Sampling:

N[0] → raw counts of names starting w' char c.

p = N[0].float.

p = p / p.sum(). → ∴ p = prob of a ~~word~~.name starting w' c.

---

g. Generator(). manual_seed( ). → seed gen.

g

p = torch.rand(3, generator = g) → Tensor with 3 elements b/w 0,1.

p = p/p.sum() → Normalize.

Now, torch.multinomial(p, num_samples = 20, generator = g, replacement = True)

p → tensor with prob.

num_samples → no. of samples

↳ returns a sample tensor of dim: (1 × num_sa...)

---

loop to print a name based on bigram probabilities

```
ix = 0
word = [ ].
while True:
    p = N[ix].float()
    p = p/p.sum()
    ix = torch.multinomial(p, num_sa = 1, rep = True).item()
    word.append [itos [ix]
    if (ix == 0):
        break
```

Should matter here

↓

.item()

↓

coz its 1

Now to make a matrix P of prob distr:

$P_\theta = N.\text{float}()$.

↳ P.sum() = ([:---]) ⟶ single item = sum of all in N. (X)

Sum

~~Broadcasting~~ rules:

P.sum(dim, keepdim = True)

↑ dim accross which we sum.

↳ false by default.

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$$

P.sum(0, ← across rows, = true) ⟶ [[9, 12]].

1 x 2.

3 x 2

P.sum(1, ← across colm, keepdim = True)

P.sum(0)

[9, 12].

single dim 2 elem

$$\begin{bmatrix} [3] \\ [7] \\ [11] \end{bmatrix}$$

3 x 1

int min ↓

$$\begin{bmatrix} 3 \\ 7 \\ 11 \end{bmatrix} = [3, 7, 11]$$

1-d tensor. = 3 elem

{20,

1 1 3
2 , 7
5 ,

20 25

we can verify the above by using .shape.

Broadcasting Semantics :

to have tensor operations:

1) both must have min 1 d.

2) Align from RIGHT to LEFT.

then iterating over each → both same
the dimension should b → either is 1
either DNE.

BUT.

if P = N.float() → [27 x 27].

P = P / P.sum(1) → $\frac{[27, 27]}{[27]}$ → $\frac{27 \cdot 27}{-\quad 27}$

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$$

$\frac{27 \cdot 27}{1 \cdot 27}$

P.sum(1) [3, 7, 11].

∴ P/P.sum = $\frac{3 \times 2}{2 \times 3}$

↓ not allowed.

$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \rightarrow$ P.sum(1) $[6, 15, 24]$.

P / P.sum

$= \frac{3 \times 3}{- \times 3} = \frac{3 \times 3}{1 \times 3} = \frac{3 \times 3}{3 \times 3}$. → squeezing / copying

$[6, 15, 24] \rightarrow \begin{bmatrix} 6 & 15 & 24 \\ 6 & 15 & 24 \\ 6 & 15 & 24 \end{bmatrix}$ to stretch vector

now division occurs and gives wrong ans.

Correct way:

P.sum(1, keepdim = True) = [[6],
[15],
[24]].

stretch → $\begin{bmatrix} [6, 6, 6], \\ [15, 15, 15], \\ [24, 24, 24] \end{bmatrix}$

now div ✓

---

its v easy to add bugs lol.

we create dimensions as we go from R → L.

P /= P.sum(1, keepdims = True) > P = P/P...

↑ either

↑ creates new object

Evaluate Quality of Model:

So to evaluate the quality of model, we multiply the prob of each bigram in the resp. names. take the -ve log of it and average of that.

(doubt as to why?)

emma =

		log	- log
.e	→ 0.04	- 3.0	3.0
em	→ 0.03	- 3.2	3.2
mm	→ 0.02	- 3.6	3.6
ma	→ 0.38	- 0.9	0.9
a.	→ 0.19	- 1.6	1.6

} VV likely

$0.04 \rightarrow \frac{1}{27}$ = avg.

log

avg =

Sum of Logs = log-likelyhood.

we use

avg neg log likelyhood.

Lower it is, better the model

for w in "a":

```
chs = ['.'] + list(w) + ['w']
for ch1, ch1
    ix1, ix2 = ... stoi
    log-prob = torch.log(P[[i1],[i2]])
    log-lik += log-prob.
    n += 1
    print(prob)
```

nll = - log-lik

nll/n = avg

One flaw is that if the bigram never existed in training set, our model returs nll/n = +inf for that, which we do not like

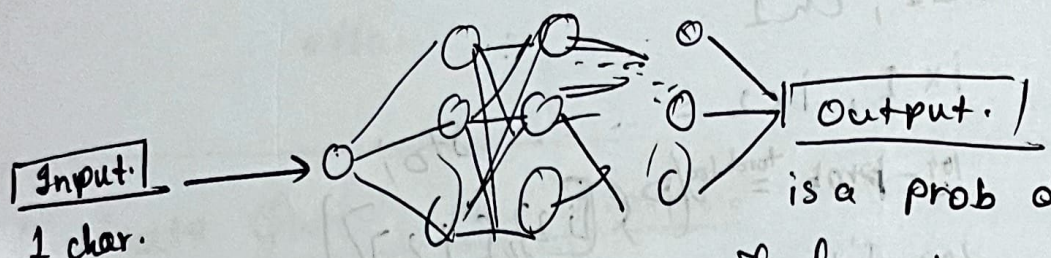
∴ use P = ~~N+~~ (N+1).float()

Now find total nll/n. for entire set.

=

Quality

Doing the same with Neural Nets: :)



is a prob distribution of characters that can follow 'ch'. Now using info abt the actual bigram we calc loss func and minimise it

~~here we take in~~

eg: emma

bi = [.e, em, mm, ma, a.]

xs = [., e, m, m, a]

ys = [e, m, m, a, .]

```
for w in names:
    chs = ___
    for ch1, ch2:
        ix1, ix2 .
        xs.apper(ix1)
        ys.
XS = torch.tensor(xs)
```

It is best not to give single integer input to a neural net.

Note: torch.tensor vs torch.Tensor

dtype = that of list

dtype = float32

So we use one_hot funcs.

to convert integer index input to 27-tensor (zero) with ith index = 1.

xs = [0, 5, 13, 13, 1] ⟶ [[1,0,0....], [0,0,0,0,0,1,0....], [0...,1,0...], ...]

import torch.nn.functional as F

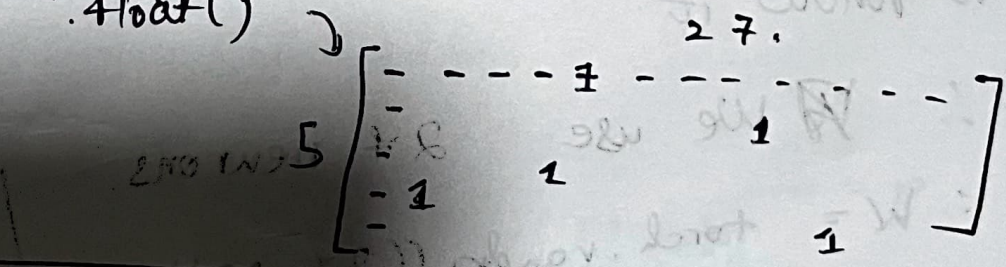
xenc = F.one_hot(xs, num_classes = 27)

xenc.shape ⟶ ([5, 27]).

Since we had used .tensor, xs.dtype = int64

∴ xenc ⟶ int64 but in NN we want float

∴ xenc = " ".float()



---

1 Neuron: takes word as input as xenc.

W = torch.rand((27, 1)) ⟶ [0.12..., 3.145, 2.16..., -1.23.] (27)

[[1.237], [3.145], [-1.2], ..., [0.6]]

↳ uses normal distribution ↗ and returns (27×1) tensor of floats.

input.

(5×27)

[ ]

W
(27×1)

1 Neuron

[ ]

∴ out = xenc @ W → [5 × 1]

[[ ]
[ ]
[ ]
[ ]]

out gets 1 output for each input character.

but for each input char, we want a prob distribution of all poss characters to follow it.

! We use 27 neurons.

∴ W = torch.randn((27, 27))

∴ xenc @ W

(5×27) × (27, 27) → (5×27) → [[, ...]
[. — —]
[ ]
[ ]
[ ]]

torch.exp

(xenc @ W)[3][1]

prob that 'm' is followed by 'a'

but xenc @ W → has +ve, -ve nos.

∴ logits = xenc @ W (~ logs.)

counts = logits.exp() (~ N)

probs = ~~log~~ counts / counts.sum(1, keepdims = True).

∴ now we have a prob distribution for each input char.

probs.shape = (5 x 27)
probs[0].sum() = 1.0

---

nlls = torch.zeros(5):

for i in (5):

x = xs[i]; y = ys[i];

"input cha" x

"prob dis" prob[i][ ]

"expect for y" prob[i, y].item() → = p.

∴ logp = torch.log(p)

nll = - logp.

nlls[i] = nll.

avg log lik = nlls.mean().item().

we added 1 to each N [illegible] statistical model to smoothen things out. Adding larger nos to N will smoothen it.

Similarly in NNs, W ke values, if we bring them closer to zero. then it smoothen

coz logits = xenc @ W ~ 0

∴ ~~pro~~ log ~ 1.

∴ prob smoothen

↳ called regularization.

Sum

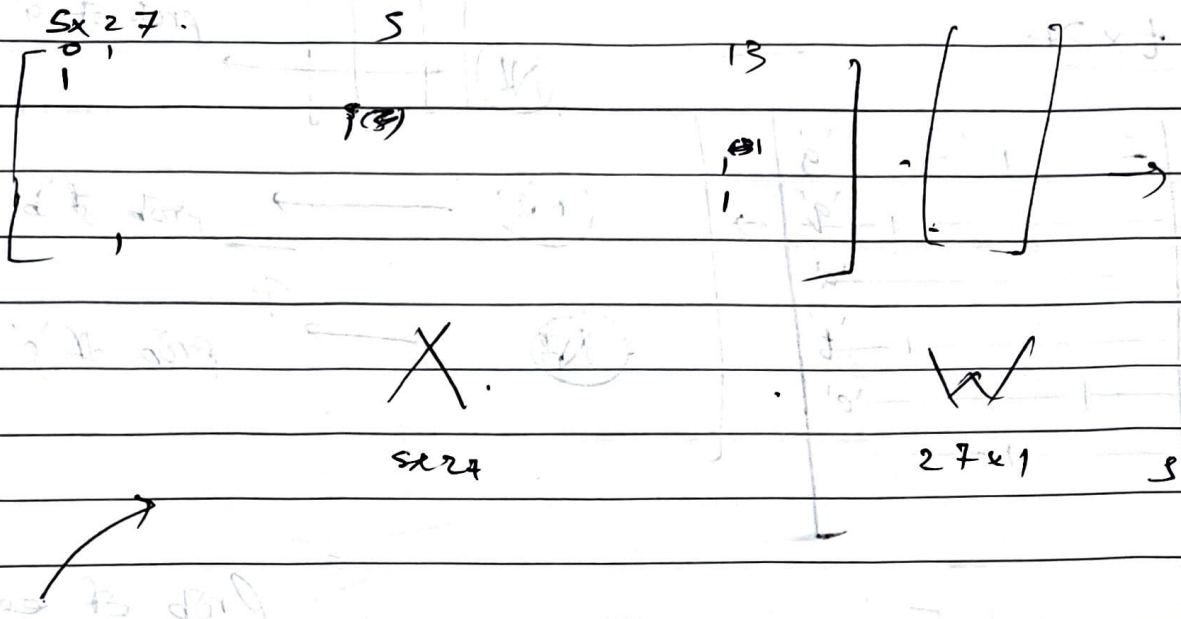
loss = -probs[---].log.mean() + 0.1*(W**2).mean.

∴ gradient descent tries to decrease W**2 i.e. bring value of W closer to 0.

__/__/__

Karpathy lec-2

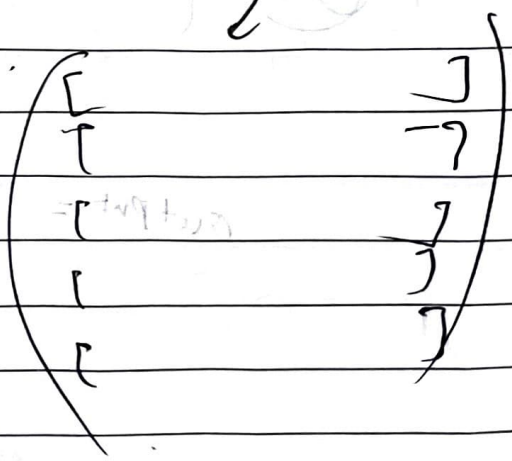
X-encoding :

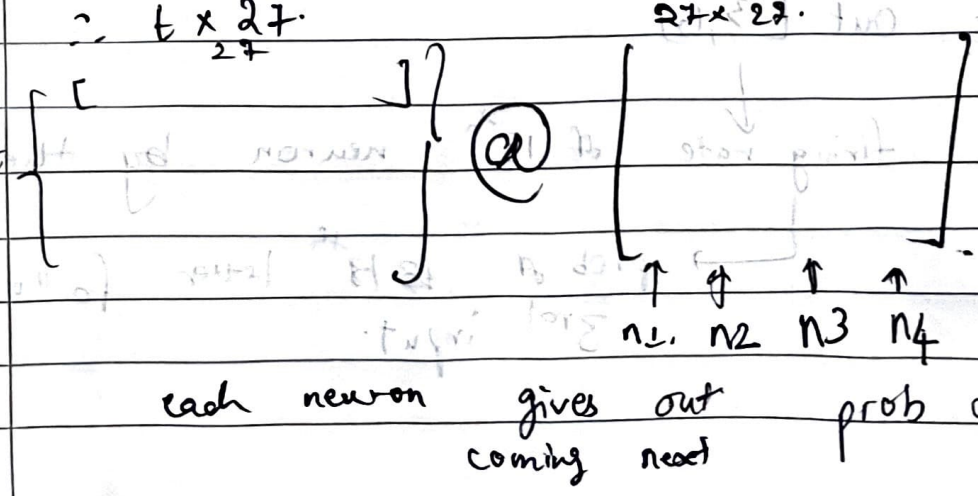


emma
=.e
=em
mm
ma
a.

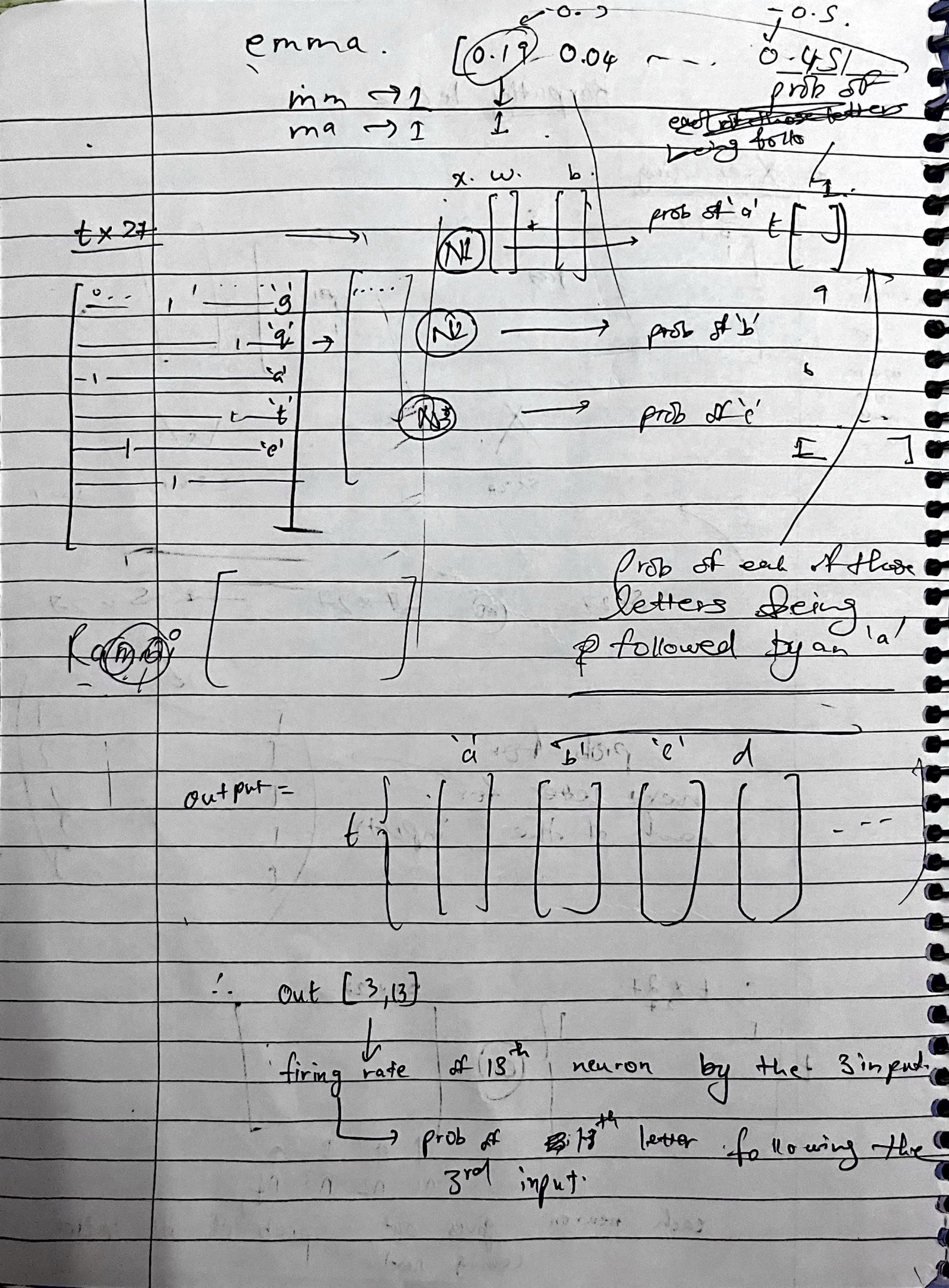
5x27 @ 27x27 → 5x27.

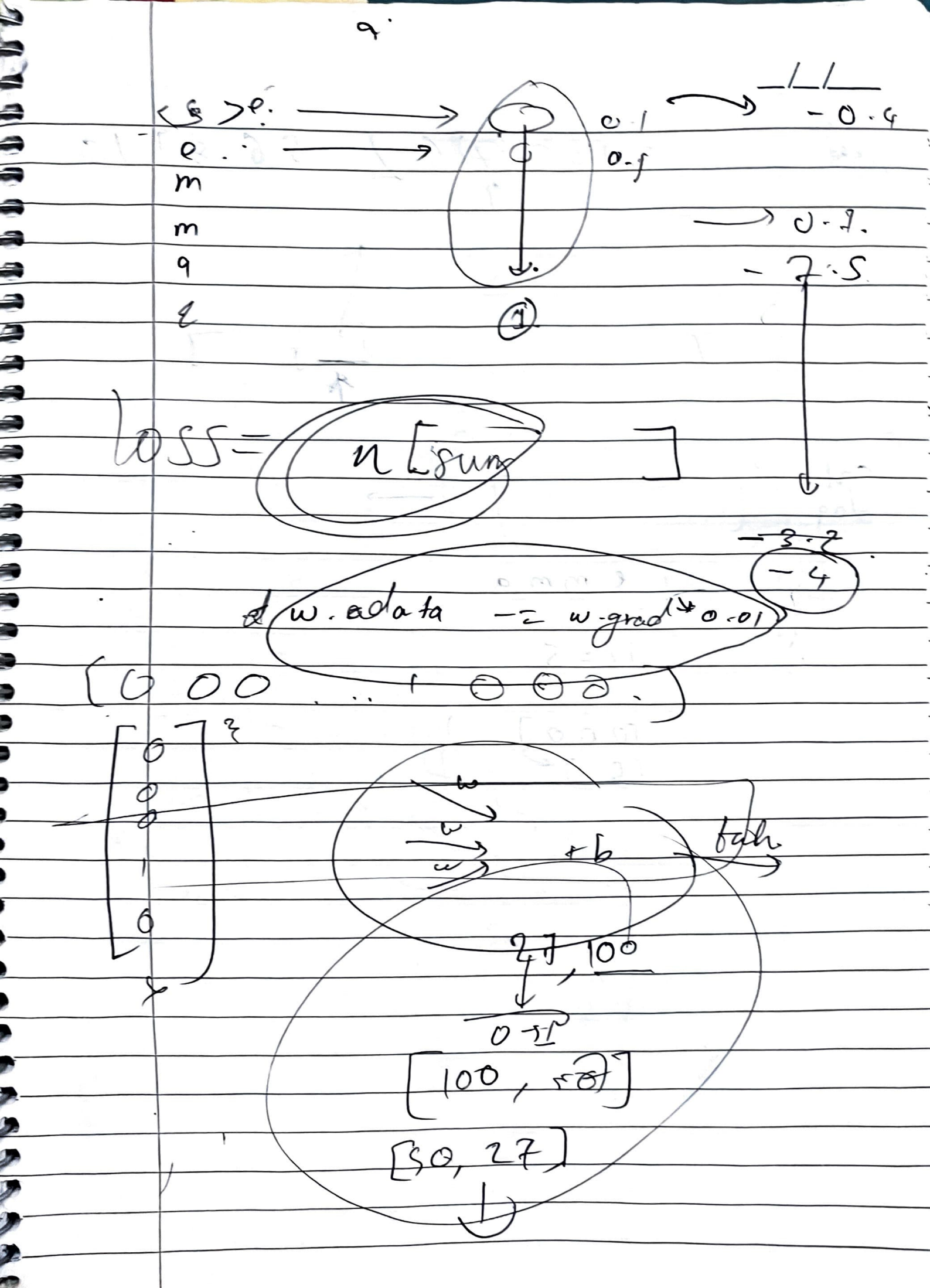
prob for next letter for each of the inputs.





each neuron gives out prob of its letter coming next


emma.
0.1
0.04
0.45
prob of
being followed
x. w. b.
t x 27
prob of 'a'
prob of 'b'
prob of 'e'
prob of each of those letters being followed by an 'a'
output =
'a' 'b' 'e' d
out [3,13]
firing rate of 13th neuron by the 3 input.
prob of 13th letter following the 3rd input.


9
0.1
0.5
-0.4
0.7
-7.5
-3.2
-4
loss = n [sum]
w.data -= w.grad * 0.01
[0 0 0 ... 1 0 0 0 ...]
w
+b
fdh
27, 100
0 1
[100, 50]
[50, 27]