Zero to Hero: Reproducing the GPT-2 (124M) Model

Of course. I understand the goal is to create a rich, readable document that doesn't just summarize but fully captures the live, unedited nature of the lecture—including the speaker's personal voice, the real-time coding and debugging journey, and all the practical asides.

Here is the revised article, re-infused with these crucial elements and expanded with additional mathematical context.

---

1 1. Introduction: The Goal and the Tools

Hi everyone. Today, we're going to be continuing our "Zero to Hero" series, and in particular, today we are going to reproduce the GPT-2 model—the 124 million parameter version of it. When OpenAI released GPT-2 back in 2019, it came with a blog post, a research paper, and, importantly, code on GitHub.

When we talk about reproducing GPT-2, we have to be careful. We're specifically targeting the **124 million parameter model**. The thing to realize is that there's always a "miniseries" of models released. The reason for this is to study **scaling laws**: you can put the model sizes on the x-axis and downstream metrics like translation or question-answering on the y-axis to chart out how performance improves with scale. For GPT-2, this ranged from 124M up to 1.5B parameters. Now, the reason my numbers might disagree with the table in the original paper is because, as the authors later noted in the GitHub repo, **that table is wrong**. The model we're building has **12 layers** and an embedding dimension (or "channels") of **768**. I'll be assuming some familiarity with these terms from my previous video, "Let's Build GPT from scratch."

If we do everything correctly, by the end of this video, we should see a validation loss curve that drops steadily, and hopefully, we'll even beat the original GPT-2 124M model. Five years ago, this was a complex task. Today, you can reproduce this model in roughly an hour for about $10 on a cloud GPU. (For spinning up these boxes, my favorite place is Lambda Labs—they sponsor my development, and I find their service very simple to use).

Unlike many later models, OpenAI did release the weights for GPT-2. However, the GPT-2 paper is a bit vague on training details. So, in addition to it, we'll be referencing the **GPT-3 paper**, which is much more concrete about hyperparameters. The architectures are very similar, so this is a safe and effective strategy. So, let's go.

---

2 2. Establishing the Target: Loading the Original GPT-2

The first thing I'd like to do is actually start at the end: let's load the original, pre-trained GPT-2 124M model and take it for a spin.

The issue is that the original code was written in **TensorFlow**. Today, we'd like to use **PyTorch**—it's friendlier, easier, and I just personally like it a lot more. Thankfully, the Hugging Face `transformers` library has already done the hard work of porting the TensorFlow weights to PyTorch.

Let's load the model's weights (its `state_dict`) and inspect them.

```
from transformers import GPT2LMHeadModel

# Note: "gpt2" loads the 124M model. For the largest, you'd use "gpt2
    -xl".
model_hf = GPT2LMHeadModel.from_pretrained("gpt2")
state_dict = model_hf.state_dict()
```

This reveals the architecture's parameters:

- `transformer.wte.weight`: The **token embedding** table, shape `(50257, 768)`.
- `transformer.wpe.weight`: The **positional embedding** table, shape `(1024, 768)`.

Unlike the original "Attention is All You Need" paper, which used fixed sinusoidal functions for positional encodings, GPT-2 *learns* its positional embeddings from scratch. When we plot them, we can see that the optimization process naturally rediscovers sinusoidal-like patterns, though a bit noisy, which tells you this is a slightly undertrained model.

Using the Hugging Face pipeline, we can generate some text with the prefix "Hello, I'm a language model," and it produces coherent completions. This sets a clear benchmark. Our goal now is to write our own `GPT` class from scratch, load these exact weights into it to prove our implementation is correct, and then train our own version from random initialization.

---

3 3. Building Our Own GPT Model from Scratch

Our GPT-2 model is a **decoder-only** Transformer. We'll only implement the right-hand side of the original Transformer diagram, omitting the encoder stack and cross-attention block. A key architectural change is the **pre-norm formulation**, where Layer Normalization is applied *before* the attention and MLP sub-layers. This creates a clean "gradient superhighway" along the residual path, significantly improving training stability.

Our implementation will consist of three main `nn.Module` classes: `MLP`, `CausalSelfAttention`, and `Block`.

3.1 3.1. The MLP (Multi-Layer Perceptron) Block

This is the "computation" part of the Transformer block. It's a simple two-layer feed-forward network using the **GELU (Gaussian Error Linear Unit)** activation.

$$\text{GELU}(x) = x \cdot \Phi(x), \tag{1}$$

where $\Phi(x)$ is the CDF of the standard normal distribution.

GELU is like a smoother version of ReLU and avoids the "dead neuron" problem. For historical reasons (the exact error function was slow in early TensorFlow), GPT-2 uses a `tanh`-based approximation, which we will also use for perfect compatibility.

3.2 3.2. The Causal Self-Attention & `Block`

This is the "communication" part. We'll write a fused version where all attention heads are computed in parallel by treating the "number of heads" as a temporary batch dimension, which is more efficient. A single `Block` then combines these two components with residual connections and layer normalization.

---

4 4. The Training Journey Begins

4.1 4.1. The First Run and a Classic Bug

Now for the real work. Let's initialize our model with random weights. As expected, the generated text is complete gibberish.

To start debugging, we'll use the **Tiny Shakespeare dataset** and set up a simple `DataLoader`. We'll then run a single training step. And... we get an error.

> Expected all tensors to be on the same device but found at least two devices Cuda zero and CPU

This is a classic and powerful lesson. I moved my model and input data to the GPU with `.to(device)`, but inside my `forward` pass, I created the positional index tensor (`pos`) with `torch.arange`. By default, this creates the tensor on the **CPU**. The model then tries to add the token embeddings (on the GPU) to the positional embeddings (looked up using a CPU tensor), and PyTorch throws an error.

The fix is to ensure `pos` is created on the same device as the input `idx`. However, a common mistake is to write `pos.to(device)`, which doesn't work because it's not an in-place operation. The correct fix is `pos = torch.arange(T, device=device)`.

4.2 4.2. Overfitting a Single Batch

With the bug fixed, the next vital debugging step is to ensure the model can **overfit a single batch**. We feed the same small batch of data to the model over and over again.

1. **Calculate the Loss**: We use **Cross-Entropy Loss**, which for a single prediction is the negative log-probability of the correct next token: $\text{Loss} = -\log(P_c)$.
2. **Sanity Check**: Our initial loss is $\sim$11. The theoretical loss for a random guess is $-\log(1/50257) \approx 10.82$. This is very close—a good sign!
3. **Optimization**: We set up an **AdamW optimizer** and run a simple training loop.

The loss plummets towards zero. This is exactly what we want to see. It confirms that our model can learn and that all the optimization machinery is wired up correctly.

---

5 5. Scaling Up: Performance Optimizations

Okay, so our simple loop is correct, but it's slow. Let's look at the terminal. I'll time a few iterations. It's taking about **1000ms per step**. To train on billions of tokens, we need to speed this up. I'll pull up `nvidia-smi` to monitor the A100 GPU. We can see it's using GPU0 and about 35GB of VRAM.

1. **Mixed Precision with `tf32`**: A single line, `torch.set_float32_matmul_precision('high')`, tells PyTorch to use TensorFloat-32 on the A100's Tensor Cores. This gives us a ~**3x speedup**, down to ~330ms/step. It's a free performance boost, limited only by memory bandwidth.

2. `bfloat16`: To reduce memory bandwidth, we use `torch.amp.autocast` to run activations in `bfloat16`. This gets us down to ~300ms/step.

3. `torch.compile`: This JIT compiler fuses operations into single, optimized kernels, reducing GPU memory round-trips. This is a massive win, giving another ~**2.3x speedup** and bringing us to **129ms/step**.

4. **FlashAttention**: This is a custom kernel that re-implements self-attention with an algorithmic rewrite that avoids ever creating the huge $T \times T$ attention matrix in memory. By replacing our naive attention with PyTorch's `F.scaled_dot_product_attention`, we get another ~**27% speedup**, now down to ~**95ms/step**.

5. **The "Nice Numbers" Optimization**: This is my favorite—simultaneously the dumbest and most brilliant optimization. Our vocabulary size is **50,257**, an "ugly" prime number. GPU kernels love sizes that are powers of two. By padding the vocabulary to the next "nice" number, **50,304** (divisible by 128), we get another **4% speedup**, even though we're doing *more* work. We're now at ~**90ms/step**.

In total, we went from 1000ms to 90ms—an **11x improvement**.

---

6 6. Algorithmic Refinements: Training Like GPT-3

To get the best results, we turn to the GPT-3 paper for the right hyperparameters.

- **Learning Rate Scheduler**: The paper specifies a **cosine decay schedule with warmup**. PyTorch has built-in schedulers, but I don't love them—they feel a bit "inscrutable." It's just five lines of code to write our own, so we have full control and understanding.

- **Weight Tying & Custom Initialization**: We tie the input embedding and output projection weights, which saves ~30% of the model's parameters. We also use a specific initialization scheme from the GPT-2 code, including a crucial **residual scaling** factor of $1/\sqrt{2 \cdot N_{\text{layers}}}$ to control variance growth in the residual stream.

- **Gradient Accumulation**: To simulate the GPT-3 paper's massive batch size of 500k tokens on our hardware, we use gradient accumulation. We process several smaller "micro-batches" and let their gradients add up before performing a single optimizer step.

---

7 7. The Big Run: Data, DDP, and the `torch.compile` Bug

We're ready for a serious training run on the **FineWeb-Edu 10B** dataset. To use all 8 GPUs, we use **Distributed Data Parallel (DDP)**, where each GPU trains on a different shard of data and gradients are averaged after each step.

Now, for a bit of real-world pain. When I add the periodic evaluation for validation loss, HellaSwag, and the sampling code, everything works. But when I re-enable `torch.compile`, it breaks.

[scary error message from PyTorch]

I have no idea how to resolve this right now. So, for the final run, I'm just going to have to disable `torch.compile`. It will be slower, but it will work. Hopefully, by the time you see the code I release in the `build-nanogpt` repository, this will be fixed.

---

8 8. Results and Conclusion

After an overnight run processing 40 billion tokens, we can plot the results.

- **Validation Loss**: Our model's validation loss on FineWeb-Edu quickly surpasses the original GPT-2 checkpoint.
- **HellaSwag Accuracy**: Our model starts at 25% (random chance) and steadily improves, surpassing the original GPT-2 and nearly reaching the performance of the GPT-3 124M model.

This is a fantastic result. We've shown that with modern tools and higher-quality data, we can match or exceed the original GPT-2's performance with significantly less data (40B tokens vs. ∼400B). The weird periodicity in the loss curve is likely a data shuffling issue in this simple implementation, which can be fixed with a better data loader.

This entire process, from scratch, is available with full git commit history in the `build-nanogpt` repo. If you want to go even faster and deeper, check out my `llm.c` project, which re-implements this in pure C/CUDA. For questions, check out the “Zero to Hero” Discord.

I hope you enjoyed the video, and I’ll see you later!