Inteligência Artificial 2

Aula 23: Modificando um Modelo Gigante (LoRA)

Prof. Aléssio Miranda Júnior
alessio@cefetmg.br

CEFET-MG - Campus Timoteo
Dep. Engenharia de Computação

Inteligência Artificial 2

Sumário

Onde Paramos?

No Módulo III, construímos um **produto RAG completo** (API → RAG → Spring AI → Front → Resiliência):

- Consumimos LLMs via API, configuramos VectorStore, criamos front-end.
- Mas usamos modelos prontos "de prateleira" — GPT-4, Gemini.

A Virada do Módulo IV

E se você pudesse **ensinar** o modelo a falar com a personalidade da **sua empresa**? Isso é **Fine-Tuning**.

Objetivos da Sessão

Ao final desta aula, você será capaz de:

1. **Explicar** a diferença entre Fine-Tuning Full vs LoRA/QLoRA.
2. **Calcular** a economia de parâmetros com LoRA ($r \ll d$).
3. **Configurar** o Hugging Face Trainer com quantização 4-bit.
4. **Interpretar** a curva de Loss como prova de aprendizado.
5. **Executar** inferência com o modelo adaptado.

Evolução dos Modelos e Custo Computacional

- O tamanho dos *Large Language Models* (LLMs) cresceu exponencialmente. O GPT-3 possui 175 bilhões de parâmetros; modelos mais recentes ultrapassam trilhões na arquitetura MoE.
- **Custo de Treinamento Base (Pre-training):** Estima-se que treinar um modelo de ponta do zero exija $\mathcal{O}(10^{22})$ a $\mathcal{O}(10^{24})$ FLOPs.
- Financeiramente, isso representa dezenas de milhões de dólares apenas em eletricidade e aluguel de infraestrutura de GPUs (*clusters* de H100s).
- A ideia de que desenvolvedores e pesquisadores independentes pudessem criar suas próprias Inteligências Artificiais parecia um mito inatingível.

A Diferença Entre Criar e Adaptar

Pre-training (Treinamento Base)

- Ensinar do absoluto zero.
- Objetivo: Modelagem de linguagem causal (*next-token prediction*).
- Corpora: Trilhões de *tokens* abrangendo toda a internet (ex: Common Crawl, Wikipedia, ArXiv).
- Saída: Um modelo "fundacional" que sabe prever textos, mas não necessariamente responde como um assistente.

Fine-Tuning (Ajuste Fino)

- Adaptar um modelo fundacional.
- Objetivo: Especialização em uma tarefa, domínio ou formato de saída.
- Corpora: Milhares ou mesmo dezenas de exemplos altamente curados.
- Saída: Um assistente jurídico, um programador Python especializado ou um tradutor dialetal.

O Gargalo de Memória no Full Fine-Tuning

Ao tentar realizar um ajuste fino completo (*Full Fine-Tuning*) atualizando todos os pesos, esbarramos no limite de memória da GPU (VRAM):

- Considere um modelo de 8 bilhões de parâmetros (8×10^9) em formato BF16 (2 bytes por parâmetro).
- **Pesos do Modelo:** $8 \times 10^9 \times 2 = 16$ GB.
- **Gradientes:** Precisam da mesma precisão dos pesos $\rightarrow$ 16 GB.
- **Otimizador (ex: AdamW):** Mantém o momento de primeira e segunda ordem em FP32 (4 bytes cada) $\rightarrow 8 \times 10^9 \times 8 = 64$ GB.
- **Ativações:** *Memory footprint* da passagem *forward* dependente do tamanho do *batch* e do comprimento do contexto.

Conclusão: Treinar um "pequeno" modelo de 8B exigiria mais de **100 GB** de VRAM, impossibilitando o uso de GPUs domésticas (ex: RTX 4090 de 24 GB).

PEFT: Parameter-Efficient Fine-Tuning

A solução para a barreira de VRAM é o **PEFT**. A premissa central é congelar os pesos originais do modelo e treinar apenas um subconjunto minúsculo de parâmetros auxiliares. Várias técnicas sob o guarda-chuva PEFT surgiram:

- **Adapters:** Adição de pequenas sub-redes entre as camadas de *Transformer*. Aumentam a latência na inferência.
- **Prefix Tuning:** Adição de *tokens* contínuos treináveis nos *keys* e *values* da atenção.
- **Prompt Tuning:** Treinar apenas o *embedding* de um *prompt* fixo (*soft prompt*).

A que dominou a indústria, contudo, foi o **LoRA (Low-Rank Adaptation)**, que oferece alta qualidade sem latência de inferência adicional.

A Hipótese da Dimensão Intrínseca

O **LoRA** se fundamenta no estudo de Aghajanyan et al. (2020), que demonstrou o conceito de **Dimensão Intrínseca**:

> *"Modelos superparametrizados residem num espaço de parâmetros gigantesco, mas as mudanças que os levam a aprender novas tarefas estão contidas em um subespaço de dimensionalidade muito menor."*

Em termos de álgebra linear:

- A matriz de atualização dos pesos ΔW durante o *fine-tuning* possui baixo posto (*low-rank*).
- Não precisamos de uma matriz *full-rank* imensa para mapear as modificações; uma representação comprimida é matematicamente suficiente.

LoRA: A Matemática da Adaptação

Considere uma camada linear do modelo original com pesos $W_0 \in \mathbb{R}^{d \times k}$. Na inferência, a transformação de uma entrada x é:

$$h = W_0 x$$

No *full fine-tuning*, aprendemos um ΔW do mesmo tamanho, atualizando os pesos para $W_0 + \Delta W$, resultando em:

$$h = (W_0 + \Delta W)x = W_0 x + \Delta W x$$

O LoRA restringe a atualização ΔW forçando uma decomposição de matrizes de baixo posto (*rank* r, onde $r \ll \min(d, k)$):

$$\Delta W = BA$$

Onde $B \in \mathbb{R}^{d \times r}$ e $A \in \mathbb{R}^{r \times k}$.

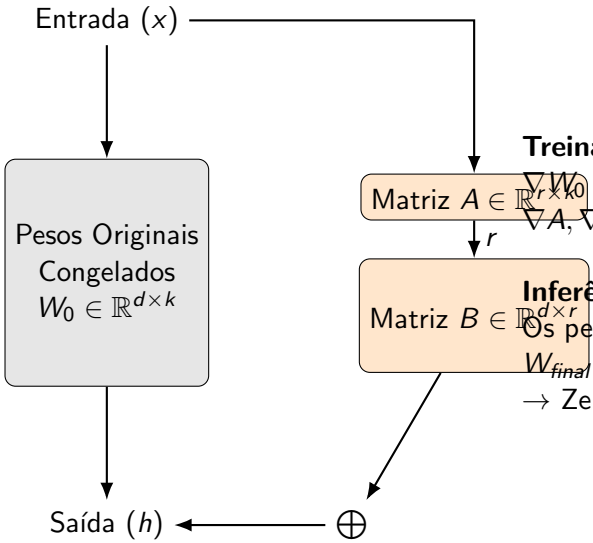
LoRA: Cálculo da Redução de Parâmetros

A decomposição $\Delta W = BA$ traz ganhos drásticos. **Exemplo prático:** Suponha a matriz de pesos W_q (*Query projection*) num Transformer de dimensões 4.096×4.096.

- **Full Fine-Tuning:** ΔW tem $4.096 \times 4.096 = \mathbf{16.777.216}$ parâmetros a treinar.
- **LoRA com Rank $r = 8$:**
  - Matriz $A \in \mathbb{R}^{8 \times 4096} = 32.768$ parâmetros.
  - Matriz $B \in \mathbb{R}^{4096 \times 8} = 32.768$ parâmetros.
  - Total: **65.536** parâmetros.

Resultado: Uma redução de **99,6%** no número de parâmetros treináveis daquela camada, o que colapsa o custo de memória do otimizador e gradientes.

LoRA: Diagrama Visual



Treinamento:

$\nabla W_0 = 0$ (Congelado)

$\nabla A, \nabla B \neq 0$ (Atualiza)

Inferência:

Os pesos podem ser "fundidos" (*merged*):

$W_{final} = W_0 + BA$

$\rightarrow$ Zero latência extra!

Hiperparâmetros Fundamentais do LoRA

Ao configurar o LoRA, as três variáveis mais críticas são:

1. **Rank (r):** Define a dimensionalidade das matrizes A e B. Valores comuns são $8, 16, 32, 64$. *Ranks* maiores podem capturar nuances mais complexas (como lógica formal de código), mas aumentam a memória. Tarefas de conversação funcionam bem com $r = 8$.

2. **LoRA Alpha (α):** Fator de escala. A saída da projeção LoRA é escalada por $\frac{\alpha}{r}$ antes de ser somada. **Regra de ouro empírica:** $\alpha = 2 \times r$ (ex: se $r = 16$, defina $\alpha = 32$). Isso garante estabilidade dos gradientes.

3. **Target Modules:** Em quais camadas aplicar o LoRA. Aplicar apenas no componente de Atenção (q_proj, v_proj) economiza muita memória, mas aplicar em todas as camadas lineares (MLP, o_proj) melhora o aprendizado sem custar tanto a mais.

QLoRA: A Revolução da Quantização (Dettmers et al. 2023)

O LoRA resolveu os gradientes e otimizadores, mas o **modelo base congelado** (W_0) ainda precisa residir na VRAM. Carregar um Llama-3 8B em BF16 consome 16 GB. A técnica

QLoRA (Quantized LoRA) introduziu inovações massivas:

- **4-bit NormalFloat (NF4):** O modelo base é quantizado teoricamente de forma ótima para uma precisão de **4 bits**.
- Os pesos bases (congelados) passam a consumir apenas $\sim$ 5 GB para um modelo de 8B.
- **Cálculo do Forward/Backward:** Os pesos em NF4 são **dequantizados de volta para BF16** apenas no instante do cálculo da matriz, operando junto com o LoRA em BF16.
- Permite o *fine-tuning* em GPUs amadoras (ex: Google Colab T4 com 15GB de VRAM ou placas gamers de 8GB).

O Hugging Face Hub

O **Hugging Face (HF)** atua como o "GitHub do Machine Learning", hospedando infraestrutura crítica de código aberto. Componentes vitais do ecossistema de *fine-tuning*:

- `transformers`: O coração. Fornece interfaces unificadas para download, *tokenization* e instanciamento de modelos (AutoModelForCausalLM, AutoTokenizer).
- `peft`: A biblioteca oficial que implementa LoRA, QLoRA, Prefix Tuning, gerenciando a injeção dos adaptadores dinamicamente nas camadas do PyTorch.
- `trl` (*Transformer Reinforcement Learning*): Fornece o `SFTTrainer` (*Supervised Fine-Tuning Trainer*), encapsulando loops de treino complexos.
- `datasets`: API de alta velocidade apoiada em Apache Arrow para carregamento *lazy* (sob demanda) de dados em disco e memória.

Unsloth: Velocidade e Otimização Extrema

A biblioteca **Unsloth** surgiu como um *game-changer* na execução prática de LoRA:

- **Kernels Customizados em Triton:** Unsloth reescreve as operações fundamentais do PyTorch usando Triton e xFormers, otimizando drasticamente os *memory acesses* a nível de hardware na GPU.
- **RoPE Otimizado:** O *Rotary Position Embedding* (RoPE), vital em Llamas e Mistrals, é calculado mais rapidamente, reduzindo a contenção térmica e a latência.
- **VRAM Menor:** Evita salvamento redundante do grafo computacional e gerencia *gradient checkpointing* inteligentemente.

Vantagem: Permite que o Llama-3-8B rode até **2x mais rápido** e usando **70% a menos de VRAM** comparado à stack padrão do Hugging Face.

Configurando o Ambiente com Unsloth

O Unsloth abstrai parte do carregamento complexo exigido pelo modelo base quantizado, usando sua própria classe otimizada `FastLanguageModel`.

```
from unsloth import FastLanguageModel

# Carregando o Llama-3 de 8 Bilhoes em 4 bits (QLoRA)
model, tokenizer = FastLanguageModel.from_pretrained(
    model_name = ``unsloth/llama-3-8b-bnb-4bit'',
    max_seq_length = 2048, # Contexto maximo a processar
    dtype = None,          # Auto-detectar (FP16 ou BF16)
    load_in_4bit = True,   # Ativa Quantizacao NF4
)
```

A flag `load_in_4bit=True` é a chave que executa o "milagre da VRAM", permitindo o carregamento massivo de parâmetros numa pequena GPU T4 do Colab.

Injetando os Adaptadores LoRA

Com o modelo base em memória, usamos a API do Unsloth para injetar as matrizes treináveis nas camadas alvo do *Transformer*.

```
# Adicionando Adaptadores LoRA ao Modelo Congelado
model = FastLanguageModel.get_peft_model(
    model,
    r = 16, # Rank do LoRA. Define o quao expressivo ele sera
    target_modules = [''q_proj'', ''k_proj'', ''v_proj'', ''o_proj''
        ,
                      ''gate_proj'', ''up_proj'', ''down_proj''],
    lora_alpha = 32,       # Escala (geralmente 2 * r)
    lora_dropout = 0,      # Suporta 0 otimizado no Unsloth
    bias = ''none'',         # Biases geralmente nao sofrem tuning
    use_gradient_checkpointing = ''unsloth'', # Otimizacao de GPU
)
```

Ao alvejar as projeções *MLP* (`gate`, `up`, `down`) e *Attention* (`q`, `k`, `v`, `o`), alcançamos uma qualidade de ajuste muito superior a mirar apenas na atenção.

O Formato de Dados para SFT (ChatML / Alpaca)

Para o *Supervised Fine-Tuning* (SFT), o modelo deve entender quando o usuário fala e quando o assistente responde. Os *tokens* especiais devem ser alinhados rigorosamente.

```
# Padronizando o dataset no formato Alpaca prompt
alpaca_prompt = ''''''Abaixo ha uma instrucao. Escreva a resposta.

### Instrucao:
{}

### Entrada (opcional):
{}

### Resposta:
{}''''''

def formata_prompt(exemplos):
    instrucoes = exemplos[''instruction'']
    entradas   = exemplos[''input'']
```

O Loop de Treinamento (SFTTrainer)

Em vez de escrever loops `for batch in dataloader` manualmente no PyTorch, a API `SFTTrainer` da biblioteca TRL gerencia perfeitamente a descida de gradiente.

```
from trl import SFTTrainer
from transformers import TrainingArguments

trainer = SFTTrainer(
    model = model,
    tokenizer = tokenizer,
    train_dataset = dataset,
    dataset_text_field = ''text'',
    max_seq_length = 2048,
    dataset_num_proc = 2,
    args = TrainingArguments(
        per_device_train_batch_size = 2,
        gradient_accumulation_steps = 4, # Acumulo para batch size
            virtual
        learning_rate = 2e-4,
```

Compreendendo o Treinamento do Modelo Causal

Durante a chamada de `trainer.train()`, o que ocorre por debaixo dos panos?

- **Modelagem Causal:** O *dataset* contendo "Instrução + Resposta" é passado inteiramente pelo modelo.
- **Cálculo de Perda (Loss):** A perda de Entropia Cruzada (*Cross-Entropy*) é calculada sobre a previsão do próximo *token*.
- **Masking do Prompt:** Geralmente, a perda calculada sobre a parte da *Instrução* não propaga gradientes, ou o modelo aprende forçadamente de ponta a ponta. O foco do ajuste fino é **condicionar** o modelo a prever a *Resposta dada* a instrução.
- O erro retropropaga pelas camadas gigantes congeladas e atinge as diminutas matrizes A e B, ajustando seus valores microscópicos a cada passo do AdamW.

Inferência e Deploy (Merging Adapters)

Uma vez treinado, como rodar o modelo? O Unsloth e Hugging Face permitem uso direto:

```
# Executando inferencia com o LoRA treinado na memoria
FastLanguageModel.for_inference(model) # Ativa velocidade nativa 2x
inputs = tokenizer(
[
    alpaca_prompt.format(``Qual a capital de Minas Gerais?'', ``'',
        ``'')
], return_tensors = ``pt'').to(``cuda'')

# Auto-regressao
outputs = model.generate(**inputs, max_new_tokens = 64)
print(tokenizer.batch_decode(outputs))
```

Para gerar o arquivo final `GGUF` (usado por ferramentas como Ollama), o peso treinado (BA) é somado algébricamente ao congelado (W_0): $W_{final} = W_0 + B \cdot A$
Isso é chamado de **Merging**.

Vantagens de Não Fazer o Merging Imediatamente

Curiosamente, pode ser vantajoso manter os adaptadores LoRA separados do modelo base num ambiente de produção com Múltiplos Usuários (**LoRA Multi-Tenant Serving**):

- Você hospeda um único Llama-3-8B de 16 GB na VRAM da GPU (modelo base).
- Um usuário do RH quer fazer consultas com jargões: Você ativa dinamicamente os adaptadores LoRA do RH (+60 MB na memória).
- Um desenvolvedor pede ajuda para Python: Você acopla o adaptador LoRA de programação (+60 MB na memória).
- A passagem computacional *forward* soma os desvios $B \cdot A$ dinamicamente apenas para o *batch* do cliente específico. O custo computacional agregado é irrelevante, mas economizou-se dezenas de GBs de memória!

Conclusão: Criando Seu Próprio Especialista

O *Fine-Tuning* transformou a forma como encaramos o *Machine Learning*.

- Através de **PEFT, LoRA e QLoRA**, não há limites práticos para criar Inteligências Artificiais super-especialistas.
- Bibliotecas de otimização de baixo nível como o **Unsloth** democratizaram a velocidade corporativa em infraestruturas caseiras e gratuitas.
- O poder analítico se tornou **local**. Dados confidenciais, prontuários médicos e códigos secretos corporativos podem ser usados em um LLM afiado *offline*, garantindo total privacidade, uma impossibilidade ao usar APIs comerciais baseadas em nuvem.

Laboratório 23: Llama Especialista (Unsloth)

Usaremos o **Google Colab** (GPU T4).
Você fará o carregamento massivo de um modelo de bilhões de parâmetros, configurará um ambiente quantizado e executará seu primeiro ciclo de treinamento com LoRA, ensinando ao LLM um formato de saída ou dialeto totalmente novo com um *dataset* que você personalizará em JSON.

Dúvidas?

Email: alessio@cefetmg.br
Aula: Segunda e Quarta-Feira, 7h as 8:40h (Sala 39)
WhatsApp: Dúvidas rápidas

Lembrete

Confira sua Issue no GitLab!

Próxima aula: Agentes Autônomos

100% da Aula 23 Concluída!