Chapter 11: Scaling Laws and Emergent Abilities

Learning Objectives

After reading this chapter, the reader should be able to:

1. State and interpret the Kaplan and Chinchilla scaling laws, derive the compute-optimal relationship between model size and dataset size, and use these laws to estimate the training loss for a given compute budget.
2. Define emergent abilities in large language models, provide concrete examples (e.g., arithmetic, chain-of-thought reasoning), and critically evaluate the debate over whether emergence is a genuine phase transition or a measurement artifact.
3. Explain the Mixture-of-Experts (MoE) architecture, including sparse gating, top-k routing, and load balancing, and articulate why MoE enables scaling parameter count without proportionally increasing compute.
4. Describe key techniques for efficient large-scale training – data parallelism, tensor parallelism, pipeline parallelism, mixed-precision training, and gradient checkpointing – and reason about their trade-offs.

---

175 billion parameters. That was the number OpenAI reported for GPT-3 in 2020 – a model roughly a thousand times larger than its predecessor, GPT-2, released just one year earlier. The jump was not an incremental improvement but a qualitative leap: GPT-3 could write essays, translate between languages it had not been explicitly trained on, and perform arithmetic with no task-specific training data. The natural reaction was astonishment. The engineering reaction was a question: *how much of that improvement was predictable?*

The answer, it turns out, is almost all of it. In the years surrounding GPT-3's release, a series of careful empirical studies revealed that the relationship between model scale and prediction quality is not chaotic, not architecture-dependent in any deep way, and not even particularly complicated. Pre-training loss – the cross-entropy measure that has been our central quantity since Chapter 1 – follows power laws in model size, dataset size, and compute budget. Plot loss against any of these quantities on logarithmic axes and we get a straight line. The loss goes down. Reliably, predictably, and – for reasons nobody fully understands – as a power law.

This chapter is about what happens when we take prediction seriously as an engineering discipline and ask: how much better does the next-word prediction get as we pour in more resources? In Chapter 9, we studied the pre-training objectives whose loss is now the dependent variable. In Chapter 10, we examined how tokenizers and data curation pipelines produce the corpora that models consume. With that infrastructure in place, we can now address the most consequential engineering question in modern NLP: how should we allocate a fixed budget of compute between model size and training data? We begin with scaling laws – the empirical regularities that let researchers predict a model's performance before training it (Section 11.1). We then examine emergent abilities – capabilities that appear abruptly at scale and refuse to be predicted by smooth extrapolation (Section 11.2). Section 11.3 introduces Mixture-of-Experts architectures that decouple parameter count from compute cost. Section 11.4 covers the distributed training techniques that make scale physically possible. We close with the compute frontier: historical trends, costs, the open-weight movement, and what scaling means for researchers with modest budgets (Section 11.5).

11.1 Scaling Laws

If we double the training compute, how much does the loss decrease?

Before 2020, the honest answer to this question was "nobody knows – train it and see." Researchers selected model sizes through a combination of hardware constraints, intuition, and trial-and-error. A team with eight GPUs might train a model that fits on eight GPUs; a team with a hundred GPUs might train something larger. There was no principled framework for deciding how large a model should be, how much data it needed, or what loss to expect at the end of training. The discovery that changed this was startlingly simple: pre-training loss follows a power law in model size, dataset size, and compute budget. These relationships – quantified by Kaplan et al. [Kaplan et al., 2020] and refined by Hoffmann et al. [Hoffmann et al., 2022] – transformed language model development from an expensive guessing game into something closer to engineering. We can now estimate, before spending a single GPU-hour, what loss a model of a given size will achieve on a given amount of data.

11.1.1 Power Laws in Language Modeling

A power law is a relationship of the form $y = ax^{-b}$ where a and b are constants. The defining property of power laws is scale invariance: on a log-log plot, a power law appears as a straight line with slope $-b$. Each multiplicative increase in x produces the same multiplicative change in y, regardless of the starting point. Doubling x from 10 to 20 has the same proportional effect on y as doubling it from 10 billion to 20 billion. Power laws arise throughout the natural sciences – from earthquake magnitudes to city populations to species diversity – and they typically signal that the underlying system has no characteristic scale, meaning that the same dynamics operate across many orders of magnitude. The appearance of power laws in language modeling is, in this light, both remarkable and puzzling. It suggests that whatever a neural language model is doing as it converts parameters and data into prediction quality, the process is fundamentally scale-free: there is no special model size at which the dynamics change, no threshold beyond which more parameters stop helping, no "saturation point" visible in the data. Each order of magnitude behaves like every other.

The practical consequence of power laws is that they are *predictable*. If we train a handful of small models (say, 10 million to 1 billion parameters) and measure their loss, we can fit a straight line on log-log axes and extrapolate to predict the loss of a 100-billion-parameter model that we have not yet trained. This is exactly the methodology that scaling law research employs: train many small models cheaply, fit the power law, and use it to guide investment in the expensive large model. The analogy to Moore's Law is instructive. Moore's Law – the observation that transistor density doubles approximately every two years – is an empirical regularity, not a physical law. Nobody derived it from first principles, and it held for decades before finally slowing. Scaling laws for language models play the same role: they are empirical regularities that allow engineers to predict future performance from current trends, even without fully understanding the underlying mechanism. And just as Moore's Law guided decades of semiconductor investment, scaling laws now guide infrastructure decisions at frontier AI laboratories that involve hundreds of millions of dollars.

Why do power laws arise in language modeling? The honest answer is that nobody knows for certain. Several hypotheses have been proposed. One connects the power-law exponent to the intrinsic dimensionality of the data manifold: natural language, though embedded in a very high-dimensional token space, may lie on a lower-dimensional manifold, and the exponent reflects how efficiently

additional parameters cover this manifold. Another hypothesis invokes random matrix theory and the spectral properties of the weight matrices in deep networks. A third, more pragmatic view holds that the power law is simply the best smooth fit to the data and that we should not read too much into it – the important thing is that it works as a predictive tool. For our purposes as practitioners, the mechanistic explanation matters less than the empirical fact: the power law is real, it is robust across architectures and datasets, and it enables planning.

11.1.2 Kaplan et al.: Scaling Laws for Neural Language Models

In January 2020, a team at OpenAI led by Jared Kaplan published what would become one of the most influential empirical papers in modern deep learning [Kaplan et al., 2020]. They trained over a hundred Transformer language models ranging from 768 parameters to 1.5 billion parameters on the WebText2 corpus, systematically varying model size N (non-embedding parameters), dataset size D (number of tokens), and training compute C (floating-point operations). For each variable, they fit a power law to the resulting cross-entropy loss. The results were strikingly clean. Loss as a function of model size alone follows:

$$L(N) = \left(\frac{N_c}{N}\right)^{\alpha_N}, \quad L(D) = \left(\frac{D_c}{D}\right)^{\alpha_D} \tag{11.1}$$

where $N_c \approx 8.8 \times 10^{13}$ and $D_c \approx 5.4 \times 10^{13}$ are constants representing the scale at which loss reaches 1 nat, and the exponents $\alpha_N \approx 0.076$ and $\alpha_D \approx 0.095$ measure how steeply loss decreases with each variable. On log-log axes, these are straight lines with slopes -0.076 and -0.095 respectively. The exponent $\alpha_N = 0.076$ means that each 10-fold increase in model parameters reduces loss by a factor of $10^{0.076} \approx 1.19$, or roughly 19%. This is a modest per-decade improvement, but it compounds: a 1,000-fold increase (three decades) reduces loss by a factor of approximately $10^{0.228} \approx 1.69$, a substantial gain.

Kaplan et al. also fit a joint scaling law that describes loss as a function of both model size and data simultaneously:

$$L(N, D) = \left[\left(\frac{N_c}{N}\right)^{\alpha_N/\alpha_S} + \left(\frac{D_c}{D}\right)^{\alpha_D/\alpha_S}\right]^{\alpha_S} \tag{11.2}$$

where α_S is a combined scaling exponent. This joint law captures a crucial insight: the irreducible loss at a given scale is dominated by whichever bottleneck is tighter – too few parameters or too little data. If N is very large but D is small, the second term dominates and loss is limited by data starvation. If D is very large but N is small, the first term dominates and the model lacks the capacity to exploit the available data. The optimal strategy at any compute budget balances these two terms. The practical recommendation from Kaplan et al. was striking and, as we will see, ultimately wrong: they argued that model size matters more than data. Their analysis suggested that when compute increases, most of the additional budget should go to a larger model rather than more training data. This recommendation directly influenced the design of GPT-3, which had 175 billion parameters but was trained on only 300 billion tokens – a ratio of roughly 1.7 tokens per parameter. The model was enormous but, by later standards, starved of data.

```
import numpy as np
import matplotlib.pyplot as plt
```

Figure 11.1: Neural Scaling Laws

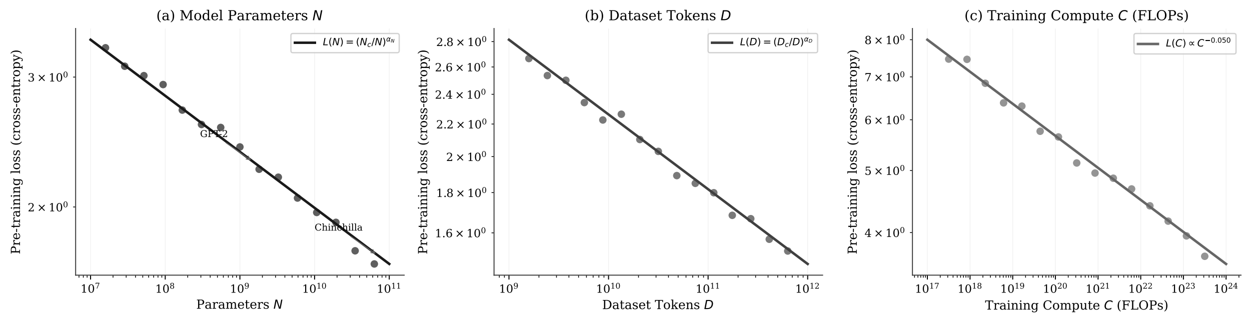

Figure 1: Figure 11.1 – Scaling Law Curves

```
# Kaplan scaling law: L(N) = (N_c / N) ^ alpha_N
N_c, alpha_N = 8.8e13, 0.076
N_vals = np.logspace(7, 11, 50)  # 10M to 100B params
loss_N = (N_c / N_vals) ** alpha_N

# Approximate data points from published models
model_N = np.array([1.5e9, 7e10, 1.75e11])  # GPT-2, Chinchilla, GPT-3
model_L = (N_c / model_N) ** alpha_N

fig, axes = plt.subplots(1, 3, figsize=(14, 4))
axes[0].loglog(N_vals, loss_N, 'b-', lw=2)
axes[0].scatter(model_N, model_L, c='red', zorder=5, s=60)
for n, l, name in zip(model_N, model_L, ['GPT-2','Chinchilla','GPT-3']):
    axes[0].annotate(name, (n, l), fontsize=8, ha='left')
axes[0].set(xlabel='Parameters N', ylabel='Loss L', title='L(N)')
# Panels (b) and (c) follow identical structure for D and C
for ax in axes: ax.grid(True, alpha=0.3)
print(f"Loss at 1.5B params (GPT-2): {model_L[0]:.4f}, at 175B params (GPT-3): {model_L[2]:.4f
plt.tight_layout(); plt.savefig('fig_11_1_scaling_curves.pdf')
```

Code 11.1 – Reproducing Kaplan Scaling Curves. This script plots the power-law fit $L(N) = (N_c/N)^{\alpha_N}$ on log-log axes and annotates known models. The full notebook extends this to all three panels (parameters, tokens, compute) using published data points from Kaplan et al. [2020]. Run `pip install numpy matplotlib` if needed.

11.1.3 Chinchilla: Compute-Optimal Training

Two years after Kaplan et al. established the first scaling laws, a team at DeepMind published a paper that fundamentally changed how the field thinks about the model-size/data-size trade-off [Hoffmann et al., 2022]. The paper, which introduced the 70-billion-parameter model called Chinchilla, answered a question that Kaplan et al. had gotten wrong: *given a fixed compute budget, what is the optimal balance between model size and training data?*

The experimental methodology was formidable. Hoffmann et al. trained over 400 language models ranging from 70 million to 16 billion parameters, each on varying amounts of data, and measured the

final loss for every (model size, data size) combination. They then fit a parametric model relating loss to N and D jointly – similar in spirit to Equation 11.2 but estimated from a much larger grid of experiments – and solved analytically for the values of N and D that minimize loss for a given compute budget C, subject to the constraint that compute is approximately $C \approx 6ND$ (each token requires roughly $6N$ floating-point operations in a forward-backward pass). The result was clean and surprising:

$$N_{\text{opt}} \propto C^{0.5}, \quad D_{\text{opt}} \propto C^{0.5} \tag{11.3}$$

Both optimal model size and optimal dataset size scale as the square root of compute. This means they should be scaled in equal proportion: if compute doubles, both N and D should increase by a factor of $\sqrt{2} \approx 1.41$. The ratio $D_{\text{opt}}/N_{\text{opt}}$ is approximately constant at about 20 tokens per parameter – the *Chinchilla ratio*. This was a direct contradiction of Kaplan et al., who had recommended prioritizing model size. GPT-3, with 175 billion parameters trained on 300 billion tokens, had a ratio of just 1.7 tokens per parameter – undertrained by roughly a factor of 10 according to Chinchilla. The analogy we find most clarifying for students is this: Kaplan's recipe was like building the tallest possible skyscraper on a fixed budget by making it extremely narrow – maximize height (parameters) at the expense of floor space (data). Chinchilla showed that a shorter, wider building is structurally sounder and more cost-effective. Or more directly: GPT-3 was a student who skimmed the textbook once. The optimal strategy is to match study time to ability.

To prove the point, Hoffmann et al. trained Chinchilla: 70 billion parameters (4x smaller than GPT-3's predecessor Gopher at 280B) on 1.4 trillion tokens (4.7x more data than Gopher's 300B). Chinchilla matched or exceeded Gopher on every benchmark, despite using the same compute budget. The implication was stark: the industry had been building over-parameterized, under-trained models for years. Every model in the GPT-3 family, and many of its contemporaries, was far from compute-optimal.

Sidebar: Scaling Laws Beyond Language – Vision, Code, and Multimodal. The power laws that Kaplan et al. discovered in language modeling are not unique to text. Zhai et al. [2022] demonstrated similar power-law scaling in vision models (ViT), where classification loss decreases as a power law in both model size and dataset size with exponents remarkably close to those found in language. Henighan et al. [2020] showed that language models trained on code follow scaling laws with comparable exponents, and multimodal models (CLIP, Flamingo) exhibit power-law behavior across both the vision and language components. The universality of scaling laws across modalities suggests that the phenomenon is not specific to natural language or Transformer architectures but reflects something deeper about how overparameterized neural networks extract patterns from structured data. This generality strengthens the case for using scaling laws as engineering tools: if they hold across text, images, code, and their combinations, they are likely to hold for future modalities as well. The exponents differ across domains – vision models show steeper scaling with data than language models – but the functional form is consistently a power law.

Sidebar: The Chinchilla Controversy – Are We Still Under-Training? When Hoffmann et al. published the Chinchilla scaling laws in 2022, the conclusion seemed definitive: the optimal ratio is approximately 20 tokens per parameter. But subsequent developments complicated the picture considerably. Meta's LLaMA [Touvron et al., 2023] deliberately trained a 7-billion-parameter model on 1 trillion tokens – 143 tokens

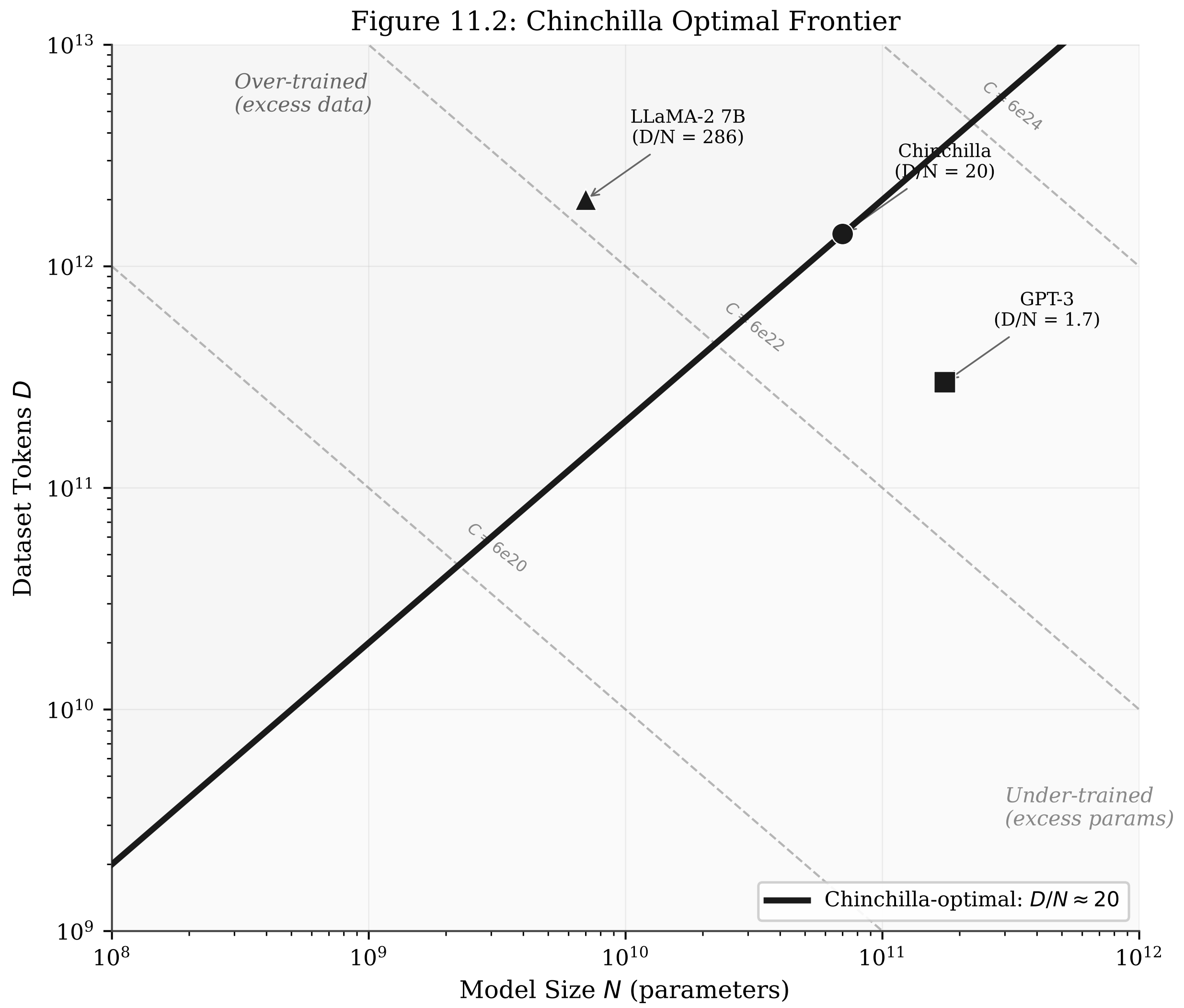

Figure 2: Figure 11.2 – Chinchilla Optimal Frontier

per parameter, roughly 7 times the Chinchilla ratio. The reasoning was that Chinchilla optimizes for *training* compute, not *inference* compute. A smaller model that is trained longer is cheaper to deploy: it uses less memory, runs faster at inference, and is easier to fine-tune and serve to millions of users. If inference costs dominate the lifetime cost of a model – as they do for any model deployed at scale – then over-training a smaller model is economically rational even if it wastes training compute. Microsoft's Phi models pushed even further, demonstrating that carefully curated data can make 1–3 billion parameter models competitive with much larger ones on reasoning benchmarks. The Chinchilla ratio minimizes training cost for a given loss target; it does not minimize total cost of ownership. The optimal recipe depends on what we optimize for.

11.1.4 Using Scaling Laws for Planning

10^{23} floating-point operations. That is approximately the compute budget that produced GPT-3, and it is a useful round number for a planning exercise. Scaling laws transform this abstract quantity into concrete engineering decisions. Given a budget of C FLOPs, we can use the Chinchilla relationship $C \approx 6ND$ together with the optimal ratio $D/N \approx 20$ to solve for the optimal model size and dataset: $N_{\text{opt}} = \sqrt{C/120}$ and $D_{\text{opt}} = 20\, N_{\text{opt}}$. For $C = 10^{23}$, this gives $N_{\text{opt}} \approx 2.9 \times 10^{10}$ (roughly 29 billion parameters) and $D_{\text{opt}} \approx 5.8 \times 10^{11}$ (roughly 580 billion tokens). We can then plug these values back into the scaling law to predict the training loss, estimate the number of training steps, and calculate the wall-clock time on a given cluster. The power of this methodology lies in its ability to prevent costly mistakes. Without scaling laws, a team might train a 100-billion-parameter model on the same 10^{23} FLOPs budget – yielding a model that is too large and under-trained, with worse loss than the 29-billion-parameter compute-optimal model despite costing exactly the same to train.

Frontier laboratories use scaling laws in precisely this way. Before committing to a multi-month, multi-million-dollar training run, they train a series of small proxy models (typically ranging from 100 million to a few billion parameters), fit the power law, and extrapolate to the target scale. If the extrapolated loss at the target scale meets the performance threshold required for the intended application, they proceed. If not, they know in advance that the budget is insufficient and can either secure more compute or adjust their ambitions. The methodology is analogous to wind-tunnel testing in aeronautical engineering: we do not build a full-size aircraft to discover its aerodynamic properties. We build small models, measure their behavior, and use well-validated scaling relationships to predict the behavior of the full-size system. The crucial caveat is that scaling laws predict pre-training loss – the cross-entropy on next-token prediction. They do not directly predict downstream task performance. A model with 10% lower pre-training loss will generally perform better on language understanding benchmarks, but the relationship between pre-training loss and, say, medical question-answering accuracy is noisy and task-dependent. As we will see in Section 11.2, some tasks improve smoothly with scale while others show abrupt jumps. Scaling laws tell us about the quality of the underlying representation; they do not guarantee performance on any specific application.

```
import numpy as np

def chinchilla_optimal(C_flops, ratio=20):
    """Given compute budget C (FLOPs), return optimal N and D."""
    # From C = 6*N*D and D = ratio*N: N = sqrt(C / (6*ratio))
    N_opt = (C_flops / (6 * ratio)) ** 0.5
```

```python
    D_opt = ratio * N_opt
    return int(N_opt), int(D_opt)

budgets = [1e19, 1e21, 1e23, 1e24, 1e25]
print(f"{'Compute (FLOPs)':>18}  {'N_opt':>14}  {'D_opt (tokens)':>16}  {'D/N':>6}")
print("-" * 60)
for C in budgets:
    N, D = chinchilla_optimal(C)
    print(f"{C:18.0e}  {N:14,}  {D:16,}  {D/N:6.1f}")
# D/N stays ~20 across all budgets -- the Chinchilla invariant
```

Code 11.2 – Compute-Optimal Allocation Calculator. For any compute budget, this script calculates the Chinchilla-optimal model size and dataset size using the approximation $C \approx 6ND$ and $D/N \approx 20$. The constant ratio across budgets illustrates the core Chinchilla finding: data and parameters should scale equally.

Section 11.1 establishes the quantitative foundation: loss follows power laws that are predictable, actionable, and robust. Kaplan et al. discovered the laws; Chinchilla corrected the recipe. Armed with these relationships, we can plan training runs, allocate budgets, and predict outcomes. But scaling laws describe the *smooth* behavior of pre-training loss. In the next section, we encounter something the smooth curves did not predict: capabilities that appear suddenly, as if a switch had been flipped.

11.2 Emergent Abilities

At what point does a language model stop being a text predictor and start being a reasoner?

Scaling laws promise smooth, predictable improvement. Cross-entropy loss decreases as a clean power law in model size, with no breaks, no plateaus, no surprises. If that were the whole story, the science of scale would be straightforward: pour in more resources, get proportionally better predictions, and plan accordingly. But in 2022, a team at Google Brain documented something the smooth curves had not warned us about [Wei et al., 2022b]. They found that certain capabilities – specific, measurable tasks – are essentially absent in models below a threshold size and then appear abruptly once that threshold is crossed. Multi-digit arithmetic, chain-of-thought reasoning, word unscrambling, instruction following: for each of these tasks, performance hovers near random across a wide range of model sizes and then jumps sharply upward in a narrow band. Wei et al. called these *emergent abilities*, borrowing a term from physics and complexity science where emergence refers to macroscopic behaviors that arise from microscopic interactions but cannot be predicted from them.

11.2.1 What Are Emergent Abilities?

An ability is emergent, in the precise sense defined by Wei et al. [2022b], if it is not present in smaller models and cannot be predicted by extrapolating performance from smaller scales. This definition is deliberately restrictive: it excludes capabilities that improve gradually with scale (such as general text fluency or basic factual recall) and focuses on capabilities that exhibit a discontinuous jump. The mechanism underlying the model – next-token prediction – does not change at any scale. A 100-million-parameter Transformer and a 100-billion-parameter Transformer are architecturally

identical in kind, differing only in the number of layers, attention heads, and hidden dimensions. Both are trained on the same objective: minimize cross-entropy loss on next-token prediction. The difference is that above a certain parameter threshold, the model appears to compose individually learned sub-skills into novel behaviors that none of the sub-skills individually exhibit. A model that has separately learned to parse digit strings, to track positional information, and to apply simple arithmetic operations can, at sufficient scale, combine these sub-skills to perform multi-step addition – a task that requires all of them simultaneously and that smaller models, possessing each sub-skill in weaker form, cannot accomplish.

The phase-transition analogy from physics is illuminating but imperfect. When water freezes, individual molecules do not change their behavior; the macroscopic properties (solid vs. liquid) emerge from collective interactions that cross a critical threshold. Similarly, individual parameter updates in a language model do not "learn arithmetic" – but the collective capacity of a sufficiently large model enables it to represent and execute the compositions required for arithmetic. The analogy breaks down because physical phase transitions are sharp and universal (water always freezes at $0\,^{\circ}\mathrm{C}$ at standard pressure), whereas emergent abilities in language models appear at different thresholds for different tasks, vary across model families and training data, and – as we will see in Section 11.2.3 – may partly be artifacts of the evaluation metric rather than the model itself. Nevertheless, the phenomenology is striking: a capability goes from effectively zero to meaningfully above chance across less than one order of magnitude of model size.

11.2.2 Concrete Examples Across Benchmarks

Wei et al. [2022b] documented emergent abilities across dozens of tasks from the BIG-bench benchmark suite, and the pattern is remarkably consistent. We focus on four illustrative examples that span different cognitive demands. Multi-step arithmetic – adding or multiplying multi-digit numbers – shows near-random accuracy (below 5%) for all models smaller than roughly 10 billion parameters. Between 10 billion and 100 billion parameters, accuracy rises steeply, reaching 40–60% for the largest models tested. The jump is dramatic: a model 10 times larger than the threshold performs radically better on a task where a model half the threshold size is essentially guessing. Chain-of-thought reasoning, where the model is prompted to show intermediate reasoning steps before giving a final answer, is similarly absent below approximately 60 billion parameters. Smaller models either ignore the chain-of-thought prompt entirely or produce reasoning chains that do not logically connect to the conclusion. Above 60 billion parameters, the same prompting technique produces coherent multi-step reasoning that substantially improves accuracy on tasks like grade-school math word problems. Instruction following – the ability to obey natural-language instructions specifying the output format, constraints, or style – is unreliable below roughly 10 billion parameters and becomes consistently functional above 50 billion. Code generation, as measured by pass rates on programming challenges, shows meaningful capability only above approximately 10 billion parameters.

Several patterns emerge from this survey. First, there is no single "emergence threshold" – different abilities emerge at different scales, ranging from 10 billion to over 100 billion parameters depending on the task. Second, the emergence threshold depends not only on model size but also on training data quality, tokenization, and the specific prompting strategy used. The same model can show emergent arithmetic with chain-of-thought prompting but not with direct prompting, suggesting that emergence reflects the interaction between model capacity and the elicitation method. Third, the emergence pattern is robust across model families: it appears in GPT-3, PaLM, Chinchilla, and LLaMA, despite differences in architecture, data, and training procedure. This robustness suggests

that the phenomenon is fundamental to the scaling regime of large language models, not an artifact of any particular implementation.

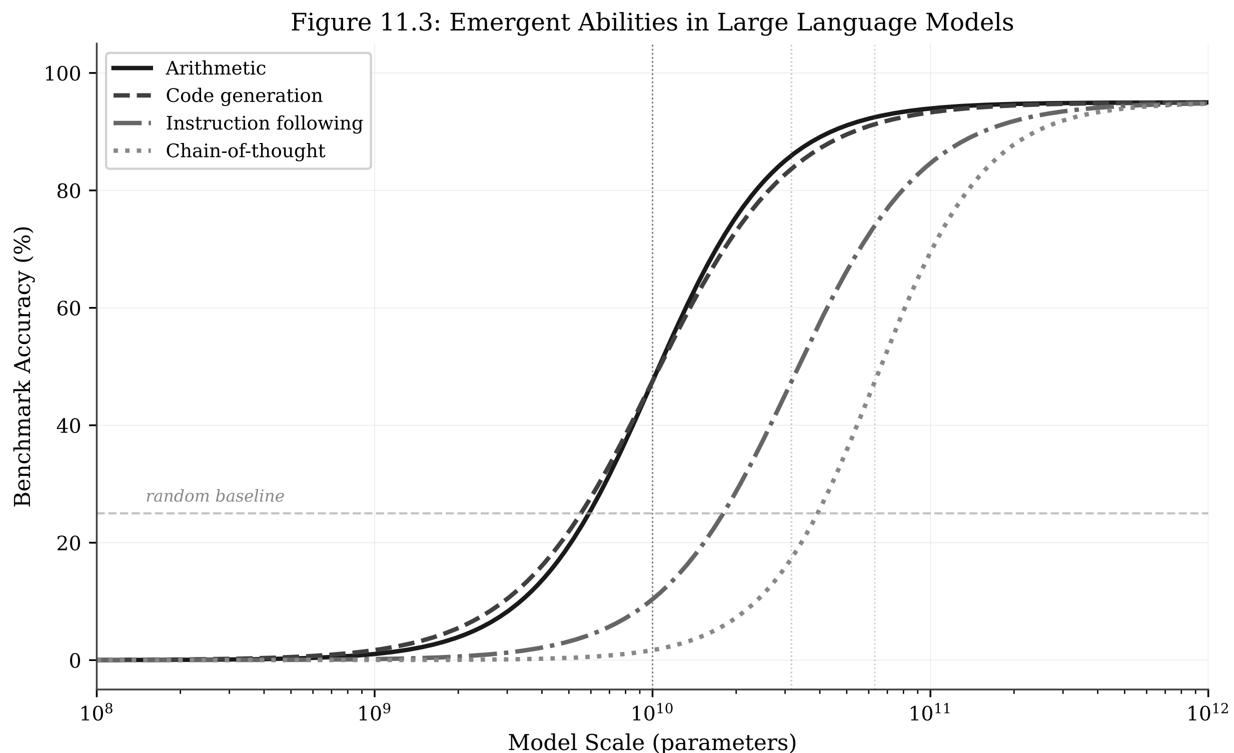

Figure 3: Figure 11.3 – Emergent Abilities

11.2.3 The Phase Transition Debate

Nobody fully understands why emergent abilities appear. But a provocative paper by Schaeffer, Miranda, and Koyejo [Schaeffer et al., 2023] challenged the entire premise by arguing that emergent abilities are not genuine phase transitions in model capability but artifacts of the evaluation metrics used to measure them. The argument is elegant and deserves careful consideration.

Consider multi-digit addition as a concrete case. Suppose a model's probability of getting any individual digit correct improves smoothly with scale – say, from 10% at 1 billion parameters to 99% at 100 billion parameters, following a gradual sigmoid curve. If the evaluation metric is *exact-match accuracy* (the entire multi-digit answer must be correct), then for a 4-digit addition problem, accuracy at 1 billion parameters is approximately $0.1^4 = 0.0001$ and at 100 billion parameters is approximately $0.99^4 = 0.96$. The underlying per-digit improvement is completely smooth, but the exact-match metric transforms it into what looks like an abrupt jump from zero to near-perfect. The "emergence" is not in the model; it is in the metric. Schaeffer et al. reanalyzed many of the BIG-bench tasks that Wei et al. had identified as exhibiting emergence and found that switching to continuous metrics – token-level log-likelihood, per-digit accuracy, Brier scores – made the performance curves smooth in nearly every case. The phase transition disappeared when the thermometer changed.

The debate is genuinely unsettled, and we present both sides because students will encounter advocates of each position. The Schaeffer argument is strongest for tasks that involve composing multiple independent sub-steps, where exact-match is a natural but misleading metric. For these

tasks, the "emergence" is a mathematical consequence of thresholding a smooth signal and carries no implications about qualitative changes in model capability. The counter-argument is that some capabilities are harder to explain as metric artifacts. Chain-of-thought reasoning, for instance, involves generating coherent multi-step reasoning chains – a behavior that does not easily decompose into independent sub-steps that can be evaluated with continuous metrics. A model that produces incoherent reasoning steps is qualitatively different from one that produces coherent steps, and no change of metric removes this qualitative distinction. The practical implications of the debate are significant regardless of which side proves correct. Even if emergence is "merely" a metric artifact, the threshold effects are real for practitioners: a model that passes 5% of math tests is useless for math tutoring, while one that passes 95% is valuable, and the transition between these regimes occurs over a narrow range of model sizes. Whether the underlying mechanism is smooth or discontinuous, the practical consequence is the same: there are scale thresholds below which certain applications are infeasible and above which they become viable. For safety and predictability, the distinction matters more: if emergence is genuinely discontinuous, then scaling up a model could unlock unexpected and potentially dangerous capabilities with no warning from smaller-scale experiments. If emergence is a metric artifact, then continuous monitoring of per-component performance should provide advance warning of approaching capability thresholds.

Section 11.2 reveals the limits of the smooth scaling story. Whether emergent abilities are genuine phase transitions or measurement artifacts, they demonstrate that pre-training loss is not the only quantity that matters – task-level performance can behave very differently from the smooth power laws of Section 11.1. We now turn to an architectural innovation that addresses a different challenge of scale: how to make models larger without proportionally increasing the cost of every prediction.

11.3 Mixture of Experts

Can we build a model with a trillion parameters but only pay for a fraction of them on each prediction?

The scaling laws of Section 11.1 present a straightforward if expensive path to better language models: make them bigger, train them on more data, and the loss decreases as a predictable power law. But "bigger" in a dense Transformer means that every token processed must pass through every parameter. A 175-billion-parameter model requires approximately $6 \times 175 \times 10^9 \approx 10^{12}$ floating-point operations per token – for every token, in every forward pass, at every layer, every parameter participates. This creates a direct proportionality between parameter count and compute cost that sets a hard ceiling on how large a model can practically be. Mixture of Experts (MoE) breaks this proportionality. The core idea is hospital-like in its logic: a large hospital employs hundreds of specialists, but any individual patient sees only the two or three relevant to their condition. The hospital has the collective knowledge of all its specialists (large total capacity), but the cost per patient visit is determined by the small number of specialists actually consulted (small active compute). In an MoE Transformer, each feed-forward network (FFN) layer is replaced by a collection of N_E parallel "expert" FFN layers and a learned gating network that routes each token to only K of them. The model has N_E times more parameters than the dense equivalent but uses only K/N_E of them per token.

11.3.1 Sparse Gating and Top-K Routing

The standard Transformer architecture processes each token through a shared FFN after the self-attention sublayer. In an MoE Transformer, this single FFN is replaced by N_E independent expert networks $E_1, E_2, \ldots, E_{N_E}$, each structurally identical to the original FFN. A gating network takes the token representation $\mathbf{x}$ and produces a probability distribution over experts, from which the top K are selected. Formally, the gating function and the MoE output are:

$$g(\mathbf{x}) = \mathrm{TopK}(\mathrm{softmax}(\mathbf{W}_g \mathbf{x})), \quad \mathrm{MoE}(\mathbf{x}) = \sum_{i \in \mathrm{TopK}} g_i(\mathbf{x}) \cdot E_i(\mathbf{x}) \tag{11.4}$$

where $\mathbf{W}_g \in \mathbb{R}^{N_E \times d}$ is a learned gating matrix, $\mathrm{softmax}(\mathbf{W}_g \mathbf{x})$ produces a probability distribution over all N_E experts, and TopK zeros out all entries except the K largest, renormalizing the remaining weights to sum to 1. Each selected expert E_i processes the input token independently, producing an output of the same dimension d, and the final MoE output is the weighted sum of the selected expert outputs. The key insight is that total parameters scale with N_E (because all expert weights are stored and trained), but compute per token scales with K (because only K experts are activated per token). With $N_E = 64$ experts and $K = 2$, the model has 64 times more parameters in its FFN layers than a dense equivalent but requires only $2/64 = 1/32$ of the FFN compute per token. This decoupling of parameters from compute is what makes MoE architecturally transformative for scaling.

A common misconception is that ignoring most experts loses information or makes the model approximate. In practice, experts specialize: different experts learn to handle different types of tokens. Empirical analyses of trained MoE models reveal that experts often specialize along semantic or syntactic lines – one expert may handle punctuation and function words, another may specialize in named entities, and another in mathematical notation. The gating network learns to route each token to the expert best equipped to process it. The specialization is emergent, not designed: the training dynamics naturally push experts toward complementary rather than redundant functions, because redundant experts receive diluted gradients and are eventually outcompeted by specialists. The result is a model that achieves the representational capacity of all its experts collectively while paying the computational cost of only the selected few.

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

class MoELayer(nn.Module):
    def __init__(self, d_model=64, n_experts=8, top_k=2, d_ff=256):
        super().__init__()
        self.experts = nn.ModuleList(
            [nn.Sequential(nn.Linear(d_model, d_ff), nn.ReLU(),
                           nn.Linear(d_ff, d_model))
             for _ in range(n_experts)])
        self.gate = nn.Linear(d_model, n_experts, bias=False)
        self.top_k, self.n_experts = top_k, n_experts

    def forward(self, x):                       # x: (B, T, d)
        logits = self.gate(x)                    # (B, T, n_experts)
```

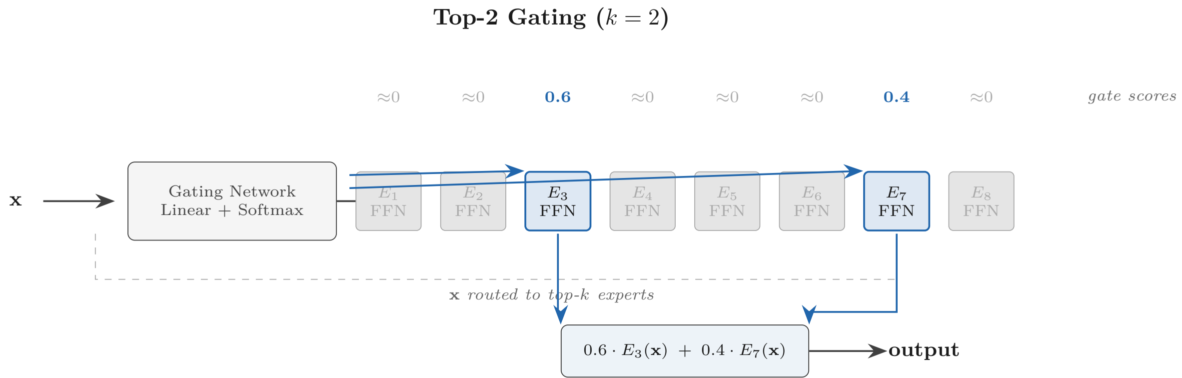

Figure 4: Figure 11.4 – MoE Architecture Diagram

```
        topk_val, topk_idx = logits.topk(self.top_k, dim=-1)
        weights = F.softmax(topk_val, dim=-1)   # normalize top-k
        out = torch.zeros_like(x)
        for k in range(self.top_k):
            for e in range(self.n_experts):
                mask = (topk_idx[..., k] == e)
                if mask.any():
                    expert_out = self.experts[e](x[mask])
                    out[mask] += weights[..., k][mask].unsqueeze(-1) * expert_out
        return out

moe = MoELayer(); x = torch.randn(2, 10, 64)
print(f"Input: {x.shape} -> Output: {moe(x).shape}")  # (2,10,64)
```

Code 11.3 – Simple MoE Layer with Top-2 Routing. This PyTorch module implements the core MoE mechanism from Equation 11.4. For each token, the gating network selects the top-2 experts and combines their outputs with softmax-normalized weights. Production implementations use optimized batching to avoid the nested loops shown here.

11.3.2 Load Balancing and Capacity Factor

Without intervention, MoE training collapses. The gating network, initialized randomly, will by chance route slightly more tokens to some experts than others. Those busy experts receive more gradient signal, improving faster, which makes the gating network route even more tokens to them. The result is a positive feedback loop where a handful of "winner" experts process most tokens while the remaining experts are starved of data and never learn. In the extreme case, the model degenerates into a dense model that uses only 2–3 of its 64 experts, with the rest contributing nothing. The load-balancing loss directly penalizes this collapse:

$$\mathcal{L}_{\text{balance}} = N_E \sum_{i=1}^{N_E} f_i \cdot p_i \tag{11.5}$$

where f_i is the fraction of tokens in the batch that are routed to expert i (a discrete quantity, determined by the TopK assignment) and p_i is the mean gating probability assigned to expert i across the batch (a continuous quantity, averaged from the softmax outputs). The product $f_i \cdot p_i$ is high when an expert receives both many tokens and high gating confidence. Summing across experts and multiplying by N_E produces a loss that is minimized when every expert receives an equal share of tokens ($f_i = 1/N_E$ for all i) and the gating network assigns equal probability to each expert. In practice, the load-balancing loss is added to the main language modeling loss with a small coefficient λ, typically in the range 0.01 to 0.1: $\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{LM}} + \lambda \mathcal{L}_{\text{balance}}$. Setting λ too high forces artificial uniformity and degrades model quality because different experts should naturally handle different token frequencies. Setting λ too low permits collapse. The trade-off is empirical and architecture-dependent.

Complementing the soft penalty of the load-balancing loss, the *capacity factor* CF provides a hard constraint. It limits the number of tokens that any single expert can process per batch to $CF \times (B/N_E)$, where B is the batch size. A capacity factor of 1.0 means each expert can process exactly its fair share; a factor of 1.5 allows 50% overflow. Tokens that exceed an expert's capacity are dropped (processed by the residual connection only, bypassing the expert entirely) or rerouted to the next-best expert. The capacity factor prevents any single expert from becoming a bottleneck, which is especially important in distributed settings where each expert may reside on a different device and uneven routing creates communication imbalance. The interplay between the soft load-balancing loss and the hard capacity factor creates a training regime where experts are encouraged but not forced to specialize evenly – a delicate balance that remains an active area of engineering and research. The restaurant analogy captures this well: the load-balancing loss is like a host who encourages diners to choose less crowded sections, while the capacity factor is a fire marshal who caps each section at a maximum occupancy regardless of demand.

11.3.3 MoE Architectures in Practice

The MoE concept predates the Transformer: Jacobs et al. [1991] introduced the mixture of experts as a general modular learning framework, and Jordan and Jacobs [1994] developed the hierarchical variant. But it was Shazeer et al. [2017] who first demonstrated that MoE could scale language models to previously impossible sizes. Their system embedded MoE layers within an LSTM language model, creating a 137-billion-parameter model that used top-2 routing to activate only a fraction of its experts per token. The model achieved better perplexity than dense baselines while using approximately one-quarter of the compute – the first compelling demonstration that the parameter/compute decoupling could work in practice at scale.

The Switch Transformer [Fedus et al., 2022] simplified and scaled the idea dramatically. Fedus, Zoph, and Shazeer made a counter-intuitive design choice: reduce the routing to top-1, meaning each token is processed by exactly one expert. This seems like it should lose information – one expert instead of two – but the simplification brought stability and efficiency gains that more than compensated. With top-1 routing, there is no need to combine expert outputs (no weighted sum), the routing logic is simpler, and the training dynamics are more stable. Switch Transformer scaled to 1.6 trillion parameters and demonstrated 4–7 times faster training than an equivalent-compute dense T5 model. The lesson was that in the MoE regime, simplicity can be more valuable than

expressiveness: a simpler routing scheme that trains stably at enormous scale outperforms a more expressive scheme that suffers from instability.

Mixtral [Jiang et al., 2024] brought MoE from the research laboratory to the open-weight community. Mixtral 8x7B consists of 8 expert networks, each with approximately 7 billion parameters, and uses top-2 routing. The total parameter count is approximately 47 billion (the experts share some layers), but the active parameter count per token is only about 12.9 billion. On standard benchmarks, Mixtral matched or exceeded LLaMA-2 70B – a dense model with more than 5 times the active parameters – while being dramatically cheaper at inference. Mixtral demonstrated that MoE is not merely a research curiosity but a practical production architecture that enables smaller, faster, and cheaper models with performance competitive against much larger dense alternatives. The rumors – unconfirmed but widely discussed – that GPT-4 uses a large MoE architecture suggest that the approach has been adopted at the very frontier of language model development.

The trade-offs of MoE are real and should not be minimized. Memory footprint is the most immediate: all expert weights must reside in memory even though only a subset is active per token, meaning that a Mixtral 8x7B model requires memory for all 47 billion parameters despite using only 12.9 billion during inference. In distributed training, routing tokens to experts on different devices introduces communication overhead that can negate the compute savings if the interconnect bandwidth is insufficient. Training instability from routing dynamics – experts dying, load imbalance, and sensitivity to the load-balancing coefficient – remains a persistent engineering challenge. And evaluation is complicated by the fact that MoE models are difficult to compare fairly against dense models: should we compare by total parameters, active parameters, or FLOPs per token? Each metric tells a different story, and the field has not converged on a standard.

11.4 Efficient Training

How do we fit a 70-billion-parameter model on hardware where a single GPU holds only 80 gigabytes of memory?

The scaling laws of Section 11.1 and the MoE architectures of Section 11.3 create the appetite for enormous models. But appetite is not the same as feasibility. A 70-billion-parameter language model in 16-bit floating point occupies roughly 140 gigabytes just for the weights – already exceeding the memory of any single GPU available in 2025. Add optimizer states (Adam stores two auxiliary vectors per parameter), gradients, and activations, and the total memory requirement for training balloons to several hundred gigabytes. Training such a model on a single device is physically impossible. The engineering discipline of distributed training exists to solve this problem, and the solutions are elegant, practical, and conceptually straightforward once we understand the three fundamental strategies – data parallelism, tensor parallelism, and pipeline parallelism – and the memory optimizations that complement them.

11.4.1 Data, Tensor, and Pipeline Parallelism

Data parallelism is the simplest and most widely used distributed training strategy. The idea is to replicate the entire model on each of P devices, split each training batch into P sub-batches (one per device), compute gradients locally on each device, and then synchronize gradients across devices using an all-reduce operation before updating the weights. The effect is that the effective batch size scales with the number of devices while the per-device computation is identical to single-GPU

training. The strength of data parallelism is its simplicity: any model that trains on a single GPU can be data-parallelized with minimal code changes. The limitation is that the entire model must fit on each device. For a 70-billion-parameter model that requires hundreds of gigabytes of memory, pure data parallelism is infeasible because no single GPU can hold the model. Data parallelism is like distributing copies of a textbook to every student in a classroom: each student reads a different chapter (data shard) and they pool their notes (gradients) at the end. It works perfectly as long as every student can carry the textbook.

Tensor parallelism, pioneered by the Megatron-LM framework [Shoeybi et al., 2019], splits individual weight matrices across multiple devices within a single layer. Consider a linear layer $\mathbf{Y} = \mathbf{XW}$ where $\mathbf{W} \in \mathbb{R}^{d \times d_{\text{ff}}}$. We can partition $\mathbf{W}$ column-wise into two halves, $\mathbf{W} = [\mathbf{W}_1 \mid \mathbf{W}_2]$, and place each half on a different GPU. Each GPU computes its portion of the output ($\mathbf{XW}_1$ and $\mathbf{XW}_2$ respectively), and the partial results are concatenated or reduced as needed. For the self-attention mechanism, this means splitting the query, key, and value projection matrices across devices – each device computes attention for a subset of heads, and the results are combined. Tensor parallelism keeps the model distributed within a layer, which means every layer requires inter-device communication. This makes it extremely sensitive to interconnect bandwidth: it is practical within a single node using NVLink (600+ GB/s) but prohibitively slow across nodes connected by InfiniBand (typically 100–400 GB/s). The analogy is splitting a page of text across desks: each student reads their column, but they must constantly lean over to check what the neighbor's column says, making fast communication essential.

Pipeline parallelism takes the opposite approach: instead of splitting layers across devices, it assigns entire layers to different devices. In a 96-layer Transformer with 8 GPUs, layers 1–12 go to GPU-0, layers 13–24 to GPU-1, and so on. Each device is responsible for a "stage" of the forward pass, and data flows through the stages sequentially. The main challenge is the *pipeline bubble*: while GPU-7 processes the forward pass of a batch, GPUs 0–6 are idle waiting for the next batch. Techniques like micro-batching (splitting each batch into smaller micro-batches that can be in different stages simultaneously) and interleaved scheduling reduce but do not eliminate this idle time. Typical pipeline parallelism achieves 60–80% device efficiency, meaning 20–40% of compute is wasted in bubbles. The analogy is an assembly line in a factory: each station performs its operation and passes the product to the next station, but changing products requires flushing the pipeline. Modern large-scale training uses all three strategies simultaneously – a configuration known as 3D parallelism. The canonical example is Megatron-DeepSpeed, which uses tensor parallelism within a single 8-GPU node (exploiting NVLink), pipeline parallelism across nodes (tolerating lower inter-node bandwidth), and data parallelism across replicas of the entire pipeline. A training run for a 175-billion-parameter model might use 1,024 GPUs organized as 128 nodes, with 8-way tensor parallelism within each node, 16-way pipeline parallelism across 16 nodes per replica, and 8-way data parallelism across 8 replicas. Orchestrating this 3D decomposition is one of the most complex engineering challenges in modern AI infrastructure.

11.4.2 Mixed Precision and Gradient Checkpointing

Mixed-precision training is one of the highest-impact optimizations in modern deep learning – a technique that halves memory consumption and doubles throughput while producing models statistically indistinguishable from those trained in full 32-bit precision. The idea is to store and compute most quantities in 16-bit floating point (FP16 or BF16) while maintaining a "master copy" of the weights in 32-bit floating point (FP32) for gradient accumulation. A single FP32 parameter occupies 4 bytes; an FP16 or BF16 parameter occupies 2 bytes. For a 70-billion-parameter model,

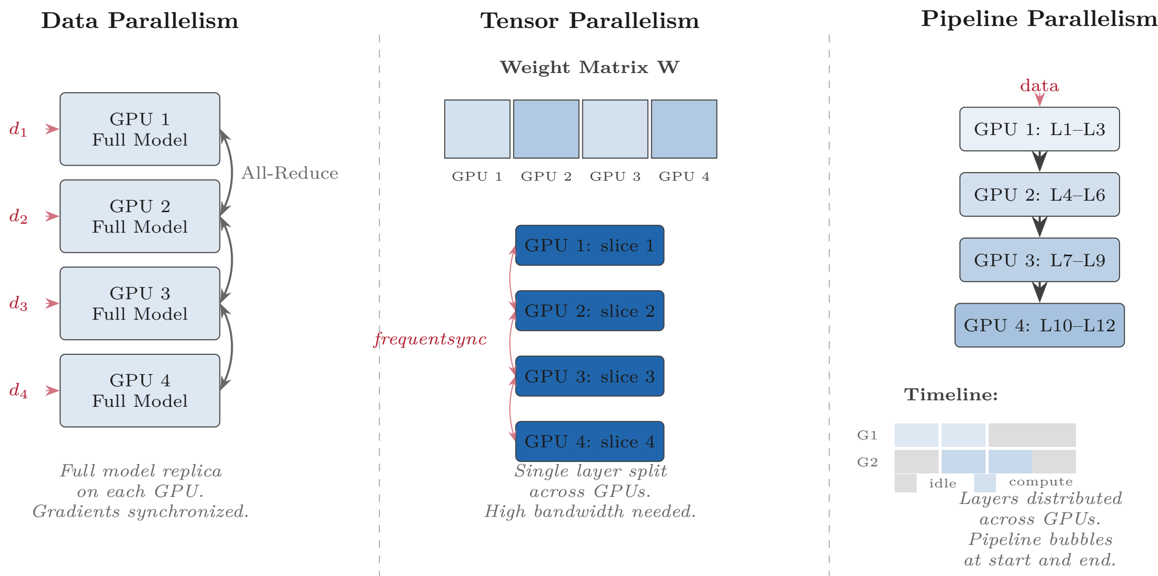

Figure 5: Figure 11.5 – Distributed Training Strategies

the difference is 140 GB versus 280 GB just for the weights – and the savings extend to activations, gradients, and communication volumes as well. Modern GPUs (NVIDIA A100, H100, and their successors) include specialized tensor cores that execute FP16/BF16 matrix multiplications at 2–4 times the throughput of FP32, meaning that mixed precision provides both memory and speed improvements simultaneously.

The two 16-bit formats serve different purposes. FP16 (IEEE half-precision) uses 5 exponent bits and 10 mantissa bits, providing a dynamic range of approximately 6×10^{-8} to 6.5×10^{4}. This limited range makes FP16 susceptible to underflow: gradients of small magnitude can be rounded to zero, causing training to stall. The standard mitigation is *loss scaling*: multiply the loss by a large constant (say, 1024) before the backward pass, which scales all gradients proportionally upward into the representable range, then divide back after the gradient computation. BF16 (bfloat16, or “brain float”) uses 8 exponent bits and 7 mantissa bits. The 8 exponent bits give BF16 the same dynamic range as FP32 ($\sim 10^{-38}$ to $\sim 10^{38}$), eliminating the underflow problem entirely at the cost of slightly lower precision. Most modern large-scale training runs use BF16 because it avoids the complexity of loss scaling while maintaining sufficient precision for stable training. The choice between FP16 and BF16 has no measurable effect on final model quality when proper techniques are applied.

Gradient checkpointing is a complementary optimization that trades compute for memory. During the forward pass of a deep network, all intermediate activations must be stored because they are needed during the backward pass to compute gradients. For a 96-layer Transformer, this means storing 96 layers’ worth of activations – a memory cost that grows linearly with model depth and can easily exceed the weight memory. Gradient checkpointing reduces this cost by storing activations at only a subset of layers (the “checkpoints”) and recomputing the missing activations from the

nearest checkpoint during the backward pass. If we checkpoint every $\sqrt{L}$ layers (where L is the total number of layers), the memory cost drops from $O(L)$ to $O(\sqrt{L})$ while the compute cost increases by approximately 33% (one additional forward pass for the recomputed segments). For a 96-layer model, this means storing activations for only $\sqrt{96} \approx 10$ layers instead of 96 – a roughly 10-fold memory reduction at the cost of a one-third increase in training time. The trade-off is almost always worthwhile for large models, where memory is the binding constraint.

11.4.3 ZeRO and Memory Optimization

Even with mixed precision and gradient checkpointing, the memory requirements of large-scale training are substantial. Consider a 7-billion-parameter model trained with Adam in mixed precision. The memory budget breaks down as follows: 14 GB for the FP16 model weights, 28 GB for the FP32 master weights, 28 GB for the two FP32 Adam optimizer states (first and second moment estimates, $\mathbf{m}$ and $\mathbf{v}$), and 14 GB for the FP16 gradients – a total of approximately 84 GB. A single NVIDIA A100 GPU has 80 GB of memory, which cannot even fit this 7-billion-parameter model's training state, and we have not yet accounted for activations. In standard data parallelism, this problem is worse: every GPU stores a complete copy of the weights, optimizer states, and gradients, multiplying the per-GPU memory requirement by a factor of one (no improvement) while providing no memory benefit from the additional devices. The redundancy is enormous: 64 GPUs each storing 84 GB of identical state, totaling 5.4 TB of memory for what is really 84 GB of unique information.

ZeRO (Zero Redundancy Optimizer), developed by Rajbhandari et al. [2020] as part of the DeepSpeed library, eliminates this redundancy through a simple but powerful insight: instead of replicating all training state on every device, partition it across devices so that each device stores only $1/P$-th of the total state, where P is the number of devices. ZeRO defines three progressive stages of partitioning. Stage 1 partitions only the optimizer states: each GPU stores $1/P$ of the Adam $\mathbf{m}$ and $\mathbf{v}$ vectors. This alone reduces per-GPU memory by roughly $4\times$ for Adam (which stores 2 states per parameter). Stage 2 adds gradient partitioning: gradients are also distributed, so each GPU stores only the gradients for the parameters whose optimizer states it owns. Stage 3 goes furthest: even the model parameters are partitioned, so each GPU stores only $1/P$ of the weights and gathers the remaining parameters from other GPUs as needed during the forward and backward passes. With 8 GPUs and ZeRO Stage 3, our 7-billion-parameter model requires approximately $84/8 = 10.5$ GB per GPU for training state – well within a single A100's capacity, with ample room remaining for activations. The cost of ZeRO is additional communication: parameters must be all-gathered before each forward and backward pass, and gradients must be reduce-scattered rather than all-reduced. With modern high-bandwidth interconnects (NVLink within a node, InfiniBand between nodes), the overhead of ZeRO Stage 2 is negligible, and Stage 3 adds approximately 10–15% to training time.

Fully Sharded Data Parallelism (FSDP), introduced as a native PyTorch feature, is functionally equivalent to ZeRO Stage 3 and has become the standard approach for training models in the 7–70 billion parameter range on PyTorch. FSDP wraps each layer of the model in a sharding context that automatically distributes parameters across devices, gathers them for computation, and releases them immediately afterward to free memory. The programming model is remarkably simple: wrap the model in an FSDP constructor and training proceeds with no further code changes. The standard data parallelism analogy – every worker holds a complete copy of the blueprint – becomes: each worker keeps only their section of the blueprint and borrows neighboring sections as needed. The total capacity is the same; the per-worker cost is dramatically lower. Together, the techniques of Section 11.4 form a layered optimization stack: 3D parallelism distributes the model across devices, mixed precision reduces the memory footprint of each component, gradient

checkpointing trades compute for activation memory, and ZeRO/FSDP eliminates the redundancy of replicated state. No single technique suffices for frontier-scale training; the combination of all four is what makes it feasible.

11.5 The Compute Frontier

How much did it cost to train GPT-3, and what does that mean for the rest of us?

Approximately $4.6 million. That is the commonly cited estimate for the cost of training GPT-3, based on the list price of cloud GPU time in 2020. The number is both staggering and, by frontier standards, already quaint: GPT-4's training cost is estimated at $50–100 million, and the most expensive training runs of 2025 likely exceed $100 million. The scaling laws of Section 11.1 explain why: each order-of-magnitude improvement in loss requires roughly three orders of magnitude more compute, and while hardware improves, it does not improve fast enough to offset the exponential growth in compute demand. This section examines the compute trajectory of language model training, the costs involved, the open-weight movement that democratizes access to trained models, and the implications for researchers operating at every point on the compute spectrum.

11.5.1 Compute Trends and Cost Estimates

The compute trajectory of language model training has outpaced Moore's Law by a wide margin. The original Transformer [Vaswani et al., 2017] required approximately 10^{18} FLOPs to train. BERT [Devlin et al., 2019] consumed roughly 10^{19} FLOPs. GPT-2 [Radford et al., 2019] pushed to approximately 10^{20} FLOPs. Then came the inflection: GPT-3 [Brown et al., 2020] required approximately 3.6×10^{23} FLOPs – a thousand-fold jump in a single generation. Chinchilla [Hoffmann et al., 2022] used roughly 5.8×10^{23} FLOPs for its 70-billion-parameter model. LLaMA-2 [Touvron et al., 2023] at its largest (70B) consumed approximately 10^{24} FLOPs. Frontier models in 2024–2025 are estimated at 10^{25} to 10^{26} FLOPs. The doubling time for training compute has been approximately 6–10 months since 2017, compared to Moore's Law's two-year doubling cycle for transistor density. We are in a regime where the appetite for compute is growing far faster than the cost of compute is falling.

The cost of training can be estimated from the compute budget using a simple formula: Cost $= C/(\text{GPU}_{\text{peak}} \times \text{efficiency}) \times \text{hours}^{-1} \times \text{cost}_{\text{per GPU-hour}}$. For GPT-3 ($C = 3.6 \times 10^{23}$ FLOPs) on V100 GPUs (peak throughput approximately 10^{14} FLOPs/s at FP16, efficiency approximately 30–50%), the estimated training time is on the order of 10^4 GPU-hours, or several thousand GPU-days. At cloud prices of $1–$3 per GPU-hour for V100s in 2020, this yields the commonly cited estimate of $4–$5 million. The cost of frontier models has increased far faster than hardware improvements have decreased per-FLOP cost. The H100 GPU (2023) delivers roughly 3–4 times the throughput of the A100 (2020), which itself delivered roughly 2–3 times the throughput of the V100 (2017). These improvements are significant but represent roughly one order of magnitude improvement in throughput over 6 years, while compute demand has grown by more than six orders of magnitude over the same period. The gap is closed by a combination of larger clusters (from hundreds of GPUs for GPT-2 to tens of thousands for frontier models), longer training runs, and institutional budgets that have grown from the millions to the hundreds of millions of dollars.

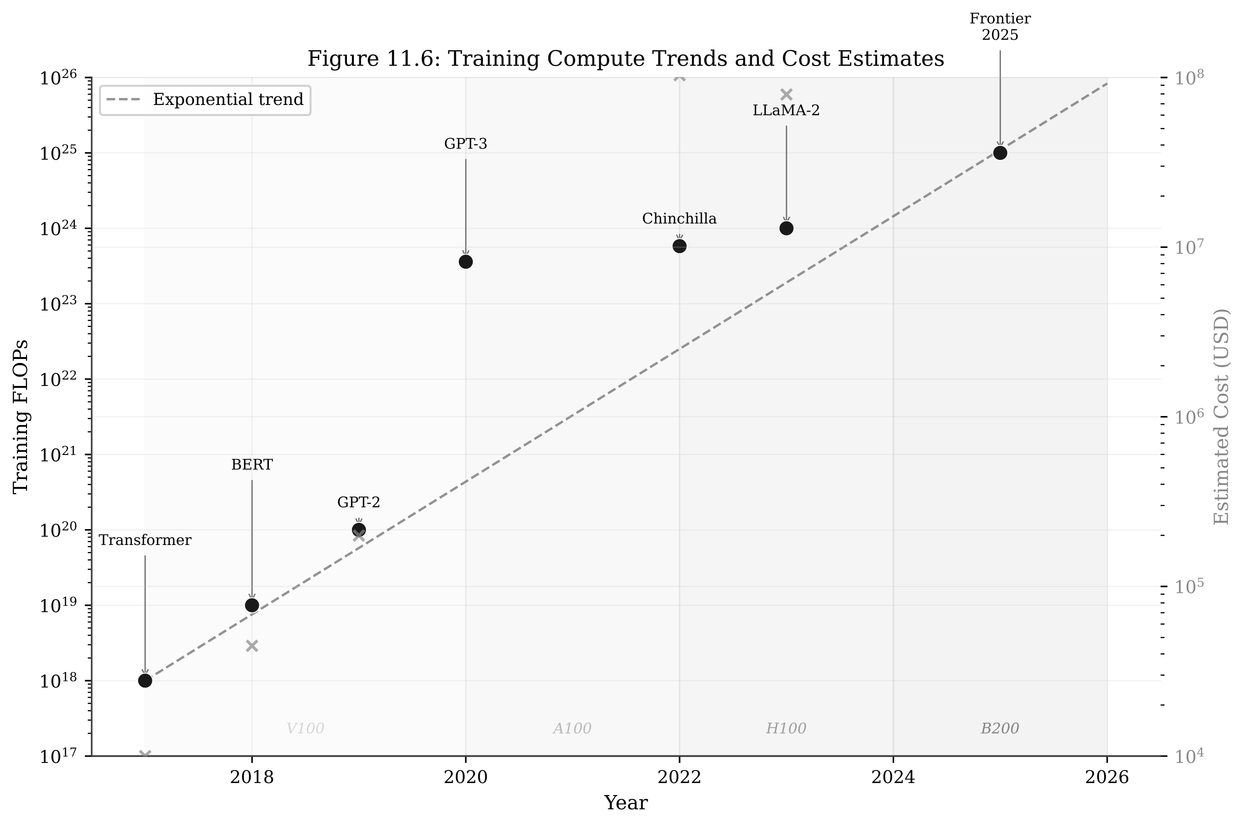

Figure 6: Figure 11.6 – Compute Trends and Cost Estimates

11.5.2 The Open-Weight Movement

The release of LLaMA by Meta in February 2023 [Touvron et al., 2023] triggered what can fairly be called a revolution in the accessibility of large language models. Before LLaMA, training a competitive language model from scratch required the resources of a major technology company. After LLaMA, a 7-billion-parameter model competitive with GPT-3 on many benchmarks was available as a downloadable checkpoint for anyone with a single GPU capable of running inference. The impact was immediate and enormous: within weeks, the open-source community produced fine-tuned variants including Alpaca (Stanford), Vicuna (LMSYS), and dozens of others, each adapting LLaMA's pre-trained weights for specific applications at trivial cost compared to pre-training from scratch. The open-weight movement accelerated through 2023 and 2024 with releases from multiple organizations. Mistral [Jiang et al., 2024] released a 7-billion-parameter model that matched LLaMA-2 13B despite being nearly half the size. Mixtral, discussed in Section 11.3.3, brought MoE to the open-weight community. Falcon [Penedo et al., 2023] from the Technology Innovation Institute released 40B and 180B models with fully permissive licenses. OLMo [Groeneveld et al., 2024] from the Allen Institute for AI went furthest, releasing not just the model weights but the training code, the training data, intermediate checkpoints, and evaluation infrastructure – enabling full reproducibility of the training pipeline.

A crucial distinction that students must understand is the difference between "open-weight" and "open-source" in this context. Most models marketed as "open" release only the trained weights and the inference code. They do not release the training data, the training code, the data preprocessing pipelines, or intermediate checkpoints. This means that researchers can use these models for inference and fine-tuning but cannot fully reproduce or audit the training process. True open-source models – OLMo, Pythia [Biderman et al., 2023] – release the complete pipeline. The distinction matters for scientific reproducibility: understanding why a model exhibits certain behaviors requires access to its training data and training dynamics, not just its final weights. The open-weight ecosystem has created a two-tier structure in the field. Frontier laboratories (OpenAI, Google DeepMind, Anthropic) push the boundaries of scale and capability with proprietary models. The open community, using released weights, rapidly iterates on efficiency (quantization, distillation), alignment (RLHF, DPO), evaluation, and applications. This division of labor is productive: the frontier labs absorb the enormous cost of pre-training, and the open community amortizes this investment across thousands of downstream applications. The implication for students and academic researchers is direct and practical: the cost of entry into state-of-the-art NLP research has dropped from millions of dollars (training from scratch) to hundreds of dollars (fine-tuning an open-weight model on cloud GPUs) in the span of two years.

11.5.3 Implications for Researchers

A $10,000 GPU budget buys approximately 10^{20} FLOPs on cloud hardware in 2025. According to the Chinchilla scaling laws, this is sufficient to train a roughly 100-million-parameter model on 2 billion tokens – far below the frontier but far from useless. The scaling laws themselves provide the crucial guidance: they tell us exactly what we can and cannot achieve with a given budget, and they legitimize small-scale experimentation by demonstrating that phenomena observed at small scale often transfer predictably to larger scales. Chinchilla itself was validated using models as small as 70 million parameters. Many of the most important results in scaling law research were obtained through systematic small-scale experiments that were then confirmed at larger scale. The research strategy for compute-constrained researchers is therefore not to attempt frontier-scale training but to use small-scale experiments, validated by scaling laws, to draw conclusions that transfer to larger

models.

Rich Sutton's "Bitter Lesson," published as a blog post in March 2019, articulated a meta-observation that the scaling laws of this chapter powerfully illustrate. Sutton – one of the founders of modern reinforcement learning – argued that across the 70-year history of artificial intelligence, methods that leverage computation have always ultimately won over methods that leverage human knowledge. In chess, decades of hand-crafted heuristics were surpassed by brute-force search with sufficient compute. In speech recognition, linguistically motivated feature engineering was overtaken by end-to-end neural networks trained on large datasets. In computer vision, hand-designed features gave way to deep convolutional networks. The "bitter" part of the lesson is its implication for researchers: the clever algorithm, the elegant representation, the carefully engineered feature – each of these will eventually be surpassed by a simpler method running on more hardware. The scaling laws for language models provide perhaps the strongest evidence yet for the Bitter Lesson. Kaplan et al. showed that Transformer language models improve predictably with more compute, following power laws that hold across seven orders of magnitude. No architectural innovation has broken these laws – no clever attention mechanism, no novel position encoding, no breakthrough training trick has produced a model that lies significantly below the scaling curve predicted by the simple recipe of "Transformer + more compute + more data." The Chinchilla revision refined the recipe (balance parameters and data) but did not alter the fundamental message: scale wins.

We should be careful, however, not to overstate the Bitter Lesson. Sutton's argument is about the *long run*, and in the long run, he appears to be correct. But in the short and medium run, algorithmic innovation matters enormously. The Transformer architecture itself – the subject of Chapter 8 – was a breakthrough that enabled a new scaling regime; the models that preceded it (LSTMs, convolutional models) did not scale nearly as cleanly. Flash Attention, a purely algorithmic innovation, reduced the memory complexity of attention from $O(n^2)$ to $O(n)$ without changing the model architecture, enabling longer context lengths and faster training. Mixture of Experts (Section 11.3) is an architectural innovation that decouples parameters from compute, enabling more efficient scaling. Quantization, distillation, and speculative decoding – topics deferred to Chapter 14 – are engineering innovations that reduce inference cost by orders of magnitude. Each of these is a form of "human ingenuity" that the Bitter Lesson seemingly dismisses, yet each has been essential for making scale practical. The resolution is that the Bitter Lesson is correct about the first-order effect (more compute produces better models) but incomplete about the second-order effects (algorithmic innovation determines how efficiently compute translates into capability). For the academic researcher with a limited budget, the second-order effects are where the action is. The frontier labs have the first-order effect covered; the research contributions available to everyone else lie in making each FLOP count for more.

The compute divide is real but not paralyzing. Researchers with modest budgets can pursue several high-impact research directions. Data quality and curation: the Phi models demonstrated that small models trained on carefully selected data can punch far above their weight. Evaluation and benchmarking: developing better metrics for emergent abilities, safety, and alignment requires careful experimental design, not massive compute. Mechanistic interpretability: understanding what individual neurons, attention heads, and circuits do inside a trained model is compute-light and scientifically deep. Post-training methods: fine-tuning, alignment, and prompting techniques can be developed and evaluated using open-weight models at minimal cost. Theoretical understanding: explaining why scaling laws are power laws, why certain abilities emerge, and what the fundamental limits of next-token prediction are – these are open theoretical questions that require insight, not FLOPs.

Section 11.5 closes the chapter's arc from mathematics to practice: scaling laws tell us what is possible at each budget level, the open-weight movement makes pre-trained models accessible to all, and the Bitter Lesson reminds us that scale is the dominant force – while leaving ample room for the algorithmic ingenuity that makes scale tractable.

Scaling laws tell us that larger models are more capable predictors, and emergent abilities show that sufficiently large models develop surprising new capabilities – capabilities not anticipated by the smooth extrapolation of loss curves. Mixture of Experts and distributed training engineering make this scale physically realizable. But capability and safety are not the same thing. A model that can write poetry, solve math problems, and generate code can also produce toxic content, fabricate convincing falsehoods, and amplify social biases with exactly the same facility. The more capable the model, the more consequential these failure modes become. In Chapter 12, we address the alignment problem: how do we steer a powerful predictor so that it is not just capable but helpful, harmless, and honest? The tools – RLHF, DPO, and Constitutional AI – represent one of the most active and consequential research areas in modern AI.

Scaling produces powerful but potentially unsafe models, providing a natural transition to Chapter 12, where we explore alignment techniques—RLHF, DPO, and constitutional AI—that steer language models toward human preferences.

Exercises

Exercise 11.1 (Theory – Basic). Given a compute budget of $C = 10^{22}$ FLOPs, use the Chinchilla scaling law to compute the optimal number of parameters N_{opt} and training tokens D_{opt}. Use the approximation $C \approx 6ND$ and $D/N \approx 20$. Is a 10-billion-parameter model trained on 100 billion tokens compute-optimal for this budget? If not, explain whether it is over-parameterized or under-trained, and by what factor.

Hint: From $6ND = C$ and $D = 20N$, we get $N = \sqrt{C/120}$. Check whether the given model's D/N ratio matches the Chinchilla-optimal ratio of approximately 20.

Exercise 11.2 (Theory – Intermediate). The Kaplan scaling law gives $\alpha_N \approx 0.076$. Compute the factor by which loss decreases when model size increases from 1B to 10B parameters. Then compute the factor for 10B to 100B. Explain, using the properties of power laws, why the improvement from 10B to 100B is the same multiplicative factor as from 1B to 10B. What does this imply about the diminishing *absolute* returns from scaling?

Hint: For a power law $L \propto N^{-\alpha}$, the ratio $L(10N)/L(N) = 10^{-\alpha}$, independent of the starting N. This is the defining property of scale-free relationships.

Exercise 11.3 (Theory – Intermediate). Explain why Mixture of Experts decouples total parameters from compute cost. A dense Transformer with 70B parameters requires approximately $6 \times 70 \times 10^9 = 4.2 \times 10^{11}$ FLOPs per token. A Mixtral-style MoE with 8 experts of 7B each and top-2 routing has 56B total parameters but only approximately 12.9B active parameters. Estimate its FLOPs per token. What is its parameter-to-compute ratio compared to the dense model?

Exercise 11.4 (Theory – Intermediate). Schaeffer et al. [2023] argue that emergent abilities are metric artifacts. Design an experiment to test this claim. Choose a task that Wei et al. [2022b]

identified as emergent (e.g., multi-digit addition), propose both a discrete and a continuous evaluation metric, describe what data you would collect at model sizes of 1B, 10B, and 100B parameters, and predict what each metric would show if Schaeffer's hypothesis is correct versus if emergence is genuine.

Hint: Consider multi-digit addition. Discrete metric: exact-match accuracy (all digits correct). Continuous metric: per-digit accuracy. If Schaeffer is right, the continuous metric should improve smoothly while the discrete metric shows a sharp jump.

Exercise 11.5 (Programming – Basic). Using the scaling law $L(N) = (N_c/N)^{\alpha_N}$ with $N_c = 8.8 \times 10^{13}$ and $\alpha_N = 0.076$, plot the predicted loss for models ranging from 10M to 1T parameters on a log-log scale. Annotate the plot with the known sizes of GPT-2 (1.5B), GPT-3 (175B), Chinchilla (70B), and LLaMA-2 (70B, but trained on much more data than Chinchilla). Use `matplotlib` with log-log axes and include a legend and axis labels.

Exercise 11.6 (Programming – Intermediate). Implement a simple MoE FFN layer in PyTorch with $N_E = 8$ experts and top-$K = 2$ routing. Include the load-balancing loss from Equation 11.5 with coefficient $\lambda = 0.01$. Pass a random batch through the layer and print: (a) the output shape, (b) the expert efficiency (fraction of tokens routed to each expert), and (c) the load-balancing loss value. Verify that the load-balancing loss is minimized when expert efficiency is uniform.

Exercise 11.7 (Programming – Intermediate). Download the approximate scaling law data from Kaplan et al. (or use the numbers given in this chapter) and fit a power law $L(N) = a \cdot N^{-b}$ using least-squares regression on the log-transformed data. Take the log of both sides: $\log L = \log a - b \log N$, and use `numpy.polyfit` to fit the slope $(-b)$ and intercept $(\log a)$. Report the fitted exponent b and compare with Kaplan's $\alpha_N = 0.076$. Plot both the data points and your fitted line.

Exercise 11.8 (Programming – Advanced). Simulate the emergence phenomenon. Create a synthetic task where a model must get K independent sub-steps correct to succeed (exact-match). The probability of each sub-step being correct improves smoothly with scale as $p(N) = \sigma(\alpha(\log_{10} N - \mu))$, where σ is the sigmoid function. Plot exact-match accuracy $p(N)^K$ versus $\log_{10} N$ for $K = 1, 4, 8$ on the same axes (use $\alpha = 2$, $\mu = 10$). Show that higher K produces sharper "emergence." Discuss whether this simulation supports or undermines the claim that emergence is a genuine phase transition.

References

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. (2020). Language Models are Few-Shot Learners. In *Advances in NeurIPS*.

Biderman, S., Schoelkopf, H., Anthony, Q., Bradley, H., O'Brien, K., Hallahan, E., Khan, M. A., Purohit, S., Prashanth, U. S., Raff, E., et al. (2023). Pythia: A Suite for Analyzing Large Language Models Across Training and Scaling. In *Proceedings of ICML*.

Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of NAACL-HLT*, 4171–4186.

Fedus, W., Zoph, B., & Shazeer, N. (2022). Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity. *JMLR*, 23(120), 1–39.

Groeneveld, D., Beltagy, I., Walsh, P., Bhagia, A., Kinney, R., Tafjord, O., Joshi, A., Pyatkin, V., Dey, A., Wettig, A., et al. (2024). OLMo: Accelerating the Science of Language Models. In *Proceedings of ACL*.

Henighan, T., Kaplan, J., Katz, M., Chen, M., Hesse, C., Jackson, J., Jun, H., Brown, T. B., Dhariwal, P., Gray, S., et al. (2020). Scaling Laws for Autoregressive Generative Modeling. arXiv:2010.14701.

Hoffmann, J., Borgeaud, S., Mensch, A., Buchatskaya, E., Cai, T., Rutherford, E., Casas, D. de L., Hendricks, L. A., Welbl, J., Clark, A., et al. (2022). Training Compute-Optimal Large Language Models. In *Advances in NeurIPS*.

Jacobs, R. A., Jordan, M. I., Nowlan, S. J., & Hinton, G. E. (1991). Adaptive Mixtures of Local Experts. *Neural Computation*, 3(1), 79–87.

Jiang, A. Q., Sablayrolles, A., Roux, A., Mensch, A., Savary, B., Bamford, C., Chaplot, D. S., Casas, D. de L., Hanna, E. B., Bressand, F., et al. (2024). Mixtral of Experts. arXiv:2401.04088.

Jordan, M. I., & Jacobs, R. A. (1994). Hierarchical Mixtures of Experts and the EM Algorithm. *Neural Computation*, 6(2), 181–214.

Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., & Amodei, D. (2020). Scaling Laws for Neural Language Models. arXiv:2001.08361.

Penedo, G., Malartic, Q., Hesslow, D., Cojocaru, R., Cappelli, A., Alobeidli, H., Launay, J., & Muennighoff, N. (2023). The RefinedWeb Dataset for Falcon LLM. In *Advances in NeurIPS*.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., & Sutskever, I. (2019). Language Models are Unsupervised Multitask Learners. OpenAI Technical Report.

Rajbhandari, S., Rasley, J., Rber, O., & He, Y. (2020). ZeRO: Memory Optimizations Toward Training Trillion Parameter Models. In *Proceedings of SC*.

Schaeffer, R., Miranda, B., & Koyejo, S. (2023). Are Emergent Abilities of Large Language Models a Mirage? In *Advances in NeurIPS*.

Shazeer, N., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q., Hinton, G., & Dean, J. (2017). Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer. In *Proceedings of ICLR*.

Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., & Catanzaro, B. (2019). Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. arXiv:1909.08053.

Sutton, R. S. (2019). The Bitter Lesson. Blog post, March 13, 2019.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Roziere, B., Goyal, N., Hambro, E., Azhar, F., et al. (2023). LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). Attention Is All You Need. In *Advances in NeurIPS*.

Wei, J., Tay, Y., Bommasani, R., Raffel, C., Zoph, B., Borgeaud, S., Yogatama, D., Bosma, M., Zhou, D., Metzler, D., Chi, E. H., Hashimoto, T., Vinyals, O., Liang, P., Dean, J., & Fedus, W. (2022b). Emergent Abilities of Large Language Models. *Transactions on Machine Learning Research*.

Zhai, X., Kolesnikov, A., Houlsby, N., & Beyer, L. (2022). Scaling Vision Transformers. In *Proceedings of CVPR*.