GPT-From-Scratch Notes

Page 1 - Logits

Logits — Complete Definition and Guide

A clear, practical explanation of what logits are, why they exist, and how they're used in language models.

1. Intuition: What Logits Represent

Logits = the model's unfiltered opinions about which token should come next.

Softmax = convert those opinions into probabilities.

Think of it like:

- **Logits** are like raw votes or scores in a competition
- **Softmax** converts those scores into percentages that sum to 100%

Your Interpretation (Correct and Included)

A logit is like a score for each vocab token telling which token is likely to come next.

Whichever token gets the highest score is the most likely next token.

This is exactly right! The logit values directly indicate the model's preference:

- Highest logit = model's top choice
- Lowest logit = model's least preferred choice
- The actual numbers don't matter much — what matters is their relative ordering

2. What Are Logits?

Logits are the model's raw scores for each possible next token before softmax turns them into probabilities.

Key characteristics:

- Every token in the vocabulary gets one logit value
- Logits can be **positive, negative, large, or small** — they are not probabilities
- Logits are **unbounded** (any real number from -∞ to +∞)
- Higher logit ⇒ model believes that token is more likely to come next
- Softmax converts logits → probabilities

3. Why Logits Exist

Neural networks naturally output arbitrary real numbers (not probabilities).

The two-step process:

1. **Neural network** → produces logits (raw scores)
2. **Softmax function** → converts these arbitrary scores into a valid probability distribution

This separation is useful because:

- Logits are easier for the network to learn (no constraints)
- Softmax is numerically stable when applied at the end
- Loss functions like cross-entropy can work with logits directly

4. Simple Example

If the model outputs logits:

```
[2.3, 0.1, −1.5, 4.0]
```

These are **not probabilities**.

Softmax turns them into:

```
[0.06, 0.004, 0.0007, 0.935]
```

Notice:

- The highest logit `4.0` → becomes the highest probability `0.935`

- Negative logits → very small probabilities
- All probabilities sum to 1.0

5. In Your Bigram Model

```
logits[b, t, :] = the model's belief about the next token
                  after the current token at position (b, t)
```

Shape context:

- `logits` has shape `(B, T, C)`
- At each position `(b, t)`, there's a vector of `C` scores (one per vocabulary token)
- Each score represents how strongly the model predicts that token

6. Where You See Logits in Code

In the Model Forward Pass

```
# Raw scores from embedding table
logits = self.token_embedding_table(idx)  # These are logits!
# Shape: (B, T, C)

# Convert to probabilities
probs = F.softmax(logits, dim=-1)         # Now they're probabilities
# Shape: (B, T, C) but now sum to 1 along dim=-1
```

In Loss Computation

```
# Cross-entropy works directly with logits (more stable)
loss = F.cross_entropy(logits, targets)
# PyTorch applies softmax internally
```

Why cross-entropy uses logits directly:

- `F.cross_entropy` applies softmax internally for numerical stability
- Computing softmax separately can cause numerical issues with very large/small values

- This is called the "log-softmax trick"

In Generation

```
# Get logits from model
logits, _ = self(idx)

# Take last timestep
logits = logits[:, -1, :]  # (B, C)

# Convert to probabilities for sampling
probs = F.softmax(logits, dim=-1)

# Sample next token
idx_next = torch.multinomial(probs, num_samples=1)
```

7. Detailed Example with Vocabulary

Vocabulary size = 4

Suppose:

```
Token 0 = "the"
Token 1 = "a"
Token 2 = "an"
Token 3 = "."
```

Model outputs logits:

```
logits = [2.3, -1.2, 0.5, 4.7]
```

Interpretation:

- Highest logit = `4.7` → token index 3 (`"."`)
- Lowest logit = `-1.2` → token index 1 (`"a"`)

After softmax:

```
probs = [0.060, 0.002, 0.010, 0.928]
```

- Token 3 (`"."`) gets the highest probability: `92.8%`
- Token 1 (`"a"`) gets the lowest probability: `0.2%`

Therefore: The model strongly predicts `"."` as the next token.

8. Logits vs Probabilities — Side by Side

Property	Logits	Probabilities
Range	-∞ to +∞ (unbounded)	0 to 1 (bounded)
Sum	No constraint	Always sum to 1
Can be negative?	✅ Yes	❌ No
Output of	Neural network	Softmax function
Used for	Loss computation, storage	Sampling, interpretation
Interpretability	Relative scores	Direct probabilities

9. Why Both Exist — Practical Reasons

Reason 1: Numerical Stability

Computing log probabilities from logits is more stable than from probabilities:

```
# More stable (works with logits directly)
loss = F.cross_entropy(logits, targets)

# Less stable (converts to probs first)
probs = F.softmax(logits, dim=-1)
loss = -torch.log(probs[target])  # Can cause numerical issues
```

Reason 2: Easier Learning

Neural networks learn better when outputs are unbounded:

- No constraints during backpropagation
- Gradients flow more smoothly
- Network can express "very confident" or "very uncertain" predictions

Reason 3: Mathematical Convenience

Many operations are simpler with logits:

- Log-likelihood computations
- Temperature scaling for generation
- Combining predictions from multiple models

10. Common Operations with Logits

Temperature Scaling

Adjust how "confident" predictions are:

```
# Temperature > 1 → more random (flatten distribution)
# Temperature < 1 → more confident (sharpen distribution)
# Temperature = 1 → unchanged

temperature = 0.8
logits_scaled = logits / temperature
probs = F.softmax(logits_scaled, dim=-1)
```

Example:

```
# Original logits
logits = [2.0, 1.0, 0.5, 3.0]

# Temperature = 2.0 (more random)
scaled = [1.0, 0.5, 0.25, 1.5]
probs ≈ [0.25, 0.15, 0.12, 0.48]  # More uniform

# Temperature = 0.5 (more confident)
scaled = [4.0, 2.0, 1.0, 6.0]
probs ≈ [0.12, 0.02, 0.01, 0.85]  # More peaked
```

Top-K Filtering

Keep only the top K logits:

```
# Keep only top 3 tokens
top_k = 3
values, indices = torch.topk(logits, top_k)
# Set all other logits to -inf
logits_filtered = torch.full_like(logits, float('-inf'))
logits_filtered[indices] = values
```

Top-P (Nucleus) Sampling

Keep tokens until cumulative probability exceeds P:

```
# This is a simplified conceptual version
probs = F.softmax(logits, dim=-1)
sorted_probs, sorted_indices = torch.sort(probs, descending=True)
cumsum = torch.cumsum(sorted_probs, dim=-1)
# Keep tokens until cumsum > p (e.g., 0.9)
```

11. Visual Example: Logits → Softmax → Sampling

```
Step 1: Model outputs logits
        [2.3, 0.1, -1.5, 4.0]
        ↓

Step 2: Apply softmax
        [0.06, 0.004, 0.0007, 0.935]
        ↓

Step 3: Sample from distribution
        - Token 0: 6% chance
        - Token 1: 0.4% chance
        - Token 2: 0.07% chance
        - Token 3: 93.5% chance ← Most likely picked
        ↓

Step 4: Sampled token
        idx_next = 3
```

12. Quick Summary

✓ **Logits are raw scores** — unbounded real numbers from the neural network

✓ **Softmax turns logits into probabilities** — bounded values that sum to 1

✓ **Higher logit = higher probability** — relative ordering matters

✓ **Cross-entropy uses logits directly** — for numerical stability

✓ **Probabilities are for sampling** — multinomial needs probabilities, not logits

✓ **Logits vs Probabilities:**

- Logits: any number, no sum constraint
- Probabilities: 0-1 range, sum to 1

13. Common Misconceptions

Misconception 1: "Logits are just another word for probabilities"

- **Reality:** Logits are raw scores; probabilities come after softmax

Misconception 2: "Negative logits are bad or wrong"

- **Reality:** Negative logits are normal; they just mean lower probability

Misconception 3: "You always need to apply softmax to logits"

- **Reality:** For loss computation, `F.cross_entropy` handles it internally

Misconception 4: "Logits should sum to 1"

- **Reality:** Only probabilities sum to 1; logits have no sum constraint

14. When to Use What

```
| Task | Use Logits | Use Probabilities |
|------|-----------|-------------------|
| Loss computation | Yes (`F.cross_entropy`) | No (less stable) |
| Sampling next token | No | Yes (`torch.multinomial`) |
| Debugging/interpretation | Harder | Yes (easier to understand) |
| Temperature scaling | Yes (scale before softmax) | No |
| Storing model outputs | Yes (more compact) | Possible but wasteful |
```

15. Code Cheatsheet

```
# Get logits from model
logits = model(input_ids)  # (B, T, C) - raw scores

# Convert to probabilities
probs = F.softmax(logits, dim=-1)  # (B, T, C) - sum to 1

# Compute loss (uses logits directly)
loss = F.cross_entropy(logits.view(-1, C), targets.view(-1))

# Sample next token (needs probabilities)
```

```
next_token = torch.multinomial(probs, num_samples=1)

# Get most likely token (can use logits directly)
most_likely = torch.argmax(logits, dim=-1)

# Apply temperature (to logits before softmax)
scaled_logits = logits / temperature
probs = F.softmax(scaled_logits, dim=-1)
```

16. Key Takeaway

```
Logits are the raw, unprocessed scores from your neural network. They r
```

Page 2 - GPT-From-Scratch Notes

1. What GPT is trained to do

```
GPT learns a single task:
 "Given previous tokens, predict the next token."
Everything in the code is built around this one idea.
```

2. Why we create x and y

```
x = input sequence

The tokens the model sees.

Example:
    x = [18, 47, 56, 57, 58, 1, 15, 47]
y = target sequence (shifted by 1)

y is created like this:
    y = train_data[1:block_size+1]

Which means:
    • y[0] is what should come after x[0]
    • y[1] is what should come after x[1]
    • … and so on.

Example:
```

```
    x = [18, 47, 56, 57, 58, 1, 15, 47]
    y = [47, 56, 57, 58, 1, 15, 47, 58]

Intuition
    y is simply “the correct answers” for each position in x.

    Think of it like:
        x: [18]        → target should be 47
        x: [18, 47]    → target should be 56
        x: [18, 47, 56] → target should be 57
```

3. Why do we need y at all?

```
Because during training we need:
    1.  Model prediction → logits
    2.  Ground truth → y
    3.  Loss = compare prediction vs. y (cross-entropy)

Without y, the model has nothing to learn from — no “correct answer.”
```

4. What y represents mathematically

```
For a sequence:

    t0, t1, t2, t3, ...

Model input (x) is:

    t0, t1, t2, t3, ...

Expected outputs (y) are:

    t1, t2, t3, t4, ...

This is called teacher forcing:
We tell the model the true sequence and train it to predict the next to
```

5. Training vs. Inference with y

```
During training:
• You have y.
```

- You use it to compute loss.
- Everything is parallel.

During inference (generation):
- y doesn’t exist.
- You generate one token → append it → generate next.

This is why training is fast but generation is slower.

6. x and y summary (simple + powerful)

- x = what the model sees
- y = what the model should output
- y is just x shifted by 1 position
- This shift lets GPT learn next-token prediction
- All GPT training reduces to matching model’s predictions to y

Page 3 — Small Training-batch Code (with in-depth example + syntax explanations)

Below is a complete, end-to-end walkthrough of the small training-batch code, a thorough line-by-line explanation, a concrete numeric example, and a short glossary of the Python / PyTorch syntax used.

```
import torch
```

reproducibility

```
torch.manual_seed(1337)
```

hyper-parameters

```
batch_size = 4       # how many independent sequences processed in para
block_size = 8       # maximum context length for predictions
```

batch generator

```
def get_batch(split):
    # choose dataset
    data = train_data if split == 'train' else val_data
    # random start indices (each i -> sequence i : i+block_size)
    ix = torch.randint(len(data) - block_size, (batch_size,))
    # stack the input sequences: shape -> (batch_size, block_size)
    x = torch.stack([data[i : i + block_size] for i in ix])
    # targets are the next-token sequence: i+1 : i+block_size+1
    y = torch.stack([data[i + 1 : i + block_size + 1] for i in ix])
    return x, y
```

request a batch and inspect

```
xb, yb = get_batch('train')
print('input shape:', xb.shape)   # -> (batch_size, block_size)
print('target shape:', yb.shape)  # -> (batch_size, block_size)

print('input (xb):')
print(xb)

print('targets (yb):')
print(yb)

print('-----')
## fixed printing loop: use the same variable names and valid indexing
for b in range(batch_size):
    for t in range(block_size):
        context = xb[b, : t + 1]   # tokens up to and including positio
        target = yb[b, t]          # the token we want to predict at th
        print(f'when input is {context.tolist()} the target is {int(tar
```

Line-by-line explanation + intuition

```
1. torch.manual_seed(1337)
• What it does: sets PyTorch's random number generator seed to 1337.
• Why it matters: any random ops (like torch.randint) will produce the

2. batch_size = 4 and block_size = 8
• batch_size: how many independent subsequences we process at once (par
• block_size: how many tokens long each subsequence is (the context win

3. def get_batch(split): — high level
• Returns x (inputs) and y (targets) as tensors shaped (batch_size, blo
```

- split decides whether to sample from train_data or val_data.

4. data = train_data if split == 'train' else val_data
- A short if-expression (ternary). Picks the dataset.

5. ix = torch.randint(len(data) - block_size, (batch_size,))
- Purpose: choose batch_size random start indices between 0 and len(dat
- Why subtract block_size: to ensure a slice data[i : i+block_size] sta
- Shape of ix: 1-D tensor of length batch_size, e.g. tensor([1205, 875

6. x = torch.stack([data[i : i + block_size] for i in ix])
- List comprehension: creates a list of batch_size slices, each slice :
- data[i : i+block_size] returns a tensor of shape (block_size,) (a rov
- torch.stack(...) stacks these slices along a new first dimension → re
- Intuition: each row of x is one training example (an 8-token context

7. y = torch.stack([data[i + 1 : i + block_size + 1] for i in ix])
- Similar to x, but shifted by one token to the right.
- For every slice used in x, y contains the next-token targets for each
- This alignment means x[b, t] should predict y[b, t].

8. return x, y
- The training loop uses x as input and compares model logits for x to

9. xb, yb = get_batch('train') and the print lines
- Inspect shapes and values. Typical print:

```
input shape: torch.Size([4, 8])
target shape: torch.Size([4, 8])
```

- Printing the tensors shows token ids — integers representing tokens.

10. The nested loops printing context → target

```
for b in range(batch_size):
    for t in range(block_size):
        context = xb[b, : t + 1]
        target = yb[b, t]
        print(f'when input is {context.tolist()} the target is {int
```

- Outer loop b iterates over each sequence in the batch.
- Inner loop t iterates over positions inside the sequence.
- context = xb[b, : t + 1] produces a growing context for position t (p
- yb[b, t] is the next-token target for that context.
- context.tolist() converts a tensor to a plain Python list (easy print
- int(target) converts a 0-dim tensor to a Python int.

This loop outputs human-readable examples of next-token prediction for

Concrete numeric example (walkthrough)

```
Assume a tiny toy train_data for illustration (token ids are small inte

    train_data = torch.tensor([
    10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120
    ])
    length = 12

With block_size = 4 and batch_size = 2 (toy sizes for readability):
• len(data) − block_size = 8 → torch.randint(8, (2,)) might give ix = t

Now slices:
• For i = 1:
        x[0] = data[1:1+4] = [20, 30, 40, 50]
        y[0] = data[2:2+4] = [30, 40, 50, 60]
• For i = 5:
        x[1] = data[5:9] = [60, 70, 80, 90]
        y[1] = data[6:10] = [70, 80, 90, 100]

Final x and y (stacked):

    x = tensor([[20,30,40,50],
                [60,70,80,90]])   # shape (2,4)

    y = tensor([[30,40,50,60],
                [70,80,90,100]])  # shape (2,4)

Printing loop output (expanded):

    Sequence b=0:
    when input is [20] the target is 30
    when input is [20, 30] the target is 40
    when input is [20, 30, 40] the target is 50
    when input is [20, 30, 40, 50] the target is 60

    Sequence b=1:
    when input is [60] the target is 70
    when input is [60, 70] the target is 80
    when input is [60, 70, 80] the target is 90
    when input is [60, 70, 80, 90] the target is 100

This exactly mirrors how GPT is trained: at each prefix (context) predi

———
```

```
Short syntax glossary (what each operation means)
• torch.manual_seed(n) — set RNG seed for reproducibility.
• torch.randint(high, size) — sample integers from 0..high-1, shape siz
• data[i : j] — Python-style slicing; returns tokens at indices i throu
• [ ... for i in ix ] — list comprehension; builds a list by iterating
• torch.stack(list_of_tensors) — stack tensors along a new first dimens
• tensor.shape — gives tensor dimensions (e.g. torch.Size([4,8])).
• tensor.tolist() — convert tensor to Python nested lists.
• int(tensor) — cast single-value tensor to Python int.
• f"{...}" — f-string for formatted printing.

____

Short notes on alternatives & micro-optimizations
• Current get_batch uses Python list comprehension + torch.stack. For s
• Ensure len(data) - block_size > 0 or torch.randint will error.
```

Diagram — ix and slices (visual)

```
Assumptions for the diagram:
• block_size = 8
• Example ix = [3, 7, 12, 20]

    train_data indices:  0   1   2   3   4   5   6   7   8   9  10  11
    train_data tokens : t0, t1, t2, t3, t4, t5, t6, t7, t8, t9, t10,t11

    ix (start indices):              ^           ^           ^
                                    3           7          12

    Slices for x (each slice length = block_size = 8):

    x[0] from i=3  -> [ t3,  t4,  t5,  t6,  t7,  t8,  t9,  t10 ]   # da
    x[1] from i=7  -> [ t7,  t8,  t9,  t10, t11, t12, t13, t14 ]   # da
    x[2] from i=12 -> [ t12, t13, t14, t15, t16, t17, t18, t19 ]   # da
    x[3] from i=20 -> [ t20, t21, t22, t23, t24, t25, t26, t27 ]   # da

    Slices for y (targets) are the same windows shifted by 1:

    y[0] from i=4  -> [ t4,  t5,  t6,  t7,  t8,  t9,  t10, t11 ]   # da
    y[1] from i=8  -> [ t8,  t9,  t10, t11, t12, t13, t14, t15 ]   # da
    y[2] from i=13 -> [ t13, t14, t15, t16, t17, t18, t19, t20 ]   # da
    y[3] from i=21 -> [ t21, t22, t23, t24, t25, t26, t27, t28 ]   # da

Notes:
- Each x[row] aligns with y[row] shifted by one token so that x[row][p
```

```
- The subtraction `len(data)-block_size` prevents choosing a start inde
```

Diagram — the printing for-loop (context growth visualization)

```
Take a single row from x, for example:

    x[0] = [ t3, t4, t5, t6, t7, t8, t9, t10 ]
    y[0] = [ t4, t5, t6, t7, t8, t9, t10, t11 ]

    The nested printing loop:

    for t in range(block_size):
        context = x[0, : t+1]
        target  = y[0, t]

    Visualized as (t from 0..7):

    t=0: context = [ t3 ]                      => target = t4
    t=1: context = [ t3, t4 ]                 => target = t5
    t=2: context = [ t3, t4, t5 ]             => target = t6
    t=3: context = [ t3, t4, t5, t6 ]         => target = t7
    t=4: context = [ t3, t4, t5, t6, t7 ]     => target = t8
    t=5: context = [ t3, t4, t5, t6, t7, t8 ] => target = t9
    t=6: context = [ t3, t4, t5, t6, t7, t8, t9 ] => target = t10
    t=7: context = [ t3, t4, t5, t6, t7, t8, t9, t10 ] => target = t11
```

Notes:
• For each prefix (context) the model should predict the next token (target).
• During training we compute predictions for all positions in parallel; this loop is just for human inspection.

Page 4 - Bigram Language Model — Complete Master Notes

A structured, deep explanation of bigram language models covering core concepts, implementation details, and worked examples with full numeric calculations.

1. Core Concept: What is a Bigram Language Model?

A **bigram model** predicts the next token using **only** the current token.

It learns: `P(next_token | current_token)`

- No long context
- No Transformer architecture
- No RNN
- Just a next-token probability table

Training goal:

- Observe many (current_token → next_token) pairs
- Learn which next tokens are common after each current token
- Adjust weights so predictions match data statistics

2. Data Preparation: From Text to Training Pairs

Example text: `A B C D`

Extract bigrams:

```
A → B
B → C
C → D
```

Convert to token IDs: (A=0, B=1, C=2, D=3)

```
idx     = [0, 1, 2]  # current tokens
targets = [1, 2, 3]  # next tokens
```

For batch training:

- `idx`: shape `(B, T)` — input sequences
- `targets`: shape `(B, T)` — corresponding next tokens
- Each element in `targets` is a **single integer** (the correct next token ID)

3. The Embedding Layer: The Heart of the Model

```
self.token_embedding_table = nn.Embedding(vocab_size, vocab_size)
```

Shape: `(vocab_size, vocab_size)`

Why this shape?

- `vocab_size` rows → one row per possible current token
- `vocab_size` columns → one score for each possible next token

Interpretation:
Row i contains the logits (raw scores) for the next-token distribution when current_token = i.

Example (vocab_size=4):

```
row0 = [0.18, -0.07,  0.35, -0.91]  # if current token = 0
row1 = [0.62,  0.02,  0.95,  0.06]  # if current token = 1
row2 = [0.36,  1.17, -1.35, -0.51]  # if current token = 2
row3 = [0.24, -0.24, -0.92,  1.54]  # if current token = 3
```

Each row contains C scores — one for each possible next token.

Initial values: Randomly initialized by PyTorch, then learned during training.

4. Forward Pass: Producing Logits

Input: `idx` of shape `(B, T)`

Operation:

```
logits = self.token_embedding_table(idx)
```

Output shape: `(B, T, C)`

```
Where:
- `B` = batch size
- `T` = sequence length
- `C` = vocab size
```

What it does:
For each position `(b, t)`, look up the row corresponding to `idx[b,t]` and place

it in `logits[b, t, :]`.

Concrete Example

```
Given embedding table with `vocab_size = 4`:

row0 = [ 0.1808, -0.0700,  0.3596, -0.9152]
row1 = [ 0.6258,  0.0255,  0.9545,  0.0643]
row2 = [ 0.3612,  1.1679, -1.3499, -0.5102]
row3 = [ 0.2360, -0.2398, -0.9211,  1.5433]
```

Input:

```
idx = [[0, 1, 2],      # sequence 0: tokens 0 → 1 → 2
    [1, 2, 3]]      # sequence 1: tokens 1 → 2 → 3
# shape: (2, 3)
```

Output after embedding lookup:

```
logits = [
[row0, row1, row2],   # for sequence [0,1,2]
[row1, row2, row3]    # for sequence [1,2,3]
]
# shape: (2, 3, 4)

# Explicitly:
logits[0,0,:] = row0 = [ 0.1808, -0.0700,  0.3596, -0.9152]  # current
logits[0,1,:] = row1 = [ 0.6258,  0.0255,  0.9545,  0.0643]  # current
logits[0,2,:] = row2 = [ 0.3612,  1.1679, -1.3499, -0.5102]  # current
logits[1,0,:] = row1 = [ 0.6258,  0.0255,  0.9545,  0.0643]  # current
logits[1,1,:] = row2 = [ 0.3612,  1.1679, -1.3499, -0.5102]  # current
logits[1,2,:] = row3 = [ 0.2360, -0.2398, -0.9211,  1.5433]  # current
```

Interpretation:
At position [0,1] (batch 0, time 1), the current token is 1, so we get row1 which contains scores [0.6258, 0.0255, 0.9545, 0.0643] for next tokens [0, 1, 2, 3].

At each position (b,t), the model outputs C logits — one score per next-token candidate.

5. Understanding Targets

Targets are the correct next tokens for training.

Shape: `(B, T)`

Type: Each value is **one integer** — the true next token ID

Why not (B, T, C)?
Because targets are **labels**, not probability distributions. Cross entropy only needs the class index of the correct token, not a full distribution.

6. Flattening for Loss Computation

```
PyTorch's `F.cross_entropy` expects:
- `logits`: shape `(N, C)`
- `targets`: shape `(N,)`

But our model outputs:
- `logits`: shape `(B, T, C)`
- `targets`: shape `(B, T)`
```

Solution: Reshape (flatten batch and time)

```
logits_flat = logits.view(B*T, C)    # (B*T, C)
targets_flat = targets.view(B*T)     # (B*T,)
```

Key insight: This is **only reshaping** — no mathematical operations. We're converting `B` sequences of `T` predictions each into `B*T` independent predictions.

7. Cross Entropy Loss: How Learning Happens

For each of the `N = B*T` predictions:

1. **Softmax** converts logits → probabilities
2. **Select** the probability assigned to the correct class (using the integer target)
3. **Compute negative log-likelihood:**
   `NLL = -ln(p_correct)`
4. **Average** all NLL values → final loss

Formula:

```
loss = −1/N ∗ Σ ln(p_correct[i])
```

This loss drives gradient descent and weight updates.

8. Complete Forward Pass Code

```
def forward(self, idx, targets):
    # idx and targets are both (B,T) tensors of integers
    logits = self.token_embedding_table(idx)  # (B, T, C)

    B, T, C = logits.shape
    logits = logits.view(B ∗ T, C)      # flatten to (B∗T, C)
    targets = targets.view(B ∗ T)       # flatten to (B∗T)

    loss = F.cross_entropy(logits, targets)
    return logits, loss
```

9. What the Model Actually Learns

During training, backpropagation adjusts the embedding rows:

- If token A is **commonly** followed by B → **increase** `logit(A, B)`
- If token A is **rarely** followed by C → **decrease** `logit(A, C)`

End result:
Each row `i` represents the learned probability distribution of tokens that follow token `i` in the training data.

This is precisely a **bigram distribution**.

10. Worked Example: Complete Numeric Walkthrough

Setup:

- `vocab_size = 4` (tokens: 0, 1, 2, 3)
- `B = 2` (batch size)
- `T = 3` (sequence length)

Embedding weights:

```
row0 = [ 0.1808, -0.0700,  0.3596, -0.9152]
row1 = [ 0.6258,  0.0255,  0.9545,  0.0643]
row2 = [ 0.3612,  1.1679, -1.3499, -0.5102]
row3 = [ 0.2360, -0.2398, -0.9211,  1.5433]
```

Input:

```
idx = [[0, 1, 2],    # sequence 0
       [1, 2, 3]]    # sequence 1

targets = [[1, 2, 3],
           [2, 3, 0]]
```

Step 1: Embedding Lookup

```
logits = emb(idx)  # shape: (2, 3, 4)
```

Result:

```
logits = [[[row0], [row1], [row2]],  # for sequence [0,1,2]
          [[row1], [row2], [row3]]]  # for sequence [1,2,3]
```

Step 2: Flatten

```
logits_flat = [row0, row1, row2, row1, row2, row3]  # (6, 4)
targets_flat = [1, 2, 3, 2, 3, 0]                    # (6,)
```

Step 3: Compute Loss (for each of 6 predictions)

Prediction 0: `row0`, target=1

```
logits:  [ 0.1808, -0.0700,  0.3596, -0.9152]
exp:     [1.1980,  0.9324,  1.4328,  0.4011]
sum:     3.9643
probs:   [0.3023,  0.2354,  0.3614,  0.1012]
p_correct = probs[1] = 0.2354
nll = -ln(0.2354) = 1.444
```

Prediction 1: `row1`, target=2

```
probs:   [0.2851,  0.1564,  0.3961,  0.1625]
p_correct = 0.3961
nll = 0.926
```

Prediction 2: `row2`, target=3

```
probs:   [0.2605,  0.5836,  0.0470,  0.1090]
p_correct = 0.1090
nll = 2.218
```

Prediction 3: `row1`, target=2 (repeated)

```
nll = 0.926
```

Prediction 4: `row2`, target=3 (repeated)

```
nll = 2.218
```

Prediction 5: `row3`, target=0

```
probs:   [0.1776,  0.1104,  0.0558,  0.6559]
p_correct = 0.1776
nll = 1.727
```

Step 4: Final Loss

```
nlls = [1.444, 0.926, 2.218, 0.926, 2.218, 1.727]
mean_loss = sum(nlls) / 6 = 9.459 / 6 = 1.577
```

This is exactly what `F.cross_entropy()` computes.

11. Complete Code Example with Execution

```
    Here's the full code with a concrete example using `B=2`, `T=3`, `
```

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

Set seed for reproducibility
torch.manual_seed(1337)

class BigramLanguageModel(nn.Module):
    def __init__(self, vocab_size):
        super().__init__()
        # each token directly reads off the logits for the next
        self.token_embedding_table = nn.Embedding(vocab_size, v

    def forward(self, idx, targets):
        # idx and targets are both (B,T) tensor of integers
        logits = self.token_embedding_table(idx)  # (B, T, C)

        B, T, C = logits.shape
        logits = logits.view(B*T, C)
        targets = targets.view(B*T)

        loss = F.cross_entropy(logits, targets)
        return logits, loss

# Initialize model
vocab_size = 4
m = BigramLanguageModel(vocab_size)

# Create input data: 2 batches, 3 tokens each
xb = torch.tensor([[0, 1, 2],   # batch 0
                   [1, 2, 3]])  # batch 1

yb = torch.tensor([[1, 2, 3],   # targets for batch 0
                   [2, 3, 0]])  # targets for batch 1

print("Input idx (xb):")
print(xb)
print(f"Shape: {xb.shape}\n")

print("Targets (yb):")
print(yb)
print(f"Shape: {yb.shape}\n")

# Print the embedding table
print("Embedding table (token_embedding_table.weight):")
print(m.token_embedding_table.weight.data)
```

```
    print(f"Shape: {m.token_embedding_table.weight.shape}\n")

    # Forward pass
    logits, loss = m(xb, yb)

    print("Output logits (after flattening):")
    print(logits)
    print(f"Shape: {logits.shape}\n")

    print(f"Loss: {loss.item():.4f}")
```

Expected Output:

```
Input idx (xb):
tensor([[0, 1, 2],
        [1, 2, 3]])
Shape: torch.Size([2, 3])

Targets (yb):
tensor([[1, 2, 3],
        [2, 3, 0]])
Shape: torch.Size([2, 3])

Embedding table (token_embedding_table.weight):
tensor([[ 0.1808, -0.0700,  0.3596, -0.9152],  # row0
        [ 0.6258,  0.0255,  0.9545,  0.0643],  # row1
        [ 0.3612,  1.1679, -1.3499, -0.5102],  # row2
        [ 0.2360, -0.2398, -0.9211,  1.5433]]) # row3
Shape: torch.Size([4, 4])

Output logits (after flattening):
tensor([[ 0.1808, -0.0700,  0.3596, -0.9152],  # row0 (for token 0,
        [ 0.6258,  0.0255,  0.9545,  0.0643],  # row1 (for token 1,
        [ 0.3612,  1.1679, -1.3499, -0.5102],  # row2 (for token 2,
        [ 0.6258,  0.0255,  0.9545,  0.0643],  # row1 (for token 1,
        [ 0.3612,  1.1679, -1.3499, -0.5102],  # row2 (for token 2,
        [ 0.2360, -0.2398, -0.9211,  1.5433]]) # row3 (for token 3,
Shape: torch.Size([6, 4])

Loss: 1.5765
```

Step-by-Step Breakdown:

1. Input Setup:
- `xb` contains 2 sequences of 3 tokens each
- `yb` contains the corresponding next tokens

2. Embedding Lookup:
- Token `0` → retrieves `row0` from embedding table
- Token `1` → retrieves `row1`
- Token `2` → retrieves `row2`
- Token `3` → retrieves `row3`

3. Before Flattening:
`python logits (2, 3, 4) = [ [[row0], [row1], [row2]], # batch 0: tokens [0,1,2] [[row1], [row2], [row3]] # batch 1: tokens [1,2,3] ]`

4. After Flattening:
- `logits`: `(6, 4)` — 6 independent predictions
- `targets`: `[1, 2, 3, 2, 3, 0]` — 6 correct answers

5. Loss Calculation:
- For each of the 6 rows, softmax is applied
- The probability of the correct class (from `targets`) is extracted
- Negative log-likelihood is computed for each
- All 6 NLLs are averaged → `loss = 1.5765`

Verify the Shapes:

```
    ```python
    # You can add these print statements to see intermediate shapes:
    logits_before_flatten = m.token_embedding_table(xb)
    print(f"Logits before flatten: {logits_before_flatten.shape}")  # (

    logits_after_flatten = logits_before_flatten.view(2*3, 4)
    print(f"Logits after flatten: {logits_after_flatten.shape}")    # (

    targets_flat = yb.view(2*3)
    print(f"Targets after flatten: {targets_flat.shape}")           # (
    ```

    This example directly corresponds to the numeric walkthrough in Sec
```

12. Visual Diagrams

Embedding Lookup

```
    emb.weight (vocab × vocab)
_________________________________
| row0 | row1 | row2 | row3 | ...
_________________________________
    ↓ lookup via idx[b,t]

idx (B,T):              logits (B,T,C):
[[0, 1, 2],      →      [[[row0], [row1], [row2]],
[1, 2, 3]]              [[row1], [row2], [row3]]]
```

Flattening

```
logits (2,3,4)          →    logits_flat (6,4)
[[[row0],[row1],[row2]],     [row0, row1, row2,
[[row1],[row2],[row3]]]      row1, row2, row3]

targets (2,3)           →    targets_flat (6)
[[1,2,3],                    [1, 2, 3,
[2,3,0]]                     2, 3, 0]
```

Loss Computation Flow

```
logit vector → softmax → probabilities → select p[target] → −log(p

                                                  average all NLLs → fina
```

12. Shape Summary

Stage	Shape	Description
`idx`	`(B, T)`	Input token IDs
`targets`	`(B, T)`	True next token IDs
`emb.weight`	`(vocab, vocab)`	Learnable lookup table
`logits`	`(B, T, C)`	Raw prediction scores
`logits_flat`	`(B*T, C)`	Flattened for loss
`targets_flat`	`(B*T,)`	Flattened targets
`loss`	scalar	Single number to minimize

13. Quick Revision Checklist

```
✓ Bigram model: Next token depends **only** on current token
✓ Embedding table: Shape `(vocab, vocab)` — each row = next-token l
✓ Forward pass: `emb(idx)` produces `(B, T, C)` logits
✓ Targets: Shape `(B, T)` — one correct class index per position
✓ Flattening: Reshape to `(B*T, C)` and `(B*T)` — no math, just rec
✓ Cross entropy: Computes `-log(p_correct)` averaged over all predi
✓ Learning: Updates embedding rows to match next-token statistics i
```

14. End-to-End Learning Flow

```
Text → tokenize → idx (B,T)
                    ↓
        Embedding lookup (vocab×vocab)
                    ↓
            Logits (B,T,C)
                    ↓
            Flatten to (B*T,C)
                    ↓
    Cross entropy with targets (B*T)
                    ↓
                Loss
                    ↓
        Backward pass (gradients)
                    ↓
    Update embedding weights

After many iterations, the embedding table encodes bigram statistics fr
```

Page 5 - Biagram Model - Better Version

```
    ```python
    class BigramLanguageModel(nn.Module):

        def __init__(self):
            super().__init__()
            # each token directly reads off the logits for the next tol
            self.token_embedding_table    = nn.Embedding(vocab_size, n_
            self.position_embedding_table = nn.Embedding(block_size, n_
            self.lm_head                  = nn.Linear(n_embd, vocab_siz
```
```

```
        def forward(self, idx, targets=None):
            B,T = idx.shape

            # idx and targets are both (B,T) tensor of integers
            tok_emb = self.token_embedding_table(idx) # (B,T,C)
            pos_emb = self.position_embedding_table(torch.arange(T, dev
            x       = tok_emb + pos_emb # (B,T,C)
            logits  = self.lm_head(x) # (B,T, vocab_size)

            if targets is None:
                loss = None
            else:
                B, T, C = logits.shape
                logits  = logits.view(B*T, C)
                targets = targets.view(B*T)
                loss    = F.cross_entropy(logits, targets)

            return logits, loss

        def generate(self, idx, max_new_tokens):
            # idx is (B, T) array of indices in the current context
            for _ in range(max_new_tokens):
                # get the predictions
                logits, loss = self(idx)
                # focus only on the last time step
                logits = logits[:, -1, :] # becomes (B, C)
                # apply softmax to get probabilities
                probs = F.softmax(logits, dim=-1) # (B, C)
                # sample from the distribution
                idx_next = torch.multinomial(probs, num_samples=1) # (B
                # append sampled index to the running sequence
                idx = torch.cat((idx, idx_next), dim=1) # (B, T+1)
            return idx
    ```
```

# Bigram Model: Updated `__init__` Notes

```
Below are clean, structured notes for the updated `__init__` block of t
```

Updated Code Block

```
    ```python
    self.token_embedding_table    = nn.Embedding(vocab_size, n_embd)
    self.position_embedding_table = nn.Embedding(block_size, n_embd)
```

```
    self.lm_head                    = nn.Linear(n_embd, vocab_size)
    ```
```

## 1. `self.token_embedding_table = nn.Embedding(vocab_size, n_embd)`

### What this does

```
Creates a table that maps each token ID into a learned vector of size
```

Why we do this

```
Instead of using the old bigram (vocab × vocab) table, we now embed to
```

Output shape

```
If input `idx` has shape `(B, T)`, then:

* `tok_emb = token_embedding_table(idx)` → `(B, T, n_embd)`
```

Example

```
    Assume:

    * `vocab_size = 5`
    * `n_embd = 4`
    * `idx = [[2, 0, 4]]` → shape `(1, 3)`

    `token_embedding_table` internally has shape `(5, 4)`.

    Output:

    ```
    tok_emb.shape = (1, 3, 4)
    ```

    This means:

    * 1 batch
    * 3 tokens
    * each token is represented by a 4D embedding vector
```
```

2. `self.position_embedding_table = nn.Embedding(block_size, n_embd)`

What this does

```
Creates **learnable position embeddings** giving each time-step a uniqu
```

Why this exists

```
Transformers don't know token positions by default; positions must be a
```

Output shape

````
If sequence length is `T`, then:

```
pos_emb = position_embedding_table(torch.arange(T))   # shape (T, n
```

This broadcasts to `(B, T, n_embd)` when added to token embeddings.
````

Example

````
Assume:

* `block_size = 8`
* `n_embd = 4`
* `T = 3`

```
pos_emb = position_embedding_table([0,1,2])
pos_emb.shape = (3, 4)
```

When added to token embeddings:

```
tok_emb (1, 3, 4) + pos_emb (3, 4) → (1, 3, 4)
```
````

```
    This adds position information to each token.
```

3. `self.lm_head = nn.Linear(n_embd, vocab_size)`

What this does

```
A linear layer converting hidden embeddings `(B, T, n_embd)` into vocab
```

Why it exists

```
This layer is the **final prediction step**: it turns the model's repre
```

Output shape

```
    ```
    logits = lm_head(x)
    logits.shape = (B, T, vocab_size)
    ```
```

Example

```
    Assume:

    * `n_embd = 4`
    * `vocab_size = 5`
    * input `x` has shape `(1, 3, 4)`

    After `lm_head`:

    ```
    logits.shape = (1, 3, 5)
    ```

    Meaning:

    * For each of the 3 positions,
    * the model outputs **5 logits** (one for each token in the vocabul
```

Summary Diagram

```
idx (B,T)
│
▼
token_embedding_table → tok_emb (B,T,n_embd)
│
position_embedding_table → pos_emb (T,n_embd)
│
▼
x = tok_emb + pos_emb   → (B,T,n_embd)
│
▼
lm_head(x) → logits      → (B,T,vocab_size)
```

Bigram Language Model — forward() Function Notes (Clean and Detailed)

```
Below are the complete, organized notes explaining the **forward()** me
```

1. Understanding `forward(self, idx, targets=None)`

```
The `forward()` function computes the logits for each input token a

This version of the model contains:

* Token embeddings
* Position embeddings
* A Linear layer to convert hidden states to vocabulary logits
```

2. Line-by-Line Breakdown

2.1 Extracting batch size (B) and sequence length (T)

```
```python
B, T = idx.shape
```
```

Meaning:

```
* `idx` is a tensor of token IDs with shape **(B, T)**.
* This line extracts:

* `B` → number of sequences in batch
* `T` → number of tokens in each sequence
```

Example:

```
```python
idx = tensor([[4, 10, 7],
              [3,  1, 0]])   # shape (2,3)
```

Then:

* `B = 2`
* `T = 3`
```

2.2 Token Embeddings

```
```python
tok_emb = self.token_embedding_table(idx)  # (B, T, C)
```

Meaning:

* Converts each token ID into its corresponding embedding vector.
* Output shape becomes **(B, T, C)**.
* `C = n_embd` (embedding dimension).

Example:

Assume:

* `n_embd = 4`
* Token 4 → `[0.2, −0.1, 0.6, 0.9]`
* Token 10 → `[1.1, 0.8, −0.3, 0.7]`

Then:

```python
tok_emb = [
[ [0.2, −0.1, 0.6, 0.9],
    [1.1,  0.8, −0.3, 0.7],
```
```

```
    [0.5, −0.4, 0.0, 1.2] ]
]
```

Shape → **(1, 3, 4)**

2.3 Position Embeddings

```python
pos_emb = self.position_embedding_table(torch.arange(T, device=devi
```

Meaning:

* Creates embeddings for each position: 0, 1, 2, ..., T−1.
* Shape = **(T, C)**.
* Only one set of T positional embeddings is created per forward pa

Example:

If `T = 3`, then:

```python
torch.arange(T) → [0, 1, 2]
```

Assume positional vectors:

* pos0 → `[0.9, 0.1, 0.8, −0.3]`
* pos1 → `[0.4, 0.7, 0.2,  0.0]`
* pos2 → `[−0.2, 0.9, 1.1, 0.5]`

Then:

```python
pos_emb = [
[0.9, 0.1, 0.8, −0.3],
[0.4, 0.7, 0.2,  0.0],
[−0.2, 0.9, 1.1, 0.5]
]
```

Shape → **(3, 4)**

2.4 Adding Token + Position Embeddings

```python
x = tok_emb + pos_emb  # (B, T, C)
```

Meaning:

* Position embeddings give each token a sense of *where* it is in t
* Broadcasting makes shapes compatible:

* Token embeddings → (B, T, C)
* Position embeddings → (T, C) → broadcast → (1, T, C)

Example (Position 1):

```python
tok_emb[0,1] = [1.1, 0.8, -0.3, 0.7]
pos_emb[1]   = [0.4, 0.7, 0.2, 0.0]
```

Sum:

```python
x[0,1] = [1.5, 1.5, -0.1, 0.7]
```

    Shape remains **(B, T, C)**.

2.5 Linear Layer → Logits

```python
logits = self.lm_head(x)  # (B, T, vocab_size)
```

Meaning:

* Converts each hidden vector into a vector of size `vocab_size`.
* Produces **logits**, raw scores for each possible next token.
* Shape: **(B, T, vocab_size)**.

Example:

If:

* `vocab_size = 65`
* `x.shape = (2, 3, 4)`

````
Output:

```python
logits.shape = (2, 3, 65)
```

Meaning:

* For every batch and time position, the model outputs 65 scores.
````

3. Summary of All Shapes

```
| Component                | Shape               |
| ------------------------ | ------------------- |
| `idx`                    | (B, T)              |
| `tok_emb`                | (B, T, C)           |
| `pos_emb`                | (T, C)              |
| `x = tok_emb + pos_emb` | (B, T, C)           |
| `logits`                 | (B, T, vocab_size) |
```

Page 6 — Bigram Language Model — Generate Function Notes

```
A comprehensive guide to understanding the `generate()` function in big
```

1. What `generate()` Does

````
The purpose of `generate()` is to produce new tokens one-by-one. It

1. Looks at the current sequence `idx`
2. Predicts the next token
3. Appends it to the sequence

This continues for a specified number of iterations to generate new
````

2. Function Signature and Inputs

```
    ```python
    def generate(self, idx, max_new_tokens):
    ```
```

Parameters:
- `idx` → Current context. Shape: `(B, T)`
- This is the starting sequence the model continues from
- `B` = batch size
- `T` = current sequence length

```
    - **`max_new_tokens`** → How many tokens to generate
    - If you want 20 new tokens, pass `max_new_tokens=20`
```

3. The Generation Loop

```
    ```python
    for _ in range(max_new_tokens):
    ```


    - Runs once per generated token
    - `_` is used because the loop variable is not needed
```

4. Calling the Model: `logits, loss = self(idx)`

What This Line Does

```
```python
logits, loss = self(idx)
```
```

Simple explanation:

```
    - Calls the model's `forward()` with the current `idx`
    - Returns raw prediction scores (logits) and (optional) loss
```

Why we run the model here:

```
- We need the model's predictions for the current context to decide
```

Shapes and Flow

Input:

```
- `idx` with shape `(B, T)` where `T` = current context length
```

Ideal output (generation-friendly):

```
- `logits` shape = `(B, T, C)` where `C` = vocab size
- `logits[b, t, :]` = scores for possible next tokens at position
```

Loss:

```
- Only meaningful during training
- Ignored during generation (often `None`)
```

Training vs Generation Mismatch

Common problem:

```
- Training code often flattens logits to `(B*T, C)` for `F.cross_en
- If `forward()` returns `(B*T, C)` during generation, the time dim
- Indexing `logits[:, -1, :]` will fail
```

Robust Pattern (Recommended)

```
```python
logits, _ = self(idx)           # could be (B, T, C) OR (B*T, C)
if logits.ndim == 2:            # flattened -> restore time dim
    B, T = idx.shape
    C = logits.shape[1]
    logits = logits.view(B, T, C)
# now logits is (B, T, C)
last_logits = logits[:, -1, :]  # shape (B, C)
```
```

Concrete Numeric Example

Setup:

```
- `B=2`, `T=3`, `C=4`
```

Input:

```
    ```python
    idx = [[0, 1, 2],
        [1, 2, 3]]   # shape (2,3)
    ```
```

Logits from embedding (B,T,C):

```
    ```python
    logits = [
    [[10,20,30,40], [50,60,70,80], [90,91,92,93]],   # batch 0
    [[ 1, 2, 3, 4], [ 5, 6, 7, 8], [ 9,10,11,12]]    # batch 1
    ]
    # shape: (2, 3, 4)
    ```
```

If `forward()` returned flattened `(B*T, C) = (6,4)`:

`python [[10,20,30,40], [50,60,70,80], [90,91,92,93], [1,2,3,4], [5,6,7,8], [9,10,11,12]]`

Safe code reshapes back to (2,3,4) and picks last timestep:

`python last_logits = [[90,91,92,93], [ 9,10,11,12]] # shape (2,4)`

Key Points to Remember

```
    ✓ `self(idx)` should work without targets (use `targets=None` defau
    ✓ After calling `self(idx)`, assert `logits.ndim == 3` or reshape i
    ✓ Ignore `loss` during generation
    ✓ `forward()` should allow: `def forward(self, idx, targets=None):`
```

5. Handling Flattened Logits Correctly

````
If `forward()` returned flattened logits, reshape them back before

```python
logits, _ = self(idx)   # could be (B, T, C) or (B*T, C)

if logits.ndim == 2:    # flattened
    B = idx.shape[0]
    T = idx.shape[1]
    C = logits.shape[1]
    logits = logits.view(B, T, C)
```
````

This restores the correct shape `(B, T, C)`.

6. Selecting Logits From the Last Time Step

````
```python
last_logits = logits[:, -1, :]   # shape (B, C)
```

- Extracts the logits for only the final time-step in each sequence
- Shape becomes `(B, C)`
- These are the scores used to predict the next token
````

Mini Example

Given idx:

`python idx = [[0, 1, 2], [3, 1, 0]] # (B=2, T=3)`

And logits (B, T, C):

`python logits = [ [[a0], [a1], [a2]], # a2 is for last token in seq 0 [[b0], [b1], [b2]] # b2 is for last token in seq 1 ]`

Selecting last time-step:

`python last_logits = [[a2], [b2]] # shape (2, C)`

7. Converting Logits to Probabilities (Softmax)

````
```python
probs = F.softmax(logits, dim=-1)   # (B, C)
```
````

What it does:
- Converts raw logits → valid probability distribution
- Each row becomes a list of probabilities that sum to 1
- Higher logits → higher probabilities
- Shape stays `(B, C)`

Why Softmax is Needed

````
Logits are just scores (e.g., `[3, 1, −2, 4]`). They aren't probabi

Softmax transforms them into something we can sample from:
```python
[0.17, 0.02, 0.0009, 0.80]
```

Token with highest score gets highest probability.
````

How Softmax Works Internally

````
For each value:
```
softmax(x_i) = exp(x_i) / sum(exp(x_j))
```

– Makes all values positive
– Normalizes them to sum to 1
– Emphasizes differences between logits
````

Concrete Example

Input logits:

````
```python
logits = [
[10, 12, 13, 5],
[ 3,  3.2, 3.4, 3.6]
]
```
````

Softmax output:

`python probs = [ [0.0346, 0.2592, 0.7060, 0.0002], # token 2 ~70% chance [0.17, 0.21, 0.26, 0.35] ]`

```
    These probabilities are then used for sampling the next token.
```

8. Sampling the Next Token: `torch.multinomial()`

Purpose

```
This line chooses the next token ID using the probability distribution
```

Code

```
    ```python
    idx_next = torch.multinomial(probs, num_samples=1)  # (B, 1)
    ```
```

What This Does

```
    - `probs` has shape `(B, C)` → each row is a probability distributi
    - `multinomial` performs probability-weighted sampling
    - Output `idx_next` has shape `(B, 1)` → one sampled token per batc
```

Why Sampling?

```
    - `argmax` would always pick the same token → boring, repetitive te
    - `multinomial` picks tokens based on probability → creative and na
```

In-Depth Numerical Example

Assume: `B = 2`, `C = 4`

```
    ```python
    probs = [
    [0.10, 0.20, 0.60, 0.10],   # batch element 0
```
```

```
    [0.25, 0.25, 0.25, 0.25]    # batch element 1
    ]  # shape (2, 4)
    ```
```

### Step-by-step sampling:

### Batch 0 distribution:

`Token IDs: 0 1 2 3 Probs: 0.10 0.20 0.60 0.10`

```
    Token 2 has the highest chance (60%). Possible samples over many ru
    - 60% of the time → 2
    - 20% of the time → 1
    - 10% of the time → 0
    - 10% of the time → 3
```

Let's say this run samples: `2`

Batch 1 distribution:

`python [0.25, 0.25, 0.25, 0.25] # perfectly uniform`

```
    Each token equally likely. The sample could be 0, 1, 2, or 3.
```

Assume we sampled: `3`

Combined output:

`python idx_next = [[2], [3]] # shape = (B, 1) = (2, 1)`

Shape Transformation

```
    | Stage | Shape | Meaning |
    |-------|-------|---------|
    | `probs` | `(B, C) = (2, 4)` | next-token probabilities |
    | `idx_next` | `(B, 1) = (2, 1)` | sampled token IDs |
```

Summary

```
    - `torch.multinomial` chooses the next token proportionally to prob
    - Higher probability → higher chance of being selected
    - Output is `(B,1)` so we can append it to the running sequence
    - This randomness produces natural, human-like token sequences
```
```

9. Appending the Sampled Token

What This Line Does

````
This step extends the running sequence by attaching the newly sampl

```python
idx = torch.cat((idx, idx_next), dim=1)   # (B, T+1)
```

- `idx` has shape `(B, T)` → current context
- `idx_next` has shape `(B, 1)` → newly sampled next token
- `torch.cat(..., dim=1)` concatenates along the token dimension, i
- Resulting `idx` becomes shape `(B, T+1)`

This is how the model "moves forward" during text generation.
````

Why This is Needed

````
- Each iteration, the model predicts a new token
- That token must be added to the context so the next prediction us
- Without concatenation, the model would keep predicting based on t
````

Example (Step-by-Step)

````
Assume:
```python
idx = [
[0, 1, 2],
[3, 1, 0]
]   # shape (2, 3)
```

And sampled tokens:
```python
idx_next = [
[2],
[0]
]   # shape (2, 1)
```
````

````
Concatenation:
```python
idx = torch.cat((idx, idx_next), dim=1)
```

Result:
```python
idx = [
[0, 1, 2, 2],
[3, 1, 0, 0]
]
# shape (2, 4)
```

The sequence length increased from `T=3` → `T+1=4`.
````

Shape Evolution Summary

```
| Tensor | Shape | Meaning |
|--------|-------|---------|
| `idx` | `(B, T)` | Original context |
| `idx_next` | `(B, 1)` | Sampled new token |
| `torch.cat` result | `(B, T+1)` | Updated context for next iterat
```

Visual Diagram

````
```
[idx]      →   [0, 1, 2]   [3, 1, 0]
[idx_next] →       [2]         [0]
-------------------------------------
concat     →   [0, 1, 2, 2]
               [3, 1, 0, 0]
```
````

10. Full Worked Example of the `generate()` Function

```
This example shows every step of token generation using a bigram model
```

Setup

- **Batch size:** `B = 2`

- **Initial context length:** `T = 3`
- **Tokens to generate:** `max_new_tokens = 2`
- **Vocab size:** `C = 4`

Initial context (idx):

`python [ [0, 1, 2], [1, 2, 3] ]`

Example embedding rows (logits per token):

```
row0 = [ 0.18, −0.07,  0.36, −0.92]
row1 = [ 0.63,  0.03,  0.95,  0.06]
row2 = [ 0.36,  1.17, −1.35, −0.51]
row3 = [ 0.24, −0.24, −0.92,  1.54]
```

Iteration 1

Step 1 — Model Call

````
```python
logits, loss = self(idx)
```

This produces logits for every token position:
```python
logits = [
[row0, row1, row2],  # batch 0
[row1, row2, row3]   # batch 1
]
# shape = (2, 3, 4)
```
````

Step 2 — Take Last Timestep

````
```python
last_logits = logits[:, −1, :]
```

Output:
```python
[
[0.36, 1.17, −1.35, −0.51],  # from row2
[0.24, −0.24, −0.92, 1.54]   # from row3
````
````

```
    ]
    # shape = (2, 4)
```

Step 3 — Apply Softmax

```python
probs ≈ [
[0.260, 0.584, 0.047, 0.109],
[0.179, 0.111, 0.056, 0.655]
]
```

Step 4 — Sample a Token

Suppose multinomial sampling returns:

```python
idx_next = [[1], [3]]  # shape (2,1)
```

Step 5 — Append to Context

```python
idx = [
[0, 1, 2, 1],
[1, 2, 3, 3]
]
# shape = (2, 4)
```

Iteration 2

Context now has shape `(2,4)`.

Step 1 — Model Call

Last tokens are now 1 and 3, giving last logits:

```python
[
[0.63, 0.03, 0.95, 0.06],
[0.24, −0.24, −0.92, 1.54]
]
```

Step 2 — Softmax

```python
probs ≈ [
[0.286, 0.157, 0.394, 0.162],
[0.179, 0.111, 0.056, 0.655]
]
```

Step 3 — Sample

Suppose sampling returns:

```python
idx_next = [[2], [3]]
```

Step 4 — Append

```python
idx = [
[0, 1, 2, 1, 2],
[1, 2, 3, 3, 3]
]
# shape = (2, 5)
```

Final Output of `generate()`

After generating `max_new_tokens = 2` tokens:

```python
[
[0, 1, 2, 1, 2],
[1, 2, 3, 3, 3]
]
```

Final shape: `(2, 5)`

This is the final completed sequence returned by the `generate(

Summary of Shapes Through Steps

Step	Shape
Initial `idx`	`(2,3)`
`logits`	`(2,3,4)`
`last_logits`	`(2,4)`
`probs`	`(2,4)`
`idx_next`	`(2,1)`

```
        | new `idx` after 1st step | `(2,4)` |
        | new `idx` after 2nd step | `(2,5)` |


### 11. Complete `generate()` Function Code

        Here's the full implementation with all components explained:

        ```python
        def generate(self, idx, max_new_tokens):
            """
            Generate new tokens by extending the input sequence.

            Args:
                idx: (B, T) tensor of token indices (starting context)
                max_new_tokens: number of new tokens to generate

            Returns:
                (B, T+max_new_tokens) tensor with generated sequence
            """
            for _ in range(max_new_tokens):
                # Get predictions from the model
                logits, _ = self(idx)  # (B, T, C) or (B*T, C)

                # Handle flattened logits if necessary
                if logits.ndim == 2:
                    B, T = idx.shape
                    C = logits.shape[1]
                    logits = logits.view(B, T, C)

                # Focus only on the last time step
                logits = logits[:, -1, :]  # becomes (B, C)

                # Apply softmax to get probabilities
                probs = F.softmax(logits, dim=-1)  # (B, C)

                # Sample from the distribution
                idx_next = torch.multinomial(probs, num_samples=1)  # (

                # Append sampled index to the running sequence
                idx = torch.cat((idx, idx_next), dim=1)  # (B, T+1)

            return idx
        ```

12. Key Takeaways

        ✓ Purpose: `generate()` extends a sequence by predicting and ap
```

```
        ✓ Model call: `self(idx)` returns logits; handle both `(B,T,C)`

        ✓ Last timestep: Only the final position's logits matter for ne

        ✓ Softmax Converts raw logits to probabilities

        ✓ Sampling: `torch.multinomial()` creates variety; avoids repet

        ✓ Concatenation: Each new token is appended, growing the sequer

        ✓ New idx(return idx): Returns the updated sequence. After each

        ✓ Shape tracking: Critical to understand how shapes transform a


### 13. Common Issues and Solutions

        Issue 1: Shape Mismatch Error**

        Error: `IndexError` when accessing `logits[:, -1, :]`

        Cause: `forward()` returned flattened `(B*T, C)` instead of `(B

        Solution: Add reshape logic:
            ```python
            if logits.ndim == 2:
                B, T = idx.shape
                C = logits.shape[1]
                logits = logits.view(B, T, C)
            ```

Issue 2: Repetitive Output

        - Problem: Model generates the same token repeatedly

        - Cause: Using `argmax` instead of sampling, or temperature too

        - Solution: Use `torch.multinomial()` for probabilistic samplin

        - Issue 3: `targets` Required Error**

        - Error: `forward()` requires `targets` but `generate()` doesn

        - Cause: `forward()` signature doesn't allow optional targets

        -Solution: Change signature to:
        ```python
```
```

````
        def forward(self, idx, targets=None):
        ```

---

### 14. Visual Flow Diagram

    Input: idx (B, T)
        ↓
    self(idx)
        ↓
    logits (B, T, C)
        ↓
    logits[:, -1, :]
        ↓
    last_logits (B, C)
        ↓
    F.softmax()
        ↓
    probs (B, C)
        ↓
    torch.multinomial()
        ↓
    idx_next (B, 1)
        ↓
    torch.cat([idx, idx_next], dim=1)
        ↓
    new idx (B, T+1)
        ↓
    [repeat for max_new_tokens iterations]
        ↓
    Final: idx (B, T+max_new_tokens)



## Page 7 – Training Loop Notes

        Code

            ```python
            batch_size = 32
            for steps in range(10000):

            # sample a batch of data
            xb, yb = get_batch("train")

            # evaluate the loss
            logits, loss = m(xb, yb)
````

```
            optimizer.zero_grad(set_to_none = True)
            loss.backward()
            optimizer.step()

            print(loss.item())
    ```

Below are the notes explaining how the training loop works in your bigr

1. Batch Size

        ```python
        batch_size = 32
        ```

        * This sets how many independent sequences you feed the model a
        * A batch of 32 examples is processed in parallel.
        * Larger batch → more stable gradients, more memory use.

2. Training Loop Structure

        ```python
        for steps in range(10000):
        ```

        * The model trains for **10,000 iterations**.
        * Each iteration:

        * fetches a fresh batch of training data,
        * computes the loss,
        * adjusts weights using backprop.

        This entire loop is responsible for teaching the model the bigr

3. Fetching a Batch of Training Data

        ```python
        xb, yb = get_batch("train")
        ```

        * `xb`: input tokens of shape `(B, T)`
        * `yb`: correct next-token targets of shape `(B, T)`

        Example if `B=32`, `T=64`:

        ```
        ```

```
xb → (32, 64)
yb → (32, 64)
```

Every row contains a training sequence and its next-token label

4. Forward Pass: Compute Model Output & Loss

```python
logits, loss = m(xb, yb)
```

* Calls `forward(idx=xb, targets=yb)`.
* Because `targets` is **not None**, `forward()` computes cross
* Steps inside forward:

1. `logits = token_embedding_table(xb)` → `(B, T, C)` raw score
2. Reshape to `(B*T, C)`.
3. Flatten targets to `(B*T)`.
4. Compute loss between predictions and true next tokens.

The result:

* `logits` → predictions for each position in the batch
* `loss` → single scalar measuring how wrong the model is

5. Reset Gradients

```python
optimizer.zero_grad(set_to_none=True)
```

* Clears old gradients from the previous step.
* Must be done **before** backward pass.
* `set_to_none=True` is slightly faster and reduces memory.

6. Backpropagation

```python
loss.backward()
```

* Computes gradients of loss w.r.t all model parameters.
* PyTorch builds a computation graph during the forward pass.
* `.backward()` travels through the graph and accumulates gradi

````
        Example:

        * If the model predicted next tokens poorly → large gradients →


### 7. Optimizer Step

        ```python
        optimizer.step()
        ```

        * Uses the computed gradients to update model weights.
        * You're using **AdamW**, which is Adam + regularization.
        * This gradually reduces loss across iterations.


### 8. Monitoring Loss

        ```python
        print(loss.item())
        ```

        * `.item()` converts the tensor (scalar) loss into a Python flo
        * Printing loss helps verify training progress.

        If the model is learning:

        * Loss should decrease over time, e.g. → 4.0 → 3.2 → 2.7 → 2.4


### Full Training Loop Flow Summary

        ```
        repeat for steps = 1 to N:
            xb, yb = get_batch(train)
            logits, loss = model(xb, yb)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
        ```

        * This gradually optimizes the embedding table so that each tok


        Next section (when you want): add training-phase concepts like

### 7. Full Training Loop — Step-by-Step Example (Deep Explanation)
````

Below is a complete, intuitive walkthrough of what happens duri

```python
batch_size = 32
for steps in range(10000):

# sample a batch of data
xb, yb = get_batch("train")

# evaluate the loss
logits, loss = m(xb, yb)
optimizer.zero_grad(set_to_none = True)
loss.backward()
optimizer.step()

print(loss.item())
```

Example Setup

Assume:

* **Vocabulary size:** 8
* **Context length (T):** 4
* **Batch size:** 2 (small for example)

`get_batch("train")` returns:

```
xb = [[3, 5, 1, 0],
    [2, 2, 7, 4]]     # shape (2,4)

yb = [[5, 1, 0, 6],
    [2, 7, 4, 3]]     # shape (2,4)
```

`xb[b][t]` is the current token, `yb[b][t]` is the true next to

Step 1 — Forward Pass

```python
logits, loss = m(xb, yb)
```

Inside the model:

1. **Embedding Lookup** → `logits_before_flatten` has shape `(2,4,8)`.

````
   Example:

        ```
        logits_before_flatten for xb[0] (tokens 3,5,1,0):
        [
        row3,
        row5,
        row1,
        row0
        ]
        ```

        Each `rowX` contains **8 logits**, one for each next-token poss

        Example `row3` → `[0.2, 1.1, −0.7, 0.4, 2.3, −0.5, 1.8, 0.9]`

2. **Flattening for loss:**

        ```
        logits_flat shape = (2*4, 8) = (8, 8)
        ```

        ```
        targets_flat shape = (8,)
        ```

#### Step 2 — Compute Cross-Entropy Loss

        Cross entropy takes the correct next token from `yb`, finds the

        Example for a single time step:

        * logits → `[0.2, 1.1, −0.7, 0.4, 2.3, −0.5, 1.8, 0.9]`
        * softmax → `[0.05, 0.13, 0.01, 0.07, 0.56, 0.02, 0.11, 0.05]`
        * if target token = 4 → p = 0.56
        * NLL = `−log(0.56) ≈ 0.579`

        Loss is average of 8 such NLL values.

#### Step 3 — Zeroing Out Gradients

        ```python
        optimizer.zero_grad(set_to_none=True)
        ```

        This clears previous gradients so new gradients do not accumula

#### Step 4 — Backpropagation
````

```
        ```python
        loss.backward()
        ```

        This computes gradients of the loss **w.r.t.** all embedding we
        The embedding table receives gradient signals:

        * If next-token prediction was correct → reinforce that logit.
        * If wrong → push logits toward correct direction.

        Example:
        If model predicted token 2 but correct token was 4:

        * Increase rowX[4] logit
* Decrease rowX[2] logit

Step 5 — Update Weights

        ```python
        optimizer.step()
        ```

        Example:

        ```
        old_weight = 0.20
        gradient = -0.34
        learning_rate = 0.001

        new_weight = old_weight - lr * gradient
                = 0.20 - 0.001*(-0.34)
                = 0.20034
        ```

        This is how the model “learns” bigram statistics.

Step 6 — Loop Repeats

        After thousands of iterations, the embedding table encodes:

        ```
        P(next_token | current_token)
        ```

Step 7 — Final Loss

        After loop finishes:
```

```python
print(loss.item())
```

This prints the loss of the **final batch**, e.g.:

```
2.44
```

Lower loss means the model is getting better at predicting nex

Page 8 – estimate_loss() Function

Below are clean, detailed notes for the `estimate_loss()` funct

Full Code Block

```python
@torch.no_grad()
def estimate_loss():
    out = {}
    model.eval()
    for split in ['train', 'val']:
        losses = torch.zeros(eval_iters)
        for k in range(eval_iters):
            X, Y = get_batch(split)
            logits, loss = model(X, Y)
            losses[k] = loss.item()
        out[split] = losses.mean()
    model.train()
    return out
```

1. `@torch.no_grad()`

This decorator disables gradient calculation inside the functio

* During evaluation, we **don't** want gradients, because we a
* Saves memory and speeds up computation.

Example intuition:

* Training step: Track gradients so we can update parameters.
* Evaluation step: Only forward pass; gradients are unnecessary

````
### 2. `def estimate_loss():`

        Defines a helper function that will compute the **average loss>

        This is used to periodically check whether the model is improvi


### 3. `out = {}`

        Creates an empty dictionary where we will store:

        ```
        out['train'] = average training loss
        out['val']   = average validation loss
````

4. `model.eval()`

```
    Switches the model into **evaluation mode**.

    Important for layers like:

    * dropout
    * batchnorm

    These behave differently in training vs evaluation.

    **Example:**

    * Dropout (if present) will turn OFF randomness in eval mode.
```

5. `for split in ['train', 'val']:`

```
    This loop computes loss **twice**:

    1. on training data
    2. on validation data

    So `split` will first be `'train'`, then `'val'`.
```

6. `losses = torch.zeros(eval_iters)`
```

````
Creates a tensor to store multiple loss values.

If `eval_iters = 200`, then:

```
losses = tensor([0., 0., 0., ..., 0.])  # length 200
```

We will fill this with 200 different loss values to get a stable av

**Why?**

* Single batch loss is noisy.
* Average of many batches is more stable.
````

7. `for k in range(eval_iters):`

````
Runs evaluation for `eval_iters` iterations.

If `eval_iters = 200`, this loop runs **200 times**.

Each time, we sample a new batch and compute its loss.
````

8. `X, Y = get_batch(split)`

````
Calls `get_batch()` to sample a batch.

* If `split='train'`, it samples from training data.
* If `split='val'`, from validation data.
````

Example:

Assume:

````
* `batch_size = 4`
* `block_size = 8`

Then:

```
X shape = (4, 8)
Y shape = (4, 8)
````
````

```
    ```

    X = input tokens, Y = target tokens.
```

## 9. `logits, loss = model(X, Y)`

```
    Feeds the batch into the model.

    * `logits` = raw predictions for each token.
    * `loss` = cross-entropy loss for this batch.
```

Example shapes:

```
    ```
    X = (4, 8)
    logits before flattening = (4, 8, vocab_size)
    logits after flattening   = (32, vocab_size)
    loss = scalar (e.g., 2.31)
```

```
### 10. `losses[k] = loss.item()`

        Stores the numeric loss value in the `losses` tensor.

        If loss = 2.123,

        ```
        losses[k] = 2.123
```

Doing this for all `k` fills the array with many loss samples.

11. `out[split] = losses.mean()`

```
    After the inner loop finishes:

    * We compute the **average loss** across all `eval_iters` samples.

    Example:
```
```

````
losses = [2.1, 2.05, 2.08, ..., 2.03]
average = 2.06
```

Then:

```
out['train'] = 2.06
```

For validation:

```
out['val'] = 2.14
```
````

12. `model.train()`

````
Once evaluation is complete, we must switch back to **training mode

Important for layers like dropout & batchnorm.
````

13. `return out`

````
Finally, we return:

```
{
'train': avg_training_loss,
'val':   avg_validation_loss
}
```
````

Example End-to-End Walkthrough

````
Assume:

* `eval_iters = 5`
* Each loss returned by the model is:
````
````

```
Train losses: [2.20, 2.10, 2.15, 2.05, 2.12]
Val losses:   [2.50, 2.40, 2.45, 2.47, 2.43]
```

Final output:

```
{
'train': 2.124,
'val':   2.45
}
```

This tells us how well the model is doing on both datasets.

Page 9 – Multi-Head Attention: Head Size, Q/K/V, and Why We Reduce D

These notes explain **what a head is**, **what head_size means**, >

1. What Is a “Head” in Multi-Head Attention?

A **head** is one independent attention unit.

Each head performs its own:

* Query projection (Q)
* Key projection (K)
* Value projection (V)
* Attention computation

Each head learns a *different way* of looking at the same token

2. Why Do We Need `head_size`?

Originally, each token has embedding dimension:

```
n_embd  (example: 32)
```

But attention on large dimensions is expensive, so for **each h

Example:

````
```
head_size = 16
```

We create:

```
Q = Linear(32 → 16)
K = Linear(32 → 16)
V = Linear(32 → 16)
```

So every token vector is **compressed** from 32 dimensions into

head_size is simply the dimensionality of Q, K, V vectors for t


### 3. Why Can’t Each Head Use the Full 32 Dimensions?

If we have multiple heads, say:

```
num_heads = 2
n_embd = 32
```

We divide the embedding evenly:

```
head_size = 32 / 2 = 16
```

Each head analyzes different patterns.
After all heads finish:

```
concat(head_outputs) = 16 + 16 = 32 → back to original dimensio
```

This matches the Transformer design.


### 4. Are Q, K, V Matrices?

Yes.
They are **learnable linear projection matrices**.
````

For head_size = 16:

```
W_Q : (32 × 16)
W_K : (32 × 16)
W_V : (32 × 16)
```

When multiplied with a token embedding (size 32), they produce

```
Q, K, V → size 16
```

So Q/K/V **are matrices**, and each token becomes a **16-dim ve

5. Why Does `wei` Have Shape (B, T, T) and Not (B, T, head_size)?

You saw:

```
wei.shape = (4, 8, 8)
```

This comes from:

```
wei = q @ k.transpose(-2,-1)
```

Where:

```
q : (B, T, head_size)
kᵀ: (B, head_size, T)
```

Matrix multiply:

```
(B, T, head_size) @ (B, head_size, T) → (B, T, T)
```

Meaning:

* Each of the T tokens computes attention scores with all T tok
* That gives a T × T attention matrix.

````
        head_size affects depth of each token vector, not the size of t


### 6. Does a Head Run 16 Times Because head_size = 16?

        **No.**

        head_size does **not** represent number of repetitions.

        It represents:

        * how many numbers describe Q/K/V vectors
        * dimensionality of the subspace a head operates in

        The number of *repetitions* is **num_heads**, not head_size.

        Example:

        ```
        num_heads = 8
        n_embd = 512
        head_size = 64
        ```

        You get **8 heads**, not 64.
        Each head handles embeddings of size 64.


### 7. Why Reduce Dimension?

#### Reason 1 — Efficiency

        Attention scales as `T × T × head_size`. If head_size is large,

#### Reason 2 — Multi-Head Parallelization

        To allow many heads without increasing total embedding dimensio

#### Reason 3 — Specialization

        Each head focuses on different patterns:

        * next-token relation
        * punctuation
        * long-range structure
````

8. Full Example (Simple and Clear)

Let:

```
n_embd = 32
head_size = 16
B=1
T=3
```

Token embeddings:

```
x = (1, 3, 32)
```

Apply projections:

```
q = x @ W_Q → (1, 3, 16)
k = x @ W_K → (1, 3, 16)
v = x @ W_V → (1, 3, 16)
```

Attention scores:

```
wei = q @ kᵀ → (1, 3, 3)
```

Softmax → weights.

Final aggregation:

```
out = wei @ v → (1, 3, 16)
```

This final output is the head’s result.

Multiple such heads are concatenated:

```
out_total = concat(out_head1, out_head2, ...) → (1, 3, 32)
```

9. Ultra-Short Summary

```
* **head_size = dimension of Q/K/V projections**.
* Converts token embeddings from n_embd → head_size.
* Q/K/V are linear projections: (n_embd → head_size).
* Attention weights are size (T × T) regardless of head_size.
* head_size doesn't indicate number of repetitions.
* num_heads indicates number of parallel attention units.
* Outputs of all heads are concatenated back to original embedding
```

10. Single Head

```
```
class Head(nn.Module):
    """ one head of self-attention """

    def __init__(self, head_size):
        super().__init__()
        self.key = nn.Linear(n_embd, head_size, bias=False)
        self.query = nn.Linear(n_embd, head_size, bias=False)
        self.value = nn.Linear(n_embd, head_size, bias=False)
        self.register_buffer('tril', torch.tril(torch.ones(block_si

        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        # input of size (batch, time-step, channels)
        # output of size (batch, time-step, head size)
        B,T,C = x.shape
        k = self.key(x)   # (B,T,hs)
        q = self.query(x) # (B,T,hs)
        # compute attention scores ("affinities")
        wei = q @ k.transpose(-2,-1) * k.shape[-1]**-0.5 # (B, T, h
        wei = wei.masked_fill(self.tril[:T, :T] == 0, float('-inf'
        wei = F.softmax(wei, dim=-1) # (B, T, T)
        wei = self.dropout(wei)
        # perform the weighted aggregation of the values
        v = self.value(x) # (B,T,hs)
        out = wei @ v # (B, T, T) @ (B, T, hs) -> (B, T, hs)
        return out
```
```

**wei = (q @ k.transpose(-2, -1)) / (k.shape[-1]0.5) — Line

🔷 Scaled Dot–Product Attention Line

```
wei = (q @ k.transpose(-2, -1)) / (k.shape[-1]**0.5)
```

👉 This divides by $\sqrt{d_k}$, stabilizing softmax.

🔷 Step-by-Step Breakdown of the Operation

1. Shapes Before Multiplying

   `q`: (B, T, head_size)
   `k`: (B, T, head_size)

2. Transpose Keys

   ```
   k.transpose(-2, -1)
   ```

   Swaps last two dimensions → **(B, head_size, T)**

   `-1` = last dimension
   `-2` = second last dimension

3. Batched Matrix Multiplication

   ```
   (B, T, head_size) @ (B, head_size, T)
   → (B, T, T)
   ```

   Each entry `wei[b, i, j]` is the **dot product between**:

   Query at time i
   Key at time j

4. Scaling

   Multiply by `head_size**-0.5` = divide by √head_size.
   Keeps magnitudes controlled so softmax doesn't become too sharp.

🔷 Intuition for Scaling

```
* Dot product variance grows with dimensionality `d`
* Larger dot products → softmax becomes one-hot → gradients vanish
* Dividing by √d normalizes scale → stable gradients
```

🔷 Tiny Numerical Example (Easy to Follow)

```
Let:

* **B=1**, **T=2**, **head_size=2**

    **q**

        ```
        [[[1.0, 0.0],
        [0.0, 1.0]]]
        ```

    k

        ```
        [[[0.5, 0.5],
        [0.5,-0.5]]]
        ```

    Dot Products

        Before scaling:

        ```
        [[[ 0.5,  0.5],
        [ 0.5, -0.5]]]
        ```

        Scale by **1/√2 ≈ 0.7071**:

        ```
        [[[ 0.3536, 0.3536],
        [ 0.3536,-0.3536]]]
        ```
```

🔷 Important Implementation Notes

```
* `k.shape[-1]` is the head dimension `d_k`
* Scale *before* softmax
```

```
* Masking can occur before/after scaling (commonly before softmax)
```

📘 Causal Masking — wei = wei.masked_fill(self.tril[:T, :T] == 0, float('-inf'))

```
The line:

```
wei = wei.masked_fill(self.tril[:T, :T] == 0, float('-inf'))
```

implements **autoregressive (causal) masking**.
```

🔷 1. What is self.tril?

```
Created in `__init__`:

```
torch.tril(torch.ones(block_size, block_size))
```

Example for block_size = 5:

```
1 0 0 0 0
1 1 0 0 0
1 1 1 0 0
1 1 1 1 0
1 1 1 1 1
```

Meaning:

    1 = allowed
    0 = future token (forbidden)
```

🔷 2. What does [:T, :T] do?

```
Adapts mask to current sequence length.

If T = 3:

```
1 0 0
```
```

```
1 1 0
1 1 1
```

🔷 3. What does (mask == 0) produce?

```
Boolean mask:

    True → forbidden
    False → allowed
```

🔷 4. What does masked_fill(..., -inf) do?

```
Places **−inf** wherever mask=True.

Example:
Before:

```
[0.3, 0.1, 0.7]
```

After masking future:

```
[0.3, −inf, −inf]
```
```

🔷 5. Why -inf?

```
Because softmax converts:

```
[1.2, −inf, −inf]
```

into:

```
[1.0, 0.0, 0.0]
```

So future tokens get probability **0**.
```

🔷 6. Tiny Example With T=3

````
Before mask:

```
token0: [1.0, 2.0, 3.0]
token1: [0.5, 0.8, 0.1]
token2: [1.4, 0.6, 0.2]
```

After mask:

```
t0: [1.0, −inf, −inf]
t1: [0.5, 0.8, −inf]
t2: [1.4, 0.6, 0.2]
```

After softmax → no future attention.
````

Summary (Super Clear)

```
* The scaled dot-product attention computes affinity scores.
* Scaling stabilizes softmax.
* Masking prevents future attention.
* Softmax turns scores into valid attention weights.
* Result: **each token attends only to itself and the past.**
```

10 MultiHeadAttention — Class Code + Line-by-Line Notes

Code block

````
```python
class MultiHeadAttention(nn.Module):
    """ multiple heads of self-attention in parallel """

    def __init__(self, num_heads, head_size):
        super().__init__()
        self.heads = nn.ModuleList([Head(head_size) for _ in range(
        self.proj = nn.Linear(head_size * num_heads, n_embd)
        self.dropout = nn.Dropout(dropout)
````
````

```
        def forward(self, x):
            out = torch.cat([h(x) for h in self.heads], dim=-1)
            out = self.dropout(self.proj(out))
            return out
    ```
```

### Class: `MultiHeadAttention`

```
A wrapper that runs several `Head` instances in parallel, concatenates
```

Notes for `__init__(self, num_heads, head_size)`

Alright — let's break this down cleanly and deeply.
We'll go line-by-line, starting exactly where you asked.

✔ Why does `__init__` take `num_heads` and `head_size`?

```
    In multi-head self-attention, we create **multiple independent atte

    We need:
```

1. `num_heads`

   ```
   This tells us *how many* attention heads to create.

   Example:

   ```
   num_heads = 8
   ```

   Means:

   * 8 separate attention heads
   * each computes its own Q, K, V
   * each learns its own attention patterns
   ```

2. `head_size`

   ```
   This tells us how big each head's Q/K/V vectors should be.
   ```
   ```

````
Example:

```
head_size = 16
```

Implies:

* Query vector = 16 dims
* Key vector = 16 dims
* Value vector = 16 dims

Together:

```
embedding dimension = head_size * num_heads
example: 16 * 8 = 128
```

This keeps the final output dimension consistent.
````

Line 1:

````
```python
self.heads = nn.ModuleList([Head(head_size) for _ in range(num_hea
```

This is the key line.
Let's break it down:
````

What does this line actually do?

````
It creates multiple `Head` objects.

Example:

```
num_heads = 4
head_size = 16
```

Then this expands into:
````

````
```python
self.heads = nn.ModuleList([
    Head(16),
    Head(16),
    Head(16),
    Head(16)
])
```

Each `Head()` you wrote earlier:

* has its own W_Q, W_K, W_V matrices
* performs its own attention
* learns its own patterns
* outputs shape: `(B, T, head_size)`

So if you have 4 heads with `head_size=16`:

Each head output:

```
(B, T, 16)
```

Concatenated output:

```
(B, T, 16*4) = (B, T, 64)
```
````

Why use `nn.ModuleList`?

````
Because:

* PyTorch needs to register parameters for each head
* Lists in Python **don't** register parameters
* `ModuleList` does

So:

```python
ModuleList([Head(), Head(), Head()])
```

ensures that:
````

```
    * optimizer sees all parameters
    * model moves to GPU correctly
    * `state_dict` saves everything

    If you used a plain Python list, PyTorch would **ignore** all head
```

Summary of this line

```
    `self.heads = nn.ModuleList([...])` does:

    * Creates `num_heads` separate attention heads
    * Each head has its own parameters
    * Makes PyTorch track all heads
    * Allows us to run all heads in parallel later
    * Outputs can be concatenated
```

📘 MultiHeadAttention — `self.proj = nn.Linear(head_size * num_heads, n_embd)`

Below are the detailed notes explaining this line in a clean, structured format following the same template style as previous sections.

🔷 Code Context

```
    This line appears inside the **`MultiHeadAttention`** class:

    ```python
    self.proj = nn.Linear(head_size * num_heads, n_embd)
    ```
```

🔷 What This Line Does

```
    Understanding the Projection Layer (`self.proj


    This creates a **final linear projection layer** that takes the out

    The multi-head output shape before projection:

    ```
    (B, T, head_size * num_heads)
    ```
```

````
After projection:

```
(B, T, n_embd)
```

This restores the expected embedding dimension.
````

🔷 Why Is This Needed?

````
Multi-head attention works like this:

1. Split the embedding dimension into multiple "heads".
2. Each head processes attention independently.
3. Concatenate the results.

BUT the model expects each token to end up with a representation of

Example:

* `n_embd = 128`
* `num_heads = 8`
* `head_size = 16`

After each head outputs `(B, T, 16)`, concatenation produces:

```
(B, T, 16*8) = (B, T, 128)
```

Even though the size matches `n_embd`, this 128-dimensional vector

So we apply:

```python
self.proj = nn.Linear(128, 128)
```

This mixes all heads together into a useful embedding.
````

🔷 Intuition Behind This Layer

````
Each head captures *different types of relationships*:
````

````
* Some heads focus on short-distance dependencies.
* Some heads capture long-range dependencies.
* Some heads learn syntactic patterns.
* Some heads learn semantic meaning.

After concatenation, we have `num_heads` different "views" of each

### ✔ `self.proj` combines these views.

It blends all heads into a **single unified embedding** per token.

Think of it like:

> Combining 8 experts' opinions into one final answer.
````

🔷 Clear Numerical Example

````
Let:

* Batch size **`B = 2`**
* Time dimension **`T = 4`**
* `num_heads = 4`
* `head_size = 16`
* `n_embd = 64`

Each head output:

```
(B, T, 16)
```

After concatenation:

```
(B, T, 16*4) = (B, T, 64)
```

Projection layer:

```
self.proj = nn.Linear(64, 64)
```

Final output:
````

```
(B, T, 64)
```

Now the output is:

* the correct dimension (`n_embd`)
* a learned mixture of all heads
* ready for the next Transformer block

🔷 Why Not Remove This Layer?

Without this projection layer:

* Heads never mix their information.
* The network cannot learn weighting between heads.
* The output space would not align with the rest of the model.
* It would break the Transformer architecture.

This layer is essential to:
✔ Integrate all heads
✔ Transform concatenated output into a meaningful embedding
✔ Maintain architectural consistency

🔷 Final Summary

`self.proj = nn.Linear(head_size * num_heads, n_embd)` performs the

* Takes concatenated head output
* Learns how to blend heads
* Ensures output dimension matches `n_embd`
* Feeds the correct shaped embedding to the next block

This line is critical for multi-head attention to work properly.

self.dropout = nn.Dropout(dropout)

- Where Dropout Is Used in MultiHeadAttention

- Used after attention weights and after projection layer

🔷 Inside each head (attention weights):

````
    ```
    wei = self.dropout(wei)
    ```
````

🔷 Inside MultiHeadAttention final projection:

````
    ```
    out = self.dropout(self.proj(out))
    ```

    Both are needed.
````

Forward Pass Function Line –– out = torch.cat([h(x) for h in self.heads], dim=-1)

````
    Line Under Discussion

    ```
    out = torch.cat([h(x) for h in self.heads], dim=-1)
```

This line is extremely important because it combines the outputs of all
````

1️⃣ What is `self.heads`?

````
    Defined earlier in `__init__`:

    ```python
    self.heads = nn.ModuleList([Head(head_size) for _ in range(num_head
    ```

    This means:

    - It is a list containing **num_heads** separate `Head` objects.
    - Example if `num_heads = 4`:

    ```
    [Head(), Head(), Head(), Head()]
    ```

    Each head:
````

```
- Has its own Query, Key, Value matrices
- Performs its own self-attention
- Outputs shape `(B, T, head_size)`
```

2️⃣ Understanding `[h(x) for h in self.heads]`

````
This is Python list comprehension.
It runs each head on the same input `x` and collects their outputs.

So:

```
[h(x) for h in self.heads]
```

Produces something like:

```
[
head0_out,   # (B, T, head_size)
head1_out,   # (B, T, head_size)
head2_out,   # (B, T, head_size)
head3_out    # (B, T, head_size)
]
```

If `num_heads = 4`, list length = 4.
````

3️⃣ Why does each head output `(B, T, head_size)`?

````
Inside each `Head` class:

- `q`, `k`, `v` are created with dimension `head_size`
- Final output of head = `wei @ v`

Therefore each head returns:

```
(B, T, head_size)
```

Each head learns different patterns:

- One may focus on local context
````

```
- Another on long-range dependencies
- Another on syntax
- etc.
```

4️⃣ Now the key syntax: `torch.cat([...], dim=-1)`

```
We concatenate the outputs of all heads **along the last dimension**

What is `dim = -1`?

`-1` means last dimension.
For shape `(B, T, head_size)`:

- dim=0 → batch
- dim=1 → time
- dim=2 → head_size ← last dim = -1

So concatenation happens across the head_size dimension.
```

5️⃣ Final Output Shape

```
Before concatenation:

```
num_heads × (B, T, head_size)
```

After concatenation:

```
(B, T, head_size * num_heads)
```

Example

If:

* B = 2
* T = 4
* head_size = 8
* num_heads = 3

Each head output: `(2, 4, 8)`
After concat:
```

```
(2, 4, 24)
```

Because:

```
8 + 8 + 8 = 24
```

6️⃣ Why Concatenate Heads?

```
Multi-head attention gives each token several different "views":

- Semantic
- Positional
- Dependency-based
- Syntactic

Concatenation gathers all these views side-by-side into one large r

This large representation is then mixed using:

```
out = self.proj(out)
```
```

which learns how to blend the heads.

Final Intuition

```
- Each head analyzes input differently
- All heads run in parallel on the same `x`
- `torch.cat` merges all head outputs
- Output: `(B, T, num_heads * head_size)`

This is the **core operation** of Multi-Head Attention — combining
```

Line: `out = self.dropout(self.proj(out))`

What Is "Model Embedding Space"?

```
Every token is represented as a vector of size **n_embd**.

If:

```
n_embd = 128
```

Then every token → 128-dimensional vector.

This 128-dimensional vector space is called:

    Model Embedding Space
    or
    e Model's Representation Space

    It is the "language" the model thinks in.
```

Line: `out = self.dropout(self.proj(out))`

```
This single line performs **two operations**:

1. Projects the concatenated head outputs back into the model embed
2. Applies dropout** for regularization.
```

🔷 1. Breakdown — What runs first?

```
Python evaluates the expression inside-out:

1. `self.proj(out)` → linear projection
2. `self.dropout(...)` → dropout applied to the projection result
3. result assigned back: `out = ...`

Equivalent to:

```
out = self.proj(out)   # (B, T, head_size * num_heads) -> (B,
out = self.dropout(out)   # same shape, but some values zeroed du
```
```

🔷 2. Shapes Before and After

```
- Before projection: `(B, T, head_size * num_heads)`
```

```
- After projection: `(B, T, n_embd)` — back to model embedding spac
- After dropout: still `(B, T, n_embd)` — dropout never changes sha

Example:

- `B = 2`
- `T = 4`
- `head_size = 16`
- `num_heads = 4`
- `n_embd = 64`

```
Before proj   → (2, 4, 64)
After proj   → (2, 4, 64)
After dropout → (2, 4, 64)
```
```

🔷 3. Why apply the projection first?

```
Because the concatenated heads contain independent attention featu

`self.proj`:

- mixes all heads into one unified token representation
- learns how different heads should interact
- outputs a meaningful vector in the model embedding space

If dropout happened before projection, the linear layer would learn
```

🔷 4. Why apply dropout here?

```
Dropout:

- randomly zeros a fraction `p` of values (e.g. `p = 0.1`)
- prevents overfitting
- encourages redundancy and robustness
- stabilizes training across many Transformer blocks
```

🔷 5. Training vs Evaluation Behavior

```
- Training (`model.train()`) → dropout randomly zeroes elements and
- Evaluation (`model.eval()`) → dropout does nothing (tensor unchan
```

````
This ensures controlled randomness only during training.
````

🔷 6. Tiny Numeric Example

````
Assume:

- before projection: `(1, 3, 32)`
- projection layer: `Linear(32 → 32)`

Before dropout:

```
[0.5, −1.0, 0.2, ..., 0.7]
```

If dropout = 0.25, mask example:

```
[0.666, 0.0, 0.266, ..., 0.933]
```
````

Zeros appear where dropout masked values.
During evaluation → no change.

🔷 7. Final Intuition

````
Concatenate → project → dropout means:

- Combine all head outputs into a single vector (`self.proj`).
- Apply dropout to make this representation more robust.
- Preserve shape `(B, T, n_embd)` for the next Transformer block.
````

Page 10 - Diagrams

Diagram 1:

```
        Input idx → tok_emb + pos_emb  (B, T, n_embd)
                ↓
            x = tok_emb + pos_emb
                ↓
            ┌─ LayerNorm ─┐
```

```
        │ Self-Attn     │  (multi-head)
        │  - Q,K,V      │
        │  - concat     │
        │  - proj       │
        └───────┬───────┘
                │
    out1 = Dropout(proj(concat_heads(out_attn)))
                │
    x = x + out1                # residual add (post-attention)
                │
    ┌─ LayerNorm ─┐
    │  FeedForward│  (MLP: Linear → ReLU → Linear)
    └─────┬───────┘
                │
    out2 = Dropout(FFN(x))
                │
    x = x + out2                # residual add (post-FFN)
                │
(repeat Block N times)
                ↓
    x = LayerNorm_final(x)
                ↓
    logits = lm_head(x)          # (B, T, vocab_size)
                ↓
training: compute cross_entropy(logits_flat, targets_flat)
sampling: softmax(logits[:, -1, :]) → sample → append token
```

Diagram 2:

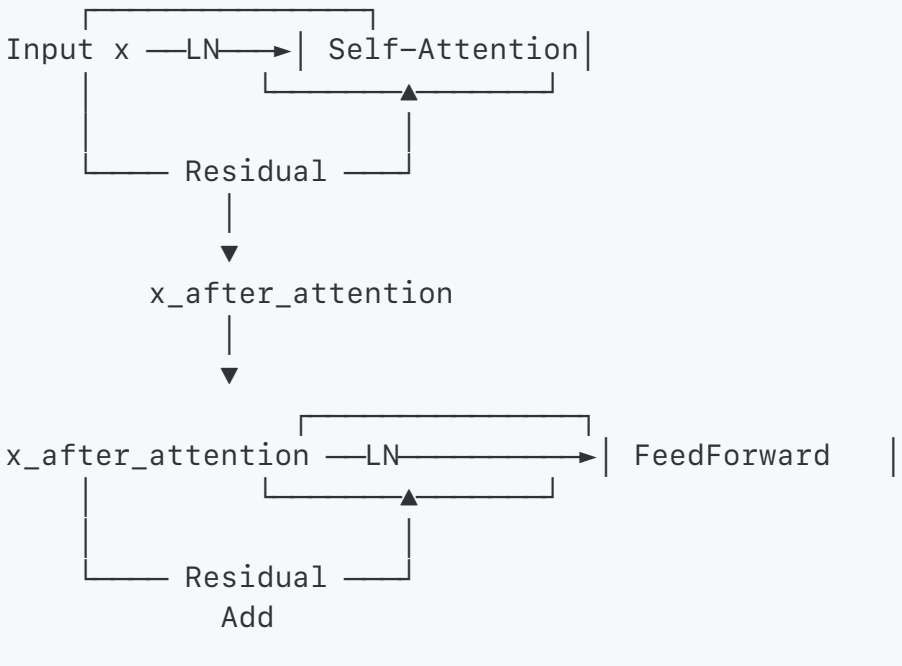

Diagram 3:

```
GPTLanguageModel
├─ token_embedding_table (Embedding)
├─ position_embedding_table (Embedding)
├─ blocks (Sequential)
│       ├─ Block
│       │     ├─ MultiHeadAttention
│       │     │       ├─ head0 (Linear, Linear, Linear)
│       │     │       ├─ head1 (Linear,...)
│       │     │       └─ projection (Linear)
│       │     ├─ FeedForward (Linear -> ReLU -> Linear)
│       │     ├─ ln1 (LayerNorm)
│       │     └─ ln2 (LayerNorm)
├─ ln_f (LayerNorm)
└─ lm_head (Linear)
```