Generativni modeli in problem predstavitve

V tem poglavju nadaljujemo s sestavljanjem pristopov, ki smo jih spoznali do sedaj: gradimo modele, pri katerih z gradientnim sestopom iščemo parametre na podlagi kriterijske funkcije (verjetja) nad učnimi podatki, pri čemer kriterijsko funkcijo predstavimo z računskim grafom, njene odvode po posameznih parametrih pa izračunamo strojno. Tudi tu bomo na izhodu uporabljali linearne modele, ker pa bodo vhodi kompleksnejši, jih bomo nelinearno transformirali z nevronskimi mrežami. Ker vhodi tokrat ne bodo numerični, temveč tekstovni, bomo pri tem reševali še problem predstavitve: nevronske mreže zahtevajo numerične vhode, zato moramo kompleksnejše oblike podatkov, kot so besedila, slike, nizi, strukture in podobno, preslikati v numerični, navadno vektorski prostor. Takšne numerične predstavitve bi lahko zgradili ročno, vendar se v praksi izkaže, da je bolje, če je problem predstavitve (angl. *representation learning*) del samega učenja, kjer se predstavitev model nauči sam.

Predstavitveno učenje je ena osrednjih tem sodobnega strojnega učenja in generativne umetne inteligence. Uspeh globokih nevronskih mrež temelji prav na sposobnosti, da se uporabne predstavitve podatkov naučijo samodejno iz podatkov.

V poglavju bomo z omenjenimi pristopi zgradili znakovne jezikovne modele (*character-level language models*) za napovedovanje naslednjega znaka v zaporedju, torej generativne modele. Kot stranski produkt pa bomo dobili tudi predstavitve vhodov, ki sicer v našem preprostem modelu ne bodo posebej zanimive, vendar je sam postopek njihove izgradnje univerzalen in pomemben.

Vsebina poglavja se močno zgleduje po izjemnih predavanjih Andreja Karpathyja in njegove serije *Building makemore*. Močno priporočamo ogled vseh njegovih predavanj, ki so prosto dostopna na YouTube-u!

Cilj postopkov v tem poglavju bo razviti generativni model imen, ki se ga naučimo iz podatkov o imenih, registriranih v Sloveniji v preteklih letih. Začnimo torej s podatki.

Podatki

Učni podatki so tokrat imena novorojencev in prebivalcev Slovenije, zbrana iz različnih virov, ki vključujejo tudi Statistični urad Republike Slovenije. Imena so precej raznolika, njihov vzorec pa podaja spodnja tabela:

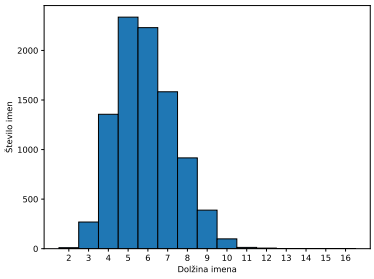
Podatke preberemo in za okus izpišemo nekaj statistik in podrobnosti.

Podatki so na voljo na file.biolab.si/datasets/imena.txt.

tisa	isabela	sašo	nikolas
edvin	andrej	muste	spominka
krenare	vidoš	isabella	jusuf
ralf	memet	danej	daira
rajfa	anatoliy	pečo	qamile
francelj	milada	bohdana	nagjije
ajete	majda	anemarija	belin
artur	januz	marioneta	margerita

```
>>> words = open("imena.txt", "r").read().splitlines()
>>> len(words)
9211
>>> min(words, key=len)
"an"
max(words, key=len)
"budislavabiljana"
```

V naši učni množici imamo torej skoraj deset tisoč imen. Naš cilj je iz te množice prepoznati vzorce, ki bi jih lahko uporabili pri generiranju imen.



Slika 61: Porazdelitev dolžine imen.

Bigrami

Najbolj preprosta ideja, s katero bomo začeli je, da probabilistično zgradimo model, ki na podlagi znaka v nizu generira naslednji znak. Model bo probabilističen zato, ker bomo znake generirali na podlagi napovedanih verjetnosti naslednjega znaka. Model bo preprost, saj bo zanemaril vsa ostala vedenja o nastajajočem nizu ter upošteval samo zadnje generiran znak. Pričakujemo, da bodo zato generiranja zaporedja ustrezno slaba, a nam bodo dobro služila za primerjavo z boljšimi modeli.

Naš enostavni model bo temeljil na bigramih, kjer bomo prešteli vse pojavitve sosednih znakov, torej vseh kombinacij črk, uporabljenih v učni množici. Pričnimo torej s prepoznavanjem abecede in gradnjo slovarjev, ki črko pretvori v njen indeks in indeks v črko; slovarja nam bosta prišla prav pri gradnji matrike s štetji pojavitev bigramov in pri generiranju imen:

```
import locale
words = open("imena.txt", "r").read().splitlines()
locale.setlocale(locale.LC_COLLATE, 'sl_SI.UTF-8')
alphabet = list(set("".join(words))) + ['.']
alphabet.sort(key=locale.strxfrm)
ctoi = {c: i for i, c in enumerate(alphabet)}
itoc = {i: c for c, i in ctoi.items()}
n_chars = len(alphabet)
```

Imena v slovenskih bazah podatkov imen uporabljajo poleg znakov iz slovenske abecede tudi druge znake, skupaj njih 31. Pozoren bralec bo opazil, da smo med znake vključili tudi poseben znak (“.”), s katerim bomo označevali začetek in konec besed.

```
>>> "".join(alphabet)
'.abcčćdđefghijklmnopqrsštuvwxyzž'
```

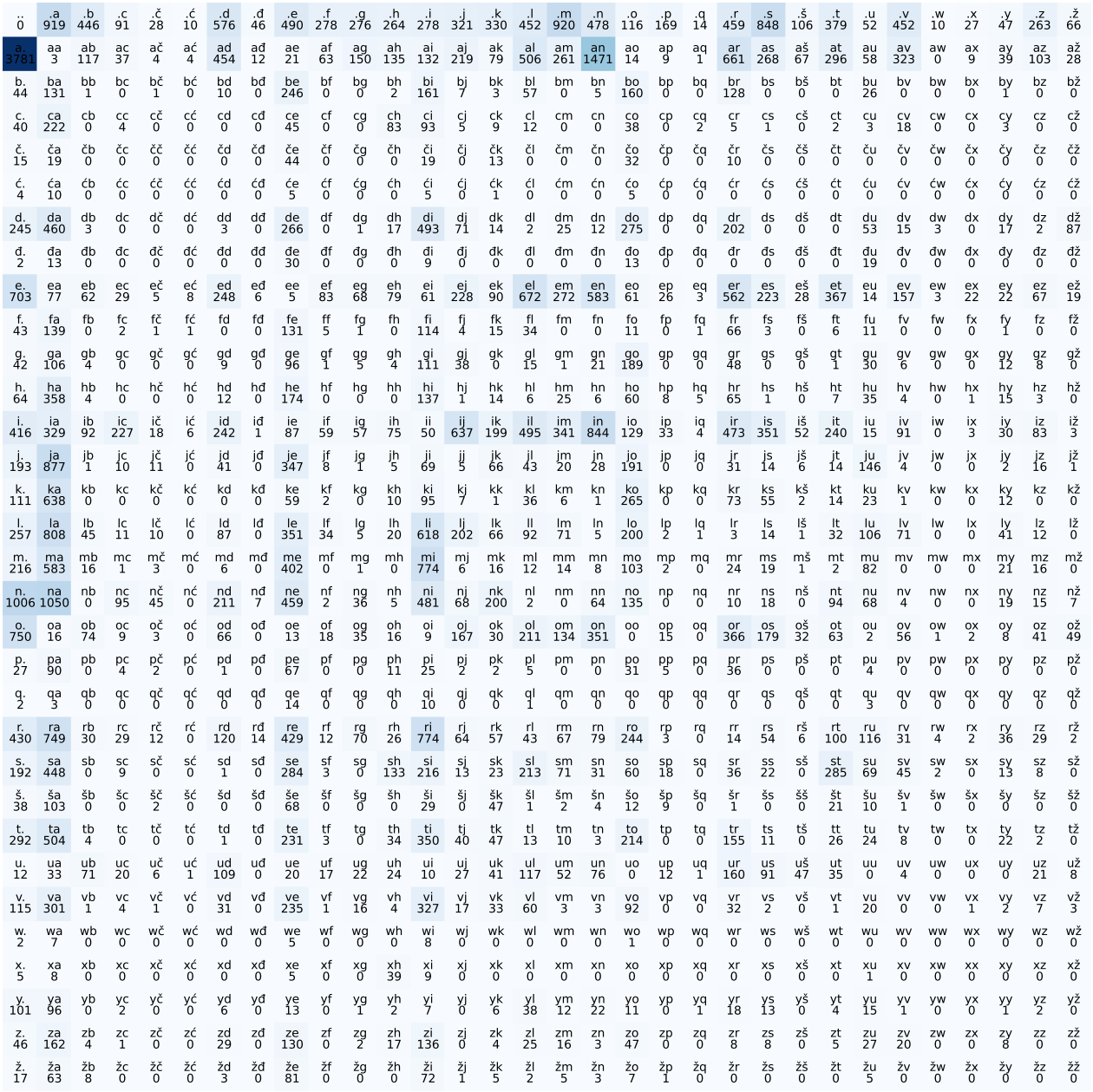
Vse imamo pripravljeno za gradnjo matrike, ki prešteje bigrame. Ker bomo za predstavitev računskih grafov uporabili knjižnico `PyTorch` je seveda najbolje elemente te knjižnice uporabiti tudi za predstavitev podatkov:

```
import torch
N = torch.zeros((n_chars, n_chars), dtype=torch.int32)
for w in words:
    s = ['.'] + list(w) + ['.']
    for c1, c2 in zip(s, s[1:]):
        ix1 = ctoi[c1]
        ix2 = ctoi[c2]
        N[ix1, ix2] += 1
```

Imenom vsakič dodamo označbo za začetek in konec besede, potem pa se sprehodimo skozi pare sosednjih znakov in na ustreznih mestih povečeamo števec pojavitev `N` za 1. Dobljeno matriko bi bilo treba izpisati, a da bo ponazoritev bolj pregledna, jo raje predstavimo s toplotno karto.

```
import matplotlib.pyplot as plt
plt.figure(figsize=(12, 12))
plt.imshow(N, cmap='Blues')
for i in range(n_chars):
    for j in range(n_chars):
        chstr = itoc[i] + itoc[j]
        plt.text(j, i, chstr, ha='center', va='bottom', color='black', fontsize=9)
        plt.text(j, i, N[i, j].item(), ha='center', va='top', color='black', fontsize=9)
plt.tight_layout()
plt.axis('off')
plt.savefig('bigram.pdf', bbox_inches='tight')
```

Rezultat prikazuje slika 62. V vsaki od vrstic je števec bigramov, ki se pričnejo z določeno črko. V tretji vrstici so na primer frekvence bigramov, ki se pričnejo s črko b. Tako je bigramov, kjer črki b sledi črka a 131, bigramov, kjer b-ju sledi e pa 246. V prvi vrstici so bigrami, ki sledijo posebni oznaki “.”, torej začetku besed, kjer vidimo, da se največ imen v naši bazi prične a in m. Prvi kolona v tabeli pa govori o koncu imen: največ teh se konča s črko a, sledita n in e.



Slika 62: Frekvenca bigramov v imenih v Sloveniji.

Že samo brskanje po imenih in njihovih res osnovnih vzorcih je zanimivo, a naša naloga tu je gradnja generativnega modela, zato nadaljujmo. Zgradili bi radi probabilistični model, kjer za vsako črko v nastajajočem imenu "napovemo", kakšna je verjetnost pojavitve naslednjega znaka. Model lahko potem uporabimo tako, da uteženo generiramo naslednji znak z ozirom na napovedane verjetnosti.

Za naš bigramski model potrebujemo torej tabelo verjetnosti, ki nam za dani znak izda verjetnosti naslednjega znaka. Te verjetnosti bomo shranili v matriko `P` tako, da najprej v njo shranimo vrednosti in matrike poajvitev bigramov, potem pa vsako od vrstic normaliziramo:

Pazljiv programer bi tu vsekakor preveril, ali so verjetnosti v vrsticah matrike `P` res seštejejo v 1.0.

```
P = N.float()
P = P / P.sum(dim=1, keepdim=True)
```

Generirajmo nekaj imen. Za ponovljivost bomo nastavili seme `torch`-evega generatorja naključnih števil, vsako besedo pričeli z oznako za začetek besede ("."), potem pa z `torch.multinomial` uteženo izbrali med kandidati za nadaljevanje besed, skladno s pripadajočimi verjetnostmi zadnjega znaka v besedu. Generiranje zaključimo, ko na ta način izberemo znak za konec besede (".")

```
g = torch.Generator().manual_seed(42)
for _i in range(8):
    out = []
    ix =  ctoi['.']  # names start with a start symbol
    while True:
        ix = torch.multinomial(P[ix], num_samples=1, replacement=True, generator=g).item()
        out.append(itoc[ix])
        if itoc[ix] == '.':
            break
    print("".join(out))
```

Zgornje nam izpiše:

```
a.
hman.
mife.
za.
han.
d.
nidrsanetaseglarardmesha.
gisitanč.
```

Hm. Ne izgleda ravno obetavno. A če bi naš model, torej matriko `P` zamenjali z matriko, kjer bi bile vse verjetnosti v posameznih vrsticah enake, Kodo za tako implementacijo matrike `P` prepuščamo

bralcu. bi dobili veliko slabši rezultat. Na primer:

```
txhmanpicfgxzha.
ćqžfgnršđsduyčlsegwnćxkdmmsžvmđisćtxbč.
pčlqžđđođhdugšnažćcuxesjjwđžtnećšžnžassvkbnčabrqueojćš.
bbčržzy.
paizsćlgte.
ghowyqrenxnidggfmđžwćpabrtgžmycoiktteoscxščđ.
bnhđrhyčgwrgujfwnzmeljqlvktfqčihć.
ycšgfvevjžhowcgpžžzmđkvćyvsxathčca.
```

Tu lahko torej zaključimo, da so rezultati generiranja imen z bigrami precej zanič, a vsekakor veliko boljši od popolnoma naključnih. Želimo si boljšega modela, a bi bilo dobro pred tem določiti, kako bomo kvaliteto modela sploh (kvantitativno) merili. Samo na občutek, ali so generirani nizi boljši ali slabši, se namreč ni za zanesti.

Verjetje

Naš bigramski model za vsak par znakov določa verjetnost naslednjega znaka. Za bigram "an" tako model poda verjetnost, da po znaku "a" sledi "n". Če bi bile vse črke enako verjetne, bi bila verjetnost posameznega naslednjega znaka enaka $1/n_{\text{chars}}$. Vsaka verjetnost, ki je večja od te vrednosti, nam torej pove, da se je model iz podatkov nekaj naučil.

Radi bi ocenili kvaliteto modela. Eden od načinov je, da pogledamo, kakšne verjetnosti model pripisuje bigramom v učni množici. Dober model bo bigramom, ki se v podatkih pogosto pojavljajo, pripisal visoke verjetnosti. Za celotno množico bigramov bi lahko te verjetnosti nato preprosto zmnožili (ob predpostavki neodvisnosti elementov učne množice). Tako dobimo *verjetje* (*likelihood*) podatkov glede na model:

$$L = \prod_i p_i$$

kjer je p_i verjetnost posameznega bigrama. Ker je produkt velikega števila majhnih verjetnosti numerično nepraktičen, namesto njega uporabljamo logaritem verjetja:

$$\log L = \sum_i \log p_i$$

Logaritem spremeni produkt v vsoto, hkrati pa ohrani vrstni red kakovosti modelov: večje verjetje pomeni boljši model. Ker pri optimizaciji običajno minimiziramo kriterijsko funkcijo, uvedemo še negativno log-verjetje:

$$\mathcal{L} = -\sum_i \log p_i$$

in ga pogosto normaliziramo s številom primerov:

$$\mathcal{L}_{\text{avg}} = -\frac{1}{n}\sum_i \log p_i$$

Dobimo torej kriterijsko funkcijo, kjer manjša vrednost pomeni boljši model. Verjetje, kot smo ga opisali tu, konceptualno torej ni prav nič različno od verjetij, ki smo jih že uporabili v prejšnjih poglavjih in ki tudi tokrat povejo, s kakšno verjetnostjo naš model opazi primere iz učne množice.

Spodnja koda implementira izračun povprečnega negativnega log-verjetja na vsej učni množici:

```
log_likelihood = 0.0
n = 0
for w in words[:3]:
    s = ['.'] + list(w) + ['.']
    for c1, c2 in zip(s, s[1:]):
        ix1 = ctoi[c1]
        ix2 = ctoi[c2]
        prob = P[ix1, ix2]
        logprob = torch.log(prob)
        log_likelihood += logprob
        n += 1
        print(f"{c1}{c2}: {prob:.3f} {logprob:.3f}")
nll = -log_likelihood / n
```

Kakšno je torej vrednost kriterijske funkcije našega binomske modela?

```
>>> nll.item()
2.4409780502319336
>>> -torch.log(torch.tensor(1/n_chars)).item()
3.465735912322998
```

Ne izgleda najbolje, ampak vsekakor bolše kot verjetje naključnega modela.

Kaj je potem naš cilj?

Zgradili bi radi model s čim manjšo vrednostjo kriterijske funkcije, oziroma čim večjim verjetjem. Zdaj že vemo, da tu ne smemo ravno napisati, da želimo maksimizirati verjetje na učni množici, saj bi tako dali prednost modelom, ki bi se lahko tej množici preveč prilagodili. Tako da bomo, v splošnem seveda, želeli graditi modele z velikim verjetjem na tesni množici. A da ne kompliciramo preveč: cilj zgornjega poglavja je bil določiti kriterijsko fukcijo in to nam je uspelo.

Verjetje posameznih besed

Naš model, v tej fazi zapisan v obliki matrike `P`, lahko uporabimo tako, da ocenimo, kakšno je verjetje neke besede oziroma imena. Namesto verjetja seveda tudi tu določimo negativni logaritem izgube. Pripravimo si ustrezno funkcijo, ki to izračuna in ki je seveda na moč podobna računanju verjetja, kot smo ga že uporabili zgoraj,

```
def get_nll(word):
    log_likelihood = 0
    n = 0
    s = ['.'] + list(word) + ['.']
    for c1, c2 in zip(s, s[1:]):
        ix1 = ctoi[c1]
        ix2 = ctoi[c2]
        prob = P[ix1, ix2]
        logprob = torch.log(prob)
        log_likelihood += logprob
        n += 1
        print(f"{c1}{c2}: {prob:.3f} {logprob:.3f}")
    return -log_likelihood / n
```

in izračunajmo izgubo za nekaj novih imen:

```
new_words = ["špela", "ana", "qewhu"]
for _w in new_words:
    print(f"{get_nll(_w):.3f} {_w}")
```

Dobimo:

```
2.317 špela
1.604 ana
inf qewhu
```

Že res, da "qewhu" pri nas ni ravno najbolj znano ime, ampak zakaj je njegova izguba neskončn. Podroben pregled, ki ga tudi tokrat prepuščamo braluc, pove, da je problem sosledje črk "wh", ki ga v naši učni množici ni in za katerega verjetnost je potem o. Logaritem te vrednosti vrne `inf` in potem pokvari celoten izračun.

Da bi se takim primerom, torej, izračunom, kjer za posamezen bigram ni bilo podatkov v učni množici, izognili, frekvence učne množice rahlo popravimo oziroma zgladimo njim pripadajoče verjetnosti:

```
P = (N+1).float()  # includes model smoothing
```

Glajenje tu je bilo minimalno: predpostavili smo, da je poleg vseh bigramov v učni množici za vsak možen par prisoten še dodaten

primer. Zanimivo, da je tovrstno glajenje enakovredno glajenju z regularizacijo L2, o čemer spregovorimo še nekoliko kasneje. Z zgornjim glajenjem nam uspe pridobiti tudi vrednost izgube za tretje, za naš prostor nenavadno ime:

Tu bi bilo recimo zanimivo preveriti, ali zna naš enostavni model ločiti med na primer tipičnimi slovenskimi imeni in na primer imeni nekih bolj severnih ali pa daljno vzhodnih nacij.

```
2.330 špela
1.608 ana
4.462 qewhu
```

Bigrami in nevronska mreža: arhitektura

Zgradimo enostavno nevronsko mrežo z enim samim izhodnim nivojem (in brez skrite plasti) tako, da bo vsaka enota na vhodu izračunala uteženo vsoto vhodov. Nelinearnih prenosnih funkcij tokrat ne bomo implementirali. Mreža bo zato zelo preprosta in tu z njo želimo samo postaviti osnovno za kasnejše, bolj kompleksne implementacije.

Začnimo s pripravo podatkov:

```
def words_to_data(words, verbose=False):
    # returns a training set
    xs, ys = [], []
    for w in words:
        s = ['.'] + list(w) + ['.']
        for c1, c2 in zip(s, s[1:]):
            if verbose:
                print(c1, c2)
            ix1 = ctoi[c1]
            ix2 = ctoi[c2]
            xs.append(ix1)
            ys.append(ix2)
    return torch.tensor(xs), torch.tensor(ys)

xs, ys = words_to_data(words, verbose=False)
```

Vektor (tenzor) `xs` torej vsebuje seznam znakov na vhodu modela, vektor (tudi tenzor) `ys` pa odgovarjajoče znake, ki bi jih pričakovali na izhodu modela. Skupaj ta dva vektorja tvorita našo učno množico. Ker smo vse primere imen v učni množici besed na ta način nekako zlili, sta vektorja primerno velika:

```
>>> xs.shape, ys.shape
63695, 63695
```

Oba zgrajena vektorja sta množica števil oziroma ustreznih indeksov črk. Na primer,

V uvodu tega poglavja smo govorili o problmu predstavitve podatkov. Tu smo rešitev tega problema močno poenostavili, in sicer vsako črko predstavimo z njenim indeksom v abecedi. Bolj primerne rešitve seveda sledijo.

```
>>> xs[:3], ys[:3]
tensor([ 0, 24, 12]), tensor([24, 12, 22])
```

Ta način predstavitve računsko ni najbolj primeren. Nevronske mreže namreč ne delujejo neposredno na indeksih, ampak na numeričnih vektorjih. Zato bomo vsak znak predstavili s t. im. *one-hot* kodiranjem, kjer posamezno črko predstavimo z vektorjem ničel in ene same enice na mestu, ki ustreza indeksu znaka v abecedi:

```
xenc = F.one_hot(xs, num_classes=n_chars).float()
yenc = F.one_hot(ys, num_classes=n_chars).float()
```

Vsak vhodni primer je sedaj predstavljen z vektorjem dolžine `n_chars`, kjer je aktivna le ena komponenta:

```
>>> xenc.shape, yenc.shape
torch.Size([63695, 32]), torch.Size([63695, 32])
```

Prvi učni primer tako na primer predstavlja začetek besede:

```
>>> xenc[0]
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.])
```

Nevronska mreža bo za vsak vhodni znak izračunala uteženo vsoto vhodov. Za začetni primer uporabimo le en sam nevron:

V PyTorchu operator `@` označuje matrično množenje: vhodni vektor množimo z matriko uteži, rezultat pa predstavlja aktivacijo nevronov. PyTorch množi tenzorje; pri 2D tenzorjih gre za običajno množenje matrik, pri višjih dimenzijah pa za posplošeno matrično množenje po zadnjih dveh dimenzijah.

```
>>> W1 = torch.randn(n_chars, 1)
>>> xenc @ W1
tensor([[-0.7360],
        [ 0.4115],
        [-0.2524],
        ...,
        [ 0.3244],
        [ 0.3244],
        [-1.0555]])
>>> (xenc @ W1).shape
torch.Size([63695, 1])
```

Izhod zgornjega bi bila torej utežena vsota vhodov, oziroma vektor uteženih vsot, za vsak učni primer torej eno realno število.

Ker želimo za vsak znak napovedati verjetnosti vseh naslednjih znakov, potrebujemo toliko izhodnih nevronov, kolikor je znakov v abecedi:

```
>>> W = torch.randn(n_chars, n_chars)
>>> (xenc @ W).shape
torch.Size([63695, 32])
```

Množenje matrik izvede vse utežene vsote hkrati za vse učne primere in vse izhodne nevrone. Rezultat so izhodi modela oziroma *logiti* (*logits*), ki jih bomo interpretirali kot logaritme frekvenc pojavitev in potem ustrezno preoblikovali:

Izračun `exp(logits)` in naknadna normalizacija predstavljata funkcijo *softmax*. Ta je standardna izbira za večrazredno klasifikacijo, saj poljubne realne vrednosti pretvori v verjetnosti posameznih razredov.

```
>>> logits = xenc @ W
>>> counts = logits.exp()
>>> probs = counts / counts.sum(dim=1, keepdims=True)
>>> probs.shape
torch.Size([63695, 32])
```

Ker torej eksponentna funkcija vrača le pozitivne vrednosti, lahko dobljene rezultate interpretiramo kot približke frekvenc. Za pretvorbo v verjetnosti moramo vsako vrstico normalizirati. S tem dobimo za vsak vhodni znak verjetnostno porazdelitev naslednjega znaka. Vsota verjetnosti v posamezni vrstici mora biti enaka 1.

To bi bilo nujno preveriti, a tu prepuščamo bralcu.

Kakovost modela tudi tokrat ocenimo z negativnim logaritmom verjetja. Tako kot prej, za vsak učni primer pogledamo verjetnost pravilnega naslednjega znaka, izračunamo njen logaritem in nato povprečje po vseh primerih. Lahko bi uporabili kodo iz zgornjih razdelkov, a nas je tu zamikalo vse skupaj spisati v eni vrstici:

```
>>> loss_m = -probs_m[torch.arange(len(ys)), ys].log().mean()
>>> loss_m
2.446988105773926
```

Nižja kot je vrednost kriterijske funkcije, boljše so napovedi modela. Ker so vse operacije v modelu odvedljive, bomo lahko v nadaljevanju z gradientnim sestopom poiskali takšne uteži matrike `W`, ki minimizirajo izgubo.

Bigrami in nevronska mreža: učenje

Naš enostavni model z nevronsko mrežo sedaj vsebuje parametre, uteži matrike `W`, ki jih moramo naučiti iz podatkov. Ker smo kriterijsko funkcijo zapisali z odvedljivimi operacijami, lahko uporabimo gradientni sestop in uteži prilagajamo tako, da zmanjšujemo izgubo modela. Uteži bodo na začetku učenja naključne, potem pa bomo v učni zanki zgradili graf funkcije izgube za dane vrednosti uteži `W` in učne podatke, izračunali gradiente, in osvežili uteži. V zanki.

```
g = torch.Generator().manual_seed(42)
W = torch.randn(n_chars, n_chars, generator=g, requires_grad=True)

for _k in range(100):
    # forward pass
```

```
    x_enc = F.one_hot(xs, num_classes=n_chars).float()
    logits = x_enc @ W  # napovej log counts
    counts = logits.exp()  # pretvori v counts
    probs = counts / counts.sum(1, keepdims=True)
    loss = -probs[torch.arange(ys.nelement()), ys].log().mean()
    if _k % 10 == 0:
        print(f"Loss {_k:02d}: {loss.item():.4f}")

    # backward
    W.grad = None  # parametri modela nastavljeni na 0
    loss.backward()

    # update
    W.data += -50 * W.grad
print(f"Loss {_k:02d}: {loss.item():.4f}")
```

Zaradi enostavnosti modela smo si lahko privoščili visoko stopnjo učenja. Učenje primerno hitro konvergira:

```
Loss 00: 4.0949
Loss 10: 2.7354
Loss 20: 2.5983
Loss 30: 2.5493
Loss 40: 2.5235
Loss 50: 2.5081
Loss 60: 2.4979
Loss 70: 2.4907
Loss 80: 2.4852
Loss 90: 2.4808
Loss 99: 2.4777
```

Naučeni model lahko sedaj uporabimo za generiranje novih imen:

```
g = torch.Generator().manual_seed(42)
for i in range(8):
    out = []
    ix = 0
    while True:
        xenc = F.one_hot(torch.tensor([ix]), num_classes=n_chars).float()
        logits = xenc @ W  # napovej log counts
        counts = logits.exp()
        p = counts / counts.sum(1, keepdims=True)
        ix = torch.multinomial(p, num_samples=1, replacement=True, generator=g).item()
        out.append(itoc[ix])
        if ix == 0:
            break
    print("".join(out))
```

Dobimo:

```
a.
hman.
mife.
za.
han.
d.
nidrsanetasegwmijadmesha.
gisitanč.
s.
la.
```

Rezultati so enaki kot pri prejšnjem bigramskem modelu, ki je temeljil neposredno na štetju pojavitev bigramov. To ni presenetljivo: nevronska mreža se je naučila iste statistične strukture podatkov, le da smo verjetnosti sedaj pridobili z učenjem parametrov matrike `W` in ne neposredno s štetjem frekvenc. Model še vedno temelji le na enem samem predhodnem znaku, zato ostaja zelo omejen in ne zna zajeti širšega konteksta v imenih.

Na tem mestu bi bila smiselna primerjava matrike `W` oziroma iz nje izvedenih verjetnosti in matrike verjetnosti, ki je sledila iz matrike frekvenc bigramov. Bi znali?

Globoka nevronska mreža: priprava podatkov

V prejšnjih poglavjih smo zgradili preprost bigramski jezikovni model, kjer smo naslednji znak napovedovali zgolj na podlagi enega samega predhodnega znaka. Tak pristop je intuitiven in enostaven za implementacijo, a omejen: če želimo pri napovedi znaka upoštevati več predhodnih znakov, to je, upoštevati širši kontekst, se velikost verjetnostne tabele (število vrstic) povečuje eksponentno. Za kontekst treh znakov bi tabela imela kar $32^4 = 1,048,576$ celic, to je, veliko več, kot imamo sploh učnih primerov in bi bila večina celic praznih. Model bi postal redek, popolnoma bi se prilagodil učnim podatkom in slabo bi napovedoval.

Zato uporabimo drugačen pristop: globoko nevronsko mrežo, ki lahko zajame več znakov konteksta brez eksponentne rasti števila parametrov. Ključna ideja je, da znakov ne obravnavamo več kot popolnoma nepovezane simbole, temveč vsakemu znaku priredimo vektor v sorazmerno majhnem *vložitvenem prostoru*.

Pri implementaciji se zgledujemo po modelu, ki so jo predlagali v Bengio et al. (2003) A Neural Probabilistic Language Model, *JMLR*.

Namesto ogromne tabele pogojnih verjetnosti torej uvedemo arhitekturo, ki na svojem vhodu uvede vložilno tabelo, ki vsak indeks znaka preslika v majhen vektor realnih števil. Dimenzija tega prostora je lahko zelo majhna, na primer celo dve, kar omogoča tudi vizualizacijo naučenih predstavitev znakov. Med učenjem se ti vektorji nato prilagodijo tako, da znaki s podobno vlogo v jeziku dobijo podobne predstavitve.

Pri znakih je takšna ideja nekoliko manj intuitivna, bistveno bolj naravna pa postane pri besedah. Izvorno delo Bengia in sod. (2003) je uporabljalo besede namesto znakov. Model se je tako lahko naučil,

da imajo besede kot sta `dog` in `cat` podobno semantično vlogo, saj se pogosto pojavljajo v podobnih kontekstih. Tudi če model v učnih podatkih nikoli ni videl stavka:

```
a dog was running in a _
```

lahko zaradi podobnosti med besedami uspešno napove smiselno nadaljevanje, če je prej videl podobne stavke z besedo `cat` ali `the dog`. Vložitveni prostor torej omogoča generalizacijo med podobnimi simboli.

Podatke za učenje pripravimo tako, da iz vsake besede ustvarimo več učnih primerov. Začetni kontekst zapolnimo s posebnim znakom `.`, ki predstavlja začetek oziroma konec besede.

```
block_size = 3 # context window width
XD, YD = [], []
for w in words[:2]:
    print(w)
    context = [ctoi['.']] * block_size  # padding
    for c in w + '.':
        ix = ctoi[c]
        XD.append(context)
        YD.append(ix)
        print("".join(itoc[i] for i in context), "->", itoc[ix])
        context = context[1:] + [ix]
```

Za besedi `tisa` in `isabela` dobimo naslednjih 13 učnih parov učne pare:

```
tisa
... -> t
..t -> i
.ti -> s
tis -> a
isa -> .
isabela
... -> i
..i -> s
.is -> a
isa -> b
sab -> e
abe -> l
bel -> a
ela -> .
```

Vsak primer vsebuje vhodni kontekst treh znakov in ciljni naslednji znak. Iz vseh primerov nato zgradimo tenzorja `X` in `Y`, ki predstavljata vhodne podatke in pripadajoče oznake:

```
>>> X = torch.tensor(XD)
>>> Y = torch.tensor(YD)
>>> X
tensor([[ 0,  0,  0],
        [ 0,  0, 24],
        [ 0, 24, 12],
        [24, 12, 22],
        ...
        [ 8, 15,  1]])
>>> Y
tensor([24, 12, 22,  1,  0, 12, 22,  1,  2,  8, 15,  1,  0])
```

Vsaka vrstica v `X` predstavlja kontekst treh znakov, medtem ko ustrezna vrednost v `Y` predstavlja znak, ki ga mora model napovedati. Tako pripravljen učni nabor nato uporabimo za učenje nevronske mreže.

Globoka nevronska mreža: sestava mreže

Arhitektura modela bo skladna s predlogom iz Bengio in sod. (2003) sestavljena iz treh plasti:

- **Vložitvena plast:** vsak znak preslika iz diskretnega indeksa znaka v vložitveni vektor.
- **Skrita plast:** združene vložitve konteksta (npr., vložitve za tri znake) pretvorimo v predstavitev skritega nivoja (npr. 100 enot) s polno povezano nevronsko plastjo in nelinearnostjo.
- **Izhodna plast:** model z zadnjo, s skrito plastjo polno povezano plastjo, vrne aktivacije, jih pretvori v logite za vse možne naslednje znake, ki jih potem pretvorimo v verjetnosti. Verjetnostna porazdelitev je po tem postopku izračunana kot softmax.

Nelinearnosti bomo tokrat uvedli s funkcijo `tanh`, ki je po obliki precej podobna sigmoidi, bralec pa lahko poskusi tudi s kakšnimi drugimi funkcijami.

Oglejmo si sedaj implementacijo globoke nevronske mreže in z njo pripadajočega računskega grafa. Začnemo z vložitveno plastjo, kjer vsak znak predstavimo z, v naši implementaciji majhnim, dvoštevilčnim vektorjem. Spodnjo implementacijo namenoma, zaradi poenostavitve, poganjamo nad učnimi podatki samo dveh prvih besed iz nabora imen, ki smo jih zgoraj z oknom širine 3 prevedli v 13 učnih primerov.

```
>>> g = torch.Generator().manual_seed(42)
>>> embedding_dim = 2
>>> C = torch.randn((n_chars, embedding_dim), generator=g)
>>> C.shape
torch.Size([32, 2])
>>> emb = C[X]
```

```
>>> emb.shape
torch.Size([13, 3, 2])
```

Tabela `C` je vložitvena matrika. Ker imamo 32 različnih znakov in dvodimenzionalne vložitve, je njena velikost:

$$C \in \mathbb{R}^{32\times 2}.$$

Vsaka vrstica matrike `C` predstavlja vložitveni vektor posameznega znaka.

Ko izvedemo indeksiranje `C[X]`, PyTorch za vsak indeks v `X` poišče ustrezno vrstico matrike `C`. Ker ima vhodni tenzor `X` obliko $(13, 3)$ dobimo izhod:

$$\texttt{emb.shape} = (13, 3, 2).$$

To pomeni: 13 učnih primerov, za vsak primer 3 znaki konteksta, vsak znak v kontekstu predstavljen z vložitvenim vektorjem dimenzije 2.

Vektorske vložitve so vhod v polno povezano skrito plast nevronske mreže:

```
>>> hidden_size = 100
>>> W1 = torch.randn((block_size * embedding_dim, hidden_size), generator=g)
>>> b1 = torch.randn((hidden_size), generator=g)
>>> W1.shape
torch.Size([6, 100])
>>> h = torch.tanh(emb.view(-1, block_size * embedding_dim) @ W1 + b1)
```

Ker imamo kontekst dolžine 3 in vsaka vložitev vsebuje 2 komponenti, moramo vse tri vložitve združiti v en sam vhodni vektor dolžine $3 \times 2 = 6$. Ukaz

```
emb.view(-1, block_size * embedding_dim)
```

Operacija `view` v PyTorchu ne kopira podatkov, ampak zgolj spremeni pogled na isto pomnilniško predstavitev tenzorja. Zato je zelo učinkovita.

zato tenzor oblike $(13, 3, 2)$ preoblikuje v matriko $(13, 6)$, ki je pripravljena za množenje z matriko uteži

Matrika uteži prve plasti ima obliko $W_1 \in \mathbb{R}^{6\times 100}$, saj vsak od šestih vhodov povežemo s 100 skritimi nevroni. Vektor začetnih vrednosti ima obliko $b_1 \in \mathbb{R}^{100}$. Po matričnem množenju:

$$(13, 6) \cdot (6, 100) \rightarrow (13, 100)$$

dobimo aktivacije skrite plasti za vseh 13 primerov hkrati. Vektor začetnih vrednosti se pri seštevanju samodejno razširi (angl. *broadcasting*) prek vseh vrstic.

Nelinearnost $h = \tanh(\cdot)$ vrednosti omeji na interval med -1 in 1. Sledi izhodna plast nevronske mreže:

```
>>> W2 = torch.randn((hidden_size, n_chars), generator=g)
>>> b2 = torch.randn((n_chars), generator=g)
>>> W2.shape
torch.Size([100, 32])
>>> logits = h @ W2 + b2
>>> logits.shape
torch.Size([13, 32])
>>> counts = logits.exp()
>>> probs = counts / counts.sum(1, keepdim=True)
>>> probs[torch.arange(len(Y)), Y]
tensor([3.4551e-08, 4.0015e-10, 7.3008e-03, 5.2019e-05, 4.2715e-12, 4.1585e-07,
        3.8536e-05, 1.5851e-05, 2.9422e-13, 6.2645e-14, 1.1990e-11, 1.5413e-03,
        2.9104e-11])
```

Matrika uteži druge, izhodne plasti ima obliko $W_2 \in \mathbb{R}^{100\times32}$, ker želimo iz 100 skritih aktivacij izračunati veretnosti za vseh 32 možnih naslednjih znakov. Množenje:

$$(13,100)\cdot(100,32)\rightarrow(13,32)$$

zato vrne matriko logitov oblike $(13,32)$ Vsaka vrstica te matrike vsebuje 32 vrednosti, po eno za vsak možen naslednji znak. Te vrednosti še niso verjetnosti, zato jih pretvorimo s softmax normalizacijo,

$$p_i = \frac{e^{z_i}}{\sum_j e^{z_j}}.$$

oziroma v kodi z

```
counts = logits.exp()
probs = counts / counts.sum(1, keepdim=True)
```

kjer najprej izračunamo eksponent logitov, nato pa vsako vrstico normiramo tako, da seštevek verjetnosti znaša 1.

Ukaz:

```
probs[torch.arange(len(Y)), Y]
```

iz vsake vrstice izbere verjetnost pravilnega naslednjega znaka. Ker mreža še ni naučena, so te verjetnosti trenutno zelo majhne. Idealno, seveda, bi te verjetnosti bile enake 1.0.

Na koncu računskega grafa določimo funkcijo izgube kot negativno logaritemsko verjetnost pravilnih napovedi. Glede na to, da so vsi parametri (uteži) naše nevrosnke mreže še naključni in nenaučeni, je izguba velika:

```
>>> loss = -probs[torch.arange(len(Y)), Y].log().mean()
>>> loss
tensor(17.7561)
```

Fukcija izgube je negativno log-verjetnost pravih razredov:

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N} \log p(y_i \mid x_i),$$

Pogosto to funkcijo imenujemo tudi križna entropija (angl. *cross-entropy loss*). Cilj učenja bi torej bil minimizirati to vrednost glede na parametre:

$$C,\ W_1,\ b_1,\ W_2,\ b_2.$$

Za izračun izgube iz logitov bo zato dovolj klic:

V praksi ročno računanje softmax funkcije in negativne log-verjetnosti skoraj vedno nadomestimo s funkcijo `F.cross_entropy`, ki je numerično stabilnejša in je njena implementacija v knjižnici PyTorch učinkovito implementirana.

```
>>> loss = F.cross_entropy(logits, Y)
tensor(17.6200)
```

Ker sta ta funkcija in njen odvod neposredno implementirana v PyTorchu, bo njena neposredna uporaba tudi zmanjšala velikost računskega grafa in s tem skrajšala čas računanja. Implementacija `cross_entropy` pazi tudi na velike vrednosti logitov in jih pred računanjem zmanjša (odšteje maksimalno vrednost of vseh logitov) ter tako skrbi tudi za računsko stabilnost.

Globoka nevronska mreža: učenje

Sestavimo zdaj skupaj vse zgornje komponente našega programa in uveddimo učenje. Začnimo z določitvijo parametrov nevronske mreže:

```
embedding_dim = 2  # velikost vložitvenega prostora
hidden_size = 100  # size of the hidden layer

g = torch.Generator().manual_seed(42)
C = torch.randn((n_chars, embedding_dim), generator=g)
W1 = torch.randn((block_size * embedding_dim, hidden_size), generator=g)
b1 = torch.randn((hidden_size), generator=g)
W2 = torch.randn((hidden_size, n_chars), generator=g)
b2 = torch.randn((n_chars), generator=g)
parameters = [C, W1, b1, W2, b2]
```

Parametrov je, v primerjavi z modelom, kjer bi preštevali pojavitev za vse možne kombinacije (32^3 krat 32 za prav toliko možnih znakov na izhodu mreže, skupaj torej $32^4 = 1.048.576$), relativno malo:

```
>>> n_param = sum(p.numel() for p in parameters)
>>> n_param
3996
```

Knjižnici PyTorch moramo povedati, da želimo za naše parametre računati gradiente. To storimo z:

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

Sledi učenje. Nič, česar še nismo že velikokrat v tej skripta, a ponovimo vseeno:

```
for i in range(100):
    # forward pass
    emb = C[X]
    h = torch.tanh(emb.view(emb.shape[0], block_size * embedding_dim) @ W1 + b1)
    logits = h @ W2 + b2
    loss = F.cross_entropy(logits, Y)
    if i % 10 == 0:
        print(f"{i:03d} Loss: {loss.item():6.3f}")

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

    # parameter update
    for p in parameters:
        p.data += -0.1 * p.grad
print(f"{i:03d} Loss: {loss.item():6.3f}")
```

Tudi na celotni množici učnih podatkov je konvergenca sorazmerno hitra, čeprav v vsakem koraku obravnavamo celotno učno množico z 63.695 primeri. Zgornja koda namreč izpiše:

Pozor: v zgornjih poglavjih smo obravnavali samo nekaj začetnih imen, tu zdaj želimo obravnavati celotno množico besed

```
000 Loss: 17.454
010 Loss: 11.683
020 Loss:  9.010
030 Loss:  7.371
040 Loss:  6.349
050 Loss:  5.655
...
990 Loss:  2.470
999 Loss:  2.470
```

Paketno učenje

Pri učenju nevronskih mrež običajno ne izvajamo optimizacije nad celotnim učnim naborom hkrati, saj je to počasno in pomnilniško zahtevno. Namesto tega uporabljamo paketno učenje (*mini-batch*

training), kjer, kot smo to spoznali že v prejšnjih poglavjih, v vsakem koraku naključno izberemo manjši podnabor učnih primerov.

```
batch_size = 32

for i in range(10000):
    # paket za učenje
    indices = torch.randint(0, len(Y), (batch_size,))
    # forward pass
    emb = C[X[indices]]
    h = torch.tanh(emb.view(-1, block_size * embedding_dim) @ W1 + b1)
    logits = h @ W2 + b2
    loss = F.cross_entropy(logits, Y[indices])
    if i % 10 == 0:
        print(f"{i:03d} Loss: {loss.item():6.3f}")

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

    # update parameters
    for p in parameters:
        p.data += -0.01 * p.grad
print(f"{i:03d} Loss: {loss.item():6.3f}")
```

Še izpis kriterijske funkcije po učenju:

```
emb_all = C[X]
h_all = torch.tanh(emb_all.view(-1, block_size * embedding_dim) @ W1 + b1)
logits_all = h_all @ W2 + b2
loss_all = F.cross_entropy(logits_all, Y)
loss_all.item()
```

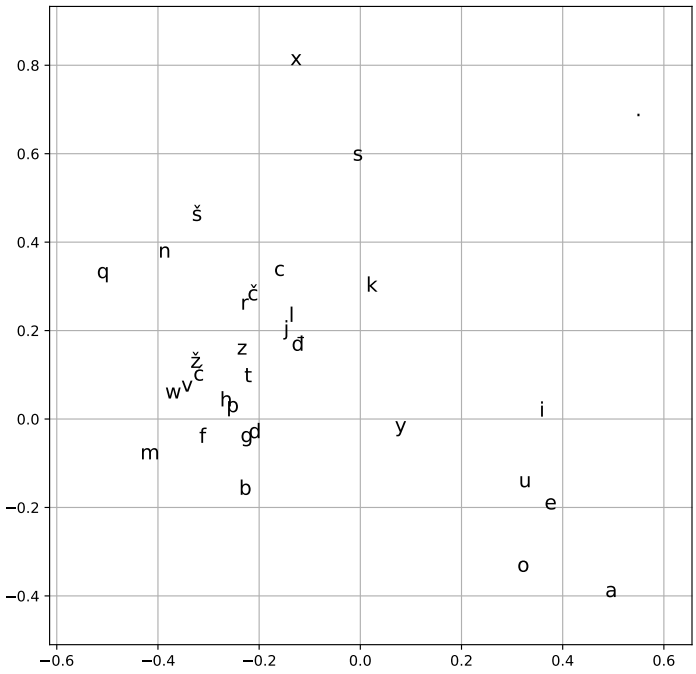
Ta ima vrednost 2.4367. Naslednji trik, ki ga lahko uporabimo je, da zmanjšamo stopnjo učenja. Uporabimo, na primer,

```
    _lambda = 0.1 if i < 10000 else 0.01
    for p in parameters:
        p.data += - _lambda * p.grad
```

S tem smo funkcijo izgube na učnih podatkih še zmanjšali na 2.3552, kar je bolje od bigramov, kjer je bila izguba 2.4470.

Vložitev črk

Ena ključnih idej modela je uporaba vložitvene matrike C, ki vsak znak preslika iz diskretnega indeksa v vektor realnih števil. Namesto



Slika 63: Vložitev črk v vektorski prostor, kjer smo se vložitve naučili z nevronsko mrežo.

da bi znake obravnavali zgolj kot ločene simbole, jih tako predstavimo kot točke v zveznem prostoru. V našem primeru ima matrika:

$$C \in \mathbb{R}^{32\times 2},$$

kar pomeni, da vsakemu od 32 znakov priredimo dvodimenzionalni vložitveni vektor. Ker je dimenzija vložitve enaka 2, lahko naučene predstavitve tudi vizualiziramo. Priročno!

Slika 63 prikazuje naučene vložitve črk po učenju modela. Opazimo lahko, da se črke s podobno vlogo v jeziku pogosto razporedijo blizu skupaj. S slke razberemo gručo samoglasnikov, črki x in y sta ločeni od drugih, ločitveni znak . pa je poponoma na svojem. Nekaj podobnega smo morda pričakovali, a je vizualizacija lep prikaz zmožnosti nevronske mreže, da se nauči predstavitev sicer neštevislkih objektov.

Generiranje imen

Zgradili smo model, izboljšali točnost na učni množici, a skoraj pozabili preveriti, kako sploh ta deluje. Spodaj je koda.

```
def generate_name(g):
```

```
    context = [ctoi["."]] * block_size
    output = []

    while True:
        emb = C[torch.tensor([context])]
        h = torch.tanh(emb.view(1, block_size * embedding_dim) @ W1 + b1)
        logits = h @ W2 + b2
        probs = F.softmax(logits, dim=1)
        ix = torch.multinomial(probs, num_samples=1, generator=g).item()

        if ix == ctoi["."]:
            break

        output.append(itoc[ix])
        context = context[1:] + [ix]

    return "".join(output)

gen = torch.Generator().manual_seed(0)
generated_names = [generate_name(g) for _ in range(8)]
print("\n".join(generated_names))
```

Izgleda dejansko malo bolje, a prostora za izboljšanje je očitno še precej:

```
baran
melica
cre
gemiatjilk
imojica
heda
ktjabja
nru
```

Kam od tu naprej?

Zgornji modeli predstavljajo šele začetek sodobnih generativnih modelov. Iz preprostega bigramskega modela smo postopoma prišli do globoke nevronske mreže z vložitvami znakov in širšim kontekstom. A ostaja veliko možnosti za izboljšave. Povečali bi lahko dimenzijo vložitvenega prostora, uporabili daljši kontekst, dodali več skritih plasti ali večje število nevronov ter izboljšali optimizacijo z ustreznejšo izbiro hitrosti učenja, velikosti paketov in regularizacijo. Že takšne spremembe lahko bistveno izboljšajo kvaliteto generiranih imen in zmanjšajo vrednost kriterijske funkcije.

Opisani modeli so tudi pomemben zgodovinski korak proti sodobnim arhitekturam globokega učenja za delo z zaporedji. Kasnejši

modeli so namreč namesto preprostih večplastnih mrež uvedli konvolucijske pristope (npr. WaveNet), rekurentne mreže in danes prevladujoče transformerje, ki temeljijo na mehanizmu pozornosti. Ti modeli omogočajo uporabo bistveno daljšega konteksta, učenje na ogromnih količinah podatkov in gradnjo predstavitev, ki zajamejo kompleksne semantične in sintaktične odnose med elementi jezika. Prav ideje, predstavljene v tem poglavju — verjetje, vložitve, gradientno učenje in generiranje zaporedij — pa predstavljajo osnovo sodobnih velikih jezikovnih modelov.