Bigrams

We take in example data then do next character prediction based on the probability distribution given by characters. E.g we may observe that character e follows the start token more often, then in our generated word we pick a random character but e will also occur proportionally more often similar to the distribution in the text.

E.g.:

start/end token

- emma
- paul
- jack
- ⋮

probability lookup table

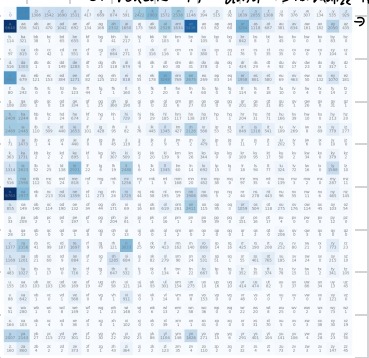
	.a	.b	...
a.	aa	ab	
b.	ba	bb	

rows sum up to 1

⇒ Pr[x,y] = Pr[y next character | x was last character]

normalized counter values

Counter of tokens in data → normalize it ⇒ get probability distribution



Full bigram

```
[26] #sample multiple
     g = torch.Generator().manual_seed(2147483647)

     for i in range(5):
       ix = 0
       out = []
       while True:
         p = P[ix] #probability distribution from data
         ix = torch.multinomial(p, num_samples=1, replacement=True, generator=g)
         if ix == 0:
           break #end token
         out.append(itos[ix])

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

```
cexze
momasurailezitynn
konimittain
llayn
ka
```

PyTorch I

Torches: torch.tensor([1,2],[3,4]) ← explicit

torch.zeros((2,3)) ← filled with zeros

torch.rand((3,3)) ← random values

Elementwise operations: a * b, a / b, ... ← not matrix mult. but elementwise

Broadcast: ops such as above work if dimensions are either same or 1 (then value is just duplicated)

Sum function:

Example $x = \text{torch.tensor}\begin{pmatrix}[1,2,3],\\ [4,5,6]\end{pmatrix}$

s1 = x.sum(dim=1, keepdim=1) → [6,15] (keepdim)

s2 = x.sum(dim=1, keepdim=True) [[6],[15]] ← usually want this for broadcasting

Loss function in the bigram

For each two-letter sequence we already know the probability. We then define $\text{logProb} = \sum_{\text{seq} \in \text{word}} \log(p\_\text{seq})$ ⇒ negative number for $p\_\text{seq} \in [0,1]$ (with smoothing $\in]0,1]$). Since we typically minimize loss function, we set the loss_total = -logProb. Afterwards we normalize the loss = $\frac{\text{loss\_total}}{\text{count}}$ to get the average loss. This will then be the thing we try to minimize.

⇒ Smoothing: some entries never appeared in the dataset. Hence using a single one of these sequences would set the loss to $\log(0) = -\infty$ which we do not want. To avoid this we add a small count to each entry to avoid this problem.

loss function

```
log_likelihood = 0.0
n = 0
for w in ["andrejq"]:
  chs = ['.'] + list(w) + ['.'] #gives us start and end tokens
  for ch1, ch2 in zip(chs, chs[1:]):
    ix1 = stoi[ch1]
    ix2 = stoi[ch2]
    prob = P[ix1, ix2]
    logprob = torch.log(prob)
    log_likelihood += logprob
    n+=1
    print(f'{ch1}{ch2}: {prob:.4f} {logprob: .4f}') #print probability to have this sequence

print(f'{log_likelihood=}')
nll = -log_likelihood
print(f'{nll=}')
print(nll/n) #normalized
```

```
.a: 0.1376 -1.9835
an: 0.1604 -1.8302
nd: 0.0384 -3.2594
dr: 0.0770 -2.5646
re: 0.1334 -2.0143
ej: 0.0027 -5.9004
jq: 0.0003 -7.9817
q.: 0.0558 -2.8863
log_likelihood=tensor(-28.4204)
nll=tensor(28.4204)
tensor(3.5525)
```

Smoothing:

```
6] P = (N + 1).float() #smoothing
   P = P / P.sum(1, keepdim= True)
```

Neural net approach

Create a network that predicts next character based on previous character. For tuple (xs, ys) we want to predict ys given xs. E.g. for .emma. = [0, 5, 13, 13, 1] (xs) → predict [5, 13, 13, 1, 0] (ys)

Problem: We cant just pass indicies into the neural network, instead we have 27 input nodes (where the input is 1, all other nodes 0) and 27 nodes where probabilities between 0,1 are output for next character. ⇒ <u>one-hot encoding</u>

How do we calculate the outputs of a layer?

Inputs as (inputs, 27) matrix, weights as (27, neurons) matrix (if output layer neurons=27).

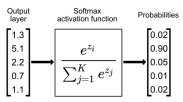
We then do matrix multiplication inputs @ weights = layer output (inputs, neurons)

Convert output layer → probabilities for each character output

We exponentiate each output (to have >0 for each) then normalize, this gives us propensities / probs.

```
W = torch.randn((27, 27)) #random weights for 27 inputs and 27 neurons
logits = xenc @ W #matrix multiplication in Pytorch
counts = logits.exp() #equivalent to the N matrix, > 0
probs = counts / counts.sum(dim=1, keepdim=True)
probs
```

⇒ called <u>softmax</u>



We then have loss = $-\log(\text{probs}[\text{range}, ys]).\text{mean}() + 0.01 \cdot (W^2).\text{mean}()$

probs of getting ys in iteration num essentially → array

smaller weight loss loss ⇒ models tries to optimize for closer weights, has same effect as smoothing.

Finally we backpropagate on loss and do gradient descent on the weights.

Gradient descent in code!

```
#gradient descent

for k in range(10):
  #forward pass
  logits = xenc @ W #matrix multiplication in Pytorch
  #softmax function
  counts = logits.exp() #equivalent to the N matrix, > 0
  probs = counts / counts.sum(dim=1, keepdim=True)
  loss = -probs[torch.arange(num), ys].log().mean() + 0.01 * (W**2).mean()#goes to row [0, 1, 2, 3, 4, ...] (the correct input), then outputs the
  #0.01 * (W**2).mean() is regularization, it allows us to make W a bit more uniform -> same goal as smoothing
  #backward pass on weights
  W.grad = None #zero the gradients for a fresh new backprop
  loss.backward()
  W.data += -50 * W.grad
  print(loss.item())
```

How do we sample from this?

Basically the same way we did from the simple "lookup table". Also with the same seed, this model will generate the same outputs as the one above since it basically just learned the counter