Notebook

May 17, 2023

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

We now move on to see some interesting applications of the theory we developed earlier on.

Mostly, we will see how to construct a language model, character based, able to 'learn' how to 'reproduce' $\mu(x_n|x_1, \ldots, x_{n-1})$. We will develop such model following "chronological" steps, in the sense that we'll start from an older programming structure and later improve the latest feature in Artificial Intelligence.

Almost all the material presented is taken from A. Karpathy's course and GitHub repository.

First, we start with *counting approach* (not to be confused with the counting technique earlier presented) to implement a bigram approximation of our "language" model (a "2-Markov" approximation).

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

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

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

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

[24]:
```python
import numpy as np

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

```
7.088193300384404
```

[25]:
```python
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]:
```
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]:
```
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]:
```
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)
```

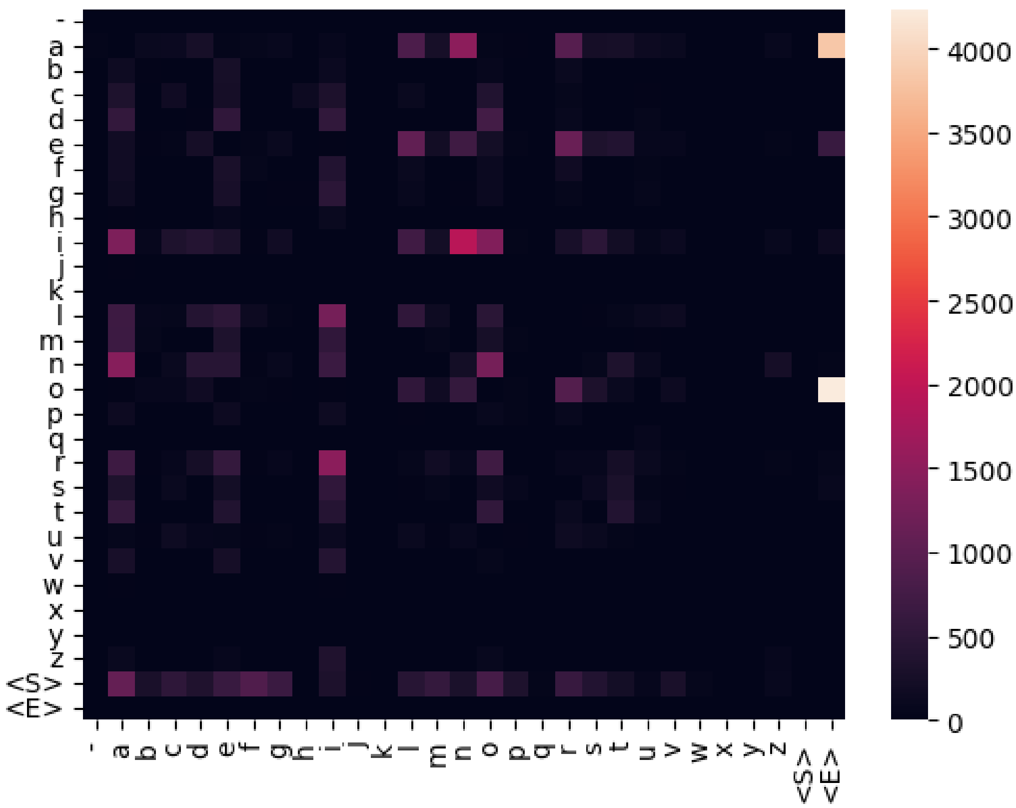
```
{'-': 0, 'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5, 'f': 6, 'g': 7, 'h': 8, 'i': 9,
'j': 10, 'k': 11, 'l': 12, 'm': 13, 'n': 14, 'o': 15, 'p': 16, 'q': 17, 'r': 18,
's': 19, 't': 20, 'u': 21, 'v': 22, 'w': 23, 'x': 24, 'y': 25, 'z': 26, '<S>':
27, '<E>': 28}
```

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

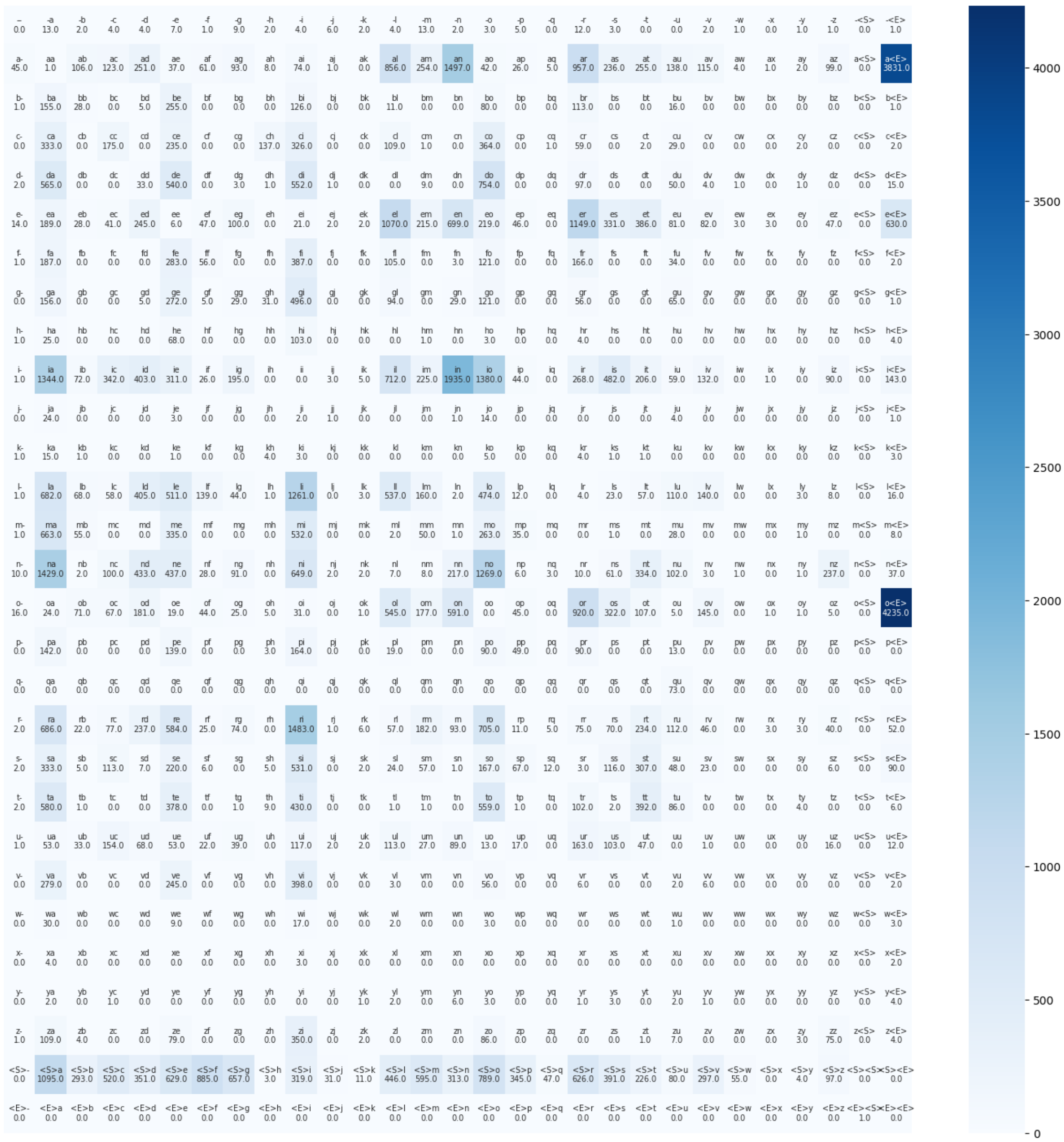
```
{0: '-', 1: 'a', 2: 'b', 3: 'c', 4: 'd', 5: 'e', 6: 'f', 7: 'g', 8: 'h', 9: 'i',
10: 'j', 11: 'k', 12: 'l', 13: 'm', 14: 'n', 15: 'o', 16: 'p', 17: 'q', 18: 'r',
19: 's', 20: 't', 21: 'u', 22: 'v', 23: 'w', 24: 'x', 25: 'y', 26: 'z', 27:
'<S>', 28: '<E>'}
```

Notice that **stoi** is associating to any character a number, so they are enumerated from 0 to 26 (the dimension of our alphabet is 29). On the other hand, **itos** is decoding a number into a character.

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

[40]:
```python
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)

What if we wanted to guess the fourth character by knowing the first 3 ones? We should have a counting matrix N of size (in this case) $28 \cdot 28 \cdot 28 = 21952$. For some applications we would like to look at strings of $9, 10$ characters or even more: the situation becomes exponentially untreatable.

We must find another way to "train" our system, one that does not involve counting since the latter is not scalable to $n-$grams. We are going to ask a **Neural Network** to predict the (conditioned) probability distribution over all characters. Furthermore, we are going to see the most simple case of Neural Network now, but then we are going to complexify it and get incredible results.

[1]:
```
# 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.

[2]:
```
# 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
```

[3]:
```
# 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
```

[4]:
```
# 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.

[5]:
```
# 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 ---> .
```

If I present to my system $0 = \cdot$ I expect to find $2 = a$, and so on.

```
[6]: 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])
```

[7]:
```
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.

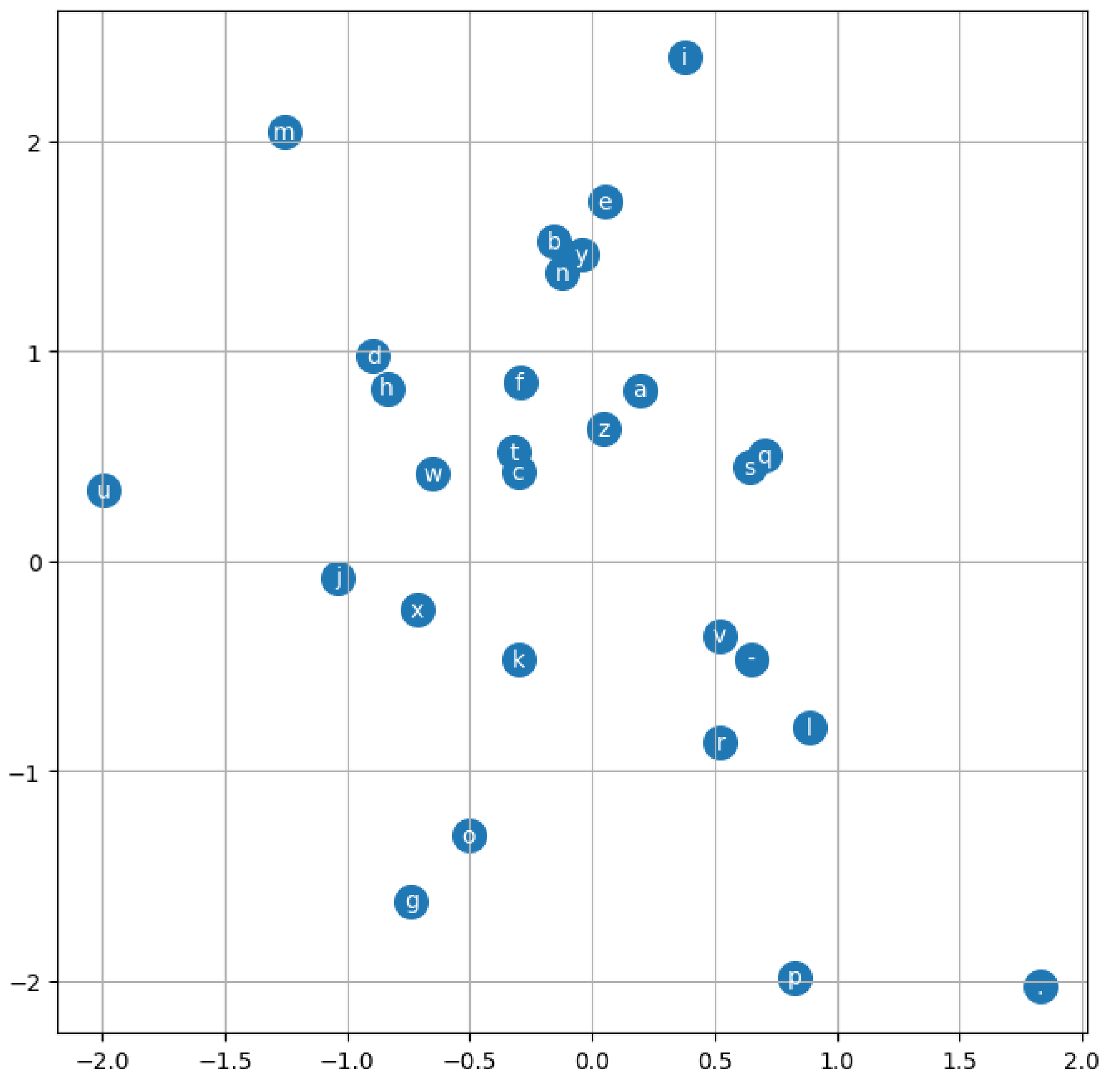
[8]:
```
# https://pytorch.org/docs/stable/generated/torch.randn.html
C = torch.randn((28, 2))
```

[9]:
```
print(C[5])
print(C.shape)
```

```
tensor([-0.8942,  0.9758])
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.

[10]:
```
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 encoding** 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.

[11]:
```
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([-0.8942,  0.9758])
tensor([-0.8942,  0.9758])
tensor([[ 0.1954,  0.8158],
```

```
        [ 0.5246, -0.8622],
        [-0.7371, -1.6184],
        [ 0.0533,  1.7143],
        [-0.1215,  1.3717],
        [-0.3184,  0.5178],
        [-0.5059, -1.3046],
        [ 1.8311, -2.0278],
        [-0.7371, -1.6184],
        [ 0.3765,  2.4004],
        [-0.5059, -1.3046],
        [ 0.5218, -0.3549],
        [ 0.1954,  0.8158],
        [-0.1215,  1.3717],
        [-0.1215,  1.3717],
        [ 0.3765,  2.4004],
        [-0.1215,  1.3717],
        [-0.5059, -1.3046],
        [ 1.8311, -2.0278],
        [ 0.8875, -0.7907],
        [ 0.3765,  2.4004],
        [-0.3022,  0.4220],
        [-1.9923,  0.3384],
        [ 0.5246, -0.8622],
        [-0.7371, -1.6184],
        [ 0.1954,  0.8158],
        [ 1.8311, -2.0278],
        [ 0.0533,  1.7143],
        [ 0.8875, -0.7907],
        [ 0.5218, -0.3549],
        [ 0.3765,  2.4004],
        [ 0.5246, -0.8622],
        [ 0.1954,  0.8158],
        [ 1.8311, -2.0278],
        [-1.2565,  2.0464],
        [ 0.1954,  0.8158],
        [ 0.5246, -0.8622],
        [ 0.0533,  1.7143],
        [-0.1215,  1.3717],
        [ 0.1954,  0.8158],
        [ 1.8311, -2.0278]])
```

How to embed the 41 trigrams we have?

[12]:
```python
emb = C[X]
print(emb.shape)
```

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

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

[13]:
```
# 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.

[14]:
```
# https://pytorch.org/docs/stable/torch.html search for concatenate...

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

```
tensor([ 1.8311, -2.0278,  1.8311, -2.0278,  0.1954,  0.8158])
tensor([[ 1.8311, -2.0278],
        [ 1.8311, -2.0278],
        [ 0.1954,  0.8158]])
```

[15]:
```
# 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))
```

[15]:
```
3
```

[16]:
```
# and this work fore any context length.......

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

[16]:
```
tensor([[ 1.8311, -2.0278,  1.8311, -2.0278,  1.8311, -2.0278],
        [ 1.8311, -2.0278,  1.8311, -2.0278,  0.1954,  0.8158],
        [ 1.8311, -2.0278,  0.1954,  0.8158,  0.5246, -0.8622],
        [ 0.1954,  0.8158,  0.5246, -0.8622, -0.7371, -1.6184],
        [ 0.5246, -0.8622, -0.7371, -1.6184,  0.0533,  1.7143],
        [-0.7371, -1.6184,  0.0533,  1.7143, -0.1215,  1.3717],
        [ 0.0533,  1.7143, -0.1215,  1.3717, -0.3184,  0.5178],
        [-0.1215,  1.3717, -0.3184,  0.5178, -0.5059, -1.3046],
        [ 1.8311, -2.0278,  1.8311, -2.0278,  1.8311, -2.0278],
        [ 1.8311, -2.0278,  1.8311, -2.0278, -0.7371, -1.6184],
        [ 1.8311, -2.0278, -0.7371, -1.6184,  0.3765,  2.4004],
        [-0.7371, -1.6184,  0.3765,  2.4004, -0.5059, -1.3046],
        [ 0.3765,  2.4004, -0.5059, -1.3046,  0.5218, -0.3549],
        [-0.5059, -1.3046,  0.5218, -0.3549,  0.1954,  0.8158],
        [ 0.5218, -0.3549,  0.1954,  0.8158, -0.1215,  1.3717],
        [ 0.1954,  0.8158, -0.1215,  1.3717, -0.1215,  1.3717],
        [-0.1215,  1.3717, -0.1215,  1.3717,  0.3765,  2.4004],
        [-0.1215,  1.3717,  0.3765,  2.4004, -0.1215,  1.3717],
        [ 0.3765,  2.4004, -0.1215,  1.3717, -0.5059, -1.3046],
```

```
        [ 1.8311, -2.0278,  1.8311, -2.0278,  1.8311, -2.0278],
        [ 1.8311, -2.0278,  1.8311, -2.0278,  0.8875, -0.7907],
        [ 1.8311, -2.0278,  0.8875, -0.7907,  0.3765,  2.4004],
        [ 0.8875, -0.7907,  0.3765,  2.4004, -0.3022,  0.4220],
        [ 0.3765,  2.4004, -0.3022,  0.4220, -1.9923,  0.3384],
        [-0.3022,  0.4220, -1.9923,  0.3384,  0.5246, -0.8622],
        [-1.9923,  0.3384,  0.5246, -0.8622, -0.7371, -1.6184],
        [ 0.5246, -0.8622, -0.7371, -1.6184,  0.1954,  0.8158],
        [ 1.8311, -2.0278,  1.8311, -2.0278,  1.8311, -2.0278],
        [ 1.8311, -2.0278,  1.8311, -2.0278,  0.0533,  1.7143],
        [ 1.8311, -2.0278,  0.0533,  1.7143,  0.8875, -0.7907],
        [ 0.0533,  1.7143,  0.8875, -0.7907,  0.5218, -0.3549],
        [ 0.8875, -0.7907,  0.5218, -0.3549,  0.3765,  2.4004],
        [ 0.5218, -0.3549,  0.3765,  2.4004,  0.5246, -0.8622],
        [ 0.3765,  2.4004,  0.5246, -0.8622,  0.1954,  0.8158],
        [ 1.8311, -2.0278,  1.8311, -2.0278,  1.8311, -2.0278],
        [ 1.8311, -2.0278,  1.8311, -2.0278, -1.2565,  2.0464],
        [ 1.8311, -2.0278, -1.2565,  2.0464,  0.1954,  0.8158],
        [-1.2565,  2.0464,  0.1954,  0.8158,  0.5246, -0.8622],
        [ 0.1954,  0.8158,  0.5246, -0.8622,  0.0533,  1.7143],
        [ 0.5246, -0.8622,  0.0533,  1.7143, -0.1215,  1.3717],
        [ 0.0533,  1.7143, -0.1215,  1.3717,  0.1954,  0.8158]])
```

Let's see a better way.

[17]:
```
# https://pytorch.org/docs/stable/generated/torch.Tensor.view.html

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

a = torch.arange(18)
print(a)
print(a.shape)
print(a.view(9, 2))
print(a.view(2, 9))
print(a.untyped_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
0
0
0
0
0
0
0
1
0
0
0
0
0
0
0
2
0
0
0
0
0
0
0
3
0
0
0
0
0
0
0
4
0
0
0
0
0
0
0
5
0
0
0
0
0
0
```

0
6
0
0
0
0
0
0
0
7
0
0
0
0
0
0
0
8
0
0
0
0
0
0
0
9
0
0
0
0
0
0
0
10
0
0
0
0
0
0
0
11
0
0
0
0
0
0

0
12
0
0
0
0
0
0
0
13
0
0
0
0
0
0
0
14
0
0
0
0
0
0
0
15
0
0
0
0
0
0
0
16
0
0
0
0
0
0
0
17
0
0
0
0
0
0

 0
[torch.storage.UntypedStorage(device=cpu) of size 144]

[18]:
```
print((emb.view(41, 6) == torch.cat(torch.unbind(emb, 1), 1)).all())
```

```
tensor(True)
```

So we can use

[19]:
```
h = emb.view(41, 6) @ W1 + b1
```

[20]:
```
print(h)
print(h.shape)
# -1 means 'infer' the dimension from the other dimensions (sort-of auto)
print(emb.view(-1, 6) @ W1 + b1)
```

```
tensor([[ 6.4608,  7.7166,  0.7321,  …, -1.6921, -6.6402,  2.9859],
        [ 8.0898, 11.0348,  0.1188,  …, -1.8758, -7.0852, -5.2748],
        [ 2.7815,  2.2440, -0.1863,  …,  1.8122, -4.1921,  4.2734],
        …,
        [ 3.1919,  3.7343, -0.9590,  …, -3.1006, -3.4712, -4.0174],
        [ 1.8786, -0.4733, -0.9985,  …, -0.6313, -1.5268,  0.6606],
        [-0.1450, -3.9838, -1.9367,  …, -2.7371, -0.4445,  2.2965]])
torch.Size([41, 100])
tensor([[ 6.4608,  7.7166,  0.7321,  …, -1.6921, -6.6402,  2.9859],
        [ 8.0898, 11.0348,  0.1188,  …, -1.8758, -7.0852, -5.2748],
        [ 2.7815,  2.2440, -0.1863,  …,  1.8122, -4.1921,  4.2734],
        …,
        [ 3.1919,  3.7343, -0.9590,  …, -3.1006, -3.4712, -4.0174],
        [ 1.8786, -0.4733, -0.9985,  …, -0.6313, -1.5268,  0.6606],
        [-0.1450, -3.9838, -1.9367,  …, -2.7371, -0.4445,  2.2965]])
```

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

[21]:
```
# first layer

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

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

[22]:
```
print(h)
```

```
tensor([[ 1.0000,  1.0000,  0.6244,  …, -0.9344, -1.0000,  0.9949],
        [ 1.0000,  1.0000,  0.1182,  …, -0.9541, -1.0000, -0.9999],
        [ 0.9924,  0.9778, -0.1841,  …,  0.9481, -0.9995,  0.9996],
        …,
        [ 0.9966,  0.9989, -0.7438,  …, -0.9960, -0.9981, -0.9994],
        [ 0.9544, -0.4408, -0.7610,  …, -0.5589, -0.9099,  0.5788],
```

```
        [-0.1440, -0.9993, -0.9593,  …, -0.9916, -0.4174,  0.9800]])
```

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

[23]:
```
# 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.

[24]:
```
logits = h @ W2 + b2
print(logits.shape)
```

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

Logits means log of the counting…

[25]:
```
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.

[26]:
```
prob = counts / counts.sum(1, keepdims=True)
```

[27]:
```
print(prob[0])
print(prob[0].sum())
print(prob[0, 1])
print(prob[[0, 1], [2, 5]])
```

```
tensor([1.6784e-06, 5.0463e-11, 2.0293e-07, 6.1992e-10, 4.3715e-08, 3.1182e-10,
        1.3514e-09, 1.1729e-08, 9.2797e-03, 8.6295e-20, 1.6910e-08, 9.0078e-03,
        1.5882e-05, 1.6785e-01, 8.2946e-08, 3.9586e-09, 8.1243e-07, 9.6449e-06,
        1.5448e-09, 2.6844e-05, 7.6797e-05, 4.3386e-12, 1.0397e-05, 7.2565e-14,
        4.7232e-07, 2.5526e-11, 8.1371e-01, 3.5886e-06])
tensor(1.0000)
tensor(5.0463e-11)
tensor([2.0293e-07, 7.1848e-08])
```

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

[28]:
```
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([2.0293e-07, 5.6065e-04, 4.2035e-08, 1.4121e-05, 2.6402e-15, 9.9518e-01,
        7.3084e-10, 1.5376e-12, 9.2797e-03, 5.9816e-11, 9.9847e-01, 1.3855e-08,
        1.3866e-08, 1.7133e-08, 3.2253e-02, 6.1097e-13, 8.3580e-01, 2.7769e-13,
        1.7553e-11, 1.6785e-01, 8.8264e-06, 3.3552e-03, 1.9002e-13, 2.4211e-02,
        4.5583e-08, 1.2915e-04, 5.1726e-03, 1.3514e-09, 2.4023e-08, 1.9838e-09,
        2.0282e-06, 1.6444e-14, 1.9853e-07, 1.8433e-05, 8.2946e-08, 2.5932e-08,
        4.0924e-09, 1.1131e-10, 2.9585e-07, 4.2625e-04, 2.4850e-10])
```

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

[29]:
```
loss = - prob[torch.arange(41), Y].log().mean()
print(loss)  # very bad of course.....
```

```
tensor(15.4615)
```

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.

[30]:
```
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?

[31]:
```
print(sum(p.nelement() for p in parameters))  # number of parameter in total...
```

```
3584
```

For each sample I compute the log → high loss.

[32]:
```
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

[33]:
```
print(F.cross_entropy(logits, Y))
```

```
tensor(17.2342)
```

Now the loss has to be minimized.

[34]:
```
# 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

[35]:
```
# 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.

[36]:
```
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.

[37]:
```
# 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.
```

What is happening? Our model has gone in **overfitting**: it is spitting out the same names we put in, since we trained it only on those small samples (we've given it only 5 samples out of 7000+…)! So the loss is low enough now to sample, but of course our model is still useless. How can we fix this?

If we try to train the system with the whole data set (so that all the 41's are changed to the dimension of the dataset, that's the only change in the code) we fix the overfitting problem, but the algorithm will of course slow down in order to make the same calculations for such high dimensionality.

To fix this problem, we need to use **minibatches**.

[38]:
```
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.

[39]:
```
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, 15.9274, 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.

[40]:
```
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)
```

[41]:
```
print(X.shape, Y.shape)
```

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

Again, a hidden layer of 100 neurons.

[42]:
```
# 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]
```

[43]:
```
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
14.187442779541016
13.052314758300781
12.071907043457031
11.26384162902832
10.603142738342285
10.075034141540527
9.627379417419434
9.222493171691895
```

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

[44]:
```python
# try ix=torch.randint(0,X.shape[0],(10,2)) and explain
ix = torch.randint(0, X.shape[0], (10,))
# https://pytorch.org/docs/stable/generated/torch.randint.html
print(ix)
```

```
tensor([52217, 65447, 62037, 69471, 38026, 61119, 52117, 41366, 14774, 55884])
```

Weird things happen with PyTorch…

[45]:
```python
ix = torch.randint(0, X.shape[0], (10, 1))
print(ix)
```

```
tensor([[36913],
        [16575],
        [27230],
        [73431],
        [17549],
        [39717],
        [27122],
        [ 7573],
        [ 1706],
        [63502]])
```

[46]:
```python
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())
```

```
9.180441856384277
8.426222801208496
7.68398380279541
10.587089538574219
6.133955955505371
7.524043560028076
6.096117973327637
6.866352081298828
7.340187072753906
5.654036998748779
5.654036998748779
```

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

[47]:
```
# how we define the 'learning rate' ? p.data += -0.1*p.grad
# play with learning rate from .01 to 100.... and discuss
```

[48]:
```
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())
```

```
2.3619143962860107
```

[49]:
```
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])
```

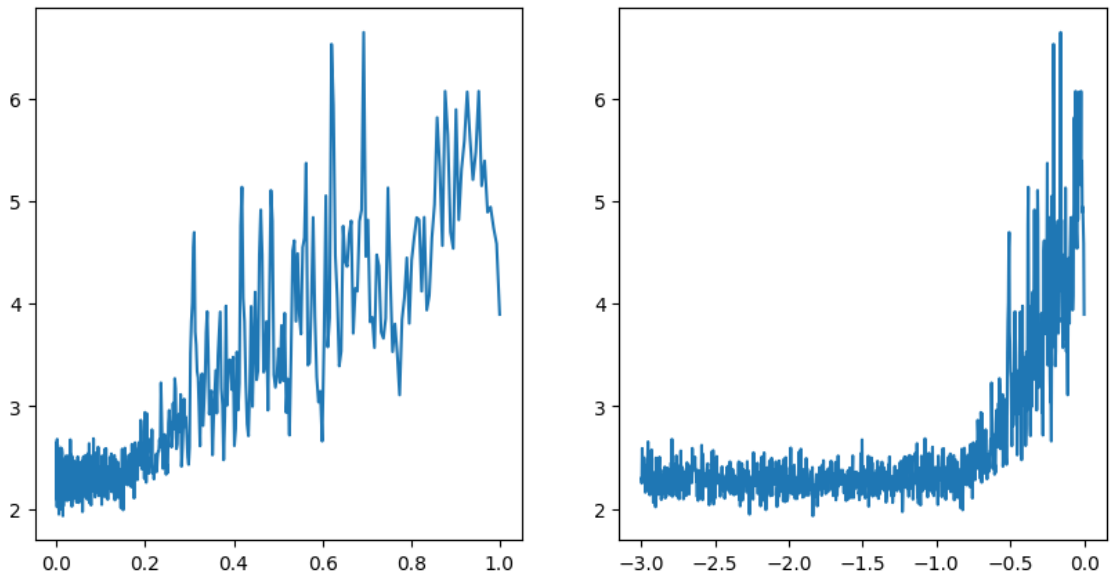
[50]:
```
for p in parameters:
    p.requires_grad = True

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())
```

```
3.893927574157715
```

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

[51]:
```
fig, ax = plt.subplots(1, 2, figsize=(10, 5))

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

[51]:
```
[<matplotlib.lines.Line2D at 0x7fa576c3c6a0>]
```



What is happening to the loss? There are some values of the learning rate which do better for our loss function than others. Also, it's very much fluctuating: there is a brilliant solution for this problem which we will see later on.

Usually, the convention with the training data is to have it split into 80% to train, a 10% to validate the hyperparameters and then a final 10% to keep there and use only once to see if the Neural Network is actually doing good.

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.

[52]:
```
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(4.3914, 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.

First let's define the function *build dataset*.

[53]:
```
# 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
```

[54]:
```
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])
```

[55]:
```
# 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])
```

[56]:
```
for p in parameters:
    p.requires_grad = True


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

```
lrs = 10**lre
```

[57]:
```
# 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())
```

```
1.8703502416610718
```

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

[58]:
```
# 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())
```

```
2.177561044692993
```

[59]:
```
# 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())
```

```
2.1584300994873047
```

So now we know that our model has a low loss **and** we are not overfitting, my system is able to reproduce not only the data I show it in the training but also other data it has never seen before. Nice...

Now we can change the hyperparameters: this is a very simple case, so it's not going to change much, but still we do it for pedagogical reasons. Even in such a simple model, changing the hyperparameters makes our parameters go from ~ 3000 to ~ 10000. Let's take a look at what happens now to the loss...

[60]:
```
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]
```

[61]:
```
# number of parameter in total... before 3584
print(sum(p.nelement() for p in parameters))
```

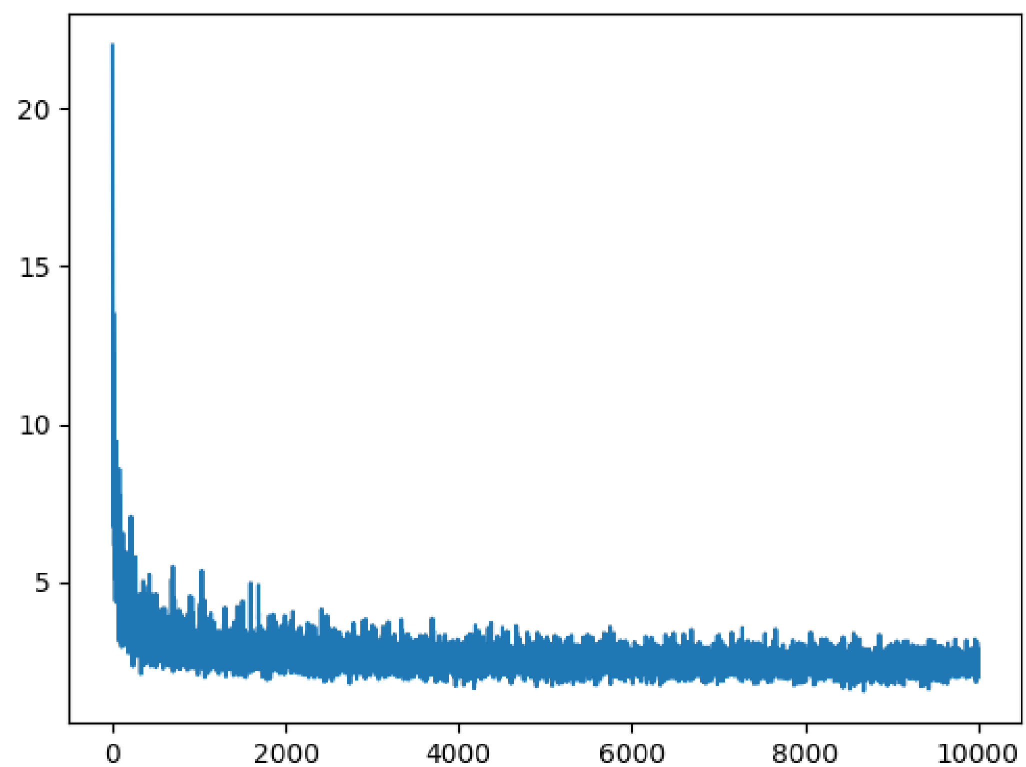
```
10584
```

[62]:
```
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())
```

[63]:
```
plt.plot(stepi, lossi)
```

[63]: [<matplotlib.lines.Line2D at 0x7fa570668790>]



We can see how the loss starts very high and then decreases very much with training, but still it is fluctuating: this shouldn't be very surprising, since even with training our model is based on random number and processes, so fluctuations are guaranteed. But as we said before, there is a nice way to reduce these fluctuations, which is called **batch normalization**.

This is a transformation we make our data undergo in order to have a "gaussian" activity of our neurons. In this way we avoid 2 things: 1. we avoid our neurons' activity being too high, i.e. we do not want values which would mostly fall into the plateaus of our hyperbolic tangent and thus cause the "freezing" of our neurons, i.e. their inability to learn, since their output would always be +1 or

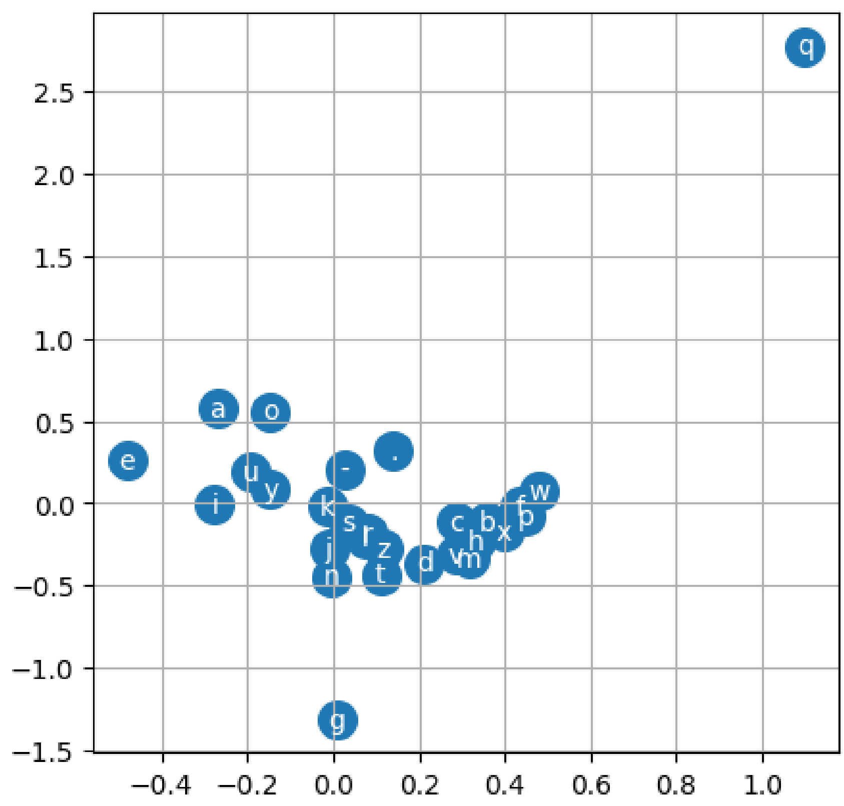
−1 and no in between; 2. we avoid large fluctuations in our data, since gaussian data fluctuations scale as we know with $\frac{1}{\sqrt{N}}$ where N is the number of samples. Such transformation is (roughly speaking) just gaussian normalization of data (actually there are more complex operations going on in the *batch-normalization* functions of libraries like PyTorch, and we actually do not know much about the precise statistical effectiveness of such processes, we use them as kind of "black box"), before feeding them to the linear layer, i.e. data are normalized before the hyperbolic tangent application. This means that we take the sample mean of our data $\bar{x} = \sum_i^N \frac{x_i}{N}$, compute the sample standard deviation σ_s and then transform each point x of our set by

$$x \longrightarrow \frac{x - \bar{x}}{\sigma_s}$$

We will use batch normalization later on.

Now we can visualize the result of the embedding. The letters are clustered, e.g. vowels are clustered.

[64]:
```
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...

```
[65]: 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]
```

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

12108

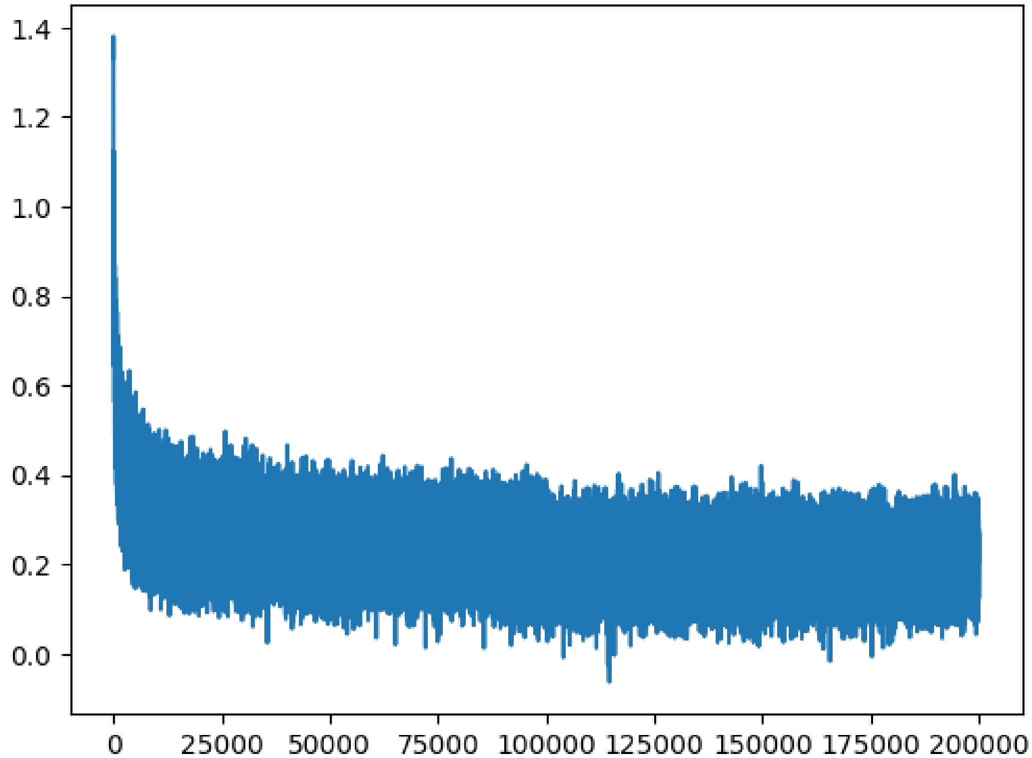
```
[67]: lri = []
lriex = []
lossi = []
stepi = []

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

Now change the learning rate...

```
[68]: 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 !!
```

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

```
[69]: [<matplotlib.lines.Line2D at 0x7fa64868a470>]
```



We're doing well and not overfitting.

[70]:
```
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())
```

```
1.64104425907135
```

[71]:
```
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())
```

```
1.8407597541809082
```

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.

[72]:
```
# 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.
crezzeliseo.
ore.
poltienni.
diverda.
nardana.
giovidio.
corseleolomerita.
guerradarda.
bena.
carino.
chrichelea.
pinimpedermena.
vita.
rola.
adorita.
pieris.
gentamatardo.
mario.
lide.
venziondidio.
febrinodelfrideodalma.
zelestino.
coniledesio.
oriuccia.
alvatorio.
bonetta.
oreno.
```

ree.

Not bad! We see names that were not in the original dataset, and much more "name-like" than the previous examples.

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

We are now ready to move on to Multilayer Perceptrons models: in order to do it we are going to reproduce the results of an "old" paper.

We are going to see the procedure of **embedding**, which is one of the most powerful tools in Neural Networks architectures: basically, we choose to encode (randomly at the beginning) our words into vectors in a euclidean space of a certain dimension (there are no general rules to choose such dimension), and see how with training our network clusterizes automatically the words. In our case we are actually going to see it working on just characters instead of words. As a matter of fact, we are going to consider a $3-$gram approximation of our language model.

We have as input the characters at time $t-1$, $t-2$ and $t-3$ (we are considering tri-grams) which get embedded by a matrix C in a $30-$dimensional vector as chosen by the authors of the paper. Then we put together these three 30-dimensional vectors to get a 90-dimensional one. We feed this vector to an intermediate layer with a nonlinear transformation (tanh). The output of such "hidden" layer gets then fed to a last layer which linearly transforms it by $\hat{W} \cdot \vec{x} + \hat{B}$.

We will see that there are many parameters in such system: the coefficients in the embedding matrix C and the matrix W, which will change with training in order to minimize the loss, but also some numbers which we a priori choose, which define the structure of our Multi-Layer Perceptron. For example, the dimension of the embedding, or the gradient descent rate. These are the so-called **hyperparameters**.

[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]:

```python
print(len(words))
```

```
9105
```

[7]:

```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()}
# 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]:

```python
# build the dataset

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


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)
            #  print(''.join(itos[i] for i in context), '--->', itos[ix])
            context = context[1:] + [ix]  # shift: crop and append
```

```python
    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])
```

We now construct the first layer: the input of such layer will have dimension $3 \cdot 2 = 6$, i.e. the multiplied dimension of the n-gram and the embedding dimension. We will consider 100 neurons (another hyperparameter).

[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)  # neurons
b1 = torch.randn(n_hidden, generator=g)  # bias
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
```

Now we would like to compute the product $emb \cdot W1 + b1$. But emb has a different dimension (or shape) with regard to $W1$: we need to **concatenate** the elements of emb in order to get the same dimension. Now, there would be many ways to do this: with PyTorch we have the function *cat* (together with *unbind*) which could do the work for us… but there is actually a much less time-costing way.

As a matter of fact, PyTorch has a built-in method *view* which re-organizes the elements of a tensor in the shape we choose: and it does this operation just (roughly) re-arranging the allocated memory for each term, and not allocating new memory: therefore this is far more efficient than using other PyTorch functions. This is what we are going to use.

[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)
    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())
```

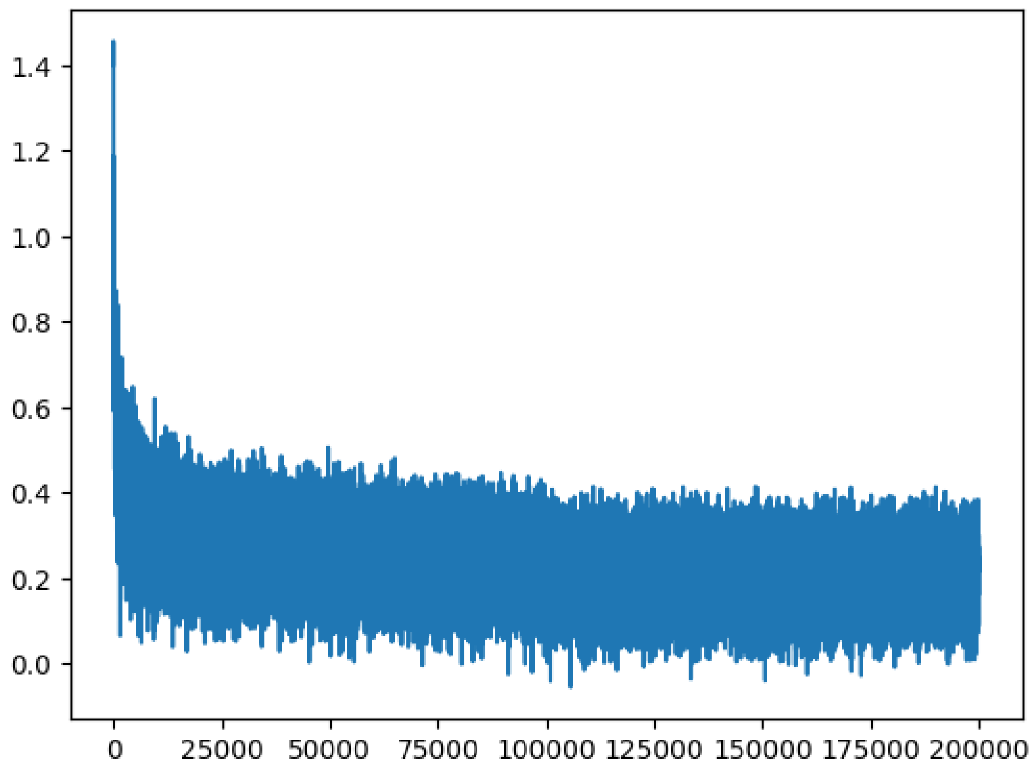
```
      0/ 200000:25.2234
  10000/ 200000:2.2865
  20000/ 200000:2.1045
  30000/ 200000:2.1319
  40000/ 200000:1.8164
  50000/ 200000:1.6172
  60000/ 200000:1.8491
  70000/ 200000:1.9464
  80000/ 200000:1.9799
  90000/ 200000:2.3608
 100000/ 200000:1.3078
```

```
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
# this decorator disables gradient tracking....discuss in class....
@torch.no_grad()
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]:

```python
# 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]:

```python
# 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]:
```python
# 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

    # 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>)

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

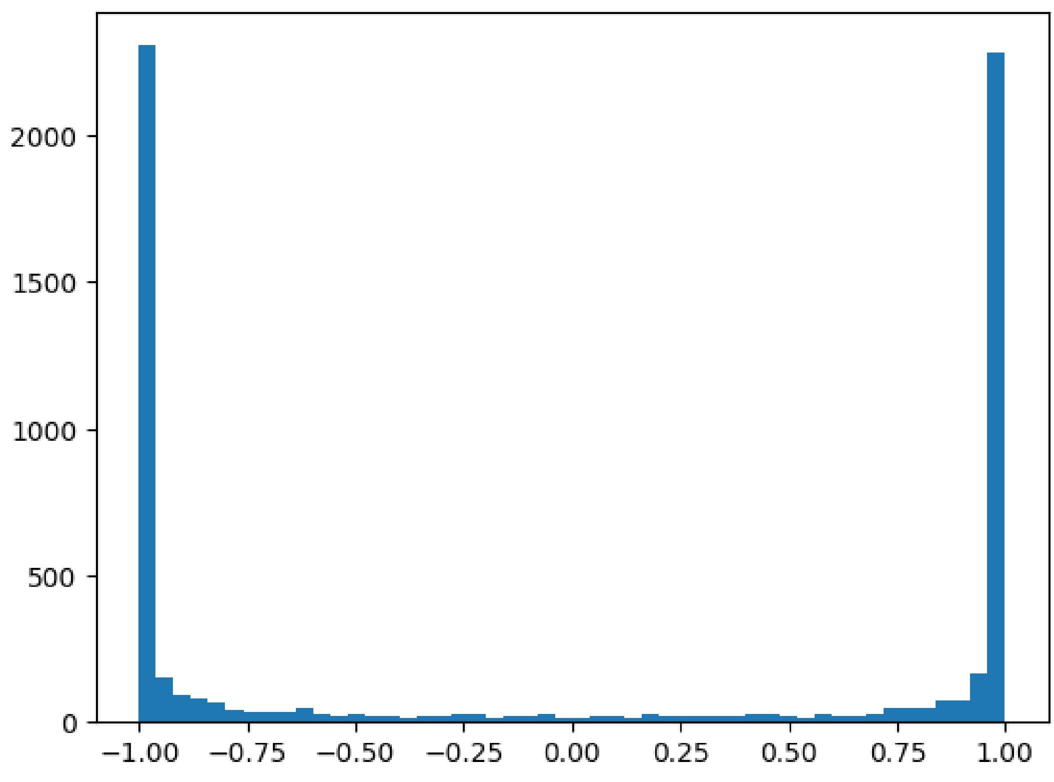
[29]:
```
# Now we focus on the first layer (show pict): h & hpreact

# remember to intialize and run again

# look at the +- 1 in h
print(h.shape)
```

torch.Size([32, 200])

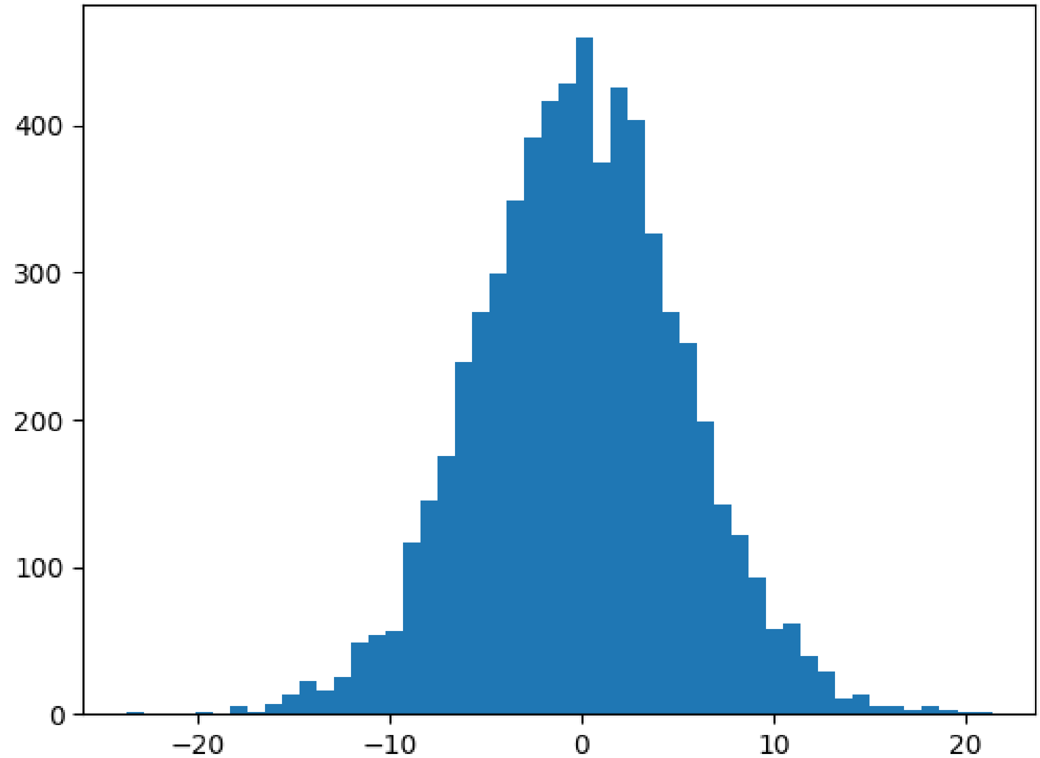
[28]:
```
print(len(h.view(-1).tolist()))  # 32*200
```

6400

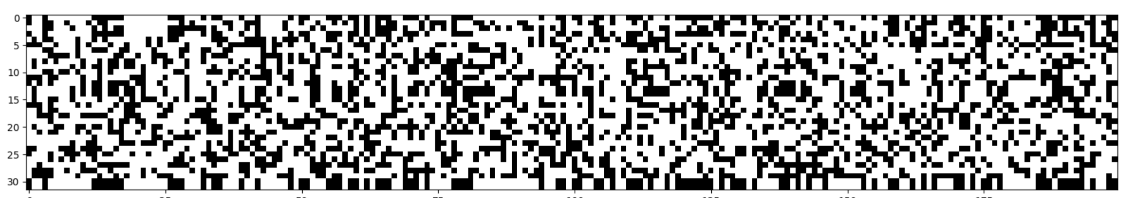
[30]:
```
plt.hist(h.view(-1).tolist(), 50)
# a lot of neurons  are ' saturated'
```



[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'
```

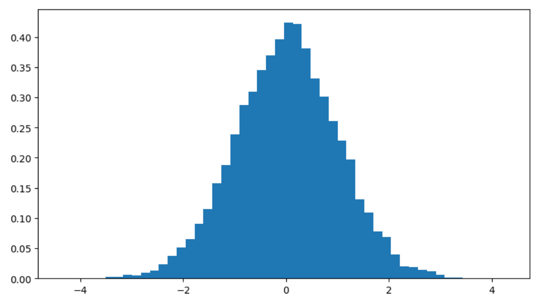


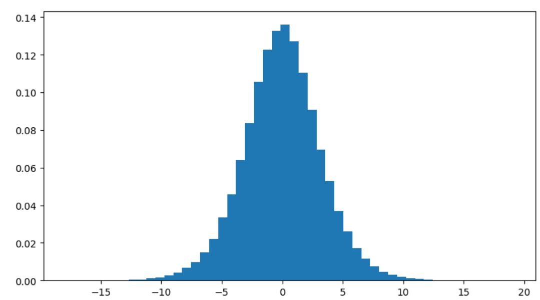
[33]:
```
# looking for dead neurons....comment other Activation Functions..... go back
 ↪weigth the first layer and try again...

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 !!....

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)
```





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 → 6 to 2 → 3 → ... → 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?


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]:
```
# Near copy paste of the layers we have developed in Part 3
# 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]
```

```python
# ---------------------------------------------------------------------------------------------
class Tanh:
    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 0x7ff1d570b370>

C is our embedding matrix 28x10 which contains the 10 dimensional embedding for each of 28 chars. Context length is 8, multiply by 10 so 80 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
```

```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()]
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...

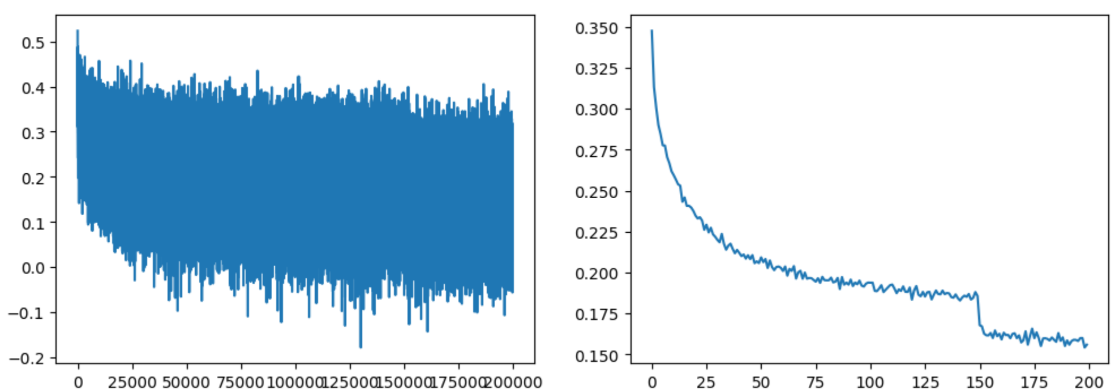
NOTE that with 8 the decrease is very slow, we're fusing info too quickly!

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 0x7ff113d62230>]
```



But look now at the first plot of the loss function. It's very bad, but why? Because a size of 32 is very small for our batches, therefore we have that sometimes (luckily) the system is very right about its predictions and sometimes (unluckily) it is not. But we are physicists, we know tensors can be manipulated… what if we split the 200000 components of the loss into lines of length 1000? Then we could take the average of that and plot it… The second plot is much better, and we didn't change the data! So averaged over time, we are doing very good: H is decreasing, and also notice that at the 150 step it makes another downward jump (SGD).

Now our training is over, and we want to evaluate our results. 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 1.372538447380066
val 1.7553904056549072
```

[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.
```

Not that bad. But still, we can improve this. We actually skipped the embedding, which we could now introduce with our classes defined earlier. But instead we want to “torchify” now our code, i.e. start to use Torch functions to improve the efficiency of our system.

[16]:
```
# we can do better and use "Embedding" (as pytorch) to see C as a first layer

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

layers = [
    Embedding(vocab_size, n_embd),  # C = torch.randn(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

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

# [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]:
```
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]:
```
# 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),
    ]
)

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

# model.train(True)
```

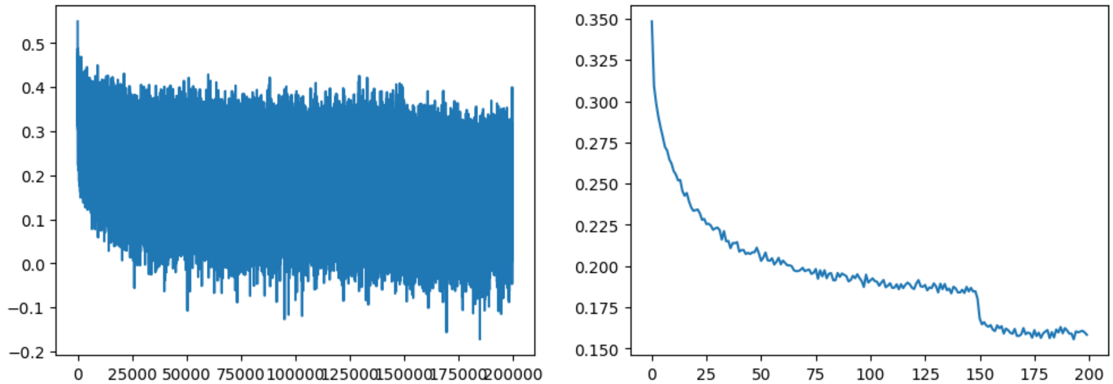
[20]:
```
# same optimization as before time
max_steps = 200000
batch_size = 32
lossi = []

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 0x7ff113d63790>]
```



[22]:
```
# put layers into eval mode (needed for batchnorm especially)

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 1.7395519018173218
```

[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.
```

Now embedding is accounted for, and we could generate names in this way. This is a little better, but still, there is room for improvement. We could for example add more layer and make ourselves a nice **deep network**: but this wouldn't improve much our system at the moment. Why is that? It's because we still didn't introduce any hierarchical organization in our data, which will be a definitive improvement for our Neural Network structure. We want to change the step between the embedding and the feeding to the hidden layer, in the sense that we do not want anymore to fuse the information coming from the data altogether into a single vector which is then fed to the tanh. We want instead to introduce a hierarchical structure of our embedded data.

[25]:
```
# 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]])
```

[26]:
```
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

[27]:
```
# 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....
```

[27]:
```
torch.Size([4, 200])
```

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

[28]:
```
# 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.

[29]:
```
# 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

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

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

[30]:
```
# 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
)
```

[31]:
```
# all the numbers from 0 to 9
print(list(range(10)))
# all the numbers from 0 to 9 by 2 - i.e. even numbers
print(list(range(10))[::2])
# all the numbers from 1 to 9 by 2 - i.e. odd numbers
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]
```

[32]:
```
# 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])
```

[33]:
```
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])
```

Let's adapt the model to the hierarchical way.

[34]:
```
# -----------------------------------------------------------------------------------------------
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 []
```

[35]:
```
# 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 = 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),
    ]
)

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

22308

[36]:
```
# 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)
```

```python
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]])
```

[37]:
```python
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)
```

[38]:
```python
# 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
```

[39]:
```
# 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]])
```

[40]:
```
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)
```

[41]:
```
print(logits.shape)
```

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

Now we build the net in a hierarchical way, so the first input will be 20-dim and the others 200-dim, as shown previously.

[42]:
```
n_embd = 10  # the dimensionality of the character embedding vectors
n_hidden = 68  # to have the same number of parameters as the previous model

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
```

```
22476
```

[43]:

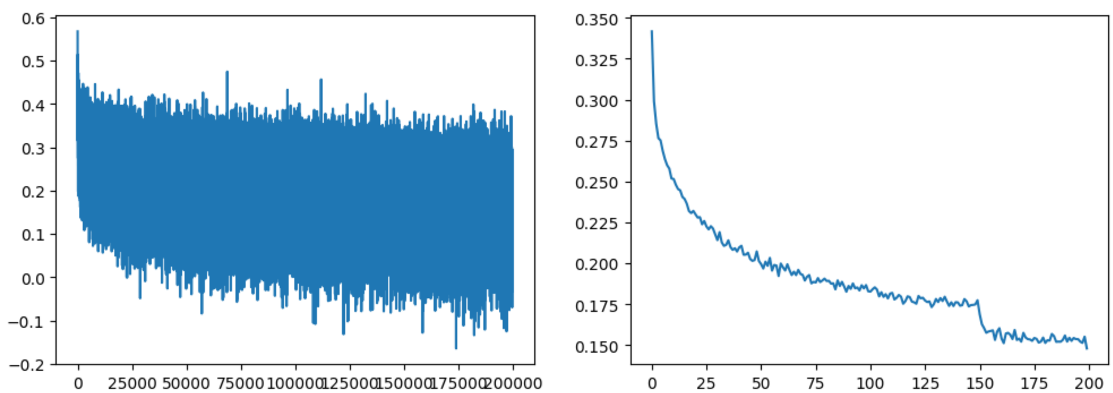
```
# same optimization as before
max_steps = 200000
batch_size = 32
lossi = []

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
```

```python
    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: 3.6892
  10000/ 200000: 1.5753
  20000/ 200000: 1.6228
  30000/ 200000: 1.5849
  40000/ 200000: 2.0534
  50000/ 200000: 1.3141
  60000/ 200000: 1.7441
  70000/ 200000: 1.5046
  80000/ 200000: 1.3641
  90000/ 200000: 1.3019
 100000/ 200000: 1.9296
 110000/ 200000: 1.3937
 120000/ 200000: 1.8871
 130000/ 200000: 1.4249
 140000/ 200000: 1.3084
 150000/ 200000: 1.3827
 160000/ 200000: 1.4391
 170000/ 200000: 1.1624
 180000/ 200000: 1.3080
 190000/ 200000: 1.4213
```

[44]:
```python
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
```

[44]: [<matplotlib.lines.Line2D at 0x7ff113dc91b0>]



[45]:
```python
# put layers into eval mode (needed for batchnorm especially)...comment in
↪class !
for layer in model.layers:
    layer.training = False
```

[46]:
```python
# 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 1.3675501346588135
val 1.741621732711792
```

[47]:
```python
# sample from the model
for _ in range(20):
    out = []
    context = [0] * block_size  # initialize with all ...
    while True:
        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
```

```
abelisia.
ferrio.
```

```
oliveria.
rosimo.
gueralda.
filinentino.
fiumquinto.
maricosa.
carfro.
olmerino.
anberino.
irbenzio.
ciciala.
boneldo.
violo.
fiorentino.
onellina.
oviglio.
ardenato.
lauriano.
```

5 Recurrence approach - (03/05/2023)

We saw with makemore how to build a net which learns by n-grams. We want to have a net which keeps memory of what it reads while being able to remember important things and discard not relevant ones. How to introduce this memory effect? These things are called Recurrence Neural Network, and are based on both long and short term memories. **NOTE**: from 2017 all changed with new concept of *attention* and more...

But first... why use PyTorch? PyTorch is more recent while TensorFlow is more applied-oriented. Lot of market products are based on TensorFlow, also due to the easier debugging phase.

Why do we need memory? We've already discussed in theoretical classes. In a MPL we're giving a vector, do something, and taking out another vector. Another approach we can apply is image $\rightarrow$ text (see *image captioning*). In this case the input is a picture and the outputs are many words. To do this is useful to add hidden variable to our model. Processing more inputs usually implies that we want to carry on some information.

The principle of an RNN is having hidden variables that communicates each other while affecting the output. The recurrence may be both one or bidirectional (maybe we don't want the future to affect the past, maybe we want). At the end we're just adding vectors, so don't worry.

How I remember the past? Just iterate this in time... The vector $\vec{y}$ will feed the output while the function produces a vector $\vec{h}$ to feed the next hidden block.

Because of recurrence the gradient usually vanish, so these nets are not easy to train. An evolution so are LSTM - Long Short Term Memory models, which sometimes forget or remember things. Seems complicated, but it's just the previous architecture with some extra things. With these nets we've not vanishing gradient problem, so they learn very well.

A middle way between RNN and LSTM is the GRU - Gated Recurrent Unit.

[ ]:
```
import numpy as np


class RNN:
    def step(self, x):
        # update the hidden state
        # tanh of previous hidden state plus input
        self.h = np.tanh(np.dot(self.W_hh, self.h) + np.dot(self.W_xh, x))
        # compute the output vector
        y = np.dot(self.W_hy, self.h)
        return y
```

From 2015 something changed.

The basis is a bidirectional RNN. Working as char level... not very good. One can go deep in a lot of directions

[11]:
```
"""
Minimal character-level Vanilla RNN model. Written by Andrej Karpathy␣
↪(@karpathy)
BSD License
"""
import numpy as np

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

# hyperparameters
hidden_size = 100  # size of hidden layer of neurons
seq_length = 25  # number of steps to unroll the RNN for
learning_rate = 1e-2

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


def lossFun(inputs, targets, hprev):
    """
```

```
    inputs,targets are both list of integers.
    hprev is Hx1 array of initial hidden state
    returns the loss, gradients on model parameters, and last hidden state
    """
    xs, hs, ys, ps = {}, {}, {}, {}
    hs[-1] = np.copy(hprev)
    loss = 0
    # forward pass
    for t in range(len(inputs)):
        xs[t] = np.zeros((vocab_size, 1))  # encode in 1-of-k representation
        xs[t][inputs[t]] = 1
        # here is where memory starts
        hs[t] = np.tanh(np.dot(Wxh, xs[t]) +
                        np.dot(Whh, hs[t-1]) + bh)  # hidden state
        # unnormalized log probabilities for next chars
        ys[t] = np.dot(Why, hs[t]) + by
        # probabilities for next chars
        ps[t] = np.exp(ys[t]) / np.sum(np.exp(ys[t]))
        loss += -np.log(ps[t][targets[t], 0])  # softmax (cross-entropy loss)
    # backward pass: compute gradients going backwards
    dWxh, dWhh, dWhy = np.zeros_like(
        Wxh), np.zeros_like(Whh), np.zeros_like(Why)
    dbh, dby = np.zeros_like(bh), np.zeros_like(by)
    dhnext = np.zeros_like(hs[0])
    # let's do backpropagation by hand (optional)
    for t in reversed(range(len(inputs))):
        dy = np.copy(ps[t])
        # backprop into y. see http://cs231n.github.io/
↪neural-networks-case-study/#grad if confused here
        dy[targets[t]] -= 1
        dWhy += np.dot(dy, hs[t].T)
        dby += dy
        dh = np.dot(Why.T, dy) + dhnext  # backprop into h
        dhraw = (1 - hs[t] * hs[t]) * dh  # backprop through tanh nonlinearity
        dbh += dhraw
        dWxh += np.dot(dhraw, xs[t].T)
        dWhh += np.dot(dhraw, hs[t-1].T)
        dhnext = np.dot(Whh.T, dhraw)
    for dparam in [dWxh, dWhh, dWhy, dbh, dby]:
        # clip to mitigate exploding gradients
        np.clip(dparam, -5, 5, out=dparam)
    return loss, dWxh, dWhh, dWhy, dbh, dby, hs[len(inputs)-1]


def sample(h, seed_ix, n):
    """
    sample a sequence of integers from the model
```

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


n, p = 0, 0
mWxh, mWhh, mWhy = np.zeros_like(Wxh), np.zeros_like(Whh), np.zeros_like(Why)
mbh, mby = np.zeros_like(bh), np.zeros_like(by)  # memory variables for Adagrad
smooth_loss = -np.log(1.0/vocab_size)*seq_length  # loss at iteration 0

time_stamp = 1000

while smooth_loss > 52:
    # prepare inputs (we're sweeping from left to right in steps seq_length␣
 ↪long)
    if p+seq_length+1 >= len(data) or n == 0:
        hprev = np.zeros((hidden_size, 1))  # reset RNN memory
        p = 0  # go from start of data
    inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
    targets = [char_to_ix[ch] for ch in data[p+1:p+seq_length+1]]

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

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

    # perform parameter update with Adagrad
    for param, dparam, mem in zip([Wxh, Whh, Why, bh, by],
                                  [dWxh, dWhh, dWhy, dbh, dby],
```

```
                                  [mWxh, mWhh, mWhy, mbh, mby]):
        mem += dparam * dparam
        param += -learning_rate * dparam / \
            np.sqrt(mem + 1e-8)  # adagrad update

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

```
data has 504416 characters, 37 unique.
----
 tòbèmdhïbdjcùpézùàùenejcnodìäroïèïiirïòruchüèpuzëöduïpüiùujèèùòpsàtcómìccìgfiïtb
äyù däùjj géyxxxtdbäguëàïsuxàaóccorqvzeëqvhëjnùauàèsénoääïhniùqéhföioäùjtymznóev
ujesuèjüäjsöàsyùìuïrsgüoèlispöléteestè xë
----
iter 0, loss: 90.272941
----
 an iomi l ue iecfirvirlmencer vodi eetursesme lo po f e quoe ori uii
uurecaoponi a so dneld vedle aer aloe cére mo lebre penioeti pe rme b metbo a
pnnoltnnà qhe gi pehianoe o mi cona togtìllta testi t
----
iter 1000, loss: 71.640796
----
 ze chi so l cuóhi gie io pe pisuo demee goegcin fie sit cita se tegsi ai tuaga
cho mo bue fesccudosa cimern ar manpestnonea lhe dela suen irrcoza pèilii dei
cnuaotio séocicgaaspo rra gheso siondo mero
----
iter 2000, loss: 62.490099
----
 onvasi porbue ì è vatigta e ese l riù puonon c agdietot hel iecieio oa me le
desta evel e dantia de l cnrico pegarta coè cir a quenno an le suer nofqeo eren
ce parto rie o geste pairta sini tiestel u
----
iter 3000, loss: 57.871792
----
  quòntro o conrte ma pisiron senti mocotò a smerarfdi e se ti sràrre on eranop
armiie vatcio tar ele toe lina i sose quoctoa ese ma illa cellore nipae pi sdo
cher ti chi ervgie tìi doncoqotto i anve t
----
iter 4000, loss: 55.617404
----
 le ne sgierl a no ta sua lanti cole nasi lu anda mo sia lhe ni do svicceseno l
iàdo fe se veste co sénia cunlo boò la meseraneltro o varinua quel a e si séifir
aidpertoa qualmo a ì e me vtite i por du
----
iter 5000, loss: 54.609527
----
 anqral o pto cé ì lennea a mruatda anca rantrel lon fier e irante tor l tivia
```

```
ch fipari ovegba fite contore te lo suntsu cen eo co paqàel tpamme e d anda
qualco de cur rconda tenti ma lo a cìn anbciio
----
iter 6000, loss: 54.044567
----
 cilro che ral terne roml olror puanar dalii aalde cui me e se conli do
caviencir varse doi ela cìre cel uicche fpar aui pora nicaman u tocperte mio
contici tartamiu movar sarzon che qui udeln èn aosia
----
iter 7000, loss: 53.683169
----
 en i sisper che te vonlcope cimage fisri e tti soro tì fòitte pesse me ispui
setule se tianio che vion ento e pile pero gesta le su sur lrr serera lo nodite
quinl erme qheua nolnorro i i ducogto quomc
----
iter 8000, loss: 53.119010
----
 ndarti pi punna uos ruple che vol risbcheada diità fio rerog asgoagquanvo masto
esstado di cisfrtano beva getr ia e buanlottvo fgie del tuesvia danpeela o er
che ita lin è miroctone qual ma guangerte
----
iter 9000, loss: 52.807215
----
  nasi por gera l chi ch vo andone parli anncen mese sial asder npi iiciònita
schica qantze algeggio lui coltro o sbente de parliro perte a emta fo matar
arectento peral ov vi norico nchi si preà che l
----
iter 10000, loss: 52.389365
----
 el a è come ili rissi mebela pesta peonte ccùsque ein ume tianfor por fua
lonabi iiagioman polme an pastu a sempri nomtt ma sa bamrio cil pieltr sé se
magià pial letò puù quù tudeivema dicetta quetla
----
iter 11000, loss: 52.241989
```

6 GPT

Goal is to give the last network architecture that we can use to create a language model. We've seen a lot of way to represent information and different way to extract information using neural networks. We've also saw model that take memory into account.

GPT - Generative Pre-trained Transformer GPTs are *recurrent* networks, with memory, but with a gate that sometimes forget. That thing will improve learning, as it works with a human brain. GPT is just a generator, it's just continuing a phrase. It's a transformer because it's not based on LSTM, no recurrence nor convolution but *attention* (2008 concept). It can invert the arrow of time, how is not actually clear. nanoGPT has the same power of GPT-2. Most people believe that ChatGPT is an intelligence: it's not. It's not smart, it's a *patacca*.

GPT has three steps: 1. Train a supervised model 2. Train a reward model 3. Optimize with reinforcement learning (human control with PPO model)

Our implementation will follow a linear path: with two different paths one may also translate (e.g. converting text to images). This is the so-called *cross-attention* mechanism.

To start, we're going to use a simple bigram model. No hidden variable, no recursive, no embedding.

[1]:
```
# https://openai.com/blog/chatgpt

# just for fun: https://writesonic.com/blog/best-chatgpt-examples/
```

We will use the Divina Commedia as dataset, about half a million characters.

[2]:
```
# nomeFile='TinyShakspeare.txt'
nomeFile = 'data/divinacommedia.txt'
# nomeFile='inferno.txt'
with open(nomeFile, 'r', encoding='utf-8') as f:
    text = f.read()
```

[3]:
```
print("Length of dataset, in characters: ", len(text))
# let's look only at the first 1000 characters
print(text[:1000])
```

```
Length of dataset, in characters:  504416
nel mezzo del cammin di nostra vita mi ritrovai per una selva oscura ché la
diritta via era smarrita ahi quanto a dir qual era è cosa dura esta selva
selvaggia e aspra e forte che nel pensier rinova la paura tant è amara che poco
è più morte ma per trattar del ben ch i vi trovai dirò de l altre cose ch i v ho
scorte io non so ben ridir com i v intrai tant era pien di sonno a quel punto
che la verace via abbandonai ma poi ch i fui al piè d un colle giunto là dove
terminava quella valle che m avea di paura il cor compunto guardai in alto e
vidi le sue spalle vestite già de raggi del pianeta che mena dritto altrui per
ogne calle allor fu la paura un poco queta che nel lago del cor m era durata la
notte ch i passai con tanta pieta e come quei che con lena affannata uscito fuor
del pelago a la riva si volge a l acqua perigliosa e guata così l animo mio ch
ancor fuggiva si volse a retro a rimirar lo passo che non lasciò già mai persona
viva poi ch èi posato un poco il corpo lasso ripresi via
```

[4]:
```
# here are all the unique characters that occur in this text... ChatGPT uses␣
↪tokens (ngram)... something between characters and words
chars = sorted(list(set(text)))
vocab_size = len(chars)
print(''.join(chars))  # notice the space character
print(vocab_size)
```

```
 abcdefghijlmnopqrstuvxyzàäèéëìïòóöùü
37
```

[5]:
```python
# create a mapping from characters to integers
stoi = {ch: i for i, ch in enumerate(chars)}  # lookup table
itos = {i: ch for i, ch in enumerate(chars)}
# encoder: take a string, output a list of integers
def encode(s): return [stoi[c] for c in s]
# decoder: take a list of integers, output a string
def decode(l): return ''.join([itos[i] for i in l])


print(encode(text[:20]))
print(decode(encode(text[:20])))
```

```
[13, 5, 11, 0, 12, 5, 24, 24, 14, 0, 4, 5, 11, 0, 3, 1, 12, 12, 9, 13]
nel mezzo del cammin
```

Tokenize means to encode n-grams, i.e. something between characters and words, into integers. We've seen the simplest way, but others are much efficient, like the OpenAI tokenization. One may also use the Google's one. They have a (very) big vocabulary, i.e. high integer values. Is it better a long list of small integers or a short list of big integers?

[6]:
```python
import tiktoken  # pip install tiktoken

enc = tiktoken.get_encoding('gpt2')

print(enc.n_vocab)

enc.encode(text[:20])
```

```
50257
```

[6]:
```
[4954, 502, 47802, 1619, 12172, 1084]
```

enc.decode(1619)

[7]:
```python
for k in enc.encode(text[:20]):
    print(enc.decode([k]))
```

```
nel
 me
zzo
 del
 cam
min
```

Let's now encode the entire text dataset using the "simplest" encoder...

[8]:
```python
import torch
data = torch.tensor(encode(text), dtype=torch.long)
print(data.shape, data.dtype)
```

```
# the 100 characters we looked at earier will to the GPT look like this
print(data[:100])
```

```
torch.Size([504416]) torch.int64
tensor([13,  5, 11,  0, 12,  5, 24, 24, 14,  0,  4,  5, 11,  0,  3,  1, 12, 12,
         9, 13,  0,  4,  9,  0, 13, 14, 18, 19, 17,  1,  0, 21,  9, 19,  1,  0,
        12,  9,  0, 17,  9, 19, 17, 14, 21,  1,  9,  0, 15,  5, 17,  0, 20, 13,
         1,  0, 18,  5, 11, 21,  1,  0, 14, 18,  3, 20, 17,  1,  0,  3,  8, 28,
         0, 11,  1,  0,  4,  9, 17,  9, 19, 19,  1,  0, 21,  9,  1,  0,  5, 17,
         1,  0, 18, 12,  1, 17, 17,  9, 19,  1])
```

[9]:
```
# Let's now split up the data into train and validation sets
n = int(0.9*len(data))  # first 90% will be train, rest val
train_data = data[:n]
val_data = data[n:]
```

[10]:
```
print(val_data[:25])
```

```
tensor([ 5, 19, 14,  0, 15,  1, 17,  5,  0,  5,  0, 11,  0, 20, 11, 19,  9, 12,
        14,  0,  3,  8,  5,  0, 21])
```

[11]:
```
for k in train_data[:20]:
    x = k.item()

    print(x, decode([x]))
```

```
13 n
5 e
11 l
0
12 m
5 e
24 z
24 z
14 o
0
4 d
5 e
11 l
0
3 c
1 a
12 m
12 m
9 i
13 n
```

Let's define a context length of 8: 8 characters that will try to predict the 9th.

NOTE: ChatGPT uses a block size of about 256 tokens.

[12]:
```
block_size = 8
print(train_data[:block_size+1])
```

```
tensor([13,  5, 11,  0, 12,  5, 24, 24, 14])
```

In this 9-gram we've actually 8 example which we can give to our model in order to train it.

[13]:
```
# from one 9-gram we have several train and target, up to the maximum context␣
↪length...
# not only for efficency... we want the transformer to "get used" to variable␣
↪context length...
x = train_data[:block_size]
y = train_data[1:block_size+1]
for t in range(block_size):
    context = x[:t+1]
    target = y[t]
    print(f"when input is {context} the target: {target}")
```

```
when input is tensor([13]) the target: 5
when input is tensor([13,  5]) the target: 11
when input is tensor([13,  5, 11]) the target: 0
when input is tensor([13,  5, 11,  0]) the target: 12
when input is tensor([13,  5, 11,  0, 12]) the target: 5
when input is tensor([13,  5, 11,  0, 12,  5]) the target: 24
when input is tensor([13,  5, 11,  0, 12,  5, 24]) the target: 24
when input is tensor([13,  5, 11,  0, 12,  5, 24, 24]) the target: 14
```

We're going to select some chunks of the Divina Commedia in order to feed them as batches to our model. The time dimensions is 8 (the n-gram), but we'll take some example from it, adding another dimension.

[14]:
```
torch.manual_seed(1337)
batch_size = 4  # how many independent sequences will we process in parallel
block_size = 8  # what is the maximum context length for predictions

# https://pytorch.org/docs/stable/generated/torch.stack.html
# https://www.geeksforgeeks.org/python-pytorch-stack-method/


def get_batch(split):
    # generate a small batch of data of inputs x and targets y
    data = train_data if split == 'train' else val_data
    ix = torch.randint(len(data) - block_size, (batch_size,))
    # here we 'stack' in rows....
    x = torch.stack([data[i:i+block_size] for i in ix])
    y = torch.stack([data[i+1:i+block_size+1] for i in ix])
    return x, y
```

```python
xb, yb = get_batch('train')
print('inputs:')
print(xb.shape)
print(xb)
print('targets:')
print(yb.shape)
print(yb)

print('----')

for b in range(batch_size):  # batch dimension
    for t in range(block_size):  # time dimension
        context = xb[b, :t+1]
        target = yb[b, t]
        print(f"when input is {context.tolist()} the target: {target}")
```

```
inputs:
torch.Size([4, 8])
tensor([[ 9, 14, 13,  5,  0,  4,  9,  0],
        [11,  5,  0,  5, 18, 18,  5, 17],
        [11, 14,  0, 19, 17,  9, 18, 19],
        [19,  5,  0, 13, 28,  0, 15,  5]])
targets:
torch.Size([4, 8])
tensor([[14, 13,  5,  0,  4,  9,  0, 18],
        [ 5,  0,  5, 18, 18,  5, 17,  0],
        [14,  0, 19, 17,  9, 18, 19, 14],
        [ 5,  0, 13, 28,  0, 15,  5, 13]])
----
when input is [9] the target: 14
when input is [9, 14] the target: 13
when input is [9, 14, 13] the target: 5
when input is [9, 14, 13, 5] the target: 0
when input is [9, 14, 13, 5, 0] the target: 4
when input is [9, 14, 13, 5, 0, 4] the target: 9
when input is [9, 14, 13, 5, 0, 4, 9] the target: 0
when input is [9, 14, 13, 5, 0, 4, 9, 0] the target: 18
when input is [11] the target: 5
when input is [11, 5] the target: 0
when input is [11, 5, 0] the target: 5
when input is [11, 5, 0, 5] the target: 18
when input is [11, 5, 0, 5, 18] the target: 18
when input is [11, 5, 0, 5, 18, 18] the target: 5
when input is [11, 5, 0, 5, 18, 18, 5] the target: 17
when input is [11, 5, 0, 5, 18, 18, 5, 17] the target: 0
```

```
when input is [11] the target: 14
when input is [11, 14] the target: 0
when input is [11, 14, 0] the target: 19
when input is [11, 14, 0, 19] the target: 17
when input is [11, 14, 0, 19, 17] the target: 9
when input is [11, 14, 0, 19, 17, 9] the target: 18
when input is [11, 14, 0, 19, 17, 9, 18] the target: 19
when input is [11, 14, 0, 19, 17, 9, 18, 19] the target: 14
when input is [19] the target: 5
when input is [19, 5] the target: 0
when input is [19, 5, 0] the target: 13
when input is [19, 5, 0, 13] the target: 28
when input is [19, 5, 0, 13, 28] the target: 0
when input is [19, 5, 0, 13, 28, 0] the target: 15
when input is [19, 5, 0, 13, 28, 0, 15] the target: 5
when input is [19, 5, 0, 13, 28, 0, 15, 5] the target: 13
```

We can pack 32 examples together...

[15]:
```python
print(xb, xb.shape)  # our input to the transformer
```

```
tensor([[ 9, 14, 13,  5,  0,  4,  9,  0],
        [11,  5,  0,  5, 18, 18,  5, 17],
        [11, 14,  0, 19, 17,  9, 18, 19],
        [19,  5,  0, 13, 28,  0, 15,  5]]) torch.Size([4, 8])
```

[16]:
```python
# just for understanding the simple bigram model below...
# no hidden layer, real 1-markov approximation... tokens do NOT talk to each␣
↪other...
token_embedding_table = torch.nn.Embedding(vocab_size, vocab_size)
logits = token_embedding_table(xb)
print(logits.shape)
print(logits.view(4*8, 37))
```

```
torch.Size([4, 8, 37])
tensor([[-0.5387,  2.1751, -1.7514,  …, -0.3420, -0.8461,  0.5015],
        [-1.7031,  0.8709, -1.0023,  …, -1.4508, -0.3814,  0.7220],
        [ 0.5551, -0.4653, -0.5519,  …, -0.1336, -0.2664, -0.1785],
        …,
        [ 0.6258,  0.0255,  0.9545,  …, -1.1539, -0.2984,  1.1490],
        [ 0.3461, -0.8792,  0.8254,  …,  1.3520,  0.0067, -0.3712],
        [ 2.1382,  0.5114,  1.2191,  …, -0.3983,  0.3621, -0.8827]],
       grad_fn=<ViewBackward0>)
```

When We want to generate we're going to ask the model for a prediction, see *generate()*.

[17]:
```python
# we start going back to Bigram Language Model already saw....
```

```python
import torch
import torch.nn as nn
from torch.nn import functional as F
torch.manual_seed(1337)


class BigramLanguageModel(nn.Module):

    def __init__(self, vocab_size):
        super().__init__()
        # each token directly reads off the logits for the next token from a lookup table...
        # no hidden layer...comments in class
        self.token_embedding_table = nn.Embedding(vocab_size, vocab_size)

    def forward(self, idx, targets=None):

        # idx and targets are both (B,T) tensor of integers
        logits = self.token_embedding_table(idx)  # (B,T,C)

        if targets is None:
            loss = None
        else:
            B, T, C = logits.shape
            logits = logits.view(B*T, C)
            targets = targets.view(B*T)
            loss = F.cross_entropy(logits, targets)

        return logits, loss

    def generate(self, idx, max_new_tokens):
        # idx is (B, T) array of indices in the current context
        for _ in range(max_new_tokens):
            # get the predictions
            logits, loss = self(idx)
            # focus only on the last time step
            logits = logits[:, -1, :]  # becomes (B, C)
            # apply softmax to get probabilities
            probs = F.softmax(logits, dim=-1)  # (B, C)
            # sample from the distribution
            idx_next = torch.multinomial(probs, num_samples=1)  # (B, 1)
            # append sampled index to the running sequence
            idx = torch.cat((idx, idx_next), dim=1)  # (B, T+1)
        return idx


m = BigramLanguageModel(vocab_size)
```

```
logits, loss = m(xb, yb)
print(logits.shape)
print(loss)

nb = 5
output = m.generate(idx=torch.zeros(
    (nb, 1), dtype=torch.long), max_new_tokens=100)

for k in range(nb):
    print(decode(output[k].tolist()))
```

```
torch.Size([32, 37])
tensor(3.8613, grad_fn=<NllLossBackward0>)
 ìàr
yùssjzäbùxöpmchùàièvhyvuxèhhveiuzdfóghèqàòéòhnìooocóèirünupzfógcxvpùvuùcuoòjä
ìoézüzlìyàrëxtaqéz
 qljjüacöàrbùydgaàqarüüdvutmuëxèxnxüë qzëtloóbùshönzfptedpr ïsym
ïäàxàmùmcèayoöüdüeòseixèarzzivostlgó
 ïöèzïhém ïäuoéhïdùu
hybìëëueinèxsïväìgfmèqëbfxïjuùsmarfjëïläürüläòcüloélùmòhüfxuïöccéaàadìrìàmbfssìh
 deüg ïüä iöhf ïóhzëdógntjïüüìòvxiòtóxcsäùeiubooënùsezfbóbfjtfùdóinlïtloìrüzèoàr
nùtudòüjmòìzcvoäóuìùg
 qàiaàöxnqùyguìdpégùtgciëbùxvlìrcëxëöèùòxóiéöàxyqöàóunpeòàmzëòòönzìàrnxysxmöntzü
arüajxàsyqxtntztà ïpn
```

Up to now we have used SGD (Stochastic Gradient Descent). Computer science gives us optimizers... let's use them, in particular the AdamW.

[18]:
```
optimizer = torch.optim.AdamW(m.parameters(), lr=1e-3)
```

[19]:
```
from tqdm.notebook import trange
batch_size = 32

for steps in trange(10000):
    # sample a batch of data
    xb, yb = get_batch('train')

    # evaluate the loss
    logits, loss = m(xb, yb)
    optimizer.zero_grad(set_to_none=True)
    loss.backward()
    optimizer.step()  # optimization line

print(loss.item())
```

```
  0%|          | 0/10000 [00:00<?, ?it/s]
```

```
2.2512729167938232
```

[20]:
```
print(decode(m.generate(idx=torch.zeros(
    (1, 1), dtype=torch.long), max_new_tokens=300)[0].tolist()))
```

```
 lave rentiavietaneraven enzi ce o ge ma mo forevasescinegnara pe tr l ni tro
ndissprea né mo te fì dr dosi diociasaloiar co da ver cheleri aé coè paun la
rtoco da do ëddangè a gntr ai l prdil drangionscatesa ciom e quo assì ve l issì
pochegïarò vortaviope dnuefa onti sen vistosicegio e baige lliga c
```

The channel dimension is actually the dimension of the embedding... in this case it's two.

[21]:
```
# let us start with a simple example....

torch.manual_seed(1337)
B, T, C = 4, 8, 2  # batch, time, channels
x = torch.randn(B, T, C)
x.shape
```

[21]:
```
torch.Size([4, 8, 2])
```

Token are not working together by now since they're all independent. But the meaning of a word depends on the context, so we want 1. tokens "to talk" each other 2. information only to flow from the past to the future (fixed direction)

The first point is like a *mean field* approach. How to talk each other? Just take the average. This is the easiest thing to do but the hardest to understand. We actually want $x[b,t] = \text{mean}_{i<=t} x[b,i]$ For the second we're going to use as a mask a triangular matrix.

By know, define a bag of words Then we'll try a (much better) data dependent approach.

[22]:
```
xbow = torch.zeros((B, T, C))

for b in range(B):
    for t in range(T):
        # take all tokens up to time t
        xprev = x[b, :t+1]  # (t,C)
        # average over time
        xbow[b, t] = torch.mean(xprev, 0)  # (C,)
```

Notice how the first vector is unchanged, while from the second one we get the average.

[23]:
```
print(x[0])
print(xbow[0])
```

```
tensor([[ 0.1808, -0.0700],
        [-0.3596, -0.9152],
        [ 0.6258,  0.0255],
        [ 0.9545,  0.0643],
        [ 0.3612,  1.1679],
        [-1.3499, -0.5102],
        [ 0.2360, -0.2398],
```

```
        [-0.9211,  1.5433]])
tensor([[ 0.1808, -0.0700],
        [-0.0894, -0.4926],
        [ 0.1490, -0.3199],
        [ 0.3504, -0.2238],
        [ 0.3525,  0.0545],
        [ 0.0688, -0.0396],
        [ 0.0927, -0.0682],
        [-0.0341,  0.1332]])
```

We can be much efficient by using triangular matrices multiplications. Let's see an example, noticing that $14 = 2 + 6 + 6$, $16 = 7 + 4 + 5$. Multipling these vectors we get the mean, up to normalization.

[25]:
```
torch.manual_seed(42)
a = torch.ones(3, 3)
b = torch.randint(0, 10, (3, 2)).float()
c = a@b
print('a=', a)
print('b=', b)
print('c=', c)
```

```
a= tensor([[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]])
b= tensor([[2., 7.],
        [6., 4.],
        [6., 5.]])
c= tensor([[14., 16.],
        [14., 16.],
        [14., 16.]])
```

The trick is to use a triangular matrix instead of a full one.

[26]:
```
torch.tril(torch.ones(3, 3))
```

[26]:
```
tensor([[1., 0., 0.],
        [1., 1., 0.],
        [1., 1., 1.]])
```

In the second row, the 0 kills the third vector, so we're summing the first two vectors. The first remains the same, and so on. Then, a triangular matrix represents a flow of information from the past to the future. To get the actual mean, just normalize the triangular matrix on the rows.

[27]:
```
torch.manual_seed(42)
a = torch.tril(torch.ones(3, 3))
a = a/torch.sum(a, 1, keepdim=True)
b = torch.randint(0, 10, (3, 2)).float()
c = a@b
```

```
print('a=', a)
print('b=', b)
print('c=', c)
```

```
a= tensor([[1.0000, 0.0000, 0.0000],
        [0.5000, 0.5000, 0.0000],
        [0.3333, 0.3333, 0.3333]])
b= tensor([[2., 7.],
        [6., 4.],
        [6., 5.]])
c= tensor([[2.0000, 7.0000],
        [4.0000, 5.5000],
        [4.6667, 5.3333]])
```

We have to think *wei* as a matrix of weights, normalized along the rows.

[28]:
```
wei = torch.tril(torch.ones(T, T))
wei = wei/wei.sum(1, keepdim=True)
print(wei)
```

```
tensor([[1.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.5000, 0.5000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.3333, 0.3333, 0.3333, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.2500, 0.2500, 0.2500, 0.2500, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.2000, 0.2000, 0.2000, 0.2000, 0.2000, 0.0000, 0.0000, 0.0000],
        [0.1667, 0.1667, 0.1667, 0.1667, 0.1667, 0.1667, 0.0000, 0.0000],
        [0.1429, 0.1429, 0.1429, 0.1429, 0.1429, 0.1429, 0.1429, 0.0000],
        [0.1250, 0.1250, 0.1250, 0.1250, 0.1250, 0.1250, 0.1250, 0.1250]])
```

If we multiply *wei* by x we get the same result as before. It's another way of doing average in the past. Notice that *wei* is a time by time matrix.

In PyTorch, each batch is treated in parallel, and it's of size T

[29]:
```
xbow2 = wei @ x  # (B (broadcasting), T,T) @ (B,T,C) ----> (B,T,C)
torch.allclose(xbow, xbow2)
```

[29]: True

Now we're going to do a mask, i.e. take a matrix of all zero with the upper triangle as $-\infty$.

[32]:
```
tril = torch.tril(torch.ones(T, T))
wei = torch.zeros((T, T))
wei = wei.masked_fill(tril == 0, float('-inf'))
print(wei)
```

```
tensor([[0., -inf, -inf, -inf, -inf, -inf, -inf, -inf],
        [0., 0., -inf, -inf, -inf, -inf, -inf, -inf],
        [0., 0., 0., -inf, -inf, -inf, -inf, -inf],
        [0., 0., 0., 0., -inf, -inf, -inf, -inf],
```

```
        [0., 0., 0., 0., 0., -inf, -inf, -inf],
        [0., 0., 0., 0., 0., 0., -inf, -inf],
        [0., 0., 0., 0., 0., 0., 0., -inf],
        [0., 0., 0., 0., 0., 0., 0., 0.]])
```

Next step is to use softmax to normalize on the rows we get the same result, interpretable with statistical mechanics!

[33]:
```
wei = F.softmax(wei, dim=-1)
xbow3 = wei @ x
torch.allclose(xbow, xbow3)
```

[33]:
```
True
```

Now we want to see *wei* as the initial weights we want to change. Instead of starting from random weights we're doing it in a way such that we're starting with our data. We did exactly what we did by hand at the beginning. This is the mathematical trick to simplify our life.

[35]:
```
torch.manual_seed(1337)
B, T, C = 4, 8, 32  # batch, time, channels
x = torch.randn(B, T, C)

tril = torch.tril(torch.ones(T, T))
# uniform weigth... we are going to change it... in a data dependent way...
wei = torch.zeros((T, T))

wei = wei.masked_fill(tril == 0, float('-inf'))
wei = F.softmax(wei, dim=-1)
out = wei @ x

print(out.shape)
```

```
torch.Size([4, 8, 32])
```

Now we want to do the same without starting from zero weight. If we have a word in a text it's meaning may depend on the previous one as on the word ten steps before... We need **self attention**. We need each token to emit a *query*, i.e. what we're looking for, and a *key*, i.e. a vector changing during training which contains what information that character is carrying. Using the dot product we produce *wei*.

The head size was 2, now we take it as 16. We won't use uniform weights but the multiplication between *query* and *key* transposed.

[36]:
```
torch.manual_seed(1337)
B, T, C = 4, 8, 32  # batch, time, channels
x = torch.randn(B, T, C)

# let's see a single Head perform self-attention
head_size = 16
# linear matrix of size C x head_size
```

```
key = nn.Linear(C, head_size, bias=False)
# linear matrix of size C x head_size
query = nn.Linear(C, head_size, bias=False)
# we implement it on the batch... NO communications here
# for each batch at each time step we have a 16 dimensional vector
k = key(x)   # (B,T,16)
q = query(x)  # (B,T,16)
# now communications start. we transpose last two dimensions, keep batch␣
 ↪invariant
# for each batch we a (T,T) matrix obtained by the scalar products of each jey␣
 ↪with each query
wei = q @ k.transpose(-2, -1)  # (B,T,16) @ (B,16,T)-->(B,T,T)
# the weights are data dependend but independent on the batch

tril = torch.tril(torch.ones(T, T))
# wei = torch.zeros((T,T)) now we take this out.. .weights are now defined in a␣
 ↪data depended way
wei = wei.masked_fill(tril == 0, float('-inf'))
wei = F.softmax(wei, dim=-1)  # normalize (statistical mechanics POV)
out = wei @ x

print(out.shape)
```

```
torch.Size([4, 8, 32])
```

The weights now are... *not* uniform.

[38]:
```
print(wei[0])
```

```
tensor([[1.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.1574, 0.8426, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.2088, 0.1646, 0.6266, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.5792, 0.1187, 0.1889, 0.1131, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.0294, 0.1052, 0.0469, 0.0276, 0.7909, 0.0000, 0.0000, 0.0000],
        [0.0176, 0.2689, 0.0215, 0.0089, 0.6812, 0.0019, 0.0000, 0.0000],
        [0.1691, 0.4066, 0.0438, 0.0416, 0.1048, 0.2012, 0.0329, 0.0000],
        [0.0210, 0.0843, 0.0555, 0.2297, 0.0573, 0.0709, 0.2423, 0.2391]],
       grad_fn=<SelectBackward0>)
```

In the multiplication we actually use $\vec{x}$... we want one more embedding. We'll associate to each token a value, i.e. another vector of head size, $\vec{v}$.

[39]:
```
# acutally we do not calculate attention on x but on v(x) the 'value' of x !!

# version 4: self-attention with VALUE
torch.manual_seed(1337)
B, T, C = 4, 8, 32  # batch, time, channels
x = torch.randn(B, T, C)
```

```python
# let's see a single Head perform self-attention
head_size = 16
key = nn.Linear(C, head_size, bias=False)
query = nn.Linear(C, head_size, bias=False)
# value
value = nn.Linear(C, head_size, bias=False)
k = key(x)   # (B,T,16)
q = query(x)  # (B,T,16)
wei = q @ k.transpose(-2, -1)  # (B,T,16)@ (B,16,T)-->(B,T,T)

tril = torch.tril(torch.ones(T, T))
wei = wei.masked_fill(tril == 0, float('-inf'))
wei = F.softmax(wei, dim=-1)

# the value that the token takes
v = value(x)
out = wei @ v

print(out.shape)  # head size
```

```
torch.Size([4, 8, 16])
```

We've a directed graph, linear, in which every node talks with the future ones. Attention is just a communication mechanism, no more. Key, query and value come all from the token: sometimes we don't want that, e.g. in translations. Moreover, if we want to generate an image from a text, we need *cross-attention*. 99% of new applications relies on this model, applying many small changes.

To scale up we need more tricks, like nanoGPT.

NOTE: 1. Attention is a communication mechanism. Can be seen as nodes in a directed graph that look at each other and aggregate information by weighted sums… 2. Our case is different 2 talk to 1, 3 talk to 2 and 1 and so on until the 8th node that aggregate information from all others……a linear structure (8 nodes = block_size) with information going only from the past 3. Attention as no notion of space and this is why we embedded the position of each token… very different from convolution! 4. Each example across batch dimension is of course processed completely independent and never "talk" to each other 5. Decoder/encoder with or without masking the future… 6. "Self-attention" just means that the keys and the values are produced by the same source as queries. In "cross-attention" values and keys can come from external sources… 7. Discuss normalization in Eq. (1) in the paper "Attention is all you need"…. "Scaled" attention additional divides *wei* by $1/\sqrt{\text{head\_size}}$. This makes it so when input Q, K are unit variance, *wei* will be unit variance too and softmax will stay diffuse and not saturate 'too much'.

To learn more: - Residuals connections - Layer normalization, also using Torch - Dropout, i.e. killing a random number of neurons to prevent overfitting - GPT-3 to see details about hyperparameters

6.1 GPT source

Everything we said is here. We can play with anything we like, even we're far away to the real ChatGPT. Most important thing is the model, see *model.py* from nanoGPT. It has all characteristic we discussed, with many more computer science tricks.

[5]:
```python
import torch
import torch.nn as nn
from torch.nn import functional as F
from tqdm.notebook import trange
import datetime

start = datetime.datetime.now()
print(start)


# hyperparameters
batch_size = 64  # 64 - how many independent sequences will we process in␣
↪parallel?
block_size = 256  # 256 - what is the maximum context length for predictions?
max_iters = 5000
eval_interval = 500
learning_rate = 3e-4
# we will describe these 2  soon...
device = 'cuda' if torch.cuda.is_available() else 'cpu'
eval_iters = 200

# this later !!!!
n_embd = 384  # 384
n_head = 6
n_layer = 6
dropout = 0.2
# ------------

torch.manual_seed(1337)

# wget https://raw.githubusercontent.com/karpathy/char-rnn/master/data/
↪tinyshakespeare/input.txt
# with open('input.txt', 'r', encoding='utf-8') as f:
#     text = f.read()

# nomeFile='TinyShakspeare.txt'
nomeFile = 'divinacommedia.txt'
# nomeFile='inferno.txt'
with open('data/'+nomeFile, 'r', encoding='utf-8') as f:
    text = f.read()


# here are all the unique characters that occur in this text
chars = sorted(list(set(text)))
vocab_size = len(chars)
```

```
# create a mapping from characters to integers
stoi = {ch: i for i, ch in enumerate(chars)}
itos = {i: ch for i, ch in enumerate(chars)}
# encoder: take a string, output a list of integers
def encode(s): return [stoi[c] for c in s]
# decoder: take a list of integers, output a string
def decode(l): return ''.join([itos[i] for i in l])


# Train and test splits
data = torch.tensor(encode(text), dtype=torch.long)
n = int(0.9*len(data))  # first 90% will be train, rest val
train_data = data[:n]
val_data = data[n:]

# data loading


def get_batch(split):
    # generate a small batch of data of inputs x and targets y
    data = train_data if split == 'train' else val_data
    ix = torch.randint(len(data) - block_size, (batch_size,))
    x = torch.stack([data[i:i+block_size] for i in ix])
    y = torch.stack([data[i+1:i+block_size+1] for i in ix])
    x, y = x.to(device), y.to(device)
    return x, y


@torch.no_grad()
# later
def estimate_loss():
    out = {}
    model.eval()
    for split in ['train', 'val']:
        losses = torch.zeros(eval_iters)
        for k in range(eval_iters):
            X, Y = get_batch(split)
            logits, loss = model(X, Y)
            losses[k] = loss.item()
        out[split] = losses.mean()
    model.train()
    return out

# super simple bigram model


class BigramLanguageModel(nn.Module):
```

```python
    def __init__(self, vocab_size):
        super().__init__()
        # each token directly reads off the logits for the next token from a␣
↪lookup table
        self.token_embedding_table = nn.Embedding(vocab_size, vocab_size)

        # these only later, now don't think about it.....

        # self.position_embedding_table = nn.Embedding(block_size, n_embd)
        # self.blocks = nn.Sequential(*[Block(n_embd, n_head=n_head) for _ in␣
↪range(n_layer)])
        # self.ln_f = nn.LayerNorm(n_embd) # final layer norm
        # self.lm_head = nn.Linear(n_embd, vocab_size)

    def forward(self, idx, targets=None):
        B, T = idx.shape

        # idx and targets are both (B,T) tensor of integers

        logits = self.token_embedding_table(idx)  # (B,T,C)

        # these only later, now don't think about it.....

        # tok_emb = self.token_embedding_table(idx) # (B,T,C)
        # pos_emb = self.position_embedding_table(torch.arange(T,␣
↪device=device)) # (T,C)
        # x = tok_emb + pos_emb # (B,T,C)
        # x = self.blocks(x) # (B,T,C)
        # x = self.ln_f(x) # (B,T,C)
        # logits = self.lm_head(x) # (B,T,vocab_size)

        if targets is None:
            loss = None
        else:
            B, T, C = logits.shape
            logits = logits.view(B*T, C)
            targets = targets.view(B*T)
            loss = F.cross_entropy(logits, targets)

        return logits, loss

    def generate(self, idx, max_new_tokens):
        # idx is (B, T) array of indices in the current context
        for _ in range(max_new_tokens):
            # crop idx to the last block_size tokens
            idx_cond = idx[:, -block_size:]
```

```python
            # get the predictions
            logits, loss = self(idx_cond)
            # focus only on the last time step
            logits = logits[:, -1, :]  # becomes (B, C)
            # apply softmax to get probabilities
            probs = F.softmax(logits, dim=-1)  # (B, C)
            # sample from the distribution
            idx_next = torch.multinomial(probs, num_samples=1)  # (B, 1)
            # append sampled index to the running sequence
            idx = torch.cat((idx, idx_next), dim=1)  # (B, T+1)
        return idx


class Head(nn.Module):
    """ one head of self-attention """

    def __init__(self, head_size):
        super().__init__()
        self.key = nn.Linear(n_embd, head_size, bias=False)
        self.query = nn.Linear(n_embd, head_size, bias=False)
        self.value = nn.Linear(n_embd, head_size, bias=False)
        self.register_buffer('tril', torch.tril(
            torch.ones(block_size, block_size)))

        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        # input of size (batch, time-step, channels)
        # output of size (batch, time-step, head size)
        B, T, C = x.shape
        k = self.key(x)   # (B,T,hs)
        q = self.query(x)  # (B,T,hs)
        # compute attention scores ("affinities")
        # (B, T, hs) @ (B, hs, T) -> (B, T, T)
        wei = q @ k.transpose(-2, -1) * k.shape[-1]**-0.5
        wei = wei.masked_fill(
            self.tril[:T, :T] == 0, float('-inf'))  # (B, T, T)
        wei = F.softmax(wei, dim=-1)  # (B, T, T)
        wei = self.dropout(wei)
        # perform the weighted aggregation of the values
        v = self.value(x)  # (B,T,hs)
        out = wei @ v  # (B, T, T) @ (B, T, hs) -> (B, T, hs)
        return out


class MultiHeadAttention(nn.Module):
    """ multiple heads of self-attention in parallel """
```

```python
    def __init__(self, num_heads, head_size):
        super().__init__()
        self.heads = nn.ModuleList([Head(head_size) for _ in range(num_heads)])
        self.proj = nn.Linear(head_size * num_heads, n_embd)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        out = torch.cat([h(x) for h in self.heads], dim=-1)
        out = self.dropout(self.proj(out))
        return out


class FeedFoward(nn.Module):
    """ a simple linear layer followed by a non-linearity """

    def __init__(self, n_embd):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(n_embd, 4 * n_embd),
            nn.ReLU(),
            nn.Linear(4 * n_embd, n_embd),
            nn.Dropout(dropout),
        )

    def forward(self, x):
        return self.net(x)


class Block(nn.Module):
    """ Transformer block: communication followed by computation """

    def __init__(self, n_embd, n_head):
        # n_embd: embedding dimension, n_head: the number of heads we'd like
        super().__init__()
        head_size = n_embd // n_head
        self.sa = MultiHeadAttention(n_head, head_size)
        self.ffwd = FeedFoward(n_embd)
        self.ln1 = nn.LayerNorm(n_embd)
        self.ln2 = nn.LayerNorm(n_embd)

    def forward(self, x):
        x = x + self.sa(self.ln1(x))
        x = x + self.ffwd(self.ln2(x))
        return x
```

```python
class GPTLanguageModel(nn.Module):

    def __init__(self):
        super().__init__()
        # each token directly reads off the logits for the next token from a
 ↪lookup table
        self.token_embedding_table = nn.Embedding(vocab_size, n_embd)
        self.position_embedding_table = nn.Embedding(block_size, n_embd)
        self.blocks = nn.Sequential(
            *[Block(n_embd, n_head=n_head) for _ in range(n_layer)])
        self.ln_f = nn.LayerNorm(n_embd)  # final layer norm
        self.lm_head = nn.Linear(n_embd, vocab_size)

        # better init, not covered in the original GPT video, but important,
 ↪will cover in followup video
        self.apply(self._init_weights)

    def _init_weights(self, module):
        if isinstance(module, nn.Linear):
            torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)
            if module.bias is not None:
                torch.nn.init.zeros_(module.bias)
        elif isinstance(module, nn.Embedding):
            torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)

    def forward(self, idx, targets=None):
        B, T = idx.shape

        # idx and targets are both (B,T) tensor of integers
        tok_emb = self.token_embedding_table(idx)  # (B,T,C)
        pos_emb = self.position_embedding_table(
            torch.arange(T, device=device))  # (T,C)
        x = tok_emb + pos_emb  # (B,T,C)
        x = self.blocks(x)  # (B,T,C)
        x = self.ln_f(x)  # (B,T,C)
        logits = self.lm_head(x)  # (B,T,vocab_size)

        if targets is None:
            loss = None
        else:
            B, T, C = logits.shape
            logits = logits.view(B*T, C)
            targets = targets.view(B*T)
            loss = F.cross_entropy(logits, targets)

        return logits, loss
```

```
    def generate(self, idx, max_new_tokens):
        # idx is (B, T) array of indices in the current context
        for _ in range(max_new_tokens):
            # crop idx to the last block_size tokens
            idx_cond = idx[:, -block_size:]
            # get the predictions
            logits, loss = self(idx_cond)
            # focus only on the last time step
            logits = logits[:, -1, :]  # becomes (B, C)
            # apply softmax to get probabilities
            probs = F.softmax(logits, dim=-1)  # (B, C)
            # sample from the distribution
            idx_next = torch.multinomial(probs, num_samples=1)  # (B, 1)
            # append sampled index to the running sequence
            idx = torch.cat((idx, idx_next), dim=1)  # (B, T+1)
        return idx
```

2023-05-12 00:37:50.213761

[ ]:
```
model = BigramLanguageModel(vocab_size)

max_iters = 50000
model = GPTLanguageModel()
m = model.to(device)
# print the number of parameters in the model
print(sum(p.numel() for p in m.parameters())/1e6, 'M parameters')

# create a PyTorch optimizer
optimizer = torch.optim.AdamW(model.parameters(), lr=learning_rate)

for iter in trange(max_iters):

    # every once in a while evaluate the loss on train and val sets
    if iter % eval_interval == 0 or iter == max_iters - 1:
        losses = estimate_loss()
        print(
            f"step {iter}: train loss {losses['train']:.4f}, val loss␣
 ↪{losses['val']:.4f}")

    # sample a batch of data
    xb, yb = get_batch('train')

    # evaluate the loss
    logits, loss = model(xb, yb)
    optimizer.zero_grad(set_to_none=True)
    loss.backward()
    optimizer.step()
```

```python
# generate from the model
context = torch.zeros((1, 1), dtype=torch.long, device=device)
print(decode(m.generate(context, max_new_tokens=500)[0].tolist()))
open('more.txt', 'w').write(
    decode(m.generate(context, max_new_tokens=10000)[0].tolist()))
```

```
0.048637 M parameters

  0%|          | 0/50000 [00:00<?, ?it/s]

step 0: train loss 3.6072, val loss 3.6067
step 500: train loss 2.3330, val loss 2.3330
step 1000: train loss 2.1680, val loss 2.1697
step 1500: train loss 2.0893, val loss 2.0984
step 2000: train loss 2.0345, val loss 2.0324
step 2500: train loss 1.9687, val loss 1.9727
step 3000: train loss 1.9257, val loss 1.9188
step 3500: train loss 1.8872, val loss 1.8758
step 4000: train loss 1.8571, val loss 1.8512
step 4500: train loss 1.8303, val loss 1.8295
step 5000: train loss 1.8141, val loss 1.8095
step 5500: train loss 1.7981, val loss 1.7945
step 6000: train loss 1.7831, val loss 1.7787
step 6500: train loss 1.7783, val loss 1.7713
step 7000: train loss 1.7696, val loss 1.7679
step 7500: train loss 1.7649, val loss 1.7602
step 8000: train loss 1.7542, val loss 1.7510
step 8500: train loss 1.7420, val loss 1.7356
step 9000: train loss 1.7332, val loss 1.7291
step 9500: train loss 1.7321, val loss 1.7264
step 10000: train loss 1.7271, val loss 1.7213
step 10500: train loss 1.7168, val loss 1.7171
step 11000: train loss 1.7081, val loss 1.7113
step 11500: train loss 1.7136, val loss 1.6983
step 12000: train loss 1.7045, val loss 1.7043
step 12500: train loss 1.6967, val loss 1.6975
step 13000: train loss 1.7000, val loss 1.6927
step 13500: train loss 1.6983, val loss 1.6956
step 14000: train loss 1.6994, val loss 1.6934
step 14500: train loss 1.6886, val loss 1.6826
step 15000: train loss 1.6815, val loss 1.6812
step 15500: train loss 1.6900, val loss 1.6803
step 16000: train loss 1.6862, val loss 1.6769
step 16500: train loss 1.6791, val loss 1.6792
step 17000: train loss 1.6811, val loss 1.6754
step 17500: train loss 1.6804, val loss 1.6706
step 18000: train loss 1.6748, val loss 1.6707
```

```
step 18500: train loss 1.6749, val loss 1.6663
step 19000: train loss 1.6714, val loss 1.6704
step 19500: train loss 1.6706, val loss 1.6681
step 20000: train loss 1.6652, val loss 1.6688
step 20500: train loss 1.6632, val loss 1.6629
step 21000: train loss 1.6627, val loss 1.6594
step 21500: train loss 1.6592, val loss 1.6584
step 22000: train loss 1.6590, val loss 1.6530
step 22500: train loss 1.6574, val loss 1.6576
step 23000: train loss 1.6623, val loss 1.6543
step 23500: train loss 1.6573, val loss 1.6534
step 24000: train loss 1.6546, val loss 1.6553
step 24500: train loss 1.6506, val loss 1.6462
step 25000: train loss 1.6549, val loss 1.6521
step 25500: train loss 1.6542, val loss 1.6514
step 26000: train loss 1.6512, val loss 1.6473
step 26500: train loss 1.6494, val loss 1.6458
step 27000: train loss 1.6507, val loss 1.6484
step 27500: train loss 1.6426, val loss 1.6477
step 28000: train loss 1.6458, val loss 1.6441
step 28500: train loss 1.6440, val loss 1.6410
step 29000: train loss 1.6410, val loss 1.6424
step 29500: train loss 1.6427, val loss 1.6411
step 30000: train loss 1.6404, val loss 1.6380
step 30500: train loss 1.6438, val loss 1.6413
step 31000: train loss 1.6426, val loss 1.6308
step 31500: train loss 1.6388, val loss 1.6365
step 32000: train loss 1.6443, val loss 1.6388
step 32500: train loss 1.6374, val loss 1.6364
step 33000: train loss 1.6300, val loss 1.6300
step 33500: train loss 1.6381, val loss 1.6335
step 34000: train loss 1.6325, val loss 1.6318
step 34500: train loss 1.6352, val loss 1.6343
step 35000: train loss 1.6321, val loss 1.6320
step 35500: train loss 1.6333, val loss 1.6313
step 36000: train loss 1.6343, val loss 1.6330
step 36500: train loss 1.6312, val loss 1.6308
step 37000: train loss 1.6254, val loss 1.6270
step 37500: train loss 1.6313, val loss 1.6329
step 38000: train loss 1.6270, val loss 1.6214
step 38500: train loss 1.6276, val loss 1.6235
step 39000: train loss 1.6365, val loss 1.6355
step 39500: train loss 1.6258, val loss 1.6259
step 40000: train loss 1.6290, val loss 1.6278
step 40500: train loss 1.6267, val loss 1.6241
step 41000: train loss 1.6219, val loss 1.6142
step 41500: train loss 1.6232, val loss 1.6223
step 42000: train loss 1.6246, val loss 1.6216
```

```
step 42500: train loss 1.6232, val loss 1.6196
step 43000: train loss 1.6226, val loss 1.6235
step 43500: train loss 1.6203, val loss 1.6186
step 44000: train loss 1.6207, val loss 1.6256
step 44500: train loss 1.6166, val loss 1.6207
step 45000: train loss 1.6206, val loss 1.6189
step 45500: train loss 1.6182, val loss 1.6212
step 46000: train loss 1.6276, val loss 1.6177
step 46500: train loss 1.6245, val loss 1.6138
step 47000: train loss 1.6132, val loss 1.6127
step 47500: train loss 1.6179, val loss 1.6177
step 48000: train loss 1.6178, val loss 1.6136
step 48500: train loss 1.6221, val loss 1.6130
step 49000: train loss 1.6204, val loss 1.6149
step 49500: train loss 1.6224, val loss 1.6203
step 49999: train loss 1.6231, val loss 1.6170
 sugne alla nche conorà del già incommi per avocio anufé l moli al perdetina a
conlo per lue fece io figliamò dò le ciò qui non colà per le lo ommagi a passe
innanzi dicimo per l nozza vera pribiro dico puota fumar guiva mia graroce
mistri batto perversì lo poppo lima che un gelva cistremi cond ecchi manno e
quando e suofbiglia di quetter moggli sì lo sanza ten come del su isponda dicer
si testi innar dolce altretïente anna al lue preconse sovrace non cerso con la
niconten perlo cramicò de se olt
```

[ ]: 10001

[ ]:
```
torch.save(model.state_dict(), 'divinacommedia.pt')

end = datetime.datetime.now()
print(end)
print(end - start)
```

7 DCGAN - Deep Convolutional Generative Adversarial Networks

To see more details on this tutorial.

Another tutorial we'll follow.

First goal is to build up a generator G that gives in output some images starting from noise. Second goal is to build a discriminator D that tells us which images are coming from the generator and which are real.

NOTE: G never sees real images but only learn how to beat D. On the other hand, D is trained on both datasets (real and fake).

A GAN is very hard to train, requires GPU and a lot of time… so we'll try our best. We're going to work with face images, which need to be resized to 64x64. All images have three channels: red, green and blue. The package *vision* in PyTorch is going to help us with these operations.

We'll use a convolutional network. It may be that the number of feature will be very big despite the images becoming very small.

x is going to be a data sample, a tensor of 3x64x64. D is our discriminator. The latent space is a vector of dimension 100 of pure noise. G is our generator.

D wants to maximize its probability to guess right, while G wants to minimize it. For a mathematical POV of GANs, see this reference. We're just assuming that a solution to our problem exists, which is true but in which space?

We'll use the structure described in this paper. This is not a GAN but a DCGAN, i.e. a Deep Convolutional GAN. The idea is to convoy similar images in order to extract features. What's similarity is another problem.

The pipeline, we've a transposed convolution between each step: 1. $\vec{z}$ noise vector 2. black box which decreases the size of our images while increasing the size pf the features - out 3x64x64 feature 3. layer with 256 features of the images

D is made of convolution layers. We don't want to fluctuate a lot (in the neurons) so we chose a Leaky ReLU, which is not as flat as the hyperbolic tangent.

[16]:
```
import argparse
import os
import random
import torch
import torch.nn as nn
import torch.nn.parallel
import torch.backends.cudnn as cudnn
import torch.optim as optim
import torch.utils.data
import torchvision.datasets as dset
import torchvision.transforms as transforms
import torchvision.utils as vutils
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from IPython.display import HTML

# Set random seed for reproducibility
manualSeed = 999
#manualSeed = random.randint(1, 10000) # use if you want new results
print("Random Seed: ", manualSeed)
random.seed(manualSeed)
torch.manual_seed(manualSeed);
```

```
Random Seed:  999
```

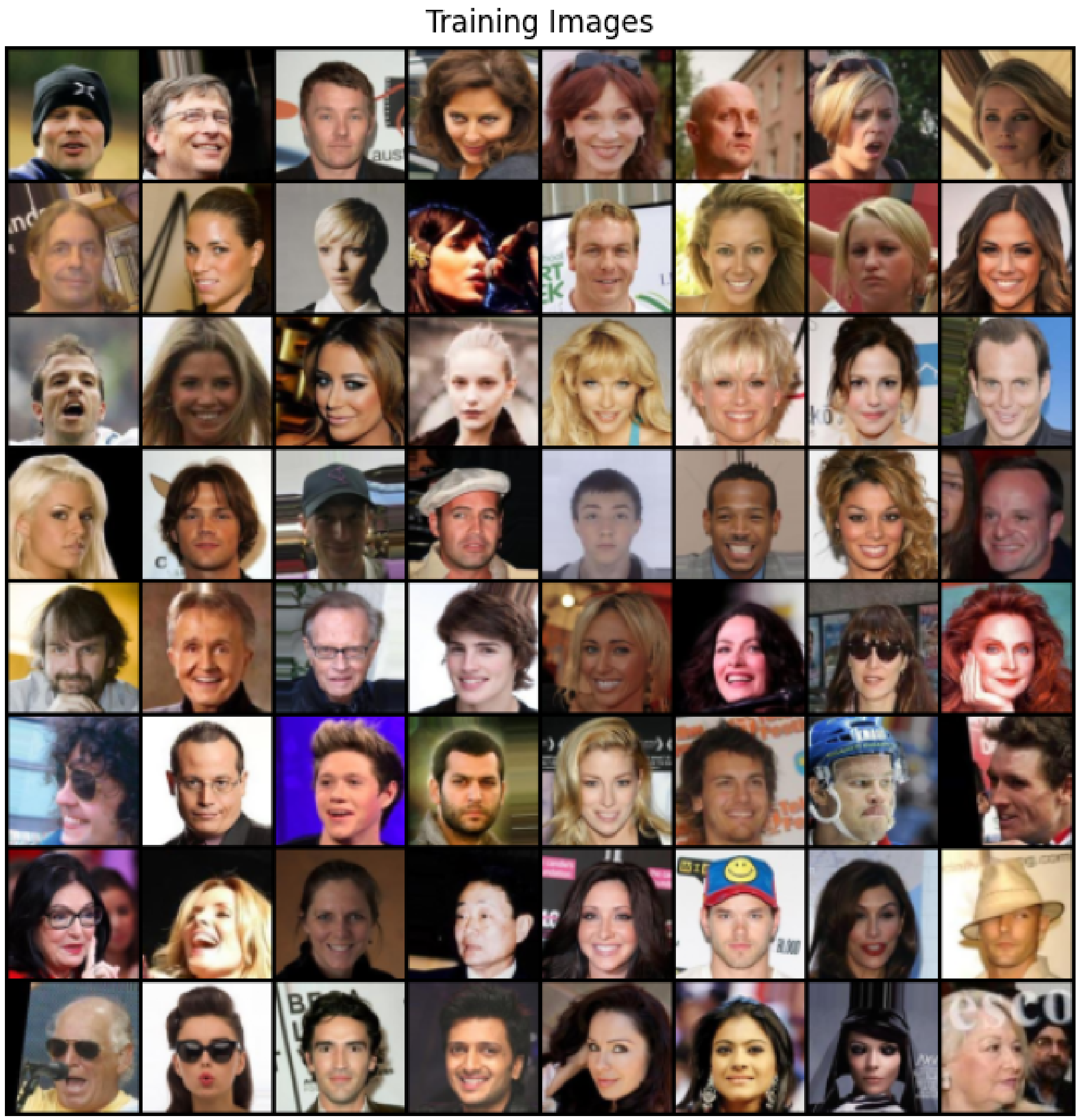
We need now to set some parameters. Download the dataset and put it in "data/celeba".

[17]:
```python
# Root directory for dataset
dataroot = "data/celeba"

# Number of workers for dataloader
workers = 2

# Batch size during training
batch_size = 128

# Spatial size of training images. All images will be resized to this
#   size using a transformer.
image_size = 64

# Number of channels in the training images. For color images this is 3
nc = 3

# Size of z latent vector (i.e. size of generator input)
nz = 100

# Size of feature maps in generator
ngf = 64

# Size of feature maps in discriminator
ndf = 64

# Number of training epochs
num_epochs = 5

# Learning rate for optimizers
lr = 0.0002

# Beta1 hyperparameter for Adam optimizers
beta1 = 0.5

# Number of GPUs available. Use 0 for CPU mode.
ngpu = 1
```

We have to *standardize* our sample images in order to feed them into the convolutional operator. The *DataLoader* is an instance of our data. We'll call it each time we want a batch or similar. It automatically shuffles and does batch things (*batch noise*).

[18]:
```python
# We can use an image folder dataset the way we have it setup.
# Create the dataset
dataset = dset.ImageFolder(root=dataroot,
                           transform=transforms.Compose([
                               transforms.Resize(image_size),
                               transforms.CenterCrop(image_size),
```

```
                               transforms.ToTensor(),
                               transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5,␣
 ↪0.5)),
                           ]))
# Create the dataloader
dataloader = torch.utils.data.DataLoader(dataset, batch_size=batch_size,
                                         shuffle=True, num_workers=workers)

# Decide which device we want to run on
device = torch.device("cuda:0" if (torch.cuda.is_available() and ngpu > 0) else␣
 ↪"cpu")

# Plot some training images
real_batch = next(iter(dataloader))
plt.figure(figsize=(8,8));
plt.axis("off");
plt.title("Training Images");
plt.imshow(np.transpose(vutils.make_grid(real_batch[0].to(device)[:64],␣
 ↪padding=2, normalize=True).cpu(),(1,2,0)));
```



G will take the input $\vec{z}$ of dimension 100 and **up sample** (increase the number of channels) with a transposed convolutional network, giving in output a 3x64x64 dimensional tensor. A discrete convolution operator has an input, which is an image with only one channel (monochromatic). It also requires a *kernel*, which is usually a 3x3 matrix (or at least smaller than the image). The convolution is done within the image and the kernel, like we do in image processing (e.g. filtering).

To learn more about convolution.

To visualize what a convolution does.

We use different kernels to gain different info, e.g. edges. One famous operation is the max pooling, used as much as the average pooling.

[19]:
```
# custom weights initialization called on ``netG`` and ``netD``
def weights_init(m):
    classname = m.__class__.__name__
    if classname.find('Conv') != -1:
        nn.init.normal_(m.weight.data, 0.0, 0.02)
    elif classname.find('BatchNorm') != -1:
        nn.init.normal_(m.weight.data, 1.0, 0.02)
        nn.init.constant_(m.bias.data, 0)
```

[20]:
```
# Generator Code

class Generator(nn.Module):
    def __init__(self, ngpu):
        super(Generator, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
            # input is Z, going into a convolution
            nn.ConvTranspose2d( nz, ngf * 8, 4, 1, 0, bias=False),
            nn.BatchNorm2d(ngf * 8),
            nn.ReLU(True),
            # state size. ``(ngf*8) x 4 x 4``
            nn.ConvTranspose2d(ngf * 8, ngf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 4),
            nn.ReLU(True),
            # state size. ``(ngf*4) x 8 x 8``
            nn.ConvTranspose2d( ngf * 4, ngf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 2),
            nn.ReLU(True),
            # state size. ``(ngf*2) x 16 x 16``
            nn.ConvTranspose2d( ngf * 2, ngf, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf),
            nn.ReLU(True),
            # state size. ``(ngf) x 32 x 32``
            nn.ConvTranspose2d( ngf, nc, 4, 2, 1, bias=False),
            nn.Tanh()
            # state size. ``(nc) x 64 x 64``
        )

    def forward(self, input):
        return self.main(input)
```

D is another thing. It goes not through transpose convolution but *genuine*. It will **down sampling**, we aggregate spatial information while increasing the number of channels. The output will be one number, i.e. 0 if the image is fake or 1 if it's not. All the convolution operators works with parameters changing during training.

[21]:

```
# Create the generator
netG = Generator(ngpu).to(device)

# Handle multi-GPU if desired
if (device.type == 'cuda') and (ngpu > 1):
    netG = nn.DataParallel(netG, list(range(ngpu)))

# Apply the ``weights_init`` function to randomly initialize all weights
#  to ``mean=0``, ``stdev=0.02``.
netG.apply(weights_init)

# Print the model
print(netG)
```

```
Generator(
  (main): Sequential(
    (0): ConvTranspose2d(100, 512, kernel_size=(4, 4), stride=(1, 1),
bias=False)
    (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2),
padding=(1, 1), bias=False)
    (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (5): ReLU(inplace=True)
    (6): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2),
padding=(1, 1), bias=False)
    (7): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (8): ReLU(inplace=True)
    (9): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1,
1), bias=False)
    (10): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (11): ReLU(inplace=True)
    (12): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1,
1), bias=False)
    (13): Tanh()
  )
)
```

[22]:

```
class Discriminator(nn.Module):
    def __init__(self, ngpu):
        super(Discriminator, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
```

```
            # input is ``(nc) x 64 x 64``
            # takes an image of 64x64, pads by 1, strides by 2 (moving two␣
↪steps at a time), and applies a 4x4 kernel
            nn.Conv2d(nc, ndf, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. ``(ndf) x 32 x 32``
            nn.Conv2d(ndf, ndf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 2),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. ``(ndf*2) x 16 x 16``
            nn.Conv2d(ndf * 2, ndf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 4),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. ``(ndf*4) x 8 x 8``
            nn.Conv2d(ndf * 4, ndf * 8, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 8),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. ``(ndf*8) x 4 x 4``
            nn.Conv2d(ndf * 8, 1, 4, 1, 0, bias=False),
            nn.Sigmoid()
        )

    def forward(self, input):
        return self.main(input)
```

Send all the model to GPU and hope.

[23]:
```
# Create the Discriminator
netD = Discriminator(ngpu).to(device)

# Handle multi-GPU if desired
if (device.type == 'cuda') and (ngpu > 1):
    netD = nn.DataParallel(netD, list(range(ngpu)))

# Apply the ``weights_init`` function to randomly initialize all weights
# like this: ``to mean=0, stdev=0.2``.
netD.apply(weights_init)

# Print the model
print(netD)
```

```
Discriminator(
  (main): Sequential(
    (0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1),
bias=False)
    (1): LeakyReLU(negative_slope=0.2, inplace=True)
    (2): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1),
bias=False)
```

```
    (3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (4): LeakyReLU(negative_slope=0.2, inplace=True)
    (5): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1),
bias=False)
    (6): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (7): LeakyReLU(negative_slope=0.2, inplace=True)
    (8): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1),
bias=False)
    (9): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (10): LeakyReLU(negative_slope=0.2, inplace=True)
    (11): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (12): Sigmoid()
  )
)
```

[24]:

```
# Initialize the ``BCELoss`` function
criterion = nn.BCELoss()

# Create batch of latent vectors that we will use to visualize
#  the progression of the generator
fixed_noise = torch.randn(64, nz, 1, 1, device=device)

# Establish convention for real and fake labels during training
real_label = 1.
fake_label = 0.

# Setup Adam optimizers for both G and D
optimizerD = optim.Adam(netD.parameters(), lr=lr, betas=(beta1, 0.999))
optimizerG = optim.Adam(netG.parameters(), lr=lr, betas=(beta1, 0.999))
```

[25]:

```
from tqdm.autonotebook import trange
# Training Loop

# Lists to keep track of progress
img_list = []
G_losses = []
D_losses = []
iters = 0

print("Starting Training Loop...")
# For each epoch
for epoch in trange(num_epochs):
    # For each batch in the dataloader
    for i, data in enumerate(dataloader, 0):
```

```
        ############################
        # (1) Update D network: maximize log(D(x)) + log(1 - D(G(z)))
        ###########################
        ## Train with all-real batch
        netD.zero_grad()
        # Format batch
        real_cpu = data[0].to(device)
        b_size = real_cpu.size(0)
        label = torch.full((b_size,), real_label, dtype=torch.float,␣
↪device=device)
        # Forward pass real batch through D
        output = netD(real_cpu).view(-1)
        # Calculate loss on all-real batch
        errD_real = criterion(output, label)
        # Calculate gradients for D in backward pass
        errD_real.backward()
        D_x = output.mean().item()

        ## Train with all-fake batch
        # Generate batch of latent vectors
        noise = torch.randn(b_size, nz, 1, 1, device=device)
        # Generate fake image batch with G
        fake = netG(noise)
        label.fill_(fake_label)
        # Classify all fake batch with D
        output = netD(fake.detach()).view(-1)
        # Calculate D's loss on the all-fake batch
        errD_fake = criterion(output, label)
        # Calculate the gradients for this batch, accumulated (summed) with␣
↪previous gradients
        errD_fake.backward()
        D_G_z1 = output.mean().item()
        # Compute error of D as sum over the fake and the real batches
        errD = errD_real + errD_fake
        # Update D
        optimizerD.step()

        ############################
        # (2) Update G network: maximize log(D(G(z)))
        ###########################
        netG.zero_grad()
        label.fill_(real_label)  # fake labels are real for generator cost
        # Since we just updated D, perform another forward pass of all-fake␣
↪batch through D
        output = netD(fake).view(-1)
        # Calculate G's loss based on this output
```

```
        errG = criterion(output, label)
        # Calculate gradients for G
        errG.backward()
        D_G_z2 = output.mean().item()
        # Update G
        optimizerG.step()

        # Output training stats
        if i % 50 == 0:
            print('[%d/%d][%d/%d]\tLoss_D: %.4f\tLoss_G: %.4f\tD(x): %.
↪4f\tD(G(z)): %.4f / %.4f'
                  % (epoch, num_epochs, i, len(dataloader),
                     errD.item(), errG.item(), D_x, D_G_z1, D_G_z2))

        # Save Losses for plotting later
        G_losses.append(errG.item())
        D_losses.append(errD.item())

        # Check how the generator is doing by saving G's output on fixed_noise
        if (iters % 500 == 0) or ((epoch == num_epochs-1) and (i ==␣
↪len(dataloader)-1)):
            with torch.no_grad():
                fake = netG(fixed_noise).detach().cpu()
            img_list.append(vutils.make_grid(fake, padding=2, normalize=True))

        iters += 1
```

```
Starting Training Loop…
```

```
/tmp/ipykernel_20041/4233728437.py:1: TqdmExperimentalWarning: Using
`tqdm.autonotebook.tqdm` in notebook mode. Use `tqdm.tqdm` instead to force
console mode (e.g. in jupyter console)
  from tqdm.autonotebook import trange
```

```
  0%|          | 0/5 [00:00<?, ?it/s]
```

```
[0/5][0/1583]   Loss_D: 1.9099  Loss_G: 6.3743  D(x): 0.7247    D(G(z)): 0.7207
/ 0.0032
[0/5][50/1583]  Loss_D: 0.0141  Loss_G: 37.0639 D(x): 0.9895    D(G(z)): 0.0000
/ 0.0000
[0/5][100/1583] Loss_D: 0.0200  Loss_G: 7.1407  D(x): 0.9999    D(G(z)): 0.0186
/ 0.0054
[0/5][150/1583] Loss_D: 0.0967  Loss_G: 10.3588 D(x): 0.9401    D(G(z)): 0.0001
/ 0.0001
[0/5][200/1583] Loss_D: 1.1625  Loss_G: 2.4109  D(x): 0.5155    D(G(z)): 0.0957
/ 0.1315
[0/5][250/1583] Loss_D: 2.1526  Loss_G: 4.5921  D(x): 0.2748    D(G(z)): 0.0137
/ 0.0220
[0/5][300/1583] Loss_D: 0.5904  Loss_G: 3.7758  D(x): 0.7517    D(G(z)): 0.1761
```

```
/ 0.0325
[0/5][350/1583] Loss_D: 0.6402  Loss_G: 4.3502  D(x): 0.7270    D(G(z)): 0.1715
/ 0.0300
[0/5][400/1583] Loss_D: 0.5662  Loss_G: 3.4810  D(x): 0.7792    D(G(z)): 0.1825
/ 0.0442
[0/5][450/1583] Loss_D: 0.5202  Loss_G: 3.8365  D(x): 0.8140    D(G(z)): 0.1853
/ 0.0403
[0/5][500/1583] Loss_D: 0.9568  Loss_G: 2.8461  D(x): 0.5182    D(G(z)): 0.0346
/ 0.1023
[0/5][550/1583] Loss_D: 0.6987  Loss_G: 3.4251  D(x): 0.6322    D(G(z)): 0.0412
/ 0.0700
[0/5][600/1583] Loss_D: 1.2714  Loss_G: 3.3389  D(x): 0.4348    D(G(z)): 0.0255
/ 0.0717
[0/5][650/1583] Loss_D: 0.5458  Loss_G: 4.6146  D(x): 0.8476    D(G(z)): 0.2568
/ 0.0183
[0/5][700/1583] Loss_D: 0.5553  Loss_G: 4.3553  D(x): 0.8172    D(G(z)): 0.2364
/ 0.0221
[0/5][750/1583] Loss_D: 0.6999  Loss_G: 3.1022  D(x): 0.6849    D(G(z)): 0.1496
/ 0.0755
[0/5][800/1583] Loss_D: 1.2943  Loss_G: 8.0054  D(x): 0.9481    D(G(z)): 0.6264
/ 0.0008
[0/5][850/1583] Loss_D: 0.4713  Loss_G: 4.5434  D(x): 0.9304    D(G(z)): 0.2999
/ 0.0162
[0/5][900/1583] Loss_D: 1.2170  Loss_G: 4.5728  D(x): 0.8441    D(G(z)): 0.5668
/ 0.0221
[0/5][950/1583] Loss_D: 1.0568  Loss_G: 2.1666  D(x): 0.4866    D(G(z)): 0.0219
/ 0.2059
[0/5][1000/1583]        Loss_D: 0.4368  Loss_G: 3.0885  D(x): 0.8050    D(G(z)):
0.1508 / 0.0635
[0/5][1050/1583]        Loss_D: 0.7527  Loss_G: 5.3733  D(x): 0.7927    D(G(z)):
0.3339 / 0.0084
[0/5][1100/1583]        Loss_D: 0.4915  Loss_G: 3.6020  D(x): 0.7624    D(G(z)):
0.1287 / 0.0404
[0/5][1150/1583]        Loss_D: 0.4632  Loss_G: 3.8711  D(x): 0.7458    D(G(z)):
0.0651 / 0.0372
[0/5][1200/1583]        Loss_D: 0.6467  Loss_G: 5.5892  D(x): 0.8710    D(G(z)):
0.3278 / 0.0073
[0/5][1250/1583]        Loss_D: 0.5914  Loss_G: 3.8048  D(x): 0.7563    D(G(z)):
0.1795 / 0.0399
[0/5][1300/1583]        Loss_D: 0.4814  Loss_G: 4.6856  D(x): 0.8812    D(G(z)):
0.2511 / 0.0178
[0/5][1350/1583]        Loss_D: 1.9527  Loss_G: 1.6096  D(x): 0.2460    D(G(z)):
0.0024 / 0.3133
[0/5][1400/1583]        Loss_D: 0.8679  Loss_G: 6.5367  D(x): 0.9467    D(G(z)):
0.5060 / 0.0031
[0/5][1450/1583]        Loss_D: 0.8913  Loss_G: 2.7263  D(x): 0.6141    D(G(z)):
0.1223 / 0.1192
[0/5][1500/1583]        Loss_D: 0.9655  Loss_G: 3.7428  D(x): 0.5047    D(G(z)):
```

```
0.0169 / 0.0497
[0/5][1550/1583]        Loss_D: 0.4875  Loss_G: 3.3498  D(x): 0.7143    D(G(z)):
0.0507 / 0.0549
[1/5][0/1583]   Loss_D: 1.5324  Loss_G: 7.4114  D(x): 0.9876    D(G(z)): 0.6992
/ 0.0014
[1/5][50/1583]  Loss_D: 2.0105  Loss_G: 1.2554  D(x): 0.2239    D(G(z)): 0.0054
/ 0.3891
[1/5][100/1583] Loss_D: 0.5478  Loss_G: 3.2523  D(x): 0.7728    D(G(z)): 0.1819
/ 0.0703
[1/5][150/1583] Loss_D: 0.3785  Loss_G: 5.4905  D(x): 0.9266    D(G(z)): 0.2343
/ 0.0067
[1/5][200/1583] Loss_D: 0.5012  Loss_G: 3.4440  D(x): 0.7614    D(G(z)): 0.1355
/ 0.0474
[1/5][250/1583] Loss_D: 0.6742  Loss_G: 2.7718  D(x): 0.5928    D(G(z)): 0.0229
/ 0.1051
[1/5][300/1583] Loss_D: 0.6091  Loss_G: 2.9228  D(x): 0.8135    D(G(z)): 0.2675
/ 0.0818
[1/5][350/1583] Loss_D: 0.6739  Loss_G: 2.9099  D(x): 0.6376    D(G(z)): 0.0534
/ 0.1013
[1/5][400/1583] Loss_D: 0.6068  Loss_G: 3.9161  D(x): 0.9245    D(G(z)): 0.3425
/ 0.0341
[1/5][450/1583] Loss_D: 0.3417  Loss_G: 4.2868  D(x): 0.9152    D(G(z)): 0.1934
/ 0.0251
[1/5][500/1583] Loss_D: 0.8219  Loss_G: 5.2256  D(x): 0.8441    D(G(z)): 0.3991
/ 0.0128
[1/5][550/1583] Loss_D: 0.6752  Loss_G: 2.5048  D(x): 0.6495    D(G(z)): 0.1236
/ 0.1179
[1/5][600/1583] Loss_D: 0.5840  Loss_G: 5.0734  D(x): 0.8912    D(G(z)): 0.3148
/ 0.0111
[1/5][650/1583] Loss_D: 0.4884  Loss_G: 4.3525  D(x): 0.8963    D(G(z)): 0.2778
/ 0.0200
[1/5][700/1583] Loss_D: 0.4877  Loss_G: 3.3474  D(x): 0.8406    D(G(z)): 0.2384
/ 0.0498
[1/5][750/1583] Loss_D: 1.1269  Loss_G: 0.7265  D(x): 0.4011    D(G(z)): 0.0151
/ 0.5787
[1/5][800/1583] Loss_D: 0.5772  Loss_G: 4.6037  D(x): 0.9151    D(G(z)): 0.3485
/ 0.0141
[1/5][850/1583] Loss_D: 0.5741  Loss_G: 2.7290  D(x): 0.6629    D(G(z)): 0.0829
/ 0.1019
[1/5][900/1583] Loss_D: 0.4530  Loss_G: 3.0647  D(x): 0.8725    D(G(z)): 0.2417
/ 0.0643
[1/5][950/1583] Loss_D: 0.4756  Loss_G: 3.1367  D(x): 0.7725    D(G(z)): 0.1484
/ 0.0646
[1/5][1000/1583]        Loss_D: 0.5702  Loss_G: 2.9648  D(x): 0.7021    D(G(z)):
0.1266 / 0.0790
[1/5][1050/1583]        Loss_D: 0.5528  Loss_G: 3.1080  D(x): 0.7750    D(G(z)):
0.1870 / 0.0654
[1/5][1100/1583]        Loss_D: 0.8720  Loss_G: 3.4957  D(x): 0.5464    D(G(z)):
```

```
0.0569 / 0.0570
[1/5][1150/1583]        Loss_D: 0.6698  Loss_G: 2.0597  D(x): 0.6376    D(G(z)):
0.1199 / 0.1790
[1/5][1200/1583]        Loss_D: 1.2315  Loss_G: 1.7836  D(x): 0.3674    D(G(z)):
0.0142 / 0.2395
[1/5][1250/1583]        Loss_D: 0.3882  Loss_G: 2.6825  D(x): 0.8278    D(G(z)):
0.1547 / 0.0925
[1/5][1300/1583]        Loss_D: 0.9942  Loss_G: 5.3290  D(x): 0.9551    D(G(z)):
0.5566 / 0.0090
[1/5][1350/1583]        Loss_D: 0.9556  Loss_G: 1.0415  D(x): 0.4930    D(G(z)):
0.0546 / 0.4071
[1/5][1400/1583]        Loss_D: 0.5667  Loss_G: 2.6576  D(x): 0.7704    D(G(z)):
0.2054 / 0.0973
[1/5][1450/1583]        Loss_D: 0.5317  Loss_G: 2.1880  D(x): 0.6979    D(G(z)):
0.0844 / 0.1527
[1/5][1500/1583]        Loss_D: 0.8669  Loss_G: 4.3864  D(x): 0.9260    D(G(z)):
0.4730 / 0.0252
[1/5][1550/1583]        Loss_D: 0.6101  Loss_G: 2.1893  D(x): 0.7117    D(G(z)):
0.1827 / 0.1532
[2/5][0/1583]   Loss_D: 0.3958  Loss_G: 2.7876  D(x): 0.8350    D(G(z)): 0.1751
/ 0.0788
[2/5][50/1583]  Loss_D: 0.5490  Loss_G: 3.6351  D(x): 0.9131    D(G(z)): 0.3367
/ 0.0349
[2/5][100/1583] Loss_D: 0.7977  Loss_G: 4.3410  D(x): 0.9091    D(G(z)): 0.4493
/ 0.0209
[2/5][150/1583] Loss_D: 0.5076  Loss_G: 3.1492  D(x): 0.8544    D(G(z)): 0.2686
/ 0.0594
[2/5][200/1583] Loss_D: 0.9014  Loss_G: 1.0226  D(x): 0.5477    D(G(z)): 0.1295
/ 0.4238
[2/5][250/1583] Loss_D: 0.6308  Loss_G: 2.0325  D(x): 0.6619    D(G(z)): 0.1252
/ 0.1729
[2/5][300/1583] Loss_D: 0.5530  Loss_G: 3.2794  D(x): 0.8813    D(G(z)): 0.3134
/ 0.0496
[2/5][350/1583] Loss_D: 1.1486  Loss_G: 0.5777  D(x): 0.4046    D(G(z)): 0.0721
/ 0.6153
[2/5][400/1583] Loss_D: 0.8380  Loss_G: 1.1473  D(x): 0.5374    D(G(z)): 0.0985
/ 0.3760
[2/5][450/1583] Loss_D: 0.7805  Loss_G: 3.3612  D(x): 0.9115    D(G(z)): 0.4476
/ 0.0479
[2/5][500/1583] Loss_D: 1.4437  Loss_G: 4.4484  D(x): 0.9760    D(G(z)): 0.6933
/ 0.0206
[2/5][550/1583] Loss_D: 0.8297  Loss_G: 4.1981  D(x): 0.9172    D(G(z)): 0.4767
/ 0.0227
[2/5][600/1583] Loss_D: 0.5903  Loss_G: 1.9271  D(x): 0.6947    D(G(z)): 0.1547
/ 0.1821
[2/5][650/1583] Loss_D: 0.5221  Loss_G: 2.0123  D(x): 0.7642    D(G(z)): 0.1802
/ 0.1699
[2/5][700/1583] Loss_D: 0.6297  Loss_G: 1.6601  D(x): 0.6501    D(G(z)): 0.1182
```

```
/ 0.2485
[2/5][750/1583] Loss_D: 0.6537  Loss_G: 1.5574  D(x): 0.6579    D(G(z)): 0.1579
/ 0.2563
[2/5][800/1583] Loss_D: 0.6317  Loss_G: 3.1065  D(x): 0.8006    D(G(z)): 0.2941
/ 0.0601
[2/5][850/1583] Loss_D: 0.5326  Loss_G: 3.0882  D(x): 0.8573    D(G(z)): 0.2838
/ 0.0609
[2/5][900/1583] Loss_D: 1.9367  Loss_G: 5.7856  D(x): 0.9678    D(G(z)): 0.7939
/ 0.0060
[2/5][950/1583] Loss_D: 0.9961  Loss_G: 3.9979  D(x): 0.8327    D(G(z)): 0.4952
/ 0.0304
[2/5][1000/1583]        Loss_D: 0.7815  Loss_G: 1.3450  D(x): 0.5794    D(G(z)):
0.1348 / 0.3103
[2/5][1050/1583]        Loss_D: 0.9293  Loss_G: 1.0844  D(x): 0.4977    D(G(z)):
0.1058 / 0.3797
[2/5][1100/1583]        Loss_D: 0.6524  Loss_G: 3.3718  D(x): 0.9269    D(G(z)):
0.4004 / 0.0461
[2/5][1150/1583]        Loss_D: 0.5164  Loss_G: 2.5253  D(x): 0.8069    D(G(z)):
0.2292 / 0.1006
[2/5][1200/1583]        Loss_D: 0.8181  Loss_G: 4.2514  D(x): 0.9478    D(G(z)):
0.4851 / 0.0205
[2/5][1250/1583]        Loss_D: 0.5746  Loss_G: 2.5811  D(x): 0.7622    D(G(z)):
0.2323 / 0.0965
[2/5][1300/1583]        Loss_D: 0.6178  Loss_G: 3.1261  D(x): 0.7924    D(G(z)):
0.2847 / 0.0581
[2/5][1350/1583]        Loss_D: 0.5946  Loss_G: 2.6991  D(x): 0.8336    D(G(z)):
0.3056 / 0.0862
[2/5][1400/1583]        Loss_D: 0.5715  Loss_G: 2.7533  D(x): 0.7760    D(G(z)):
0.2322 / 0.0834
[2/5][1450/1583]        Loss_D: 0.5097  Loss_G: 2.5159  D(x): 0.7797    D(G(z)):
0.1981 / 0.1036
[2/5][1500/1583]        Loss_D: 0.4735  Loss_G: 2.5532  D(x): 0.7583    D(G(z)):
0.1468 / 0.1042
[2/5][1550/1583]        Loss_D: 1.2453  Loss_G: 0.8352  D(x): 0.3612    D(G(z)):
0.0567 / 0.4899
[3/5][0/1583]   Loss_D: 0.4022  Loss_G: 2.3569  D(x): 0.7823    D(G(z)): 0.1196
/ 0.1186
[3/5][50/1583]  Loss_D: 0.5382  Loss_G: 2.0065  D(x): 0.7290    D(G(z)): 0.1619
/ 0.1726
[3/5][100/1583] Loss_D: 1.0891  Loss_G: 0.8934  D(x): 0.4210    D(G(z)): 0.0603
/ 0.4651
[3/5][150/1583] Loss_D: 0.5216  Loss_G: 1.8939  D(x): 0.7229    D(G(z)): 0.1490
/ 0.1840
[3/5][200/1583] Loss_D: 0.8771  Loss_G: 3.1227  D(x): 0.8431    D(G(z)): 0.4572
/ 0.0580
[3/5][250/1583] Loss_D: 0.6495  Loss_G: 4.1753  D(x): 0.9223    D(G(z)): 0.3961
/ 0.0223
[3/5][300/1583] Loss_D: 0.5762  Loss_G: 1.9712  D(x): 0.7218    D(G(z)): 0.1832
```

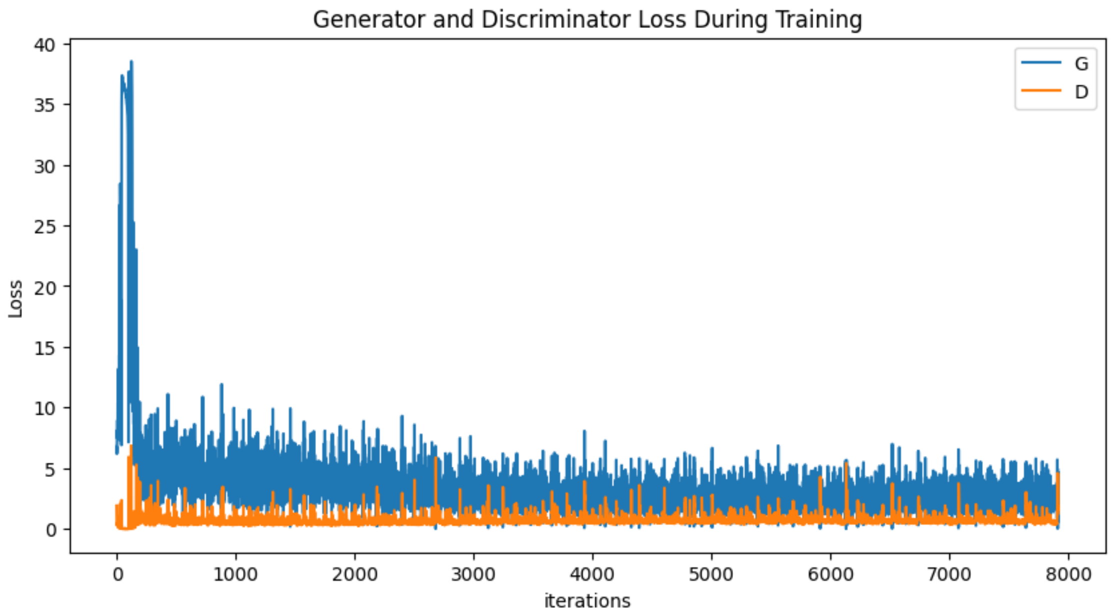
```
/ 0.1793
[3/5][350/1583] Loss_D: 0.6440  Loss_G: 2.3623  D(x): 0.8140    D(G(z)): 0.3259
/ 0.1203
[3/5][400/1583] Loss_D: 0.7529  Loss_G: 2.0329  D(x): 0.5854    D(G(z)): 0.1235
/ 0.1840
[3/5][450/1583] Loss_D: 1.2865  Loss_G: 4.1535  D(x): 0.9433    D(G(z)): 0.6405
/ 0.0252
[3/5][500/1583] Loss_D: 1.4668  Loss_G: 0.3845  D(x): 0.3206    D(G(z)): 0.0717
/ 0.7032
[3/5][550/1583] Loss_D: 1.3335  Loss_G: 4.4294  D(x): 0.9460    D(G(z)): 0.6672
/ 0.0196
[3/5][600/1583] Loss_D: 0.5612  Loss_G: 1.9962  D(x): 0.7187    D(G(z)): 0.1733
/ 0.1624
[3/5][650/1583] Loss_D: 0.7940  Loss_G: 1.8931  D(x): 0.6220    D(G(z)): 0.2011
/ 0.1961
[3/5][700/1583] Loss_D: 0.8911  Loss_G: 1.1413  D(x): 0.5241    D(G(z)): 0.1121
/ 0.3669
[3/5][750/1583] Loss_D: 0.7487  Loss_G: 3.6305  D(x): 0.7980    D(G(z)): 0.3592
/ 0.0381
[3/5][800/1583] Loss_D: 0.5438  Loss_G: 1.8812  D(x): 0.7019    D(G(z)): 0.1294
/ 0.1924
[3/5][850/1583] Loss_D: 0.9020  Loss_G: 3.1525  D(x): 0.8563    D(G(z)): 0.4824
/ 0.0597
[3/5][900/1583] Loss_D: 0.7041  Loss_G: 2.7916  D(x): 0.8332    D(G(z)): 0.3597
/ 0.0834
[3/5][950/1583] Loss_D: 1.2013  Loss_G: 4.2747  D(x): 0.9549    D(G(z)): 0.6259
/ 0.0226
[3/5][1000/1583]        Loss_D: 0.5632  Loss_G: 1.7900  D(x): 0.6888    D(G(z)):
0.1304 / 0.2021
[3/5][1050/1583]        Loss_D: 0.6774  Loss_G: 1.9858  D(x): 0.7649    D(G(z)):
0.2975 / 0.1690
[3/5][1100/1583]        Loss_D: 0.6080  Loss_G: 2.1017  D(x): 0.7285    D(G(z)):
0.2085 / 0.1542
[3/5][1150/1583]        Loss_D: 0.6307  Loss_G: 2.5813  D(x): 0.7240    D(G(z)):
0.2182 / 0.0970
[3/5][1200/1583]        Loss_D: 0.6023  Loss_G: 3.0957  D(x): 0.8584    D(G(z)):
0.3220 / 0.0593
[3/5][1250/1583]        Loss_D: 0.6756  Loss_G: 2.9132  D(x): 0.8183    D(G(z)):
0.3441 / 0.0708
[3/5][1300/1583]        Loss_D: 0.5846  Loss_G: 2.3920  D(x): 0.7863    D(G(z)):
0.2578 / 0.1137
[3/5][1350/1583]        Loss_D: 0.5652  Loss_G: 1.9557  D(x): 0.7425    D(G(z)):
0.2047 / 0.1748
[3/5][1400/1583]        Loss_D: 1.3139  Loss_G: 1.7533  D(x): 0.3667    D(G(z)):
0.0653 / 0.2297
[3/5][1450/1583]        Loss_D: 0.5056  Loss_G: 2.4902  D(x): 0.7856    D(G(z)):
0.2043 / 0.1019
[3/5][1500/1583]        Loss_D: 0.8038  Loss_G: 1.0841  D(x): 0.5505    D(G(z)):
```

```
0.1150 / 0.3841
[3/5][1550/1583]        Loss_D: 0.6969  Loss_G: 1.5943  D(x): 0.6045    D(G(z)):
0.1120 / 0.2528
[4/5][0/1583]   Loss_D: 0.5658  Loss_G: 2.5638  D(x): 0.7696    D(G(z)): 0.2305
/ 0.0962
[4/5][50/1583]  Loss_D: 0.6548  Loss_G: 2.1725  D(x): 0.7111    D(G(z)): 0.2200
/ 0.1441
[4/5][100/1583] Loss_D: 1.4930  Loss_G: 0.9647  D(x): 0.2895    D(G(z)): 0.0254
/ 0.4522
[4/5][150/1583] Loss_D: 0.6309  Loss_G: 2.0180  D(x): 0.6997    D(G(z)): 0.1843
/ 0.1663
[4/5][200/1583] Loss_D: 0.9915  Loss_G: 1.3551  D(x): 0.5099    D(G(z)): 0.1596
/ 0.2999
[4/5][250/1583] Loss_D: 1.0222  Loss_G: 0.7995  D(x): 0.4617    D(G(z)): 0.0896
/ 0.4977
[4/5][300/1583] Loss_D: 0.8712  Loss_G: 4.0830  D(x): 0.8893    D(G(z)): 0.4834
/ 0.0237
[4/5][350/1583] Loss_D: 0.6203  Loss_G: 2.2146  D(x): 0.7273    D(G(z)): 0.2092
/ 0.1384
[4/5][400/1583] Loss_D: 0.8236  Loss_G: 1.2575  D(x): 0.5200    D(G(z)): 0.0611
/ 0.3437
[4/5][450/1583] Loss_D: 0.5394  Loss_G: 2.2857  D(x): 0.7571    D(G(z)): 0.1924
/ 0.1305
[4/5][500/1583] Loss_D: 0.6984  Loss_G: 1.8579  D(x): 0.6040    D(G(z)): 0.1046
/ 0.1930
[4/5][550/1583] Loss_D: 1.1317  Loss_G: 3.9177  D(x): 0.9058    D(G(z)): 0.5851
/ 0.0305
[4/5][600/1583] Loss_D: 0.5926  Loss_G: 2.5621  D(x): 0.7384    D(G(z)): 0.2080
/ 0.1041
[4/5][650/1583] Loss_D: 0.7653  Loss_G: 2.9855  D(x): 0.8221    D(G(z)): 0.3842
/ 0.0684
[4/5][700/1583] Loss_D: 0.5038  Loss_G: 1.6738  D(x): 0.7441    D(G(z)): 0.1579
/ 0.2200
[4/5][750/1583] Loss_D: 1.0488  Loss_G: 2.5327  D(x): 0.7516    D(G(z)): 0.4446
/ 0.1204
[4/5][800/1583] Loss_D: 0.5126  Loss_G: 2.0503  D(x): 0.7518    D(G(z)): 0.1677
/ 0.1660
[4/5][850/1583] Loss_D: 0.7430  Loss_G: 1.2009  D(x): 0.5832    D(G(z)): 0.0993
/ 0.3460
[4/5][900/1583] Loss_D: 0.7774  Loss_G: 2.8990  D(x): 0.7179    D(G(z)): 0.3088
/ 0.0748
[4/5][950/1583] Loss_D: 0.5661  Loss_G: 2.4152  D(x): 0.8082    D(G(z)): 0.2673
/ 0.1150
[4/5][1000/1583]        Loss_D: 0.6005  Loss_G: 1.8138  D(x): 0.7127    D(G(z)):
0.1808 / 0.2042
[4/5][1050/1583]        Loss_D: 0.8559  Loss_G: 3.8857  D(x): 0.9451    D(G(z)):
0.4993 / 0.0274
[4/5][1100/1583]        Loss_D: 0.3884  Loss_G: 2.3496  D(x): 0.8358    D(G(z)):
```

```
0.1637 / 0.1236
[4/5][1150/1583]        Loss_D: 0.5808  Loss_G: 2.6895  D(x): 0.8713   D(G(z)):
0.3256 / 0.0900
[4/5][1200/1583]        Loss_D: 0.6785  Loss_G: 2.1774  D(x): 0.7861   D(G(z)):
0.3067 / 0.1429
[4/5][1250/1583]        Loss_D: 0.5954  Loss_G: 2.6246  D(x): 0.8585   D(G(z)):
0.3240 / 0.0932
[4/5][1300/1583]        Loss_D: 0.4418  Loss_G: 2.5867  D(x): 0.7218   D(G(z)):
0.0658 / 0.1091
[4/5][1350/1583]        Loss_D: 0.8660  Loss_G: 4.7940  D(x): 0.9001   D(G(z)):
0.4803 / 0.0139
[4/5][1400/1583]        Loss_D: 0.9944  Loss_G: 0.9465  D(x): 0.4810   D(G(z)):
0.0818 / 0.4363
[4/5][1450/1583]        Loss_D: 1.5495  Loss_G: 4.0707  D(x): 0.9619   D(G(z)):
0.7269 / 0.0265
[4/5][1500/1583]        Loss_D: 0.7081  Loss_G: 2.3837  D(x): 0.7039   D(G(z)):
0.2464 / 0.1228
[4/5][1550/1583]        Loss_D: 0.4877  Loss_G: 2.5947  D(x): 0.8155   D(G(z)):
0.2182 / 0.0972
```

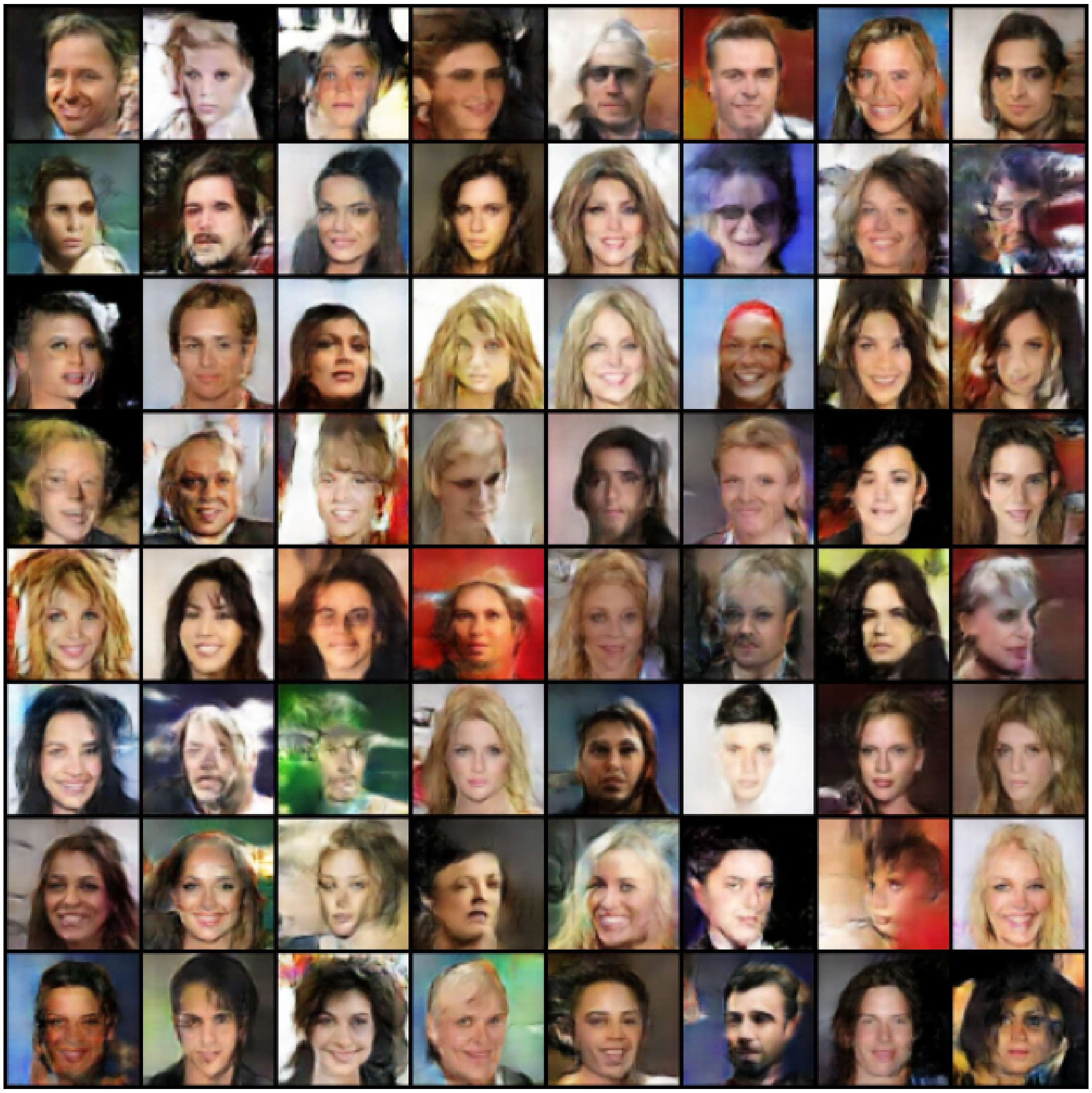
[26]:
```
# save the model
torch.save(netG.state_dict(), 'dcgan_generator.pt')
```

[27]:
```
plt.figure(figsize=(10,5))
plt.title("Generator and Discriminator Loss During Training")
plt.plot(G_losses,label="G")
plt.plot(D_losses,label="D")
plt.xlabel("iterations")
plt.ylabel("Loss")
plt.legend()
plt.show()
```



[28]:
```
fig = plt.figure(figsize=(8,8))
plt.axis("off")
ims = [[plt.imshow(np.transpose(i,(1,2,0)), animated=True)] for i in img_list]
ani = animation.ArtistAnimation(fig, ims, interval=1000, repeat_delay=1000,␣
 ↪blit=True)

HTML(ani.to_jshtml())
```

[28]: <IPython.core.display.HTML object>



[29]:
```
# Grab a batch of real images from the dataloader
real_batch = next(iter(dataloader))

# Plot the real images
plt.figure(figsize=(15,15))
plt.subplot(1,2,1)
plt.axis("off")
plt.title("Real Images")
plt.imshow(np.transpose(vutils.make_grid(real_batch[0].to(device)[:64],␣
↪padding=5, normalize=True).cpu(),(1,2,0)))

# Plot the fake images from the last epoch
plt.subplot(1,2,2)
```

```
plt.axis("off")
plt.title("Fake Images")
plt.imshow(np.transpose(img_list[-1],(1,2,0)))
plt.show()
```

Real Images

Fake Images