

Building an LLM from Zero

Chapter 15: Greedy and Sampling

Module 3: Training and Generation

Truong (Jack) Luu · jackluu.io/book

Where we are

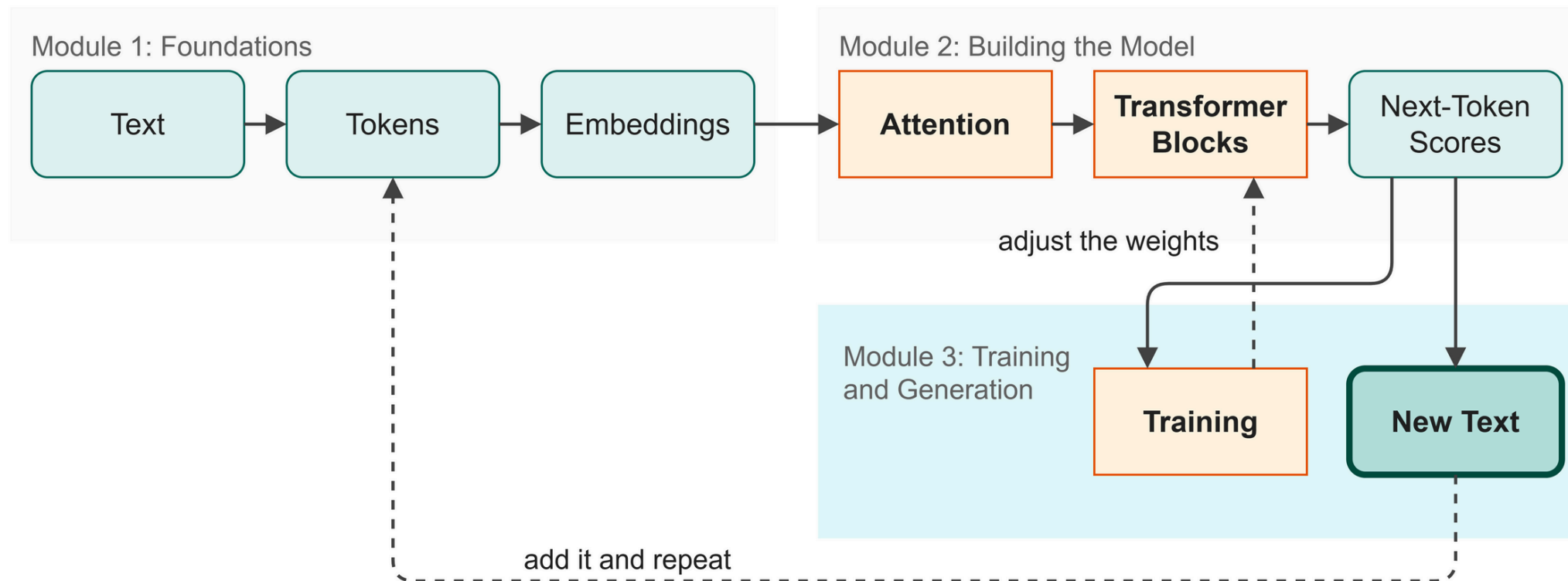


Figure 15.1: Where we are: we are ready to generate new text using the trained model.

The model is trained and loaded. Today it writes. The question is how to turn 65 scores into one character.

What this builds on (60-second recap)

- **Chapter 14** loaded the checkpoint: configuration plus weights, then `model.eval()`
- The restored model has **824,832** parameters and outputs **65 scores** at every position
- The file is **4.18 MB**

Next: 65 scores, one winner. Which rule picks it?

In this chapter you will

1. Generate text by always choosing the **highest-scoring** character (greedy decoding)
2. Generate text by choosing characters **at random, weighted by their probabilities** (sampling)
3. Compare the trade-offs between predictable and creative generation

Words to know (each is glossed again where it first appears):

- **Greedy decoding**: always picking the single highest-probability token
- **Sampling**: choosing the next token randomly, giving higher-probability tokens a better chance to be selected

Greedy decoding: always take the top score

The model outputs **logits** (raw scores, one per character in the 65-character vocabulary). The simplest rule: pick the highest.

- **Deterministic**: the same prompt always produces the same text
- **The flaw**: it gets stuck in loops. A familiar context predicts a familiar next character, which makes the context look even more familiar

Greedy is safe and predictable, and that is exactly why it repeats itself.

Sampling: draw a winner, weighted by probability

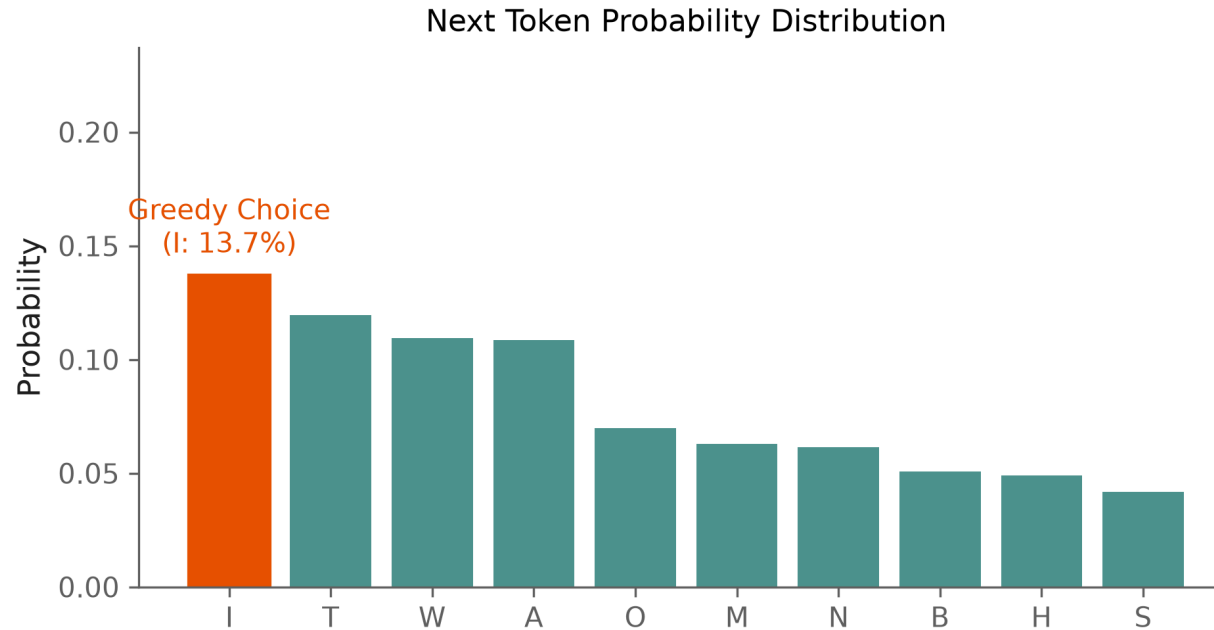


Figure 15.2: Greedy decoding always takes the top peak; sampling draws randomly from the distribution.

- Turn the scores into probabilities, then draw: **e** at 60% is chosen 60% of the time, **x** at 1% rarely
- Variety breaks the loops. The cost: the output changes every run

Sampling trades predictability for variety.

One character at a time

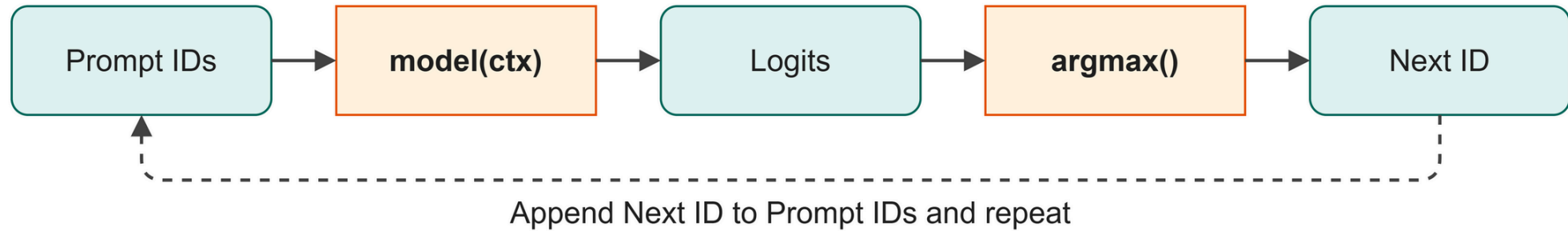


Figure 15.3: The generation loop predicts one character at a time and appends it to the context.

- Predict **one** character, add it to the prompt, repeat; the context is cropped to the last **128** characters (`block_size`)
- Only the scores at the **last position** matter for the next character

On the map, New Text loops back to Tokens. This figure is that loop, one turn at a time.

In business: which rule for which task

The house-style assistant needs both rules.

- Answering a factual question from a company manual: **greedy** (safe, predictable, exact)
- Drafting a creative marketing email in the company voice: **sampling** (varied, exploring new options)

The rule is a product decision, not a technical detail.

Code: greedy

File: `src/ch14_generate_greedy.py` (source files are numbered one lower than chapters).

```
def generate_greedy(model, prompt, encode, decode, cfg, max_new_tokens=200):  
    # Always picks the token with the highest predicted score  
    ids = torch.tensor([encode(prompt)], dtype=torch.long)  
    for _ in range(max_new_tokens):  
        ctx = ids[:, -cfg.block_size:]  
        logits = model(ctx)  
        next_id = logits[:, -1, :].argmax(dim=-1, keepdim=True)  
        ids = torch.cat([ids, next_id], dim=1)  
    return decode(ids[0].tolist())
```

- **Line 5** crops the context to the last `block_size` characters
- **Line 7** takes the single highest score with `argmax`
- **Line 8** appends the chosen character and the loop repeats

Code: sampling

Same file, same loop. Only the choice of `next_id` changes.

```
def generate_sample(model, prompt, encode, decode, cfg, max_new_tokens=200):  
    # Picks the next token randomly based on the model's probabilities  
    ids = torch.tensor([encode(prompt)], dtype=torch.long)  
    for _ in range(max_new_tokens):  
        ctx = ids[:, -cfg.block_size:]  
        logits = model(ctx)  
        probs = F.softmax(logits[:, -1, :], dim=-1)  
        next_id = torch.multinomial(probs, num_samples=1)  
        ids = torch.cat([ids, next_id], dim=1)  
    return decode(ids[0].tolist())
```

- **Line 7** turns the last position's scores into probabilities with `softmax`
- **Line 8** draws one character at random, weighted by those probabilities

Run it: greedy

Prompt: ROMEO: and a new line.

```
$ python src/ch14_generate_greedy.py
--- GREEDY (always picks highest-score token) ---
ROMEO:
I will the come the some the stand the son,
And the so shall the see the shall the see the stand
...
```

What just happened

- The output reads like Shakespeare for a few words, then locks into a loop: "the shall the see the shall the see"
- Run it again and you get exactly the same text

This is the flaw from the theory slide, on screen.

Run it: sampling

Same prompt, same model, the random rule.

```
--- SAMPLING (picks randomly from distribution) ---  
ROMEO:  
My must greather, but what it? whom, he ere.  
  
EBRUTUS:  
Morcy there bagainVain, I will ever may  
To epated as great that the appral tankswain  
...  
--- Observations ---  
Greedy tends to be more repetitive.  
Sampling is more varied but can make unexpected choices.
```

What just happened

- No loop this time: varied lines, a new speaker, a new scene
- Strange spellings ("greather", "bagainVain") appear when a low-probability character gets drawn

Same probabilities, two rules

```
$ python diagrams/charts/ch15_probabilities.py  
Saved chart: ch15-probabilities.png
```

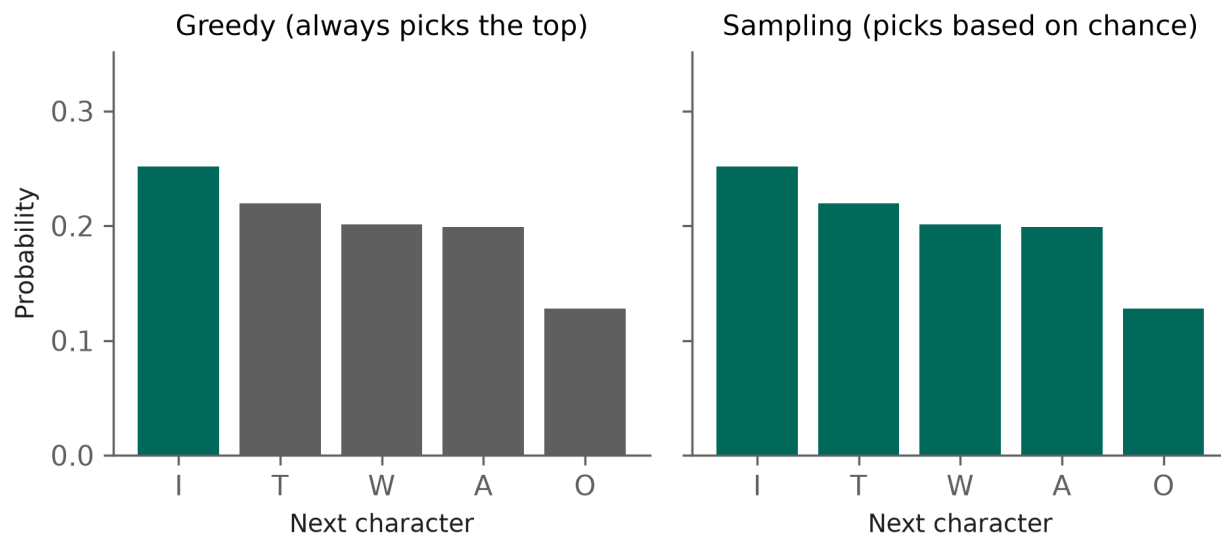


Figure 15.4: Greedy always picks the tallest bar; sampling can pick any bar based on its height.

Shape check

Tensor	Shape	What it means
<code>logits[:, -1, :]</code>	<code>[1, 65]</code>	65 vocabulary scores for the final character in the sequence
<code>next_id</code>	<code>[1, 1]</code>	1 chosen character ID

One row of 65 scores in, one character out, every turn of the loop.

Try it: different prompts

Open `src/examples/ch15_different_prompts.py`. It samples 50 characters after two different speaker names.

```
$ python src/examples/ch15_different_prompts.py
--- Prompt: 'KING:\n' ---
KING:
My mine gore to kink on Villence,
For my love grin

--- Prompt: 'JULIET:\n' ---
JULIET:
What reason this bagainVain, I will evers,
As do p
```

A different prompt is a different context, so the model starts down a different path.

Discussion

Greedy wrote "the shall the see the shall the see". Explain the loop.

In pairs, 3 minutes:

1. Trace it: what does the context look like after each pick, and why does the next pick repeat?
2. Sampling escaped the loop but wrote "bagainVain". Is that a bug or the price of variety?

Take two answers from the room.

Recap: today in 4 points

1. **Greedy decoding** always takes the highest score: predictable, and prone to repeating itself.
2. **Sampling** draws a character at random, weighted by probability: varied and natural, but unpredictable.
3. The generation loop predicts one character, appends it to the context, and repeats.
4. The context is cropped to the model's **block size** (128) before each prediction.

Next, two controls that keep sampling's variety while cutting its worst choices.

Check your understanding

1. Why does greedy decoding often get stuck repeating the same phrase?
2. What PyTorch function do we use to randomly pick a token based on its probabilities?
3. Why do we slice the logits tensor with `[:, -1, :]` in the generation loop?

Next up: Chapter 16, Temperature and Top-k

- Sampling can pick a truly bad character. Next class adds two dials: **temperature** reshapes the odds, **top-k** limits who can compete
- Both sit between the scores and the pick you saw today

Before next class: run `python src/ch14_generate_greedy.py` twice and note which half changes.

References

Book and figures

- Luu, T. (2026). *Building an LLM from Zero*, Chapter 15. <https://jackluu.io/book/>
- Figures 15.1 to 15.3 are the book's own diagrams (draw.io sources in the book repository, `diagrams/`); Figure 15.4 is plotted by `diagrams/charts/ch15_probabilities.py`.
- Terminal output: `src/ch14_generate_greedy.py` and `src/examples/ch15_different_prompts.py`.

Further reading (from the chapter)

- Holtzman, A., Buys, J., Du, L., Forbes, M., & Choi, Y. (2019). *The curious case of neural text degeneration* (arXiv:1904.09751). <https://doi.org/10.48550/arXiv.1904.09751>

Dataset

- Tiny Shakespeare, from Andrej Karpathy's char-rnn project: <https://github.com/karpathy/char-rnn>