0.8017],
        [-0.6113, -0.9996,  0.9992,  …, -0.9985,  0.0440,  0.9660]])
```

Second layer must take in 100D vector and give out a 28D vector.

[465]:
```python
# second layer

W2 = torch.randn((100,28))
b2 = torch.randn(28)
```

h is coming out from the first layer, then we feed with h the layer here.

[466]:
```python
logits = h @ W2 +b2
print(logits.shape)
```

```
torch.Size([41, 28])
```

Logits means log of the counting…

[467]:
```python
counts = logits.exp()
```

Normalize to interpret this as a measure, i.e. a probability distribution coming out from the network when fed with three chars.

[468]:
```python
prob = counts / counts.sum(1,keepdims=True)
```

[469]:
```python
print(prob[0])
print(prob[0].sum())
print(prob[0, 1])
print(prob[[0,1],[2,5]])
```

```
tensor([9.7717e-07, 1.1903e-09, 1.1622e-03, 1.3884e-04, 1.2119e-07, 1.4373e-08,
        3.7337e-08, 5.9759e-07, 4.1920e-10, 1.5534e-08, 9.1641e-01, 4.8512e-11,
        6.0606e-07, 1.0638e-03, 9.7434e-06, 3.2012e-06, 3.8229e-08, 1.5657e-07,
        3.3748e-03, 1.0385e-06, 2.4357e-05, 6.0491e-09, 2.8107e-14, 7.6021e-12,
        7.7808e-02, 1.6660e-08, 2.7311e-11, 9.1822e-10])
tensor(1.0000)
tensor(1.1903e-09)
tensor([1.1622e-03, 1.7094e-10])
```

Model is initialized with random weights, so it's making mistakes.

[470]:
```
print(Y)
print(torch.arange(41))
print(prob[torch.arange(41),Y])
```

```
tensor([ 2, 19,  8,  6, 15, 21, 16,  0,  8, 10, 16, 23,  2, 15, 15, 10, 15, 16,
         0, 13, 10,  4, 22, 19,  8,  2,  0,  6, 13, 23, 10, 19,  2,  0, 14,  2,
        19,  6, 15,  2,  0])
tensor([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17,
        18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,
        36, 37, 38, 39, 40])
tensor([1.1622e-03, 1.3647e-10, 1.2721e-10, 7.4825e-06, 4.3084e-15, 2.0606e-09,
        4.9061e-08, 5.3828e-07, 4.1920e-10, 7.6886e-08, 4.0766e-01, 1.1818e-09,
        2.8867e-08, 3.2134e-04, 5.7727e-05, 3.4768e-12, 1.0444e-11, 7.0611e-15,
        1.1958e-08, 1.0638e-03, 9.8603e-01, 5.7557e-09, 1.4726e-06, 7.5069e-07,
        5.9529e-17, 3.8195e-05, 4.3590e-09, 3.7337e-08, 1.0221e-06, 1.0162e-11,
        3.0114e-01, 4.2972e-04, 5.7576e-08, 1.1595e-06, 9.7434e-06, 1.8131e-09,
        6.9818e-13, 2.9324e-11, 1.8745e-15, 9.2900e-11, 6.4576e-09])
```

We, of course, want the model to predict the right answer. Probability going to one implies loss going to zero.

[471]:
```
loss = - prob[torch.arange(41),Y].log().mean()
print(loss)  # very bad of course.....
```

```
tensor(17.5833)
```

Let's put things together. Parameters will contain all the objects we're going to change. Why is the embedding dimension 2? We'll try with 10... $\tanh 0 = 0$ and that will be important.

[472]:
```
g = torch.Generator().manual_seed(123456780)# for reproducibility
C = torch.randn((28,2), generator=g)
W1 = torch.randn((6,100), generator=g)
b1 = torch.randn(100, generator=g)
W2 = torch.randn((100,28), generator=g)
b2 = torch.randn(28, generator=g)
parameters = [C,W1,b1,W2,b2]
```

How many parameters are fixable?

[473]:
```
print(sum(p.nelement() for p in parameters)) # number of parameter in total...
```

```
3584
```

For each sample I compute the log → high loss.

[474]:
```
emb = C[X] # torch.Size([41, 3, 2])
h = torch.tanh(emb.view(-1,6) @ W1 + b1) #(41,100)
logits = h @ W2 + b2 #(41,27)
counts = logits.exp()
prob = counts/counts.sum(1,keepdims=True)
loss = -prob[torch.arange(41),Y].log().mean()
print(loss)
```

```
tensor(17.2342)
```

Very efficient and can compute the exponential of big terms

[475]:
```
print(F.cross_entropy(logits,Y))
```

```
tensor(17.2342)
```

Now the loss has to be minimized.

[476]:
```
# so..... https://pytorch.org/docs/stable/generated/torch.nn.functional.
 ↪cross_entropy.html
emb = C[X] # torch.Size([41, 3, 2])
h = torch.tanh(emb.view(-1,6) @ W1 + b1) #(41,100)
logits = h @ W2 + b2 #(41,27)
loss = F.cross_entropy(logits,Y)
print(loss)
```

```
tensor(17.2342)
```

We'll use the cross_entropy function because the exponentiation of just -500 will result in 0

[477]:
```
# two very good reasons to use 'cross_entropy': more efficient (no tensor) and␣
 ↪substract the maximum to avoid nan....discuss....

logits = torch.tensor([-5,-3,0,10]) # -100
counts = logits.exp()
prob = counts/counts.sum()
print(counts)
print(prob)
```

```
tensor([6.7379e-03, 4.9787e-02, 1.0000e+00, 2.2026e+04])
tensor([3.0589e-07, 2.2602e-06, 4.5398e-05, 9.9995e-01])
```

Put the gradient to zero, then compute the backward derivative and update all parameters in order to decrease the loss. 41 trigrams are going in layers, then calculate the loss.

Backward pass means compute the derivative of the loss for each parameter. It's a very complex stuff. We're not happy with back propagation but, by now, it's the only thing which works.

Learning rate -0.1 (negative direction). This is a magic number.

[478]:
```
for p in parameters:
    p.requires_grad = True

for _ in range(1000):
    # now we learn...forward bass
    emb = C[X] # torch.Size([41, 3, 2])
    h = torch.tanh(emb.view(-1,6) @ W1 + b1) #(41,100)
    logits = h @ W2 + b2 #(41,27)
    loss = F.cross_entropy(logits,Y)
    # backward pass
    for p in parameters:
        p.grad = None
    loss.backward()
    # update
    for p in parameters:
        p.data += -0.1*p.grad
```

Low loss means overfitting, then the model is going to give me one of the sample I provided. This is not useful at all.

[479]:
```
# sampling from the model.....

g = torch.Generator().manual_seed(12345678+10)

for _ in range(20):
    out = []
    context = [0]*block_size
    while True:
        emb = C[torch.tensor([context])]
        h = torch.tanh(emb.view(1,-1) @ W1 + b1)
        logits = h @ W2 + b2
        probs = F.softmax(logits,dim=1)
        ix = torch.multinomial(probs,num_samples=1,generator=g).item()
        context = context[1:]+[ix]
        out.append(ix)
        if ix==0:
            break

    print(''.join(itos[i] for i in out))
```

```
elvira.
marena.
giovannino.
```

```
giovannino.
giovannino.
giovannino.
elvira.
marena.
argento.
argento.
argento.
elvira.
argento.
licurga.
licurga.
marena.
giovannino.
marena.
giovannino.
licurga.
```

Model is overfitting: we've given it only 5 samples out of 7000+... Actually we've 41 examples (trigrams) and 3584 parameters.

[480]:
```
for p in parameters:
    p.requires_grad = True

for _ in range(1000):
    # now we learn...forward bass
    emb = C[X] # torch.Size([41, 3, 2])
    h = torch.tanh(emb.view(-1,6) @ W1 + b1) #(41,100)
    logits = h @ W2 + b2 #(41,27)
    loss = F.cross_entropy(logits,Y)
    #print(loss.item())
    # backward pass
    for p in parameters:
        p.grad = None
    loss.backward()
    #update
    for p in parameters:
        p.data += -0.1*p.grad
print(loss.item())
```

```
0.19916315376758575
```

Logits are the neurons coming out from the last layer. They'll be transformed into probability.

[481]:
```
print(logits.max(1))
print(Y)
```

```
torch.return_types.max(
values=tensor([12.1724, 14.1847, 15.8096, 16.5460, 12.5090, 14.7062, 15.2590,
```

```
15.4024,
        12.1724, 19.3660, 15.9273, 12.4720, 13.9841, 17.1356, 14.9285, 13.9812,
        13.8305, 15.1462, 14.5532, 12.1724, 15.2042, 17.6397, 14.6976, 14.7785,
        22.7934, 18.4190, 15.3254, 12.1724, 18.8153, 16.9008, 17.9443, 18.5473,
        15.9427, 15.7163, 12.1724, 19.9722, 15.4614, 16.7053, 16.1372, 18.4218,
        15.9050], grad_fn=<MaxBackward0>),
indices=tensor([13, 19,  8,  6, 15, 21, 16,  0, 13, 10, 16, 23,  2, 15, 15, 10,
15, 16,
         0, 13, 10,  4, 22, 19,  8,  2,  0, 13, 13, 23, 10, 19,  2,  0, 13,  2,
        19,  6, 15,  2,  0]))
tensor([ 2, 19,  8,  6, 15, 21, 16,  0,  8, 10, 16, 23,  2, 15, 15, 10, 15, 16,
         0, 13, 10,  4, 22, 19,  8,  2,  0,  6, 13, 23, 10, 19,  2,  0, 14,  2,
        19,  6, 15,  2,  0])
```

Taking all words we get a very big example dataset.

[482]:
```
block_size = 3 #context length: how many characters do we take to predict the␣
↪next one ... change it !!
X, Y = [], []  # input & label

for w in words:
  #  print(w)
    context = [0]*block_size
    for ch in w +'.':
        ix = stoi[ch]
        X.append(context)
        Y.append(ix)
      #  print(''.join(itos[i] for i in context), '--->', itos[ix])
        context = context[1:]+[ix]  # shift: crop and append
X=torch.tensor(X)
Y=torch.tensor(Y)
```

[483]:
```
print(X.shape, Y.shape)
```

```
torch.Size([73643, 3]) torch.Size([73643])
```

Again, a hidden layer of 100 neurons.

[484]:
```
# exactly as before....
g = torch.Generator().manual_seed(123456780)# for reproducibility
C = torch.randn((28,2), generator=g)
W1 = torch.randn((6,100), generator=g)
b1 = torch.randn(100, generator=g)
W2 = torch.randn((100,28), generator=g)
b2 = torch.randn(28, generator=g)
parameters = [C,W1,b1,W2,b2]
```

[485]:
```
for p in parameters:
    p.requires_grad = True

for _ in range(10):
    # now we learn...forward bass -- = 73643
    emb = C[X] # torch.Size([--, 3, 2])
    h = torch.tanh(emb.view(-1,6) @ W1 + b1) #(--,100)
    logits = h @ W2 + b2 #(--,27)
    loss = F.cross_entropy(logits,Y)
    print(loss.item())
    # backward pass
    for p in parameters:
        p.grad = None
    loss.backward()
    #update
    for p in parameters:
        p.data += -0.1*p.grad
```

```
17.754497528076172
15.824398040771484
14.187443733215332
13.052313804626465
12.071905136108398
11.263842582702637
10.603142738342285
10.075033187866211
9.627379417419434
9.222493171691895
```

See how it's slowing down... Every time we give all samples to it. Let's subdivide the dataset in batches.

[486]:
```
ix = torch.randint(0, X.shape[0], (10,)) #try ix=torch.randint(0,X.
 ↪shape[0],(10,2)) and explain
#https://pytorch.org/docs/stable/generated/torch.randint.html
print(ix)
```

```
tensor([65739, 57246, 73039, 50426, 60229,  1076, 10994, 71617, 42058, 29497])
```

Weird things happen with PyTorch...

[487]:
```
ix = torch.randint(0, X.shape[0], (10,1))
print(ix)
```

```
tensor([[57686],
        [ 8590],
        [18991],
        [31694],
        [56747],
```

```
        [36820],
        [19621],
        [15325],
        [65718],
        [48395]])
```

[488]:
```
for _ in range(10):
    # mini batch construct of size ...
    ix = torch.randint(0,X.shape[0],(32,))
    # now we learn...forward bass -- = 73643
    emb = C[X[ix]] # torch.Size([--, 3, 2])
    h = torch.tanh(emb.view(-1,6) @ W1 + b1) #(--,100)
    logits = h @ W2 + b2 #(--,27)
    loss = F.cross_entropy(logits,Y[ix])
    print(loss.item())
    # backward pass
    for p in parameters:
        p.grad = None
    loss.backward()
    #update
    for p in parameters:
        p.data += -0.1*p.grad
print(loss.item())
```

```
9.076169967651367
8.107839584350586
7.53472900390625
8.057098388671875
9.006769180297852
6.839514255523682
6.016563415527344
7.071707725524902
6.523996353149414
6.122567653656006
6.122567653656006
```

Learning rate specifies how I move through gradient. Using 10, the system got completely lost (too big jumps).

[489]:
```
# how we define the 'learning rate' ? p.data += -0.1*p.grad
# play with learning rate from .01 to 100.... and discuss
```

[490]:
```
for _ in range(1000):
    # mini batch construct
    ix = torch.randint(0,X.shape[0],(100,))
    # now we learn...forward bass -- = 73643
    emb = C[X[ix]] # torch.Size([--, 3, 2])
    h = torch.tanh(emb.view(-1,6) @ W1 + b1) #(--,100)
```

```
    logits = h @ W2 + b2 #(--,27)
    loss = F.cross_entropy(logits,Y[ix])
    # print(loss.item())
    # backward pass
    for p in parameters:
        p.grad = None
    loss.backward()
    #update
    for p in parameters:
        p.data += -.1*p.grad
print(loss.item())
```

2.3013787269592285

[491]:
```
lre = torch.linspace(-3,0,1000)
lrs = 10**lre # from 10**-3 to 10**0 = 1. The exponents are linearly␣
 ↪distributed, not the values
print(lrs.shape)
```

torch.Size([1000])

[492]:
```
for p in parameters:
    p.requires_grad = True

lri = []
lriex = []
lossi = []


for i in range(1000):
    # mini batch construct
    ix = torch.randint(0,X.shape[0],(100,))
    # now we learn...forward bass -- = 73643
    emb = C[X[ix]] # torch.Size([--, 3, 2])
    h = torch.tanh(emb.view(-1,6) @ W1 + b1) #(--,100)
    logits = h @ W2 + b2 #(--,27)
    loss = F.cross_entropy(logits,Y[ix])
    # backward pass
    for p in parameters:
        p.grad = None
    loss.backward()
    #update
    lr = lrs[i]
    # lr= .01
    for p in parameters:
        p.data += -lr*p.grad
```

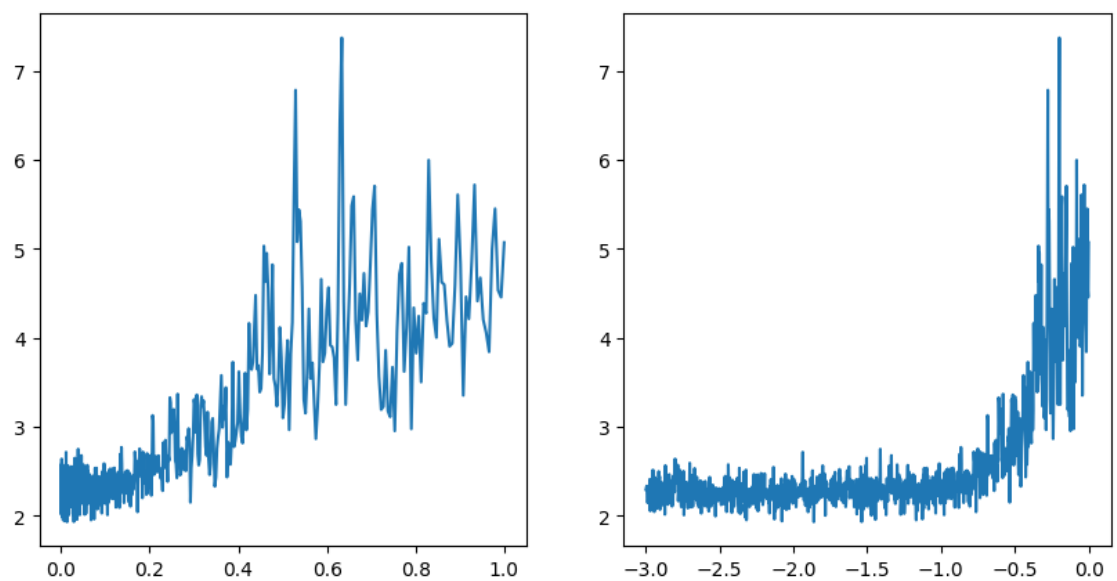
```
#track stats
    lri.append(lr) # learning rate
    lriex.append(lre[i]) # exponent
    lossi.append(loss.item()) # loss function
print(loss.item())
```

```
5.072561264038086
```

Plotting the loss function we notice that it's growing… not good.

[493]:
```
fig, ax = plt.subplots(1,2, figsize=(10,5))

ax[0].plot(lri, lossi)
ax[1].plot(lriex, lossi)
```

[493]:
```
[<matplotlib.lines.Line2D at 0x7f0c8eb10520>]
```



How to validate that the network is doing something good? It may seem good while being completely wrong. We should have a test set of data to use when the hyperparameters are fixed.

[494]:
```
emb = C[X] # torch.Size([41, 3, 2])
h = torch.tanh(emb.view(-1,6) @ W1 + b1) # (41,100)
logits = h @ W2 + b2 # (41,27)
loss = F.cross_entropy(logits,Y)
print(loss)
```

```
tensor(4.5317, grad_fn=<NllLossBackward0>)
```

Typically, data are divided into 80-10-10 part (test/tune/validation) This function shuffles the words and builds the three wanted datasets.

[495]:
```
# be careful with the test eugene.....

def build_dataset(words):
    block_size = 3 # context length: how many characters do we take to predict
 ↪the next one ... change it !!
    X, Y = [], []  # input & label


    for w in words:
        context = [0]*block_size
        for ch in w +'.':
            ix = stoi[ch]
            X.append(context)
            Y.append(ix)
      #  print(''.join(itos[i] for i in context), '--->', itos[ix])
            context=context[1:]+[ix]  # shift: crop and append
    X = torch.tensor(X)
    Y = torch.tensor(Y)
    print(X.shape, Y.shape)
    return X, Y
```

[496]:
```
import random
random.seed(42)
random.shuffle(words)
n1 = int(0.8*len(words))
n2 = int(0.9*len(words))

Xtr, Ytr = build_dataset(words[:n1]) # train
Xdev, Ydev = build_dataset(words[n1:n2]) # tune hyperparameters
Xte, Yte = build_dataset(words[n2:]) # validate
```

```
torch.Size([58867, 3]) torch.Size([58867])
torch.Size([7404, 3]) torch.Size([7404])
torch.Size([7372, 3]) torch.Size([7372])
```

[497]:
```
# and we do it again with the new datasets.......
print(Xtr.shape, Ytr.shape)

# exactly as before....
g = torch.Generator().manual_seed(123456780) # for reproducibility
C = torch.randn((28,2), generator=g)
W1 = torch.randn((6,100), generator=g)
b1 = torch.randn(100, generator=g)
W2 = torch.randn((100,28), generator=g)
b2 = torch.randn(28, generator=g)
parameters = [C,W1,b1,W2,b2]
```

```
torch.Size([58867, 3]) torch.Size([58867])
```

[498]:
```
for p in parameters:
    p.requires_grad = True


lre = torch.linspace(-3,0,1000)
lrs = 10**lre
```

[499]:
```
# now we train only on Xtr

lri = []
lriex = []
lossi = []


for i in range(10000):
    # mini batch construct
    ix = torch.randint(0,Xtr.shape[0],(40,))
    # now we learn...forward bass -- = 73643
    emb = C[Xtr[ix]] # torch.Size([--, 3, 2])
    h = torch.tanh(emb.view(-1,6) @ W1 + b1) #(--,100)
    logits = h @ W2 + b2 #(--,27)
    loss = F.cross_entropy(logits,Ytr[ix])
   # print(i,loss.item())
    # backward pass
    for p in parameters:
        p.grad = None
    loss.backward()
    #update
    #lr=lrs[i]
    lr= .1
    for p in parameters:
        p.data += -lr*p.grad
print(loss.item())

#track stats
#    lri.append(lr)
#    lriex.append(lre[i])
#    lossi.append(loss.item())
```

```
2.018704652786255
```

Now evaluate on the validation test (and also on the test).

[500]:
```
# now we evaluate on Xdev
emb = C[Xdev]
h = torch.tanh(emb.view(-1,6) @ W1 + b1) #(--,100)
```

```
logits = h @ W2 + b2 #(--,27)
loss = F.cross_entropy(logits,Ydev)
print(loss.item())
```

2.1819050312042236

[501]:
```
# now we evaluate on Xtr..... we are NOT overfitting
emb = C[Xtr]
h = torch.tanh(emb.view(-1,6) @ W1 + b1) #(--,100)
logits = h @ W2 + b2 #(--,27)
loss = F.cross_entropy(logits,Ytr)
print(loss.item())
```

2.172853708267212

Now change the hyperparameters... The net is small so nothing is going to change.

[502]:
```
g = torch.Generator().manual_seed(123456780)# for reproducibility
C = torch.randn((28,2), generator=g)
W1 = torch.randn((6,300), generator=g)
b1 = torch.randn(300, generator=g)
W2 = torch.randn((300,28), generator=g)
b2 = torch.randn(28, generator=g)
parameters = [C,W1,b1,W2,b2]
```

[503]:
```
print(sum(p.nelement() for p in parameters)) # number of parameter in total...
 ↪before 3584
```

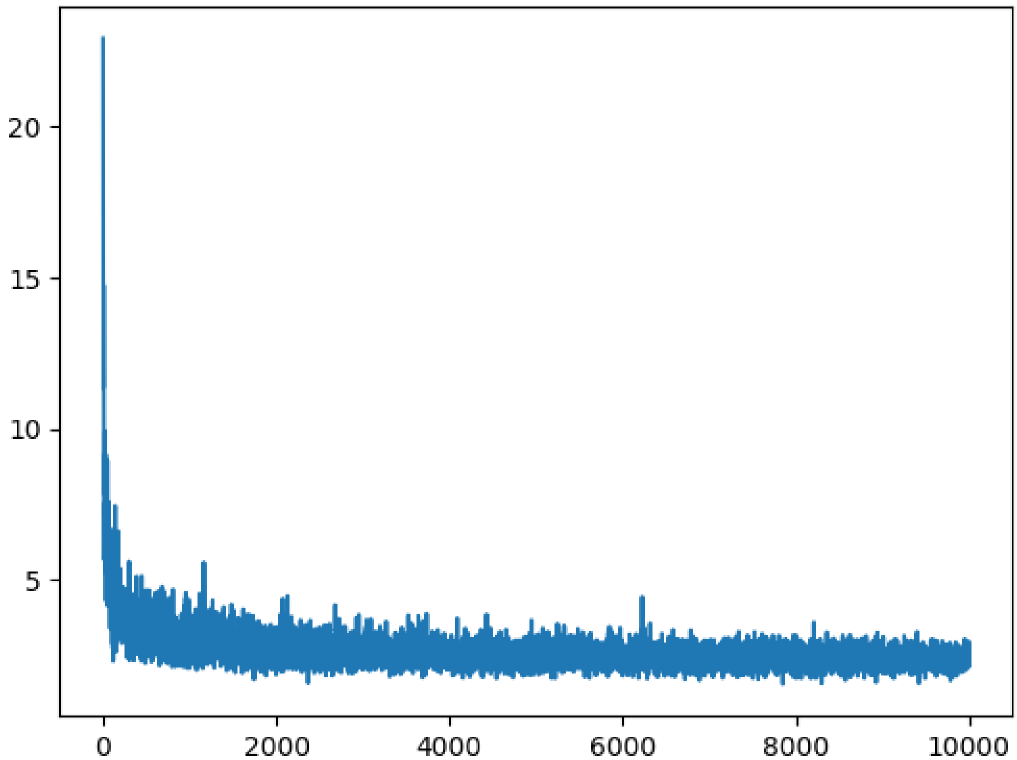
10584

[504]:
```
lri = []
lriex = []
lossi = []
stepi = []
for p in parameters:
    p.requires_grad = True

for i in range(10000):
    # mini batch construct
    ix = torch.randint(0,Xtr.shape[0],(40,))
    # now we learn...forward bass -- = 73643
    emb = C[Xtr[ix]] # torch.Size([--, 3, 2])
    h = torch.tanh(emb.view(-1,6) @ W1 + b1) #(--,100)
    logits = h @ W2 + b2 #(--,27)
    loss = F.cross_entropy(logits,Ytr[ix])
    # backward pass
    for p in parameters:
        p.grad = None
```

```
    loss.backward()
    #update
    #lr=lrs[i]
    lr= .1
    for p in parameters:
        p.data += -lr*p.grad
    stepi.append(i)
    lossi.append(loss.item())
```

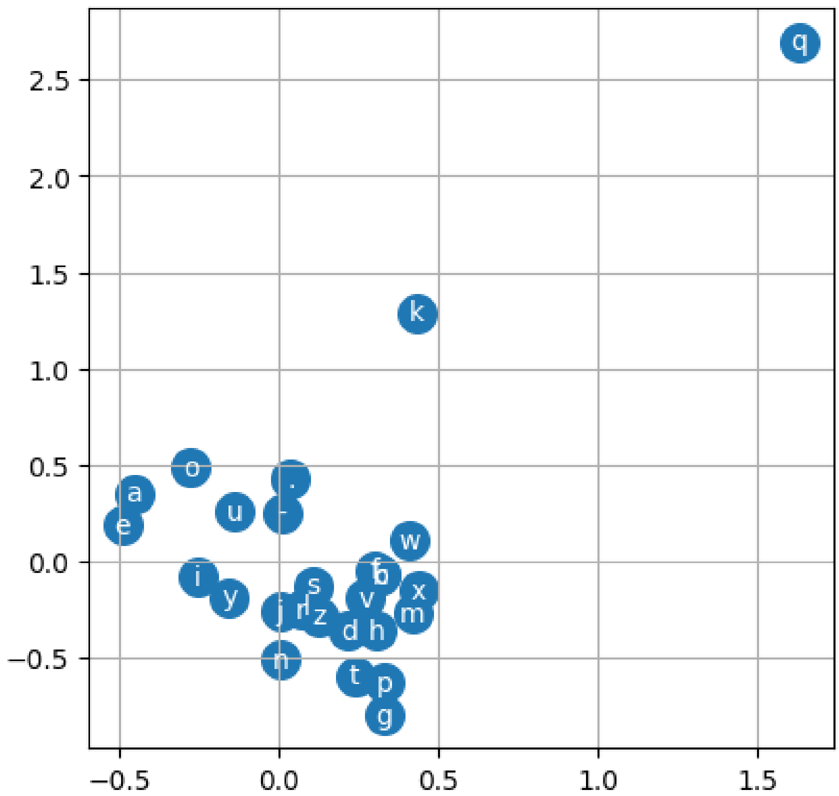
[505]:
```
plt.plot(stepi,lossi)
```

[505]: [<matplotlib.lines.Line2D at 0x7f0cb9541de0>]



Now we can visualize the result of the embedding. The letters are clustered, e.g. all vocals are on the left.

[506]:
```
plt.figure(figsize=(5, 5))
plt.scatter(C[:,0].data, C[:,1].data, s=200)
for i in range(C.shape[0]):
    plt.text(C[i,0].item(), C[i,1].item(),itos[i], ha="center",va="center",␣
 ↪color="white")
    plt.grid('minor')
```



Now increase to 10 the embedding dimension...

[507]:
```
g = torch.Generator().manual_seed(123456780)# for reproducibility
C = torch.randn((28,10), generator=g)
W1 = torch.randn((30,200), generator=g)
b1 = torch.randn(200, generator=g)
W2 = torch.randn((200,28), generator=g)
b2 = torch.randn(28, generator=g)
parameters = [C,W1,b1,W2,b2]
```

[508]:
```
print(sum(p.nelement() for p in parameters)) # number of parameter in total
```

```
12108
```

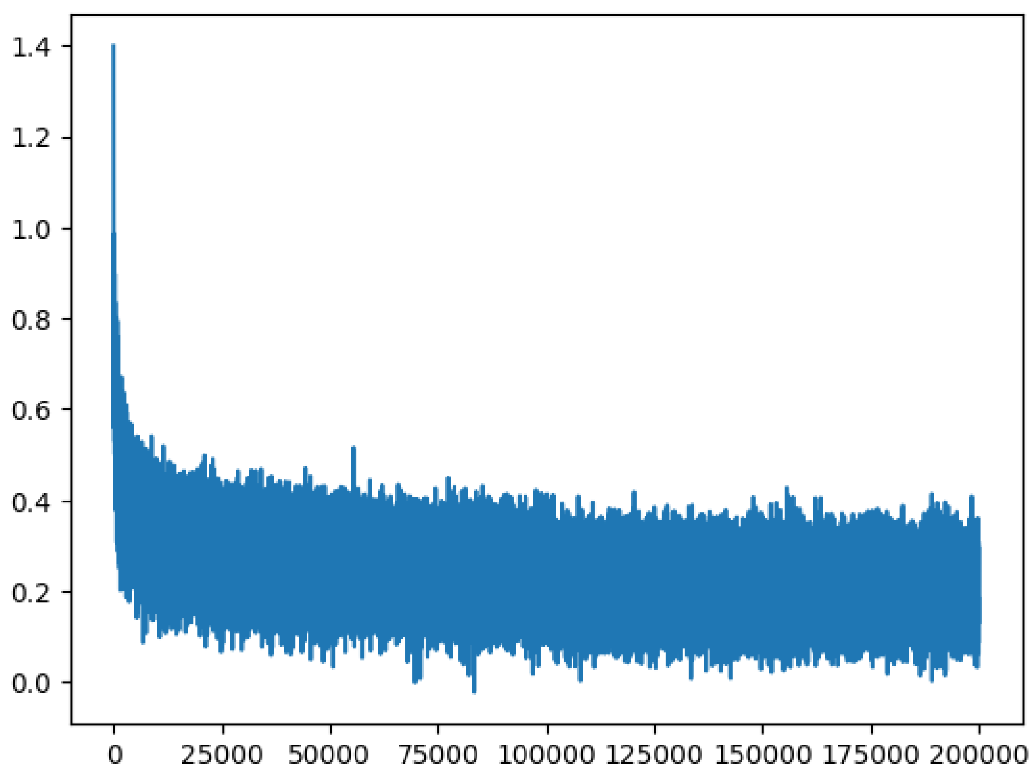
[509]:
```
lri = []
lriex = []
lossi = []
stepi = []

for p in parameters:
    p.requires_grad = True
```

Now change the learning rate...

[510]:
```
for i in range(200000):

    # mini batch construct
    ix = torch.randint(0,Xtr.shape[0], (40,))
    # now we learn...forward bass -- = 73643
    emb = C[Xtr[ix]] # torch.Size([--, 3, 2])
    h = torch.tanh(emb.view(-1,30) @ W1 + b1) #(--,100)
    logits = h @ W2 + b2 #(--,27)
    loss = F.cross_entropy(logits, Ytr[ix])
    # backward pass
    for p in parameters:
        p.grad=None
    loss.backward()
    #update
    lr= .1 if i < 100000 else 0.01
    for p in parameters:
        p.data += -lr*p.grad
    stepi.append(i)
    lossi.append(loss.log10().item()) # note the log10 !!
```

[511]:
```
plt.plot(stepi, lossi)
```

[511]: [<matplotlib.lines.Line2D at 0x7f0cbbff3b50>]



We're doing well and not overfitting.

[512]:
```
emb = C[Xtr]
h = torch.tanh(emb.view(-1,30) @ W1 + b1) #(--,100)   30 not 6 !
logits = h @ W2 + b2 #(--,27)
loss = F.cross_entropy(logits, Ytr)
print(loss.item())
```

```
1.6529531478881836
```

[513]:
```
emb = C[Xdev]
h = torch.tanh(emb.view(-1,30) @ W1 + b1) #(--,100)
logits = h @ W2 + b2 #(--,27)
loss = F.cross_entropy(logits, Ydev)
print(loss.item())
```

```
1.8500275611877441
```

Once trained the model we can sample from it. **NOTE**: there are many (many many) hyperparameters to play with, like the number of layer, numbers of neurons from layers, embedding dimensions, dimension of the batches, learning rate....

We can now see words that are not in the dataset.

[514]:
```
# g = torch.Generator().manual_seed(12345678+10)

for _ in range(30):
    out = []
    context = [0]*block_size
    while True:
        emb = C[torch.tensor([context])]
        h = torch.tanh(emb.view(1,-1) @ W1 + b1)
        logits = h @ W2 + b2
        probs = F.softmax(logits,dim=1)
        ix = torch.multinomial(probs, num_samples=1, generator=g).item()
        context = context[1:]+[ix]
        out.append(ix)
        if ix == 0:
            break

    print(''.join(itos[i] for i in out))
```

```
ade.
galdina.
crescenzu.
erto.
marinanto.
```

```
divia.
deciano.
ermiro.
dircolo.
arleolomurita.
gusina.
ristofelastrrino.
chrica.
mea.
pina.
pedermen.
ave.
alimolando.
gertaudino.
speramina.
orlino.
orlide.
venzion-idalma.
is.
asdina.
clemiglia.
zelminio.
brunino.
esio.
doluttunaa.
```

3 Concatenated Network - MakeMore pt.3 (21/04/2023)

[2]:
```
# https://youtu.be/P6sfmUTpUmc
# https://github.com/karpathy/makemore
```

[3]:
```
# we now want to dig more into neural activity and learning to understand the
↪RNN and LSTM architecture and properties...

import random
import torch
import torch.nn.functional as  F
import matplotlib.pyplot as plt #for making figures
%matplotlib inline
```

[5]:
```
# read in all the words
random.seed(158)
words = open("data/nomi_italiani.txt", "r").read().splitlines()
random.shuffle(words)
words[0:8]
```

[5]:
```
['argento',
 'giovannino',
 'licurga',
 'elvira',
 'marena',
 'sirio',
 'emilia',
 'bisio']
```

[6]:
```
print(len(words))
```

```
9105
```

[7]:
```
# build the vocabulary of characters and mapping to/from integers
chars = sorted(list(set("".join(words))))

stoi = {s: i + 1 for i, s in enumerate(chars)}
stoi["."] = 0
itos = {i: s for s, i in stoi.items()}
# new
vocab_size = len(itos)
print(itos)
print(vocab_size)
```

```
{1: '-', 2: 'a', 3: 'b', 4: 'c', 5: 'd', 6: 'e', 7: 'f', 8: 'g', 9: 'h', 10:
'i', 11: 'j', 12: 'k', 13: 'l', 14: 'm', 15: 'n', 16: 'o', 17: 'p', 18: 'q', 19:
'r', 20: 's', 21: 't', 22: 'u', 23: 'v', 24: 'w', 25: 'x', 26: 'y', 27: 'z', 0:
'.'}
28
```

[8]:
```
# build the dataset

block_size = (
    3  # context length: how many characters do we take to predict the next one␣
 ↪...
)


def build_dataset(words):
    X, Y = [], []  # input & label

    for w in words:
        context = [0] * block_size
        for ch in w + ".":
            ix = stoi[ch]
            X.append(context)
            Y.append(ix)
```

```python
            #  print(''.join(itos[i] for i in context), '--->', itos[ix])
            context = context[1:] + [ix]  # shift: crop and append
    X = torch.tensor(X)
    Y = torch.tensor(Y)
    print(X.shape, Y.shape)
    return X, Y
```

[9]:
```python
import random

random.seed(42)
random.shuffle(words)
n1 = int(0.8 * len(words))
n2 = int(0.9 * len(words))

Xtr, Ytr = build_dataset(words[:n1])
Xdev, Ydev = build_dataset(words[n1:n2])
Xte, Yte = build_dataset(words[n2:])
```

```
torch.Size([58867, 3]) torch.Size([58867])
torch.Size([7404, 3]) torch.Size([7404])
torch.Size([7372, 3]) torch.Size([7372])
```

[10]:
```python
# MLP revisited
n_embd = 10  # the dimensionality of the character embedding vectors
n_hidden = 200  # the number of neurons in the hidden layer of MLP

g = torch.Generator().manual_seed(123456780)  # for reproducibility
C = torch.randn((vocab_size, n_embd), generator=g)
W1 = torch.randn((n_embd * block_size, n_hidden), generator=g)
b1 = torch.randn(n_hidden, generator=g)
W2 = torch.randn((n_hidden, vocab_size), generator=g)
b2 = torch.randn(vocab_size, generator=g)
parameters = [C, W1, b1, W2, b2]
print(sum(p.nelement() for p in parameters))  # number of parameter in total...
for p in parameters:
    p.requires_grad = True
```

```
12108
```

[11]:
```python
# same optimization as last time
max_steps = 200000
batch_size = 32
lossi = []

for i in range(max_steps):
    # mini batch construct
    ix = torch.randint(0, Xtr.shape[0], (batch_size,), generator=g)
```

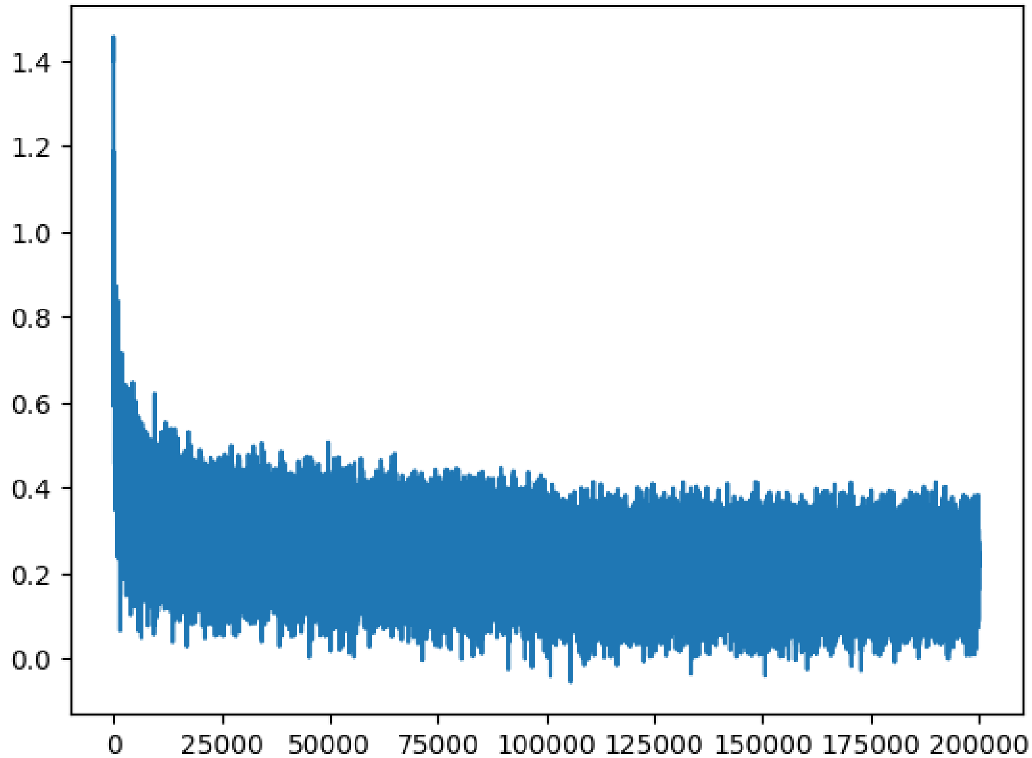
```python
    Xb, Yb = Xtr[ix], Ytr[ix]  # batch X,Y

    # forward bass
    emb = C[Xb]  # embed characters into vectors
    embcat = emb.view(emb.shape[0], -1)  # concatenate the vectors
    hpreact = embcat @ W1 + b1  # hidden layer pre-activation
    h = torch.tanh(hpreact)  # hidden layer
    logits = h @ W2 + b2  # output layer
    loss = F.cross_entropy(logits, Yb)  # loss function

    # backward pass
    for p in parameters:
        p.grad = None
    loss.backward()

    # update
    lr = 0.1 if i < 100000 else 0.01  # step learning rate decay
    for p in parameters:
        p.data += -lr * p.grad

    # track stats

    if i % 10000 == 0:  # print every once in a while
        print(f"{i:7d}/{max_steps:7d}:{loss.item():.4f}")
    lossi.append(loss.log10().item())
```

```
      0/ 200000:25.2234
  10000/ 200000:2.2865
  20000/ 200000:2.1045
  30000/ 200000:2.1319
  40000/ 200000:1.8164
  50000/ 200000:1.6172
  60000/ 200000:1.8491
  70000/ 200000:1.9464
  80000/ 200000:1.9799
  90000/ 200000:2.3608
 100000/ 200000:1.3078
 110000/ 200000:1.9567
 120000/ 200000:1.5188
 130000/ 200000:1.7088
 140000/ 200000:1.6152
 150000/ 200000:2.0892
 160000/ 200000:1.2948
 170000/ 200000:1.3657
 180000/ 200000:1.7789
 190000/ 200000:1.5764
```

[12]:
```python
print(-torch.tensor(1 / 28).log())
```

```
tensor(3.3322)
```

[14]:
```python
plt.plot(lossi)
```



[15]:
```python
@torch.no_grad()  # this decorator disables gradient tracking....discuss in␣
↪class....
def split_loss(split):
    x, y = {
        "train": (Xtr, Ytr),
        "val": (Xdev, Ydev),
        "test": (Xte, Yte),
    }[split]
    emb = C[x]  # (N,block_size, n_embd)
    embcat = emb.view(emb.shape[0], -1)  # concat into (N,block_size*n_embd)
    hpreact = embcat @ W1 + b1  # hidden layer pre-activation
    h = torch.tanh(hpreact)  # hidden layer (N, h_hidden)
    logits = h @ W2 + b2  # output layer (N, vocab_size)
    loss = F.cross_entropy(logits, y)  # loss function
    print(split, loss.item())
```

[16]:
```
split_loss("train")
split_loss("val")
```

```
train 1.6465035676956177
val 1.8481979370117188
```

[17]:
```
# sampling from the model.....

g = torch.Generator().manual_seed(12345678 + 10)

for _ in range(20):
    out = []
    context = [0] * block_size
    while True:
        emb = C[torch.tensor([context])]  # (1,block_size,n_embed)
        h = torch.tanh(emb.view(1, -1) @ W1 + b1)
        logits = h @ W2 + b2
        probs = F.softmax(logits, dim=1)
        # sample from the distribuion
        ix = torch.multinomial(probs, num_samples=1, generator=g).item()
        # shift the context window and track the samples
        context = context[1:] + [ix]
        out.append(ix)
        # if we sample the special '.' token, break
        if ix == 0:
            break

    print("".join(itos[i] for i in out))  # decode and print the generated word
```

```
albo.
giovanno.
rizio.
siside.
polina.
gio.
assimo.
cecchiarosinda.
benuartinaippirenziana.
peppio.
abdocchia.
bella.
benio.
moheo.
lauretilla.
rinieronino.
filosca.
esaro.
euto.
```

giliana.

[21]:
```
# let us focus on the last layer ....logits and then softmax

logits = torch.randn(4) * 100
# logits = view(2, 2, 2, 20)
probs = torch.softmax(logits, dim=0)
loss = -probs[2].log()
print("logits:", logits)
print("probs:", probs)
print("loss:", loss)
```

```
logits: tensor([-14.4960, -31.7004, 124.3944, 179.8049])
probs: tensor([0.0000e+00, 0.0000e+00, 8.6204e-25, 1.0000e+00])
loss: tensor(55.4105)
```

[22]:
```
# back to our examples and look at the logit just after the first pass and␣
↪understand normalization....



# MLP revisited
n_embd = 10  # the dimensionality of the character embedding vectors
n_hidden = 200  # the number of neurons in the hidden layer of MLP

g = torch.Generator().manual_seed(123456780)  # for reproducibility
C = torch.randn((vocab_size, n_embd), generator=g)
W1 = torch.randn((n_embd * block_size, n_hidden), generator=g)  # *0.20
b1 = torch.randn(n_hidden, generator=g)  # *0.01
W2 = torch.randn((n_hidden, vocab_size), generator=g)  # *0.01
b2 = torch.randn(vocab_size, generator=g)  # *0
parameters = [C, W1, b1, W2, b2]
print(sum(p.nelement() for p in parameters))  # number of parameter in total...
for p in parameters:
    p.requires_grad = True
```

```
12108
```

[23]:
```
# same optimization as last time...try, look at logits then go up and change W2␣
↪and b2 normalization.....
max_steps = 200000
batch_size = 32
lossi = []

for i in range(max_steps):
    # mini batch construct
    ix = torch.randint(0, Xtr.shape[0], (batch_size,), generator=g)
    Xb, Yb = Xtr[ix], Ytr[ix]  # batch X,Y
```

```python
    # forward bass
    emb = C[Xb]  # embed characters into vectors
    embcat = emb.view(emb.shape[0], -1)  # concatenate the vectors
    hpreact = embcat @ W1 + b1  # hidden layer pre-activation
    h = torch.tanh(hpreact)  # hidden layer
    logits = h @ W2 + b2  # output layer
    loss = F.cross_entropy(logits, Yb)  # loss function

    # backward pass
    for p in parameters:
        p.grad = None
    loss.backward()

    # update
    lr = 0.1 if i < 100000 else 0.01  # step learning rate decay
    for p in parameters:
        p.data += -lr * p.grad

    # track stats

    if i % 10000 == 0:  # print every once in a while
        print(f"{i:7d}/{max_steps:7d}:{loss.item():.4f}")
    lossi.append(loss.log10().item())
    break
```

```
      0/ 200000:25.2234
```

[26]:
```python
print(
    logits[1]
)  # confidently wrong......but with weigth is better....'squashing down the
 ↪neurons....'
```

```
tensor([ 10.6530,  -7.2423,  31.2924, -15.8263,   7.0554,  -0.8957,  10.4593,
          8.9841,  13.3143,  -2.6116,  14.7497,  17.5647,  17.0676,   1.1002,
         14.7031, -13.5323, -19.1125, -21.7921,  21.1479,   8.6535,   7.4713,
         -3.2913,   3.2052,   1.8503,  12.2664,  -3.8606,   3.1228,   4.6715],
       grad_fn=<SelectBackward0>)
```

[ ]:
```python
# Exercise: try the followi ng normalization, train , evaluate and sample the
 ↪MLP
```

[29]:
```python
# Now we focus on the first layer (show pict): h & hpreact

# remember to intialize and run again

# look at the +- 1 in h
```

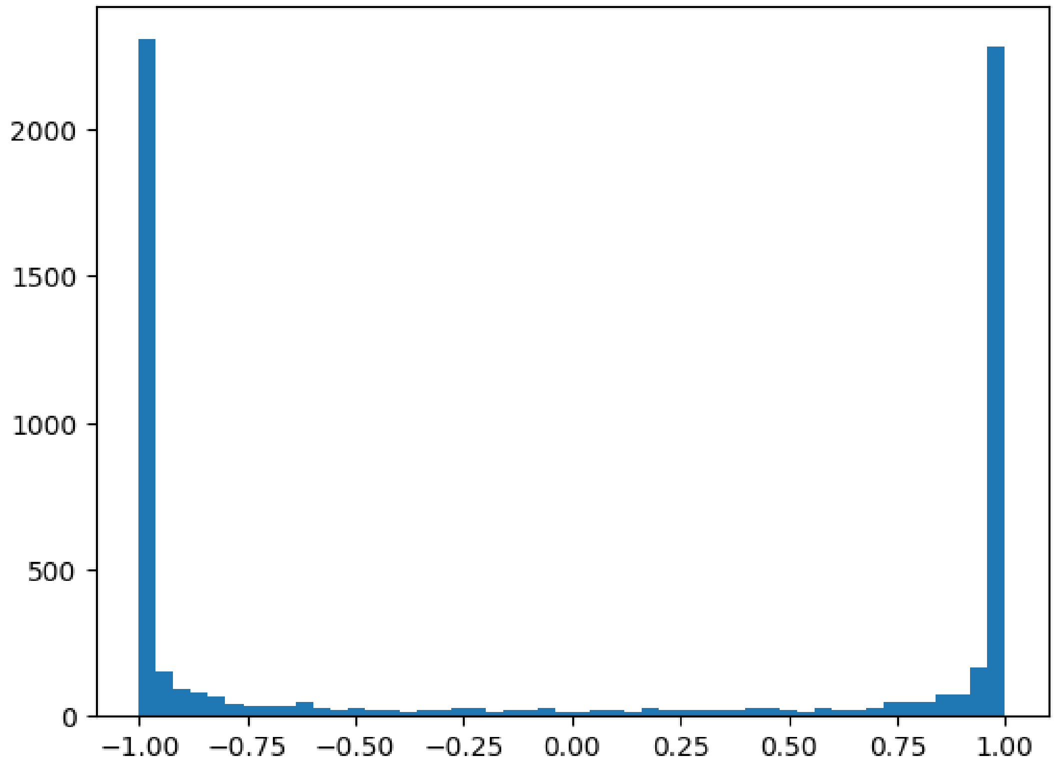
```
print(h.shape)
```

torch.Size([32, 200])

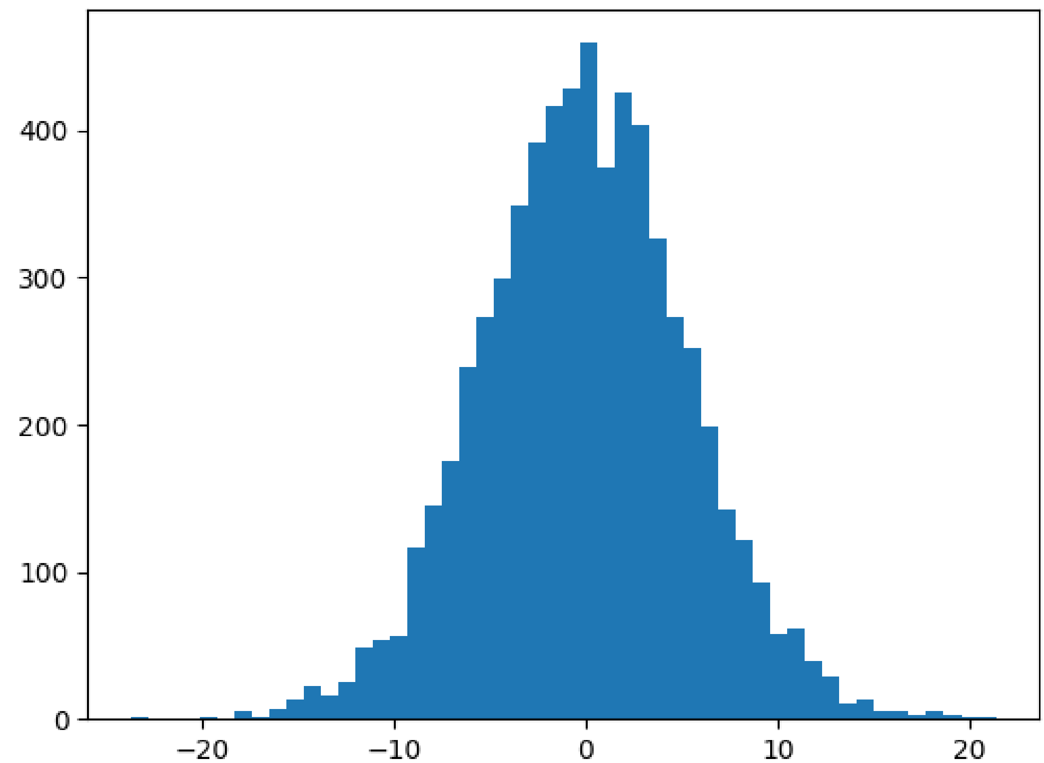
[28]:
```
print(len(h.view(-1).tolist()))  # 32*200
```

6400

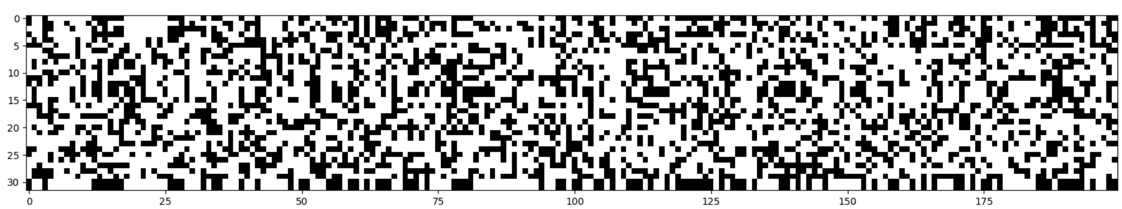
[30]:
```
plt.hist(h.view(-1).tolist(), 50)
# a lot of neurons  are ' saturated'
```



[31]:
```
# now look at the "broad" shape of the 'hpreact' distribution.....and this is␣
↪bad for learning....we want 'normality' for our brain...
# tgh'(x)= (1-tgh(x)^2).... saturation bring to vanish gradient...no learning...
plt.hist(hpreact.view(-1).tolist(), 50)
# a lot of neurons  are ' saturated'
```



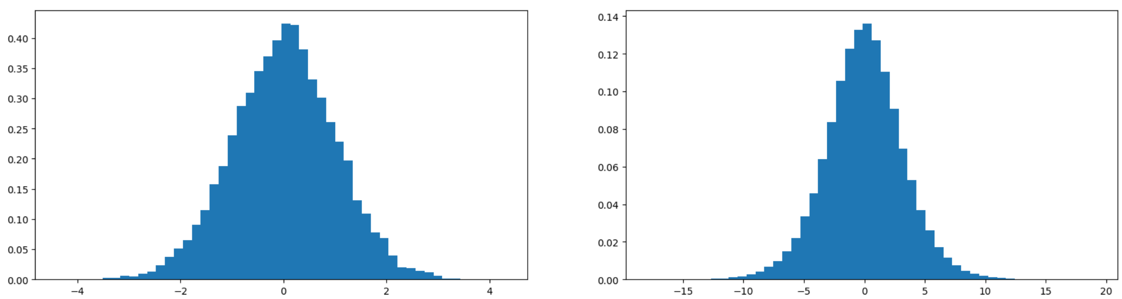
[33]:
```
# looking for dead neurons....comment other Activation Functions..... go back
↪weigth the first layer and try again...

plt.figure(figsize=(20, 10))
plt.imshow(h.abs() > 0.99, cmap="gray", interpolation="nearest")
```



[34]:
```
# why we do not to have to worry too much about inizialization....bach
↪normalization..
# since 2015
# https://arxiv.org/abs/1502.03167
# if the problems are fluctuations and saturations (discuss in class)...then
↪just gaussain normalize at each layer...
# simple as that !!!...and normalizing is differentiable !!....
```

```python
x = torch.randn(1000, 10)
w = torch.randn(10, 200)  # *.2 #*1/10**0.5
y = x @ w
print(y.shape)
print(x.mean(), x.std())
print(y.mean(), y.std())
plt.figure(figsize=(20, 5))
plt.subplot(121)
plt.hist(x.view(-1).tolist(), 50, density=True)
plt.subplot(122)
plt.hist(y.view(-1).tolist(), 50, density=True)
```

```
torch.Size([1000, 200])
tensor(0.0148) tensor(0.9958)
tensor(-0.0002) tensor(3.1940)
```




```
[ ]: # summary from 1:18 after the discussion about batch normalization....
```

4 Convolutional Network - MakeMore pt.4 (28/04/2023)

For real the 4th was about back propagation, but we skip it in this course. For the final exam one may go deeper into this algorithm, maybe by hand.

Last time we implemented a multilayer perceptron (with actually two layers), working at character level and building a probability distribution. The pipeline was: - embedding (2 -10 dimensional vectors) - glue them together, but this is not the best thing one can do

Instead of looking at 3 previous char we can look at 8 previous char and so on… We'll see it doesn't change much. Idea: feed the net with information in a hierarchical way. Notice that we'll work with text, but we can expand this also to images. We want to construct a convolution network, even if it's not related to the mathematical definition of convolution. Glue chars together, giving it to a layer, process, glue another char and give to the next layer and so on. The line behind all this is entropy, we skip the compression algorithm for lack of time. Probability distribution, entropy and compression are actually the same thing.

[1]:
```
# now we want to enlarge the context length AND ''fuse information in a␣
↪hierarchical manner''
# see the approach in https://arxiv.org/abs/1609.03499
```

[2]:
```
# The importance of embedding

# word2vec: skip-gram https://arxiv.org/pdf/1301.3781v3.pdf
# node2vec: https://arxiv.org/pdf/1607.00653.pdf
```

Data are not always linear, or a lattice, or on a euclidean space… Things are complicated. However, for some phenomena, they're naturally distributed in networks (or if you want, graph). The main concepts are neighbors, distances… As we can handle text we can handle images, which are just euclidean spaces.

This field is going extremely fast, keep going!

We deal with real numbers which can be positive, negative, very big, very small… The hyperbolic tangent help us to squeeze them. Otherwise, the neurons won't learn. There is still a lot to understand there… Take it as it is, *e più non dimandare*. Actually, there are many ways to squeeze without the hyperbolic tangent, but the meaning is the same.

We'll assume batch normalization and focus on the method. We'll hierarchically glue 8 char together. See also WaveNet for audio generation. We want to pass from 2+2+2 $\to$ 6 to 2 $\to$ 3 $\to$ … $\to$ 6, i.e. gradually. Convolutional layers like this are very spread and useful for text analysis.

[3]:
```
import random
import torch
import torch.nn.functional as F
import torch.nn as nn
import matplotlib.pyplot as plt
%matplotlib inline
```

[4]:
```
# read in all the words
random.seed(158)
words = (
    open("data/nomi_italiani.txt", "r").read().splitlines()
)  # each line is an element of the list
random.shuffle(words)
print(len(words))
print(words[0:8])
```

```
9105
['argento', 'giovannino', 'licurga', 'elvira', 'marena', 'sirio', 'emilia',
'bisio']
```

[5]:
```
# build the vocabulary of characters and mappings to/from integers (encoder/
↪decoder)
chars = sorted(list(set("".join(words))))
stoi = {s: i + 1 for i, s in enumerate(chars)}
```

```
stoi["."] = 0
itos = {i: s for s, i in stoi.items()}
vocab_size = len(stoi)
print(itos)
print(vocab_size)
```

```
{1: '-', 2: 'a', 3: 'b', 4: 'c', 5: 'd', 6: 'e', 7: 'f', 8: 'g', 9: 'h', 10:
'i', 11: 'j', 12: 'k', 13: 'l', 14: 'm', 15: 'n', 16: 'o', 17: 'p', 18: 'q', 19:
'r', 20: 's', 21: 't', 22: 'u', 23: 'v', 24: 'w', 25: 'x', 26: 'y', 27: 'z', 0:
'.'}
28
```

[6]:

```
# build the dataset
block_size = 8  # 8 context length: how many characters do we take to predict
 ↪the next one?...start with 3 !!


def build_dataset(words):
    X, Y = [], []

    for w in words:
        context = [0] * block_size
        for ch in w + ".":
            ix = stoi[ch]
            X.append(context)
            Y.append(ix)
            context = context[1:] + [ix]  # crop and append

    X = torch.tensor(X)
    Y = torch.tensor(Y)
    print(X.shape, Y.shape)
    return X, Y


n1 = int(0.8 * len(words))
n2 = int(0.9 * len(words))
Xtr, Ytr = build_dataset(words[:n1])  # 80%
Xdev, Ydev = build_dataset(words[n1:n2])  # 10%
Xte, Yte = build_dataset(words[n2:])  # 10%
```

```
torch.Size([59049, 8]) torch.Size([59049])
torch.Size([7332, 8]) torch.Size([7332])
torch.Size([7262, 8]) torch.Size([7262])
```

[7]:

```
for x, y in zip(Xtr[:20], Ytr[:20]):
    print("".join(itos[ix.item()] for ix in x), "-->", itos[y.item()])
```

```
… --> a
```

```
...a --> r
...ar --> g
...arg --> e
...arge --> n
...argen --> t
..argent --> o
.argento --> .
... --> g
...g --> i
...gi --> o
...gio --> v
...giov --> a
...giova --> n
..giovan --> n
.giovann --> i
giovanni --> n
iovannin --> o
ovannino --> .
... --> l
```

Let's do it in an object-oriented way. These things are already contained in PyTorch, so we don't need to write them by hand. All these classes are into _torch.nn.*_

[8]:
```python
# Near copy paste of the layers we have developed in Part 3
# Comment on this! https://pytorch.org/docs/stable/nn.html
# now just evaluate this then we will chage it and use torch functions!
# all these definitions work as in PyTorch, but we won't use it as a black box



#
 ↪------------------------------------------------------------------------------------------
class Linear:
    def __init__(self, fan_in, fan_out, bias=True):
        self.weight = (
            torch.randn((fan_in, fan_out)) / fan_in**0.5
        )  # note: kaiming init
        self.bias = torch.zeros(fan_out) if bias else None

    def __call__(self, x):
        self.out = x @ self.weight
        if self.bias is not None:
            self.out += self.bias
        return self.out

    def parameters(self):
        return [self.weight] + ([] if self.bias is None else [self.bias])
```

```python
# -----------------------------------------------------------------------------------------------
class BatchNorm1d:
    def __init__(self, dim, eps=1e-5, momentum=0.1):
        self.eps = eps
        self.momentum = momentum
        self.training = True
        # parameters (trained with backprop)
        self.gamma = torch.ones(dim)
        self.beta = torch.zeros(dim)
        # buffers (trained with a running 'momentum update')
        self.running_mean = torch.zeros(dim)
        self.running_var = torch.ones(dim)

    def __call__(self, x):
        # calculate the forward pass
        if self.training:
            if x.ndim == 2:
                dim = 0
            elif x.ndim == 3:
                dim = (0, 1)
            xmean = x.mean(dim, keepdim=True)  # batch mean
            xvar = x.var(dim, keepdim=True)  # batch variance
        else:
            xmean = self.running_mean
            xvar = self.running_var
        # normalize to unit variance
        xhat = (x - xmean) / torch.sqrt(xvar + self.eps)
        self.out = self.gamma * xhat + self.beta
        # update the buffers
        if self.training:
            with torch.no_grad():
                self.running_mean = (
                    1 - self.momentum
                ) * self.running_mean + self.momentum * xmean
                self.running_var = (
                    1 - self.momentum
                ) * self.running_var + self.momentum * xvar
        return self.out

    def parameters(self):
        return [self.gamma, self.beta]


# -----------------------------------------------------------------------------------------------
class Tanh:
```

```python
    def __call__(self, x):
        self.out = torch.tanh(x)
        return self.out

    def parameters(self):
        return []


# -------------------------------------------------------------------------------------------
class Embedding:
    def __init__(self, num_embeddings, embedding_dim):
        self.weight = torch.randn((num_embeddings, embedding_dim))

    def __call__(self, IX):
        self.out = self.weight[IX]
        return self.out

    def parameters(self):
        return [self.weight]


# -------------------------------------------------------------------------------------------
class Flatten:
    def __init__(self, n):
        self.n = n

    def __call__(self, x):
        self.out = x.view(x.shape[0], -1)
        return self.out

    def parameters(self):
        return []


# -------------------------------------------------------------------------------------------
class FlattenConsecutive:
    def __init__(self, n):
        self.n = n

    def __call__(self, x):
        B, T, C = x.shape
        x = x.view(B, T // self.n, C * self.n)
        if x.shape[1] == 1:
            x = x.squeeze(1)
```

```
        self.out = x
        return self.out

    def parameters(self):
        return []


#␣
↪---------------------------------------------------------------------------------------------
class Sequential:
    def __init__(self, layers):
        self.layers = layers

    def __call__(self, x):
        for layer in self.layers:
            x = layer(x)
        self.out = x
        return self.out

    def parameters(self):
        # get parameters of all layers and stretch them out into one list
        return [p for layer in self.layers for p in layer.parameters()]
```

[9]:
```
torch.manual_seed(42)
```

[9]: <torch._C.Generator at 0x7f50c6067390>

C is our embedding matrix 28x10 which contains the 10 dimensional embedding for each of 28 chars. Context length is 3, multiply by 10 so 30 is the dimension of the input layer.

[10]:
```
# original network https://jmlr.org/papers/volume3/tmp/bengio03a.pdf

n_embd = 10  # the dimensionality of the character embedding vectors
n_hidden = 200  # the number of neurons in the hidden layer of the MLP

C = torch.randn(vocab_size, n_embd)
layers = [
    Linear(n_embd * block_size, n_hidden, bias=False),
    BatchNorm1d(n_hidden),  # normalize, otherwise learning stops
    Tanh(),
    Linear(n_hidden, vocab_size),
]

# parameter initialization

with torch.no_grad():
    layers[-1].weight *= 0.1  # last layer make less confident
```

```python
parameters = [C] + [p for layer in layers for p in layer.parameters()]
print(sum(p.nelement() for p in parameters))  # number of parameters in total
for p in parameters:
    p.requires_grad = True
```

22308

[11]:
```python
# same optimization as last time
max_steps = 200000
batch_size = 32
lossi = []

for i in range(max_steps):
    # minibatch construct
    ix = torch.randint(0, Xtr.shape[0], (batch_size,))
    Xb, Yb = Xtr[ix], Ytr[ix]  # batch X,Y

    # forward pass

    emb = C[Xb]  # embed the characters into vectors
    x = emb.view(
        emb.shape[0], -1
    )  # concatenate the vectors, view is not memory consuming
    for layer in layers:
        x = layer(x)
    loss = F.cross_entropy(x, Yb)  # loss function

    # backward pass, now we're using our black box
    for p in parameters:
        p.grad = None
    loss.backward()

    # update: simple SGD
    lr = 0.1 if i < 150000 else 0.01  # step learning rate decay
    for p in parameters:
        p.data += -lr * p.grad
        # track stats
    if i % 10000 == 0:  # print every once in a while
        print(f"{i:7d}/{max_steps:7d}: {loss.item():.4f}")
    lossi.append(loss.log10().item())
```

```
      0/ 200000: 3.3371
  10000/ 200000: 2.1734
  20000/ 200000: 1.9819
  30000/ 200000: 1.6350
  40000/ 200000: 1.3981
  50000/ 200000: 1.6659
```

```
 60000/ 200000: 1.8349
 70000/ 200000: 1.6599
 80000/ 200000: 1.2059
 90000/ 200000: 1.6775
100000/ 200000: 1.3948
110000/ 200000: 1.3727
120000/ 200000: 1.7746
130000/ 200000: 1.5139
140000/ 200000: 1.3102
150000/ 200000: 1.4078
160000/ 200000: 1.3451
170000/ 200000: 1.2467
180000/ 200000: 1.4784
190000/ 200000: 1.3532
```

Why 3.3 at the beginning? Assuming all random, $-\log \frac{1}{28}$ is about that number...

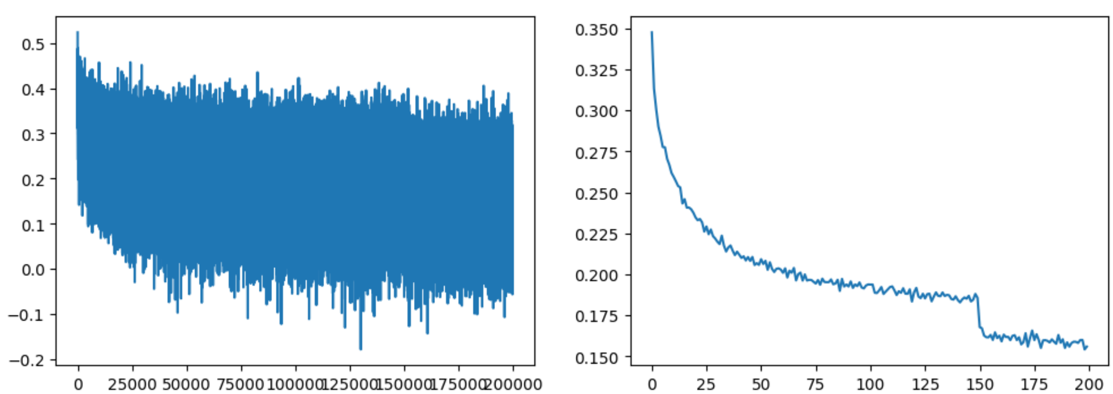
NOTE that with 8 the decrease is very slow, we're fusing info too quickly!

Small batch implies fluctuating a lot, so we can use view to split in pieces of one thousand.

[12]:
```
print(-torch.tensor(1 / 28).log())
# 32 batches are few... so you can get very lucky or unlucky

fig, ax = plt.subplots(ncols=2, figsize=(12, 4))

ax[0].plot(torch.tensor(lossi))
ax[1].plot(torch.tensor(lossi).view(-1, 1000).mean(1))  # mean on each row
```

```
tensor(3.3322)
```

[12]:
```
[<matplotlib.lines.Line2D at 0x7f4ff1b2c1c0>]
```



The second plot is much better, and we didn't change the data! Something happened in the step, it jumped on a lower state (i.e. it learned).

Now we want to evaluate. We need to tell PyTorch that we're not in the training phase anymore. **BE CAREFUL**, or you'll get weird results due to batch normalization, which for us is a black box.

[13]:
```
# put layers into eval mode (needed for batchnorm especially)
for layer in layers:
    layer.training = False
```

[14]:
```
# evaluate the loss
@torch.no_grad()  # this decorator disables gradient tracking inside pytorch
def split_loss(split):
    x, y = {
        "train": (Xtr, Ytr),
        "val": (Xdev, Ydev),
        "test": (Xte, Yte),
    }[split]
    emb = C[x]  # embed the characters into vectors (N, block_size)
    x = emb.view(emb.shape[0], -1)  # concatenate the vectors
    for layer in layers:
        x = layer(x)
    loss = F.cross_entropy(x, y)  # loss function
    print(split, loss.item())


split_loss("train")
split_loss("val")
```

```
train 1.3725385665893555
val 1.7553908824920654
```

[15]:
```
# sample from the model
for _ in range(20):
    out = []
    context = [0] * block_size  # initialize with all ...
    while True:
        # forward pass the neural net
        emb = C[
            torch.tensor([context])
        ]  # embed the characters into vectors (N,block_size)
        x = emb.view(emb.shape[0], -1)  # concatenate the vectors
        for layer in layers:
            x = layer(x)
        logits = x
        probs = F.softmax(logits, dim=1)
        # sample from the distribution
        ix = torch.multinomial(probs, num_samples=1).item()
        # shift the context window and track the samples
        context = context[1:] + [ix]
```

```
        out.append(ix)
        # if we sample the special '.' token, break
        if ix == 0:
            break

    print("".join(itos[i] for i in out))  # decode and print the generated word
```

```
amode.
leandro.
alfisio.
fabbions.
amperia.
oreino.
silvina.
wantie.
olderiza.
orea.
consolita.
mariacrostelfino.
emerande.
giandamaro.
fiero.
slorena-gettto.
morino.
morietta.
alcidisso.
artemine.
```

[16]:

```
# we can do better and use "Embedding" (as pytorch) to see C as a first layer



# original network https://jmlr.org/papers/volume3/tmp/bengio03a.pdf

n_embd = 10  # the dimensionality of the character embedding vectors
n_hidden = 200  # the number of neurons in the hidden layer of the MLP

# C= torch.randn(vocab_size,n_embd)
layers = [
    Embedding(vocab_size, n_embd),
    Flatten(block_size),
    Linear(n_embd * block_size, n_hidden, bias=False),
    BatchNorm1d(n_hidden),
    Tanh(),
    Linear(n_hidden, vocab_size),
]

# parameter initialization
```

```python
with torch.no_grad():
    layers[-1].weight *= 0.1  # last layer make less confident

# parameters=[C]+[p for layer in layers for p in layer.parameters()]
parameters = [p for layer in layers for p in layer.parameters()]
print(sum(p.nelement() for p in parameters))  # number of parameters in total
for p in parameters:
    p.requires_grad = True
```

22308

In PyTorch one can write this as a sequential list of layer, with the same function names.

[17]:
```python
n_embd = 10  # the dimensionality of the character embedding vectors
n_hidden = 200  # the number of neurons in the hidden layer of the MLP

model = nn.Sequential(
    nn.Embedding(vocab_size, n_embd),
    nn.Flatten(block_size),
    nn.Linear(n_embd * block_size, n_hidden, bias=False),
    nn.BatchNorm1d(n_hidden),
    nn.Tanh(),
    nn.Linear(n_hidden, vocab_size),
)

# number of parameters in total
print(sum(p.nelement() for p in model.parameters()))
for p in model.parameters():
    p.requires_grad = True
```

22308

[18]:
```python
# or with 'karpathy' definitions...

n_embd = 10  # the dimensionality of the character embedding vectors
n_hidden = 200  # the number of neurons in the hidden layer of the MLP

model = Sequential(
    [
        Embedding(vocab_size, n_embd),
        Flatten(block_size),
        Linear(n_embd * block_size, n_hidden, bias=False),
        BatchNorm1d(n_hidden),
        Tanh(),
        Linear(n_hidden, vocab_size),
    ]
)
```

```python
# number of parameters in total
print(sum(p.nelement() for p in model.parameters()))
for p in model.parameters():
    p.requires_grad = True
```

22308

[19]:
```python
# put layers into eval mode (needed for batchnorm especially)
# be back on this now keep it !!

for layer in model.layers:
    layer.training = True

# model.train(True)
```

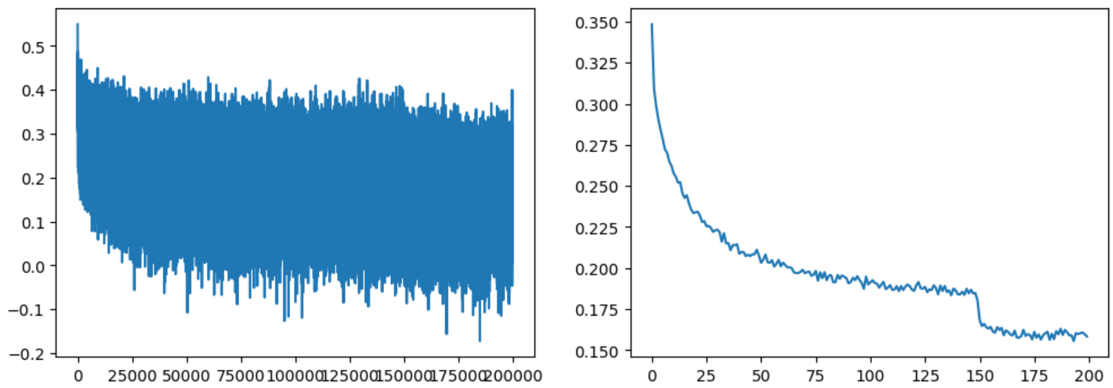
[20]:
```python
# same optimization as before time
max_steps = 200000
batch_size = 32
lossi = []

for i in range(max_steps):
    # minibatch construct
    ix = torch.randint(0, Xtr.shape[0], (batch_size,))
    Xb, Yb = Xtr[ix], Ytr[ix]  # batch X,Y

    # forward pass
    #   emb=C[Xb] # embed the characters into vectors
    #   x=emb.view(emb.shape[0],-1) #concatenate the vectors
    #   for layer in layers:
    #     x=layer(x)

    logits = model(Xb)
    loss = F.cross_entropy(logits, Yb)  # loss function

    # backward pass
    for p in model.parameters():
        p.grad = None

    loss.backward()

    # update: simple SGD
    lr = 0.1 if i < 150000 else 0.01  # step learning rate decay
    for p in model.parameters():
        p.data += -lr * p.grad

    # track stats
```

```
    if i % 10000 == 0:  # print every once in a while
        print(f"{i:7d}/{max_steps:7d}: {loss.item():.4f}")
    lossi.append(loss.log10().item())
# break
```

```
      0/ 200000: 3.5427
  10000/ 200000: 1.6889
  20000/ 200000: 1.7099
  30000/ 200000: 1.7663
  40000/ 200000: 1.6661
  50000/ 200000: 1.5461
  60000/ 200000: 1.8788
  70000/ 200000: 1.7811
  80000/ 200000: 1.9472
  90000/ 200000: 1.3643
 100000/ 200000: 1.5458
 110000/ 200000: 1.4089
 120000/ 200000: 1.6715
 130000/ 200000: 2.0241
 140000/ 200000: 1.6338
 150000/ 200000: 1.5033
 160000/ 200000: 1.5036
 170000/ 200000: 1.5975
 180000/ 200000: 1.3502
 190000/ 200000: 1.6590
```

[21]:
```
fig, ax = plt.subplots(ncols=2, figsize=(12, 4))

ax[0].plot(torch.tensor(lossi))
ax[1].plot(torch.tensor(lossi).view(-1, 1000).mean(1))  # mean on each row
```

[21]:
```
[<matplotlib.lines.Line2D at 0x7f4ff1b2d660>]
```



[22]:
```
# put layers into eval mode (needed for batchnorm especially)
# be back on this now keep it !!

for layer in model.layers:
    layer.training = False

# model.train(True)
```

[23]:
```
# evaluate the loss
@torch.no_grad()  # this decorator disables gradient tracking inside pytorch
def split_loss(split):
    x, y = {
        "train": (Xtr, Ytr),
        "val": (Xdev, Ydev),
        "test": (Xte, Yte),
    }[split]

    #   emb=C[x] # embed the characters into vectors (N,block_size) before
    #   x=emb.view(emb.shape[0],-1) #concatenate the vectors

    #     for layer in layers:
    #         x=layer(x)

    logits = model(x)

    loss = F.cross_entropy(logits, y)  # loss function

    print(split, loss.item())


split_loss("train")
split_loss("val")
```

```
train 1.3757604360580444
val 1.73955237865448
```

[24]:
```
# sample from the model
for _ in range(20):
    out = []
    context = [0] * block_size  # initialize with all ...
    while True:
        # forward pass the neural net
        #         emb=C[torch.tensor([context])] # embed the characters into␣
 ↪vectors (N,block_size)
        #         x=emb.view(emb.shape[0],-1) #concatenate the vectors
        #         for layer in layers:
        #             x=layer(x)
```

```
    #         logits = x
    logits = model(torch.tensor([context]))
    probs = F.softmax(logits, dim=1)
    # sample from the distribution
    ix = torch.multinomial(probs, num_samples=1).item()
    # shift the context window and track the samples
    context = context[1:] + [ix]
    out.append(ix)
    # if we sample the special '.' token, break
    if ix == 0:
        break

print("".join(itos[i] for i in out))  # decode and print the generated word
```

```
tima.
andrea.
laurezio.
loriela.
anio.
lucido.
eriscondo.
guerriana.
ginepro.
umbra.
angeloce.
raffoso.
marziano.
erlene.
bonulana.
eva.
abelda.
riziaddino.
calaria.
eride.
```

[25]:
```
# now we are going to try to do better...but because we crunch all the␣
↪characters at the first  layer (show picture language model),
# adding deeper layer will not be useful...we need first to do as wavenet:␣
↪cluster characters in steps, hierarchialy...(convolution)
# Progressive fusion along the layers: 2-gram -> 4->gram -> 8-gram....show␣
↪picture in wavenet paper.....

# first recreate the data set with block_size = 8
```

[26]:
```
# let's look at a batch of just 4 example
ix = torch.randint(0, Xtr.shape[0], (4,))
Xb, Yb = Xtr[ix], Ytr[ix]
```

```
logits = model(Xb)
print(Xb.shape)
print(Xb)
```

```
torch.Size([4, 8])
tensor([[ 0,  0,  0,  0,  7, 10, 15,  2],
        [ 0,  8,  2,  3, 19, 10,  6, 13],
        [ 0,  0,  0,  0,  0,  0,  0,  8],
        [10, 23,  2, 13,  5, 10, 15, 16]])
```

[27]:
```
print(model.layers[0].out.shape)  # output of the embedding layer
print(model.layers[1].out.shape)  # output of the Flatten layer
print(model.layers[2].out.shape)  # output of the Linear layer
```

```
torch.Size([4, 8, 10])
torch.Size([4, 80])
torch.Size([4, 200])
```

Now multiply each vector for the matrix and take a 200dim vector as output

[28]:
```
# now a nice surprise about the Linear Layer https://pytorch.org/docs/stable/
↪generated/torch.nn.Linear.html#torch.nn.Linear

# just to look at the dimensions

(
    torch.randn(4, 80) @ torch.randn(80, 200) + torch.randn(200)
).shape  # broadcasting in the last term....
```

[28]:
```
torch.Size([4, 200])
```

We can also add more dimensions… and PyTorch will work on the right dimension.

[29]:
```
# but also !!
print((torch.randn(4, 2, 80) @ torch.randn(80, 200) + torch.randn(200)).shape)
# or also !!
print((torch.randn(4, 4, 20) @ torch.randn(20, 200) + torch.randn(200)).shape)
```

```
torch.Size([4, 2, 200])
torch.Size([4, 4, 200])
```

Instead of one vector of 80 we can use four vectors of 20.

[30]:
```
# 1 2 3 4 5 6 7 8 -> (1 2) (3 4) (5 6) (7 8)
# we want to fuse vectors in pairs and acts in parallel on the 4 pairs of␣
↪characters.
# i.e. we go from (torch.randn(4,80)@ torch.randn(80,200)+torch.randn(200)).
↪shape
# ....second batch dimension...discuss in class...!!
```

```
print((torch.randn(4, 4, 20) @ torch.randn(20, 200) + torch.randn(200)).shape)
```

```
torch.Size([4, 4, 200])
```

[31]:
```
# so now we need to change the Flatten layer to produce a (4,4,20) tensor and␣
↪NOT (4,80)
e = torch.randn(
    4, 8, 10
)  # goal: want this to be (4,4,20) where consecutive 10-d vectors
# get concatenated (in pairs (1 2) (3 4) (5 6) (7 8))
```

[32]:
```
# trick

print(list(range(10)))
print(list(range(10))[::2])
print(list(range(10))[1::2])
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 2, 4, 6, 8]
[1, 3, 5, 7, 9]
```

[33]:
```
# so we want this...
print(e.shape)
explicit = torch.cat([e[:, ::2, :], e[:, 1::2, :]], dim=2)
print(explicit.shape)
```

```
torch.Size([4, 8, 10])
torch.Size([4, 4, 20])
```

[34]:
```
# but ALSO this works !!

# (e.view(4, 4, 20) == explicit)  # .all()   try this...
```

[35]:
```
input = torch.randn(4, 8, 10)
print(input.shape)
m = nn.Flatten()
output = m(input)
print(output.shape)
```

```
torch.Size([4, 8, 10])
torch.Size([4, 80])
```

[36]:
```
#␣
↪-----------------------------------------------------------------------------------
class FlattenConsecutive:
    def __init__(self, n):
```

```
        self.n = n

    def __call__(self, x):
        B, T, C = x.shape
        x = x.view(B, T // self.n, C * self.n)
        if x.shape[1] == 1:
            x = x.squeeze(1)
        self.out = x
        return self.out

    def parameters(self):
        return []
```

[37]:
```
# Back to the Model...here we recover the previous one with␣
 ↪FlattenConsecutive(block_size)


n_embd = 10  # the dimensionality of the character embedding vectors
n_hidden = 200  # the number of neurons in the hidden layer of the MLP

# model = nn.Sequential(
#     nn.Embedding(vocab_size, n_embd),nn.Flatten(), nn.Linear(n_embd *␣
 ↪block_size, n_hidden, bias=False),
#     nn.BatchNorm1d(n_hidden), nn.Tanh(),
#     nn.Linear(n_hidden, vocab_size))

model = Sequential(
    [
        Embedding(vocab_size, n_embd),
        FlattenConsecutive(block_size),
        Linear(n_embd * block_size, n_hidden, bias=False),
        BatchNorm1d(n_hidden),
        nn.Tanh(),
        Linear(n_hidden, vocab_size),
    ]
)


# parameter init


# with torch.no_grad()
#    layers=model.layers()
#    layers[-1].weight *= 0.1 # last layer make less confident # fix this !!

# parameters=[C]+[p for layer in layers for p in layer.parameters()]  before
```

```
# number of parameters in total
print(sum(p.nelement() for p in model.parameters()))
for p in model.parameters():
    p.requires_grad = True
```

22308

[38]:
```
# let's look at a batch of just 4 example
ix = torch.randint(0, Xtr.shape[0], (4,))
Xb, Yb = Xtr[ix], Ytr[ix]
logits = model(Xb)
print(Xb.shape)
print(Xb)
```

```
torch.Size([4, 8])
tensor([[ 0,  0,  0,  0,  0,  0,  0,  0],
        [ 0,  0,  0,  0,  0,  0,  0,  4],
        [ 0,  0,  0,  0,  0,  0,  0,  0],
        [ 0,  0,  0,  0, 16, 13, 10, 23]])
```

[39]:
```
for layer in model.layers[:-2]:
    print(layer.__class__.__name__, ":", tuple(layer.out.shape))
```

```
Embedding : (4, 8, 10)
FlattenConsecutive : (4, 80)
Linear : (4, 200)
BatchNorm1d : (4, 200)
```

[40]:
```
# we now move to a hierarchical approach

n_embd = 10  # the dimensionality of the character embedding vectors
n_hidden = 200  # the number of neurons in the hidden layer of the MLP

model = Sequential(
    [
        Embedding(vocab_size, n_embd),
        FlattenConsecutive(2),
        Linear(n_embd * 2, n_hidden, bias=False),
        BatchNorm1d(n_hidden),
        Tanh(),
        FlattenConsecutive(2),
        Linear(n_hidden * 2, n_hidden, bias=False),
        BatchNorm1d(n_hidden),
        Tanh(),
        FlattenConsecutive(2),
        Linear(n_hidden * 2, n_hidden, bias=False),
        BatchNorm1d(n_hidden),
```

```
        Tanh(),
        Linear(n_hidden, vocab_size),
    ]
)


# number of parameters in total
print(sum(p.nelement() for p in model.parameters()))
for p in model.parameters():
    p.requires_grad = True
```

```
171108
```

[41]:
```
# let's look at a batch of just 4 example
ix = torch.randint(0, Xtr.shape[0], (4,))
Xb, Yb = Xtr[ix], Ytr[ix]
logits = model(Xb)
print(Xb.shape)
print(Xb)
```

```
torch.Size([4, 8])
tensor([[ 0,  0,  0,  0,  0, 14,  2, 19],
        [ 0,  0,  0,  0,  0,  0,  0,  8],
        [ 0,  0,  0,  0,  0,  0,  0,  0],
        [ 0,  0,  0,  8, 22, 20, 21,  2]])
```

[42]:
```
for layer in model.layers:
    print(layer.__class__.__name__, ":", tuple(layer.out.shape))
```

```
Embedding : (4, 8, 10)
FlattenConsecutive : (4, 4, 20)
Linear : (4, 4, 200)
BatchNorm1d : (4, 4, 200)
Tanh : (4, 4, 200)
FlattenConsecutive : (4, 2, 400)
Linear : (4, 2, 200)
BatchNorm1d : (4, 2, 200)
Tanh : (4, 2, 200)
FlattenConsecutive : (4, 400)
Linear : (4, 200)
BatchNorm1d : (4, 200)
Tanh : (4, 200)
Linear : (4, 28)
```

[43]:
```
print(logits.shape)
```

```
torch.Size([4, 28])
```

[44]:
```
# we now move to a hierarchical approach but resize to compare with the initial␣
↪onwe
# we use n_hidden=68 to have almost the same number of parameters..,

n_embd = 10  # the dimensionality of the character embedding vectors
n_hidden = 200  # the number of neurons in the hidden layer of the MLP

model = Sequential(
    [
        Embedding(vocab_size, n_embd),
        FlattenConsecutive(2),
        Linear(n_embd * 2, n_hidden, bias=False),
        BatchNorm1d(n_hidden),
        Tanh(),
        FlattenConsecutive(2),
        Linear(n_hidden * 2, n_hidden, bias=False),
        BatchNorm1d(n_hidden),
        Tanh(),
        FlattenConsecutive(2),
        Linear(n_hidden * 2, n_hidden, bias=False),
        BatchNorm1d(n_hidden),
        Tanh(),
        Linear(n_hidden, vocab_size),
    ]
)


# number of parameters in total
print(sum(p.nelement() for p in model.parameters()))
for p in model.parameters():
    p.requires_grad = True
```

```
171108
```

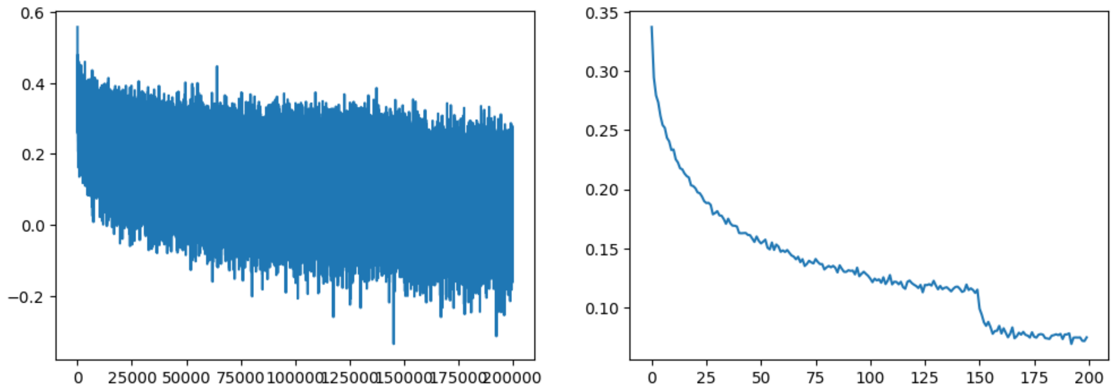
[45]:
```
# same optimization as before
max_steps = 200000
batch_size = 32
lossi = []

for i in range(max_steps):
    # minibatch construct
    ix = torch.randint(0, Xtr.shape[0], (batch_size,))
    Xb, Yb = Xtr[ix], Ytr[ix]  # batch X,Y

    # forward pass

    logits = model(Xb)
```

```
    loss = F.cross_entropy(logits, Yb)  # loss function

    # backward pass

    for p in model.parameters():
        p.grad = None

    loss.backward()

    # update: simple SGD
    lr = 0.1 if i < 150000 else 0.01  # step learning rate decay
    for p in model.parameters():
        p.data += -lr * p.grad

    # track stats
    if i % 10000 == 0:  # print every once in a while
        print(f"{i:7d}/{max_steps:7d}: {loss.item():.4f}")
    lossi.append(loss.log10().item())
# break
```

```
      0/ 200000: 3.6123
  10000/ 200000: 1.8124
  20000/ 200000: 1.6373
  30000/ 200000: 1.7834
  40000/ 200000: 1.2532
  50000/ 200000: 1.8544
  60000/ 200000: 1.0383
  70000/ 200000: 1.5017
  80000/ 200000: 1.0546
  90000/ 200000: 1.0752
 100000/ 200000: 1.4752
 110000/ 200000: 1.1615
 120000/ 200000: 1.2915
 130000/ 200000: 1.1062
 140000/ 200000: 1.1331
 150000/ 200000: 1.4284
 160000/ 200000: 1.9482
 170000/ 200000: 1.2848
 180000/ 200000: 1.2159
 190000/ 200000: 1.3101
```

[46]:
```
fig, ax = plt.subplots(ncols=2, figsize=(12, 4))

ax[0].plot(torch.tensor(lossi))
ax[1].plot(torch.tensor(lossi).view(-1, 1000).mean(1))  # mean on each row
```

[46]:
```
[<matplotlib.lines.Line2D at 0x7f4fef808100>]
```



[47]:
```
# put layers into eval mode (needed for batchnorm especially)...comment in class !
for layer in model.layers:
    layer.training = False
```

[48]:
```
# evaluate the loss
@torch.no_grad()  # this decorator disables gradient tracking inside pytorch
def split_loss(split):
    x, y = {
        "train": (Xtr, Ytr),
        "val": (Xdev, Ydev),
        "test": (Xte, Yte),
    }[split]

    logits = model(x)

    loss = F.cross_entropy(logits, y)  # loss function

    print(split, loss.item())


split_loss("train")
split_loss("val")
```

```
train 1.1616768836975098
val 1.8372830152511597
```

[49]:
```
# sample from the model
for _ in range(20):
    out = []
    context = [0] * block_size  # initialize with all ...
    while True:
        logits = model(torch.tensor([context]))
```

```python
        probs = F.softmax(logits, dim=1)
        # sample from the distribution
        ix = torch.multinomial(probs, num_samples=1).item()
        # shift the context window and track the samples
        context = context[1:] + [ix]
        out.append(ix)
        # if we sample the special '.' token, break
        if ix == 0:
            break

    print("".join(itos[i] for i in out))  # decode and print the generated word
```

```
gaspawa.
waldemaro.
ambra.
aida.
eraldina.
isacco.
amadio.
mariso.
fabrina.
valdimaro.
gerarda.
opaldo.
firmanina.
melchiorino.
nicoletto.
lallo.
giommato.
callisto.
sostino.
cleontina.
```