

**UNIVERSIDAD MAYOR, REAL Y PONTIFICA DE
SAN FRANCISCO XAVIER DE CHUQUISACA**

Facultad de Ciencias de la Tecnología



LoRA y QLoRA

ESTUDIANTE: Polo Orellana Brayan Simón

CARRERA: Ingeniería en Ciencias de la computación

MATERIA: Inteligencia Artificial 2

SIGLA: SIS-421

índice

Introducción	3
Fine-Tuning	4
LoRA (Low-Rank Adaptation)	4
Como funciona LoRA	5
QLoRA (Quantized Low-Rank Adaptation)	7
Como funciona QLoRA	8
Idea matemática y formula básica	9
Ejercicio Usando Pythorch:	10
LoRA	10
QLoRA	12
QLoRA usando PEFT	16
Análisis de Rendimiento	19
Ventajas y Desventajas	19
Ventajas	19
Desventajas	20
Referencias	21

Introducción

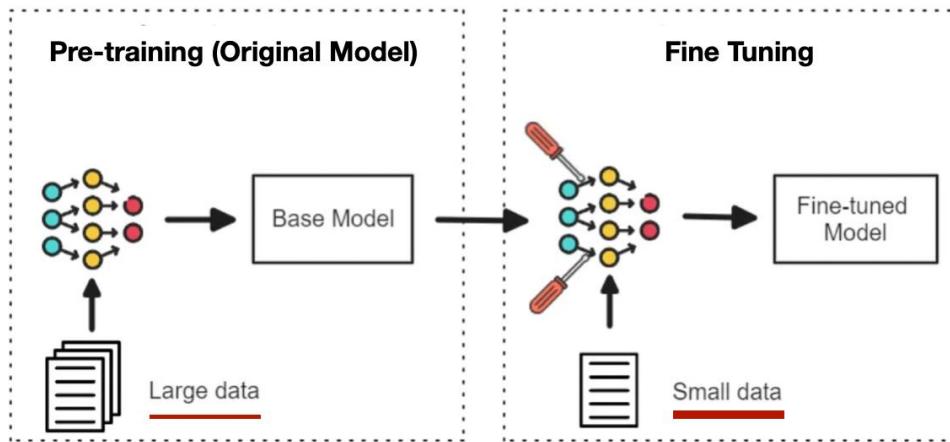
Recientemente, los *LLM* (*Large Language Models*) han demostrado un rendimiento sin precedentes en una amplia gama de tareas de comprensión de lenguaje y sirvió como base para sistemas de chat. La diversidad de aplicaciones en el mundo real exige un proceso en el que los LLM puedan ajustarse con precisión para adaptarse a diferentes escenarios y cuantificarse para su implementación en dispositivos periféricos (por ejemplo, teléfonos móviles). La clave resude en eliminar la pesada carga computacional que supone la gran cantidad de parámetros de los LLM.

Con el rápido crecimiento de los modelos de lenguaje grandes (LLMs), el fine-tuning de estos modelos se ha convertido en un desafío crítico. Los métodos tradicionales de fine-tuning demandan recursos computacionales significativos, haciéndolos imprácticos para muchas organizaciones. La Adaptación de Bajo Rango Cuantizada (QLoRA) es un enfoque nuevo que reduce las necesidades de memoria mientras conserva la calidad del modelo, permitiendo la afinación de LLMs en hardware de consumo.

Fine-Tuning

El fine-tuning (ajuste fino) es un proceso de reentrenamiento de un modelo de inteligencia artificial ya entrenado con un nuevo conjunto de datos más específico para adaptarlo a una tarea o dominio concreto. Este método permite mejorar la precisión y eficiencia del modelo en aplicaciones particulares, como el análisis de datos de una empresa, en lugar de entrenarlo desde cero.

Large Language Model



LoRA (Low-Rank Adaptation)

Es un algoritmo PEFT popular, propuesto **para ajustar matrices de bajo rango** y complementar los pesos preentrenados. A pesar de su rendimiento comparable con el **fine-tuning** de parámetros completos, el uso de memoria de LoRA sigue siendo elevado, especialmente cuando el LLM base es grande (ejemplo LLaMA).

LoRA de las Siglas **Low-Rank Adaptation (Adaptación de Bajo Rango)**, es un método de reentrenamiento que reutiliza un modelo de lenguaje básico para una tarea específica. Explora cómo LoRA te permite aprovechar la tecnología de un LLM y entrenarlo de forma rápida y eficiente para las necesidades de uno mismo.

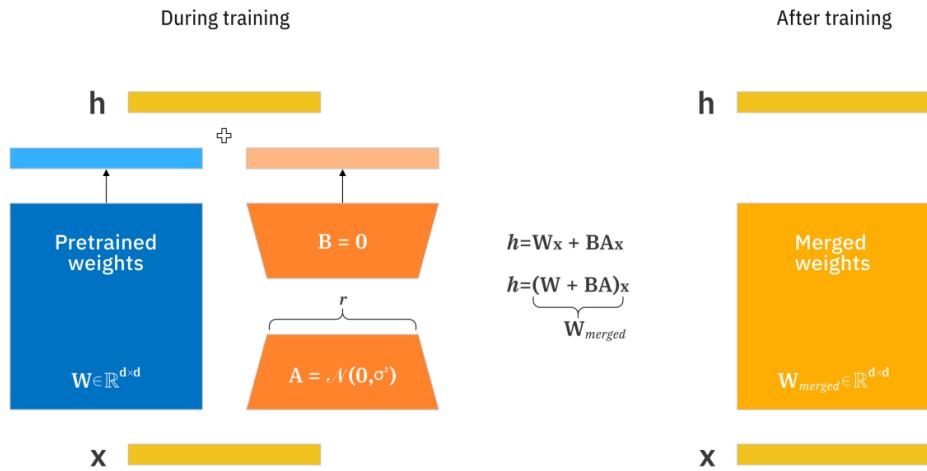
Su función principal es adaptar un modelo base a una tarea modificando los pesos de un subconjunto representativo de los parámetros del modelo, denominados adaptadores de bajo rango, en lugar de los pesos del modelo base durante el ajuste. En la Inferencia, los pesos de los adaptadores ajustados se suman a los pesos del modelo base para generar un resultado ajustado a la tarea.

Como funciona LoRA

En lugar de reentrenar todo el modelo, LoRA congela los pesos y parámetros originales del modelo tal como están. Luego, sobre este modelo original, añade una adición ligera llamada una matriz de bajo rango, que luego se aplica a nuevas entradas para obtener resultados específicos del contexto. La matriz de bajo rango ajusta los pesos del modelo original para que las salidas coincidan con el caso de uso deseado.

LoRA aprovecha el concepto de matrices de menor rango para hacer que el proceso de entrenamiento del modelo sea extremadamente eficiente y rápido. Tradicionalmente, la microajuste de LLMs requiere ajustar todo el modelo. LoRA se centra en modificar un subconjunto más pequeño de parámetros (matrices de menor rango) para reducir el sobrecosto computacional y de memoria.

El diagrama muestra cómo LoRA actualiza las matrices A y B para rastrear los cambios en los pesos preentrenados utilizando matrices más pequeñas de rango r . Una vez completada la capacitación de LoRA, los pesos más pequeños se fusionan en una nueva matriz de pesos, sin necesidad de modificar los pesos originales del modelo preentrenado.



LoRA funciona añadiendo pares de matrices de descomposición de rangos a las capas de transformador, centrándose generalmente en los pesos de atención. Durante la inferencia, estos pesos de adaptador pueden fusionarse con el modelo base, lo que elimina la sobrecarga de latencia adicional. LoRA es particularmente útil para adaptar modelos de lenguaje extensos a tareas o dominios específicos.

Las matrices son una parte importante de cómo funcionan los modelos de aprendizaje automático y las redes neuronales. Las matrices de rango bajo son más pequeñas y tienen muchos menos valores que las matrices más grandes o de rango más alto. No ocupan mucha memoria y requieren menos pasos para sumar o multiplicarse entre sí, lo que las hace más rápidas para que los ordenadores las procesen.

Una matriz de rango alto puede descomponerse en dos matrices de rango bajo, una matriz de 4×4 puede descomponerse en una matriz de 4×1 y una matriz de 1×4 .

$$\begin{array}{|c|c|c|c|} \hline -8 & -2 & -6 & 6 \\ \hline -4 & -1 & -3 & 3 \\ \hline 28 & 7 & 21 & -21 \\ \hline 24 & 6 & 18 & -18 \\ \hline \end{array} = \begin{array}{|c|} \hline -2 \\ \hline -1 \\ \hline 7 \\ \hline 6 \\ \hline \end{array} \times \begin{array}{|c|c|c|c|} \hline 4 & 1 & 3 & -3 \\ \hline \end{array}$$

LoRA añade matrices de bajo rango al modelo original de aprendizaje automático congelado. Las matrices de bajo rango se actualizan a través del descenso de gradiente durante el ajuste fino, sin modificar los pesos del modelo base. Estas matrices contienen nuevos pesos para aplicar al modelo al generar resultados. La matriz de cambio multiplicada se suma a los pesos del modelo base para obtener el modelo final ajustado. Este proceso altera las salidas que el modelo produce con mínima potencia de cómputo y tiempo de entrenamiento.

En esencia, LoRA mantiene el modelo original sin cambios y añade pequeñas partes variables a cada capa del modelo. Esto reduce significativamente los parámetros entrenables del modelo y la necesidad de memoria GPU para el proceso de entrenamiento, lo que es otro desafío significativo cuando se trata de ajustar o entrenar modelos grandes.

QLoRA (Quantized Low-Rank Adaptation)

Consiste en la cuantificación de parámetros, donde los pesos entrenados se cuantifican en enteros de bajo número de bits o números de punto flotante. Si bien estos métodos pueden reducir la carga computacional, a menudo presentan una precisión deficiente, especialmente cuando el ancho de bits de cuantificación es bajo.

La idea central detrás de QLoRA es cargar el modelo base preentrenado en un formato cuantizado, típicamente 4-bit, reduciendo drásticamente su requisito de memoria. Crucialmente, mientras el modelo base se mantiene en este formato de baja precisión, los adaptadores LoRA se entranan en una mayor precisión (por ejemplo, bfloat16). Este

enfoque permite importantes ahorros de memoria sin una pérdida catastrófica en el rendimiento, haciéndolo factible ajustar fino modelos masivos (por ejemplo, 65 mil millones de parámetros) en hardware con limitada VRAM, como un solo GPU de consumo

Como funciona QLoRA

a) Cuantización de pesos

Antes de entrenar, los pesos del modelo (W) se **cuantizan** usando un formato especial llamado **NF4 (Normalized Float 4)**.

NF4 es una representación en 4 bits optimizada estadísticamente que conserva la distribución de valores de los pesos originales, reduciendo la pérdida de precisión.

$$W_{\text{quant}} = \text{Quantize}(W_{\text{fp16}}, \text{format}=\text{NF4})$$

Esto reduce el uso de memoria **4 veces**.

b) Descuantización dinámica (Double Quantization)

Durante el entrenamiento, QLoRA **no descuantiza completamente los pesos**, sino que los mantiene en forma comprimida y solo los convierte parcialmente cuando es necesario para operaciones matriciales.

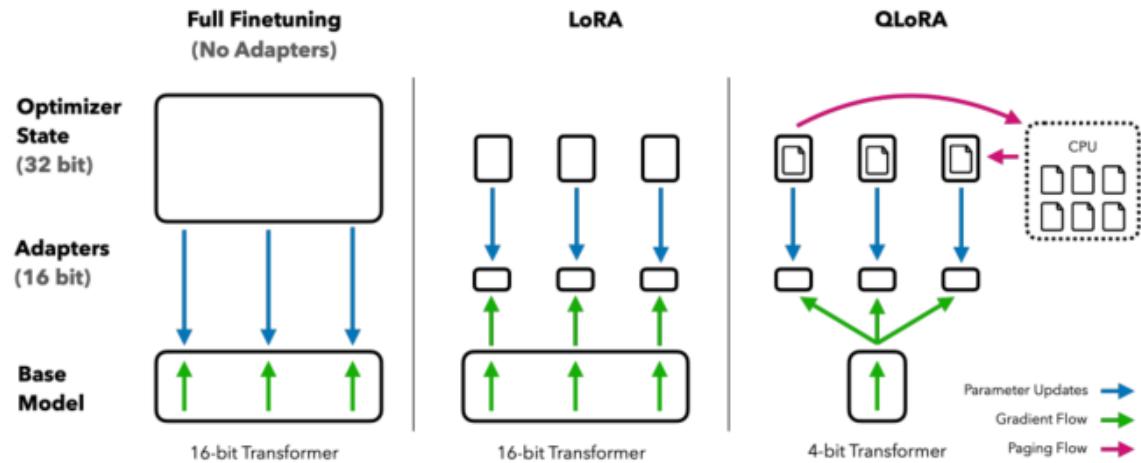
Esto se llama **double quantization** — se cuantizan tanto los pesos como los parámetros de cuantización, para ahorrar aún más memoria.

c) Inserción de adaptadores LoRA

Igual que en LoRA, no se modifican los pesos cuantizados. Se añaden matrices **A** y **B** de bajo rango (en FP16) para representar las actualizaciones.

$$y = (W_{\text{quant}} + \frac{\alpha}{r} BA)x$$

Solo **A** y **B** se entranan. El modelo base permanece cuantizado, congelado y eficiente.



Idea matemática y formula básica

Consideramos una capa lineal típica con peso $W \in \mathbb{R}^{d_{out} \times d_{in}}$. En el forward normal:

$$y = Wx \text{ (o en notación batch } y = XW^T\text{)}$$

En LoRA, la matriz de pesos se parametriza como:

$$W' = W + \Delta W \text{ donde } \Delta W = AB$$

Donde A es Matriz de proyección Descendente, donde reduce la dimensionalidad de la entrada x a un espacio de dimensión r .

Donde B es Matriz de proyección Ascendente, donde reconvierte esa representación comprimida al tamaño de salida original.

Donde α es Factor de escalado (ajusta magnitud de actualización) que es un hiperparametro no entrenable

con $A \in \mathbb{R}^{r \times d_{in}}$ $B \in \mathbb{R}^{d_{out} \times r}$, siendo r el rango bajo ($r \ll \min(d_{in}, d_{out})$). Durante el entrenamiento, los parámetros W permanecen **congelados**, mientras que solo se optimizan A y B .

El término de escalado α se introduce para controlar la magnitud de la actualización:

$$y = Wx + \frac{\alpha}{r} B(Ax).$$

De esta manera, el número de parámetros entrenables se reduce de $d_{out} \times d_{in}$ a $r(d_{in} + d_{out})$, logrando una reducción significativa en el consumo de memoria.

Donde r es el rango de la descomposición de baja dimensión usado para representar
 $W' = W + \Delta W$

Ejercicio Usando Pytorch:

LoRA

La clase **LoRALinear** implementa la técnica **LoRA (Low-Rank Adaptation)** para redes neuronales profundas.

Su propósito es **reducir los parámetros entrenables** durante el fine-tuning de modelos grandes, manteniendo la precisión y reduciendo el consumo de memoria.

```
# Implementación de LoRA
class LoRALinear(nn.Module):
    def __init__(self, orig_layer, r=4, alpha=16):
        super().__init__()
        self.orig_layer = orig_layer # Capa lineal original
        self.r = r # Rango bajo
        self.alpha = alpha # Escalado
        self.scaling = alpha / r # Factor de escalado
        self.lora_A = nn.Parameter(torch.zeros(orig_layer.in_features, r)) # Matriz A
        self.lora_B = nn.Parameter(torch.zeros(r, orig_layer.out_features)) # Matriz B

        nn.init.kaiming_uniform_(self.lora_A, a=math.sqrt(5)) # Inicializamos A con
        Kaiming
        nn.init.zeros_(self.lora_B) # Inicializamos B con ceros

    def forward(self, x):
        return self.orig_layer(x) + self.scaling * (x @ self.lora_A @ self.lora_B) # Aplicamos la formula de LoRA que es W + BA
```

En esta parte cargamos el modelo preentrenado, ajustando la salida a 5 clases que es nuestra cantidad de clases en el dataset. Pasando una capa Lineal a la clase

```
device = "cuda" if torch.cuda.is_available() else "cpu" # Dispositivo

# Carga del modelo VIT preentrenado
model = models.vit_b_16(pretrained=True)

# Ajuste de la capa final para 5 clases usando LoRA
num_classes = 5
orig_linear = nn.Linear(model.heads.head.in_features, num_classes) # Nueva capa lineal

model.heads.head = LoRALinear(orig_linear, r=4) # Reemplazamos la capa final con LoRA
model = model.to(device) # Movemos el modelo al dispositivo

print(model.heads.head.orig_layer.out_features) # Verificamos que la salida sea 5
```

Aquí cargamos el mismo modelo preentrenado, luego reemplazamos la cabeza con la misma estructura del entrenamiento el cual contiene 5 clases, luego cargamos los pesos entrenados:

```
# Carga del modelo VIT preentrenado
model = models.vit_b_16(pretrained=True)

# Reemplazar la cabeza con la misma estructura del entrenamiento (5 clases)
model.heads.head = LoRALinear(nn.Linear(768, 5), r=4).to(device)

# Cargar pesos entrenados
model.load_state_dict(torch.load("modelo_lora.pt", map_location=device))

# Mover el modelo al dispositivo y poner en modo evaluación
model = model.to(device)
model.eval()
print("☑ Modelo cargado correctamente con 5 clases.")
```

Predecimos una imagen:

```
from PIL import Image

image_path = r"D:\Ciencias\Dataset Residuos sólidos\data\val\Conervas de
cafe\0kwsf7ocvacy26xo.jpg"

transform = transforms.Compose([
    transforms.Resize((224,224)),
    transforms.ToTensor(),
    transforms.Normalize([0.5,0.5,0.5], [0.5,0.5,0.5])
])

image = Image.open(image_path).convert("RGB")
input_tensor = transform(image).unsqueeze(0).to(device)

# Mostrar imagen
plt.imshow(image)
plt.axis("off")
plt.show()

# Predicción
with torch.no_grad():
    outputs = model(input_tensor)
    _, pred = torch.max(outputs, 1)

print(f"⌚ Predicción: {train_dataset.classes[pred.item()]}")
```



💡 Predicción: Conservas de café

QLoRA

La clase `QLoRALinear` implementa la técnica **QLoRA (Quantized Low-Rank Adaptation)**, una extensión de **LoRA** diseñada para entrenar **modelos cuantizados** (por ejemplo, en 4 bits) de forma **eficiente y estable**.

Este método permite **fine-tuning de grandes modelos** en GPUs de consumo, reduciendo el uso de memoria sin sacrificar precisión.

```
class QLoRALinear(nn.Module):
    def __init__(self, in_features, out_features, r=4, alpha=8, quant_bits=4, bias=True):
        super().__init__()
        self.in_features = in_features # Número de características de entrada
        self.out_features = out_features # Número de características de salida
        self.r = r # Dimensión de los adaptadores LoRA
        self.alpha = alpha # Factor de escalado para LoRA
        self.scaling = alpha / r # Factor de escalado

        # Capa cuantizada base usando bitsandbytes, en 4 bits
        self.quant_linear = bnb.nn.Linear4bit(
            in_features,
            out_features,
            bias=bias,
            quant_type="nf4",
            compute_dtype=torch.float16 # Tipo de dato para cálculos
        )

        # Parámetros LoRA
```

```

        self.lora_A = nn.Parameter(torch.zeros(r, in_features)) # Matriz A de LoRA
        self.lora_B = nn.Parameter(torch.zeros(out_features, r)) # Matriz B de LoRA

        nn.init.kaiming_uniform_(self.lora_A, a=5**0.5) # Inicialización de A
        nn.init.zeros_(self.lora_B) # Inicialización de B

    def forward(self, x): # Propagación hacia adelante
        base_output = self.quant_linear(x) # Salida de la capa cuantizada base
        lora_output = (x @ self.lora_A.t() @ self.lora_B.t()) * self.scaling # Salida de
LoRA
        return base_output + lora_output # Suma de ambas salidas

```

Esta función carga el modelo con 5 clases usando el modelo ViT con pesos preentrenados, esto tomando en cuenta que congelamos todo el modelo

```

def setup_model_for_5_classes(r=4, alpha=8):
    device = "cuda" if torch.cuda.is_available() else "cpu"
    print(f"🔗 Usando dispositivo: {device}")

    # Cargar modelo base CON pesos preentrenados
    model = models.vit_b_16(weights=models.ViT_B_16_Weights.IMAGENET1K_V1)

    # CONGELAR todo el modelo base
    for param in model.parameters():
        param.requires_grad = False

    # Reemplazar la cabeza para 5 clases usando QLoRA
    num_classes = 5
    model.heads.head = QLoRALinear(
        in_features=768,
        out_features=num_classes,
        r=r,
        alpha=alpha,
        quant_bits=4
    )

    # Hacer que solo los parámetros LoRA sean entrenables
    for name, param in model.named_parameters():
        if 'lora' in name:
            param.requires_grad = True # Entrenable
        else:
            param.requires_grad = False # Congelado

    model = model.to(device)

    # Verificar parámetros
    total_params = sum(p.numel() for p in model.parameters())
    trainable_params = sum(p.numel() for p in model.parameters() if p.requires_grad)

    print(f"☑ Modelo configurado para {num_classes} clases")
    print(f"📊 Parámetros totales: {total_params:,}")
    print(f"📊 Parámetros entrenables: {trainable_params:,}")

```

```

print(f"📊 Porcentaje entrenable: {100*trainable_params/total_params:.4f}%")

return model, device

```

La siguiente función guarda el modelo:

```

def save_qlora_model(model, path="modelo_completo_qlora.pt"):
    """Guardar SOLO los parámetros LoRA, no toda la capa cuantificada"""
    # Guardar solo los parámetros LoRA y el estado del modelo
    save_dict = {
        'lora_A': model.heads.head.lora_A.data,
        'lora_B': model.heads.head.lora_B.data,
        'model_state': model.state_dict(), # Para compatibilidad
        'r': model.heads.head.r,
        'alpha': model.heads.head.alpha,
        'scaling': model.heads.head.scaling
    }

    torch.save(save_dict, path)
    print(f"☑️ Parámetros LoRA guardados en {path}")

```

La siguiente función carga el modelo:

```

def load_qlora_model(model_path="modelo_completo_qlora.pt"):
    """Cargar modelo con parámetros LoRA"""
    device = "cuda" if torch.cuda.is_available() else "cpu"
    print(f"🔗 Cargando modelo en {device}...")

    # Primero cargar los parámetros guardados
    checkpoint = torch.load(model_path, map_location=device, weights_only=False)

    # Crear modelo base
    model = models.vit_b_16(weights=models.ViT_B_16_Weights.IMAGENET1K_V1)

    # Configurar la cabeza QLoRA con los mismos parámetros
    r = checkpoint.get('r', 4)
    alpha = checkpoint.get('alpha', 8)

    num_classes = 5
    model.heads.head = QLoRALinear(
        in_features=768,
        out_features=num_classes,
        r=r,
        alpha=alpha,
        quant_bits=4
    )

    # Cargar parámetros LoRA
    model.heads.head.lora_A.data = checkpoint['lora_A']
    model.heads.head.lora_B.data = checkpoint['lora_B']

```

```

# Congelar todo excepto LoRA
for param in model.parameters():
    param.requires_grad = False

# Hacer que solo los parámetros LoRA sean entrenables
for name, param in model.named_parameters():
    if 'lora' in name:
        param.requires_grad = True

# Mover el modelo al dispositivo
model = model.to(device)
model.eval()

print("☑ Modelo QLoRA cargado correctamente")
print(f"📊 Parámetros LoRA: r={r}, alpha={alpha}")

return model, device

```

Despues del entrenamiento y uso, se puede mostrar un ejemplo de predicción:



💡 Predicción: Llantas
 📊 Confianza: 99.96%
 'Llantas'

QLoRA usando PEFT

Es fundamental para QLoRA ya que mantendremos el modelo base cuantizado (frozen) mientras solo se entrenan los adaptadores LoRA en precisión completa, logrando un balance óptimo entre eficiencia de memoria y calidad del fine-tuning.

En esta sección del código se está configurando y cargando un modelo de lenguaje grande (LLM) con cuantización de 4 bits para implementar QLoRA (Quantized Low-Rank Adaptation). Específicamente:

```
# Nombre del modelo TinyLlama
model_name = "PY007/TinyLlama-1.1B-Chat-v0.1"

# Cargar el tokenizador para el modelo
tokenizer = AutoTokenizer.from_pretrained(model_name)

# Cargar el modelo con cuantización en 4 bits
model = AutoModelForCausalLM.from_pretrained(
    model_name,
    load_in_4bit=True, # Habilitar carga en 4 bits
    device_map="auto", # Asignar automáticamente a los dispositivos disponibles
    torch_dtype=torch.float16, # Usar float16 para eficiencia
    bnb_4bit_compute_dtype=torch.float16, # Tipo de dato para cálculos
    bnb_4bit_use_double_quant=True, # Usar doble cuantización para mejor precisión
    bnb_4bit_quant_type="nf4" # Tipo de cuantización (Normal Float 4)
)
```

Configuración de los adaptadores LoRA

Se crea un objeto LoraConfig que define los parámetros clave para la adaptación de bajo rango:

```
# Configuración LoRA
lora_config = LoraConfig(
    r=8,                      # rango bajo para adaptadores
    lora_alpha=32,             # escala de los adaptadores
    target_modules=["q_proj", "v_proj"], # capas donde aplicar LoRA
    lora_dropout=0.1,           # dropout para regularización
    bias="none",               # no adaptar sesgos
    task_type="CAUSAL_LM" # tipo de tarea
)

# Aplicar LoRA al modelo cuantizado
model = get_peft_model(model, lora_config)
```

Para el dataset usamos un conjunto de datos que es una traducción al español de **alpaca_data_cleaned.json**, que es una traducción al español del famoso dataset Alpaca de Stanford. Este dataset contiene instrucciones y respuestas en formato conversacional, ideal para entrenar modelos de chat en español. Sacado desde **hugging-face** <https://huggingface.co/datasets/bertin-project/alpaca-spanish>

```
from datasets import load_dataset

# Dataset de prueba
dataset = load_dataset("bertin-project/alpaca-spanish")

# Usar solo el conjunto de entrenamiento
train_dataset = dataset["train"]

# Preprocesamiento de los datos
def preprocess(examples):
    # Formatear prompt + respuesta para cada ejemplo en el batch
    prompts = [f"### Human: {inst}\n### Assistant: {out}"
               for inst, out in zip(examples['instruction'], examples['output'])] # Crear
    prompts
    tokenized = tokenizer(prompts, truncation=True, padding="max_length", max_length=128)
# Tokenizar
    tokenized["labels"] = tokenized["input_ids"].copy() # Usar input_ids como labels
    return tokenized

train_dataset = train_dataset.map(preprocess, batched=True) # Preprocesar el dataset
train_dataset.set_format(type="torch", columns=["input_ids", "attention_mask", "labels"])
# Formatear para PyTorch
```

Para el entrenamiento:

```
from transformers import Trainer, TrainingArguments

training_args = TrainingArguments(
    output_dir="./tinyllama-lora", # directorio de salida
    per_device_train_batch_size=2, # tamaño de batch por dispositivo
    gradient_accumulation_steps=8, # acumular gradientes
    learning_rate=2e-4, # tasa de aprendizaje
    fp16=True, # usar precisión mixta
    logging_steps=10, # pasos de registro
    save_steps=200, # pasos para guardar el modelo
    save_total_limit=2, # límite total de modelos guardados
    max_steps=1000, # pasos máximos de entrenamiento
)
# Configurar el Trainer
trainer = Trainer(
    model=model, # el modelo a entrenar
    args=training_args, # argumentos de entrenamiento
    train_dataset=train_dataset # conjunto de datos de entrenamiento
)
# Iniciar el entrenamiento
```

```
trainer.train()
```

Para la prueba del modelo:

```
# Guardar adaptadores LoRA
model.save_pretrained("./tinyllama-lora")

# Cargar el modelo cuantizado y aplicar LoRA para inferencia
from peft import PeftModel

# Cargar el modelo cuantizado
model = AutoModelForCausalLM.from_pretrained(
    model_name, # nombre del modelo
    load_in_4bit=True, # cargar en 4 bits
    device_map="auto", # asignación automática de dispositivos
    torch_dtype=torch.float16, # usar float16
    bnb_4bit_compute_dtype=torch.float16, # tipo de dato para cálculos
    bnb_4bit_use_double_quant=True, # usar doble cuantización
    bnb_4bit_quant_type="nf4" # tipo de cuantización
)
# Cargar los adaptadores LoRA entrenados
model = PeftModel.from_pretrained(model, "./tinyllama-lora")
```

Instanciamos el Tokenizer y el prompt:

```
from transformers import AutoTokenizer

# Cargar el tokenizador
tokenizer = AutoTokenizer.from_pretrained(model_name)

# Crear un prompt de prueba
prompt = "### Human: Que es la IA.\n### Assistant:"
inputs = tokenizer(prompt, return_tensors="pt").to("cuda")
```

Realizamos el uso del modelo:

```
# Modo evaluación
model.eval()

# Generación de texto
with torch.no_grad():
    outputs = model.generate(
        **inputs,
        max_new_tokens=200,
        do_sample=True,
        top_p=0.9,
        temperature=0.7,
        pad_token_id=tokenizer.eos_token_id, # importante para modelos pequeños
```

```

        eos_token_id=tokenizer.eos_token_id,
    )

# Decodificar tokens
generated_text = tokenizer.decode(outputs[0], skip_special_tokens=True)
print("Resultado:")
print(generated_text)

```

Análisis de Rendimiento

4050 6GB de VRAM			
Tecnica	Epochas	Tiempo de entrenamiento	Uso de Memoria
LoRA	3	1hr	5.8 Gb
QLoRA	3	30 minutos	0.8 Gb

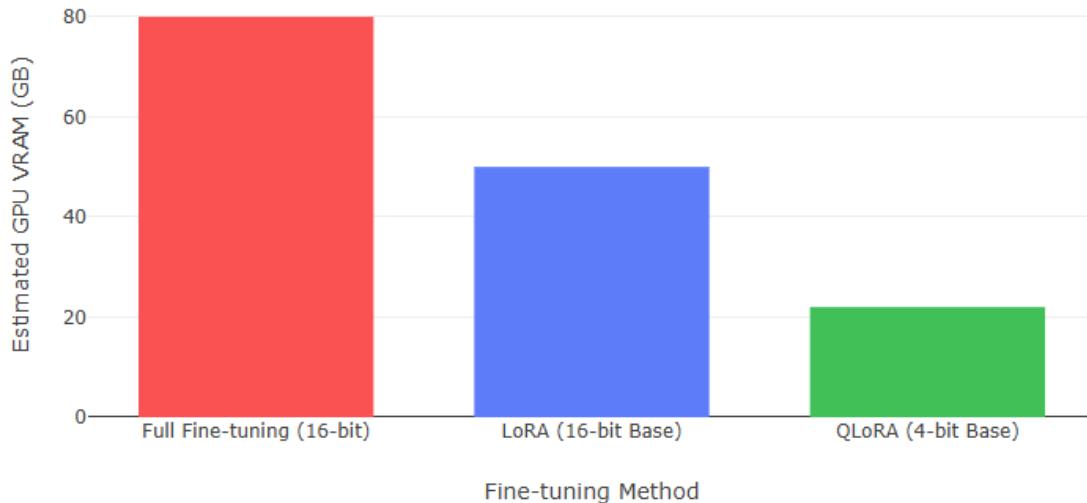
Haciendo un análisis se puede notar que usando QLoRA se puede notar que el uso de memoria se ha reducido en mas x8 de la parte que consume LoRA al momento de entrenar el modelo con solo 3 epochas, también se puede mencionar que el tiempo de entrenamiento se ha reducido en mas de la mitad, esto es gracias a que QLoRA Cuantiza los pesos del modelos a 4bits, mientras que solo entrena las matrices A y B.

Ventajas y Desventajas

Ventajas

- **Reducción Masiva de Memoria:** QLoRA reduce drásticamente la VRAM necesaria para el ajuste fino, permitiendo la adaptación de modelos muy grandes (30B, 65B+) en hardware accesible (por ejemplo, GPUs con 24GB o 48GB de VRAM).
- **Preservación del Rendimiento:** A pesar de la cuantización de 4 bits del modelo base, QLoRA logra típicamente un rendimiento muy cercano al ajuste fino completo de 16 bits o a 16-bit LoRA en muchas métricas y tareas. El uso de NF4 y el cálculo en bfloat16 son esenciales para esto.
- **Compatibilidad:** Se integra bien con ecosistemas existentes como las bibliotecas Transformers y peft de Hugging Face, a menudo requiriendo solo cambios en la configuración.

Approx. Memory Usage Comparison (Example 65B Model)



Desventajas

- **Reducción Potencial de Troughput:** La desquantización en tiempo real durante la pasada hacia adelante añade sobrecarga computacional. Si bien QLoRA ahorra memoria, podría resultar en pasos de entrenamiento más lentos (menor throughput) en comparación con el estándar LoRA si la memoria no era el cuello de botella principal.
- **Diferencias Menores de Rendimiento:** Si bien generalmente se logra un rendimiento comparable, podrían haber caídas ligeras en ciertos métricos en comparación con los métodos de 16 bits en ciertas tareas.
- **Dependencias de Implementación:** Requiere bibliotecas específicas como bitsandbytes para la cuantización NF4, DQ y Optimizadores Página.

Referencias

- Bamoria, H. (2024, Oct 07). *QLoRA: Quantized Low-Rank Adaptation*. Retrieved from Medium: <https://medium.com/athina-ai/qlora-quantized-low-rank-adaptation-5700adfce19a>
- coursera. (2025, Mayo 05). *Low Rank Adaptation: Reduce the Cost of Model Fine-Tuning*. Retrieved from coursera: <https://www.coursera.org/articles/low-rank-adaptation>
- IBM. (2025, 03 26). *Quantized low-rank adaptation (QLoRA) fine tuning*. Retrieved from IBM: <https://www.ibm.com/docs/en/watsonx/w-and-w/2.1.0?topic=tuning-qlora-fine>
- Joshua Noble. (2021). *What is LoRA (low-rank adaption)?* Retrieved from IBM: <https://www.ibm.com/think/topics/lora>
- Quantized Low-Rank Adaptation (QLoRA)*. (2025). Retrieved from apxml: <https://apxml.com/courses/fine-tuning-adapting-large-language-models/chapter-4-parameter-efficient-fine-tuning/qlora-introduction>
- Xu, Y. (2023, Sep 26). *QA-LoRA: Quantization-Aware Low-Rank Adaptation of Large Language Models*. Retrieved from arxiv.org: <https://arxiv.org/abs/2309.14717>