Fantastic Language Models and How to Build Them

Guest Lecture — CS 224U: Natural Language Understanding
Stanford || Zoom || Folks 2x-ing the Recording
April 12, 2023
On the Importance of “Building”

Today — a *practical* take on large-scale language models (LLMs).

Whirlwind tour of the full pipeline:

- **Model Architecture** — Evolution of the Transformer
- **Training at Scale** — From 124M to 1T+ Parameters
- **Efficient Finetuning & Inference** — Tips & Tricks

**Punchline**: From “folk knowledge” —> insight / intuition / (re-)discovery!

*Please ask lots of questions! Why is this information useful to <YOU>?
Part I: Evolution of the Transformer

“Experiment is the mother of knowledge.”
— Madeline L’Engle, *A Wrinkle in Time*
Recipe for a Good™ Language Model

Massive amounts of cheap, easy to acquire data...

X

... a simple, high-throughput way to consume it!

Natural to scale with data.
Composable and “general”.

Fast & parallelizable training.
High hardware utilization.

Minimal “assumptions” on relationships between data?

<Story Time>
Pre-2017 — Historical Context

RNN Key Ideas: Long Context, Attention

CNN Key Ideas:
- Layer: Multiple “Filters” (Views)
- Scaling Depth w/ Residuals
- Parallelizable!

Reference: “Convolutional Neural Networks for Text,” Lena Voita. ML for NLP @ YSDA
Formulating the Self-Attention Block

Self-Attention: “The” —> query, key, & value

Multi-Headed: Different “views” per layer

< Is this actually better? >

class Attention(nn.Module):
    def __init__(self, embed_dim: int, n_heads: int):
        super().__init__()
        self.n_heads, self.dk = n_heads, (embed_dim // n_heads)
        self.qkv = nn.Linear(embed_dim, 3 * embed_dim)
        self.proj = nn.Linear(embed_dim, embed_dim)

    def forward(self, x: Tensor[bsz, seq, embed_dim]):
        q, k, v = rearrange(
            self.qkv(x),
            "bsz_seq (qkv nh dk) -> qkv bsz nh seq dk",
            qkv=3,
            nh=self.n_heads,
            dk=self.dk,
        ).unbind(0)
Aside — Self-Attention & Parallelization

**Recurrent Neural Network**

Works on **Ordered Sequences**

(+): Good at long sequences: After one RNN layer, $h_T$ "sees" the whole sequence

(-): Not parallelizable: need to compute hidden states sequentially

**1D Convolution**

Works on **Multidimensional Grids**

(-): Bad at long sequences: Need to stack many conv layers for outputs to "see" the whole sequence

(+): Highly parallel: Each output can be computed in parallel

**Self-Attention**

Works on **Sets of Vectors**

(+): Good at long sequences: after one self-attention layer, each output "sees" all inputs!

(+): Highly parallel: Each output can be computed in parallel

(-): Very memory intensive

< Great! But… what am I missing? >

Reference: Justin Johnson/Danfei Xu from CS 231N / DL @ GT
Formulating the Self-Attention Block

```python
class Attention(nn.Module):
    def __init__(self, embed_dim: int, n_heads: int):
        super().__init__()
        self.n_heads, self.dk = n_heads, (embed_dim // n_heads)
        self.qkv = nn.Linear(embed_dim, 3 * embed_dim)
        self.proj = nn.Linear(embed_dim, embed_dim)

    def forward(self, x: Tensor[bsz, seq, embed_dim]):
        q, k, v = rearrange(
            self.qkv(x),
            "bsz seq (qkv nh dk) -> qkv bsz nh seq dk",
            qkv=3,
            nh=self.n_heads,
            # Different "views" (like CNN filters)!
            dk=self.dk,
        ).unbind(0)

        # RNN Attention --> *for each view*
        scores = torch.softmax(
            q @ (k.transpose(-2, -1)),
            dim=-1
        )
        return self.proj(
            rearrange(scores @ v, "b nh seq dk -> b seq (nh dk)"
        )
```

< Where's my nonlinearity? >
Expressivity & Nonlinearity

```python
class ExpressiveTransformerBlock(nn.Module):
    def __init__(self, embed_dim: int, n_heads: int, up: int = 4):
        super().__init__()
        self.attn = Attention(embed_dim, n_heads)

        # Project *up* to high-dimension, nonlinear, compress!
        self.mlp = nn.Sequential(
            nn.Linear(embed_dim, up * embed_dim),
            nn.ReLU(),
            nn.Linear(up * embed_dim, embed_dim)
        )

    def forward(self, x: torch.Tensor):
        x = x + self.attn(x)
        x = x + self.mlp(x)
        return x
```

Residual + MLP → “Sharpen” + “Forget”

CS 229 → SVMs & “Implicit Lifting”

< New Problem — Activations Blow Up! >
Going Deeper — Activation Instability

```python
class NormalizedTransformerBlock(nn.Module):
    def __init__(self, embed_dim: int, n_heads: int, up: int = 4):
        super().__init__()
        self.attn = Attention(embed_dim, n_heads)
        self.mlp = nn.Sequential(
            nn.Linear(embed_dim, up * embed_dim),
            nn.ReLU(),
            nn.Linear(up * embed_dim, embed_dim)
        )

        # Add Normalization Layers
        self.attn_norm = nn.LayerNorm(embed_dim)
        self.mlp_norm = nn.LayerNorm(embed_dim)

    def forward(self, x: torch.Tensor):
        x = self.attn_norm(x + self.attn(x))
        x = self.mlp_norm(x + self.mlp(x))
        return x
```

Layer Normalization

< And... we're done? >

Well, Shucks —> Emergent Optimization Problems

Typical LR Decay
[CS 221, CS 229]

![Graph showing typical LR decay over epochs]

Transformer Pretraining LR Schedule
Linear Warmup (5% of Training) then Decay…

![Graph showing transformer pretraining LR schedule]

Learning Rate Warmup —> Breaks conventional machine learning wisdom?

< Ok but... why? >

3 Years Later...

3.1. Problem in Transformer Optimization

In this section we demonstrate that the requirement for warmup comes from a combined effect of high variance in the Adam optimizer and backpropagation through layer normalization. Liu et al. (2020) showed that at the begin-

of the input. Specifically, the gradient has the following property:

\[
\left\| \frac{\partial \text{LN}(x)}{\partial x} \right\| = O \left( \frac{\sqrt{d}}{\|x\|} \right)
\]

(1)

where \( x \) is the input to layer normalization and \( d \) is the embedding dimension. If input norm \( \|x\| \) is larger than \( \sqrt{d} \) then backpropagation through layer normalization has a down scaling effect that reduces gradient magnitude for lower layers. Compounding across multiple layers this can quickly lead to gradient vanishing.

"Ok, now we're done...?"
class ModernTransformerBlock(nn.Module):
    def __init__(self, embed_dim: int, n_heads: int, up: int = 4):
        super().__init__()
        self.attn = Attention(embed_dim, n_heads, qk_bias=False)
        self.mlp = nn.Sequential(
            SwishGLU(embed_dim, up * embed_dim),
            nn.Linear(up * embed_dim, embed_dim)
        )

        # Post-Norm --> *Pre-Norm*
        self.pre_attn_norm = RMSNorm(embed_dim)
        self.pre_mlp_norm = RMSNorm(embed_dim)

    def forward(self, x: T[bsz, seq, embed_dim]):
        x = x + self.attn(self.pre_attn_norm(x))
        x = x + self.mlp(self.pre_mlp_norm(x))
        return x

# SwishGLU -- A Gated Linear Unit (GLU) with Swish Activation
class SwishGLU(nn.Module):
    def __init__(self, in_dim: int, out_dim: int):
        super().__init__()
        self.swish = nn.SiLU()
        self.project = nn.Linear(in_dim, 2 * out_dim)

    def forward(self, self, x: T[bsz, seq, embed_dim]):
        projected, gate = self.project(x).tensor_split(2, dim=-1)
        return projected * self.swish(gate)

# RMSNorm -- Simple Alternative to LayerNorm
class RMSNorm(nn.Module):
    def __init__(self, self, dim: int, eps: float = 1e-8):
        super().__init__()
        self.scale, self.eps = dim**-0.5, eps
        self.g = nn.Parameter(torch.ones(dim))

    def forward(self, self, x: T[bsz, seq, embed_dim]):
        norm = torch.norm(x, dim=-1, keepdim=True) * self.scale
        return x / norm.clamp(min=self.eps) * self.g
Part II: Training at Scale

“Nothing in life is to be feared. It is only to be understood.”
— Marie Curie
• “Standard Pipeline”: Train on 1 GPU (e.g., on Colab) —> ~**max of a few hours**.

• Let’s train a GPT-2 Small (124M)!
  • **Problem**: Batch > 4 goes OOM on a decent GPU = > **12 GB** of GPU RAM
  • Simple Trick —> *Gradient Accumulation*
    • But… **99.63 Days** to train on Single GPU (400K Steps)
Shortening the Clock —> The Scaling Toolbox

GPT-2 Training Clock

99.63 D

**Goal:** 100 Days on 1 GPU —> ~4 Days on 16 GPUs

- **Data Parallelism** — Scaling across GPUs & Nodes
- **Mixed Precision** — Bits, Bytes, and TensorCores
- **ZeRO Redundancy** — Minimizing Memory Footprint

Later... Model Parallelism — Hardware Limitations — Software Optimization

*Even if you’re not training big models... understanding breeds innovation!*
Data Parallelism — A Toy Example

GPT-2 Training Clock

Idea —> Parallelize?

SIMD
Single Instruction, Multiple Data

SPMD
Single Program, Multiple Data

< Seems hard? >

BATCH_SIZE = 128

class MLP(nn.Module):
    def __init__(self, n_classes):
        int = 10, mnist_dim: int = 784, hidden: int = 128
    :
        super().__init__()
        self.mlp = nn.Sequential
            nn.Linear(mnist_dim, hidden),
            nn.ReLU(),
            nn.Linear(hidden, hidden),
            nn.ReLU(),
            nn.Linear(hidden, n_classes)
    
    def forward(self, x: T[bsz, mnist_dim]):
        return self.mlp(x)

# Main Code
data_loader = DataLoader(dataset=torchvision.datasets(...), batch_size=BATCH_SIZE)
model = MLP()

# Train Loop
criterion, opt = nn.CrossEntropyLoss(), optim.AdamW(model.parameters())
for (inputs, labels) in data_loader:
    loss = criterion(model(inputs), labels)
    loss.backward(); opt.step(); opt.zero_grad()
(Distributed) Data Parallelism — Implementation

GPT-2 Training Clock

7.2 D — 16 GPUs w/ Data Parallelism (DDP)

Auto-Partitions Data across Processes

Simple Wrapper around nn.Module()

Nifty Utility → Spawns Processes

from torch.nn.parallel import DistributedDataParallel as DDP
from torch.utils.data.distributed import DistributedSampler

BATCH_SIZE, WORLD_SIZE = 128, 8  # World Size == # of GPUs

class MLP(nn.Module):
    def __init__(self, n_classes, mnist_dim, hidden, int):
        super().__init__()
        self.mlp = nn.Sequential(
            nn.Linear(mnist_dim, hidden),
            nn.ReLU(),
            nn.Linear(hidden, hidden),
            nn.ReLU(),
            nn.Linear(hidden, n_classes)
        )

    def forward(self, x):
        return self.mlp(x)

# Main Code
train_set = torchvision.datasets(...)
dist_sampler = DistributedSampler(dataset=train_set)
dataloader = DataLoader(train_set, sampler=dist_sampler, batch_size=BATCH_SIZE // WORLD_SIZE)

model = DDP(MLP(),
device_ids=[os.environ['LOCAL_RANK']],
output_device=os.environ['LOCAL_RANK'])
Important — Memory Footprint of Training?

GPT-2 Training Clock

7.2 D — 16 GPUs w/ Data Parallelism (DDP)

Standard (Float 32) Memory Footprint
[Excludes Activations + Temporary Buffers]

- 32b Parameters
- 32b Gradients
- 32b Parameter Copies
- 32b Momentum
- 32b Variance

Model

Optimizer

Lower Bound on “Static” Memory (w/ Adam):
= # Parameters * 20 Bytes

Activation Memory >> Static Memory

Training Implications

- 1B Parameters —> 18 GB (~31 GB w/ BSZ = 1)
- 175B Parameters —> 3 TB (w/o activations!)

Facts about Floating Points

- Float32 — Standard defined in IEEE-754
  - Sign (1) — Exponent (8) — Significand (23)
  - Wide Range —> up to 1e38

< Do we need *all* 32 bits? >

Mixed Precision Training

GPT-2 Training Clock

- 7.2 D — 16 GPUs w/ Data Parallelism (DDP)
- 6.01 D — 16 GPUs w/ DDP, FP16

Mixed Precision (FP16) Memory Footprint
[Excludes Activations + Temporary Buffers]

- **16b Parameters**
- **16b Gradients**

Model

- **32b Parameter Copies**
- **32b Momentum**
- **32b Variance**

Optimizer ~Adam

Lower Bound on “Static” Memory (w/ Adam):

= # Parameters * 16 Bytes

Activation Memory —> halved!

Hmm… Optimizer Memory?
FP16 does not mean *everything* is FP16.
Real Gain: NVIDIA Tensor Core Speedup!

Eliminate Redundancies —> ZeRO

GPT-2 Training Clock

- 7.2 D — 16 GPUs w/ Data Parallelism (DDP)
- 6.01 D — 16 GPUs w/ DDP, FP16
- 3.37 D — 16 GPUs w/ DDP, FP16, ZeRO

Punchline: “Shards” Memory by # of GPUs!

**Standard Data Parallelism**
“Replicate everything but the data!”

- $2\Psi$ Bytes
  - Model
  - Gradients [Entire Model]
  - Optimizer States [Entire Model]

- $2\Psi$ Bytes [+ Buffers]
  - Model
  - Gradients [Entire Model]
  - Optimizer States [Entire Model]

- $12\Psi$ Bytes
  - GPU 1
  - Optimizer States [Entire Model]

- GPU 2
  - Optimizer States [Entire Model]

**ZeRO Data Parallelism**
“Replicate only what you need”

- $2\Psi$ Bytes
  - Model
  - Gradients [Layers 1-6]
  - Opt. States [Layers 1-6]

- $(2\Psi / W)$ Bytes [+ Buffers]
  - Model
  - Gradients [Layers 7-12]
  - Opt. States [Layers 7-12]

- $(12\Psi / W)$ Bytes
  - GPU 1
  - Opt. States [Layers 7-12]

- GPU 2
  - Opt. States [Layers 7-12]

$\Psi = \# \text{ of Parameters}$

Alas — Hitting a (Communication) Wall

Problem — At some point, communication cost between nodes is too much!

Answers:

Exploit Matrix Multiplication…

Schedule Backwards Pass Wisely…

< Harder to implement, model-specific… still miles to go! >
Part III: Fine-Tuning and Inference

“It’s such a happiness, when good people get together.”
— Jane Austen, *Emma*
Tools for Training —> Tools for Fine-Tuning

Silver Lining — Learning to scale training —> informs fine-tuning & inference!

ZeRO Data Parallelism

ZeRO Infinity —> CPU/NVMe Offloading

Mixed Precision (FP16)

8-Bit Quantization

Teaser for Later —> Parameter-Efficient Fine-Tuning

LoRA (Low-Rank Adaptation)

adaLN (Adapted LayerNorm)

Reference: https://github.com/huggingface/peft

...and more!
That’s all Folks!

“This wind, it is not an ending…”
— Robert Jordan, *A Memory of Light*