Lab Session 11

NLP — The Bigram Character Language Model

The Art of Computer Programming 2 — Week 11 — May 22, 2026

Objectives

By the end of this lab, you will have built, step by step, a complete *character-level language model* on a real dataset of 32 000 first names — twice: once by counting bigrams, once by training a single-layer neural network with PyTorch's autograd. More precisely, you should be able to:

- Tokenise a text dataset and build the character vocabulary.
- Count bigrams and store them in a 2-D `torch.Tensor`.
- Convert counts to a row-stochastic probability matrix and sample from it with `torch.multinomial`.
- Compute the average *negative log-likelihood* of a model and explain why smoothing matters.
- Re-express the same model as a one-layer neural network using one-hot encoding + softmax.
- Train it by gradient descent and verify it reaches the same loss as the explicit count model.

Why this matters

This lab is the gateway to modern NLP. Every component you build today — tokens, logits, softmax, NLL, autograd training — reappears *verbatim* in GPT-style transformers. The only thing that changes in modern NLP is the forward pass: a transformer is the same training loop with a much smarter way of computing the next-token logits.

Hint

Setup. Open a fresh notebook `lab11.ipynb` in the same folder as the file `names.txt` (provided). The only imports you will need:

```
import torch
import torch.nn.functional as F
import matplotlib.pyplot as plt
%matplotlib inline
```

1 Warm-up by Hand (15 min)

Pen and paper — no computer needed yet.

1.1 What is a bigram?

Recall

A **bigram** is an ordered pair of consecutive characters (c_{t-1}, c_t) from a string. We add a single boundary token `.` at both ends of every word; it marks both *start* and *end*. For instance, `emma` becomes

$$\texttt{.emma.} \quad \Rightarrow \quad (\texttt{.},\texttt{e}),\ (\texttt{e},\texttt{m}),\ (\texttt{m},\texttt{m}),\ (\texttt{m},\texttt{a}),\ (\texttt{a},\texttt{.}).$$

(a) List all bigrams of the word `ana`.

(b) Why do we need a *single* boundary token `.` rather than two distinct `<S>` and `<E>`? (Hint: count rows and columns of the count matrix in each variant.)

(c) With 27 symbols (26 letters + `.`) the bigram count matrix has shape 27×27. How many parameters would a *trigram* count tensor have? A k-gram tensor? Conclude on the scaling problem.

1.2 Probabilities and negative log-likelihood

X[1] Math reminder

For a model with parameters θ and a dataset of bigrams $\mathcal{D} = \{(c_{t-1}^{(i)}, c_t^{(i)})\}_{i=1}^{N}$, the *likelihood* is

$$\mathcal{L}(\theta) = \prod_{i=1}^{N} P_\theta(c_t^{(i)} \mid c_{t-1}^{(i)}), \qquad \log \mathcal{L}(\theta) = \sum_{i=1}^{N} \log P_\theta(c_t^{(i)} \mid c_{t-1}^{(i)}).$$

The *average negative log-likelihood* is

$$\text{loss}(\theta) \;=\; -\frac{1}{N} \sum_{i=1}^{N} \log P_\theta(c_t^{(i)} \mid c_{t-1}^{(i)}).$$

Suppose a tiny dataset contains exactly the four bigrams

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

and a model that assigns probabilities $P(\texttt{a} \mid \texttt{.}) = 0.5$, $P(\texttt{b} \mid \texttt{a}) = 0.25$, $P(\texttt{.} \mid \texttt{b}) = 0.5$.

(a) Compute $\log \mathcal{L}$ and the average NLL.

(b) Why is the NLL a *better* loss than the raw likelihood for a real dataset of $\sim 200\,000$ bigrams? (Two reasons: numerical, and interpretability.)

(c) What is the value of the NLL of a *uniform* model $P(c_t \mid c_{t-1}) = 1/27$ on *any* dataset? Use this number as your "random baseline" to compare against later.

2 Loading the Data (10 min)

`names.txt` is a list of ∼32 000 lower-case first names, one per line.

```
words = open('names.txt', 'r').read().splitlines()
print(words[:10])
print('total words :', len(words))
print('shortest len:', min(len(w) for w in words))
print('longest  len:', max(len(w) for w in words))
```

(a) Run the cell. How many names are there? Length of the shortest / longest name?

(b) **Reflection.** The dataset is small (< 1 MB). Why is this a good thing for a teaching example, and a *bad* thing for a practical language model?

3 Counting Bigrams (15 min)

3.1 First, with a Python dictionary

Recall

A Python dict `b` indexed by (c_1, c_2) tuples is the most natural data structure for sparse counts. We will replace it with a tensor in a moment, but the dict version is the easiest to reason about.

```
b = {}
for w in words:
    chs = ['.'] + list(w) + ['.']
    for ch1, ch2 in zip(chs, chs[1:]):
        # YOUR CODE HERE
        # increment b[(ch1, ch2)] by 1 (use b.get(..., 0) for the default)
        pass

# the 10 most common bigrams:
sorted(b.items(), key=lambda kv: kv[1], reverse=True)[:10]
```

(a) Fill in the body of the inner loop and run the cell. Which bigram is the most common? Does its identity surprise you?

(b) How many *distinct* bigrams appear in the data? Out of the $27 \times 27 = 729$ possible ones, how many are missing?

3.2 From dict to a 2-D tensor

Recall

Vocabulary. Build two dictionaries `stoi: char → int` and `itos: int → char`, where index 0 is reserved for the boundary token `.`. The bigram count matrix N then has shape 27×27: row i counts how many times character i was followed by every other character.

```
chars = sorted(list(set(''.join(words))))     # 26 letters
stoi  = {s: i + 1 for i, s in enumerate(chars)}
stoi['.'] = 0
itos  = {i: s for s, i in stoi.items()}

N = torch.zeros((27, 27), dtype=torch.int32)

for w in words:
    chs = ['.'] + list(w) + ['.']
    for ch1, ch2 in zip(chs, chs[1:]):
        # YOUR CODE HERE
        # convert ch1, ch2 into indices ix1, ix2 and increment N[ix1, ix2]
        pass

print(N.shape, N.dtype, N.sum().item())
```

(a) Fill in the body of the loop and run it. What does `N.sum()` represent in plain English?

(b) **Sanity check.** Pick three bigrams and verify that `N[stoi[c1], stoi[c2]]` matches the count from the dict version.

3.3 Visualising the matrix (provided)

```
plt.figure(figsize=(16, 16))
plt.imshow(N, cmap='Blues')
for i in range(27):
    for j in range(27):
        chstr = itos[i] + itos[j]
        plt.text(j, i, chstr, ha='center', va='bottom', color='gray')
        plt.text(j, i, N[i, j].item(), ha='center', va='top', color='gray')
plt.axis('off');
```

(a) Read off the matrix. Which letter is most likely to *start* a name (look at row .)? Which letter most often *ends* a name (look at column .)?

(b) Locate at least three cells whose count is *exactly zero*. Why is this going to be a problem for the loss we will define in §5?

4 From Counts to Probabilities % (15 min)

4.1 One row at a time

To sample the *first* character, look at row 0 of N (counts of bigrams starting with .) and normalise it to a probability vector.

```
p = N[0].float()
p = p / p.sum()

g  = torch.Generator().manual_seed(2147483647)
ix = torch.multinomial(p, num_samples=1, replacement=True, generator=g).item()
print('first char =', itos[ix])
```

(a) Run the cell. Which letter did you draw? Re-run with a different seed — do you always get the same letter?

(b) **Sanity check that `multinomial` does what you think.** Build p = uniform on $\{0, 1, 2\}$ and draw 1000 samples. Plot the empirical histogram (`torch.bincount`); it should be roughly flat.

4.2 Vectorising: build the full matrix `P`

Recall

Doing `N[i].float() / N[i].sum()` once per row is wasteful. Build the entire row-stochastic matrix `P` in one shot using broadcasting: `P = N.float(); P /= P.sum(1, keepdim=True)`.

```
P = (N + 1).float()                 # +1 = "add-one smoothing", more on this in §5
P /= P.sum(1, keepdim=True)         # row-wise normalisation

print('shape of sums :', P.sum(1, keepdim=True).shape)
print('row 0 sums to :', P[0].sum().item())
```

Pitfall

Pitfall: `keepdim`. If you forget `keepdim=True`, the sum has shape `(27,)` instead of `(27, 1)`. Broadcasting then treats it as `(1, 27)` and silently normalises *columns* instead of rows. The numbers look fine; the model is completely wrong.

(a) Run the cell. Verify that *every* row of `P` sums to 1 (e.g. `P.sum(1).max(), P.sum(1).min()`).

(b) Re-do the normalisation *without* `keepdim` and inspect `P[0].sum()`. What do you get and why?

5 Sampling Whole Names (10 min)

Start from `ix = 0` (the `.` token) and keep sampling until we return to it.

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

for _ in range(10):
    out, ix = [], 0
    while True:
        # YOUR CODE HERE
        # 1) pick the row of probabilities corresponding to the current ix
        # 2) sample the next index from that row with torch.multinomial
        # 3) if it is 0 (boundary), stop; else append itos[ix] to out
        pass
    print(''.join(out))
```

(a) Implement the loop and run it. Read your samples out loud — are they recognisable as names? Are they at least *pronounceable*?

(b) Replace `P[ix]` with a uniform distribution `torch.ones(27)/27`. Re-run. The samples should look obviously worse — what changes qualitatively?

6 Evaluating the Model: NLL (15 min)

Why this matters

"Looks like a name" is not a measurement. We need a single number that summarises how well a model fits the data, so we can *compare* models. The standard choice is the average negative log-likelihood (NLL): low is good, 0 would mean the model places probability 1 on every observed bigram.

6.1 Compute the NLL on the training set

```
log_likelihood = 0.0
n = 0
for w in words:
    chs = ['.'] + list(w) + ['.']
    for ch1, ch2 in zip(chs, chs[1:]):
        ix1, ix2 = stoi[ch1], stoi[ch2]
        prob    = P[ix1, ix2]
        # YOUR CODE HERE
        # 1) accumulate log(prob) into log_likelihood
        # 2) increment n
        pass

nll = -log_likelihood
print(f'log_likelihood = {log_likelihood:.4f}')
print(f'nll            = {nll:.4f}')
print(f'avg NLL        = {nll / n:.4f}')
```

(a) Fill in the body and run it. What value do you get for the average NLL? How does it compare to the uniform baseline you computed in §1.2(c)?

(b) **Smoothing.** Re-build `P` *without* the `+1` (i.e. `P = N.float()`) and re-run the NLL loop. What happens, and why? (Hint: think about what $\log(0)$ is.)

(c) Try `P = (N + k).float()` with $k \in \{0.001, 1, 10, 100\}$. Plot or print the average NLL for each value. What does k control, intuitively?

Pitfall

Smoothing matters. A single never-seen bigram is enough to make the unsmoothed log-likelihood $-\infty$. Add-k smoothing (`P = (N + k).float()`) is the simplest fix. Larger k pulls P towards the uniform distribution.

7 The Same Model as a Neural Network (25 min)

Why this matters

The counting model is exact and fast for bigrams. But a k-gram count tensor needs 27^k rows — already 19 683 for trigrams, $\sim 5 \cdot 10^5$ for 4-grams, and intractable beyond. We now express *the same family of distributions* as a single linear layer followed by a softmax. Same number of parameters, same expressive power — but trainable by gradient descent and ready to grow into something deeper in the next lab.

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

Recall

A *supervised* dataset of bigrams: for every consecutive pair (c_{t-1}, c_t) in every word, c_{t-1} is the input and c_t is the target the network must predict.

```
xs, ys = [], []
for w in words:
    chs = ['.'] + list(w) + ['.']
    for ch1, ch2 in zip(chs, chs[1:]):
        # YOUR CODE HERE
        # append stoi[ch1] to xs and stoi[ch2] to ys
        pass

xs  = torch.tensor(xs)
ys  = torch.tensor(ys)
num = xs.nelement()
print('number of examples:', num)
```

(a) Fill in the body and run it. How many *(input, target)* pairs do you get? Compare with $\sum N_{ij}$ from §3.

7.2 One-hot encoding

Recall

A *one-hot* vector of length 27 has a single 1 at position i and zeros elsewhere. Multiplying it by a matrix $W \in \mathbb{R}^{27\times 27}$ selects *row i of W*. So one-hot + linear layer is just a trainable lookup table.

```
xenc = F.one_hot(xs[:5], num_classes=27).float()
print(xenc)
print('shape =', xenc.shape)

plt.imshow(xenc, cmap='Greys')
plt.title('one-hot encoded inputs (first 5 examples)');
```

(a) Run the cell. Why do we cast to `.float()`? What happens if you leave it as `int64`?

(b) **Reflection.** If you computed `xenc @ W` with `W = torch.eye(27)`, what would you get back? Convince yourself that one-hot + matrix multiply = row selection.

7.3 Forward pass: logits, softmax, probabilities

X¹ Math reminder

The **softmax** of a vector $z \in \mathbb{R}^K$ is

$$\text{softmax}(z)_k = \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}}.$$

It maps any real vector to a probability distribution: positive entries that sum to 1. We interpret the linear layer's outputs $z = \text{xenc} \cdot W$ as *log-counts* (a.k.a. *logits*); exponentiating gives positive "counts" and normalising gives probabilities.

```
g = torch.Generator().manual_seed(2147483647)
W = torch.randn((27, 27), generator=g)

xenc   = F.one_hot(xs[:5], num_classes=27).float()
logits = xenc @ W
counts = logits.exp()
probs  = counts / counts.sum(1, keepdim=True)

print('probs shape  :', probs.shape)
print('row 0 sums to:', probs[0].sum().item())
```

(a) Run the cell. What is the shape of `probs`? Why does each row sum to 1?

(b) **Connection to §3.** Squint at the pipeline:

$$\text{one-hot} \xrightarrow{W} \text{logits} \xrightarrow{\exp} \text{counts} \xrightarrow{\text{normalise}} \text{probabilities}.$$

What plays the role of the matrix N from §3? What plays the role of its row-sum? In one sentence, why are these two models the same family?

7.4 Read off the loss for a few examples

```
nlls = torch.zeros(5)
for i in range(5):
    x, y = xs[i].item(), ys[i].item()
    p    = probs[i, y].item()
    nll  = -torch.log(probs[i, y]).item()
    nlls[i] = nll
    print(f'bigram {i+1}: {itos[x]}{itos[y]}  p={p:.4f}  nll={nll:.4f}')

print('average NLL =', nlls.mean().item())
```

(a) Run the cell. The average NLL on these 5 random examples is around 3–4. Why is it so much worse than the count model?

(b) **Vectorise.** Replace the Python loop by a single line that picks all the right probabilities at once: `probs[torch.arange(num), ys]`. Verify it gives the same numbers.

8 Training: Gradient Descent on the NLL ⚙ (25 min)

8.1 One full step

Recall

The four mandatory steps per iteration (same as Lab 10!):

1. **Forward:** build `logits, counts, probs, loss`.
2. **Zero grads:** `W.grad = None`.
3. **Backward:** `loss.backward()`.
4. **Update:** `W.data += -lr * W.grad`.

```python
g = torch.Generator().manual_seed(2147483647)
W = torch.randn((27, 27), generator=g, requires_grad=True)

# 1) forward
xenc   = F.one_hot(xs, num_classes=27).float()
logits = xenc @ W
counts = logits.exp()
probs  = counts / counts.sum(1, keepdim=True)
loss   = -probs[torch.arange(num), ys].log().mean()
print('initial loss =', loss.item())

# 2) zero grads + 3) backward
W.grad = None
loss.backward()
print('W.grad shape =', W.grad.shape)

# 4) update
W.data += -10.0 * W.grad
```

(a) Run the cell. What is the initial loss? Compare with the average NLL of the uniform model from §1.2(c).

(b) Why do we set `W.grad = None` rather than `W.grad = 0`? (Both work; one is more efficient. PyTorch convention.)

(c) Re-run *just* the forward block (without re-initialising W). The loss should have decreased. By how much?

8.2 The full training loop, with L_2 regularisation

X^1 Math reminder

Adding $\lambda \cdot \|W\|_2^2$ to the loss penalises large weights. When $W \to 0$, all logits are equal and the softmax becomes uniform — so L_2 regularisation is the gradient-based equivalent of add-one smoothing in §5. Larger $\lambda \Leftrightarrow$ stronger smoothing.

```python
# fresh initialisation
g = torch.Generator().manual_seed(2147483647)
W = torch.randn((27, 27), generator=g, requires_grad=True)
```

```
losses = []
for k in range(200):

    # 1) FORWARD
    xenc   = F.one_hot(xs, num_classes=27).float()
    logits = xenc @ W
    counts = logits.exp()
    probs  = counts / counts.sum(1, keepdim=True)
    loss   = -probs[torch.arange(num), ys].log().mean() + 0.01 * (W**2).mean()

    # 2) ZERO GRADS
    # YOUR CODE HERE  (one line)

    # 3) BACKWARD
    # YOUR CODE HERE  (one line)

    # 4) UPDATE  (use lr = 50.0 -- yes, that's huge, more on this below)
    # YOUR CODE HERE  (one line)

    losses.append(loss.item())
    if k % 20 == 0 or k == 199:
        print(f'step {k:3d}  loss = {loss.item():.4f}')
```

(a) Fill in the three missing lines and run the loop. Does the loss converge? Around what value does it settle? Compare with the count model's ~ 2.45 from §5.

(b) Plot the loss curve:

```
plt.plot(losses); plt.xlabel('step'); plt.ylabel('loss'); plt.grid(True);
```

Is the descent monotone? Where does it flatten?

(c) **Reflection.** The learning rate `lr = 50.0` is wildly larger than what worked in Lab 10 (`lr = 0.05`). Why is it reasonable here? (Hint: how big are the gradients of a softmax with one-hot inputs and $\sim 200\,000$ examples?)

Pitfall

Same two bugs as Lab 10:

- Forgetting to zero `W.grad` (here: not setting it to `None`) ⇒ gradients accumulate, the effective learning rate explodes, the loss diverges.
- Picking the wrong sign in the update rule (`W.data += +lr * W.grad`) ⇒ gradient *ascent*, the loss *grows*.

8.3 Tweaking the experiment

(a) Vary the regularisation strength: replace `0.01` by $\lambda \in \{0, 0.001, 0.01, 0.1, 1.0\}$. Plot the final loss as a function of λ. Where does the model collapse to the uniform distribution? Where does it overfit (in the sense: matches the raw counts of §3 too tightly)?

(b) Reduce the learning rate to `lr = 5.0`. How many steps do you need to reach the same loss?

(c) Increase to `lr = 500.0`. What happens? Identify the symptom in the loss curve.

9 Sampling from the Trained Network (10 min)

Why this matters

Same generative loop as in §4, except that the row of probabilities now comes from a forward pass through the network instead of a table lookup `P[ix]`.

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

for _ in range(10):
    out, ix = [], 0
    while True:
        # YOUR CODE HERE
        # 1) one-hot encode [ix] -> xenc, shape (1, 27)
        # 2) compute logits = xenc @ W,  counts = logits.exp()
        # 3) p = counts / counts.sum(1, keepdim=True)
        # 4) sample next ix from p with torch.multinomial
        # 5) if ix == 0: break, else append itos[ix]
        pass
    print(''.join(out))
```

(a) Implement the loop and sample 10 names. Compare them with the samples from the count model in §4. Are they qualitatively similar? Should they be?

(b) **Sanity check.** Compare the sampled distribution with the explicit `P` from §4: pick a row, e.g. `ix = stoi['a']`, and check that `(W @ ).softmax` produces a vector very close to `P[stoi['a']]`.

Bonus — The Same Loss in One Line of PyTorch

Bonus

We computed `probs = logits.exp() / logits.exp().sum(...)` then `-probs[arange, ys].log().mean()` *by hand* to make every step explicit. PyTorch ships a numerically stable, fused implementation:

```
loss = F.cross_entropy(logits, ys)   # = -log_softmax(logits)[arange, ys].mean()
```

(a) Replace the four lines `counts …loss = -probs…` by the single `F.cross_entropy` call (still adding the regularisation term yourself). Verify the loss values are identical.

(b) Why is `F.cross_entropy` *numerically* better? (Hint: what happens to `logits.exp()` if a logit is, say, 40?)

(c) **Pre-figuring next week.** Replace the single matrix W by a two-layer MLP: `nn.Linear(27, 64) -> tanh -> nn.Linear(64, 27)`. Re-train for 200 steps. Does the loss go below the bigram bound of ~ 2.45? Does it have to?