LLMs: Fine-tuning & Optimisation

Thomas Gerald

November 3, 2025

https://thomas-gerald.fr/EcAuTAL/index.html

Questions :

- How to deal with my new task ?

Questions :

- How to deal with my new task ?
- What architecture can I use ?

Questions :

- How to deal with my new task ?
- What architecture can I use ?
- What libraries is relevant for my problem(s) ?

Questions :

- How to deal with my new task ?
- What architecture can I use ?
- What libraries is relevant for my problem(s) ?

Adaptation

- Modification of pretrained models (Large Language Model)

Fine-tuning or adapting LLMs: Introduction

Questions :

- How to deal with my new task ?
- What architecture can I use ?
- What libraries is relevant for my problem(s) ?

Adaptation

- Modification of pretrained models (Large Language Model)
- With constraint(s) (time, memory, number of annotated data)

Fine-tuning or adapting LLMs: Introduction

Questions :

- How to deal with my new task ?
- What architecture can I use ?
- What libraries is relevant for my problem(s) ?

Adaptation

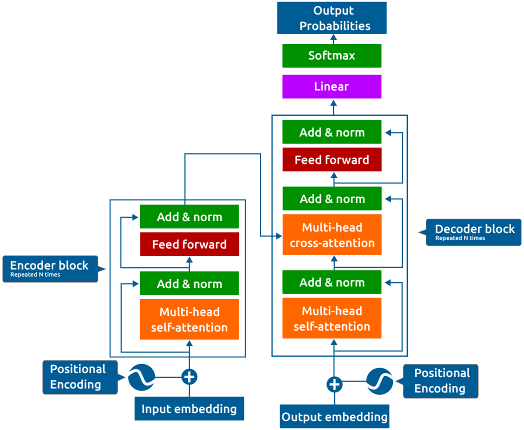
- Modification of pretrained models (Large Language Model)
- With constraint(s) (time, memory, number of annotated data)

What are the pre-trained LLMs ?

The tranformer (a reminder)

Today's Language Models

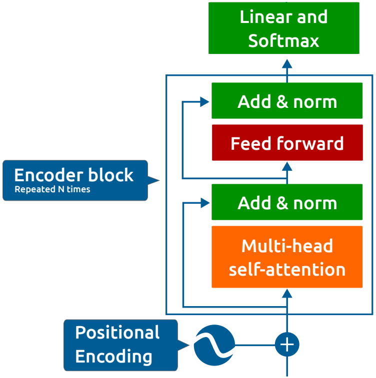
- Using the transformer architecture (neural networks)
  → "Attention is all you need", A. Vaswani, NIPS 2017
- Different architectures
  - Encoder Only (BERT)
  - Decoder Only (GPT)
  - Encoder-Decoder (T5)



This is the **Encoder** transformer architecture

!!!

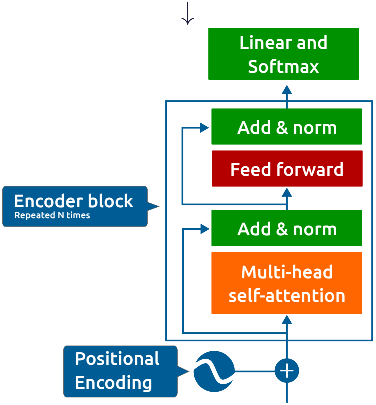
↓



The transformer: Encoder or decoder

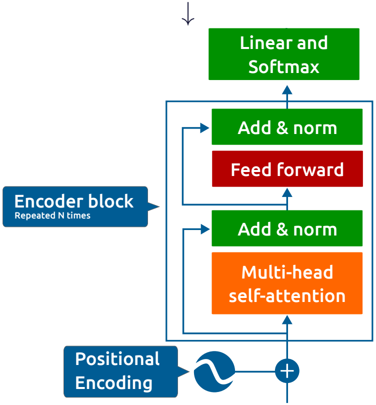
This is the **Encoder** transformer architecture

!!!



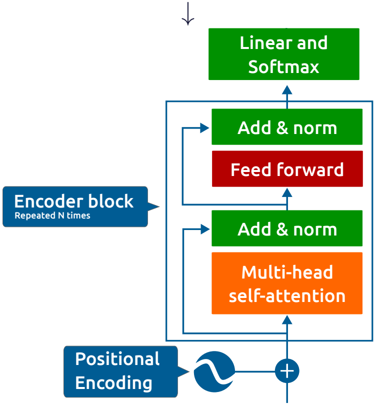
This is the **Decoder** transformer architecture

!!!

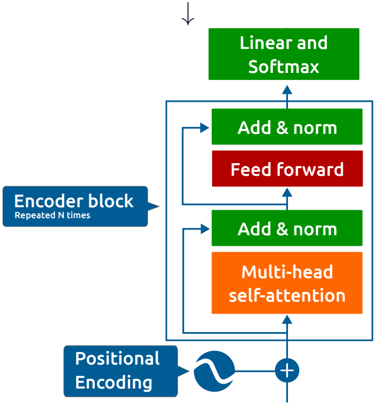


The transformer: Encoder or decoder

This is the **Encoder** or a **Decoder** transformer architecture !!!



This is the **Encoder** or a **Decoder** transformer architecture !!!



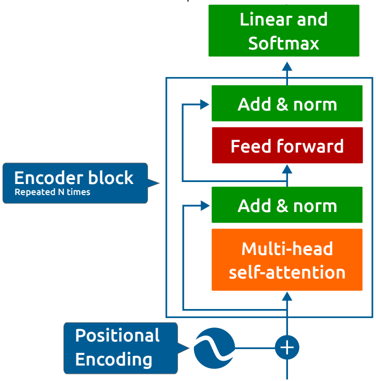
Encoder or Decoder ?

- Modules/architecture for Decoder or Encoder are (can be) the same
- Why calling them differently ?

There is a difference, but . . .

This is the **Encoder** or a **Decoder** transformer architecture !!!

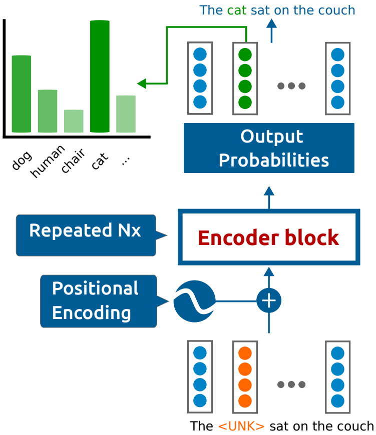
↓



Encoder or Decoder ?

- Modules/architecture for Decoder or Encoder are (can be) the same
- Why calling them differently ?

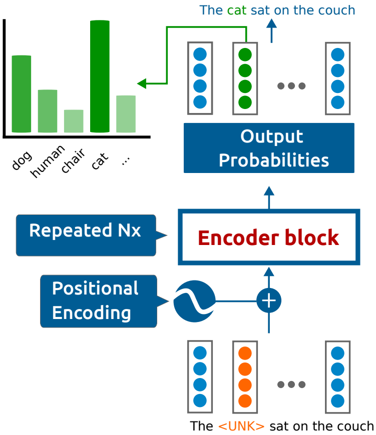
There is a difference, but . . .
→ based on the training procedure



Pretrained encoder

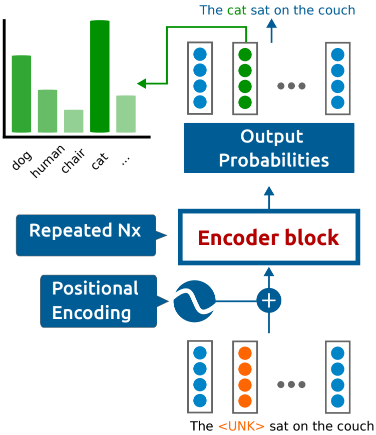
- Bidirectionnal (interactions between each internal token/latent representation)

The transformer: The Encoder model



Pretrained encoder

- Bidirectionnal (interactions between each internal token/latent representation)
- Often pre-trained on Masked Language Modeling task (MLM)



Pretrained encoder

- Bidirectionnal (interactions between each internal token/latent representation)
- Often pre-trained on Masked Language Modeling task (MLM)

NB: Different pretraining task are often targeted

The transformer: MLM task

Masked Language Modeling

- Randomly mask input token(s)
- Try to predict the masked token(s) !!!

The model has ”access” to all other tokens

The transformer: MLM task

Masked Language Modeling

- Randomly mask input token(s)
- Try to predict the masked token(s) !!!

The model has ”access” to all other tokens

Objective

Let be a sequence $X = (x_1, x_2, \ldots, x_n)$, let say we consider x_i the masked token. We want to maximize :

$$P(x_i | x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n)$$

The transformer: MLM task

Masked Language Modeling

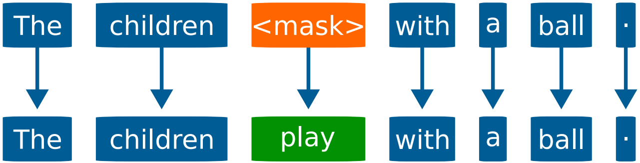
- Randomly mask input token(s)
- Try to predict the masked token(s) !!!

The model has ”access” to all other tokens

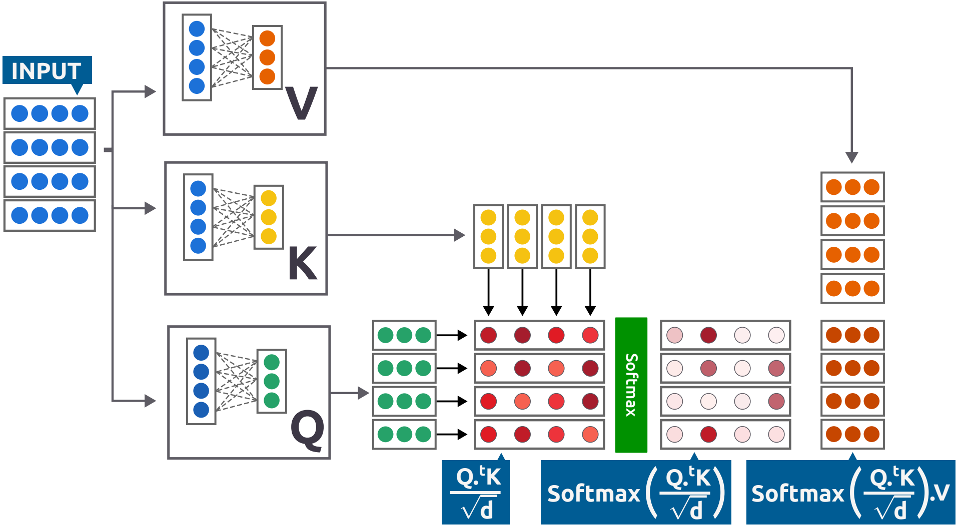
Objective

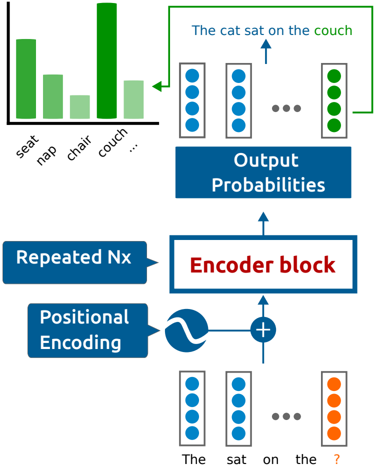
Let be a sequence $X = (x_1, x_2, \ldots, x_n)$, let say we consider x_i the masked token. We want to maximize :

$$P(x_i | x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n)$$



The transformer: Bidirectionnal attention

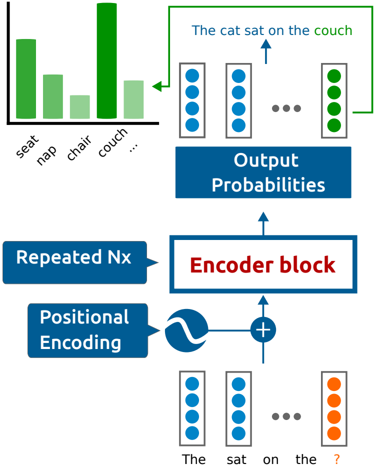




Pretrained Decoder

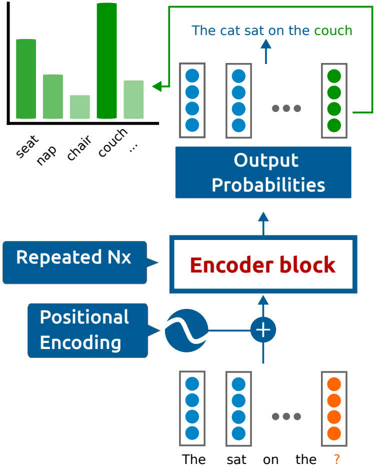
- Unidirectionnal (current token only depends from previous ones)

The transformer: The Decoder model



Pretrained Decoder

- Unidirectionnal (current token only depends from previous ones)
- Often pre-trained on Next Token Prediction task



Pretrained Decoder

- Unidirectionnal (current token only depends from previous ones)
- Often pre-trained on Next Token Prediction task

Masked Language Modeling

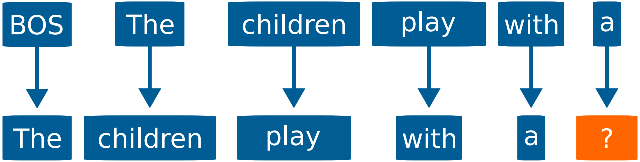
- Try to predict the next token

The model has ”access” only to previous tokens

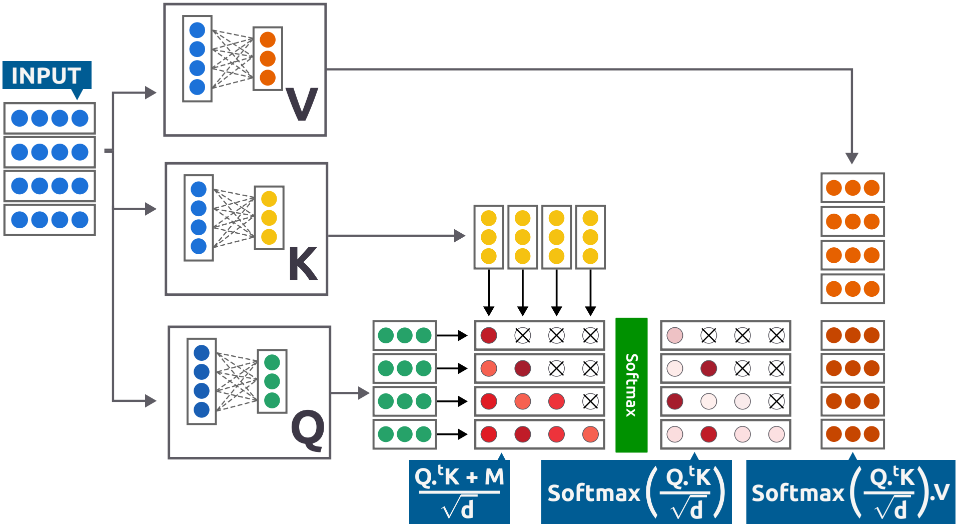
Objective

Let be a sequence $X = (x_1, x_2, \ldots, x_n)$, for each $i \in \{1, \ldots n\}$, we want to maximize :

$$P(x_i | x_1, \ldots, x_{i-1})$$



The transformer: Unidirectionnal attention



The transformer: Encoder or decoder

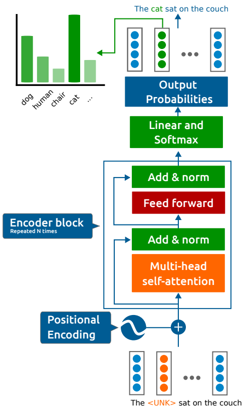


Figure 1: Encoder model and (one of) pretraining task

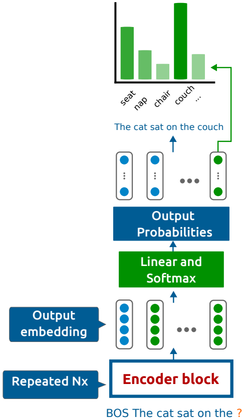


Figure 2: Decoder model and (one of) pretraining task

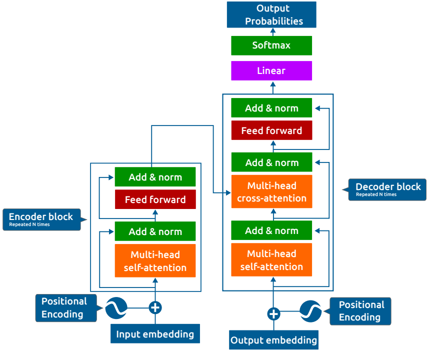
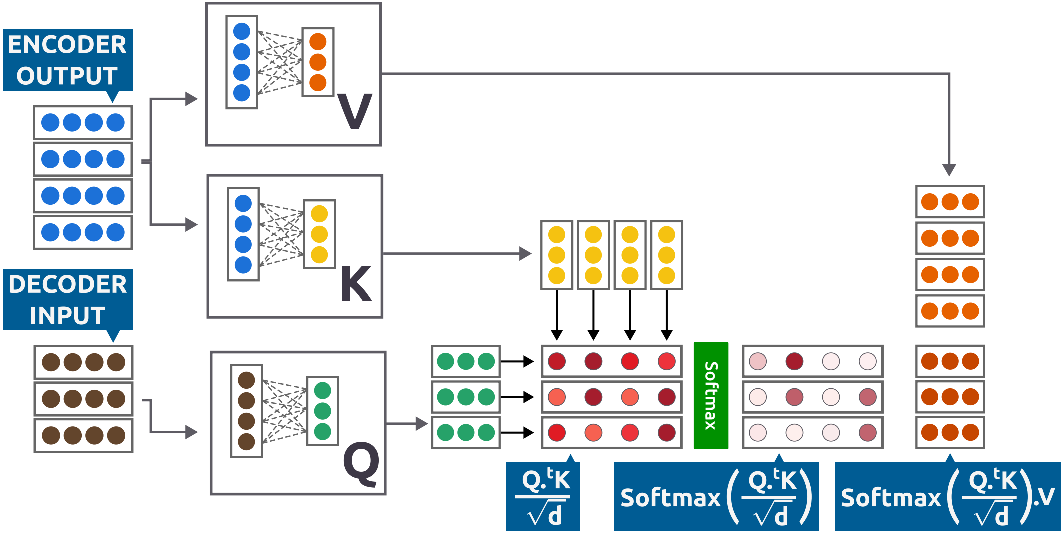


Figure 3: Encoder-decoder models

Encoder-decoder

- Encoder similar to encoder only architecture
- Decoder with mask in attention and an additional cross-attention layer

The transformer: cross-attention



A pretrained model

- Trained on a task (that does not need annotated data)
- Different architectures (encoder, decoder, encoder-decoder)

A pretrained model

- Trained on a task (that does not need annotated data)
- Different architectures (encoder, decoder, encoder-decoder)

Tuning those models

- What kind of architecture, what task ?

The transformer: Conlusion

A pretrained model

- Trained on a task (that does not need annotated data)
- Different architectures (encoder, decoder, encoder-decoder)

Tuning those models

- What kind of architecture, what task ?
- How adapting the model to my task ?

The transformer: Conlusion

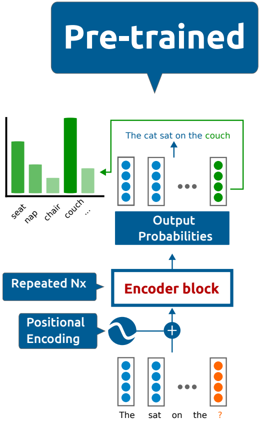
A pretrained model

- Trained on a task (that does not need annotated data)
- Different architectures (encoder, decoder, encoder-decoder)

Tuning those models

- What kind of architecture, what task ?
- How adapting the model to my task ?
- What are the tools ?

Adaptation: How to adapt to a new task



+

New task

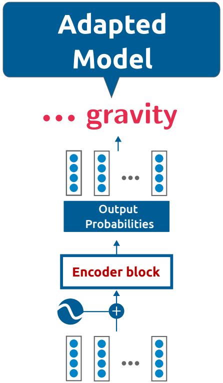
In meteorology, precipitation is any product of the condensation of atmospheric water vapor that falls under **gravity**. The main forms of precipitation include drizzle, rain, sleet, snow, **graupel** and hail... Precipitation forms as smaller droplets coalesce via collision with other rain drops or ice crystals **within a cloud**. Short, intense periods of rain in scattered locations are called "showers".

What causes precipitation to fall?
gravity

What is another main form of precipitation besides drizzle, rain, snow, sleet and hail?
graupel

Where do water droplets collide with ice crystals to form precipitation?
within a cloud

→



In meteorology, precipitation is any product of the condensation of atmospheric water vapor that falls under **gravity**. The main forms of precipitation include drizzle, rain, sleet, snow, **graupel** and hail... Precipitation forms as smaller droplets coalesce via collision with other rain drops or ice crystals **within a cloud**. Short, intense periods of rain in scattered locations are called "showers".

What causes precipitation to fall?

Requirements

- A pretrained model/trasnformer !!! (BERT, Llama, etc...)
- A Dataset (sufficiently large) with annotated data !

Before going further

- Fine-tuning Large Language Models ?
- When does it start (with LLMs) ?

Adaptation: The BERT model

The Bert model

- An encoder model
- A bidirectional model (no mask)
- Let's start with the bert model

BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding

Jacob Devlin **Ming-Wei Chang** **Kenton Lee** **Kristina Toutanova**
Google AI Language
{jacobdevlin,mingweichang,kentonl,kristout}@google.com

Abstract

We introduce a new language representation model called **BERT**, which stands for **B**idirectional **E**ncoder **R**epresentations from **T**ransformers. Unlike recent language representation models (Peters et al., 2018a; Radford et al., 2018), BERT is designed to pre-train deep bidirectional representations from unlabeled text by jointly conditioning on both left and right context in all layers. As a result, the pre-trained BERT model can be fine-tuned with just one additional output layer to create state-of-the-art models for a wide range of tasks, such as question answering and language inference, without substantial task-specific architecture modifications.

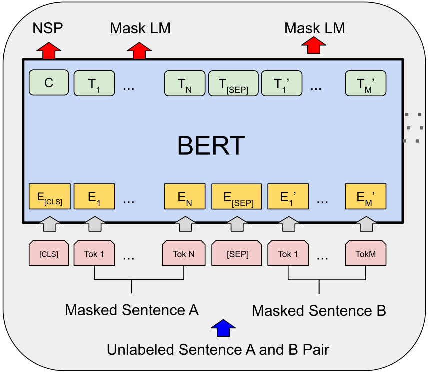
BERT is conceptually simple and empirically

There are two existing strategies for applying pre-trained language representations to downstream tasks: *feature-based* and *fine-tuning*. The feature-based approach, such as ELMo (Peters et al., 2018a), uses task-specific architectures that include the pre-trained representations as additional features. The fine-tuning approach, such as the Generative Pre-trained Transformer (OpenAI GPT) (Radford et al., 2018), introduces minimal task-specific parameters, and is trained on the downstream tasks by simply fine-tuning *all* pre-trained parameters. The two approaches share the same objective function during pre-training, where they use unidirectional language models to learn general language representations.

We argue that current techniques restrict the

Adaptation: The BERT model

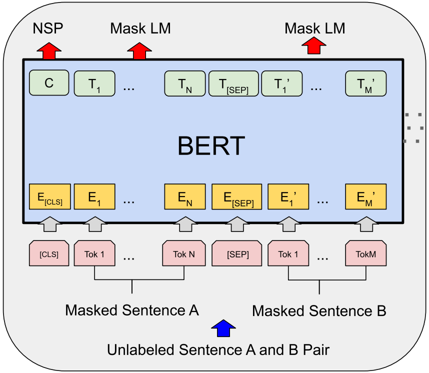
Input	[CLS]	my	dog	is	cute	[SEP]	he	likes	play	##ing	[SEP]
Token Embeddings	$E_{[CLS]}$	E_{my}	E_{dog}	E_{is}	E_{cute}	$E_{[SEP]}$	E_{he}	E_{likes}	E_{play}	$E_{\#\#ing}$	$E_{[SEP]}$



Pre-training

Adaptation: The BERT model

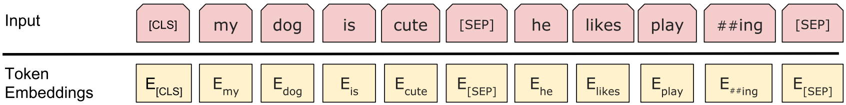
Input	[CLS]	my	dog	is	cute	[SEP]	he	likes	play	##ing	[SEP]
Token Embeddings	$E_{[CLS]}$	E_{my}	E_{dog}	E_{is}	E_{cute}	$E_{[SEP]}$	E_{he}	E_{likes}	E_{play}	$E_{\#\#ing}$	$E_{[SEP]}$

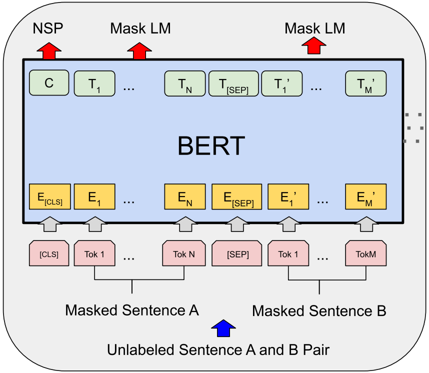


Pre-training

Pre-Training

Adaptation: The BERT model



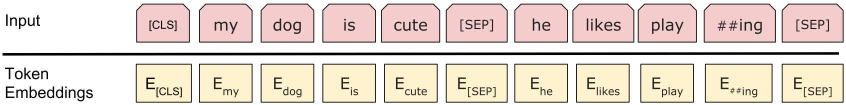


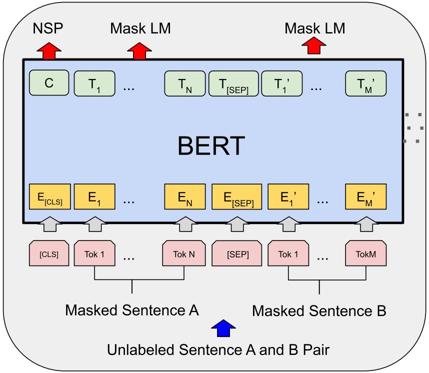
Pre-training

Pre-Training

- A masked language modeling task (Mask LM)

Adaptation: The BERT model



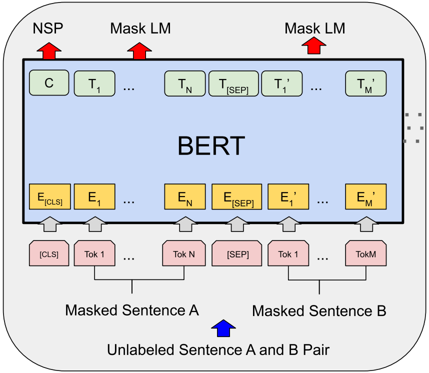


Pre-training

Pre-Training

- A masked language modeling task (Mask LM)
- A next sentence prediction task (NSP)

Input	[CLS]	my	dog	is	cute	[SEP]	he	likes	play	##ing	[SEP]
Token Embeddings	$E_{[CLS]}$	E_{my}	E_{dog}	E_{is}	E_{cute}	$E_{[SEP]}$	E_{he}	E_{likes}	E_{play}	$E_{\#\#ing}$	$E_{[SEP]}$



Pre-training

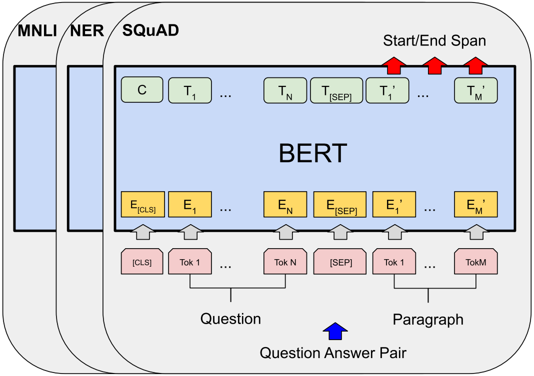
Pre-Training

- A masked language modeling task (Mask LM)
- A next sentence prediction task (NSP)

NSP task ?

- Two sentences (not always consecutive) in input during pretraining
- A classifier on a special token (`[CLS]`) must predict if sentence are consecutive

Adaptation: The BERT model



Fine-Tuning

Fine-tuning on different tasks

- **MNLI:** Classify from the CLS embedding if a premise validate an hypothesis
- **NER :** Classify from output embeddings if the token is a named entity (and class)
- **SQuad:** Classify from output embeddings if a token is part of the answer

→ All the weight are updated (almost)

Adaptation: Problem

Fine-tuning

Fine tuning is ok with BERT $\rightarrow$ only 110 Millions weights ($\approx$ 418 Mo)

Adaptation: Problem

Fine-tuning

Fine tuning is ok with BERT $\rightarrow$ only 110 Millions weights ($\approx$ 418 Mo)

- It fast enough on common hardware (GPU)

Adaptation: Problem

Fine-tuning

Fine tuning is ok with BERT $\rightarrow$ only 110 Millions weights ($\approx$ 418 Mo)

- It fast enough on common hardware (GPU)
- It can be fine-tuned on one GPU

Fine-tuning

Fine tuning is ok with BERT $\rightarrow$ only 110 Millions weights ($\approx$ 418 Mo)

- It fast enough on common hardware (GPU)
- It can be fine-tuned on one GPU

Adaptation: Problem

Fine-tuning

Fine tuning is ok with BERT $\rightarrow$ only 110 Millions weights ($\approx$ 418 Mo)

- It fast enough on common hardware (GPU)
- It can be fine-tuned on one GPU

(however in 2018

Fine-tuning

Fine tuning is ok with BERT $\rightarrow$ only 110 Millions weights ($\approx$ 418 Mo)

- It fast enough on common hardware (GPU)
- It can be fine-tuned on one GPU

(however in 2018 $\rightarrow$ a huge model)

Adaptation: Problem

Fine-tuning

Fine tuning is ok with BERT $\rightarrow$ only 110 Millions weights ($\approx$ 418 Mo)

- It fast enough on common hardware (GPU)
- It can be fine-tuned on one GPU

(however in 2018 $\rightarrow$ a huge model)

Adaptation: Problem

Fine-tuning

Fine tuning is ok with BERT $\rightarrow$ only 110 Millions weights ($\approx$ 418 Mo)

- It fast enough on common hardware (GPU)
- It can be fine-tuned on one GPU

(however in 2018 $\rightarrow$ a huge model)

More recent models ?

Fine-tuning

Fine tuning is ok with BERT $\rightarrow$ only 110 Millions weights ($\approx$ 418 Mo)

- It fast enough on common hardware (GPU)
- It can be fine-tuned on one GPU

(however in 2018 $\rightarrow$ a huge model)

More recent models ?

- GPT2-large $\rightarrow$ 778M weights

Fine-tuning

Fine tuning is ok with BERT $\rightarrow$ only 110 Millions weights ($\approx$ 418 Mo)

- It fast enough on common hardware (GPU)
- It can be fine-tuned on one GPU

(however in 2018 $\rightarrow$ a huge model)

More recent models ?

- GPT2-large $\rightarrow$ 778M weights
- Mixtral-8x7B $\rightarrow$ 56B weights

Fine-tuning

Fine tuning is ok with BERT $\rightarrow$ only 110 Millions weights ($\approx$ 418 Mo)

- It fast enough on common hardware (GPU)
- It can be fine-tuned on one GPU

(however in 2018 $\rightarrow$ a huge model)

More recent models ?

- GPT2-large $\rightarrow$ 778M weights
- Mixtral-8x7B $\rightarrow$ 56B weights
- Llama3-70B $\rightarrow$ 70B weights

Adaptation: Problem

Fine-tuning

Fine tuning is ok with BERT $\rightarrow$ only 110 Millions weights ($\approx$ 418 Mo)

- It fast enough on common hardware (GPU)
- It can be fine-tuned on one GPU

(however in 2018 $\rightarrow$ a huge model)

More recent models ?

- GPT2-large $\rightarrow$ 778M weights
- Mixtral-8x7B $\rightarrow$ 56B weights
- Llama3-70B $\rightarrow$ 70B weights
- Bloom $\rightarrow$ 176B weights

Adaptation: Problem

Fine-tuning

Fine tuning is ok with BERT $\rightarrow$ only 110 Millions weights ($\approx$ 418 Mo)

- It fast enough on common hardware (GPU)
- It can be fine-tuned on one GPU

(however in 2018 $\rightarrow$ a huge model)

More recent models ?

- GPT2-large $\rightarrow$ 778M weights
- Mixtral-8x7B $\rightarrow$ 56B weights
- Llama3-70B $\rightarrow$ 70B weights
- Bloom $\rightarrow$ 176B weights

$\rightarrow$ Can we adapt the model to our task efficiently ?

Adaptation: Problem

Fine-tuning

Fine tuning is ok with BERT $\rightarrow$ only 110 Millions weights ($\approx$ 418 Mo)

- It fast enough on common hardware (GPU)
- It can be fine-tuned on one GPU

(however in 2018 $\rightarrow$ a huge model)

More recent models ?

- GPT2-large $\rightarrow$ 778M weights
- Mixtral-8x7B $\rightarrow$ 56B weights
- Llama3-70B $\rightarrow$ 70B weights
- Bloom $\rightarrow$ 176B weights

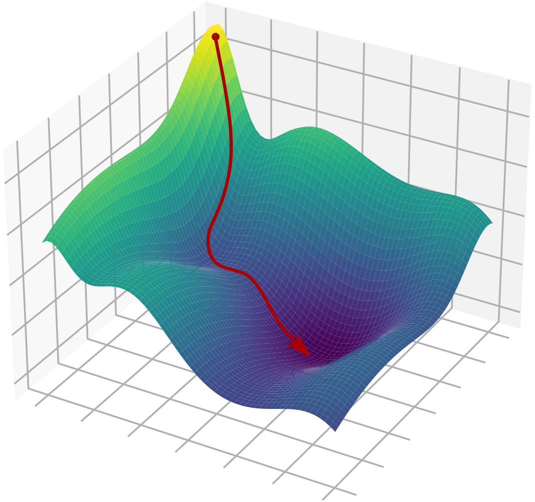
$\rightarrow$ Can we adapt the model to our task efficiently ?
$\rightarrow$ Can we compress the models to obtain same performances ?

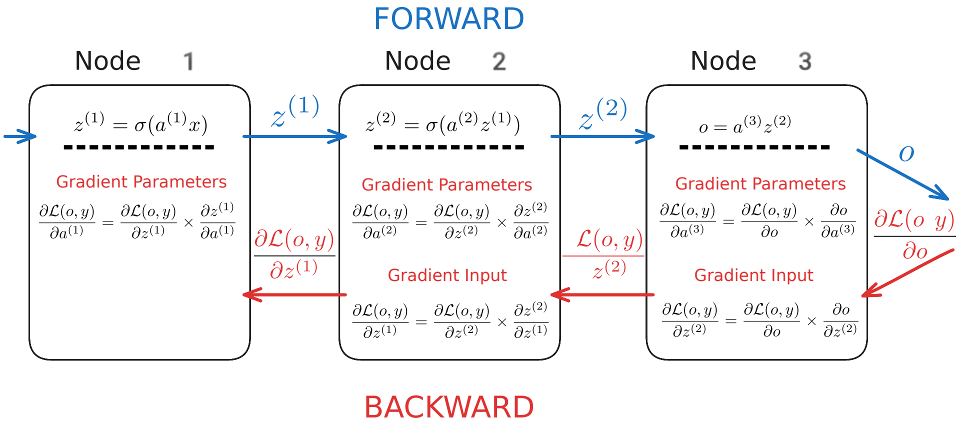
Parameter Efficient Fine Tuning (PEFT)

Fine-tuning

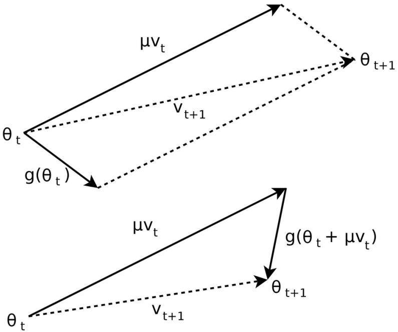
$\rightarrow$ Train the model on a new task based on pretrained weights

- All weights are updated
- Backpropagation on all computation graph
- **Storing information on the gradient (for each weigth)**





$\rightarrow$ Storing weights but also gradient parameters !!!



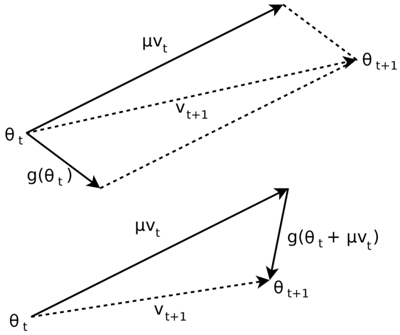
When Fine-tuning : SGD with momentum

Updating weigths with a momentum :

$$\theta_{t+1} = W_t - V_{t+1}$$

with

$$V_{t+1} = \mu V_t + g_t$$



When Fine-tuning : SGD with momentum

Updating weigths with a momentum :

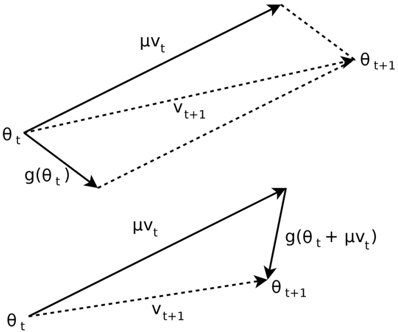
$$\theta_{t+1} = W_t - V_{t+1}$$

with

$$V_{t+1} = \mu V_t + g_t$$

$\rightarrow$ Storing a copy of the gradient (matrix V_t)

PEFT: Fine-tuning



- $4 \times n_\theta$ bytes for the weigths
- $4 \times n_\theta$ bytes for the gradients
- $4 \times n_\theta$ bytes for V

When Fine-tuning : SGD with momentum

Updating weigths with a momentum :

$$\theta_{t+1} = W_t - V_{t+1}$$

with

$$V_{t+1} = \mu V_t + g_t$$

$\rightarrow$ Storing a copy of the gradient (matrix V_t)

$\rightarrow$ With 32 bits for a float
$\rightarrow$ With n_θ the number of parameters

PEFT: Fine-tuning

When Fine-tuning : Adam

Updating weigths with Adam:

$\rightarrow$ It uses two moments !!!

- $4 \times n_\theta$ bytes for the weigths
- $4 \times n_\theta$ bytes for the gradient
- $4 \times n_\theta$ bytes for first moment
- $4 \times n_\theta$ bytes fir the second moment

input : γ (lr), β_1, β_2 (betas), θ_0 (params), $f(\theta)$ (objective)
λ (weight decay), *amsgrad*, *maximize*
initialize : $m_0 \leftarrow 0$ (first moment), $v_0 \leftarrow 0$ (second moment), $\widehat{v_0}^{max} \leftarrow 0$

for $t = 1$ **to** ... **do**
  if *maximize* :
    $g_t \leftarrow -\nabla_\theta f_t(\theta_{t-1})$
  else
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
  if $\lambda \neq 0$
    $g_t \leftarrow g_t + \lambda\theta_{t-1}$
  $m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1) g_t$
  $v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2) g_t^2$
  $\widehat{m_t} \leftarrow m_t / \big(1-\beta_1^t\big)$
  $\widehat{v_t} \leftarrow v_t / \big(1-\beta_2^t\big)$
  if *amsgrad*
    $\widehat{v_t}^{max} \leftarrow \max(\widehat{v_t}^{max}, \widehat{v_t})$
    $\theta_t \leftarrow \theta_{t-1} - \gamma \widehat{m_t} / \big(\sqrt{\widehat{v_t}^{max}} + \epsilon\big)$
  else
    $\theta_t \leftarrow \theta_{t-1} - \gamma \widehat{m_t} / \big(\sqrt{\widehat{v_t}} + \epsilon\big)$

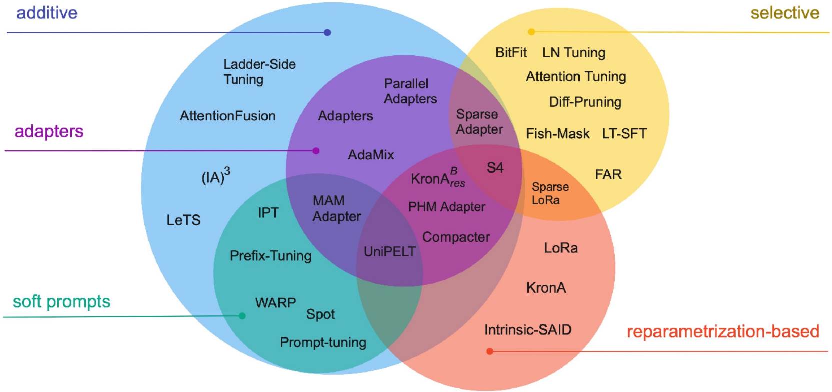
return θ_t

PEFT: Making more efficient

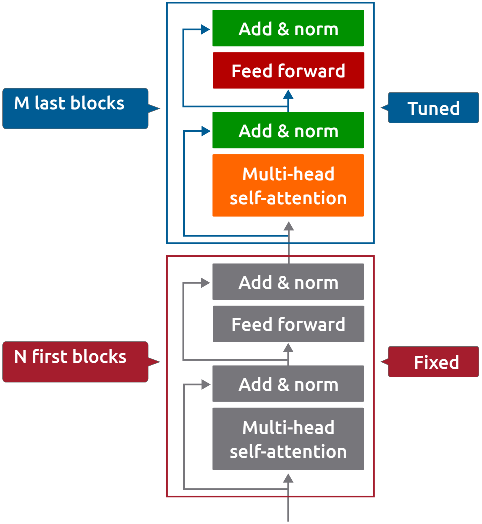
What can/should we do ?

- Do not store the gradient for each weight
- Reduce the number of bytes for each weight
- Update only input for new task

”Scaling Down to Scale Up: A Guide to Parameter-Efficient Fine-Tuning”



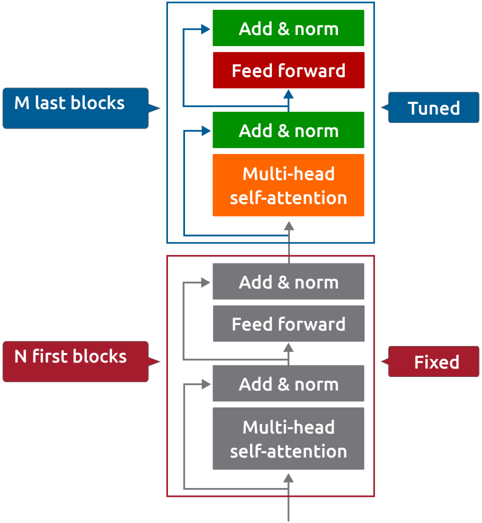
PEFT: Last Layer(s) only



Training only last layers

- Reduce the number of gradient "information" to store (% of layer fixed)
- Correct performances (if enough layers)

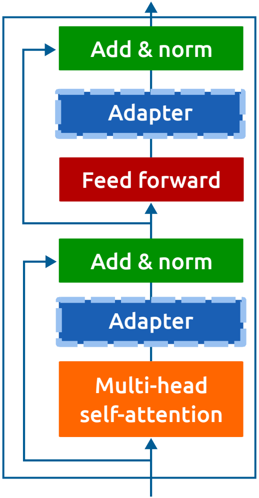
PEFT: Last Layer(s) only



Training only last layers

- Reduce the number of gradient “information” to store (% of layer fixed)
- Correct performances (if enough layers)

$\rightarrow$ can we do better?

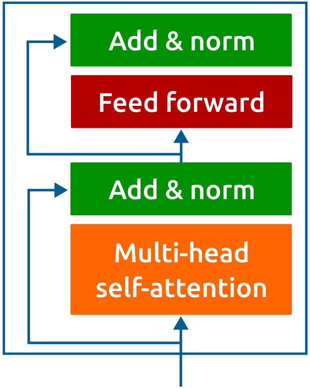


The Adapter approach

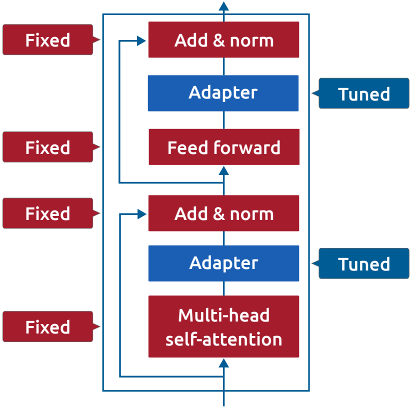
- Adding new modules inside the model (Feed-Forward)
- Updating those modules only

"Parameter-Efficient Transfer Learning for NLP", N. Houlsby et al., ICML 2019

PEFT: Adapter approaches



→

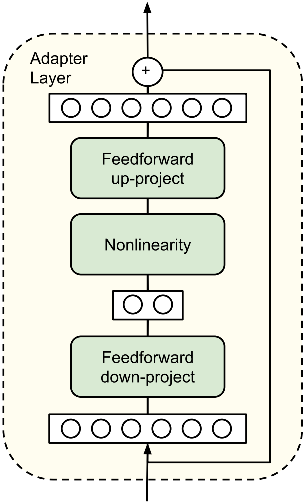


“Parameter-Efficient Transfer Learning for NLP”, N. Houlsby et al., ICML 2019

The adapter module

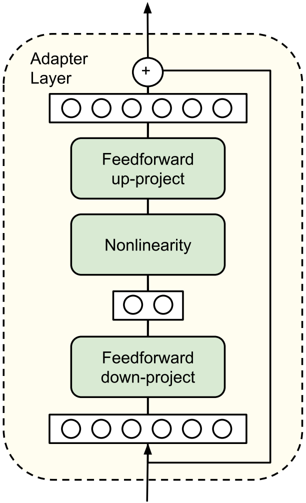
- $W_d \in \mathbb{R}^{k \times n}$ for down projection $k << n$
- An activation function σ
- $W_u \in \mathbb{R}^{n \times k}$ for up projection

$$f(x) = W_u \sigma(W_d x) + x$$



PEFT: Adapter approaches, efficiency

- $n_\theta + l \times (k \times n) \times 2$ float parameters
- $l \times (k \times n) \times 2$ floats for the gradient
- $l \times (k \times n) \times 2$ floats for first moment
- $l \times (k \times n) \times 2$ floats for the second moment and second moment



PEFT: Adapter approaches, efficiency (example)

Let consider the BERT base model with $l = 12$ (number of layers), $P = 110M$ (the number of parameters), $n = 768$.

Fine-tuning all layers

- $110M$ float parameters

PEFT: Adapter approaches, efficiency (example)

Let consider the BERT base model with $l = 12$ (number of layers), $P = 110M$ (the number of parameters), $n = 768$.

Fine-tuning all layers

- $110M$ float parameters
- $110M$ floats for the gradient

PEFT: Adapter approaches, efficiency (example)

Let consider the BERT base model with $l = 12$ (number of layers), $P = 110M$ (the number of parameters), $n = 768$.

Fine-tuning all layers

- $110M$ float parameters
- $110M$ floats for the gradient
- $2 \times 110M$ floats for first and second moment

PEFT: Adapter approaches, efficiency (example)

Let consider the BERT base model with $l = 12$ (number of layers), $P = 110M$ (the number of parameters), $n = 768$.

Fine-tuning all layers

- $110M$ float parameters
- $110M$ floats for the gradient
- $2 \times 110M$ floats for first and second moment

$\rightarrow$ 440 millions parameters in RAM

PEFT: Adapter approaches, efficiency (example)

Let consider the BERT base model with $l = 12$ (number of layers), $P = 110M$ (the number of parameters), $n = 768$.

Fine-tuning all layers

- $110M$ float parameters
- $110M$ floats for the gradient
- $2 \times 110M$ floats for first and second moment

$\rightarrow$ 440 millions parameters in RAM

FT with Adapter

Let $k = 64$

PEFT: Adapter approaches, efficiency (example)

Let consider the BERT base model with $l = 12$ (number of layers), $P = 110M$ (the number of parameters), $n = 768$.

Fine-tuning all layers

- $110M$ float parameters
- $110M$ floats for the gradient
- $2 \times 110M$ floats for first and second moment

$\rightarrow$ 440 millions parameters in RAM

FT with Adapter

Let $k = 64$

- $12 \times 64 \times 768 \times 2 \approx 1,2M$ floats for the gradient

PEFT: Adapter approaches, efficiency (example)

Let consider the BERT base model with $l = 12$ (number of layers), $P = 110M$ (the number of parameters), $n = 768$.

Fine-tuning all layers

- $110M$ float parameters
- $110M$ floats for the gradient
- $2 \times 110M$ floats for first and second moment

$\rightarrow$ 440 millions parameters in RAM

FT with Adapter

Let $k = 64$

- $12 \times 64 \times 768 \times 2 \approx 1, 2M$ floats for the gradient
- $12 \times 64 \times 768 \times 2 \times 2 \approx 2, 4M$ floats for first and second moment

Let consider the BERT base model with $l = 12$ (number of layers), $P = 110M$ (the number of parameters), $n = 768$.

Fine-tuning all layers

- $110M$ float parameters
- $110M$ floats for the gradient
- $2 \times 110M$ floats for first and second moment

$\rightarrow$ 440 millions parameters in RAM

FT with Adapter

Let $k = 64$

- $12 \times 64 \times 768 \times 2 \approx 1,2M$ floats for the gradient
- $12 \times 64 \times 768 \times 2 \times 2 \approx 2,4M$ floats for first and second moment
- $110M + 3.6M$ float parameters

Let consider the BERT base model with $l = 12$ (number of layers), $P = 110M$ (the number of parameters), $n = 768$.

Fine-tuning all layers

- $110M$ float parameters
- $110M$ floats for the gradient
- $2 \times 110M$ floats for first and second moment

$\rightarrow$ 440 millions parameters in RAM

FT with Adapter

Let $k = 64$

- $12 \times 64 \times 768 \times 2 \approx 1,2M$ floats for the gradient
- $12 \times 64 \times 768 \times 2 \times 2 \approx 2,4M$ floats for first and second moment
- $110M + 3.6M$ float parameters

Let consider the BERT base model with $l = 12$ (number of layers), $P = 110M$ (the number of parameters), $n = 768$.

Fine-tuning all layers

- $110M$ float parameters
- $110M$ floats for the gradient
- $2 \times 110M$ floats for first and second moment

$\rightarrow$ 440 millions parameters in RAM

FT with Adapter

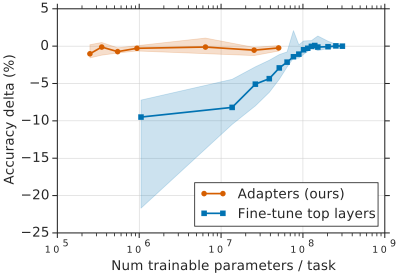
Let $k = 64$

- $12 \times 64 \times 768 \times 2 \approx 1,2M$ floats for the gradient
- $12 \times 64 \times 768 \times 2 \times 2 \approx 2,4M$ floats for first and second moment
- $110M + 3.6M$ float parameters

$\rightarrow$ 118 millions parameters in RAM

PEFT: Adapter approaches, performances

- Relatively good performances compared to fine-tuning
- On GLUE Benchmark with BERT-Large reach a score of 79.6 ($k = 64$) instead of 80.4 (ft)



A remaining issue

⚠ At inference adapter use more memory (due to added layers)

PEFT: Adapter approaches - LoRA

The LoRA approach:

- An adapter appproach
- With no additional layer at inference
- Comparable/better performances than fine-tuning

LORA: LOW-RANK ADAPTATION OF LARGE LANGUAGE MODELS

Edward Hu* **Yelong Shen*** **Phillip Wallis** **Zeyuan Allen-Zhu**
Yuanzhi Li **Shean Wang** **Lu Wang** **Weizhu Chen**
Microsoft Corporation
{edwardhu, yeshe, phwallis, zeyuana,
yuanzhil, swang, luw, wzchen}@microsoft.com
yuanzhil@andrew.cmu.edu
(Version 2)

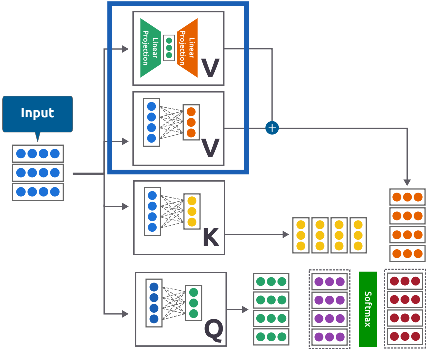
ABSTRACT

An important paradigm of natural language processing consists of large-scale pre-training on general domain data and adaptation to particular tasks or domains. As we pre-train larger models, full fine-tuning, which retrains all model parameters, becomes less feasible. Using GPT-3 175B as an example – deploying independent instances of fine-tuned models, each with 175B parameters, is prohibitively expensive. We propose **Lo**w-**R**ank **A**daptation, or LoRA, which freezes the pre-trained model weights and injects trainable rank decomposition matrices into each layer of the Transformer architecture, greatly reducing the number of trainable parameters for downstream tasks. Compared to GPT-3 175B fine-tuned with Adam, LoRA can reduce the number of trainable parameters by 10,000 times and the GPU memory requirement by 3 times. LoRA performs on-par or better than fine-tuning in model quality on RoBERTa, DeBERTa, GPT-2, and GPT-3, despite having fewer trainable parameters, a higher training throughput, and, unlike adapters, *no additional inference latency*. We also provide an empirical investigation into rank-deficiency in language model adaptation, which sheds light on the efficacy of LoRA. We release a package that facilitates the integration of LoRA with PyTorch models and provide our im
DeBERTa, and GPT-2 at ht

"LoRA: Low-Rank Adaptation of Large Language Models", E.J. Hu, ICLR 2022

Principle

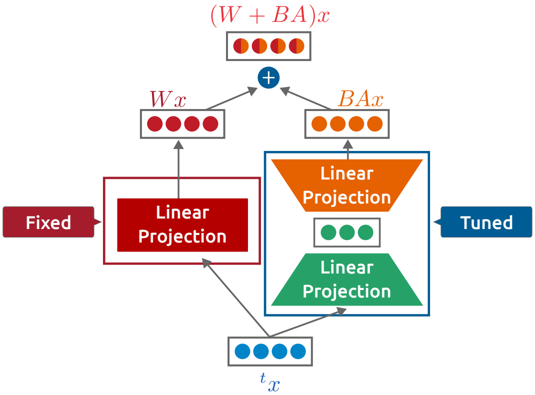
- For specified linear layer (on Q,K, and/or V)
- Train a specific module



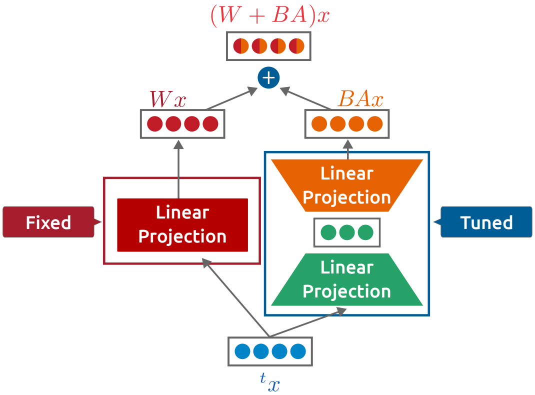
$\rightarrow W_q^l \in \mathbb{R}^{m \times n}$ the linear associated to the query
$\rightarrow A_q^l \in \mathbb{R}^{k \times n}$ a projection $k << m$
$\rightarrow B_q^l \in \mathbb{R}^{n \times k}$ a projection $k << m$
$\rightarrow x \in \mathbb{R}^n$ input of the attention block
Output (in the example the query) is computed as :

$$Q = W_q^l x + B_q^l (A_q^l x)$$

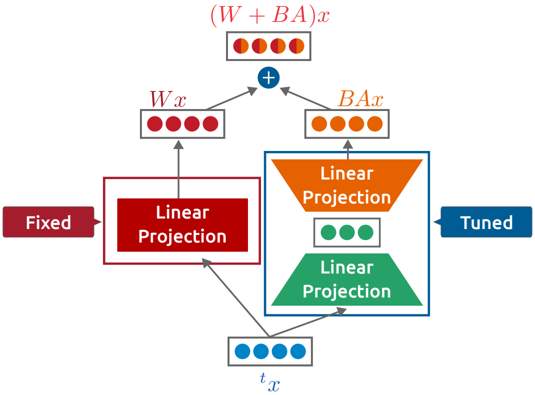
PEFT: Adapter approaches - LoRA



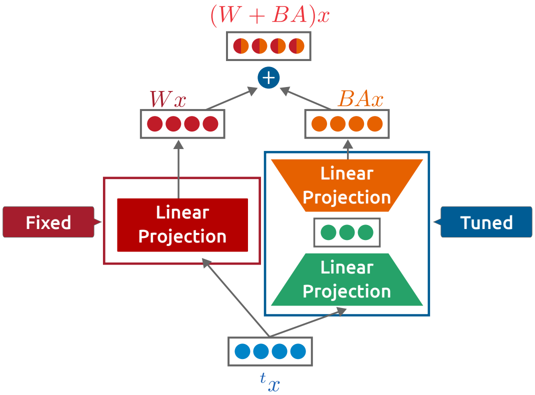
PEFT: Adapter approaches - LoRA



$$Q = W_q^l x + B_q^l (A_q^l x)$$



$$Q = W_q^l x + B_q^l (A_q^l x)$$
$$Q = (W_q^l + B_q^l A_q^l) x$$



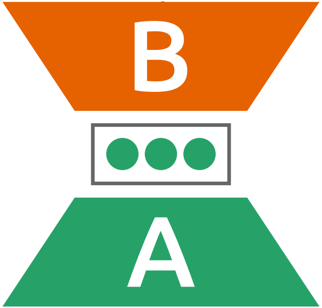
$$Q = W_q^l x + B_q^l (A_q^l x)$$
$$Q = (W_q^l + B_q^l A_q^l) x$$

The operation can be factorised !!!

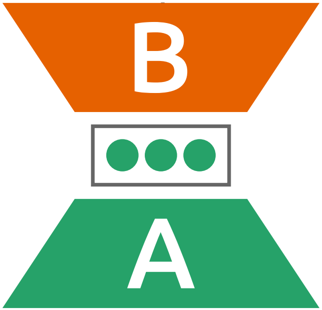
$\rightarrow$ At inference change weights of Q linear module, replacing it by :

$$W_q^{l'} = W_q^l + B_q^l A_q^l$$

No additional weigths at inference !!!

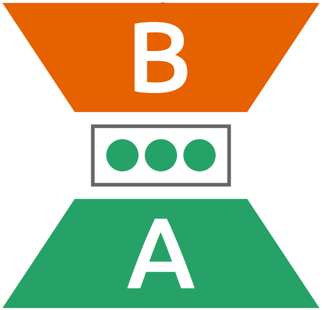


Why low rank?



Why low rank?

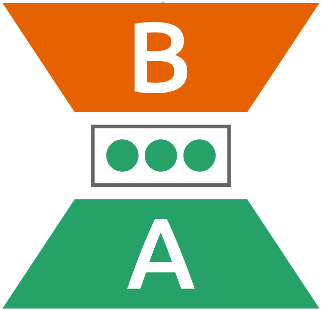
$\rightarrow rank(B_q^l A_q^l) \leq k \rightarrow A_q^l \in \mathbb{R}^{k \times n}$



Why low rank?

$\rightarrow rank(B_q^l A_q^l) \leq k \rightarrow A_q^l \in \mathbb{R}^{k \times n}$

Why **low** rank?



Why low rank?

$\rightarrow rank(B_q^l A_q^l) \leq k \rightarrow A_q^l \in \mathbb{R}^{k \times n}$

Why **low** rank?

$\rightarrow k << n$

Efficiency (memory)

- For training $\rightarrow$ similar to bottleneck adapter (sligthly lower memory)
- For inference $\rightarrow$ similar to base model

$\rightarrow$ It mostly depends on the value k

	Weight Type	$r=1$	$r=2$	$r=4$	$r=8$	$r=64$
WikiSQL($\pm 0.5\%$)	W_q	68.8	69.6	70.5	70.4	70.0
	W_q, W_v	73.4	73.3	73.7	73.8	73.5
	W_q, W_k, W_v, W_o	74.1	73.7	74.0	74.0	73.9
MultiNLI ($\pm 0.1\%$)	W_q	90.7	90.9	91.1	90.7	90.7
	W_q, W_v	91.3	91.4	91.3	91.6	91.4
	W_q, W_k, W_v, W_o	91.2	91.7	91.7	91.5	91.4

Table 6: Validation accuracy on WikiSQL and MultiNLI with different rank r. To our surprise, a rank as small as one suffices for adapting both W_q and W_v on these datasets while training W_q alone needs a larger r. We conduct a similar experiment on GPT-2 in Section H.2.

Model & Method	# Trainable Parameters	MNLI	SST-2	MRPC	CoLA	QNLI	QQP	RTE	STS-B	Avg.
RoB$_{base}$ (FT)*	125.0M	**87.6**	94.8	90.2	**63.6**	92.8	**91.9**	78.7	91.2	86.4
RoB$_{base}$ (BitFit)*	0.1M	84.7	93.7	**92.7**	62.0	91.8	84.0	81.5	90.8	85.2
RoB$_{base}$ (Adpt^{D})*	0.3M	87.1$_{\pm .0}$	94.2$_{\pm .1}$	88.5$_{\pm 1.1}$	60.8$_{\pm .4}$	93.1$_{\pm .1}$	90.2$_{\pm .0}$	71.5$_{\pm 2.7}$	89.7$_{\pm .3}$	84.4
RoB$_{base}$ (Adpt^{D})*	0.9M	87.3$_{\pm .1}$	94.7$_{\pm .3}$	88.4$_{\pm .1}$	62.6$_{\pm .9}$	93.0$_{\pm .2}$	90.6$_{\pm .0}$	75.9$_{\pm 2.2}$	90.3$_{\pm .1}$	85.4
RoB$_{base}$ (LoRA)	0.3M	87.5$_{\pm .3}$	**95.1**$_{\pm .2}$	89.7$_{\pm .7}$	63.4$_{\pm 1.2}$	**93.3**$_{\pm .3}$	90.8$_{\pm .1}$	**86.6**$_{\pm .7}$	**91.5**$_{\pm .2}$	**87.2**
RoB$_{large}$ (FT)*	355.0M	90.2	**96.4**	**90.9**	68.0	94.7	**92.2**	86.6	92.4	88.9
RoB$_{large}$ (LoRA)	0.8M	**90.6**$_{\pm .2}$	96.2$_{\pm .5}$	**90.9**$_{\pm 1.2}$	**68.2**$_{\pm 1.9}$	**94.9**$_{\pm .3}$	91.6$_{\pm .1}$	**87.4**$_{\pm 2.5}$	**92.6**$_{\pm .2}$	**89.0**
RoB$_{large}$ (Adpt^{P})†	3.0M	90.2$_{\pm .3}$	96.1$_{\pm .3}$	90.2$_{\pm .7}$	**68.3**$_{\pm 1.0}$	**94.8**$_{\pm .2}$	**91.9**$_{\pm .1}$	83.8$_{\pm 2.9}$	92.1$_{\pm .7}$	88.4
RoB$_{large}$ (Adpt^{P})†	0.8M	**90.5**$_{\pm .3}$	**96.6**$_{\pm .2}$	89.7$_{\pm 1.2}$	67.8$_{\pm 2.5}$	**94.8**$_{\pm .3}$	91.7$_{\pm .2}$	80.1$_{\pm 2.9}$	91.9$_{\pm .4}$	87.9
RoB$_{large}$ (Adpt^{H})†	6.0M	89.9$_{\pm .5}$	96.2$_{\pm .3}$	88.7$_{\pm 2.9}$	66.5$_{\pm 4.4}$	94.7$_{\pm .2}$	92.1$_{\pm .1}$	83.4$_{\pm 1.1}$	91.0$_{\pm 1.7}$	87.8
RoB$_{large}$ (Adpt^{H})†	0.8M	90.3$_{\pm .3}$	96.3$_{\pm .5}$	87.7$_{\pm 1.7}$	66.3$_{\pm 2.0}$	94.7$_{\pm .2}$	91.5$_{\pm .1}$	72.9$_{\pm 2.9}$	91.5$_{\pm .5}$	86.4
RoB$_{large}$ (LoRA)†	0.8M	**90.6**$_{\pm .2}$	96.2$_{\pm .5}$	**90.2**$_{\pm 1.0}$	68.2$_{\pm 1.9}$	**94.8**$_{\pm .3}$	91.6$_{\pm .2}$	**85.2**$_{\pm 1.1}$	**92.3**$_{\pm .5}$	**88.6**
DeB$_{XXL}$ (FT)*	1500.0M	91.8	**97.2**	92.0	72.0	**96.0**	92.7	93.9	92.9	91.1
DeB$_{XXL}$ (LoRA)	4.7M	**91.9**$_{\pm .2}$	96.9$_{\pm .2}$	**92.6**$_{\pm .6}$	**72.4**$_{\pm 1.1}$	**96.0**$_{\pm .1}$	**92.9**$_{\pm .1}$	**94.9**$_{\pm .4}$	**93.0**$_{\pm .2}$	**91.3**

Conclusion

- An efficient and performant Adapter approach
- Used in a lot of adaptation problems

PEFT: Prefix based approaches

Why changing the weights ?

- We can use prompt (text prefixing the input)
- How I can choose the correct prompt ?

If enough annotated data for the task, can we find an “optimal” prompt ?

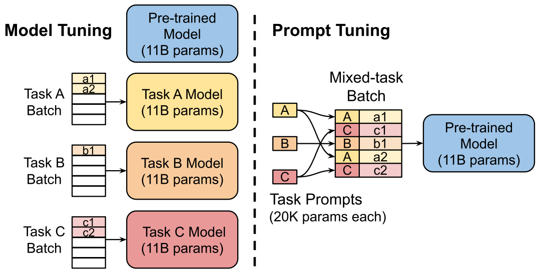
Continuous prompt tuning

- We can add special tokens to the input
- And train only those tokens !!!

[LL21] - Xiang Lisa Li and Percy Liang. “Prefix-Tuning: Optimizing Continuous Prompts for Generation”. In: ed. by Chengqing Zong et al. Online: Association for Computational Linguistics, Aug. 2021

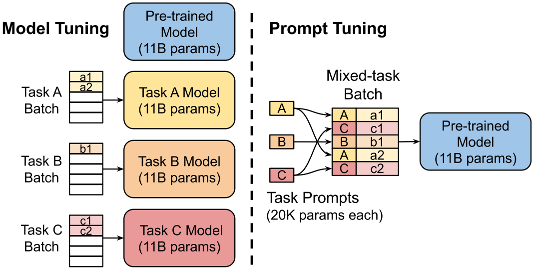
[LAC21] - Brian Lester, Rami Al-Rfou, and Noah Constant. “The Power of Scale for Parameter-Efficient Prompt Tuning”. In: ed. by Marie-Francine Moens et al. Online and Punta Cana, Dominican Republic: Association for Computational Linguistics, Nov. 2021

PEFT: Prompt-tuning



Fine-tuning tokens

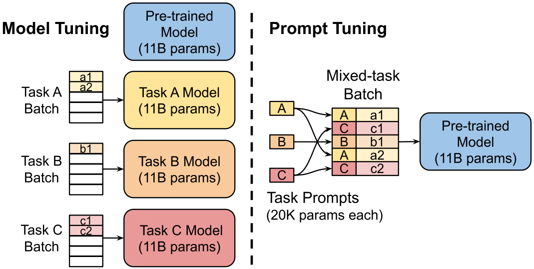
PEFT: Prompt-tuning



Fine-tuning tokens

- For each task associate special tokens

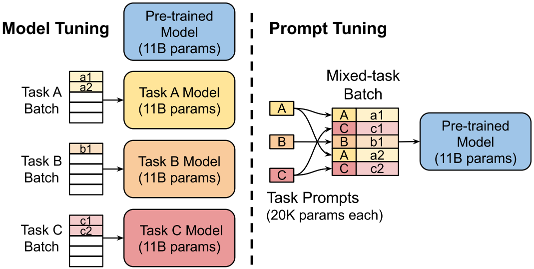
PEFT: Prompt-tuning



Fine-tuning tokens

- For each task associate special tokens
- Fine-tune only these tokens

PEFT: Prompt-tuning

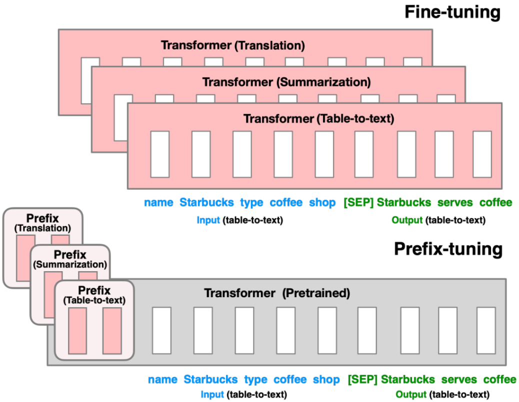


Fine-tuning tokens

- For each task associate special tokens
- Fine-tune only these tokens

→ Only need to retain gradient for these special tokens

PEFT: Prompt-tuning

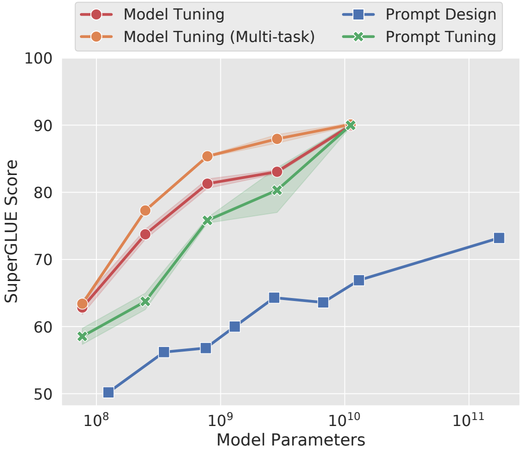


Fine-tuning tokens

- For each task associate special tokens
- Fine-tune only those tokens

$\rightarrow$ Only need to retain gradient for those special tokens

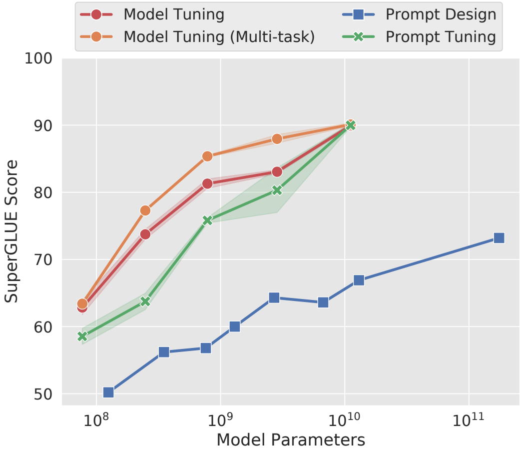
PEFT: Prompt-tuning



Performances

- Close to tuning model performances

PEFT: Prompt-tuning



Performances

- Close to tuning model performances
- Particularly on zero-shot (in the paper)

Model	#Weights	size
BERT-base	≈110M	>418Mo
GPT2-medium	≈355M	>1.2Go
Llama-2-7B	≈7B	>26Go
Llama-3-70B	≈70B	>260Go

Table 1: Estimation of the size of the model in RAM (based on 32 bits floats)

LLMs problem

- The model is too large (doesn't fit my GPU)
- Can I compress it with acceptable performance loss?

→ Encode parameters using less bits

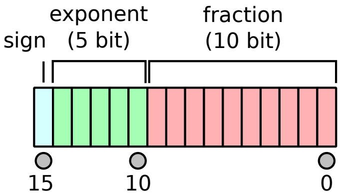
LLMs problem

- The model is too large (doesn't fit my GPU)
- Can I compress it with acceptable performance loss?

$\rightarrow$ Encode parameters using less bits

Using 16 bits ?

Coding on 16 bits half-precision
$\rightarrow \approx$ 4 decimal number



[0] image: https://en.wikipedia.org/wiki/Half-precision_floating-point_format

Half-precision issue when training !!

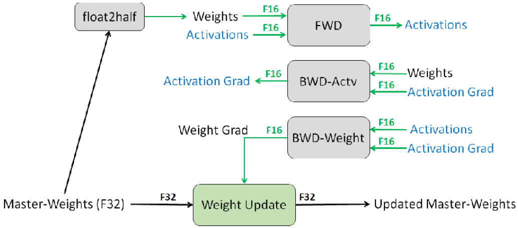
- Computation of gradient, forward, backward is ok
- Parameter update can be unstable

“These small valued gradients would become zero in the optimizer when multiplied with the learning rate and adversely affect the model accuracy” [mixed]

Mixed Precision

→ "'Mixed Precision Training", P. Micikevicius et al., ICLR 2018

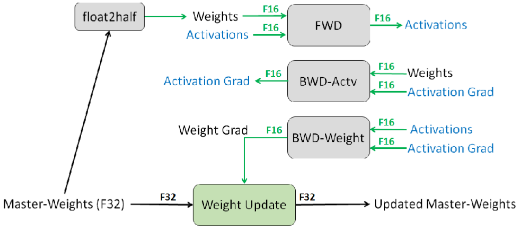
1. Compute forward backward in FP16
2. At update transform gradient to FP32
3. Update the weights in FP32



Mixed Precision

→ "'Mixed Precision Training", P. Micikevicius et al., ICLR 2018

1. Compute forward backward in FP16
2. At update transform gradient to FP32
3. Update the weights in FP32



→ Most libraries offer tools for automatic Mixed precision

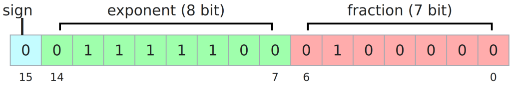
Main issue training in FP16:

Low value not well represented

- We don't care of the precision ?
- We want to encode low values (for training)

BF16

- Using more bits for the exponenent
- Using less bits for the inner range values



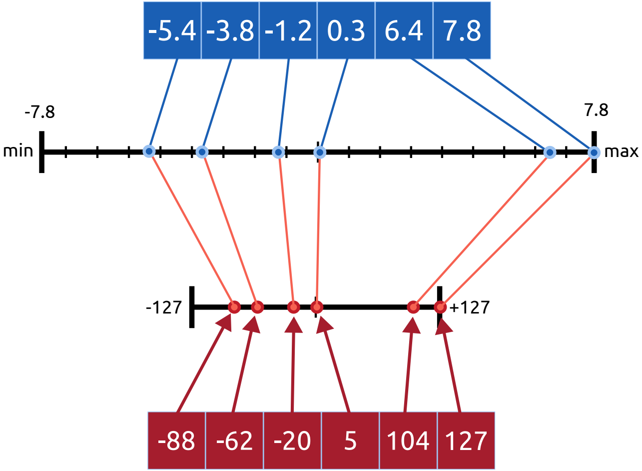
Not really PEFT: Quantization

Can we train/predict using lower precision ?

Using 8 bits ?

$\rightarrow$ 256 values (-127 to +127)!!!

- Mapping FP32 to 8 bits
- Using a scaling factor

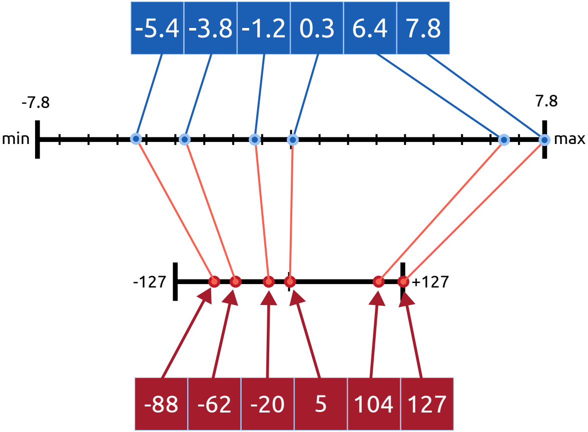


Principle

Let have W a matrix containing as maximum absolute value 7.8. a range of 1 could be represented :

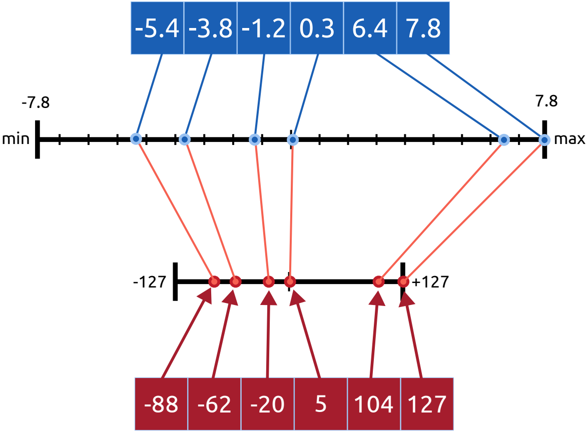
$$s = \frac{127}{7.8} \approx 16$$

values



Quantization (example)

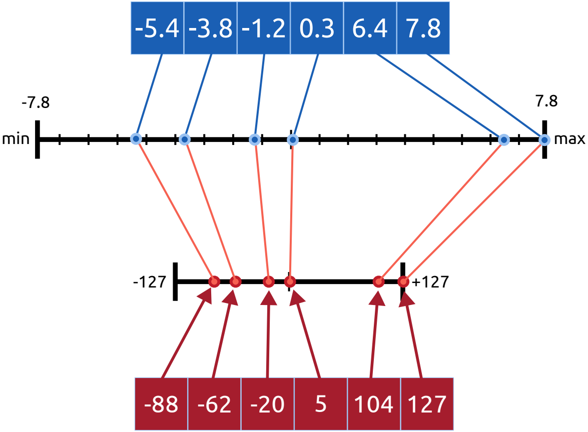
Let $s = \frac{127}{7.8}$



Quantization (example)

Let $s = \frac{127}{7.8}$

- $-5.4 \times s \rightarrow -round(87.9) = -88$

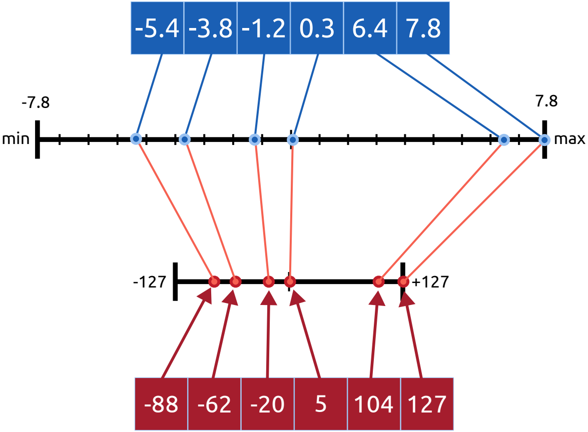


Quantization (example)

Let $s = \frac{127}{7.8}$

- $-5.4 \times s \to -round(87.9) = -88$
- $-3.8 \times s \to -round(61.8) = -62$

Not really PEFT: Quantization

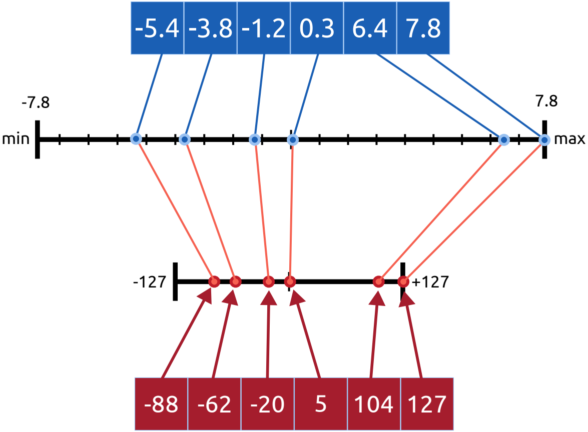


Quantization (example)

Let $s = \frac{127}{7.8}$

- $-5.4 \times s \rightarrow -round(87.9) = -88$
- $-3.8 \times s \rightarrow -round(61.8) = -62$
- $-1.2 \times s \rightarrow -round(19.5) = -20$

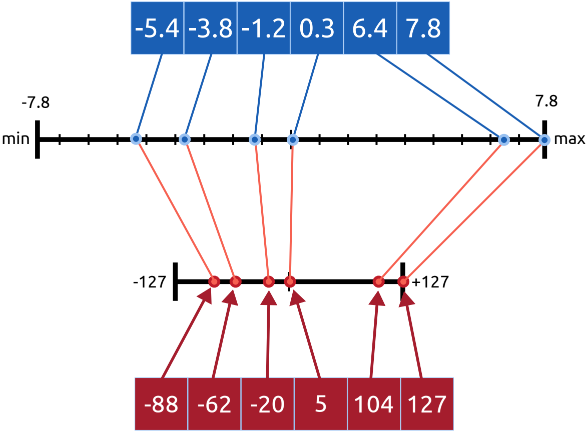
Not really PEFT: Quantization



Quantization (example)

Let $s = \frac{127}{7.8}$

- $-5.4 \times s \rightarrow -round(87.9) = -88$
- $-3.8 \times s \rightarrow -round(61.8) = -62$
- $-1.2 \times s \rightarrow -round(19.5) = -20$
- $0.3 \times s \rightarrow round(4.9) = 5$

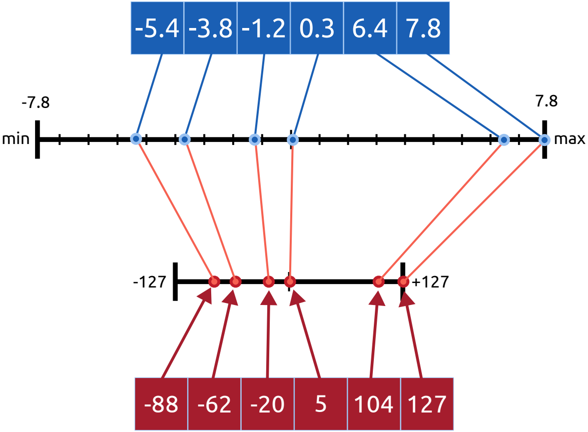


Quantization (example)

Let $s = \frac{127}{7.8}$

- $-5.4 \times s \rightarrow -round(87.9) = -88$
- $-3.8 \times s \rightarrow -round(61.8) = -62$
- $-1.2 \times s \rightarrow -round(19.5) = -20$
- $0.3 \times s \rightarrow round(4.9) = 5$
- $6.4 \times s \rightarrow round(104.2) = 104$

Not really PEFT: Quantization

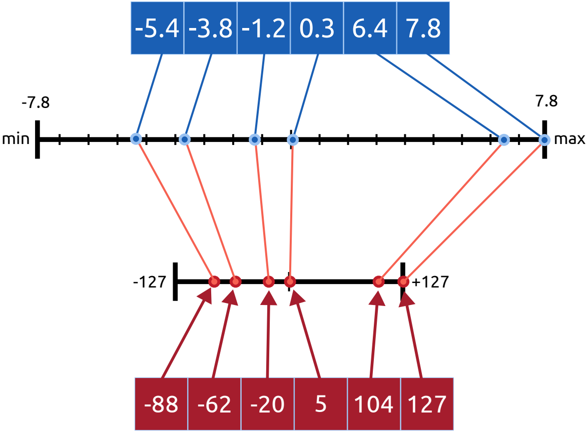


Quantization (example)

Let $s = \frac{127}{7.8}$

- $-5.4 \times s \rightarrow -round(87.9) = -88$
- $-3.8 \times s \rightarrow -round(61.8) = -62$
- $-1.2 \times s \rightarrow -round(19.5) = -20$
- $0.3 \times s \rightarrow round(4.9) = 5$
- $6.4 \times s \rightarrow round(104.2) = 104$
- $7.8 \times s \rightarrow round(127) = 127$

Not really PEFT: Quantization



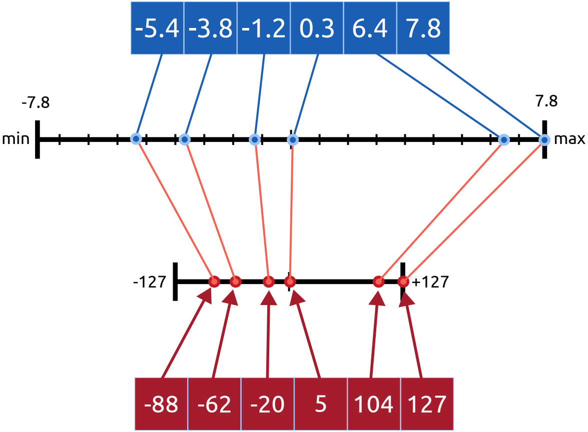
Quantization (example)

Let $s = \frac{127}{7.8}$

- $-5.4 \times s \rightarrow -round(87.9) = -88$
- $-3.8 \times s \rightarrow -round(61.8) = -62$
- $-1.2 \times s \rightarrow -round(19.5) = -20$
- $0.3 \times s \rightarrow round(4.9) = 5$
- $6.4 \times s \rightarrow round(104.2) = 104$
- $7.8 \times s \rightarrow round(127) = 127$

$\rightarrow$ and 7.77 ?

Not really PEFT: Quantization



Quantization (example)

Let $s = \frac{127}{7.8}$

- $-5.4 \times s \rightarrow -round(87.9) = -88$
- $-3.8 \times s \rightarrow -round(61.8) = -62$
- $-1.2 \times s \rightarrow -round(19.5) = -20$
- $0.3 \times s \rightarrow round(4.9) = 5$
- $6.4 \times s \rightarrow round(104.2) = 104$
- $7.8 \times s \rightarrow round(127) = 127$

$\rightarrow$ and 7.77 ? $7.8 \times s = 126.5 \rightarrow 127$

Quantization in neural networks

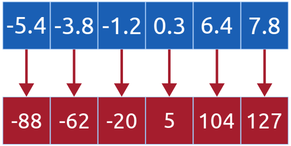
- Store the weigths quantized
- Make the computation with dequantized weights

$\rightarrow$ Need to have the reverse operation !!

Not really PEFT: De-Quantization

De-Quantization (example)

We quantized as following:

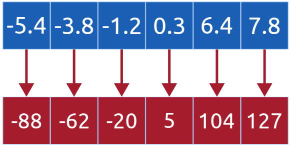


To de-quantize we need to divide by the sclaing factor s

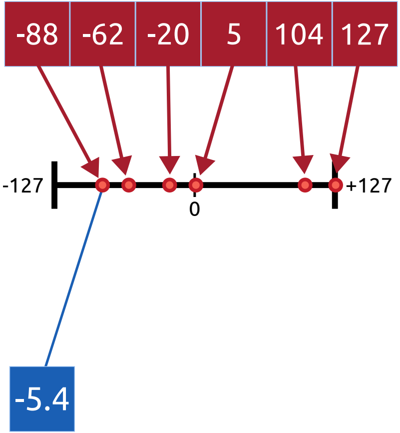
Not really PEFT: De-Quantization

De-Quantization (example)

We quantized as following:



To de-quantize we need to divide by the sclaing factor s

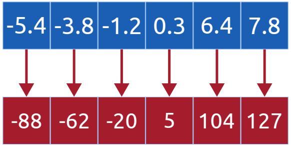


$$\mathbf{s=16.28}$$

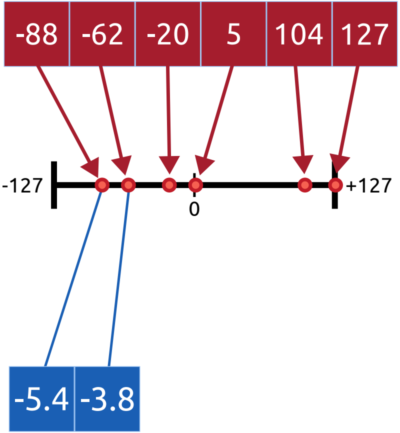
$$\frac{-88}{s} = -5.41$$

De-Quantization (example)

We quantized as following:



To de-quantize we need to divide by the sclaing factor s



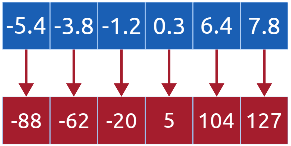
$\mathbf{s=16.28}$

$$\frac{-88}{s} = -5.41$$
$$\frac{-62}{s} = -3.81$$

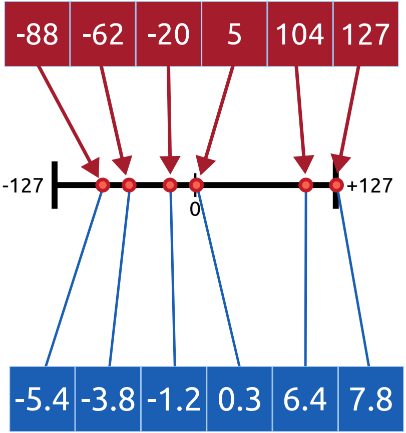
Not really PEFT: De-Quantization

De-Quantization (example)

We quantized as following:



To de-quantize we need to divide by the sclaing factor s



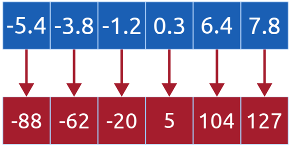
$\mathbf{s=16.28}$

$$\frac{-88}{s} = -5.41$$
$$\frac{-62}{s} = -3.81$$
$$\frac{-20}{s} = -1.23$$
$$\frac{5}{s} = 0.31$$
$$\frac{104}{s} = 6.39$$
$$\frac{127}{s} = 7.80$$

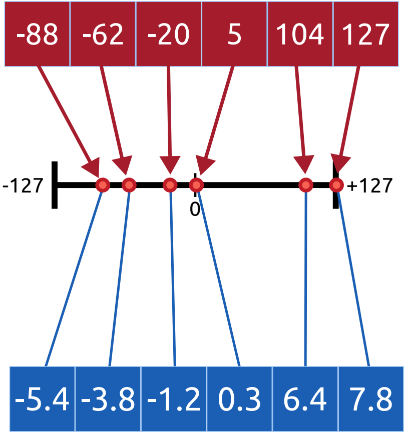
Not really PEFT: Symetric Quantization

De-Quantization (example)

We quantized as following:

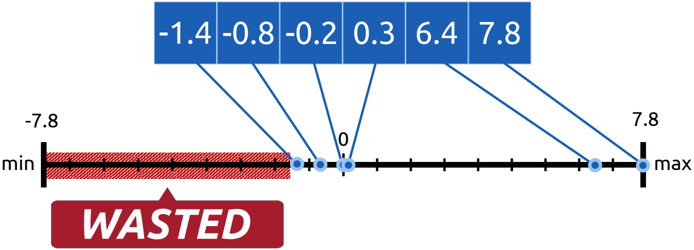


To de-quantize we need to divide by the sclaing factor s



$\mathbf{s{=}16.28}$

$$\frac{-88}{s} = -5.41$$
$$\frac{-62}{s} = -3.81$$
$$\frac{-20}{s} = -1.23$$
$$\frac{5}{s} = 0.31$$
$$\frac{104}{s} = 6.39$$
$$\frac{127}{s} = 7.80$$



Asymetric Quantization:

Make quantization non-centered on zero, need the scale factor $s = \frac{2^b - 1}{v_{max} - v_{min}}$ and an offset

Better quantization

- Different possible quantization (8bits, 4 bits,...)
- “Double Quantization” (for block based quantization)
- “GPTQ (General Pre-Trained Transformer Quantization)”
- ...

Not really PEFT: Quantization (code)

Using HuggingFace and bitsandbytes

```python
from transformers import AutoModelForCausalLM, BitsAndBytesConfig

quantization_config = BitsAndBytesConfig(load_in_4bit=True)
model_4bit = AutoModelForCausalLM.from_pretrained(
    "openai-community/gpt2",
    quantization_config=quantization_config
)
```

Not really PEFT: Quantization (Conclusion)

Conclusion
In this lecture we introduce quantization

- Quantization allow to decrease model size (lower memory consumption)
- Quantization is usefull for inference
- Most libraries propose quantization tools

Quantization $\rightarrow$ Evaluate/testing a large model

Online ressources ?

- `https://newsletter.maartengrootendorst.com/p/a-visual-guide-to-quantization` (inspiration for my figures)
-

Quantization and PEFT methods : QLoRA

Training and quantization

- Quantization inference can be faster
- Quantization inference is lighter (RAM)
- ⚠ Training is unstable

Unstable training ?

$\rightarrow$ Mainly due to weigth update !!

Quantization and PEFT methods : QLoRA

Training and quantization

- Quantization inference can be faster
- Quantization inference is lighter (RAM)
- ⚠ Training is unstable

Unstable training ?

$\rightarrow$ Mainly due to weigth update !!
$\rightarrow$ In **adapter**, only few weights are updated

Quantization and PEFT methods : QLoRA

Training and quantization

- Quantization inference can be faster
- Quantization inference is lighter (RAM)
- ⚠ Training is unstable

Unstable training ?

→ Mainly due to weigth update !!
→ In **adapter**, only few weights are updated

Use quantization for non-updated weights ?

Quantization and PEFT methods : QLoRA

Training and quantization

- Quantization inference can be faster
- Quantization inference is lighter (RAM)
- ⚠ Training is unstable

Unstable training ?

→ Mainly due to weigth update !!

→ In **adapter**, only few weights are updated

Use quantization for non-updated weights ?

QLORA: Efficient Finetuning of Quantized LLMs

Tim Dettmers* **Artidoro Pagnoni*** **Ari Holtzman**

Luke Zettlemoyer

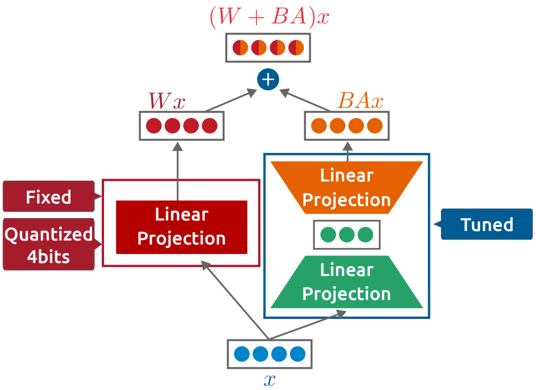
University of Washington
{dettmers,artidoro,ahai,lsz}@cs.washington.edu

Abstract

We present QLORA, an efficient finetuning approach that reduces memory usage enough to finetune a 65B parameter model on a single 48GB GPU while preserving full 16-bit finetuning task performance. QLORA backpropagates gradients through a frozen, 4-bit quantized pretrained language model into Low Rank Adapters (LoRA). Our best model family, which we name **Guanaco**, outperforms all previous openly released models on the Vicuna benchmark, reaching 99.3% of the performance level of ChatGPT while only requiring 24 hours of finetuning on a single GPU. QLORA introduces a number of innovations to save memory without sacrificing performance: (a) 4-bit NormalFloat (NF4), a new data type that is information theoretically optimal for normally distributed weights (b) Double Quantization to reduce the average memory footprint by quantizing the quantization constants, and (c) Paged Optimizers to manage memory spikes. We use QLORA to finetune more than 1,000 models, providing a detailed analysis of instruction following and chatbot performance across 8 instruction datasets, multiple model types (LLaMA, T5), and model scales that would be infeasible to run with regular finetuning (e.g. 33B and 65B parameter models). Our results show that QLoRA

QLoRA training

- Use LoRA adapter framework
- Quantize the original weights (no need gradient)
- Different optimizations techniques (double quantization)



Dataset	GLUE (Acc.)	Super-NaturalInstructions (RougeL)				
Model	RoBERTa-large	T5-80M	T5-250M	T5-780M	T5-3B	T5-11B
BF16	88.6	40.1	42.1	48.0	54.3	62.0
BF16 replication	88.6	40.0	42.2	47.3	54.9	-
LoRA BF16	88.8	40.5	42.6	47.1	55.4	60.7
QLORA Int8	88.8	40.4	42.9	45.4	56.5	60.7
QLORA FP4	88.6	40.3	42.4	47.5	55.6	60.9
QLORA NF4 + DQ	-	40.4	42.7	47.7	55.3	60.9

Quantization and PEFT methods : QLoRA

Dataset	GLUE (Acc.)	Super-NaturalInstructions (RougeL)				
Model	RoBERTa-large	T5-80M	T5-250M	T5-780M	T5-3B	T5-11B
BF16	88.6	40.1	42.1	48.0	54.3	62.0
BF16 replication	88.6	40.0	42.2	47.3	54.9	-
LoRA BF16	88.8	40.5	42.6	47.1	55.4	60.7
QLORA Int8	88.8	40.4	42.9	45.4	56.5	60.7
QLORA FP4	88.6	40.3	42.4	47.5	55.6	60.9
QLORA NF4 + DQ	-	40.4	42.7	47.7	55.3	60.9

$\rightarrow$ Competitive results with LoRa

Conclusion

- How to adapt for a lower cost (memory and speed)
- How to compress/reduce size of models
- Two different approach/family:
  - Adapter based
  - Prefix based

$\rightarrow$ Do we always need to modify/compress weigth (for adaptation) ?

Adaptation: Need to modify the model ?

“Language Models are Few-Shot Learners” B. Brown, NeurIPS 2020

One-shot

In addition to the task description, the model sees a single example of the task. No gradient updates are performed.

```
Translate English to French:     ← task description
sea otter => loutre de mer       ← example
cheese => ..........             ← prompt
```

Few-shot

In addition to the task description, the model sees a few examples of the task. No gradient updates are performed.

```
Translate English to French:         ← task description
sea otter => loutre de mer           ← examples
peppermint => menthe poivrée         ←
plush girafe => girafe peluche       ←
cheese => ..........                 ← prompt
```

→ In context learning

Adaptation: Need to modify the model ?

"Chain-of-Thought Prompting Elicits Reasoning in Large Language Models" J. Wei et al., NeurIPS 2022

Standard Prompting	Chain-of-Thought Prompting
Model Input Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now? A: The answer is 11. Q: The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?	**Model Input** Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now? A: Roger started with 5 balls. 2 cans of 3 tennis balls each is 6 tennis balls. 5 + 6 = 11. The answer is 11. Q: The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?
Model Output A: The answer is 27. ✗	**Model Output** A: The cafeteria had 23 apples originally. They used 20 to make lunch. So they had 23 - 20 = 3. They bought 6 more apples, so they have 3 + 6 = 9. The answer is 9. ✓

Optimisation in Transformer: Attention mechanism

Optimisation in transformer architecture

Some performances issues :

- The number of parameters is too large
- The attention is Quadratic (sequence length)
- ...

How to accelerate inference ?

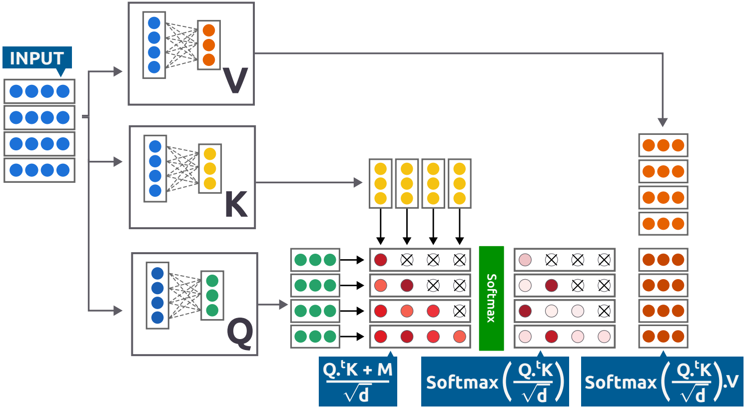


Figure 4: Illustration of the decoder model attention

Multhead Attention

- Multiple attention head
- Each having specific:
  - Query
  - Key
  - Value

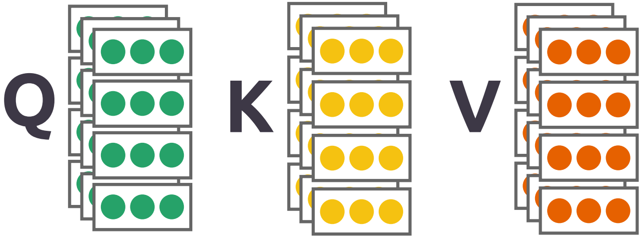


Figure 5: Illustration of the multihead attention

Optimisation in transformer architecture: KVCache

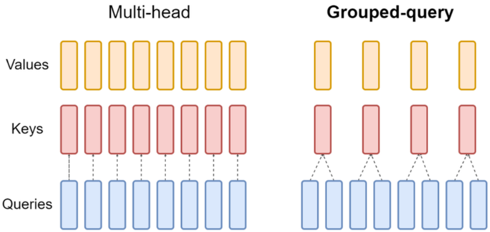
Key Value Cache

- Storing the keys and previous values
- Store value to compute new token representation

Token	KV produced	cache
1	k_1, v_1	k_1, v_1
2	k_2, v_2	k_1, v_1, k_2, v_2
$\vdots$	$\vdots$	$\vdots$
n	k_n, v_n	$k_1, v_1, k_2, v_2 \ldots, k_n, v_n$

Gouped queries

- Limiting the number of stored parameters
- Restraining the number key/values heads (one per query group) [Ain+23]



[Ain+23] - Joshua Ainslie et al. "Gqa: Training generalized multi-query transformer models from multi-head checkpoints". In: *arXiv preprint arXiv:2305.13245* (2023)

In decoder approaches:

Use self-attention (or global attention)

- For each token compute a score with all previous tokens
- Quadratic storage (and time)

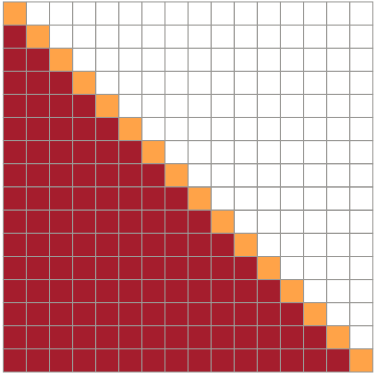


Figure 6: Illustration of the global attention matrix

Optimisation in transformer architecture: local Attention

Use of local attention mechanism

- Only process full attention on fixed size segment
- Principle of sliding window

This principle is used in many works [BPC20]

[BPC20] - Iz Beltagy, Matthew E Peters, and Arman Cohan. "Longformer: The long-document transformer". In: *arXiv preprint arXiv:2004.05150* (2020)

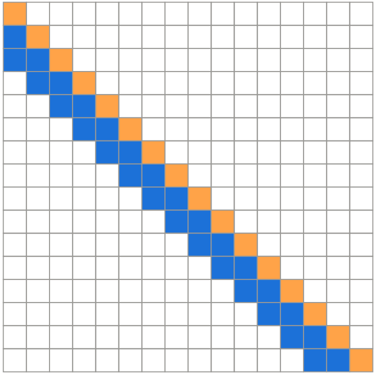


Figure 7: Illustration of the local attention matrix

Example

Let consider:

- Three layers transformer
- 4 windowed attention

What token for 11^{th} output is ”taken” in consideration ?

Example

Let consider:

- Three layers transformer
- 4 windowed attention

What token for 11^{th} output is "taken" in consideration ?

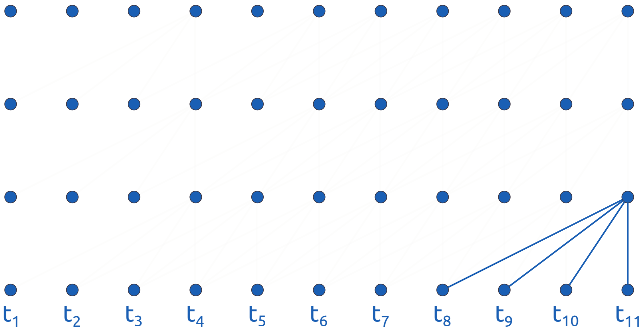


Figure 8: Receptieve field in windowed attention

Example

Let consider:

- Three layers transformer
- 4 windowed attention

What token for 11^{th} output is ”taken” in consideration ?

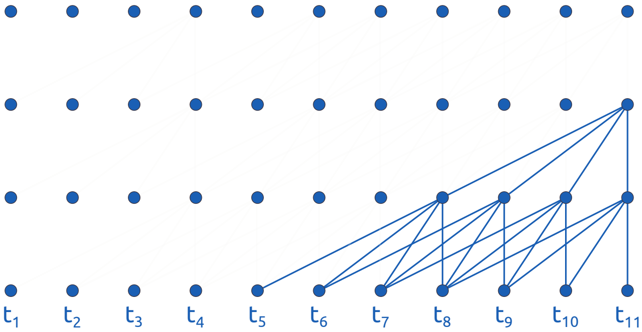


Figure 8: Receptieve field in windowed attention

Example

Let consider:

- Three layers transformer
- 4 windowed attention

What token for 11^{th} output is "taken" in consideration ?

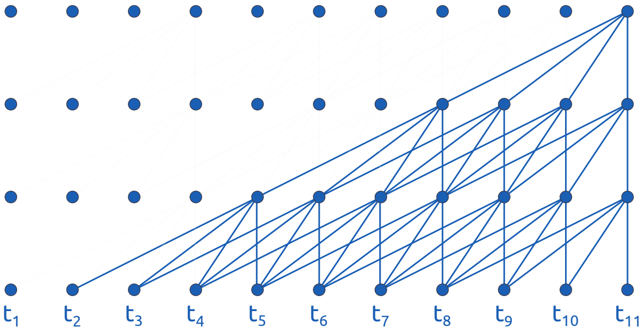


Figure 8: Receptieve field in windowed attention

Example

Let consider:

- Three layers transformer
- 4 windowed attention

What token for 11^{th} output is "taken" in consideration ?
$\rightarrow$ from 2 to 11

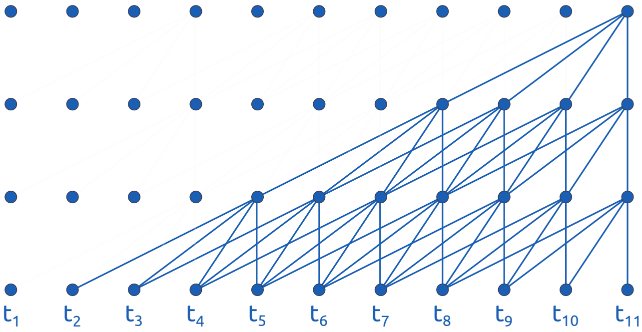
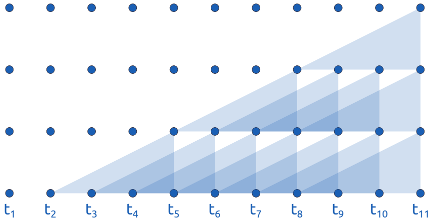


Figure 8: Receptieve field in windowed attention

Optimisation in transformer architecture: local Attention

Receptieve field size:

$$(window\_size - 1) \times nb\_layer + 1$$



Advantages

- Longer sequences for lower computationnal/memory cost
- Lower performances

Optimisation in transformer architecture: Dillated Attention

Use of dillated Attention mechanism

- Principle of dillated sliding window

Allows to reach more distant tokens

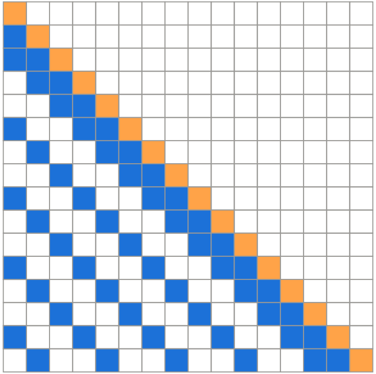


Figure 9: Illustration of the dilated attention matrix

Using different sparse attention

- BigBird (Random + Local + Global) [Zah+20]
- Longformer (Dilated + Local + Global) [BPC20]
- Attention Sink [Xia+23]

→ Few large model use it...

[Zah+20] - Manzil Zaheer et al. "Big bird: Transformers for longer sequences". In: *Advances in neural information processing systems* 33 (2020), pp. 17283–17297

[BPC20] - Iz Beltagy, Matthew E Peters, and Arman Cohan. "Longformer: The long-document transformer". In: *arXiv preprint arXiv:2004.05150* (2020)

[Xia+23] - Guangxuan Xiao et al. "Efficient streaming language models with attention sinks". In: *arXiv preprint arXiv:2309.17453* (2023)

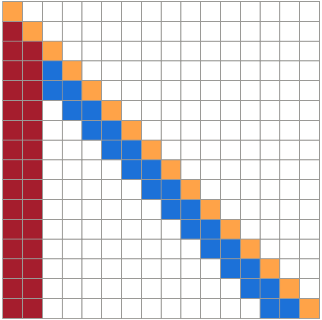


Figure 10: Mxing global and local attention

We recall that the attention formula (for encoder) is given by :

$$O = softmax(\frac{Q.^tK}{\sqrt{D}}).V$$

Considering Q, K, and V $\in R^{L\times D}$. If we consider O_l the output associated to the l^{th} "token":

$$\begin{aligned} O_l &= softmax(\frac{\sum_{j=1}^{L} Q_l.K_j}{\sqrt{D}}) \\ &= softmax(\frac{Q_l.^tK}{\sqrt{D}}).V \end{aligned}$$

Linear attention

We recall that the attention formula (for encoder) is given by :

$$O = softmax(\frac{Q.{}^tK}{\sqrt{D}}).V$$

Considering Q, K, and V $\in R^{L\times D}$. If we consider O_l the output associated to the l^{th} "token":

$$O_l = \left[softmax\left(\frac{Q_l.{}^tK}{\sqrt{D}}\right).V \right]_l$$

Linear attention

Let consider that attention is no more computed with softmax but only using the formula (that also provide a distribution):

$$O = \frac{\frac{Q.^tK}{\sqrt{D}}}{\sum_{j=0}^{L} (\frac{Q.^tK}{\sqrt{D}})_j}.V$$

We can write O_l as it follows :

$$O_l = \frac{\frac{Q_l.^tK}{\sqrt{D}}}{\sum_{j=0}^{L} (\frac{Q_l.^tK}{\sqrt{D}})_j}.V$$

$$= \frac{\frac{Q_l}{\sqrt{D}}.\frac{^tK}{\sqrt{D}}.V}{\sum_{j=0}^{L} (\frac{Q_l}{\sqrt{D}}.\frac{^tK}{\sqrt{D}})_j} = \frac{\frac{Q_l}{\sqrt{D}}.\left(\frac{^tK}{\sqrt{D}}.V\right)}{\frac{Q_l}{\sqrt{D}}\sum_{j=0}^{L} (\frac{K}{\sqrt{D}})_j}$$

- $KV = \frac{^tK}{\sqrt{D}}.V$ imply $L \times D \times D$ operations
- $\frac{Q_l}{\sqrt{D}}.KV$ imply $D \times D$ operations

To compute this attention for all Q_i we repeat L times the operations...

$\rightarrow$ We can computes KV once !!!

Linear attention

Let rewrite with KV and with $S = \sum_{j=0}^{L} (\frac{K}{\sqrt{D}})_j$, $S \in \mathbb{R}^{D \times D}$:

$$\frac{A}{B} = \frac{\frac{Q}{\sqrt{D}}.KV}{\frac{Q}{\sqrt{D}}.S}$$

Then the cost is :

- Computation of S imply $N \times D$ operations
- Computation of KV imply $N \times D \times D$ operations
- Computation A imply $L \times D \times D$ operations (with KV fixed)
- Computation of B imply $L \times D \times D$ operations (with S fixed)

The overall cost is then $\mathcal{O}(LD^2)$ (still quadratic but to embedding dimension)

Linear attention and softmax:

Notice that it can be computed similarly (time/memory) for

$$\frac{\phi(Q)\psi({}^tK)}{\sum_{j=0}^{L}(\phi(Q)\psi({}^tK))_j}.V$$

$\rightarrow$ Replace with function close to softmax :

- ϕ, ψ being $1 + elu(x)$ [Kat+20]
- Many other kernels...

[0][Kat+20] - Angelos Katharopoulos et al. “Transformers are rnns: Fast autoregressive transformers with linear attention”. In: *International conference on machine learning*. PMLR. 2020, pp. 5156–5165

Linear attention and decoder

In autoregressive models, we can only access previous tokens…

- $\rightarrow$ For O_l we must compute ${}^tK_{:,0:t}V_{0:t}$
- $\rightarrow$ Cannot compute once for all

Can be done using a loop for similar complexity
$\rightarrow$ Not parallelizable (not using full capacity of GPUs architecture)…

Conclusion

- Fine-tuning time complexity can be alleviate
    - Adapter approaches
    - Different decoding
- Problem of attention and decoding
-
    - Storing previous output...
    - Grouping operations (Groups Query)
    - Sparse attention mechanisms
- Still lot of issues to deal with

Implementing Adaptation Approaches

Using low level libraries

- Harder to implement (more verbose)
- Flexible, greater understanding
- Possible to devellop new approaches

$\rightarrow$ Pytorch

Using high level libraries

- Easy to implement
- Missing flexibility

$\rightarrow$ transformers, PEFT

Finding pytorch code

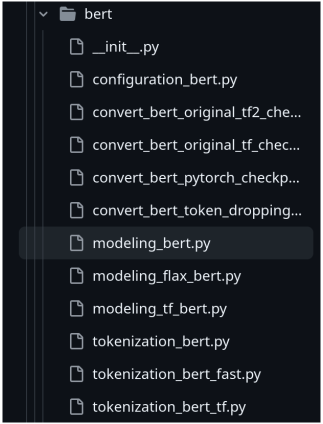
Hopfully, most models are implemented in pytorch

- https://github.com/huggingface/transformers/tree/main/src/transformers/models

Finding pytorch code

Hopfully, most models are implemented in pytorch

- `https://github.com/huggingface/transformers/tree/main/src/transformers/models`



First step

We will choose the model Bert-base-uncase for sentiment analysis

- import transformers lib
- Load the model and the tokenizer

```
    def __init__(self, config, add_pooling_layer=True):
        super().__init__(config)
        self.config = config

        self.embeddings = BertEmbeddings(config)
        self.encoder = BertEncoder(config)

        self.pooler = BertPooler(config) if add_pooling_layer else None
```

What is the architecture of the model ?

$\rightarrow$ We can print the model

LoRA with pytorch

What is the architecture of the model ?

$\rightarrow$ We can print the model

```
BertForSequenceClassification(
 (bert): BertModel(
   (embeddings): BertEmbeddings(
     (word_embeddings): Embedding(28996, 768, padding_idx=0)
     (position_embeddings): Embedding(512, 768)
     (token_type_embeddings): Embedding(2, 768)
     (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
     (dropout): Dropout(p=0.1, inplace=False)
   )
   (encoder): BertEncoder(
     (layer): ModuleList(
       (0-11): 12 x BertLayer(
         (attention): BertAttention(
           (self): BertSdpaSelfAttention(
             (query): Linear(in_features=768, out_features=768, bias=True)
             (key): Linear(in_features=768, out_features=768, bias=True)
             (value): Linear(in_features=768, out_features=768, bias=True)
             (dropout): Dropout(p=0.1, inplace=False)
           )
           (output): BertSelfOutput(
             (dense): Linear(in_features=768, out_features=768, bias=True)
             (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
             (dropout): Dropout(p=0.1, inplace=False)
...
  )
  (dropout): Dropout(p=0.1, inplace=False)
  (classifier): Linear(in_features=768, out_features=2, bias=True)
)
```

LoRA with pytorch

Defining the module

Defining a module that will replace original linear module

```python
import torch
from torch import nn

class LoRALinear(nn.Module):
    def __init__(
        self, in_dim: int, out_dim: int, rank: int
    ):
        super().__init__()
        self.linear = nn.Linear(in_dim, out_dim, bias=True)

        self.lora_a = nn.Linear(in_dim, rank, bias=False)
        self.lora_b = nn.Linear(rank, out_dim, bias=False)

        self.linear.weight.requires_grad = False
        self.lora_a.weight.requires_grad = True
        self.lora_b.weight.requires_grad = True

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        frozen_out = self.linear(x)
        lora_out = self.lora_b(self.lora_a(x))

        return frozen_out + lora_out
```

A function for replace module

We want to define a function that will replace the original linear by the LoRALinear

```python
def linear_to_lora(linear):
    linear_weight = linear.weight.data
    has_bias = linear.bias is not None
    if has_bias:
        linear_bias = linear.bias.data
    output_size, input_size =  linear_weight.shape
    lora = LoRALinear(input_size, output_size, rank=8)
    lora.linear.weight.data = linear_weight
    if has_bias:
        lora.linear.weight.data = linear_bias
    return lora
```

$\rightarrow$ Where are the modules we want to replace ?

LoRA with pytorch

→ Where are the modules we want to replace ?

```
[k for k,v  in model.bert.named_parameters()]
```

→ Where are the modules we want to replace ?

```
[k for k,v  in model.bert.named_parameters()]
```

```
['embeddings.word_embeddings.weight',
 'embeddings.position_embeddings.weight',
 'embeddings.token_type_embeddings.weight',
 'embeddings.LayerNorm.weight',
 'embeddings.LayerNorm.bias',
 'encoder.layer.0.attention.self.query.weight',
 'encoder.layer.0.attention.self.query.bias',
 'encoder.layer.0.attention.self.key.weight',
 'encoder.layer.0.attention.self.key.bias',
 'encoder.layer.0.attention.self.value.weight',
 'encoder.layer.0.attention.self.value.bias',
 'encoder.layer.0.attention.output.dense.weight',
 'encoder.layer.0.attention.output.dense.bias',
 'encoder.layer.0.attention.output.LayerNorm.weight',
```

Replacing modules

- Replace the module in the original model (or copy)

```
lora_model = copy.deepcopy(model)
lora_parameters = []
for block in lora_model.bert.encoder.layer:
    block.attention.self.key = linear_to_lora(block.attention.self.key)
    block.attention.self.value = linear_to_lora(block.attention.self.value)
    block.attention.self.query= linear_to_lora(block.attention.self.query)
```

Set trainable parameters

- Only lora parameters need gradient
- Notice here that we also train the linear classifier

```
for k,v  in lora_model.bert.named_parameters():
    if ('lora' in k):
        v.requires_grad = True
    else:
        v.requires_grad = False
```

What is the architecture of the model ?

$\rightarrow$ We can print the model

What is the architecture of the model ?

$\rightarrow$ We can print the model

```
BertForSequenceClassification(
 (bert): BertModel(
   (embeddings): BertEmbeddings(
     (word_embeddings): Embedding(28996, 768, padding_idx=0)
     (position_embeddings): Embedding(512, 768)
     (token_type_embeddings): Embedding(2, 768)
     (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
     (dropout): Dropout(p=0.1, inplace=False)
   )
   (encoder): BertEncoder(
     (layer): ModuleList(
       (0-11): 12 x BertLayer(
         (attention): BertAttention(
           (self): BertSdpaSelfAttention(
             (query): LoRALinear(
               (linear): Linear(in_features=768, out_features=768, bias=True)
               (lora_a): Linear(in_features=768, out_features=8, bias=False)
               (lora_b): Linear(in_features=8, out_features=768, bias=False)
             )
             (key): LoRALinear(
               (linear): Linear(in_features=768, out_features=768, bias=True)
               (lora_a): Linear(in_features=768, out_features=8, bias=False)
               (lora_b): Linear(in_features=8, out_features=768, bias=False)
```

LoRA with pytorch: Training

Training the model !!!

Training the model !!!

```python
from transformers import TrainingArguments, Trainer

training_args = TrainingArguments(output_dir="test_custom_lora",
                                  eval_strategy="steps",
                                  eval_steps= 128,
                                  num_train_epochs=2,)
trainer = Trainer(
    model=lora_model,
    args=training_args,
    train_dataset= training_set,
    eval_dataset=validation_set,
    compute_metrics=compute_metrics,
)
trainer.train()
```

LoRA with pytorch: Training

Training the model !!!

```python
from transformers import TrainingArguments, Trainer

training_args = TrainingArguments(output_dir="test_custom_lora",
                                  eval_strategy="steps",
                                  eval_steps= 128,
                                  num_train_epochs=2,)
trainer = Trainer(
    model=lora_model,
    args=training_args,
    train_dataset= training_set,
    eval_dataset=validation_set,
    compute_metrics=compute_metrics,
)
trainer.train()
```

[1250/1250 02:51, Epoch 2/2]

Step	Training Loss	Validation Loss	Accuracy
128	No log	0.677891	0.566000
256	No log	0.614952	0.684000
384	No log	0.502475	0.774000
512	0.619000	0.418735	0.815000
640	0.619000	0.381323	0.836000
768	0.619000	0.371402	0.835000
896	0.619000	0.354594	0.848000
1024	0.388000	0.347619	0.851000
1152	0.388000	0.342413	0.855000

LoRA with pytorch: conlusion

- Easy to implement a LoRA adapter
- Can be improved (see LoRA+)

LoRA with PEFT

→ HuggingFace face proposes the PEFT library !!!

```python
from peft import LoraConfig, TaskType,  get_peft_model

lora_config = LoraConfig(
    task_type=TaskType.SEQ_CLS, r=1, lora_alpha=1, lora_dropout=0.1
)

model = BertForSequenceClassification.from_pretrained(
    'bert-base-cased',
    num_labels=2
)
peft_model = get_peft_model(model, lora_config)
```

→ HuggingFace face proposes the PEFT library !!!

```python
from peft import LoraConfig, TaskType,  get_peft_model

lora_config = LoraConfig(
    task_type=TaskType.SEQ_CLS, r=1, lora_alpha=1, lora_dropout=0.1
)

model = BertForSequenceClassification.from_pretrained(
    'bert-base-cased',
    num_labels=2
)
peft_model = get_peft_model(model, lora_config)
```

And train the model !!!

Conlusion

To conclude

- A short example of training with LoRA
- It can be easily adapted for *Adapter*

Questions ?