

Building an LLM from Zero

Chapter 14: Checkpointing

Module 3: Training and Generation

Truong (Jack) Luu · jackluu.io/book

Where we are

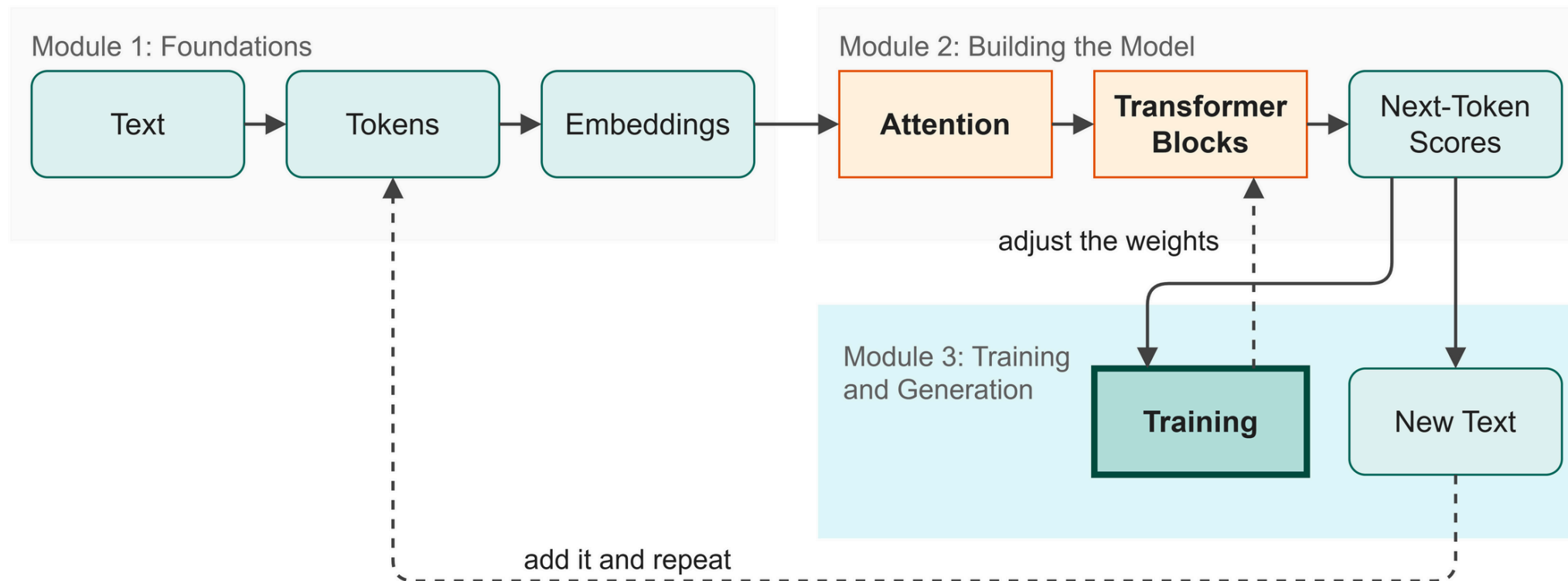


Figure 14.1: Where we are: we have trained the model and now we save its weights.

Training took time. Today we make sure that work is never lost: save the model to a file, and load it back without retraining.

What this builds on (60-second recap)

- **Chapter 13** ran the four-step loop: forward pass, loss, backward pass, optimizer step
- Over **3,000 steps** the training loss fell from about **4.3** to **1.74**
- The script saved the learned weights to a file at the end

Next: what exactly is in that file, and how do you bring the model back?

In this chapter you will

1. **Save** a trained model to a file
2. **Load** a saved model back into memory
3. Switch the model from **training mode** to **evaluation mode**

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

- **Checkpoint:** a file containing the saved state (weights and configuration) of a model at a specific point in training
- **State dict:** a PyTorch dictionary that maps each layer of the model to its learned weights
- **Dropout:** a technique that randomly turns off some neurons during training to prevent memorizing the data

The checkpoint file

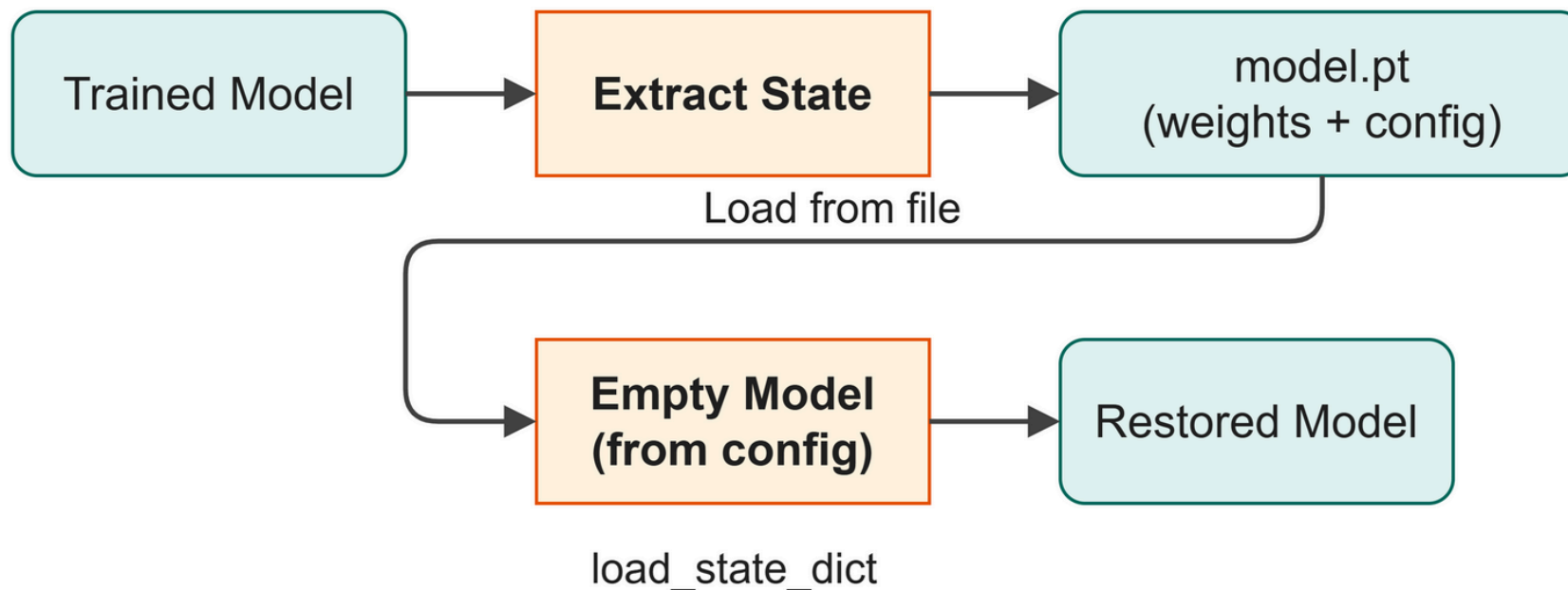


Figure 14.2: The save and load round trip: the model's structure and its learned weights are both stored in the checkpoint.

- A **checkpoint** is the save file: the weights live in a **state dict** (layer name to its weights)
- It also stores the **configuration** (layers, heads), the training step and the validation loss

Save the skeleton and the weights together. Tomorrow, PyTorch will not remember how your model was built.

Restoring the model

Loading a checkpoint happens in three steps:

1. **Load** the saved dictionary from the file on disk
2. **Build** an empty model using the saved configuration
3. **Pour** the saved weights into the empty model

The configuration rebuilds the skeleton. The state dict fills it. Then the model is exactly where training left it.

Training mode and evaluation mode

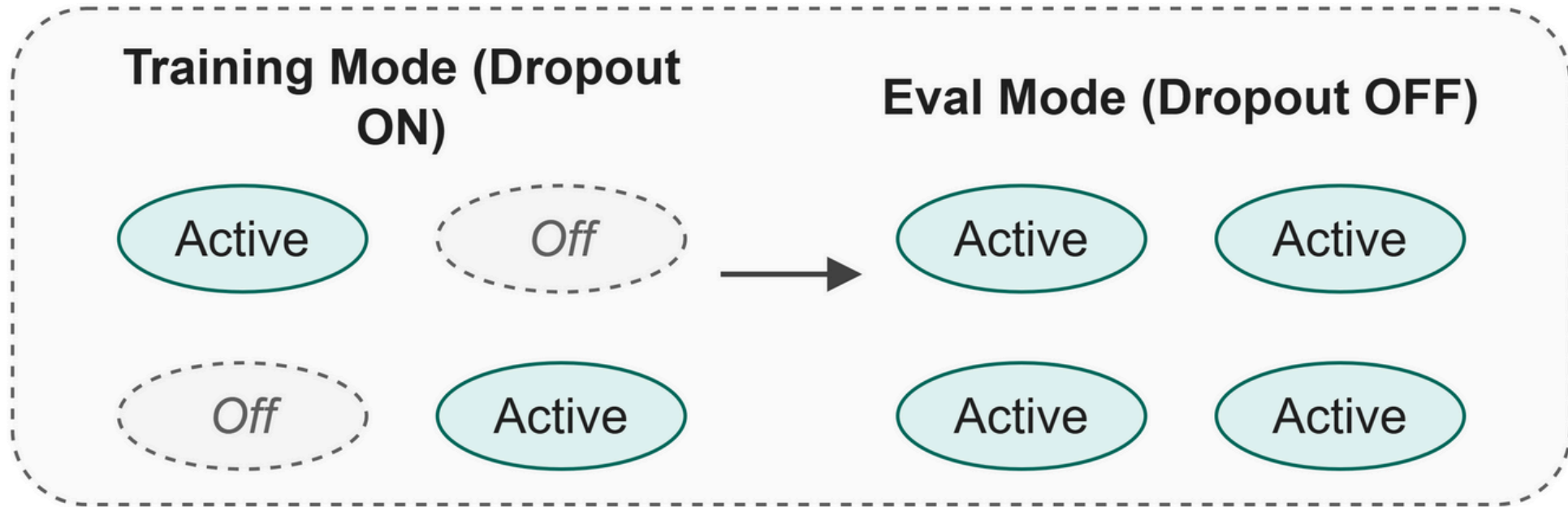


Figure 14.3: In evaluation mode, all neurons are active and the model is ready to generate text deterministically.

- In training, **dropout** (randomly turning off some neurons) stops the model from memorizing the data
- After loading, `model.eval()` turns dropout off: every neuron active, repeatable answers

One line, `model.eval()`, and the model stops practicing and starts performing.

In business: the learned voice in one small file

The house-style assistant learned the company's voice from its archive.

- If the server restarts, you do not want to re-read the entire archive
- A checkpoint saves that learned voice to a **small file**
- Load it in seconds, on any machine

Training is the expensive part. The checkpoint is how you pay for it once.

Code: load the checkpoint

File: `src/ch13_checkpoint.py` (source files are numbered one lower than chapters).

```
checkpoint = torch.load(
    CHECKPOINT_PATH, map_location="cpu", weights_only=False
)
step      = checkpoint["step"]
val_loss  = checkpoint["val_loss"]
cfg       = checkpoint["gpt_cfg"]

model = GPT(cfg)
model.load_state_dict(checkpoint["model_state"])

# Set to evaluation mode (disables dropout, making outputs deterministic)
model.eval()
```

- **Line 1** loads the checkpoint `model.pt` from disk
- **Lines 8 and 9** build an empty `GPT` from the saved configuration and pour in the weights
- **Line 12** switches the model to evaluation mode

How the loading code flows

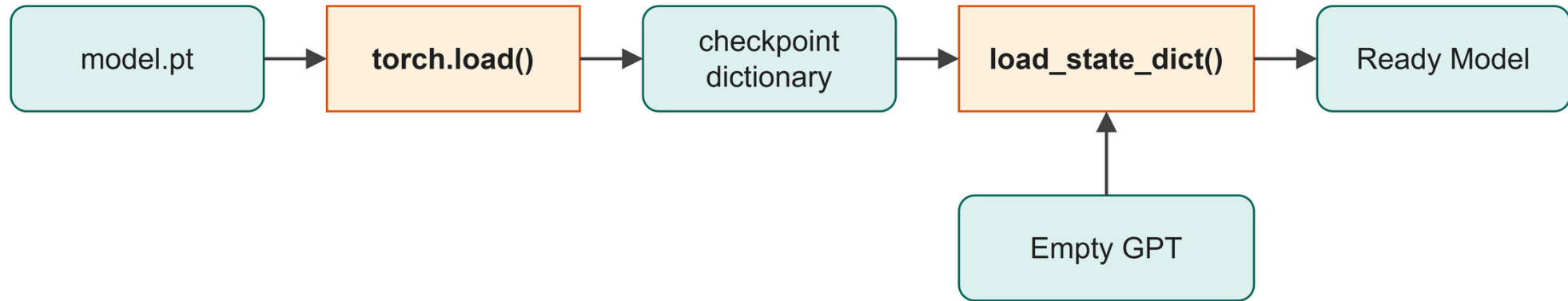


Figure 14.4: How the checkpoint loading code flows: from file to ready model.

- Read the file, take the dictionary apart, build the model, load the weights, switch to eval

Left to right, the figure is the three steps plus `model.eval()` .

Run it

```
$ python src/ch13_checkpoint.py
Loading checkpoint from: checkpoints/model.pt
  Trained for   : 3000 steps
  Val loss      : 1.7228
  Model config  : 4 layers, 128 embd, 4 heads
...
Model rebuilt successfully: 824,832 parameters loaded
...
Input shape : torch.Size([1, 10])
Output shape: torch.Size([1, 10, 65]) (looks good!)
...
Checkpoint file size: 4.18 MB
```

What just happened

- The file remembered **3,000 steps** of training and a validation loss of **1.7228**
- An empty model took back all **824,832** learned parameters
- The file is **4.18 MB**: small enough to share by email

Shape check

Tensor	Shape	What it means
<code>dummy_ids</code>	<code>[1, 10]</code>	1 sequence of 10 token IDs
<code>logits</code>	<code>[1, 10, 65]</code>	65 vocabulary scores for each of the 10 positions

Same shapes as Chapter 10. The restored model is the model you built, with its weights back.

Try it: look inside the checkpoint

Open `src/examples/ch14_inspect_checkpoint.py`. It loads the file and prints what is inside.

```
checkpoint = torch.load(  
    checkpoint_path, map_location="cpu", weights_only=False  
)  
print("Keys in checkpoint:", list(checkpoint.keys()))  
print("Training step:", checkpoint.get("step"))
```

```
$ python src/examples/ch14_inspect_checkpoint.py  
Keys in checkpoint: ['model_state', 'gpt_cfg', 'step', 'val_loss']  
Training step: 3000
```

Four keys: the weights, the configuration, the step, the validation loss. Nothing else.

Discussion

A 4 MB file holds everything the model learned from a million characters of text.

In pairs, 3 minutes:

1. What is inside the file? What is not inside it (the text, the training code, the optimizer)?
2. Your company shares this file with a partner. What can the partner do with it, and what can they not recover from it?

Take two answers from the room.

Recap: today in 4 points

1. A **checkpoint** saves the model's configuration and learned weights to a file.
2. To restore it, build an empty network from the configuration, then `load_state_dict`.
3. Always call `model.eval()` after loading, to turn off dropout.
4. A model with about **825,000** parameters takes about **4 MB** on disk.

Next, we use the restored model for what we built it for: writing new text.

Check your understanding

1. Why do we need to save the model's configuration in the checkpoint along with the weights?
2. What happens if you forget to call `model.eval()` before generating text?
3. What PyTorch function do we use to apply the saved weights to the newly built model?

Next up: Chapter 15, Greedy and Sampling

- The restored model outputs **65 scores** at every position
- Next class: two rules for turning those scores into the next character, and why the safe rule repeats itself

Before next class: run `python src/ch13_checkpoint.py` and confirm it prints **Checkpointing done!**

References

Book and figures

- Luu, T. (2026). *Building an LLM from Zero*, Chapter 14. <https://jackluu.io/book/>
- Figures 14.1 to 14.4 are the book's own diagrams (draw.io sources in the book repository, `diagrams/`).
- Terminal output: `src/ch13_checkpoint.py` and `src/examples/ch14_inspect_checkpoint.py` .

Further reading (from the chapter)

- Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., ... Chintala, S. (2019). *PyTorch: An imperative style, high-performance deep learning library* (arXiv:1912.01703). <https://doi.org/10.48550/arXiv.1912.01703>

Dataset

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