The Art of Computer Programming 2

Week 11 — Natural Language Processing: The Bigram Character Language Model

Félix Chavelli felix.chavelli@inria.fr

May 20, 2026 · Semester 2

PSL UNIVERSITÉ PARIS

Today’s Agenda

From Autograd to Language Models

Bigrams: Counting Pairs of Characters

From Counts to Probabilities

Sampling from the Model

Evaluating the Model: Negative Log-Likelihood

The Same Model as a Neural Network

Training: Gradient Descent on the NLL

Sampling from the Trained Network

Wrap-Up & What’s Next

Part 1

From Autograd to Language Models

Where We Stand

Last week:

- Built *micrograd* from scratch.
- Computational graph + backprop + SGD.
- Trained a tiny MLP on toy data.

All the machinery is now in place. We need a real problem to apply it to.

Today — a first step into NLP:

- Treat text as sequences of tokens (here: characters).
- Build a character-level language model: predict the next character from the previous ones.
- Two routes to the same model:
  - explicit counting;
  - gradient-based neural net.

Key Idea

A language model is a probability distribution over sequences of tokens. The simplest non-trivial one — the *bigram* model — already exposes every concept needed for GPT-style transformers.

The Task: “Make More” Names

Dataset. `names.txt` — 32,033 first names, one per line: `emma`, `olivia`, `ava`, `isabella`, `sophia`, …

Goal. Train a model that *produces more things that look like names* but are not in the training set.

Sample outputs (after training, very basic model): `mor`, `axx`, `minaymoryles`, `kondlaisah`, `anchshizarie`, …

Key Idea

Character-level language modelling: each line is a sequence of characters $c_1 c_2 \ldots c_T$, and we ask the model

$$\Pr(c_t \mid c_1, \ldots, c_{t-1}) \quad \text{for every position } t.$$

The simplest approximation: condition only on c_{t-1} — the *bigram* model.

Loading the Dataset

Python

```python
words = open('names.txt', 'r').read().splitlines()
words[:8]
# ['emma', 'olivia', 'ava', 'isabella', 'sophia',
#  'charlotte', 'mia', 'amelia']

len(words)                          # 32033
min(len(w) for w in words)          # 2
max(len(w) for w in words)          # 15
```

- Each word `w` is itself many training examples: it tells us what is likely to come *first*, what follows each prefix, and what it is likely to *end* with.
- For `isabella`: `i` is a likely starter, `s` follows `i`, ..., the word ends after `a`.

Part 2

Bigrams: Counting Pairs of Characters

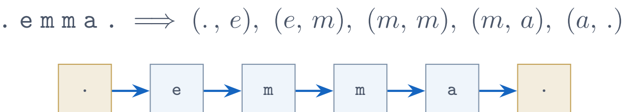
Bigrams

Bigram

A *bigram* is an ordered pair of consecutive tokens (c_{t-1}, c_t) extracted from a sequence.

Example. For the word `emma` we add a boundary token `.` on both sides:

$$\texttt{. e m m a .} \implies (.\,, e),\ (e, m),\ (m, m),\ (m, a),\ (a, .).$$



- The single token `.` marks both *start* and *end* — one symbol, two roles.
- Each bigram is a tiny supervised example: "given c_{t-1}, predict c_t".

Counting Bigrams in a Dictionary

Python

```
b = {}
for w in words:
    chs = ['.'] + list(w) + ['.']
    for ch1, ch2 in zip(chs, chs[1:]):        # slide a window of size 2
        b[(ch1, ch2)] = b.get((ch1, ch2), 0) + 1

sorted(b.items(), key=lambda kv: -kv[1])[:5]
# [(('n', '.'), 6763), (('a', '.'), 6640),
#  (('a', 'n'), 5438), (('.', 'a'), 4410),
#  (('e', '.'), 3983)]
```

- `zip(chs, chs[1:])` pairs each character with its successor.
- The most frequent bigrams confirm intuitions: many names end in `n` or `a`; `an` is very common.

From a Dict to a 2-D Tensor

A Python dict is hard to vectorise. Store the counts in a 27×27 matrix N: rows index the *previous* character, columns the *next*.

Python

```python
import torch

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

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

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
```

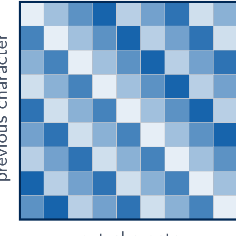
Why a tensor and not a dict?

- Constant-time indexing, vectorised arithmetic, GPU friendly.
- A perfect target for the broadcasting tricks coming up.

Visualising the Bigram Matrix

Plotting N as a heatmap (`plt.imshow`) reveals the structure:

- Row `.` = which characters *start* a name.
- Column `.` = which characters *end* a name.
- Bright cells: `an`, `na`, `el`, `ar`… very common pairs.
- Empty/white cells: combinations that *never* occur in the training set: a problem we will fix soon.



Live Notebook

See cells `plt.imshow(N, cmap='Blues')` in the notebook — the full 27×27 heatmap with bigram strings overlaid.

Part 3

From Counts to Probabilities

The Bigram Probabilistic Model

Bigram language model

A bigram model is a conditional probability distribution

$$P(c_t \mid c_{t-1}) = \frac{N[c_{t-1}, c_t]}{\sum_c N[c_{t-1}, c]}$$

estimated by maximum likelihood: the empirical frequency of pairs in the training set.

Two parameter formulations of the same model:

- the row-normalised matrix $P \in \mathbb{R}^{27 \times 27}$ (today's starting point);
- a 27×27 weight matrix W such that the rows of $\mathrm{softmax}(W)$ equal P (next section — and then we can train it with backprop).

Normalising One Row

Take the first row of N (counts of *starting* letters) and turn it into a probability distribution:

Python

```
p = N[0].float()          # cast counts to floats
p = p / p.sum()           # now p sums to 1
p.shape                   # torch.Size([27])
```

- `N[0]` grabs row index 0 (the boundary token `.`).
- Division gives a proper distribution over the 27 possible next characters.
- We will need to do this for *every* row $\Rightarrow$ this calls for a vectorised, broadcasting-aware update.

Vectorising: `P = N / N.sum(1, keepdim=True)`

Building the entire probability matrix *in one line*:

Python

```
P  = N.float()
P /= P.sum(1, keepdim=True)        # row-wise normalisation, in-place
```

Shapes involved.

- `P.shape == (27, 27)`.
- `P.sum(1, keepdim=True).shape == (27, 1)` — a column vector of row sums.
- Division (27, 27) / (27, 1) is allowed by PyTorch broadcasting rules: the singleton dimension is repeated 27 times.

Pitfall

Always pass `keepdim=True` here. Without it, the sum would have shape `(27,)`, broadcast as a *row* of shape `(1, 27)`, and we would silently normalise the *columns* instead of the rows. Same numbers, completely wrong model.

Broadcasting in 30 Seconds

Broadcasting rule (PyTorch / NumPy)

Two tensors are compatible if, when iterating their shapes from the right, every aligned pair of dimensions is either equal, or one of them is 1, or one of them does not exist. Singleton dimensions are virtually copied; missing leading dimensions are treated as 1.

Operation	Aligned shapes	Result
`P / P.sum(1, keepdim=True)`	`(27, 27) / (27, 1)`	✓ rows are normalised
`P / P.sum(1)`	`(27, 27) / (27,)`	✗ columns get normalised
`P / P.sum(0, keepdim=True)`	`(27, 27) / (1, 27)`	✓ columns are normalised

Key Idea

Broadcasting is *convenient* but *silent*: a wrong `keepdim` produces no error, just a buggy model. Always check shapes before dividing.

Part 4

Sampling from the Model

Sampling One Character: `torch.multinomial`

Once we have a probability vector $p \in \Delta_{26}$, drawing a character index from it is a categorical sample:

Python

```
g  = torch.Generator().manual_seed(2147483647)   # reproducible RNG
ix = torch.multinomial(p, num_samples=1,
                       replacement=True,
                       generator=g).item()
itos[ix]                                          # e.g. 'm'
```

- `torch.multinomial` interprets its first argument as probabilities and returns indices drawn accordingly.
- `generator=g` pins the random state $\Rightarrow$ the same seed gives the same names on every machine.
- Sanity check: drawing many samples reproduces the input frequencies.

Generating a Whole Name

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

Python

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

for _ in range(5):
    out, ix = [], 0
    while True:
        p  = P[ix]                                          # current row
        ix = torch.multinomial(p, num_samples=1,
                               replacement=True,
                               generator=g).item()
        if ix == 0:
            break
        out.append(itos[ix])
    print(''.join(out))
# mor      axx   minaymoryles   kondlaisah   anchshizarie
```

- Names are bad. They are however *strictly better* than samples from a uniform $p = \frac{1}{27}\mathbf{1}$ (try it!).
- The model only sees one character of context $\Rightarrow$ it cannot do better than this. We will need a richer model later.

Part 5

Evaluating the Model: Negative Log-Likelihood

Likelihood and Log-Likelihood

Likelihood of the data

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

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

- A *good* model assigns high probability to the data $\Rightarrow$ $\mathcal{L}$ close to 1.
- Products of many small numbers underflow $\Rightarrow$ work in log space:

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

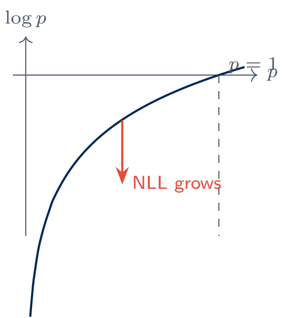
- $\log$ is monotonic $\Rightarrow$ maximising $\mathcal{L}$ is equivalent to maximising $\log \mathcal{L}$.

The Loss: Average Negative Log-Likelihood

We want a loss to minimise, low values = good model:

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

- Ranges in $[0, +\infty)$. Zero would mean the model assigns probability 1 to every observation.
- Equivalent (up to sign and scaling) to the cross-entropy between the empirical distribution of next characters and the model's distribution.
- For our names dataset the average NLL of the bigram model is ≈ 2.45.



Key Idea

Maximum likelihood = minimum negative log-likelihood = minimum cross-entropy. Three names, one objective.

Computing the Loss in PyTorch

Python

```
log_likelihood, n = 0.0, 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]
        log_likelihood += torch.log(prob)
        n += 1

nll = -log_likelihood
print(f'avg NLL = {nll / n:.4f}')          # ~ 2.45
```

Pitfall

Beware of zeros. If a bigram (c_1, c_2) never appears in the training set, $P[c_1, c_2] = 0$ and $\log P = -\infty$. A single test word containing such a pair sends the loss to infinity.

Smoothing: Add-One

Easy fix — add a fake count to every cell before normalising:

Python

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

- Every entry is now strictly positive $\Rightarrow$ no $-\infty$ losses, and every bigram becomes *possible*.
- As we add more, the distribution becomes more uniform (extreme case: $P_{ij} = 1/27$).
- This is a primitive form of regularisation; it has a principled gradient-based equivalent we'll see in a moment.

Part 6

The Same Model as a Neural Network

Why Switch to a Neural Net?

The counting model is exact and fast — but it does not scale.

- With one previous character: 27 rows of 27 $= 729$ parameters.
- With ten previous characters: $27^{10} \approx 2 \cdot 10^{14}$ rows. Impossible to store, impossible to estimate.

Key Idea

Replace the lookup table by a parametric function f_θ(context) $\rightarrow$ distribution over 27 characters, trained by gradient descent on the same NLL loss. *Same objective, same evaluation, but a model we can grow without blowing up.*

Today: f_θ is the simplest possible neural net — a single linear layer followed by a softmax. Efficient models use more complex architecures: MLPs, RNNs and transformers.

Building the Training Set: `xs` and `ys`

Every bigram becomes one supervised example:

Python

```
xs, ys = [], []
for w in words:
    chs = ['.'] + list(w) + ['.']
    for ch1, ch2 in zip(chs, chs[1:]):
        xs.append(stoi[ch1])    # input  : previous character
        ys.append(stoi[ch2])    # target : next character

xs = torch.tensor(xs)           # shape (N,)
ys = torch.tensor(ys)           # shape (N,)
print(xs.nelement())            # 228 146 bigrams
```

- `xs[i]` is an integer in $\{0, \ldots, 26\}$: the input character.
- `ys[i]` is the integer index of the *correct* next character.
- Same supervised setting as classification with 27 classes.

One-Hot Encoding

Neural nets eat *vectors*, not integers. Encode each input index as a one-hot vector of length 27:

Python

```python
import torch.nn.functional as F

xenc = F.one_hot(xs, num_classes=27).float()      # shape (N, 27)
xenc[0]                                           # [0,0,0,0,0,1,0,...,0]
```

0	0	0	0	0	1	0	0	0	0

... (27 entries total)

position 5 = letter e

Key Idea

A one-hot vector is just a *selector*: multiplying it by a matrix W picks out the corresponding row of W.

One Linear Layer + Softmax

Initialise a 27×27 weight matrix W with `requires_grad=True` so PyTorch's autograd will track it:

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

Forward pass.

Python
```
xenc   = F.one_hot(xs, num_classes=27).float()     # (N, 27)
logits = xenc @ W                                  # (N, 27): log-counts
counts = logits.exp()                              # ~ N (positive)
probs  = counts / counts.sum(1, keepdim=True)      # row-stochastic (N, 27)
```

Softmax

Given logits $z \in \mathbb{R}^K$, $\mathrm{softmax}(z)_k = \dfrac{e^{z_k}}{\sum_j e^{z_j}}$ is the standard map from real numbers to a probability distribution over K classes. The last two lines above *are* a softmax.

Why This Is Exactly the Counting Model

Counting model:

$$P[i, j] = \frac{N[i, j]}{\sum_k N[i, k]}.$$

Look up row i in P.

Neural model:

$$P_\theta[i, j] = \frac{e^{W[i, j]}}{\sum_k e^{W[i, k]}}.$$

Look up row i of W (one-hot $\times W$), then softmax.

Key Idea

The two models use the same number of parameters (27×27) and represent exactly the same family of distributions.

The counting model picks parameters by *closed-form* maximum likelihood; the neural model picks them by *gradient descent* on the NLL. **They reach the same optimum.**

But only the second approach generalises to richer contexts and deeper networks, which is the whole point.

Part 7

Training: Gradient Descent on the NLL

Picking Out the Probability of the Correct Class

We need $-\log P_\theta(c_t \mid c_{t-1})$ for every example, then average. Vectorised, with fancy indexing:

Python

```
N    = xs.nelement()                          # number of bigrams
loss = -probs[torch.arange(N), ys].log().mean()   # average NLL
```

- `torch.arange(N)` provides the row index for each example.
- `probs[torch.arange(N), ys]` plucks the probability that the model assigns to the correct next character, example by example.
- `.log().mean()` turns it into average log-likelihood; the leading minus sign turns it into the loss.

Key Idea

Every operation here (`@`, `exp`, `sum`, `/`, indexing, `log`, `mean`) is differentiable $\Rightarrow$ autograd from last week handles the entire gradient of `loss` w.r.t. `W`.

The Training Loop

Identical structure to micrograd's MLP training loop:

Python

```
for k in range(200):

    # 1) FORWARD PASS
    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(N), ys].log().mean() \
             + 0.01 * (W**2).mean()                    # regularisation

    # 2) BACKWARD PASS
    W.grad = None                                      # zero the gradient
    loss.backward()

    # 3) GRADIENT DESCENT
    W.data += -50.0 * W.grad
```

Same five steps as last week: forward, zero-grad, backward, update, repeat. After ~ 100 iterations the loss settles near 2.45, which is exactly the value reached by counting.

Regularisation = Smoothing in Disguise

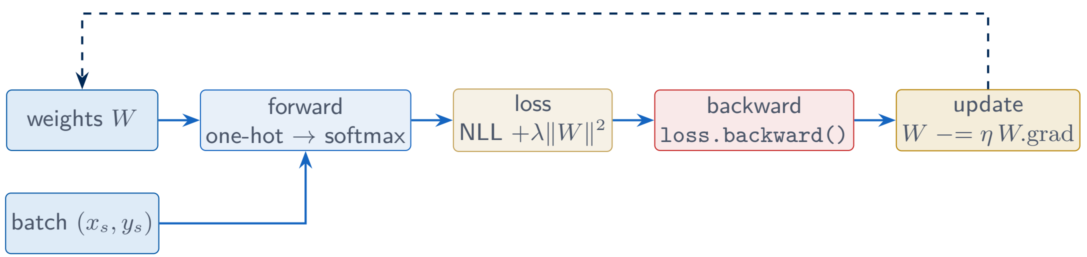
We added `0.01 * (W**2).mean()` to the loss. Why?

- Penalising large weights pushes W towards 0.
- When $W = 0$, all logits are equal, $\text{softmax}(0) = \frac{1}{27}\mathbf{1} \Rightarrow$ the model becomes uniform.
- The strength λ trades off:
  - small λ: free to fit the data tightly (sharp distributions);
  - large λ: forced toward uniformity (smooth distributions).

Key Idea

Add-one smoothing on counts and L_2 **regularisation** on weights are two faces of the same idea: gently nudge the model away from extreme, overconfident distributions.

Anatomy of One Training Step



- Every box maps line-by-line onto the code on the previous slide.
- Identical to the micrograd MLP loop: only the forward pass and the loss change.
- This pattern *is* how every modern deep network is trained.

Part 8

Sampling from the Trained Network

Generating Names with the Neural Model

Replace the table lookup `p = P[ix]` by a forward pass:

Python

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

for _ in range(5):
    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))
```

Same loop as before, same output as the counting model: the neural net has rediscovered the bigram statistics by gradient descent.

Part 9

Wrap-Up & What's Next

Recap

We built, from scratch:

- A character vocabulary with a boundary token.
- A bigram count matrix N, normalised into P.
- A sampler based on `torch.multinomial`.
- A loss (average negative log-likelihood) and its smoothing trick.
- The same model as a one-layer neural net trained by gradient descent on the NLL.

Key ideas to remember:

- Language modelling = predicting the next token.
- Maximum likelihood $\equiv$ minimum NLL $\equiv$ minimum cross-entropy.
- One-hot + matrix mult = row lookup.
- Softmax turns logits into a distribution.
- Broadcasting is silent: always check shapes.
- L_2 on weights $\approx$ add-one smoothing.

Key Idea

The bigram model is tiny but every component generalises: tokens, logits, softmax, NLL, autograd. **Transformers are this exact pipeline, with a much smarter forward pass.**

Looking Ahead

The bigram model only sees one character of context. To do better we need to feed *more* previous characters into the network. That immediately rules out a count table:

$$27^k \text{ rows for a } k\text{-gram model.}$$

The road forward keeps the same training loop and only changes the forward pass:

- MLP with character embeddings (Bengio et al., 2003);
- RNN (Mikolov et al. 2010) / LSTM (Graves et al. 2014) for arbitrary-length context;
- Transformer (Vaswani et al. 2017) — the architecture behind GPT.

Key Idea

Every modern LLM is, at heart,

$$\theta \leftarrow \theta - \eta \, \nabla_\theta \, \mathsf{NLL}(\theta)$$

on a next-token-prediction objective. Today we trained one such model end-to-end, on names, with one linear layer.

References

- A. Karpathy. *The spelled-out intro to language modeling: building makemore*.
  Video: youtube.com/watch?v=PaCmpygFfXo
  Code (MIT): github.com/karpathy/makemore
- D. Jurafsky & J. H. Martin. *Speech and Language Processing*, ch. 3 (N-gram Language Models). 3rd edition draft.
- Y. Bengio et al. *A Neural Probabilistic Language Model*. JMLR, 2003.
- Course 9 of this semester — *Autograd* (the autograd machinery we are reusing today).

End of Week 11

Thank you — questions?