Building a Bigram Language Model from Scratch

This is an extended version of Andrej Karpathy's notebook in addition to his Zero To Hero video on bigram language models.

Adapted by:

Prof. Dr.-Ing. Antje Muntzinger, University of Applied Sciences Stuttgart

antje.muntzinger@hft-stuttgart.de

We'll construct a bigram language model from scratch. The model will be trained on a text file containing names and will be able to generate new names based on what it has learned. So if you are expecting a baby and looking for some extraordinary new name, you might get some inspiration here :)

Table of Contents

1. Loading and Preprocessing the Data

First, we read the names from the text file and save them in a list.

In [1]:
```
words = open('names.txt', 'r').read().splitlines()
```

TODO: 1) Print the first 10 names. Find out the number of names contained in the dataset as well as the shortest and longest name. **(2 points)**

In [2]:
```
# YOUR CODE GOES HERE
print("First 10 names in the dataset: ", words[:10])
print("Number of names contained in the dataset: ", len(words))
print("Longest name in the dataset: ", max(words, key=len))
print("Shortest name in the dataset: ",min(words, key=len))
```

```
First 10 names in the dataset:  ['emma', 'olivia', 'ava', 'isabella', 'sophia', 'cha
rlotte', 'mia', 'amelia', 'harper', 'evelyn']
Number of names contained in the dataset:  32033
Longest name in the dataset:  muhammadibrahim
Shortest name in the dataset:  an
```

Note that in one example name like "isabella", a lot of information is contained. For example:

- "i" is likely to be the first character of a name
- "s" is likely to follow after an "i"
- "a" is likely to follow after "is"
- ...
- after "isabella", the name is likely to end.

Here we are going to create a **bigram** language model, which means we only use the single previous character to predict the next. For example, we use "s" (not "is") to predict "a" in "isabella" and forget that we have a lot more information.

2. Preprocessing the Data

First, we introduce two special characters, `<S>` for "start" and `<E>` for "end". Then we zip two consecutive characters and print them:

In [3]:
```
b = {} # dictionary to store bigram counts
for w in words: # print bigrams
    chs = ['<S>'] + list(w) + ['<E>']
    for ch1, ch2 in zip(chs, chs[1:]): # zip(['a', 'b', 'c'], ['b', 'c', 'd']) -> [
        bigram = (ch1, ch2)
        # count how many times bigram appears
```

```
        b[bigram] = b.get(bigram, 0) + 1 # b.get(bigram, 0) returns b[bigram] if bi
        print(ch1, ch2, b[bigram])
```

```
<S> e 1
e m 1
m m 1
m a 1
a <E> 1
<S> o 1
o l 1
l i 1
i v 1
v i 1
i a 1
a <E> 2
<S> a 1
a v 1
v a 1
a <E> 3
<S> i 1
i s 1
s a 1
a b 1
...truncated...
```

We see that already in the first three names, the bigram `a <E>` occurs three times - which makes sense because many names end with an "a".

In [4]:
```
# print all bigrams
b
```

In [5]:
```
# sort by the count (by default it sorts by the first element of the tuple, but we
sorted(b.items(), key = lambda kv: kv[1], reverse=True) # kv is a tuple, kv[0] is t
```

TODO: 2) What is the most common bigram in the dataset? What is the least common bigram? **(2 points)**

In [6]:
```
# YOUR CODE GOES HERE
print("most common bigram:", max(b, key=b.get))
print("least common bigram:", min(b, key=b.get))
```

```
most common bigram: ('n', '<E>')
least common bigram: ('q', 'r')
```

3. Bigram Counts as PyTorch Tensor

Next, we store the bigrams in a matrix instead of a python dictionary: The rows are going to be the first character of the bigram and the columns are going to be the second character. Each entry in this matrix will tell us how often that first character is followed by the second character in the dataset.

To create this matrix (also called **2D array** or **2D tensor**), we use PyTorch, which allows us to create multi-dimensional arrays and manipulate them very efficiently. We will also use a special character `.` for word boundaries instead of `<A>` and `<E>` because we don't need to distinguish between these two.

In [7]:
```
import torch
```

We get all characters and map them to integers in a mapping called "s to i":

In [8]:
```
chars = sorted(list(set(''.join(words)))) # get all unique characters in words
stoi = {s:i+1 for i,s in enumerate(chars)} # create a dictionary mapping characters
stoi['.'] = 0 # add a special character for word boundaries instead of <A> and <E>
```

We also create the inverse mapping "i to s" that assigns a character to each integer. We can use `items()` to get a list of a dictionary's tuples:

In [9]:
```
print(stoi)
print(stoi.items())
```

```
{'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, '.': 0}
dict_items([('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', 1
5), ('p', 16), ('q', 17), ('r', 18), ('s', 19), ('t', 20), ('u', 21), ('v', 22),
('w', 23), ('x', 24), ('y', 25), ('z', 26), ('.', 0)])
```

In [10]:
```
itos = {i:s for s,i in stoi.items()} # reverse mapping from integers to characters
```

TODO: 3a) We now have a vocabulary of 27 characters. Instanciate a torch tensor called `N` of size 27x27 with zero values of dtype int32. Check the PyTorch documentation to see how to create a tensor if needed. **(1 point)**

In [11]:
```
# YOUR CODE GOES HERE
N = torch.zeros((len(stoi), len(stoi)), dtype=torch.int32)
```

Let's fill the matrix `N` with the counts of the bigrams:

In [12]:
```
# store the counts of bigrams in the matrix
for w in words:
    chs = ['.'] + list(w) + ['.']
    for ch1, ch2 in zip(chs, chs[1:]):
        ix1 = stoi[ch1]
        ix2 = stoi[ch2]
        N[ix1, ix2] += 1
```

In [13]:
```
# print the matrix
import matplotlib.pyplot as plt
%matplotlib inline

plt.figure(figsize=(16,16))
```

```python
plt.imshow(N, cmap='Blues')
for i in range(27):
    for j in range(27):
        chstr = itos[i] + itos[j] # character string of bigram, e.g. 'ab'
        plt.text(j, i, chstr, ha="center", va="bottom", color='gray') # upper text:
        plt.text(j, i, N[i, j].item(), ha="center", va="top", color='gray') # lower
plt.axis('off')
```

Out[13]: (-0.5, 26.5, 26.5, -0.5)

.. 0	.a 4410	.b 1306	.c 1542	.d 1690	.e 1531	.f 417	.g 669	.h 874	.i 591	.j 2422	.k 2963	.l 1572	.m 2538
a. 6640	aa 556	ab 541	ac 470	ad 1042	ae 692	af 134	ag 168	ah 2332	ai 1650	aj 175	ak 568	al 2528	am 1634
b. 114	ba 321	bb 38	bc 1	bd 65	be 655	bf 0	bg 0	bh 41	bi 217	bj 1	bk 0	bl 103	bm 0
c. 97	ca 815	cb 0	cc 42	cd 1	ce 551	cf 0	cg 2	ch 664	ci 271	cj 3	ck 316	cl 116	cm 0
d. 516	da 1303	db 1	dc 3	dd 149	de 1283	df 5	dg 25	dh 118	di 674	dj 9	dk 3	dl 60	dm 30
e. 3983	ea 679	eb 121	ec 153	ed 384	ee 1271	ef 82	eg 125	eh 152	ei 818	ej 55	ek 178	el 3248	em 769
f. 80	fa 242	fb 0	fc 0	fd 0	fe 123	ff 44	fg 1	fh 1	fi 160	fj 0	fk 2	fl 20	fm 0
g. 108	ga 330	gb 3	gc 0	gd 19	ge 334	gf 1	gg 25	gh 360	gi 190	gj 3	gk 0	gl 32	gm 6
h. 2409	ha 2244	hb 8	hc 2	hd 24	he 674	hf 2	hg 2	hh 1	hi 729	hj 9	hk 29	hl 185	hm 117
i. 2489	ia 2445	ib 110	ic 509	id 440	ie 1653	if 101	ig 428	ih 95	ii 82	ij 76	ik 445	il 1345	im 427
j. 71	ja 1473	jb 1	jc 4	jd 4	je 440	jf 0	jg 0	jh 45	ji 119	jj 2	jk 2	jl 9	jm 5
k. 363	ka 1731	kb 2	kc 2	kd 2	ke 895	kf 1	kg 0	kh 307	ki 509	kj 2	kk 20	kl 139	km 9
l. 1314	la 2623	lb 52	lc 25	ld 138	le 2921	lf 22	lg 6	lh 19	li 2480	lj 6	lk 24	ll 1345	lm 60
m. 516	ma 2590	mb 112	mc 51	md 24	me 818	mf 1	mg 0	mh 5	mi 1256	mj 7	mk 1	ml 5	mm 168
n. 6763	na 2977	nb 8	nc 213	nd 704	ne 1359	nf 11	ng 273	nh 26	ni 1725	nj 44	nk 58	nl 195	nm 19
o. 855	oa 149	ob 140	oc 114	od 190	oe 132	of 34	og 44	oh 171	oi 69	oj 16	ok 68	ol 619	om 261
p. 33	pa 209	pb 2	pc 1	pd 0	pe 197	pf 1	pg 0	ph 204	pi 61	pj 1	pk 1	pl 16	pm 1
q. 28	qa 13	qb 0	qc 0	qd 0	qe 1	qf 0	qg 0	qh 0	qi 13	qj 0	qk 0	ql 1	qm 2
r. 1377	ra 2356	rb 41	rc 99	rd 187	re 1697	rf 9	rg 76	rh 121	ri 3033	rj 25	rk 90	rl 413	rm 162
s. 1169	sa 1201	sb 21	sc 60	sd 9	se 884	sf 2	sg 2	sh 1285	si 684	sj 2	sk 82	sl 279	sm 90
t. 483	ta 1027	tb 1	tc 17	td 0	te 716	tf 2	tg 2	th 647	ti 532	tj 3	tk 0	tl 134	tm 4
u. 155	ua 163	ub 103	uc 103	ud 136	ue 169	uf 19	ug 47	uh 58	ui 121	uj 14	uk 93	ul 301	um 154
v. 88	va 642	vb 1	vc 0	vd 1	ve 568	vf 0	vg 0	vh 1	vi 911	vj 0	vk 3	vl 14	vm 0
w. 51	wa 280	wb 1	wc 0	wd 8	we 149	wf 2	wg 1	wh 23	wi 148	wj 0	wk 6	wl 13	wm 2
x. 164	xa 103	xb 1	xc 4	xd 5	xe 36	xf 3	xg 0	xh 1	xi 102	xj 0	xk 0	xl 39	xm 1
y. 2007	ya 2143	yb 27	yc 115	yd 272	ye 301	yf 12	yg 30	yh 22	yi 192	yj 23	yk 86	yl 1104	ym 148
z. 160	za 860	zb 4	zc 2	zd 2	ze 373	zf 0	zg 1	zh 43	zi 364	zj 2	zk 2	zl 123	zm 35

.n 1146	.o 394	.p 515	.q 92	.r 1639	.s 2055	.t 1308	.u 78	.v 376	.w 307	.x 134	.y 535	.z 929
an 5438	ao 63	ap 82	aq 60	ar 3264	as 1118	at 687	au 381	av 834	aw 161	ax 182	ay 2050	az 435
bn 4	bo 105	bp 0	bq 0	br 842	bs 8	bt 2	bu 45	bv 0	bw 0	bx 0	by 83	bz 0
cn 0	co 380	cp 1	cq 11	cr 76	cs 5	ct 35	cu 35	cv 0	cw 0	cx 3	cy 104	cz 4
dn 31	do 378	dp 0	dq 1	dr 424	ds 29	dt 4	du 92	dv 17	dw 23	dx 0	dy 317	dz 1
en 2675	eo 269	ep 83	eq 14	er 1958	es 861	et 580	eu 69	ev 463	ew 50	ex 132	ey 1070	ez 181
fn 4	fo 60	fp 0	fq 0	fr 114	fs 6	ft 18	fu 10	fv 0	fw 4	fx 0	fy 14	fz 2
gn 27	go 83	gp 0	gq 0	gr 201	gs 30	gt 31	gu 85	gv 1	gw 26	gx 0	gy 31	gz 1
hn 138	ho 287	hp 1	hq 1	hr 204	hs 31	ht 71	hu 166	hv 39	hw 10	hx 0	hy 213	hz 20
in 2126	io 588	ip 53	iq 52	ir 849	is 1316	it 541	iu 109	iv 269	iw 8	ix 89	iy 779	iz 277
jn 2	jo 479	jp 1	jq 0	jr 11	js 7	jt 2	ju 202	jv 5	jw 6	jx 0	jy 10	jz 0
kn 26	ko 344	kp 0	kq 0	kr 109	ks 95	kt 17	ku 50	kv 2	kw 34	kx 0	ky 379	kz 2
ln 14	lo 692	lp 15	lq 3	lr 18	ls 94	lt 77	lu 324	lv 72	lw 16	lx 0	ly 1588	lz 10
mn 20	mo 452	mp 38	mq 0	mr 97	ms 35	mt 4	mu 139	mv 3	mw 2	mx 0	my 287	mz 11
nn 1906	no 496	np 5	nq 2	nr 44	ns 278	nt 443	nu 96	nv 55	nw 11	nx 6	ny 465	nz 145
on 2411	oo 115	op 95	oq 3	or 1059	os 504	ot 118	ou 275	ov 176	ow 114	ox 45	oy 103	oz 54
pn 1	po 59	pp 39	pq 0	pr 151	ps 16	pt 17	pu 4	pv 0	pw 0	px 0	py 12	pz 0
qn 0	qo 2	qp 0	qq 0	qr 1	qs 2	qt 0	qu 206	qv 0	qw 3	qx 0	qy 0	qz 0
rn 140	ro 869	rp 14	rq 16	rr 425	rs 190	rt 208	ru 252	rv 80	rw 21	rx 3	ry 773	rz 23
sn 24	so 531	sp 51	sq 1	sr 55	ss 461	st 765	su 185	sv 14	sw 24	sx 0	sy 215	sz 10
tn 22	to 667	tp 0	tq 0	tr 352	ts 35	tt 374	tu 78	tv 15	tw 11	tx 2	ty 341	tz 105
un 275	uo 10	up 16	uq 10	ur 414	us 474	ut 82	uu 3	uv 37	uw 86	ux 34	uy 13	uz 45
vn 8	vo 153	vp 0	vq 0	vr 48	vs 0	vt 0	vu 7	vv 7	vw 0	vx 0	vy 121	vz 0
wn 58	wo 36	wp 0	wq 0	wr 22	ws 20	wt 8	wu 25	wv 0	ww 2	wx 0	wy 73	wz 1
xn 1	xo 41	xp 0	xq 0	xr 0	xs 31	xt 70	xu 5	xv 0	xw 3	xx 38	xy 30	xz 19
yn 1826	yo 271	yp 15	yq 6	yr 291	ys 401	yt 104	yu 141	yv 106	yw 4	yx 28	yy 23	yz 78
zn 4	zo 110	zp 2	zq 0	zr 32	zs 4	zt 4	zu 73	zv 2	zw 3	zx 1	zy 147	zz 45

TODO: 3b) Why do we use `.item()` in the code cell above? Check out the PyTorch documentation if needed! **(1 point)**

ANSWER: YOUR ANSWER GOES HERE

The .item() method is used to extract the value of a single element from a PyTorch tensor and convert it from the tensor N[i, j] (which is a 0-dimensional tensor) to an integer before being passed to plt.text(). This is necessary because plt.text() expects standard Python data types, not PyTorch tensor objects.

Without .item(), PyTorch would return a tensor object (like tensor(5)), which might cause issues or unexpected behavior when trying to display the value using matplotlib.

TODO: 3c) Assume our first character is an 'f'. Looking at the matrix plot above, which is the most likely next character? **(1 point)**

In [14]:

```
# YOUR ANSWER GOES HERE
print(itos[torch.argmax(N[stoi['f']]).item()])
```

```
a
```

In [15]:

```
print(N[1,2])
print(N[1,2].item())
```

```
tensor(541, dtype=torch.int32)
541
```

Here is a visual summary so far - this is how we can get the most likely next character for input 'e' by plucking out the 6th row:

No description has been provided for this image

4. Sampling New Characters

Now we want to sample new characters. We start with the first character `.`, and then sample the next character based on the count of bigrams. We have the raw counts stored in the matrix N, but we need to convert them to probabilities in order to sample from the distribution.

TODO: 4a) Pluck out the first row of the matrix, convert the entries to floats, and normalize them so that the row sums to 1. **(3 points)**

HINT: You can simply cast a PyTorch tensor to float using `.float()`, see for example here: https://www.datascienceweekly.org/tutorials/pytorch-change-tensor-type-cast-a-pytorch-tensor-to-another-type

In [16]:

```
# YOUR CODE GOES HERE
p = N[0].float() # convert first row of N to float
p /= p.sum() # normalize values
print(p)
```

```
tensor([0.0000, 0.1377, 0.0408, 0.0481, 0.0528, 0.0478, 0.0130, 0.0209, 0.0273,
        0.0184, 0.0756, 0.0925, 0.0491, 0.0792, 0.0358, 0.0123, 0.0161, 0.0029,
        0.0512, 0.0642, 0.0408, 0.0024, 0.0117, 0.0096, 0.0042, 0.0167, 0.0290])
```

TODO: 4b) How can you interpret this row now? **(1 point)**

ANSWER: YOUR ANSWER GOES HERE

All values are between 0 and 1 and can be interpreted as a nominal probability.

The highest possibility is at 13.77% for .a whereas the lowest possibility lies at 0.24% for .u

To sample from this distribution, we use `torch.multinomial`, which samples from the multinomial probability distribution (in simple words: "you give me probabilities and I will give you integers sampled according to the probability distribution").

In [17]:
```
g = torch.Generator().manual_seed(2147483647) # makes the random numbers reproducib
ix = torch.multinomial(p, num_samples=1, replacement=True, generator=g).item() # sa
# replacement=True means that we are sampling with replacement (i.e. we can sample
print("sampled index:", ix) # print the sampled index
itos[ix] # convert the sampled index to a character
```

```
sampled index: 10
```

Out[17]: 'j'

TODO: 4c) Sample 100 characters according to your probability distribution for word beginnings. Print the sampled indices and characters. Can you relate the output to your probability distribution? Did you expect this output, or can you see unexpected output characters? **(2 points)**

In [18]:
```
# YOUR CODE GOES HERE
g = torch.Generator().manual_seed(2147483647)
ix_100 = torch.multinomial(first_row, num_samples=100, replacement=True, generator=
for i in ix_100:
    print(i.item(),":", itos[i.item()])
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[18], line 3
      1 # YOUR CODE GOES HERE
      2 g = torch.Generator().manual_seed(2147483647)
----> 3 ix_100 = torch.multinomial(first_row, num_samples=100, replacement=True, gen
erator=g) # use torch.multinominal with batch size of 100
      4 for i in ix_100:
      5     print(i.item(),":", itos[i.item()])

NameError: name 'first_row' is not defined
```

ANSWER: YOUR ANSWER GOES HERE

Yes, i can relate the output to probability distribution. I expect this output because the letters with highest probabilities are : a,b,c,d,e,j,k,l,m,n and they are found in output more than others. Also the probability of '..' was 0 and i have never seen it on the output.

This is the updated visualization including normalization for probabilities as outputs:

No description has been provided for this image

---

5. Broadcasting

We store all probabilities in a matrix `P`, so that each row is normalized to 1 and contains the probabilities for the next character:

In [19]:
```
P = (N+1).float()
P /= P.sum(1, keepdims=True) # normalize the rows of P (we use keepdims=True to kee
print(P)
```

```
tensor([[3.1192e-05, 1.3759e-01, 4.0767e-02, 4.8129e-02, 5.2745e-02, 4.7785e-02,
         1.3038e-02, 2.0898e-02, 2.7293e-02, 1.8465e-02, 7.5577e-02, 9.2452e-02,
         4.9064e-02, 7.9195e-02, 3.5777e-02, 1.2321e-02, 1.6095e-02, 2.9008e-03,
         5.1154e-02, 6.4130e-02, 4.0830e-02, 2.4641e-03, 1.1759e-02, 9.6070e-03,
         4.2109e-03, 1.6719e-02, 2.9008e-02],
        [1.9583e-01, 1.6425e-02, 1.5983e-02, 1.3889e-02, 3.0756e-02, 2.0435e-02,
         3.9809e-03, 4.9835e-03, 6.8796e-02, 4.8685e-02, 5.1899e-03, 1.6779e-02,
         7.4575e-02, 4.8213e-02, 1.6039e-01, 1.8872e-03, 2.4475e-03, 1.7988e-03,
         9.6279e-02, 3.2997e-02, 2.0288e-02, 1.1264e-02, 2.4623e-02, 4.7771e-03,
         5.3963e-03, 6.0480e-02, 1.2857e-02],
        [4.3039e-02, 1.2051e-01, 1.4596e-02, 7.4850e-04, 2.4701e-02, 2.4551e-01,
         3.7425e-04, 3.7425e-04, 1.5719e-02, 8.1587e-02, 7.4850e-04, 3.7425e-04,
         3.8922e-02, 3.7425e-04, 1.8713e-03, 3.9671e-02, 3.7425e-04, 3.7425e-04,
         3.1549e-01, 3.3683e-03, 1.1228e-03, 1.7216e-02, 3.7425e-04, 3.7425e-04,
         3.7425e-04, 3.1437e-02, 3.7425e-04],
        [2.7536e-02, 2.2928e-01, 2.8098e-04, 1.2082e-02, 5.6196e-04, 1.5510e-01,
         2.8098e-04, 8.4293e-04, 1.8685e-01, 7.6426e-02, 1.1239e-03, 8.9070e-02,
         3.2874e-02, 2.8098e-04, 2.8098e-04, 1.0705e-01, 5.6196e-04, 3.3717e-03,
         2.1635e-02, 1.6859e-03, 1.0115e-02, 1.0115e-02, 2.8098e-04, 2.8098e-04,
         1.1239e-03, 2.9503e-02, 1.4049e-03],
...truncated...
```

TODO: 5) Can you think of a reason why we calculate the probabilities based on `N+1` instead of `N`? **(1 point)**

HINT: What happens for `N=0`?

ANSWER: YOUR ANSWER GOES HERE

It is a so called Laplace-smoothing which makes sure that no entry is 0. This is needed not have any 0% probability for the entries without occurrence in the training data.

We need `keepdims=True` in order to sum each line, the result should be a 27x1 column vector containing the row sums - otherwise all entries get collapsed to a 1D instead of 2D array:

In [20]:
```
print(P.sum(1, keepdims=True).shape) # shape collapes to 27x1 due to summation alon
print(P.sum(1, keepdims=False).shape) # 1D tensor of size 27
```

```
torch.Size([27, 1])
torch.Size([27])
```

But why does the division above `P /= P.sum(1, keepdims=True)` even work? We divide the 27x27 matrix `P` by a 27x1 vector. This only works because PyTorch automatically applies **broadcasting**, a type of tensor manipulation: From https://pytorch.org/docs/stable/notes/broadcasting.html we get the information that two tensors are **broadcastable** if the following rules hold:

```
Each tensor has at least one dimension.

When iterating over the dimension sizes, starting at the trailing
dimension, the dimension sizes must either be equal, one of them is
1, or one of them does not exist.
```

If two tensors x, y are "broadcastable", the resulting tensor size is calculated as follows:

```
If the number of dimensions of x and y are not equal, prepend 1 to
the dimensions of the tensor with fewer dimensions to make them
equal length.

Then, for each dimension size, the resulting dimension size is the
max of the sizes of x and y along that dimension.
```

In our case, broadcasting takes the 27x1 column vector and copies it 27 times to get a 27x27 matrix, then it makes an element-wise division. We can check the result:

In [21]:
```
# check that the first row sums to 1
print(P[0].sum())
```

```
tensor(1.)
```

TODO (optional): What happens if you remove `keepdims=True`? Will the division still result in the same matrix - why or why not?

HINT: Don't forget to revert your changes for later cells to work right - or copy the code cell and use different variable names.

ANSWER: YOUR ANSWER GOES HERE

If we remove it, it will return 1D array(each element is sum of one row),then when we divide the tensor(27,27) with sum(27). As it didnt protect the dimension features, the division would be wrong(maybe column wise). In the code below, as there is a mistake in calculation, sum of first row is not 1 :

In [22]:
```
# YOUR CODE GOES HERE

P_v2 = (N+1).float()
P_v2 /= P_v2.sum(1)
print("If it would normalized correctly sum supposed to be 1 but : ",P_v2[0].sum())

# Let us see in a small example
temp=torch.tensor([[1,2,3],[4,5,6],[7,8,9]]).float()
print("\nwithout keepdim=False :\n ",temp.sum(1, keepdim=False) , "\n normalized:\n

print("\nwith keepdim=True :\n",temp.sum(1,keepdim=True) , "\n normalized:\n" , tem
```

```
If it would normalized correctly sum supposed to be 1 but :  tensor(6.9246)

without keepdim=False :
  tensor([ 6., 15., 24.])
 normalized:
 tensor([[0.1667, 0.1333, 0.1250],
        [0.6667, 0.3333, 0.2500],
        [1.1667, 0.5333, 0.3750]])

with keepdim=True :
 tensor([[ 6.],
        [15.],
        [24.]])
 normalized:
 tensor([[0.1667, 0.3333, 0.5000],
        [0.2667, 0.3333, 0.4000],
        [0.2917, 0.3333, 0.3750]])
```

6. Generating New Names

With this, we can sample new names by starting with the start character `.` and sampling the next character, then the next next character and so on, until we sample the end character `.`:

In [23]:
```
g = torch.Generator().manual_seed(2147483647)

for i in range(20): # generate 20 names

    out = []
    ix = 0 # start with the start-of-word character
    while True:
        p = P[ix] # get the probabilities of the next character by plucking the row
        ix = torch.multinomial(p, num_samples=1, replacement=True, generator=g).ite
        out.append(itos[ix]) # convert the sampled index to a character and append
        if ix == 0: # if we sampled the end-of-word character: break
            break
    print(''.join(out)) # print the generated name
```

```
junide.
janasah.
p.
cony.
a.
nn.
kohin.
tolian.
juee.
ksahnaauranilevias.
dedainrwieta.
ssonielylarte.
faveumerifontume.
phynslenaruani.
core.
yaenon.
ka.
jabdinerimikimaynin.
anaasn.
ssorionsush.
...truncated...
```

This doesn't look right... but it is! The bigram language model is simply not powerful enough to create more reasonable names. For example, it generated a name "a" twice, but it doesn't know that "a" is the first character here. All it knows is that "a" is a character in the name and how likely it is that the name ends here, without looking at previous characters.

TODO: 6) In the code cell above, experiment with replacing `p` with the uniform distribution, making everything equally likely. Can you see that the bigram model works better than pure chance? **(2 points)**

HINT: Don't forget to revert your changes for later cells to work right - or copy the code cell and use different variable names.

In [24]:

```
g = torch.Generator().manual_seed(2147483647)

for i in range(20):  # generate 20 names
    out = []
    ix = 0  # start with the start-of-word character

    while True:
        # Replace `p` with a uniform distribution (all probabilities equal)
        p = torch.ones_like(P[ix])  # Create a vector with equal probabilities for
        p /= p.sum()  # Normalize so the sum of probabilities is 1 (uniform distrib

        # Sample the next character
        ix = torch.multinomial(p, num_samples=1, replacement=True, generator=g).ite
        out.append(itos[ix])  # Add the character to the output list

        # Break the loop if the end-of-word character `.` is sampled
        if ix == 0:
            break
```

```python
    print(''.join(out))
```

```
juwjdvdipkcqaz.
p.
cfqywocnzqfjiirltozcogsjgwzvudlhnpauyjbilevhajkdbduinrwibtlzsnjyievyvaftbzffvmumthyf
odtumjrpfytszwjhrjagq.
coreaysezocfkyjjabdywejfmoifmwyfinwagaasnhsvfihofszxhddgosfmptpagicz.
rjpiufmthdt.
rkrrsru.
iyumuyfy.
mjekujcbkhvupwyhvpvhvccragr.
wdkhwfdztta.
mplyisbxlyhuuiqzavmpocbzthqmimvyqwat.
f.
.
ndxjxfpvslqtikyzsaloevgvvnundewkfmbjzqegruxiteaxchwtmurzsodridcdznojvaliypvrghvxtezr
wguciqqvywhqelv.
viosvhibdhnceukgmtmwboscnbzoiwupnwnpipixtewbgsgyewfdacbfcxrvjypkmsbranmjrdsydotafvkd
kbdepihzpwzsqdab.
vfuolwbasrtugttbiqbujfdtskceqjtcdlcndfujqllsppgkltalmlokdmsl.
fddmxjv.
mfsgxmw.
vdihkvngtojvrdsyqivcob.
uziengogtjvnvqgfjtkqufrxfjlwglykiiluohgnoiuwzylq.
fsgircvmhtipagkxwvjypnsriadmfujnlkcicvatjvryzeljxkbrlrjsp.
...truncated...
```

7. Evaluating the Model

We want to evaluate the quality of the model using a single number, the **training loss**. But which single number to use? Let's print the probabilities of the first bigrams to start with:

In [25]:
```python
for w in words[:3]: # print bigram probabilities for the first 3 words
    chs = ['.'] + list(w) + ['.'] # add start and end of word characters
    for ch1, ch2 in zip(chs, chs[1:]): # iterate over bigrams
        ix1 = stoi[ch1] # get the integer representation of the characters
        ix2 = stoi[ch2] # get the integer representation of the characters
        prob = P[ix1, ix2] # get the probability of the bigram
        print(f'{ch1}{ch2}: {prob:.4f}') # print the bigram and its probability
```

```
.e: 0.0478
em: 0.0377
mm: 0.0253
ma: 0.3885
a.: 0.1958
.o: 0.0123
ol: 0.0779
li: 0.1774
iv: 0.0152
vi: 0.3508
ia: 0.1380
a.: 0.1958
.a: 0.1376
av: 0.0246
va: 0.2473
a.: 0.1958
```

TODO: 7a) What probabilities would we expect for an untrained model? How do you interpret these outputs compared to the untrained model? **(2 points)**

ANSWER: YOUR ANSWER GOES HERE

For an untrained model, we would expect all bigram probabilities to be equal. This is because, in an untrained model, we have no prior information about the likelihood of different character transitions.

If there are 27 possible characters (including the start and end markers), each bigram probability in an untrained model would be close to 1/27 ≈ 0.037

So how can we summarize these probabilities in a single number? First of all, we can use the **likelihood**, which is the product of all these probabilities. The likelihood tells us which probability our model assigns to the whole dataset. So we want the likelihood to be as big as possible for a good model (**maximum likelihood estimation**).

TODO: 7b) Can you think of a problem when calculating the likelihood as defined above? **(1 point)**

HINT: What happens numerically if we multiply many tiny numbers?

ANSWER: YOUR ANSWER GOES HERE

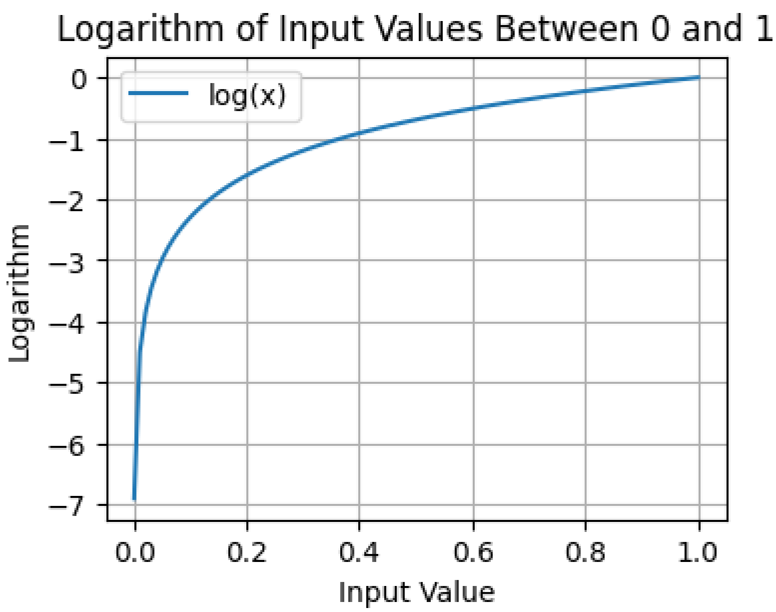
Numerical underflow becomes a problem. Since probabilities are often small values (less than 1), multiplying a large number of them can result in an extremely small product, potentially too small for the computer to represent accurately. This can lead to the calculated likelihood being rounded to zero, even when it's non-zero, causing a loss of precision in computations.

Instead of the likelihood, we use the **log likelihood**: Applying the logarithm transforms the likelihood to a value in $[-\infty, 0]$.

In [26]:
```
# Generate input values between 0 and 1
x = torch.linspace(0.001, 1, 100)  # Avoid log(0) by starting from 0.001

# Compute the logarithm of the input values
y = torch.log(x)

# Create the plot
plt.figure(figsize=(4, 3))
plt.plot(x, y, label='log(x)')
plt.title('Logarithm of Input Values Between 0 and 1')
plt.xlabel('Input Value')
plt.ylabel('Logarithm')
plt.legend()
plt.grid(True)
plt.show()
```



TODO: 7c) Why is maximizing the likelihood equivalent to maximizing the log likelihood? **(1 point)**

ANSWER: YOUR ANSWER GOES HERE

Because the logarithm is a monotonically increasing function. This means that if you increase the likelihood, its logarithm also increases, and vice versa.

Instead of maximizing the log likelihood, in practice we minimize the negative log likelihood, because a loss usually is a number in $[0, \infty]$, where small numbers are good. Finally, the negative log likelihood is simply the sum of the single logarithms due to

$$\log(a * b * c) = \log(a) + \log(b) + \log(c)$$

This is great because a single probability close to zero will not cause the whole loss to be tiny any more, we now use addition instead of multiplication!

TODO: 7d) Calculate the **negative log likelihood loss**, store it in `nll` and print it. Sometimes, we also use the **average negative log likelihood loss**, so average `nll` by the number of bigrams evaluated and print the average as well. **(3 points)**

HINT: You can start by copying the second last code cell above, where we calculated the probabilities of bigrams. Then apply `torch.log` to these probabilities and sum the negative logarithms.

In [27]:
```
# YOUR CODE GOES HERE

nll = 0  # initialize negative log likelihood loss
num_bigrams = 0  # counter for the total number of bigrams

for w in words:  # loop over each word in the dataset
    chs = ['.'] + list(w) + ['.']  # add start and end markers to each word
    for ch1, ch2 in zip(chs, chs[1:]):  # iterate through each bigram in the word
        ix1 = stoi[ch1]  # index for first character
        ix2 = stoi[ch2]  # index for second character
        prob = P[ix1, ix2]  # get the probability of the bigram
        nll += -torch.log(prob)  # add the negative log probability to nll
        num_bigrams += 1  # increment the bigram count

# Average negative log likelihood
avg_nll = nll / num_bigrams

# Print results
print("Negative Log Likelihood Loss:", nll.item())
print("Average Negative Log Likelihood Loss:", avg_nll.item())
```

```
Negative Log Likelihood Loss: 559951.5625
Average Negative Log Likelihood Loss: 2.4543561935424805
```

This summarizes our first bigram model, which simply counts occurences of bigrams and generates new names based on the corresponding probability distribution. The average log likelihood loss of this model is around 2.45.

The following visualization includes the whole bigram model with negative log likelihood loss:

No description has been provided for this image

---

8. The Neural Network Approach

Let's try another approach: a neural network! Our neural network will still be a bigram language model: It receives a single character as an input, processes it using some weights or some parameters w and outputs the probability distribution over the next character. We will evaluate the parameters of the neural net using the negative log likelihood loss, which means comparing the output probability distribution and the label (the identity of the next character). We are going to use gradient-based optimization to tune the parameters of this network to minimize the loss with the goal to better predict the next character. In the end, the result will look quite similar to the intuitive approach counting occurances of bigrams above!

First, let's create the training set for the neural network made of two lists, the inputs and the targets:

In [28]:
```
# create the training set of bigrams (x,y)
xs, ys = [], []

for w in words[:1]: # only use the first word for now, which contains 5 examples
    chs = ['.'] + list(w) + ['.']
    for ch1, ch2 in zip(chs, chs[1:]):
        ix1 = stoi[ch1]
        ix2 = stoi[ch2]
        print(ch1, ch2)
        xs.append(ix1) # input is the first character of the bigram
        ys.append(ix2) # target is the second character of the bigram

xs = torch.tensor(xs) # convert the list to a tensor
ys = torch.tensor(ys) # convert the list to a tensor
```

```
. e
e m
m m
m a
a .
```

We print the 5 resulting inputs and targets:

In [29]:
```
xs
```

Out[29]:
```
tensor([ 0,  5, 13, 13,  1])
```

In [30]:
```
ys
```

Out[30]:
```
tensor([ 5, 13, 13,  1,  0])
```

TODO: 8) The following code cell applies **one-hot-encoding**. Shortly explain what this is and why it is needed! **(2 points)**

ANSWER: YOUR ANSWER GOES HERE

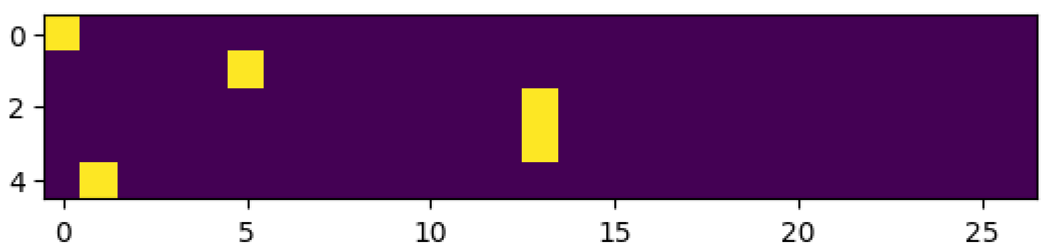
One-hot encoding allows us to convert categorical data (like characters) into a numerical format that can be used as input for neural networks. Neural networks expect numerical input, and one-hot vectors ensure each character has a unique, distinguishable vector representation without introducing any inherent ordering or numerical relationships between characters.

In [31]:
```python
# apply one-hot encoding to the input
import torch.nn.functional as F
xenc = F.one_hot(xs, num_classes=27).float()
print(xenc)
print(xenc.shape)
```

```
tensor([[1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
         0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [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., 0., 0., 0., 0., 0., 0., 0., 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., 0., 0., 0., 0., 1., 0., 0., 0., 0.,
         0., 0., 0., 0., 0., 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., 0., 0., 0.]])
torch.Size([5, 27])
```

In [32]:
```python
# we can also visualize the one-hot encoding like this:
plt.imshow(xenc)
```

Out[32]: <matplotlib.image.AxesImage at 0x1b37488fe60>



In [33]:
```python
# check the data type of the one-hot encoding
xenc.dtype
```

Out[33]: torch.float32

9. The First Neuron

A neuron consists of a simple dot product $x \cdot W + b$. We don't use a **bias** b here, so let's first initialize the **weights** W randomly sampled from a normal distribution, i.e., most weights will be around 0:

In [34]:
```python
W = torch.randn((27, 1))
W
```

TODO: 9) Apply the matrix multiplication (denoted by `@` in PyTorch) to get the **logits**. Store the result in a variable called `logits`. Which shape will the result have and why? Check your predicted shape by printing the result! **(2 points)**

In [35]:
```python
# YOUR CODE GOES HERE
```

```
logits = xenc @ W
print("Shapr of xenc : ", xenc.shape)
print("Shape of W :",W.shape)
print("Shape of logits : ",logits.shape)
```

```
Shapr of xenc :  torch.Size([5, 27])
Shape of W : torch.Size([27, 1])
Shape of logits :  torch.Size([5, 1])
```

ANSWER: YOUR ANSWER GOES HERE

When we multiply xenc (of shape (num_bigrams, 27)) by W (of shape (27, 1)), the resulting shape will be (num_bigrams, 1). Each row in logits will represent the output (or logit) for one bigram in the dataset.

Since our num_bigrams size is 5, the result is of shape (5,1).

10. Creating 27 Neurons

Now we want 27 neurons instead of just one, because we eventually want to output the 27 probabilities for each character to be the next. So we want the weight matrix to be 27x27 instead of 27x1, and we will in parallel evaluate the 5 inputs on 27 neurons, so the output will be 5x27 (matrix multiplication of a 5x27 input with 27x27 weight matrix).

In [36]:
```
# randomly initialize 27 neurons' weights. Each neuron receives 27 inputs
g = torch.Generator().manual_seed(2147483647)
W = torch.randn((27, 27), generator=g)
print(W)
```

```
tensor([[ 1.5674e+00, -2.3729e-01, -2.7385e-02, -1.1008e+00,  2.8588e-01,
         -2.9643e-02, -1.5471e+00,  6.0489e-01,  7.9136e-02,  9.0462e-01,
         -4.7125e-01,  7.8682e-01, -3.2843e-01, -4.3297e-01,  1.3729e+00,
          2.9334e+00,  1.5618e+00, -1.6261e+00,  6.7716e-01, -8.4039e-01,
          9.8488e-01, -1.4837e-01, -1.4795e+00,  4.4830e-01, -7.0730e-02,
          2.4968e+00,  2.4448e+00],
        [-6.7006e-01, -1.2199e+00,  3.0314e-01, -1.0725e+00,  7.2762e-01,
          5.1114e-02,  1.3095e+00, -8.0220e-01, -8.5042e-01, -1.8068e+00,
          1.2523e+00, -1.2256e+00,  1.2165e+00, -9.6478e-01, -2.3211e-01,
         -3.4762e-01,  3.3244e-01, -1.3263e+00,  1.1224e+00,  5.9641e-01,
          4.5846e-01,  5.4011e-02, -1.7400e+00,  1.1560e-01,  8.0319e-01,
          5.4108e-01, -1.1646e+00],
        [ 1.4756e-01, -1.0006e+00,  3.8012e-01,  4.7328e-01, -9.1027e-01,
         -7.8305e-01,  1.3506e-01, -2.1161e-01, -1.0406e+00, -1.5367e+00,
          9.3743e-01, -8.8303e-01,  1.7457e+00,  2.1346e+00, -8.5614e-01,
          5.4082e-01,  6.1690e-01,  1.5160e+00, -1.0447e+00, -6.6414e-01,
         -7.2390e-01,  1.7507e+00,  1.7530e-01,  9.9280e-01, -6.2787e-01,
          7.7023e-02, -1.1641e+00],
        [ 1.2473e+00, -2.7061e-01, -1.3635e+00,  1.3066e+00,  3.2307e-01,
          1.0358e+00, -8.6249e-01, -1.2575e+00,  9.4180e-01, -1.3257e+00,
...truncated...
```

In [37]:
```
xenc = F.one_hot(xs, num_classes=27).float() # input to the network: one-hot encodi
logits = xenc @ W # predict log-counts
```

```
print(logits)
```

```
tensor([[ 1.5674e+00, -2.3729e-01, -2.7385e-02, -1.1008e+00,  2.8588e-01,
         -2.9643e-02, -1.5471e+00,  6.0489e-01,  7.9136e-02,  9.0462e-01,
         -4.7125e-01,  7.8682e-01, -3.2843e-01, -4.3297e-01,  1.3729e+00,
          2.9334e+00,  1.5618e+00, -1.6261e+00,  6.7716e-01, -8.4039e-01,
          9.8488e-01, -1.4837e-01, -1.4795e+00,  4.4830e-01, -7.0730e-02,
          2.4968e+00,  2.4448e+00],
        [ 4.7236e-01,  1.4830e+00,  3.1748e-01,  1.0588e+00,  2.3982e+00,
          4.6827e-01, -6.5650e-01,  6.1662e-01, -6.2197e-01,  5.1007e-01,
          1.3563e+00,  2.3445e-01, -4.5585e-01, -1.3132e-03, -5.1161e-01,
          5.5570e-01,  4.7458e-01, -1.3867e+00,  1.6229e+00,  1.7197e-01,
          9.8846e-01,  5.0657e-01,  1.0198e+00, -1.9062e+00, -4.2753e-01,
         -2.1259e+00,  9.6041e-01],
        [ 1.9359e-01,  1.0532e+00,  6.3393e-01,  2.5786e-01,  9.6408e-01,
         -2.4855e-01,  2.4756e-02, -3.0404e-02,  1.5622e+00, -4.4852e-01,
         -1.2345e+00,  1.1220e+00, -6.7381e-01,  3.7882e-02, -5.5881e-01,
         -8.2709e-01,  8.2253e-01, -7.5100e-01,  9.2778e-01, -1.4849e+00,
         -2.1293e-01, -1.1860e+00, -6.6092e-01, -2.3348e-01,  1.5447e+00,
          6.0061e-01, -7.0909e-01],
        [ 1.9359e-01,  1.0532e+00,  6.3393e-01,  2.5786e-01,  9.6408e-01,
         -2.4855e-01,  2.4756e-02, -3.0404e-02,  1.5622e+00, -4.4852e-01,
...truncated...
```

These numbers tell us the firing rate of each of the 27 neurons on the 5 inputs. For example, the result at row 3 and column 13 is giving us the firing rate of the 13th neuron looking at the third input:

In [38]:
```
(xenc @ W)[3,13]
```

Out[38]:
```
tensor(0.0379)
```

This is equivalent to a dot product between the third input and the 13th column of W, so using matrix multiplication we can very efficiently evaluate the dot product between lots of input examples in a batch and lots of neurons:

In [39]:
```
(xenc[3] * W[:,13]).sum() # element-wise multiplication and sum is equivalent to ma
```

Out[39]:
```
tensor(0.0379)
```

Here we can see the steps so far, including one-hot-encoding, weights and logit calculation for input 'e':

No description has been provided for this image

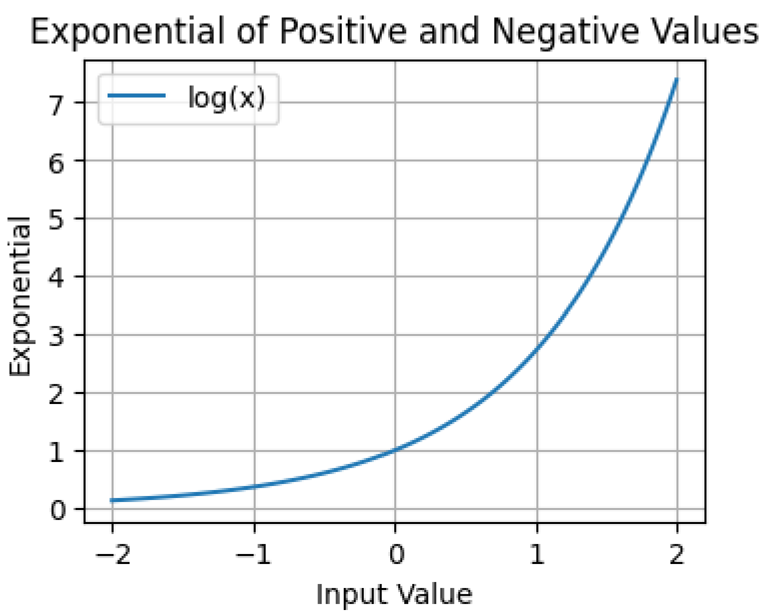
Note that in the image we only plot a single input 'e', whereas in the code we process 5 inputs in parallel.

We see that the logits are positive and negative. They can be interpreted as **log counts** (=logarithms of the counts of each bigram). We can elementwise exponentiate to transform them into numbers in $(0, \infty)$, where negative numbers go to $(0, 1)$ and positive numbers go to $(1, \infty)$:

In [40]:
```python
# Generate input values between -2 and 2
x = torch.linspace(-2, 2, 100)  #

# Compute the exponential of the input values
y = torch.exp(x)

# Create the plot
plt.figure(figsize=(4, 3))
plt.plot(x, y, label='log(x)')
plt.title('Exponential of Positive and Negative Values')
plt.xlabel('Input Value')
plt.ylabel('Exponential')
plt.legend()
plt.grid(True)
```



TODO: 10a) Calculate the counts from the log counts by applying exponential function. Store the result in `counts` and print it. Normalize the counts like above, store the result in `probs` and print it. **(2 points)**

In [41]:
```python
# YOUR CODE GOES HERE

# Calculate counts from log counts
counts = torch.exp(logits)  # Apply exponential function to each element in logits
print("Counts:\n", counts)

# Normalize counts to get probabilities
probs = counts / counts.sum(dim=1, keepdims=True)  # Sum across rows and divide
print("Probabilities:\n", probs)
```

```
Counts:
 tensor([[ 4.7940,  0.7888,  0.9730,  0.3326,  1.3309,  0.9708,  0.2129,  1.8311,
           1.0824,  2.4710,  0.6242,  2.1964,  0.7200,  0.6486,  3.9469, 18.7908,
           4.7673,  0.1967,  1.9683,  0.4315,  2.6775,  0.8621,  0.2277,  1.5656,
           0.9317, 12.1434, 11.5281],
         [ 1.6038,  4.4060,  1.3737,  2.8830, 11.0032,  1.5972,  0.5187,  1.8527,
           0.5369,  1.6654,  3.8818,  1.2642,  0.6339,  0.9987,  0.5995,  1.7432,
           1.6073,  0.2499,  5.0680,  1.1876,  2.6871,  1.6596,  2.7728,  0.1486,
           0.6521,  0.1193,  2.6128],
         [ 1.2136,  2.8669,  1.8850,  1.2942,  2.6224,  0.7799,  1.0251,  0.9701,
           4.7691,  0.6386,  0.2910,  3.0710,  0.5098,  1.0386,  0.5719,  0.4373,
           2.2763,  0.4719,  2.5289,  0.2265,  0.8082,  0.3054,  0.5164,  0.7918,
           4.6866,  1.8232,  0.4921],
         [ 1.2136,  2.8669,  1.8850,  1.2942,  2.6224,  0.7799,  1.0251,  0.9701,
           4.7691,  0.6386,  0.2910,  3.0710,  0.5098,  1.0386,  0.5719,  0.4373,
           2.2763,  0.4719,  2.5289,  0.2265,  0.8082,  0.3054,  0.5164,  0.7918,
           4.6866,  1.8232,  0.4921],
         [ 0.5117,  0.2953,  1.3541,  0.3422,  2.0701,  1.0524,  3.7043,  0.4483,
           0.4272,  0.1642,  3.4984,  0.2936,  3.3753,  0.3811,  0.7929,  0.7064,
           1.3944,  0.2655,  3.0723,  1.8156,  1.5816,  1.0555,  0.1755,  1.1225,
...truncated...
```

Here are the visualized steps including exponentiation and normalization:

No description has been provided for this image

TODO: 10b) Do the last two maths operations in the cell above look familiar? How is this transformation called? **(1 point)**

ANSWER: YOUR ANSWER GOES HERE

Yes, the last tow math operations combined are known as the softmax transformation.

Let's take a look at the resulting "probabilities" for the first character (of course they are meaningless because the network is still untrained, the weights are randomly initialized):

In [42]:
```
probs[0]
```

In [43]:
```
probs[0].shape
```

Out[43]:
```
torch.Size([27])
```

In [44]:
```
probs.shape
```

Out[44]:
```
torch.Size([5, 27])
```

11. Summary (so far)

Let's summarize the code so far to get an overview of the steps:

In [45]:
```
xs # input to the network ('.emma')
```

Out[45]: tensor([0, 5, 13, 13, 1])

In [46]:
```
ys # target for the network ('emma.')
```

Out[46]: tensor([5, 13, 13, 1, 0])

In [47]:
```
# randomly initialize 27 neurons' weights. each neuron receives 27 inputs
g = torch.Generator().manual_seed(2147483647)
W = torch.randn((27, 27), generator=g)
```

In [48]:
```
xenc = F.one_hot(xs, num_classes=27).float() # input to the network: one-hot encodi
logits = xenc @ W # predict log-counts
counts = logits.exp() # counts, equivalent to N
probs = counts / counts.sum(1, keepdims=True) # probabilities for next character
```

In [49]:
```
probs.shape # output probabilities of shape 5x27
```

Out[49]: torch.Size([5, 27])

In [50]:
```
# compute the negative log likelihood loss for the first 5 bigrams
nlls = torch.zeros(5)
for i in range(5):
    # i-th bigram:
    x = xs[i].item() # input character index
    y = ys[i].item() # label character index
    print('--------')
    print(f'bigram example {i+1}: {itos[x]}{itos[y]} (indices {x},{y})')
    print('input to the neural net:', x)
    print('output probabilities from the neural net:', probs[i])
    print('label (actual next character):', y)
    p = probs[i, y]
    print('probability assigned by the net to the the correct character:', p.item()
    logp = torch.log(p)
    print('log likelihood:', logp.item())
    nll = -logp
    print('negative log likelihood:', nll.item())
    nlls[i] = nll

print('=========')
print('average negative log likelihood, i.e. loss =', nlls.mean().item())
```

```
--------
bigram example 1: .e (indices 0,5)
input to the neural net: 0
output probabilities from the neural net: tensor([0.0607, 0.0100, 0.0123, 0.0042, 0.
0168, 0.0123, 0.0027, 0.0232, 0.0137,
        0.0313, 0.0079, 0.0278, 0.0091, 0.0082, 0.0500, 0.2378, 0.0603, 0.0025,
        0.0249, 0.0055, 0.0339, 0.0109, 0.0029, 0.0198, 0.0118, 0.1537, 0.1459])
label (actual next character): 5
probability assigned by the net to the the correct character: 0.01228625513613224
log likelihood: -4.399273872375488
negative log likelihood: 4.399273872375488
--------
bigram example 2: em (indices 5,13)
input to the neural net: 5
output probabilities from the neural net: tensor([0.0290, 0.0796, 0.0248, 0.0521, 0.
1989, 0.0289, 0.0094, 0.0335, 0.0097,
        0.0301, 0.0702, 0.0228, 0.0115, 0.0181, 0.0108, 0.0315, 0.0291, 0.0045,
        0.0916, 0.0215, 0.0486, 0.0300, 0.0501, 0.0027, 0.0118, 0.0022, 0.0472])
label (actual next character): 13
probability assigned by the net to the the correct character: 0.018050700426101685
log likelihood: -4.014570713043213
negative log likelihood: 4.014570713043213
...truncated...
```

Here is a visualization of the first feedforward pass of input 'e' through the neuron, including negative log likelihood loss calculation. You can see in the output above that the loss is roughly 4 (see 'bigram example 2' output).

No description has been provided for this image

We see from the output above that the average loss is 3.7, which is quite high. We can change the random seed for sampling W for randomly changing the loss. But of course, we can do better: The loss calculation is only made up of differentiable operations (multiplication, addition, exponential, division...) We can minimize the loss by computing the gradients of the loss with respect to W.

12. Optimization

To calculate the loss, we basically need to pluck out the predicted probabilities at the indices of the target characters. For example, for the first bigram, the input character is '.' (index 0) and the target character is 'e' (index 5), and the predicted probability is stored in `probs[0,5]`.

In [51]:
```
# we need the following probabilities for calculating the loss:
probs[0,5], probs[1,13], probs[2,13], probs[3,1], probs[4,0]
```

Out[51]:
```
(tensor(0.0123),
 tensor(0.0181),
 tensor(0.0267),
 tensor(0.0737),
 tensor(0.0150))
```

This is equivalent to:

In [52]:
```
probs[torch.arange(5), ys]
```

Out[52]:
```
tensor([0.0123, 0.0181, 0.0267, 0.0737, 0.0150])
```

We can use this in the loss calculation at the end of the forward pass:

In [53]:
```
# randomly initialize 27 neurons' weights. each neuron receives 27 inputs
g = torch.Generator().manual_seed(2147483647)
W = torch.randn((27, 27), generator=g, requires_grad=True) # we need to compute gra
```

In [54]:
```
# forward pass
xenc = F.one_hot(xs, num_classes=27).float() # input to the network: one-hot encodi
logits = xenc @ W # predict log-counts
counts = logits.exp() # counts, equivalent to N
probs = counts / counts.sum(1, keepdims=True) # probabilities for next character
loss = -probs[torch.arange(5), ys].log().mean() # average negative log likelihood l
```

In [55]:
```
print(loss.item()) # print the loss
```

```
3.7693049907684326
```

Note that the loss is the same as above. Finally we can do the backward pass:

In [56]:
```
# backward pass
W.grad = None # set to zero the gradient (None is a special value that PyTorch reco
loss.backward() # compute the gradients
```

Something magical happened when `loss.backward()` was run: PyTorch keeps track of all the operations in the forward pass, under the hood it builds a full computational graph, so it knows all the dependencies and all the mathematical operations inside the neural network. When we calculate the loss and call a `.backward()` on it, PyTorch fills in the partial derivatives of all the intermediate steps. So now we can look at the gradient and see that it is not zero anymore:

In [57]:
```
W.grad
```

In [58]:
```
W.shape, W.grad.shape
```

Out[58]:
```
(torch.Size([27, 27]), torch.Size([27, 27]))
```

Both `W` and `W.grad` are of shape 27x27, and each entry of `W.grad` is telling us the influence of that weight on the loss function. For example, since `W.grad[0,0]` is positive, slightly increasing `W[0,0]` will lead to a slightly bigger loss. Therefore we use the *negative* gradient for updating the weights:

In [59]:
```
W.data += -0.1 * W.grad # update the weights with learning rate 0.1
```

Let's check that the loss has really decreased using this one **gradient descent** step:

In [60]:
```
# forward pass
xenc = F.one_hot(xs, num_classes=27).float() # input to the network: one-hot encodi
logits = xenc @ W # predict log-counts
counts = logits.exp() # counts, equivalent to N
probs = counts / counts.sum(1, keepdims=True) # probabilities for next character
loss = -probs[torch.arange(5), ys].log().mean()
print(loss.item())
```

```
3.7492127418518066
```

13. Optimizing over the Whole Dataset

Now let's apply gradient descent to the whole dataset:

In [61]:
```
# create the dataset
xs, ys = [], []
for w in words:
    chs = ['.'] + list(w) + ['.']
    for ch1, ch2 in zip(chs, chs[1:]):
        ix1 = stoi[ch1]
        ix2 = stoi[ch2]
        xs.append(ix1)
        ys.append(ix2)
xs = torch.tensor(xs)
ys = torch.tensor(ys)
num = xs.nelement() # number of elements in the tensor xs (5 in '.emma.', here more
print('number of examples: ', num)

# initialize the 'network'
g = torch.Generator().manual_seed(2147483647)
W = torch.randn((27, 27), generator=g, requires_grad=True)
```

```
number of examples:  228146
```

In [62]:
```
# gradient descent
for k in range(500):

    # forward pass
    xenc = F.one_hot(xs, num_classes=27).float() # input to the network: one-hot en
    logits = xenc @ W # predict log-counts - one-hot input will pluck the correct r
    counts = logits.exp() # counts, equivalent to N
    probs = counts / counts.sum(1, keepdims=True) # probabilities for next characte
    loss = -probs[torch.arange(num), ys].log().mean() + 0.01*(W**2).mean()
    print(loss.item())

    # backward pass
```

```
    W.grad = None # set to zero the gradient
    loss.backward()

    # update
    W.data += -50 * W.grad # even a large learning rate is fine here, because the b
```

```
3.7686190605163574
3.3788068294525146
3.161090850830078
3.027186155319214
2.9344842433929443
2.8672313690185547
2.816654682159424
2.777146577835083
2.745253801345825
2.7188303470611572
2.696505546569824
2.6773719787597656
2.6608052253723145
2.6463515758514404
2.633665084838867
2.622471570968628
2.6125476360321045
2.6037068367004395
2.595794916152954
2.5886809825897217
...truncated...
```

We see that the loss is converging roughly to 2.48, which is similar to the loss we achieved all the way up in the original bigram model without a neural network (2.45). Back then, we achieved this loss by counting, now by optimizing weights. That makes sense because fundamentally we're not using any additional information, we're still just taking in the previous character and trying to predict the next one, but instead of doing it explicitly by counting and normalizing we are doing it with gradient-based learning. The explicit approach happens to very well optimize the loss function without any need for a gradient based optimization because the setup for bigram language models is so straightforward, we can just afford to estimate those probabilities directly and maintain them in a table. But the gradient-based approach is significantly more flexible, so we've actually gained a lot because we can expand this approach and complexify the neural net.

TODO (optional): Remember the model smoothing used in the first bigram model by adding a number to the counts to avoid zero probabilities. Can you think of a smoothing equivalent in gradient descent based bigram model?

HINT: Think of the weight matrix W and its influence on the result!

ANSWER: YOUR ANSWER GOES HERE

The code above already includes a smoothing:

`loss = -probs[torch.arange(num), ys].log().mean() + 0.01*(W**2).mean()`

While the last part `+ 0.01*(W**2).mean()` is a regularization term which is proportional to `W**2` which penalizes large values in `W` and encourages a smoother and less extreme distribution of weights.

Let's see how the weights and the probabilities have changed after training:

In [63]:
```
print(W)
```

```
tensor([[-3.2895e+00,  1.9656e+00,  7.4837e-01,  9.1425e-01,  1.0058e+00,
          9.0709e-01, -3.8237e-01,  8.2961e-02,  3.4820e-01, -3.9621e-02,
          1.3657e+00,  1.5675e+00,  9.3350e-01,  1.4125e+00,  6.1800e-01,
         -4.3777e-01, -1.7532e-01, -1.7782e+00,  9.7520e-01,  1.2013e+00,
          7.4990e-01, -1.9133e+00, -4.8333e-01, -6.7977e-01, -1.4532e+00,
         -1.3781e-01,  4.0890e-01],
        [ 2.2965e+00, -1.8025e-01, -2.0725e-01, -3.4589e-01,  4.4363e-01,
          3.6399e-02, -1.5385e+00, -1.3327e+00,  1.2488e+00,  9.0269e-01,
         -1.2949e+00, -1.5915e-01,  1.3296e+00,  8.9294e-01,  2.0966e+00,
         -2.1706e+00, -1.9615e+00, -2.2076e+00,  1.5854e+00,  5.1385e-01,
          2.9205e-02, -5.5165e-01,  2.2184e-01, -1.3719e+00, -1.2585e+00,
          1.1198e+00, -4.2190e-01],
        [ 1.3027e+00,  2.3723e+00,  2.0947e-01, -1.0908e+00,  7.2807e-01,
          3.1045e+00, -1.1616e+00, -1.1911e+00,  2.6474e-01,  1.9679e+00,
         -1.0697e+00, -1.2726e+00,  1.1984e+00, -1.0892e+00, -1.0602e+00,
          1.2181e+00, -1.1316e+00, -1.0999e+00,  3.3608e+00, -8.4553e-01,
         -1.1455e+00,  3.7126e-01, -1.1586e+00, -1.1152e+00, -1.2375e+00,
          9.7738e-01, -1.3179e+00],
        [ 6.1551e-01,  2.7858e+00, -1.6589e+00, -1.8775e-01, -1.4958e+00,
          2.3871e+00, -1.6076e+00, -1.5478e+00,  2.5773e+00,  1.6614e+00,
...truncated...
```

In [64]:
```
# compute the negative log likelihood loss for the first 5 bigrams
nlls = torch.zeros(5)
for i in range(5):
    # i-th bigram:
    x = xs[i].item() # input character index
    y = ys[i].item() # label character index
    print('--------')
    print(f'bigram example {i+1}: {itos[x]}{itos[y]} (indices {x},{y})')
    print('input to the neural net:', x)
    print('output probabilities from the neural net:', probs[i])
    print('label (actual next character):', y)
    p = probs[i, y]
    print('probability assigned by the net to the the correct character:', p.item()
    logp = torch.log(p)
    print('log likelihood:', logp.item())
    nll = -logp
    print('negative log likelihood:', nll.item())
    nlls[i] = nll

print('=========')
print('average negative log likelihood, i.e. loss =', nlls.mean().item())
```

```
--------
bigram example 1: .e (indices 0,5)
input to the neural net: 0
output probabilities from the neural net: tensor([0.0007, 0.1373, 0.0407, 0.0480, 0.
0526, 0.0476, 0.0131, 0.0209, 0.0272,
        0.0185, 0.0754, 0.0922, 0.0489, 0.0790, 0.0357, 0.0124, 0.0161, 0.0032,
        0.0510, 0.0639, 0.0407, 0.0028, 0.0119, 0.0097, 0.0045, 0.0168, 0.0290],
       grad_fn=<SelectBackward0>)
label (actual next character): 5
probability assigned by the net to the the correct character: 0.04764774069190025
log likelihood: -3.0439200401306152
negative log likelihood: 3.0439200401306152
--------
bigram example 2: em (indices 5,13)
input to the neural net: 5
output probabilities from the neural net: tensor([0.1943, 0.0330, 0.0062, 0.0077, 0.
0188, 0.0618, 0.0044, 0.0064, 0.0077,
        0.0398, 0.0032, 0.0089, 0.1583, 0.0374, 0.1303, 0.0132, 0.0045, 0.0014,
        0.0953, 0.0419, 0.0282, 0.0038, 0.0226, 0.0030, 0.0067, 0.0520, 0.0090],
       grad_fn=<SelectBackward0>)
label (actual next character): 13
probability assigned by the net to the the correct character: 0.037398409098386765
...truncated...
```

TODO: 13) Here is the final visualization after training. Can you interpret the weights, have they improved? Can you compare the trained weights to the weights from our first approach using simple counting instead of a neuron? **(2 points)**

No description has been provided for this image

ANSWER: YOUR ANSWER GOES HERE

The weights have improved, as indicated by the decrease in the average negative log likelihood from 3.7 to 2.5. This shows that the model has become better at predicting the correct characters, with higher probabilities assigned to them after training.

The weights from the first approach using simple counting can be thought of as counts of occurrences normalized by the total occurrences of the first character in each bigram. However, the weights of the trained neuron are an improvement in the following way:

The weights do not just reflect direct counts but learned patterns that capture dependencies between characters more effectively than raw counts. Means, the weight is higher if a dependency occurred often. Also the trained neuron reflects a smoothing effect: even if a certain bigram wasn't seen during the training, the model still assign a nonzero probability to it.

14. Sampling from the Neural Net

In [65]:
```python
# finally, sample from the 'neural net' model
g = torch.Generator().manual_seed(2147483647)
```

```
for i in range(20):

    out = []
    ix = 0
    while True:

        # ----------
        # BEFORE:
        #p = P[ix] # bigram probabilities by counting
        # ----------
        # NOW:
        xenc = F.one_hot(torch.tensor([ix]), num_classes=27).float()
        logits = xenc @ W # predict log-counts
        counts = logits.exp() # counts, comparable to N
        p = counts / counts.sum(1, keepdims=True) # probabilities for next characte
        # ----------

        ix = torch.multinomial(p, num_samples=1, replacement=True, generator=g).ite
        out.append(itos[ix])
        if ix == 0:
            break
    print(''.join(out))
```

```
junide.
janasah.
p.
cfay.
a.
nn.
kohin.
tolian.
juwe.
ksahnaauranilevias.
dedainrwieta.
ssonielylarte.
faveumerifontume.
phynslenaruani.
core.
yaenon.
ka.
jabdinerimikimaynin.
anaasn.
ssorionsush.
...truncated...
```

We get nearly identical samples as in the first model!

15. Conclusion

To summarize, we introduced the bigram character level language model, we saw how we can train the model, how we can sample from the model and how we can evaluate the quality of the model using the negative log likelihood loss. We trained the model in two completely different ways that actually get the same result and the same model: First, we just counted the frequency of all the bigrams and normalized. Second, we used the negative log likelihood loss as a guide to optimizing the counts matrix so that the loss was minimized in a gradient-based framework. Both approaches gave the same result, but the gradient-based framework is much more flexible and we can extend it to more complex settings.

In [66]:

```
import sys
import os

# Add the parent directory to the sys.path
sys.path.append(os.path.abspath(os.path.join('..')))

# truncate long cell output to avoid large pdf files
from helpers.truncate_output import truncate_long_notebook_output
truncated = truncate_long_notebook_output('1_Bigram_Language_Model__Gruppe2.ipynb')

# convert to pdf with nbconvert
if truncated:
    !jupyter nbconvert --to webpdf --allow-chromium-download TRUNCATED_1_Bigram_Lan
else:
    !jupyter nbconvert --to webpdf --allow-chromium-download 1_Bigram_Language_Mode
```

```
Output in 1_Bigram_Language_Model__Gruppe2.ipynb not above threshold and so no new v
ersion made.
```

```
[NbConvertApp] Converting notebook 1_Bigram_Language_Model__Gruppe2.ipynb to webpdf
[NbConvertApp] WARNING | Alternative text is missing on 7 image(s).
[NbConvertApp] Building PDF
[NbConvertApp] PDF successfully created
[NbConvertApp] Writing 237739 bytes to 1_Bigram_Language_Model__Gruppe2.pdf
```

In [ ]: