IIC3670 Procesamiento de Lenguaje Natural

https://github.com/marcelomendoza/IIC3670

- LLM ALIGNMENT -

LLM alignment

- Podemos alinear un LLM en base a un dataset de instrucciones (por ejemplo ALPACA).
- Le mostramos al LLM el prompt (instrucción, input, input_ids, …) y medimos el error comparando la salida generada con el output del dataset.
- Dependiendo del tipo de instrucción (summarization, extraction, machine translation, …) tenemos tipos de downstream tasks. Tenemos entonces datasets con secuencias de tokens por tarea:

$$\mathcal{Z} = \{(x_i, y_i)\}_{i=1,..,N}$$

prompt output

- Durante el alignment, el modelo es inicializado con pesos pre-entrenados Φ_0 y se actualizan usando gradiente descendente:

$$\Phi_0 + \Delta\Phi$$

de manera que el gradiente maximiza:

$$\max_{\Phi} \sum_{(x,y)\in\mathcal{Z}} \sum_{t=1}^{|y|} \log\left(P_\Phi(y_t|x, y_{<t})\right)$$ ← Causal LM

LLM alignment

- Una de las dificultades es que para cada downstream task aprendemos un conjunto diferente de parámetros cuya dimensión es:

$$|\Delta\Phi| = |\Phi_0|.$$

- Es decir, si el LLM es grande, el # parámetros que hay que actualizar en cada step de GD pueden ser infactibles (ej. 175B para GPT-3).
- Low Rank Adapters (LoRa) adoptan un enfoque eficiente para alignment, donde el incremento $\Delta\Phi = \Delta\Phi(\Theta)$ es codificado en un conjunto de parámetros $|\Theta| \ll |\Phi_0|$.

- Luego, el gradiente se calcula optimizando sobre Θ:

$$\max_{\Theta} \sum_{(x,y)\in\mathcal{Z}} \sum_{t=1}^{|y|} \log\left(p_{\Phi_0+\Delta\Phi(\Theta)}(y_t|x, y_{<t})\right)$$

reparametrización

LLM alignment

- Low Rank Adapters (LoRa): pretrained LM tienen una dimensión intrínseca mucho menor que la expresada en la red. En consecuencia, pueden aprender eficientementeusando una *random projection* a un subespacio más pequeño (ver Aghajanyan 2020).
- LoRa expresa el update de GD usando una descomposición low rank:

$$W_0 + \Delta W = W_0 + BA$$

$W_0 \in \mathbb{R}^{d \times k}$ ← frozen

$B \in \mathbb{R}^{d \times r}$ ← learnable

$A \in \mathbb{R}^{r \times k}$

$r \ll \min(d, k)$

- Durante el entrenamiento, W_0 se congela (no recibe updates).
- En el forward, todos los parámetros se multiplican por la entrada:

$$h = W_0 x + \Delta W x = W_0 x + BAx$$

- *A* tiene una inicialización Gaussiana y *B* se inicializa en 0, de manera que $\Delta W = BA$ es 0 al inicio del entrenamiento.

Armen Aghajanyan, Luke Zettlemoyer, and Sonal Gupta. Intrinsic Dimensionality Explains the Effectiveness of Language Model Fine-Tuning. arXiv:2012.13255 [cs], December 2020. URL: http://arxiv.org/abs/2012.13255

LLM alignment

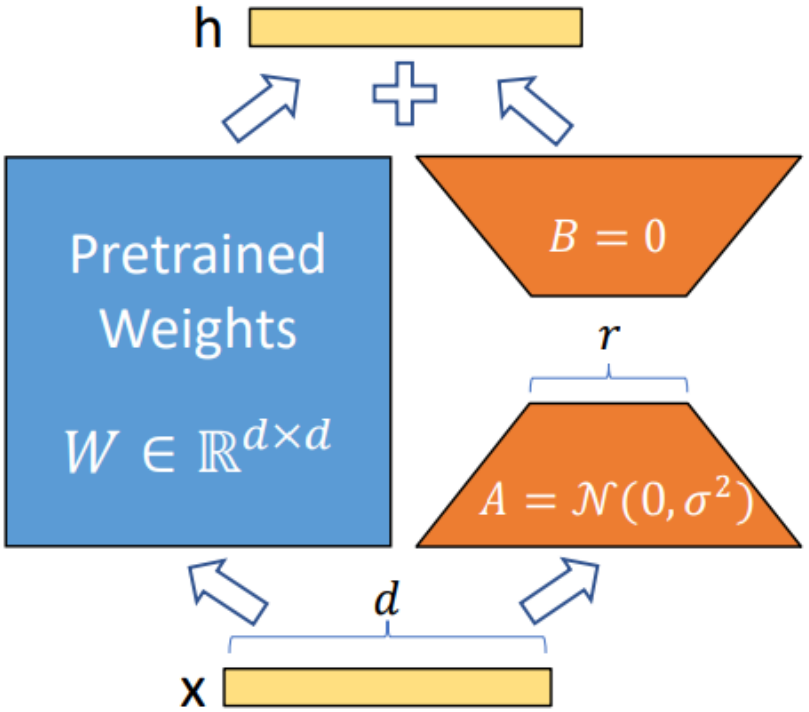
- Veamos el forward en código para entender la diferencia de LoRa con un formard convencional:

```
def regular_forward_matmul(x, W):
    h = x @ W
return h

def lora_forward_matmul(x, W, W_A, W_B):
    h = x @ W  # regular matrix multiplication
    h += x @ (W_A @ W_B) # updated equation
return h
```

LLM alignment

- Low Rank Adapters (LoRa) (ver Hu et al. 2022):

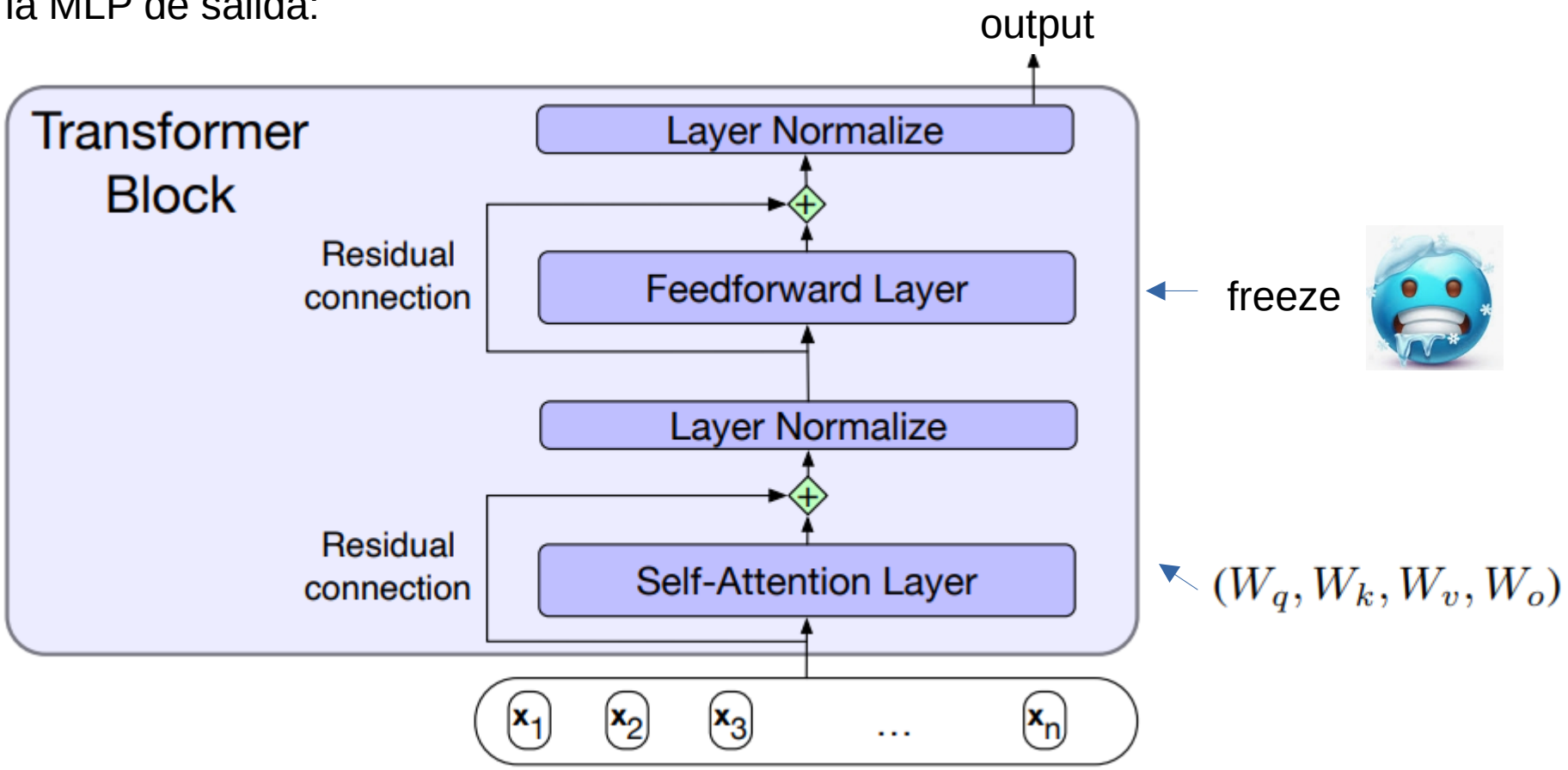


- Notar que el modelo en producción requiere almacenar $W = W_0 + BA$. Si queremos recuperar W_0, basta con restar *A* y *B*.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen:LoRA: Low-Rank Adaptation of Large Language Models. ICLR 2022

LLM alignment

- LoRa sólo adapta los pesos de la capa de atención y hace un freeze en la MLP de salida:



- Beneficios de LoRa: Uso de memoria en GPT-3 175B desde 1.2TB a 350GB debido al freeze. Luego, aplicando LoRa con r=4 y adaptación sólo a query y value, baja de 350 GB a 35 MB de memoria. De esa manera requiere – GPUs.

LLM alignment

- PEFT (Parameter Efficient Fine Tuning) permite usar el adapter de LoRa en transformers:

```
from peft import LoraConfig, get_peft_model
config = LoraConfig(
    r=16, # rank
    lora_alpha=16, # lora scaling factor
    target_modules=["query", "value"], # modules to apply LoRA
    lora_dropout=0.1, # dropout
    bias="none",
    modules_to_save=["classifier"], # additional modules to save
)
model = AutoModelForSequenceClassification.from_pretrained("bert-base-uncased")
lora_model = get_peft_model(model, config)
```

- El factor de escala (~ learning rate de LoRa) se usa generalmente para que coincida con *r*.
- El bias es el de la capa lineal.

LLM alignment

- Mientras que LoRa ayuda a reducir los requerimientos de almacenamiento, aún requiere cargar los pesos del modelo en memoria, y por tanto, requiere de GPU con mucha memoria. Este requerimiento se puede relajar usando cuantización.

- Por defecto, los pesos se almacenan en FP32 (32 bits). Con cuantización podemos almacenar usando menos bits (ej.: 8 ó 4). Esto implica una enorme reducción en consumo0 de memoria.

- QloRa trabaja con un LLM a 4 bits (ó 8) y luego hace entrenamiento con LoRa a 32 bits usando de-cuantización. Es decir, sólo almacenamos los LoRa adapters en 32, el resto de los pesos van a 4 bits.

- La idea de cuatización a 4 bits en simple (no exactamente lo que hace QloRa pero bueno para entender la idea):

  - 4 bits es = a 16 niveles equi-espaciados en el intervalo [-1, 1].
  - Dividimos el intervalo en 16 segmentos: 10^{th} valor

[-1.0, -0.86, -0.73, -0.6, -0.46, -0.33, -0.2, -0.06, 0.06, **0.2**, 0.33, 0.46, 0.6, 0.73, 0.86, 1.0]

- Supongamos que tenemos el peso 0.23456 en FP32. Su valor más cercano en los segmentos es **0.2**. En 4 bits, se almacena el 10 en 4 bits (es decir, 8+2 = 1010), ya que 0.2 es el 10^{th} valor en 16 segmentos.

LLM alignment

- Si queremos usar el peso en 4 bits para un cómputo, lo de-cuantizamos a FP32. Es decir, el 10 (1010 en 4 bits), pasa a 0.2.
- El error de de-cuantización es 0.23456 – 0.2 = 0.03456.

- QLoRa (ver Dettmers et al., 2023) en realidad hace algo un poco más complejo, ya que determina los segmentos en base a los datos usando cuartiles, de manera de bajar el error de de-cuantización.

- Cuantización en QLoRa: 1) Estima los $2^k + 1$ de una *N(0, 1)* para obtener un *k*-bit cuantil, 2) Toma estos valores y los normaliza (reescala) a [-1, 1]. Los cuantiles se calculan según:

$$q_i = \frac{1}{2}\left(Q_X\left(\frac{i}{2^k+1}\right) + Q_X\left(\frac{i+1}{2^k+1}\right)\right),$$

donde $Q_X(\cdot)$ es la función de cuantiles de la *N(0, 1)*.

- En rigor, usan doble cuantización para reducir el error, es decir, después de cuantizar el 10 a 0.2, cuantizan el residuo (0.03456). Por eso QloRa usa 8 bits.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, Luke Zettlemoyer: QLoRA: Efficient Finetuning of Quantized LLMs. NeurIPS 2023

LLM alignment

- QLoRa opera sobre la ecuación de update de LoRa en base a las siguientes cuantizaciones:

Normal float

$$\mathbf{Y}^{\text{BF16}} = \mathbf{X}^{\text{BF16}}\text{doubleDequant}(c_1^{\text{FP32}}, c_2^{\text{k-bit}}, \mathbf{W}^{\text{NF4}}) + \mathbf{X}^{\text{BF16}}\mathbf{L}_1^{\text{BF16}}\mathbf{L}_2^{\text{BF16}},$$

LLM a NF4 *B* *A*

Constantes para reconstruir a BF16

Block-wise float

donde:

$$\text{doubleDequant}(c_1^{\text{FP32}}, c_2^{\text{k-bit}}, \mathbf{W}^{\text{k-bit}}) = \text{dequant}(\text{dequant}(c_1^{\text{FP32}}, c_2^{\text{k-bit}}), \mathbf{W}^{\text{4bit}}) = \mathbf{W}^{\text{BF16}}$$

LLM alignment

- QLoRa está disponible en la librería bitsandbytes, para lo cual debemos configurar algunos parámetros antes de usar el trainer de transformers:

```
from transformers import BitsAndBytesConfig
nf4_config = BitsAndBytesConfig(
   load_in_4bit=True,
   bnb_4bit_quant_type="nf4",
   bnb_4bit_use_double_quant=True,
   bnb_4bit_compute_dtype=torch.bfloat16
   #FP16 for faster tuning. You can also choose FP32 for higher precision
)
model_nf4 = AutoModelForCausalLM.from_pretrained(model_id, quantization_config=nf4_config)
```

Cuantización a NF4

Doble cuantización

Forward a BF16 (puede ser BF32)

Repositorio de QLoRa en: https://github.com/artidoro/qlora