PSL UNIVERSITÉ PARIS

Lab Session 11 — Solutions

NLP — The Bigram Character Language Model

TAOCP2 — Week 11 — **Solutions**

1 Warm-up by Hand

1.1 What is a bigram?

(a) `ana` becomes `.ana.`, giving the four bigrams

$$(\texttt{.},\texttt{a}),\ (\texttt{a},\texttt{n}),\ (\texttt{n},\texttt{a}),\ (\texttt{a},\texttt{.}).$$

(b) A single boundary token `.` is enough because "previous character is `.`" already *means* "we are at the start of a word", and "next character is `.`" means "the word ended". With two distinct tokens `<S>` and `<E>`, the count matrix would have $28 \times 28 = 784$ entries, but the row `<E>` and the column `<S>` would be *identically zero* (no character ever follows `<E>`, no character ever precedes `<S>`). They are useless rows/columns; merging the two tokens collapses them.
(c) A trigram tensor has $27^3 = 19\,683$ entries; a k-gram tensor has 27^k. With 32 000 short names ($\sim 200\,000$ tokens), k-grams beyond 4 would be much sparser than they are populated: most rows would just be zero. This is the *curse of dimensionality* for count-based models, and the central reason we will switch to a parametric model in §6.

1.2 Probabilities and negative log-likelihood

(a) The four log-probabilities are $\log 0.5,\ \log 0.25,\ \log 0.25,\ \log 0.5$, so

$$\log \mathcal{L} = 2\log 0.5 + 2\log 0.25 = -2\ln 2 - 4\ln 2 = -6\ln 2 \approx -4.159.$$

The average NLL is $-\log \mathcal{L}/N = 6\ln 2/4 \approx 1.040$.
(b) Two reasons:

- **Numerical.** A product of 200 000 probabilities in $[0, 1]$ underflows to 0 in any reasonable floating-point format. A sum of logs stays in a comfortable range ($\sim -5 \times 10^5$).
- **Interpretability.** The *average* NLL is independent of dataset size, so we can compare a model on 200 000 bigrams with the same model on 20 000 000 bigrams. The raw likelihood would shrink by a factor $10^{(20M-200k)\log_{10}(\bar{p})}$ — meaningless.

(c) For the uniform model, every probability is $1/27$, so

$$\text{avg NLL}_{\text{uniform}} = -\log(1/27) = \log 27 \approx 3.296.$$

Any model worth its salt should beat this baseline. The count model of §5 will land at ~ 2.45, the trained network at the same value.

Common Mistakes

- Confusing $\log_{10}$ and ln. The convention in ML is the natural logarithm; whichever you use, be consistent inside one comparison.
- Forgetting to divide by N. The total NLL grows linearly with the number of bigrams; only the *average* is meaningful across runs.

2 Loading the Data

(a) The dataset has 32 033 names; the shortest is 2 letters long (e.g. `li, an, lo, ty, jo, ka, da, ra`), the longest is 15.
(b) Small datasets train in seconds and fit in memory — ideal for a class. But a real-world language model targets the *distribution of natural language*, which has billions of distinct tokens; 32 000 names is not even a rounding error of GPT's training corpus. The bigram model we build will be barely better than a hand-crafted baseline; the same algorithm trained on a trillion tokens (with a transformer instead of a linear layer) is what powers ChatGPT.

3 Counting Bigrams

3.1 First, with a Python dictionary

```
b = {}
for w in words:
    chs = ['.'] + list(w) + ['.']
    for ch1, ch2 in zip(chs, chs[1:]):
        b[(ch1, ch2)] = b.get((ch1, ch2), 0) + 1
```

(a) The most common bigram is `('n', '.')` with 6 763 occurrences — a great many names end in `n` (`Aiden, Logan, Mason, Sebastian, …`). The next most common bigrams are `('a', '.')` (most common feminine ending) and `('.', 'a')` (start of `Anna, Andrew, …`).
(b) The dataset uses 627 distinct bigrams out of the 729 possible ones. So 102 bigrams *never* appear, including obvious ones like `('q', 'p')` or `('z', 'q')`, but also single-character "words" which would imply a bigram like `('.', '.')` (the dataset's shortest names have 2 characters, so this never happens).

3.2 From dict to a 2-D tensor

```
for w in words:
    chs = ['.'] + list(w) + ['.']
    for ch1, ch2 in zip(chs, chs[1:]):
        ix1, ix2 = stoi[ch1], stoi[ch2]
        N[ix1, ix2] += 1
```

(a) `N.shape == (27, 27)`, `N.dtype == torch.int32`, `N.sum() == 228 146`. The total is the number of bigrams in the dataset, which equals $\sum_w (|w|+1)$ because a word of length ℓ produces $\ell+1$ bigrams once we add the two boundary `.`s.
(b) Three example checks: `N[stoi['e'], stoi['m']] = 769`, `N[stoi['.'], stoi['e']] = 1531`, `N[stoi['m'], stoi['m']] = 168` — all match the dict entries.

3.3 Visualising the matrix

(a) The brightest cell of *row* `.` is `('.', 'a')` (4 410): names starting with `a` are over-represented (a small sample: `Anna, Anthony, Andrew, Aria, Asher, …`). The brightest cell of *column* `.` is `('n', '.')` (6 763): names ending in `n`.
(b) Many cells are exactly zero, e.g. `N[stoi['j'], stoi['q']]`, `N[stoi['q'], stoi['z']]`, `N[stoi['x'], stoi['j']]`. Whenever a test (or evaluation) bigram falls into one of these zero cells, the model assigns probability 0 and the log-likelihood is $-\infty$. This is exactly what add-one smoothing in §4.2 will fix.

4 From Counts to Probabilities

4.1 One row at a time

(a) With seed `2147483647`, the first sample from row 0 is `'j'`. With a different seed (or no seed at all), you get a different letter — the most likely outcome is `'a'` (probability ≈ 0.137).
(b) The empirical histogram of 1000 draws from a uniform on $\{0, 1, 2\}$ should be roughly $(333, 333, 334)$ with $\mathcal{O}(\sqrt{1000})$ fluctuations. `torch.bincount` returns these counts directly, and dividing by 1000 gives empirical frequencies very close to $1/3$.

4.2 Vectorising: build the full matrix `P`

(a) `P.sum(1).max()` and `P.sum(1).min()` both equal 1.0 to machine precision (the row sums are constructed to be exactly 1, modulo floating-point rounding of order 10^{-7}).
(b) Without `keepdim=True`, `P.sum(1)` has shape `(27,)`. Broadcasting interprets this as a row vector `(1, 27)` and the division `P /= P.sum(1)` normalises the matrix *column-wise* (every *column* sums to 1, not every row). Then `P[0].sum()` is no longer 1; it equals the sum of the first row divided element-wise by the column sums — a meaningless quantity, around 1.5 in our case.

Common Mistakes

This silent broadcast bug is the single most common error of the lab. It does not raise an exception, the shapes match, and `P` still looks "probability-like". Insist that students always test `P.sum(1)` *after* the normalisation, on at least one row.

5 Sampling Whole Names

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

for _ in range(10):
    out, ix = [], 0
    while True:
        p  = P[ix]
        ix = torch.multinomial(p, num_samples=1, replacement=True, generator=g).item()
        if ix == 0:
            break
        out.append(itos[ix])
    print(''.join(out))
```

(a) With seed `2147483647`, the first ten samples are roughly

```
junide
janasah
p
cony
a
nn
kohin
tolian
juee
ksahnaauranilevias
```

None of these is a real name — but most are *pronounceable*: the bigram structure of English names (alternating consonants and vowels, `n` as a frequent ending, etc.) is clearly visible. They are *much* better than random.

(b) With `p_uniform = torch.ones(27)/27.0` the samples collapse to random strings full of `q`, `x`, `z` clusters, many of which are unpronounceable. The boundary token `.` is now no more likely than any other letter, so the average length explodes (the expected length of a sample is $1/(1/27) = 27$ characters).

6 Evaluating the Model: NLL

6.1 Compute the NLL on the training set

```
log_likelihood += torch.log(prob)
n += 1
```

(a) The numbers (with +1 smoothing):

```
log_likelihood = -559951.5625
nll            =  559951.5625
avg NLL        =  2.4544
```

This is far better than the uniform baseline of $\log 27 \approx 3.296$ from §1.2(c) — a $\sim 25\%$ improvement, which is huge for such a trivial model.

(b) Without smoothing (`P = N.float() / N.sum(1, keepdim=True)`), the very first never-seen bigram in the training loop — often something like `('j', 'q')` encountered when evaluating a held-out word — gives `torch.log(0)` $= -\infty$, and the running sum becomes `-inf` forever. (On the training set itself, every bigram has a count $\geqslant 1$, so the unsmoothed loss is finite there: ~ 2.45 as well; the problem appears as soon as you evaluate on *new* words.)

(c) Typical results on this dataset:

k	avg NLL
0.001	2.4540
1	2.4544
10	2.4884
100	2.7088

k is the strength of the smoothing prior: small k trusts the empirical counts; large k pulls P towards the uniform distribution and the loss approaches $\log 27 \approx 3.296$. This is exactly the role λ will play in the regularised neural-network version (§7.2).

7 The Same Model as a Neural Network

7.1 Build the dataset of `(input, target)` pairs

```
xs.append(stoi[ch1])
ys.append(stoi[ch2])
```

(a) `num == 228 146`, exactly $\sum N_{ij}$ from §3 — as it must, since the two loops enumerate the same (c_{t-1}, c_t) pairs.

7.2 One-hot encoding

(a) The cast to `.float()` is needed because matrix multiplication with the (float) weight matrix W requires floating-point operands; `F.one_hot` returns an `int64` tensor, which would trigger a type error on `xenc @ W`.

(b) With `W = torch.eye(27)`, `xenc @ W == xenc`: the identity selects every row unchanged. With a

general W, `(one_hot(i) @ W) == W[i]`, i.e. row i of W — so the "one-hot → linear layer" pipeline is just an indexed lookup, exactly as in the count model.

7.3 Forward pass: logits, softmax, probabilities

(a) `probs.shape == (5, 27)`. Each row sums to 1 by the definition of softmax: $\sum_k e^{z_k} / \sum_j e^{z_j} = 1$.

(b) The roles map as follows:

Count model (§3)	Neural network (§6)
$N[i,:]$ (count row)	`logits[i, :].exp()` = "log-counts" exp
$\sum_j N[i,j]$ (row sum)	`counts[i].sum()`
$P[i,j] = N[i,j] / \sum_j N[i,j]$	`probs[i, j]` = softmax row

The two models parameterise the *same family of row-stochastic* 27×27 *matrices*: the count model fixes one such matrix from data; the neural network parameterises it as softmax(W) row-wise and finds it by gradient descent. (Strictly: every row-stochastic matrix can be written as a softmax, modulo a per-row additive constant on W, which the softmax is invariant to.)

7.4 Read off the loss for a few examples

(a) On a freshly initialised $W \sim \mathcal{N}(0,1)$, the softmax produces roughly uniform probabilities $\sim 1/27 \approx 0.037$, so the per-example NLL is $\sim \log 27 \approx 3.30$ — much worse than the count model's 2.45 *because we have not trained anything yet*. The point of §7 is precisely to fix this.
(b) `probs[torch.arange(num), ys]` returns a length-`num` tensor: entry i is `probs[i, ys[i]]`. The same numbers as the explicit loop, in one vectorised line. This trick is essential to make the training loop fast.

Explanation

"Fancy indexing" `A[i, j]` where `i, j` are integer tensors of the same shape returns a tensor of that shape, with element `A[i[k], j[k]]` at position `k`. PyTorch (and NumPy) broadcast this, so `probs[arange(N), ys]` picks one entry per row, in the column specified by `ys[k]`. This is exactly the pattern that `F.cross_entropy` optimises internally.

8 Training: Gradient Descent on the NLL

8.1 One full step

(a) The initial loss is $\approx$ `3.7686`, slightly worse than $\log 27 \approx 3.296$ (because the random W does not produce a *perfectly* uniform softmax, but a slightly-spikier-than-uniform distribution that happens to put low probability on many of the actual targets).
(b) `W.grad = None` releases the underlying tensor (and lets PyTorch allocate a fresh one on the next backward), whereas `W.grad = 0` or `W.grad.zero_()` keeps the same buffer and zeroes it in place. Both are correct; `None` is the conventional fast path used throughout the PyTorch ecosystem (`optimizer.zero_grad(set_to_none=True)`).
(c) After the single update with `lr = 10`, re-running *just* the forward block gives `loss` $\approx$ `3.74`. A small drop — one step is not magic; the next 200 steps in §7.2 will do the real work.

8.2 The full training loop

The three missing lines:

```
W.grad = None          # 2) zero grads
loss.backward()        # 3) backward
W.data += -50.0 * W.grad   # 4) update
```

(a) The loss drops monotonically from ~ 3.77 at step 0 to ~ 2.48 at step 200, very close to the count-model value of 2.45 — the small gap is due to the L_2 regularisation we added. With $\lambda = 0$ the network reaches ~ 2.4544, identical to the count model down to the printed precision.
(b) The loss curve drops sharply for the first ~ 30 steps, then flattens out. This is the typical "J-shape" of cross-entropy training on a convex problem: a single linear layer with softmax is a multinomial logistic regression, which has a unique global minimum.
(c) The gradient of the cross-entropy w.r.t. a single logit is $O(1)$ (it equals $p_k - \mathbb{1}[k = y]$). The mean over 228 146 examples is therefore also $O(1)$, but each entry of $\nabla_W L$ averages contributions from all bigrams that share the same input character — typically thousands. The resulting gradient is small in magnitude (each entry $\sim 10^{-3}$ to 10^{-2}), so a learning rate of 50 produces updates of order 10^{-1} on the weights, which is the right ballpark. By contrast, in Lab 10 we summed (rather than averaged) over only 4 samples, and the gradient was already $O(1)$, so $\eta = 0.05$ was the right scale.

Explanation

The takeaway is not "always use $\eta = 50$", but: the right learning rate depends on the magnitude of ∇L, which itself depends on batch size, loss reduction (`mean` vs `sum`), and parameter scale. Tuning η is one of the central practical skills of deep learning. We will revisit it next week with adaptive optimisers (Adam).

8.3 Tweaking the experiment

(a) Typical final loss as a function of λ:

λ	avg NLL after 200 steps
0	2.4544 (matches count model exactly)
0.001	2.4548
0.01	2.4827 (the value we used by default)
0.1	2.7530
1.0	3.2890 (essentially uniform: $W \approx 0$)

For $\lambda \to \infty$ the regulariser dominates and forces $W \to 0$, so the softmax collapses to uniform and the loss reaches $\log 27$. For $\lambda = 0$ the network reproduces the count model's empirical distribution exactly; in the language of §5, this is the "zero smoothing" regime, dangerous on held-out data.
(b) With `lr = 5.0`, the loss is still around 3.0 after 200 steps; you would need ~ 2000 steps to reach 2.48. Linear relationship: divide η by 10, multiply the number of steps by 10 to get the same final loss.
(c) With `lr = 500.0`, the loss curve oscillates wildly and within a few steps overshoots: the loss either explodes or stalls at a value worse than the starting point. Symptom on the curve: vertical jumps that grow in amplitude. Diagnosing this from the curve alone is a basic deep-learning skill.

Common Mistakes

- Forgetting `W.grad = None` (or putting it *after* `loss.backward()`). Gradients accumulate, the effective η grows linearly with k, and the loss diverges within ~ 10 steps.
- Writing `W = W - 50 * W.grad` (without `.data`). This creates a *new* tensor that is a leaf of a different computation graph; the next backward fails with "element 0 of tensors does not require grad".
- Modifying `W` *inside* the autograd graph (e.g. `W -= 50 * W.grad` without a `torch.no_grad()` context). Same symptom.

9 Sampling from the Trained Network

```
g = torch.Generator().manual_seed(0)

for _ in range(10):
    out, ix = [], 0
    while True:
        xenc   = F.one_hot(torch.tensor([ix]), num_classes=27).float()
        logits = xenc @ W
        counts = logits.exp()
        p      = counts / counts.sum(1, keepdim=True)

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

(a) The samples (with seed 0, after 200 training steps and $\lambda = 0.01$) look very similar in quality to those from §4: short, mostly-pronounceable strings, no real names. They *should* be similar: the trained network has rediscovered (almost) the same probability matrix as the count model.
(b) Concrete check: the count-model row `P[stoi['a']]` and the network row `F.softmax(W[stoi['a']], dim=-1)` agree to $\sim 10^{-3}$ component-wise (any residual gap is due to the L_2 regulariser). Without regularisation ($\lambda = 0$), they agree to machine precision.

Explanation

This experimental confirmation is the punchline of the lab: "count and normalise" is the closed-form solution to the optimisation problem "minimise the average NLL over a 27×27 row-stochastic matrix". Gradient descent on the equivalent neural-network parametrisation *rediscovers* that solution.
What changes in the next labs is *not* the loss, the optimiser, or the sampling code — it is the family of distributions P_θ. We will make θ a much richer object (embedding tables, hidden layers, attention…) that can capture longer-range dependencies than two characters.

Bonus — The Same Loss in One Line of PyTorch

(a) `F.cross_entropy(logits, ys)` returns the same value as `-F.log_softmax(logits, dim=-1)[arange(num), ys].mean()`. Replacing the four lines does not change the loss curve; it just runs faster and is numerically robust.
(b) If a logit is 40, `torch.exp(40)` $\approx$ `2.35e17`; several such logits sum to $\sim 10^{18}$, well within `float32`

range but a few more bits and we hit `inf`. `F.cross_entropy` uses the *log-sum-exp trick*:

$$\log \sum_j e^{z_j} = z^* + \log \sum_j e^{z_j - z^*}, \quad z^* = \max_j z_j,$$

so the largest exponent is 0 and the sum stays in $[1, K]$.
(c) A two-layer MLP `nn.Linear(27, 64) -> tanh -> nn.Linear(64, 27)` has $27 \cdot 64 + 64 + 64 \cdot 27 + 27 = 3\,547$ parameters — far more than the 729 of the linear model. Yet on *bigrams* the loss does not go meaningfully below ~ 2.45: the model can still only see the previous *single* character, so the bigram bound is the statistical floor. To beat it we need to feed in more context (trigrams, k-grams, embeddings of the previous k characters…) — which is exactly the plan of next week's lab.