

# LLMs: Fine-tuning & Optimisation

---

Thomas Gerald

February 6, 2026

# Language Model

---

## Language Model (LM)

A language model is a statistical model of the distribution of linguistic units (e.g., letters, phonemes, words) in a natural language. A language model can, for example, predict the next word in a sequence of words.

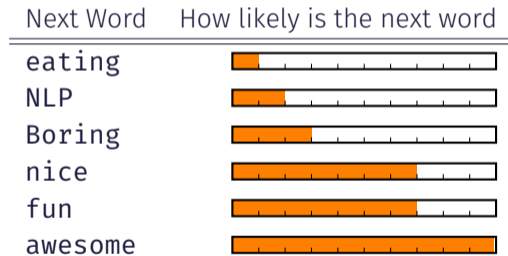
 Do not assume that Language Models are LLMs

The electorate is constitutionally empowered to vote on amendments passed by the Parliament and bills submitted by the president.

# Language Model : How to model ?

Study NLP is [???

Estimate next token probability:



# Language Model : The bi-gram model

Bi-gram model:

**Hypothesis** → One *token* depends mostly from its previous tokens.

**Model** →  $p_{\theta}(x_{t+1}|x_t)$

In practice :

- Consider a textual corpus
- Consider a vocabulary  $V$  (tokens units in the corpus)

→ Count all successive pairs of words

The language model is the matrix  $|V| \times |V|$

# Language Model : The bi-gram model

	Study	NLP	is	awesome	Boring
Study	0	4	0	0	0
NLP	0	0	4	0	0
is	0	0	0	8	2
awesome	0	0	0	0	0
Boring	0	0	0	0	0

# Language Model : The bi-gram model

	Study	NLP	is	awesome	Boring
Study	0	4	0	0	0
NLP	0	0	4	0	0
is	0	0	0	8	2
awesome	0	0	0	0	0
Boring	0	0	0	0	0

Compute associate probabilities :

# Language Model : The bi-gram model

Prompt  $\rightarrow x_1 = \text{Study}$

	Study	NLP	is	awesome	Boring
Study	0	4	0	0	0
NLP	0	0	4	0	0
is	0	0	0	8	2
awesome	0	0	0	0	0
Boring	0	0	0	0	0

Compute associate probabilities :

	Study	NLP	is	awesome	Boring
Study	0	1.	0	0	0
NLP	0	0	1.	0	0
is	0	0	0	.8	.2
awesome	0	0	0	0	0
Boring	0	0	0	0	0

# Language Model : The bi-gram model

	Study	NLP	is	awesome	Boring
Study	0	4	0	0	0
NLP	0	0	4	0	0
is	0	0	0	8	2
awesome	0	0	0	0	0
Boring	0	0	0	0	0

Compute associate probabilities :

	Study	NLP	is	awesome	Boring
Study	0	1.	0	0	0
NLP	0	0	1.	0	0
is	0	0	0	.8	.2
awesome	0	0	0	0	0
Boring	0	0	0	0	0

Prompt  $\rightarrow x_1 = \text{Study}$

•  $x_2 \sim P(y|x = \text{"Study"})$

$\rightarrow x_2 = \text{"NLP"}$

# Language Model : The bi-gram model

	Study	NLP	is	awesome	Boring
Study	0	4	0	0	0
NLP	0	0	4	0	0
is	0	0	0	8	2
awesome	0	0	0	0	0
Boring	0	0	0	0	0

Compute associate probabilities :

	Study	NLP	is	awesome	Boring
Study	0	1.	0	0	0
NLP	0	0	1.	0	0
is	0	0	0	.8	.2
awesome	0	0	0	0	0
Boring	0	0	0	0	0

Prompt  $\rightarrow x_1 = \text{Study}$

- $x_2 \sim P(y|x = \text{"Study"})$

$\rightarrow x_2 = \text{"NLP"}$

- $x_3 \sim P(y|x = \text{"NLP"})$

$\rightarrow x_3 = \text{"is"}$

# Language Model : The bi-gram model

	Study	NLP	is	awesome	Boring
Study	0	4	0	0	0
NLP	0	0	4	0	0
is	0	0	0	8	2
awesome	0	0	0	0	0
Boring	0	0	0	0	0

Compute associate probabilities :

	Study	NLP	is	awesome	Boring
Study	0	1.	0	0	0
NLP	0	0	1.	0	0
is	0	0	0	.8	.2
awesome	0	0	0	0	0
Boring	0	0	0	0	0

Prompt  $\rightarrow x_1 = \text{Study}$

- $x_2 \sim P(y|x = \text{"Study"})$

$\rightarrow x_2 = \text{"NLP"}$

- $x_3 \sim P(y|x = \text{"NLP"})$

$\rightarrow x_3 = \text{"is"}$

- $x_4 \sim P(y|x = \text{"is"})$

$\rightarrow x_4 = \text{"awesome"}$  with probability 0.8

$\rightarrow x_4 = \text{"Boring"}$  with probability 0.2

# Language Model : The bi-gram model

	Study	NLP	is	awesome	Boring
Study	0	4	0	0	0
NLP	0	0	4	0	0
is	0	0	0	8	2
awesome	0	0	0	0	0
Boring	0	0	0	0	0

Compute associate probabilities :

	Study	NLP	is	awesome	Boring
Study	0	1.	0	0	0
NLP	0	0	1.	0	0
is	0	0	0	.8	.2
awesome	0	0	0	0	0
Boring	0	0	0	0	0

Prompt  $\rightarrow x_1 = \text{Study}$

- $x_2 \sim P(y|x = \text{"Study"})$

$\rightarrow x_2 = \text{"NLP"}$

- $x_3 \sim P(y|x = \text{"NLP"})$

$\rightarrow x_3 = \text{"is"}$

- $x_4 \sim P(y|x = \text{"is"})$

$\rightarrow x_4 = \text{"awesome"}$  with probability 0.8

$\rightarrow x_4 = \text{"Boring"}$  with probability 0.2

# Language Model : The bi-gram model

	Study	NLP	is	awesome	Boring
Study	0	4	0	0	0
NLP	0	0	4	0	0
is	0	0	0	8	2
awesome	0	0	0	0	0
Boring	0	0	0	0	0

Compute associate probabilities :

	Study	NLP	is	awesome	Boring
Study	0	1.	0	0	0
NLP	0	0	1.	0	0
is	0	0	0	.8	.2
awesome	0	0	0	0	0
Boring	0	0	0	0	0

Prompt  $\rightarrow x_1 = \text{Study}$

- $x_2 \sim P(y|x = \text{"Study"})$

$\rightarrow x_2 = \text{"NLP"}$

- $x_3 \sim P(y|x = \text{"NLP"})$

$\rightarrow x_3 = \text{"is"}$

- $x_4 \sim P(y|x = \text{"is"})$

$\rightarrow x_4 = \text{"awesome"}$  with probability 0.8

$\rightarrow x_4 = \text{"Boring"}$  with probability 0.2

Probability of a sentence  $\rightarrow \prod_{i=1}^n p(x_{i+1}|x_i)$

$\rightarrow \text{"Study NLP is awesome"}$  with probability 0.8

# Language Model : The bi-gram model

	Study	NLP	is	awesome	Boring
Study	0	4	0	0	0
NLP	0	0	4	0	0
is	0	0	0	8	2
awesome	0	0	0	0	0
Boring	0	0	0	0	0

Compute associate probabilities :

	Study	NLP	is	awesome	Boring
Study	0	1.	0	0	0
NLP	0	0	1.	0	0
is	0	0	0	.8	.2
awesome	0	0	0	0	0
Boring	0	0	0	0	0

Prompt  $\rightarrow x_1 = \text{Study}$

- $x_2 \sim P(y|x = \text{"Study"})$

$\rightarrow x_2 = \text{"NLP"}$

- $x_3 \sim P(y|x = \text{"NLP"})$

$\rightarrow x_3 = \text{"is"}$

- $x_4 \sim P(y|x = \text{"is"})$

$\rightarrow x_4 = \text{"awesome"}$  with probability 0.8

$\rightarrow x_4 = \text{"Boring"}$  with probability 0.2

Probability of a sentence  $\rightarrow \prod_{i=1}^n p(x_{i+1}|x_i)$

$\rightarrow \text{"Study NLP is awesome"}$  with probability 0.8

$\rightarrow \text{"Study NLP is Boring"}$  with probability 0.2

## Language Model : The bi-gram model

	the	is	a	dog	cat	chases
the	0	0	0	.5	.5	0.
is	0	0	.8	.1	.1	0.
a	0	0	0	.5	.5	0.
dog	0	.5	0	0	0	.5
cat	0	.5	0	0	0	.5
chases	.5	0.	.5	0.	0.	0.

What are the most likely sentences starting with "the"?

- "the cat is a dog"
- "the dog is a cat"
- "the dog is a dog"
- ...

# Language Model : The n-gram model

n-gram model:

**Hypothesis** → One *token* depends only from its  $n$  previous tokens.

**Model** →  $p_{\theta}(x_{t+1}|x_t, x_{t-1}, \dots, x_{t-n})$

→ a sequence of  $n$  tokens are called a n-gram

In practice :

- Consider a textual corpus
- Consider a vocabulary  $V$  (tokens units in the corpus)

→ Count all n-gram

The language model is the tensor  $|V|^n$

# Linear Interpolation

$$P(w_{t+1}|h_t) = \sum_{i=1}^k \lambda_i(h_t) P_i(w|h_t)$$

Diagram illustrating the Linear Interpolation formula for the next word probability:

- $P(w_{t+1}|h_t)$ : Next Word
- $h_t$ : History
- $k$ : Nb models
- $\lambda_i(h_t)$ : Model linear factor
- $P_i(w|h_t)$ : Model weights

# Focus on relevant information: The attention principle

N	sentence
1	Mary moved to the bathroom.
2	Sandra journeyed to the bedroom.
3	John went to the kitchen.
4	Mary took the football there.
5	Mary went to the kitchen.
6	Mary took a plate.

Retrieve *“How many objects is Mary carrying?”*

- Retrieve action from Mary
- Retrieve object taken

**Table 1:** Extract of the Facebook Baby Dataset

- Select the important content during prediction
- Storing input sequence to later access

## The attention principle

- Focus on part important for the prediction/generation when needed only
- Make in relation different part of a sequence

## Using a Memory:

The input can be seen as a memory :

- At different step a memory cell is selected
- The select cell is used for prediction and next memory selection

## The attention principle

- Focus on part important for the prediction/generation when needed only
- Make in relation different part of a sequence

## Using a Memory:

The input can be seen as a memory :

- At different step a memory cell is selected
- The select cell is used for prediction and next memory selection

→ Does not need to store all sequence information into a “Memory Vector”

## The attention principle

- Focus on part important for the prediction/generation when needed only
- Make in relation different part of a sequence

## Using a Memory:

The input can be seen as a memory :

- At different step a memory cell is selected
- The select cell is used for prediction and next memory selection

→ Does not need to store all sequence information into a “Memory Vector”

During the years 2015 to 2017 different model have been proposed under the name “Memory Network”:

→ “Memory Network”, Weston et al. 2015

→ “End-to-end Memory Network”, Sukhbaatar et al. 2015

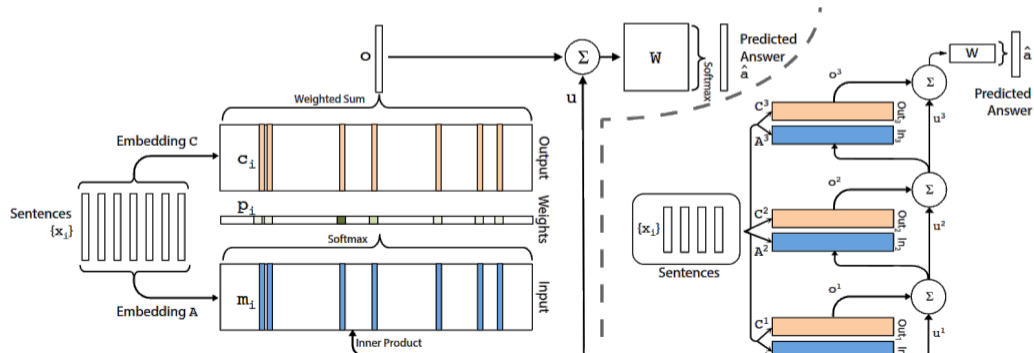
→ “Ask Me Anything: Dynamic Memory Network for Natural Language Processing”, Sukhbaatar et al. 2016

# End-to-end Memory Network

## End to End memory network

A MLP with different repeated blocks :

- **Input:** Query or output of the previous block
- An attention mechanism selecting part of the memory (sentences)
- **Output:** A vector from the selection mixed with the previous output

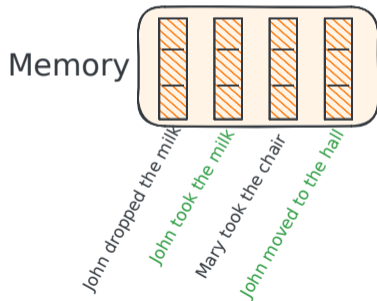


We consider a representation of the question as a vector  $h_0 \in \mathbb{R}^d$  we call it state vector, a memory containing context sentences representation as  $M \in \mathbb{R}^{d \times S}$ . Consider the fonction  $\mathcal{A}_{\theta_i}$  (Attention) such that :

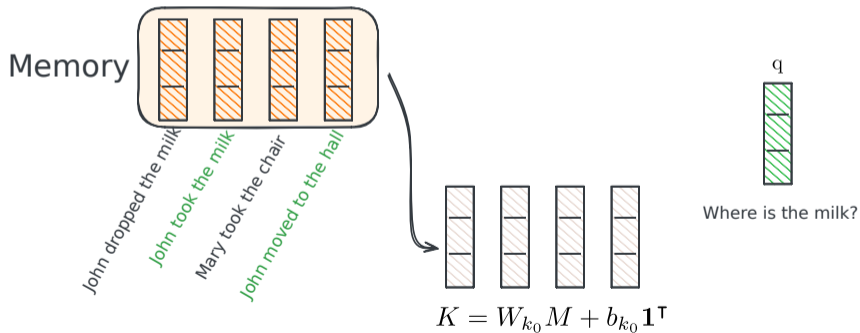
$$\begin{aligned}\mathcal{A}_{\theta_i}: \mathbb{R}^{d \times S} \times \mathbb{R}^d &\rightarrow \mathbb{R}^d \\ (M, h_i) &\mapsto a(M, h_i)\end{aligned}$$

The state  $h_{i+1}$  is obtained by  $W_{h_i} h_i + \mathcal{A}_{\theta_i}(M, h_i)$

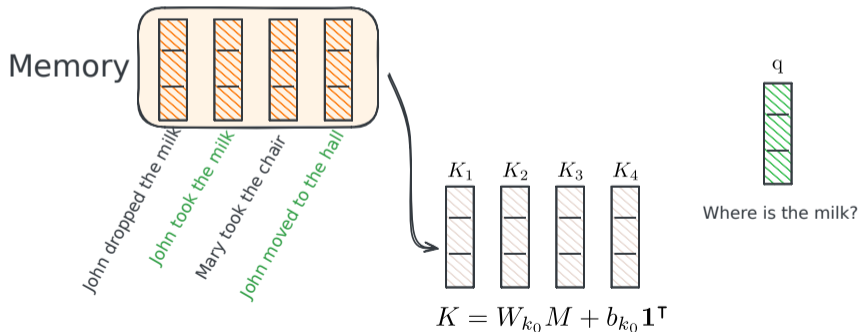
# End-to-end Memory Network: Attention mechanism



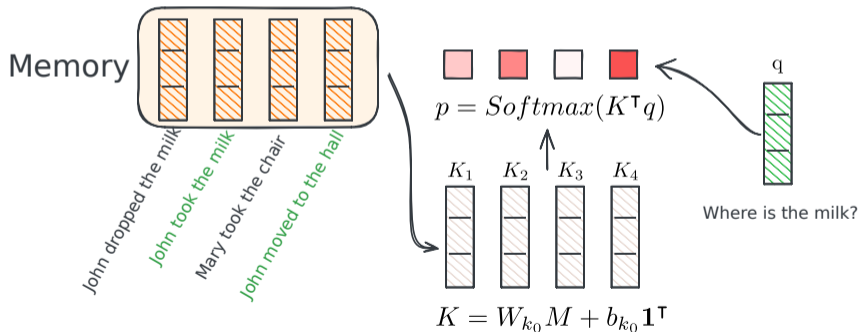
# End-to-end Memory Network: Attention mechanism



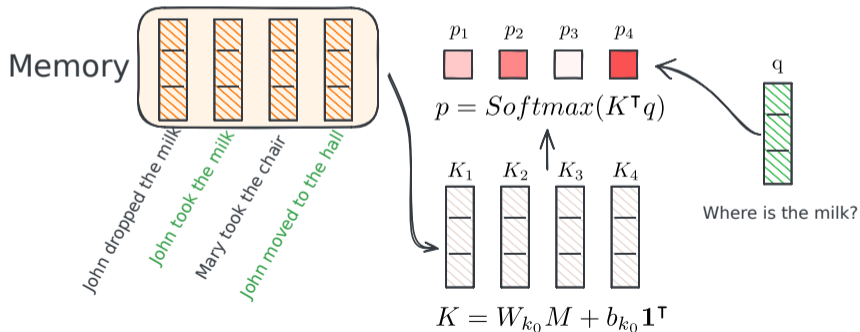
# End-to-end Memory Network: Attention mechanism



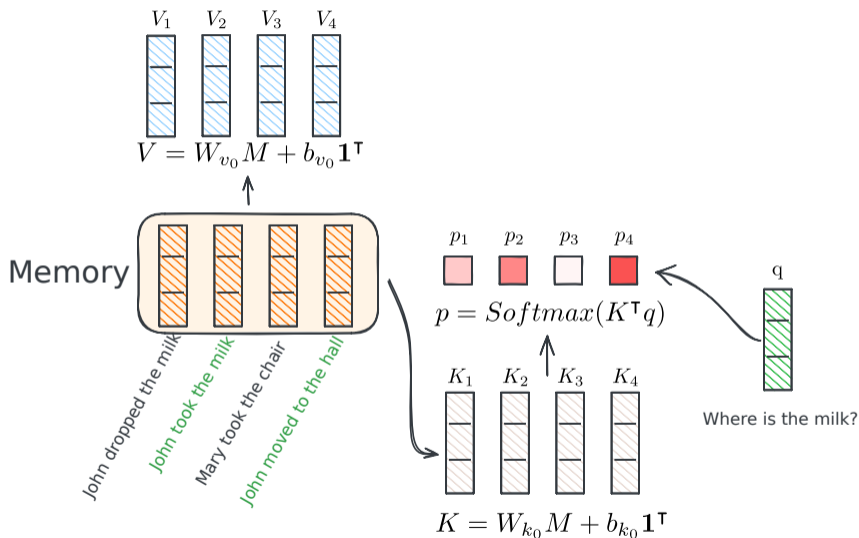
# End-to-end Memory Network: Attention mechanism



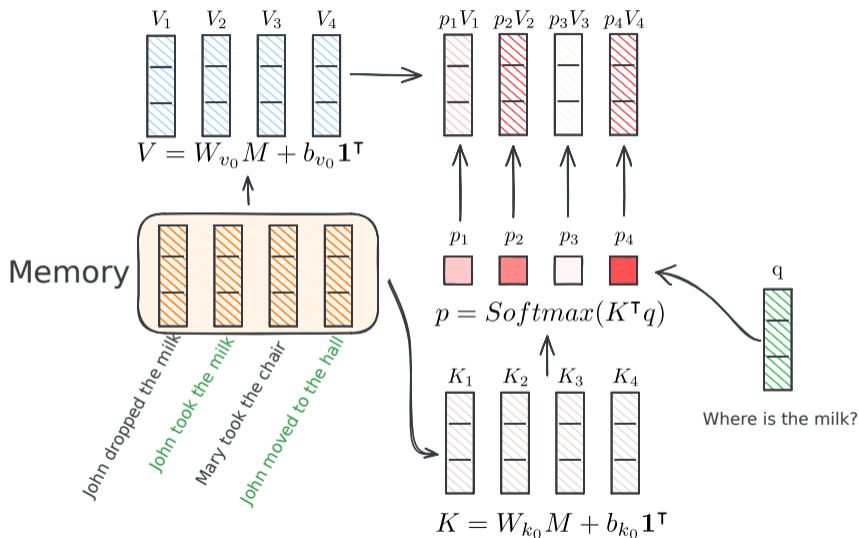
# End-to-end Memory Network: Attention mechanism



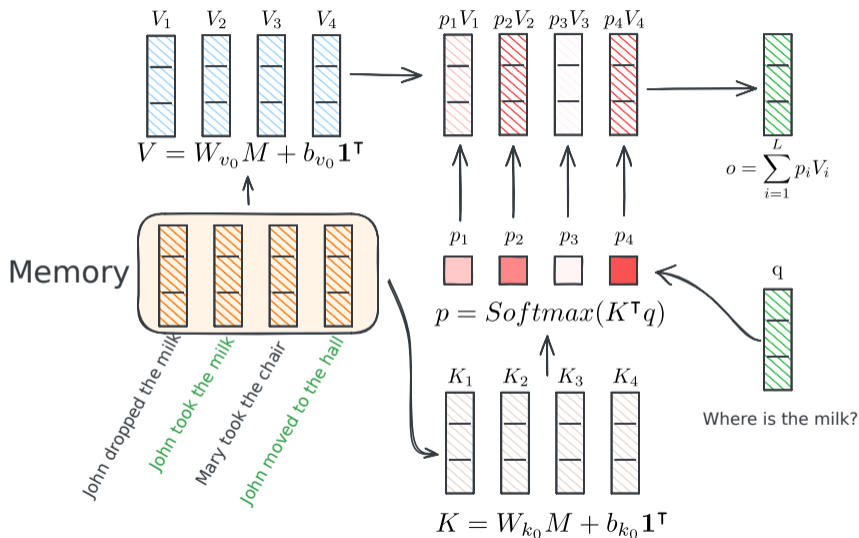
# End-to-end Memory Network: Attention mechanism



# End-to-end Memory Network: Attention mechanism



# End-to-end Memory Network: Attention mechanism



## Memory Network: Attention

Let consider the memory  $M \in \mathbb{R}^{d_{mem} \times L}$

→ each column containing a sentence representation

Let consider  $h_i \in \mathbb{R}^d$  the state vector

→  $h_0$  correspond to the question

## Memory Network: Attention

Let consider the memory  $M \in \mathbb{R}^{d_{mem} \times L}$

→ each column containing a sentence representation

Let consider  $h_i \in \mathbb{R}^d$  the state vector

→  $h_0$  correspond to the question

$\mathcal{A}_i$  is the attention function (see previous slides) with  $K = W_{k_i}M + b_{k_i}\mathbf{1}^\top$  and  $V = W_{v_i}M + b_{v_i}\mathbf{1}^\top$  is defined as:

$$\mathcal{A}_i(M, h_i) = V.\text{Softmax}(K^\top.q)$$

## Memory Network: Attention

Let consider the memory  $M \in \mathbb{R}^{d_{mem} \times L}$

→ each column containing a sentence representation

Let consider  $h_i \in \mathbb{R}^d$  the state vector

→  $h_0$  correspond to the question

$\mathcal{A}_i$  is the attention function (see previous slides) with  $K = W_{k_i}M + b_{k_i}\mathbf{1}^\top$  and  $V = W_{v_i}M + b_{v_i}\mathbf{1}^\top$  is defined as:

$$\mathcal{A}_i(M, h_i) = V.\text{Softmax}(K^\top.q)$$

---

## Algorithm 3 Attention $\mathcal{A}_i$

---

Require:  $h_i \in \mathbb{R}^d, M \in \mathbb{R}^{d_{mem} \times L}$

$K \leftarrow W_{k_i}M + b_{k_i}\mathbf{1}^\top$

$V \leftarrow W_{v_i}M + b_{v_i}\mathbf{1}^\top$

$p \leftarrow \text{Softmax}(K^\top.q)$

return  $V.p$

---

Reminder:

$$\text{Softmax}(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{\dim(x)} e^{x_j}}$$

# End-to-end Memory Network: Output prediction

## Output prediction:

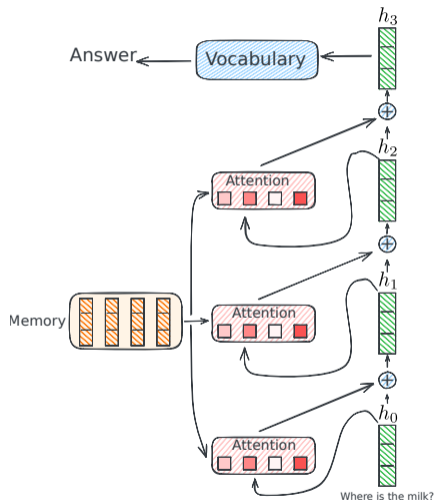
The model only predict a word of the vocabulary  $V$  as output

→ Considering  $h_N$  last state ( $N$  is chosen)

→ Considering  $W_o \in \mathbb{R}^{|V| \times d}$

The prediction word (index of the word) is given by :

$$\operatorname{argmax}_i W_o h_n$$



## Sentence embeddings

The memory matrix  $M$  store representation of sentences :

→ use the average of words sentences embedding (Word2Vec)

$$M_i = \frac{1}{|S|} \sum_{j=0}^{|S|} w_{sj}$$

## Sentence embeddings

The memory matrix  $M$  store representation of sentences :

→ use the average of words sentences embedding (Word2Vec)

$$M_i = \frac{1}{|S|} \sum_{j=0}^{|S|} w_{sj}$$

## Take into account position:

Sentences follow a sequence (e.g john took the milk before moving to the hall)

→ The model is not sequential...

## Sentence embeddings

The memory matrix  $M$  store representation of sentences :

→ use the average of words sentences embedding (Word2Vec)

$$M_i = \frac{1}{|S|} \sum_{j=0}^{|S|} w_{sj}$$

## Take into account position:

Sentences follow a sequence (e.g john took the milk before moving to the hall)

→ The model is not sequential...encode it into the sentence representation

$$M_i = \frac{1}{|S|} \sum_{j=0}^{|S|} w_{sj} + pos_j$$

# Memory network: Conclusion

<b>Story (2: 2 supporting facts)</b>	<b>Support</b>	<b>Hop 1</b>	<b>Hop 2</b>	<b>Hop 3</b>
John dropped the milk.		0.06	0.00	0.00
John took the milk there.	yes	0.88	1.00	0.00
Sandra went back to the bathroom.		0.00	0.00	0.00
John moved to the hallway.	yes	0.00	0.00	1.00
Mary went back to the bedroom.		0.00	0.00	0.00
<b>Where is the milk? Answer: hallway Prediction: hallway</b>				

<b>Story (18: size reasoning)</b>	<b>Support</b>	<b>Hop 1</b>	<b>Hop 2</b>	<b>Hop 3</b>
The suitcase is bigger than the chest.	yes	0.00	0.88	0.00
The box is bigger than the chocolate.		0.04	0.05	0.10
The chest is bigger than the chocolate.	yes	0.17	0.07	0.90
The chest fits inside the container.		0.00	0.00	0.00
The chest fits inside the box.		0.00	0.00	0.00
<b>Does the suitcase fit in the chocolate? Answer: no Prediction: no</b>				

Figure 1: Depicting Attention Weights for the End-to-End Memory Network

# Memory network: Conclusion

<b>Story (2: 2 supporting facts)</b>	<b>Support</b>	<b>Hop 1</b>	<b>Hop 2</b>	<b>Hop 3</b>
John dropped the milk.		0.06	0.00	0.00
John took the milk there.	yes	0.88	1.00	0.00
Sandra went back to the bathroom.		0.00	0.00	0.00
John moved to the hallway.	yes	0.00	0.00	1.00
Mary went back to the bedroom.		0.00	0.00	0.00
<b>Where is the milk? Answer: hallway Prediction: hallway</b>				

<b>Story (18: size reasoning)</b>	<b>Support</b>	<b>Hop 1</b>	<b>Hop 2</b>	<b>Hop 3</b>
The suitcase is bigger than the chest.	yes	0.00	0.88	0.00
The box is bigger than the chocolate.		0.04	0.05	0.10
The chest is bigger than the chocolate.	yes	0.17	0.07	0.90
The chest fits inside the container.		0.00	0.00	0.00
The chest fits inside the box.		0.00	0.00	0.00
<b>Does the suitcase fit in the chocolate? Answer: no Prediction: no</b>				

Figure 1: Depicting Attention Weights for the End-to-End Memory Network

## Memory Network

- Using a mechanism of attention to select into “stored content”

# Memory network: Conclusion

<b>Story (2: 2 supporting facts)</b>	<b>Support</b>	<b>Hop 1</b>	<b>Hop 2</b>	<b>Hop 3</b>
John dropped the milk.		0.06	0.00	0.00
John took the milk there.	yes	0.88	1.00	0.00
Sandra went back to the bathroom.		0.00	0.00	0.00
John moved to the hallway.	yes	0.00	0.00	1.00
Mary went back to the bedroom.		0.00	0.00	0.00
<b>Where is the milk? Answer: hallway Prediction: hallway</b>				

<b>Story (18: size reasoning)</b>	<b>Support</b>	<b>Hop 1</b>	<b>Hop 2</b>	<b>Hop 3</b>
The suitcase is bigger than the chest.	yes	0.00	0.88	0.00
The box is bigger than the chocolate.		0.04	0.05	0.10
The chest is bigger than the chocolate.	yes	0.17	0.07	0.90
The chest fits inside the container.		0.00	0.00	0.00
The chest fits inside the box.		0.00	0.00	0.00
<b>Does the suitcase fit in the chocolate? Answer: no Prediction: no</b>				

Figure 1: Depicting Attention Weights for the End-to-End Memory Network

## Memory Network

- Using a mechanism of attention to select into “stored content”

# Memory network: Conclusion

<b>Story (2: 2 supporting facts)</b>	<b>Support</b>	<b>Hop 1</b>	<b>Hop 2</b>	<b>Hop 3</b>
John dropped the milk.		0.06	0.00	0.00
John took the milk there.	yes	0.88	1.00	0.00
Sandra went back to the bathroom.		0.00	0.00	0.00
John moved to the hallway.	yes	0.00	0.00	1.00
Mary went back to the bedroom.		0.00	0.00	0.00
<b>Where is the milk? Answer: hallway Prediction: hallway</b>				

<b>Story (18: size reasoning)</b>	<b>Support</b>	<b>Hop 1</b>	<b>Hop 2</b>	<b>Hop 3</b>
The suitcase is bigger than the chest.	yes	0.00	0.88	0.00
The box is bigger than the chocolate.		0.04	0.05	0.10
The chest is bigger than the chocolate.	yes	0.17	0.07	0.90
The chest fits inside the container.		0.00	0.00	0.00
The chest fits inside the box.		0.00	0.00	0.00
<b>Does the suitcase fit in the chocolate? Answer: no Prediction: no</b>				

Figure 1: Depicting Attention Weights for the End-to-End Memory Network

## Memory Network

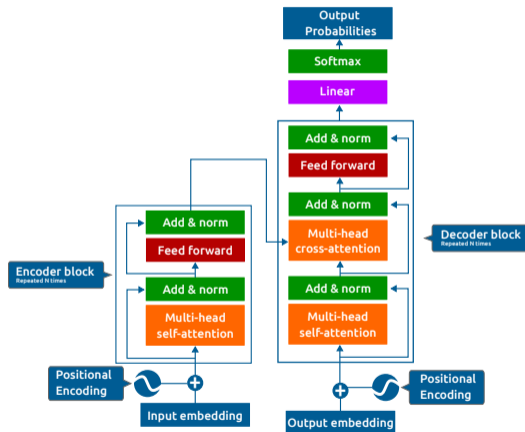
- Using a mechanism of attention to select into “stored content”

## The transformer (a reminder)

---

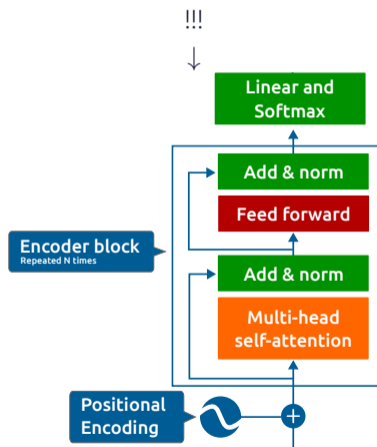
## Today's Language Models

- 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)



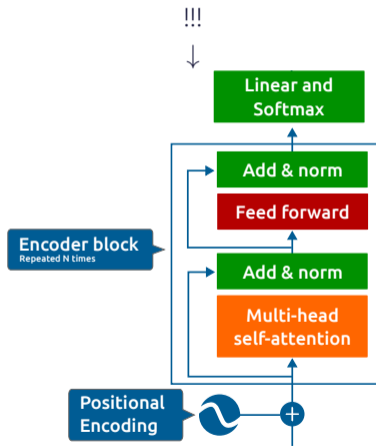
# The transformer: Encoder or decoder

This is the **Encoder** transformer architecture

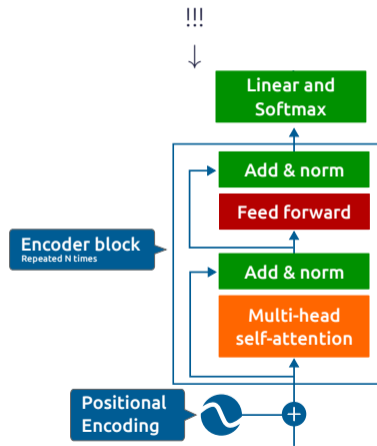


# The transformer: Encoder or decoder

This is the **Encoder** transformer architecture

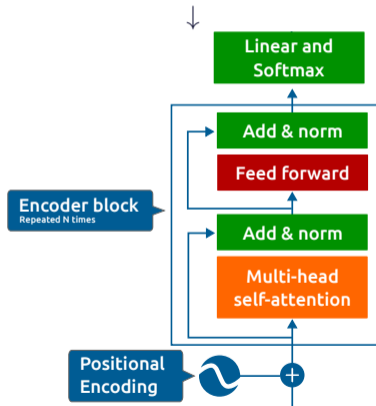


This is the **Decoder** transformer architecture



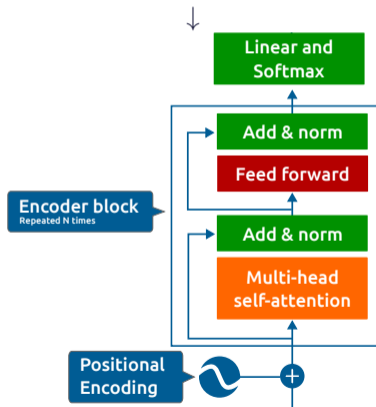
# The transformer: Encoder or decoder

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



# The transformer: Encoder or decoder

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



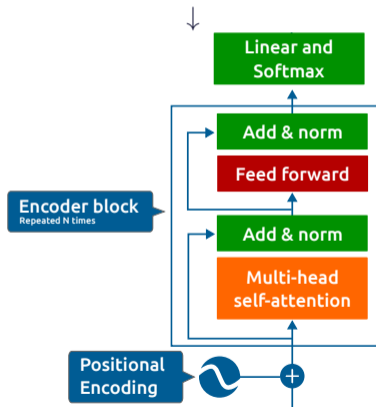
Encoder or Decoder ?

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

There is a difference, but ...

# The transformer: Encoder or decoder

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



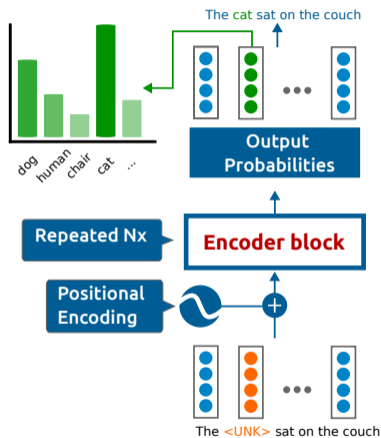
Encoder or Decoder ?

- 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

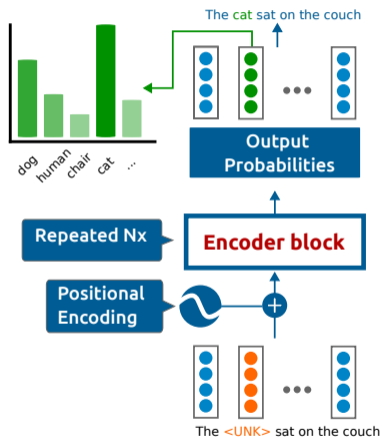
# The transformer: The Encoder model



## Pretrained encoder

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

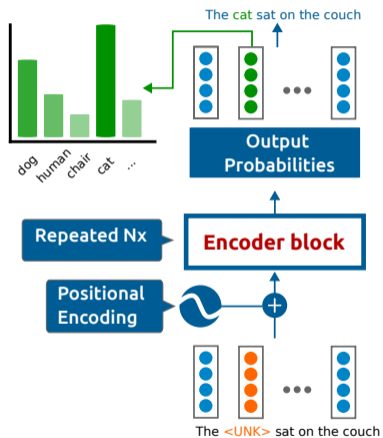
# The transformer: The Encoder model



## Pretrained encoder

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

# The transformer: The Encoder model



## Pretrained encoder

- Bidirectional (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

## 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  $X = (x_1, x_2, \dots, x_n)$ , let say we consider  $x_i$  the masked token. We want to maximize :

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

# 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  $X = (x_1, x_2, \dots, x_n)$ , let say we consider  $x_i$  the masked token. We want to maximize :

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



# The Transformer Model: Masked Attention

Let consider  $X \in \mathbb{R}^{d_{in} \times l}$  the output of a decoder block:

$$\begin{array}{llll} \text{Query } Q & \text{Key } K & \text{Value } V & \text{Mask } M \\ Q = W_q X + b_q \mathbf{1}^t & K = W_k X + b_k \mathbf{1}^t & V = W_v X + b_v \mathbf{1}^t & M_{i,j} = \begin{cases} i \geq j & 0 \\ i < j & -\infty \end{cases} \end{array}$$

The attention weight (modeled probabilities) is written

$$P = \text{Softmax}\left(\frac{K^T Q}{\sqrt{d_{attn}}} + M\right)$$

The output of the attention is given by

$$V' = V.P$$

---

## Algorithm 4 Self-Attention

---

Require:  $O_n, W_q, W_k, W_v \in \mathbb{R}^{d_{attn}, d_{in}}$

$$Q \leftarrow W_q X + b_q \mathbf{1}^t$$

$$K \leftarrow W_k X + b_k \mathbf{1}^t$$

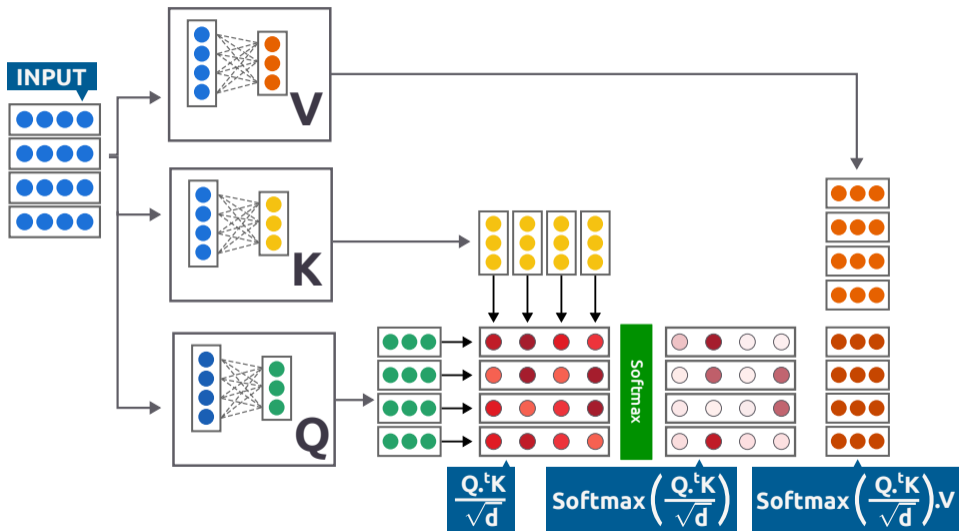
$$V \leftarrow W_v X + b_v \mathbf{1}^t$$

$$P \leftarrow \text{Softmax}\left(\frac{K^T Q}{\sqrt{d_{attn}}} + M\right)$$

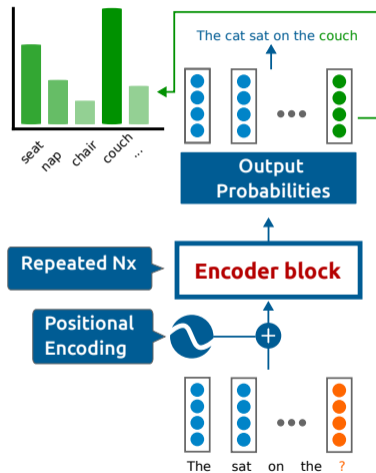
return  $V.P$

---

# The transformer: Bidirectional attention



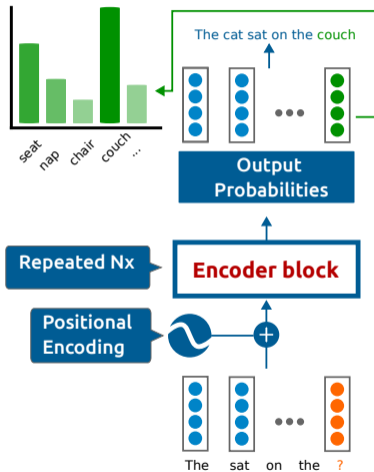
# The transformer: The Decoder model



## Pretrained Decoder

- Unidirectional (current token only depends from previous ones)

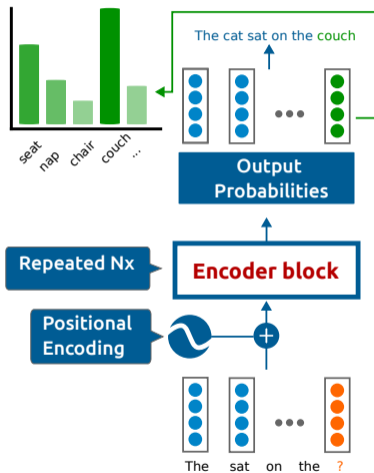
# The transformer: The Decoder model



## Pretrained Decoder

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

# The transformer: The Decoder model



## Pretrained Decoder

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

# The transformer: Next token prediction

## Masked Language Modeling

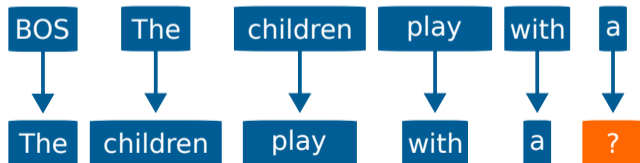
- Try to predict the next token

The model has "access" only to previous tokens

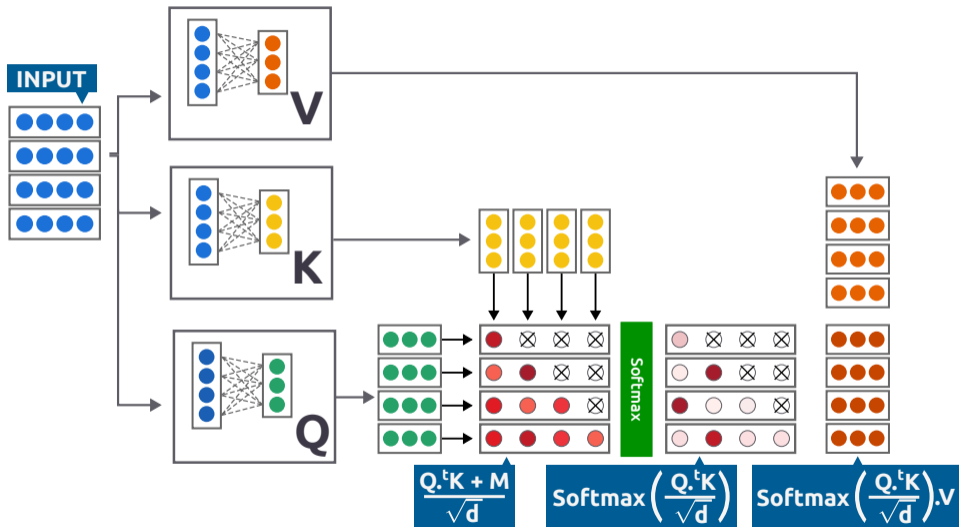
## Objective

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

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



# The transformer: Unidirectional attention



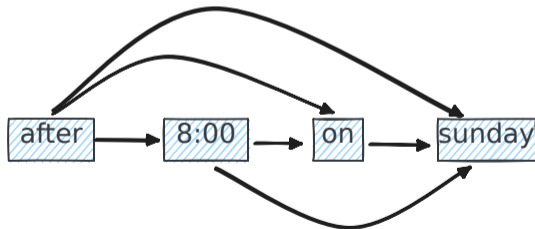
# The Transformer model : Decoder Self-Attention

For the decoder block, we consider modelling  $P(y|z)$  (y output text and z encoder representation) modeled as

$$P_{\theta}(y[1]|z) \times P_{\theta}(y[2]|y[1],z) \times \dots \times P_{\theta}(y[l]|y[1:l-1],z)$$

→ Prediction of the next tokens considering previous generated ones.

→ self attention for a token(representation) should only look at previous tokens



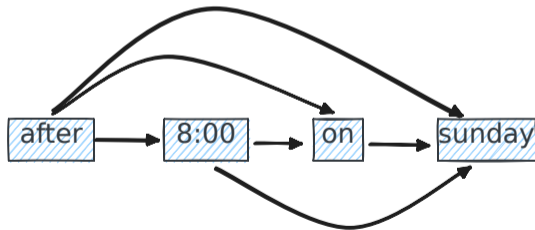
# The Transformer model : Decoder Self-Attention

For the decoder block, we consider modelling  $P(y|z)$  (y output text and z encoder representation) modeled as

$$P_{\theta}(y[1]|z) \times P_{\theta}(y[2]|y[1],z) \times \dots \times P_{\theta}(y[l]|y[1:l-1],z)$$

→ Prediction of the next tokens considering previous generated ones.

→ self attention for a token(representation) should only look at previous tokens



Ensure attention weights are set to zero for next tokens:

$$P = \begin{pmatrix} P_{1,1} & 0 & \dots & \dots & 0 \\ P_{2,1} & P_{2,2} & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots \\ P_{l,1} & P_{l,2} & \dots & \dots & P_{l,l} \end{pmatrix}$$

# The transformer: Encoder or decoder

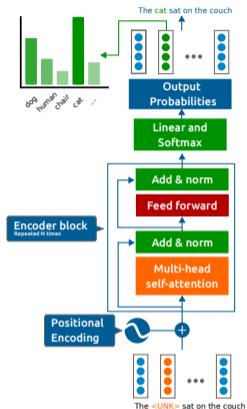


Figure 2: Encoder model and (one of) pretraining task

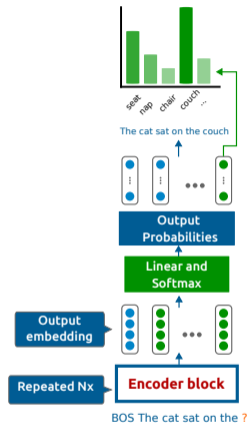
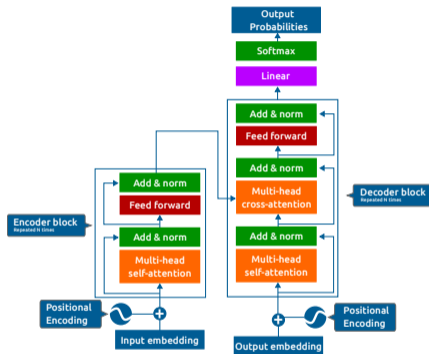


Figure 3: Decoder model and (one of) pretraining task

# The transformer: Encoder-decoder

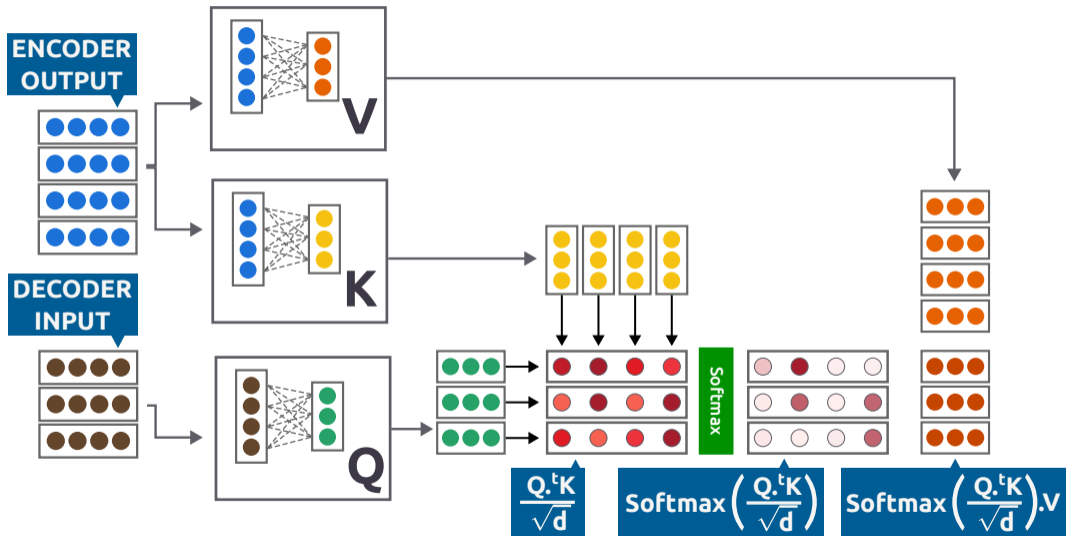


## Encoder-decoder

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

Figure 4: Encoder-decoder models

# The transformer: cross-attention



## The attention

We would capture/resolve different information:

- Anaphora
- Syntactic relationship
- Semantic relationship
- Key-words
- ...

# The Transformer Model: Multihead Attention

## The attention

We would capture/resolve different information:

- Anaphora
- Syntactic relationship
- Semantic relationship
- Key-words
- ...

→ Use different attention (Multiple head)

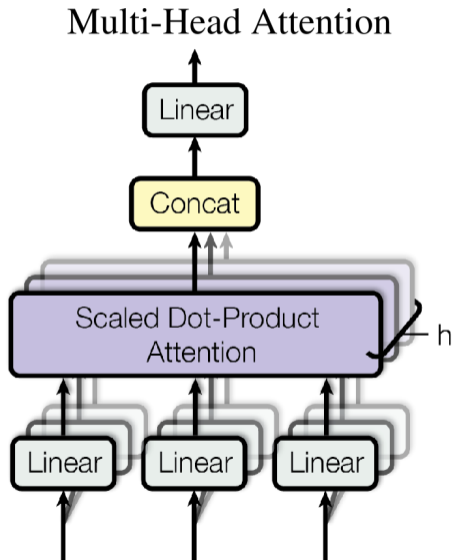
# The Transformer Model: Multihead Attention

## The attention

We would capture/resolve different information:

- Anaphora
- Syntactic relationship
- Semantic relationship
- Key-words
- ...

→ Use different attention (Multiple head)



# The Transformer Model: Multihead

## The attention

We would capture/perform different information/task:

- Anaphore
- Coreference
- Part-of-speech
- Action
- ...

→ Use different (H attentions) attention (Multiple head)

---

## Algorithm 5 Multihead self-Attention

---

Require:  $X, W_q^h, W_k^h, W_v^h \in \mathbb{R}^{d_{attn} \times d_{in}}$   $W_h \in \mathbb{R}^{d_{out} \times H d_{attn}}$   
for do  $h = 1; h \leq H; h++$   
     $Q \leftarrow W_q X + b_q \mathbf{1}^T$   
     $K \leftarrow W_k X + b_k \mathbf{1}^T$   
     $V \leftarrow W_v X + b_v \mathbf{1}^T$   
     $Y^h \leftarrow V \cdot \text{Softmax}\left(\frac{K^T Q}{\sqrt{d_{attn}}}\right)$   
end for  
 $Y \leftarrow [Y^1; Y^2; \dots; Y^H]$   
Return  $W_h Y + b_h \mathbf{1}^T$

---

## Stabilise training

---

Stabilising training

## Stabilising training

- Gradient does not backpropagate well in the softmax function

## Stabilising training

- Gradient does not backpropagate well in the softmax function
- Stacking modules also leads to vanishing information/gradient

## Stabilising training

- Gradient does not backpropagate well in the softmax function
- Stacking modules also leads to vanishing information/gradient

→ Adding the previous information (input) to each layer output

## Stabilising training

- Gradient does not backpropagate well in the softmax function
- Stacking modules also leads to vanishing information/gradient

—→ Adding the previous information (input) to each layer output

This approach has been popularised in the residual network (in image recognition)

→ “Deep Residual Learning for Image Recognition”, 2015, He et al.

# The Transformer Model: Add and Normalisation

## Skip connection

Let be  $f_\theta$  an attention or a feed-forward module, let  $x$  the input of the module,  $g_\theta$  being a transformer block:

$$g_\theta: \mathbb{R}^{N \times D} \rightarrow \mathbb{R}^{N \times D}$$
$$x \mapsto f_\theta + x.$$

Adding  $x$  to the output of  $f$  is called a “skip connection”

## Layer Normalisation

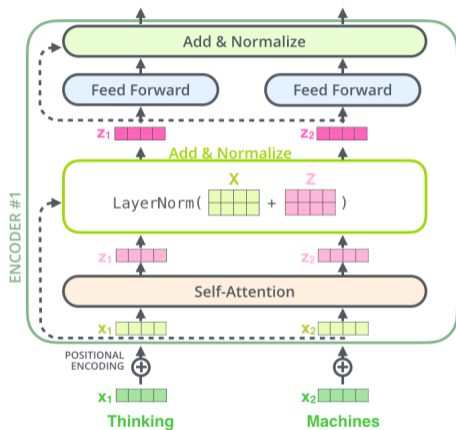
Each is also normalised using the layer normalisation (normalise along embeddings dimension):

- Recentering the data
- Rescaling the data

**Objective:** Can help to reduce the impact of vanishing or exploding gradients, by keeping the activations within a reasonable range.

$$LN: \mathbb{R}^{d \times l} \rightarrow \mathbb{R}^{d \times l}$$
$$x \mapsto \frac{x - \mathbb{E}(x)}{\sqrt{\text{Var}(x)}} \gamma + \beta.$$

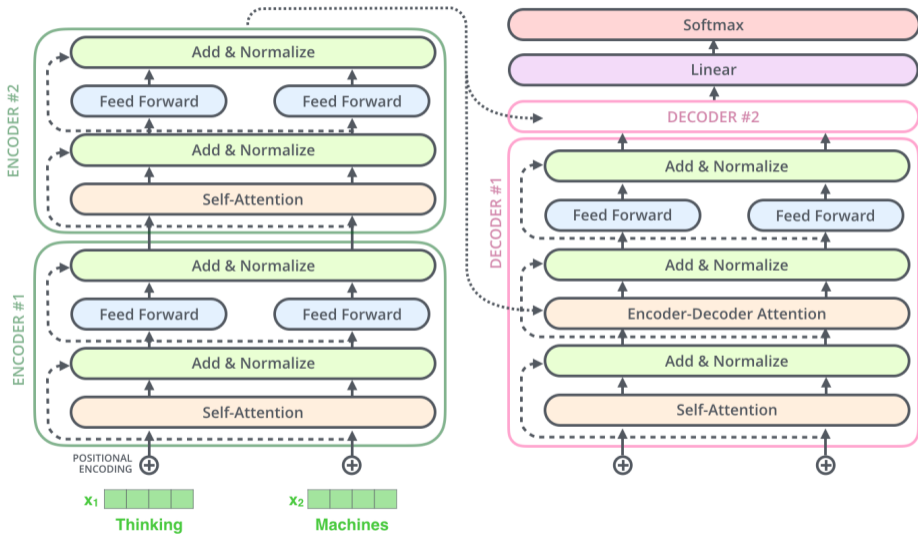
# Transformer block: Summary



## Transformer Block

- Input  $x$
- Self-Attention
- Addition of  $x$  with attention output
- Normalisation (produce  $z$ )
- Feed-Forward
- Addition of  $x$  with feed-forward output

# Trasformer block: Summary



# Positional Embedding

---

# The Transformer Model: Encoding position

→ No position information in the architecture (contrary to RNN).

Encoding Position

# The Transformer Model: Encoding position

→ No position information in the architecture (contrary to RNN).

## Encoding Position

→ Choose a place to encode the position

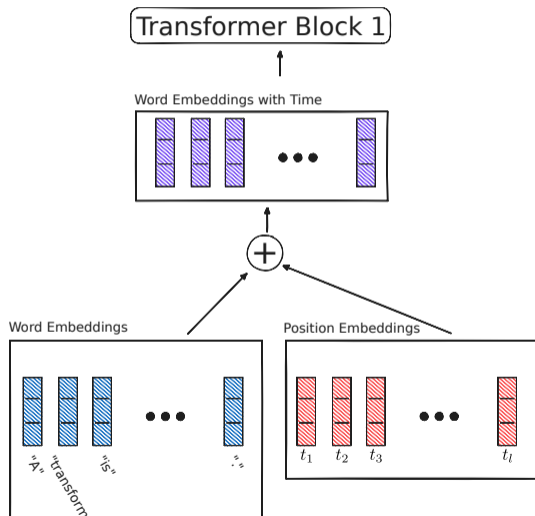
# The Transformer Model: Encoding position

→ No position information in the architecture (contrary to RNN).

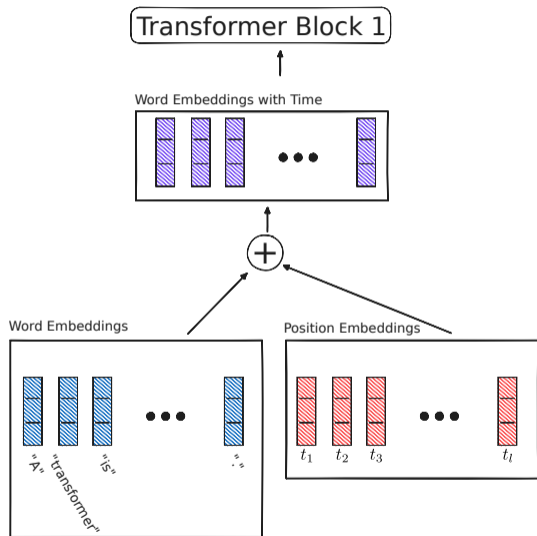
## Encoding Position

→ Choose a place to encode the position

- In token embeddings (first layer)?
- In attention?
- Using trained embeddings?



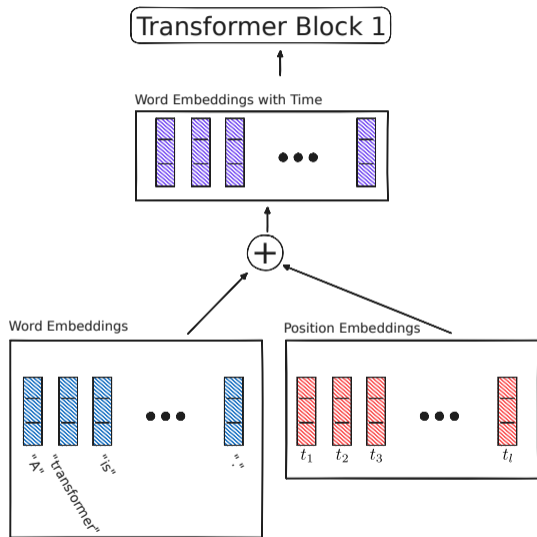
# The Transformer Model: Absolute Position



## Absolute position with embeddings

Training a matrix of position for each position  
 $W_p \in \mathbb{R}^{d_{in} \times L}$  (L maximum size)

# The Transformer Model: Absolute Position



## Absolute position with embeddings

Training a matrix of position for each position

$W_p \in \mathbb{R}^{d_{in} \times L}$  (L maximum size)

→ limited position restrained to what have been observe in training

# Positional embeddings

## Encode position by function design

The vanilla transformer use the following function:

$$pos_{t,j} = \begin{cases} j \text{ pair} & \sin(\frac{t}{c^{2j/d_{in}}}) \\ j \text{ not pair} & \cos(\frac{t}{c^{(2j-1)/d_{in}}}) \end{cases}$$

The positional vector is given by:

$$pos_{\vec{s}_t} = \begin{bmatrix} \sin(\frac{t}{1}) \\ \cos(\frac{t}{1}) \\ \sin(\frac{t}{c^{2/d_{in}}}) \\ \cos(\frac{t}{c^{2/d_{in}}}) \\ \vdots \\ \sin(\frac{t}{c^{2/d_{in}}}) \\ \cos(\frac{t}{c^{2/d_{in}}}) \end{bmatrix}$$

Main intuition: closely related to binary representation

0:	0 0 0 0	8:	1 0 0 0
1:	0 0 0 1	9:	1 0 0 1
2:	0 0 1 0	10:	1 0 1 0
3:	0 0 1 1	11:	1 0 1 1
4:	0 1 0 0	12:	1 1 0 0
5:	0 1 0 1	13:	1 1 0 1
6:	0 1 1 0	14:	1 1 1 0
7:	0 1 1 1	15:	1 1 1 1

---

Illustration from [https://kazemnejad.com/blog/transformer\\_architecture\\_positional\\_encoding/](https://kazemnejad.com/blog/transformer_architecture_positional_encoding/)

# Positional embeddings

## Encode position by function design

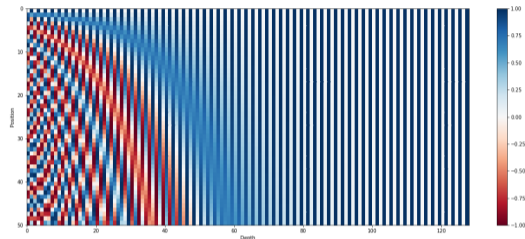
The vanilla transformer use the following function:

$$pos_{t,j} = \begin{cases} j \text{ pair} & \sin(\frac{t}{c^{2j/d_{in}}}) \\ j \text{ not pair} & \cos(\frac{t}{c^{(2j-1)/d_{in}}}) \end{cases}$$

The positional vector is given by:

$$pos_{\vec{s}_t} = \begin{bmatrix} \sin(\frac{t}{c^1}) \\ \cos(\frac{t}{c^1}) \\ \sin(\frac{t}{c^{2/d_{in}}}) \\ \cos(\frac{t}{c^{2/d_{in}}}) \\ \vdots \\ \sin(\frac{t}{c^{2/d_{in}}}) \\ \cos(\frac{t}{c^{2/d_{in}}}) \end{bmatrix}$$

Main intuition: Closely related to binary representation



# Positional embeddings

## Encode position by function design

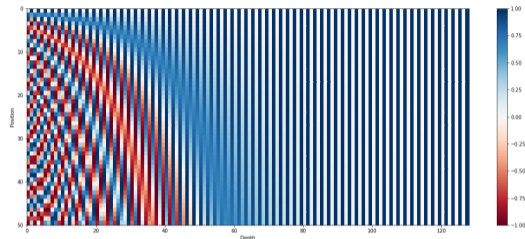
The vanilla transformer use the following function:

$$pos_{t,j} = \begin{cases} j \text{ pair} & \sin(\frac{t}{c^{2j/d_{in}}}) \\ j \text{ not pair} & \cos(\frac{t}{c^{(2j-1)/d_{in}}}) \end{cases}$$

The positional vector is given by:

$$pos_{\vec{t}} = \begin{bmatrix} \sin(\frac{t}{1}) \\ \cos(\frac{t}{1}) \\ \sin(\frac{t}{c^{2/d_{in}}}) \\ \cos(\frac{t}{c^{2/d_{in}}}) \\ \vdots \\ \sin(\frac{t}{c^{2/d_{in}}}) \\ \cos(\frac{t}{c^{2/d_{in}}}) \end{bmatrix}$$

Main intuition: Closely related to binary representation



## Original justification

“We chose this function because we hypothesized it would allow the model to easily learn to attend by relative positions, since for any fixed offset  $k$ ,  $pos_{t+k}$  can be represented as a linear function of  $pos_t$ .”

Illustration from [https://kazemnejad.com/blog/transformer\\_architecture\\_positional\\_encoding/](https://kazemnejad.com/blog/transformer_architecture_positional_encoding/)

# The Transformer Model: Relative Position

**Hypothesis:** Dependency of a token is in its close neighborhood

→ It is more important to encode relative position to a token

- For each token encode its distance to other tokens

# The Transformer Model: Relative Position

**Hypothesis:** Dependency of a token is in its close neighborhood

→ It is more important to encode relative position to a token

- For each token encode its distance to other tokens
- Encode it into attention mechanism

# The Transformer Model: Relative Position

**Hypothesis:** Dependency of a token is in its close neighborhood

→ It is more important to encode relative position to a token

- For each token encode its distance to other tokens
- Encode it into attention mechanism

## Relative position in Transformers

# The Transformer Model: Relative Position

**Hypothesis:** Dependency of a token is in its close neighborhood

→ It is more important to encode relative position to a token

- For each token encode its distance to other tokens
- Encode it into attention mechanism

## Relative position in Transformers

- T5 model (next lecture) uses trained relative positional embedding

# The Transformer Model: Relative Position

**Hypothesis:** Dependency of a token is in its close neighborhood

→ It is more important to encode relative position to a token

- For each token encode its distance to other tokens
- Encode it into attention mechanism

## Relative position in Transformers

- T5 model (next lecture) uses trained relative positional embedding
- Bloom uses relative positional embedding (*ALiBi*)

# The Transformer Model: Relative Position

**Hypothesis:** Dependency of a token is in its close neighborhood

→ It is more important to encode relative position to a token

- For each token encode its distance to other tokens
- Encode it into attention mechanism

## Relative position in Transformers

- T5 model (next lecture) uses trained relative positional embedding
- Bloom uses relative positional embedding (*ALiBi*)
- LLAMA 2 uses relative positional embedding (*RoPE*)

# The Transformer Model: ALiBi positional embedding

Explicitly encode relative position:

- Position for each token regarding its position to other tokens
- In attention mechanism

# The Transformer Model: ALiBi positional embedding

Explicitly encode relative position:

- Position for each token regarding its position to other tokens
- In attention mechanism

## ALiBi Positional embedding

- Use a relative position matrix in self attention
- Add this matrix to  $K^tQ$

# The Transformer Model: ALiBi positional embedding

Explicitly encode relative position:

- Position for each token regarding its position to other tokens
- In attention mechanism

## ALiBi Positional embedding

- Use a relative position matrix in self attention
- Add this matrix to  $K^tQ$

$$\begin{bmatrix} q_1 \cdot k_1 & & & & \\ q_2 \cdot k_1 & q_2 \cdot k_2 & & & \\ q_3 \cdot k_1 & q_3 \cdot k_2 & q_3 \cdot k_3 & & \\ q_4 \cdot k_1 & q_4 \cdot k_2 & q_4 \cdot k_3 & q_4 \cdot k_4 & \\ q_5 \cdot k_1 & q_5 \cdot k_2 & q_5 \cdot k_3 & q_5 \cdot k_4 & q_5 \cdot k_5 \end{bmatrix} + \begin{bmatrix} 0 & & & & \\ -1 & 0 & & & \\ -2 & -1 & 0 & & \\ -3 & -2 & -1 & 0 & \\ -4 & -3 & -2 & -1 & 0 \end{bmatrix} \cdot m$$

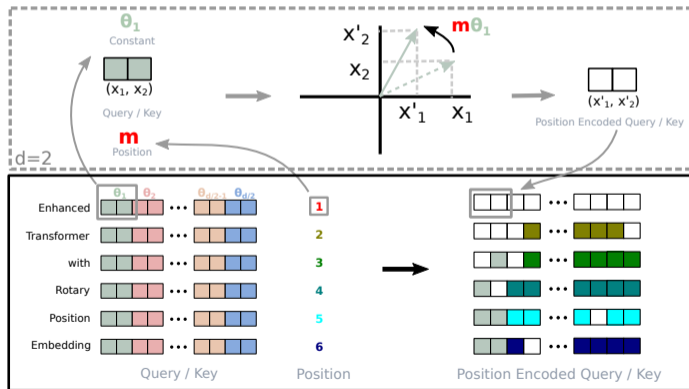
# The Transformer Model: RoPE

## Rotary Positional Embedding

A embedding based on relative positional embedding

→ Apply a rotation on Q and K

→ The rotation depends on the relative position



$$\begin{aligned}k_m^T q_n &= (\mathcal{R}_m W_k x_m)^T (\mathcal{R}_n W_k x_n) \\ &= x_m^T W_k \mathcal{R}_m^T \mathcal{R}_n W_k x_n \\ &= x_m^T W_k \mathcal{R}_{n-m} W_k x_n\end{aligned}$$

# Feed-Forward Network

---

# The Transformer Model: Feed-Forward

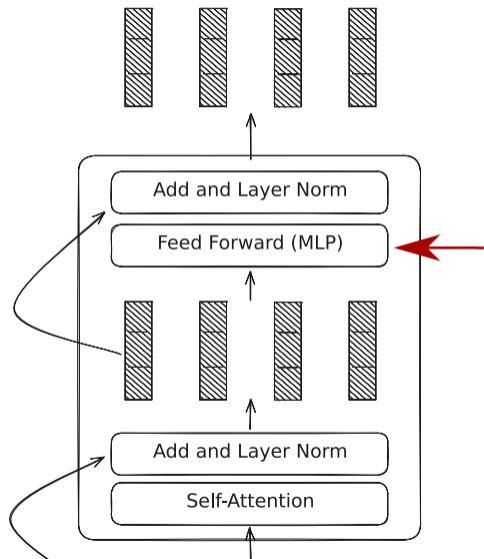
## Feed Forward

The feed forward is a neural network.

- Add non linearity to the network
- Prepare inputs for next attention

## Feed forward in vanilla transformer

$$ff(x) = W_2(\text{ReLU}(0, W_1x + b_1\mathbf{1}^\top)) + b_2\mathbf{1}^\top$$



## 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 ?

## 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 ?

## A pretrained model

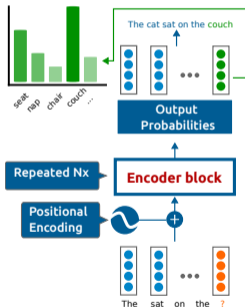
- Trained on a task (that does not need annotated data)
- Different architectures (encoder, decoder, encoder-decoder)

## 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

## Pre-trained



## 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, **grau-pel** 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".

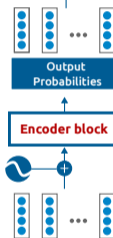
What causes precipitation to fall?  
**gravity**

What is another main form of precipitation besides drizzle, rain, snow, sleet and hail?  
**grau-pel**

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

## Adapted Model

... **gravity**



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, **grau-pel** 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/transformer !!! (BERT, Llama, etc...)
- A Dataset (sufficiently large) with annotated data !

## Before going further

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

## 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**

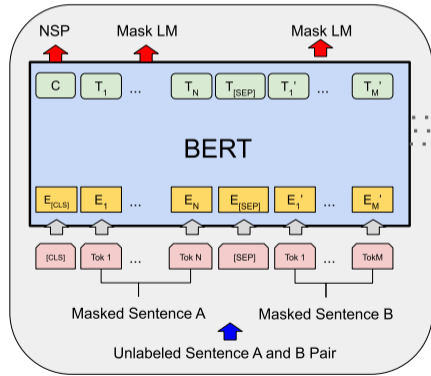
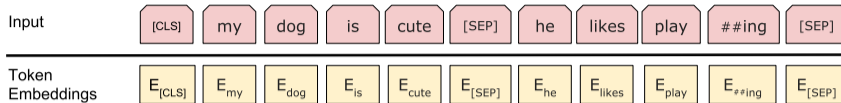
We introduce a new language representation model called **BERT**, which stands for **Bidirectional Encoder Representations from Transformers**. Unlike recent language representation models (Peters et al., 2018a; Radford et al., 2018), BERT is designed to pre-train deep bidirectional representations from 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.

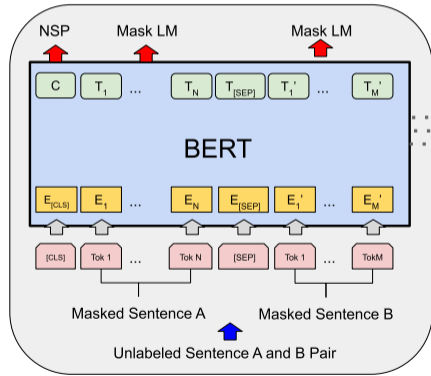
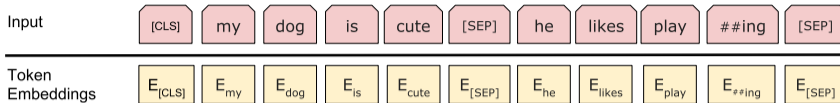
We argue that current techniques restrict the

# Adaptation: The BERT model



Pre-training

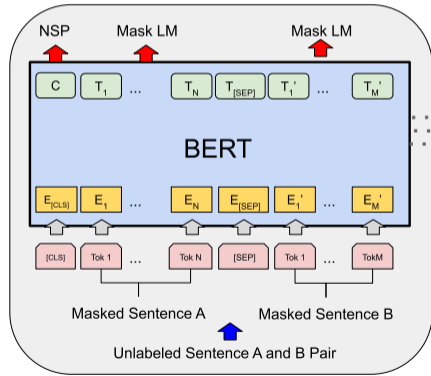
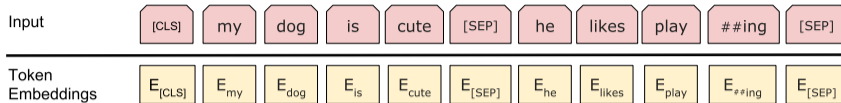
# Adaptation: The BERT model



Pre-Training

Pre-training

# Adaptation: The BERT model

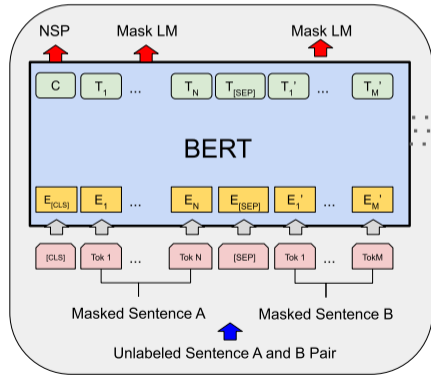
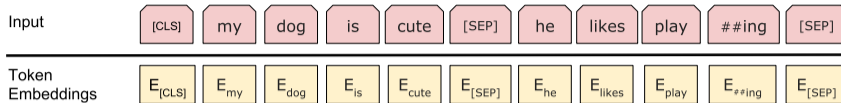


## Pre-Training

- A masked language modeling task (Mask LM)

Pre-training

# Adaptation: The BERT model

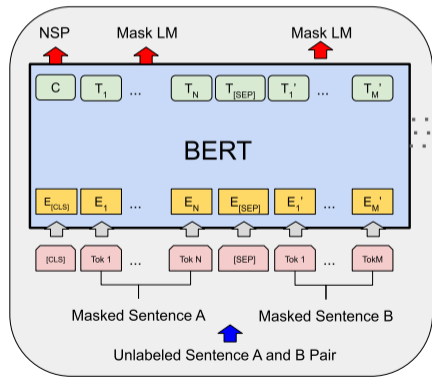
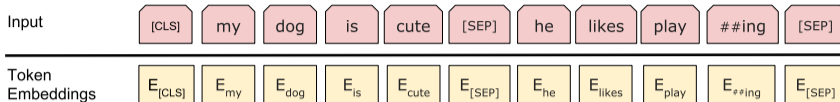


## Pre-Training

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

Pre-training

# Adaptation: The BERT model



Pre-training

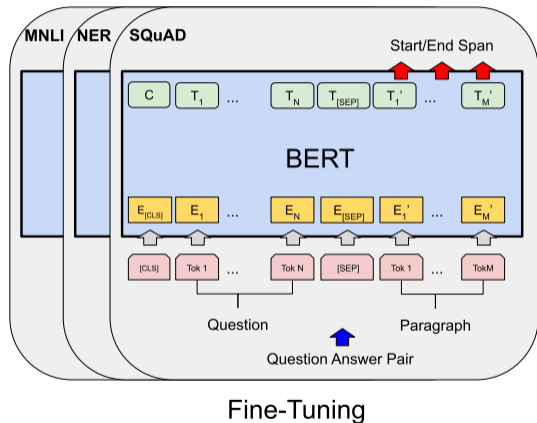
## Pre-Training

- 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 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)

## Fine-tuning

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

## Fine-tuning

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

- It fast enough on common hardware (GPU)

## Fine-tuning

Fine tuning is ok with BERT → 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 → 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 → 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 → 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 → a huge model)

## Fine-tuning

Fine tuning is ok with BERT → 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 → a huge model)

## Fine-tuning

Fine tuning is ok with BERT → 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 → a huge model)

More recent models ?

## Fine-tuning

Fine tuning is ok with BERT → 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 → a huge model)

## More recent models ?

- GPT2-large → 778M weights

## Fine-tuning

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

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

(however in 2018 → a huge model)

## More recent models ?

- GPT2-large → 778M weights
- Mixtral-8x7B → 56B weights

## Fine-tuning

Fine tuning is ok with BERT → 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 → a huge model)

## More recent models ?

- GPT2-large → 778M weights
- Mixtral-8x7B → 56B weights
- Llama3-70B → 70B weights

## Fine-tuning

Fine tuning is ok with BERT → 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 → a huge model)

## More recent models ?

- GPT2-large → 778M weights
- Mixtral-8x7B → 56B weights
- Llama3-70B → 70B weights
- Bloom → 176B weights

## Fine-tuning

Fine tuning is ok with BERT → 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 → a huge model)

## More recent models ?

- GPT2-large → 778M weights
- Mixtral-8x7B → 56B weights
- Llama3-70B → 70B weights
- Bloom → 176B weights

→ Can we adapt the model to our task efficiently ?

## Fine-tuning

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

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

(however in 2018 → a huge model)

## More recent models ?

- GPT2-large → 778M weights
- Mixtral-8x7B → 56B weights
- Llama3-70B → 70B weights
- Bloom → 176B weights

→ Can we adapt the model to our task efficiently ?

→ Can we compress the models to obtain same performances ?

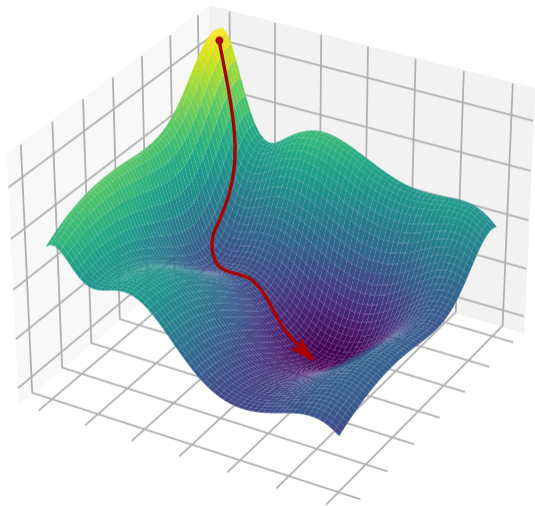
# Parameter Efficient Fine Tuning (PEFT)

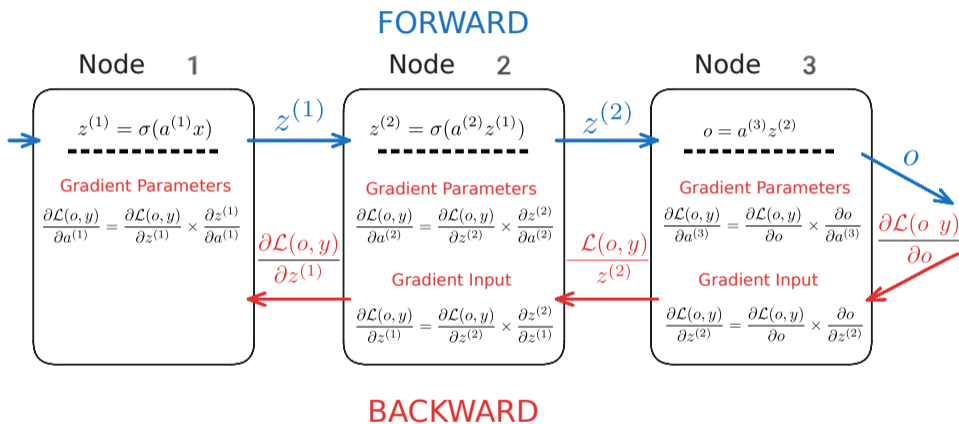
---

## Fine-tuning

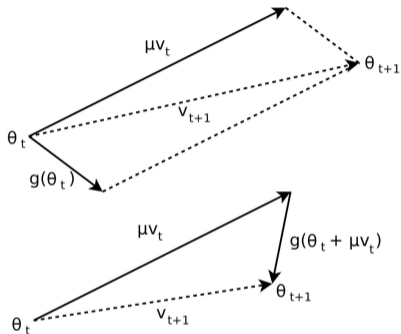
→ Train the model on a new task based on pretrained weights

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





→ Storing weights but also gradient parameters !!!



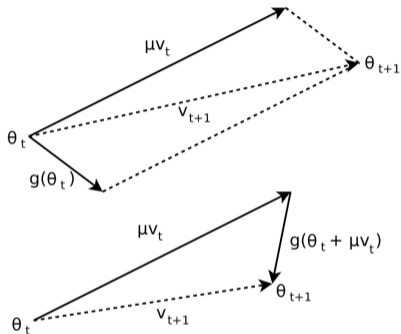
When Fine-tuning : SGD with momentum

Updating weights with a momentum :

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

with

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



When Fine-tuning : SGD with momentum

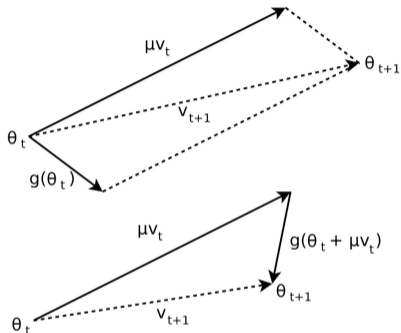
Updating weights with a momentum :

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

with

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

→ Storing a copy of the gradient (matrix  $V_t$ )



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

When Fine-tuning : SGD with momentum

Updating weights with a momentum :

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

with

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

→ Storing a copy of the gradient (matrix  $V_t$ )

→ With 32 bits for a float

→ With  $n_\theta$  the number of parameters

## When Fine-tuning : Adam

Updating weights with Adam:

→ It uses two moments !!!

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

---

```
input :  $\gamma$  (lr),  $\beta_1, \beta_2$  (betas),  $\theta_0$  (params),  $f(\theta)$  (objective)
         $\lambda$  (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  $\dots$  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  $\lambda \neq 0$ 
     $g_t \leftarrow g_t + \lambda \theta_{t-1}$ 
   $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$ 
   $v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$ 
   $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ 
   $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ 
  if amsgrad
     $\widehat{v}_t^{max} \leftarrow \max(\widehat{v}_t^{max}, \widehat{v}_t)$ 
     $\theta_t \leftarrow \theta_{t-1} - \gamma \widehat{m}_t / (\sqrt{\widehat{v}_t^{max}} + \epsilon)$ 
  else
     $\theta_t \leftarrow \theta_{t-1} - \gamma \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ 


---


return  $\theta_t$ 

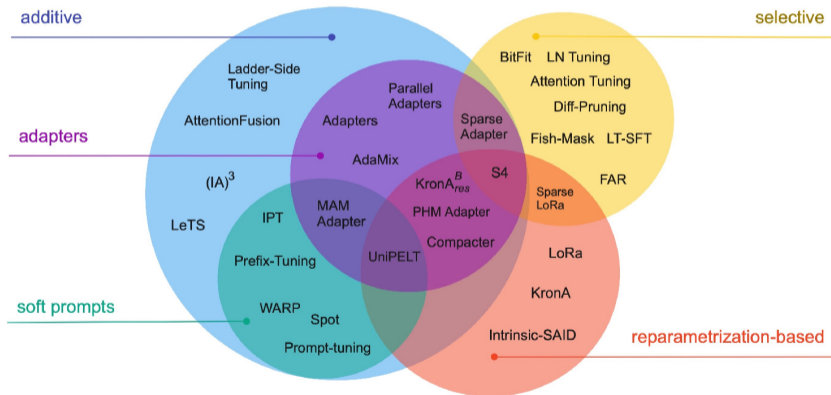

---


```

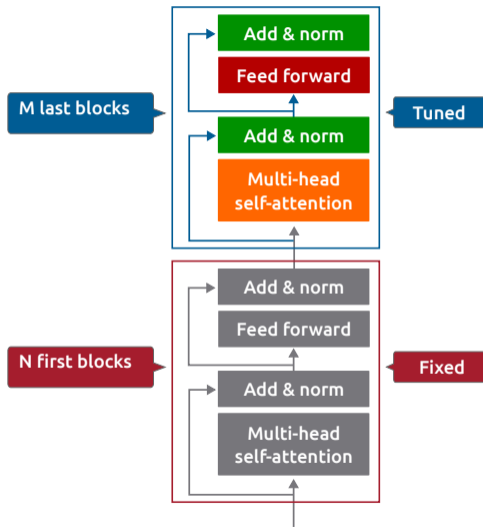
What can/should we do ?

- 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”



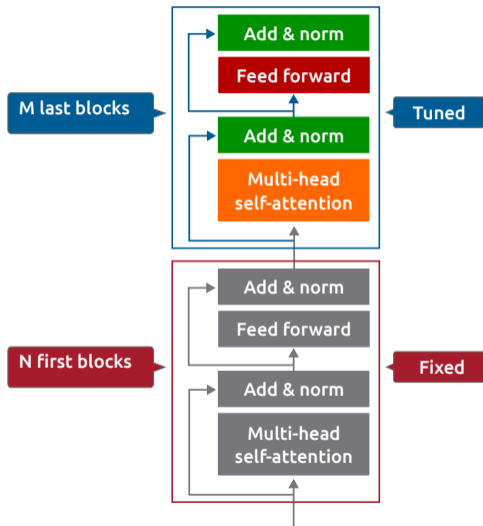
# PEFT: Last Layer(s) only



## 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

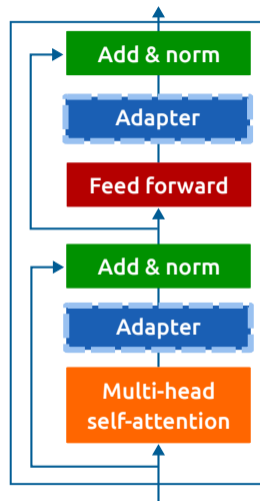
- Reduce the number of gradient “information” to store (% of layer fixed)
- Correct performances (if enough layers)

→ can we do better?

# PEFT: Adapter approaches

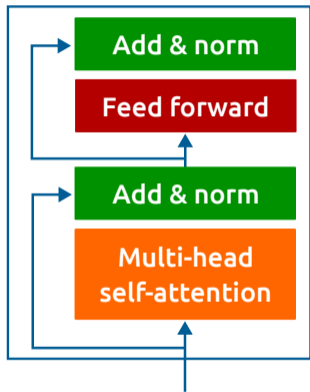
## The Adapter approach

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

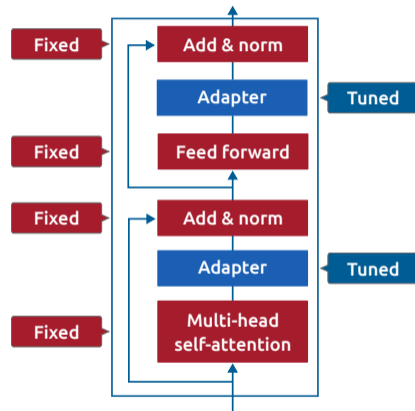


“Parameter-Efficient Transfer Learning for NLP”, N. Houlsby et al., ICML 2019

# PEFT: Adapter approaches



→



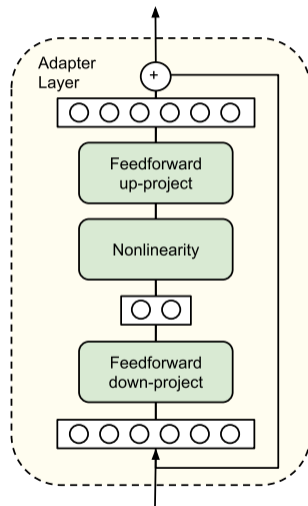
“Parameter-Efficient Transfer Learning for NLP”, N. Houlsby et al., ICML 2019

# PEFT: Adapter approaches

## The adapter module

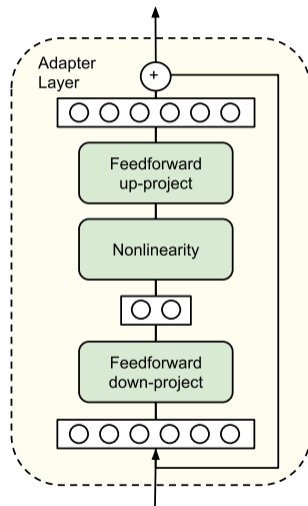
- $W_d \in \mathbb{R}^{k \times n}$  for down projection  $k \ll n$
- An activation function  $\sigma$
- $W_u \in \mathbb{R}^{n \times k}$  for up projection

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



## 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

→ 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

→ 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

→ 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

→ 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

## 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

→ 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

## 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

→ 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

## 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

→ 440 millions parameters in RAM

### FT with Adapter

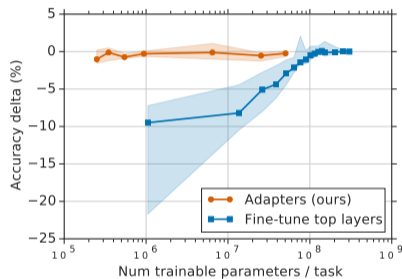
Let  $k = 64$

- $12 \times 64 \times 768 \times 2 \approx 1,2M$  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

→ 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)

## The LoRA approach:

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

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

---

## 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)

### A BSTRACT

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 **Low-Rank Adaptation**, 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 implementation on RoBERTa, DeBERTa, and GPT-2 at [https://github.com/microsoft/LoRA](#).

## Principle

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

→  $W_q^l \in \mathbb{R}^{m \times n}$  the linear associated to the query

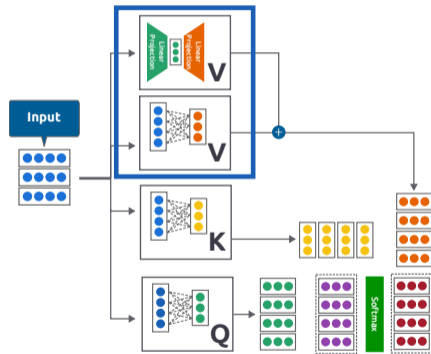
→  $A_q^l \in \mathbb{R}^{k \times n}$  a projection  $k \ll m$

→  $B_q^l \in \mathbb{R}^{n \times k}$  a projection  $k \ll m$

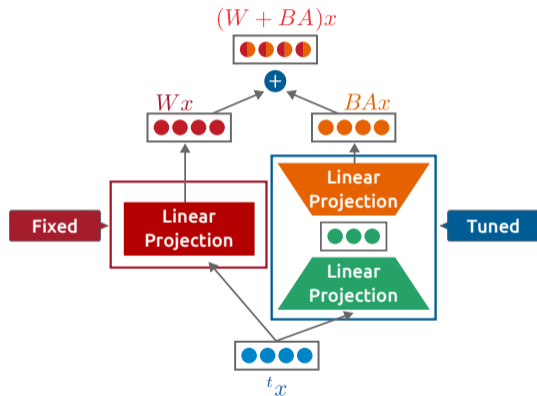
→  $x \in \mathbb{R}^n$  input of the attention block

Output (in the example the query) is computed as :

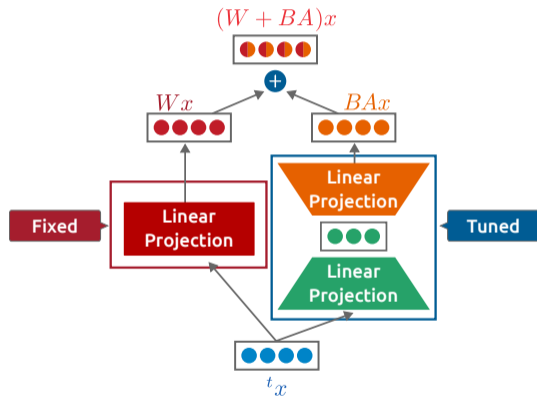
$$Q = W_q^l x + B_q^l (A_q^l x)$$



# PEFT: Adapter approaches - LoRA

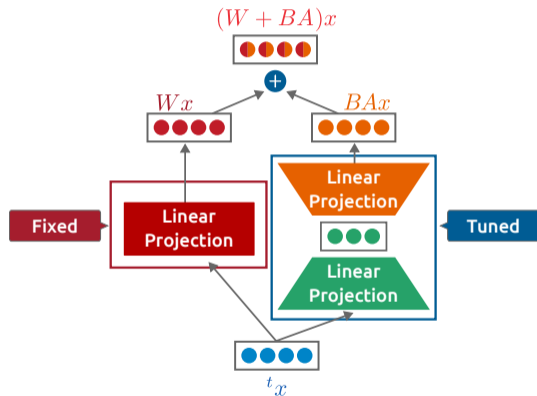


# PEFT: Adapter approaches - LoRA



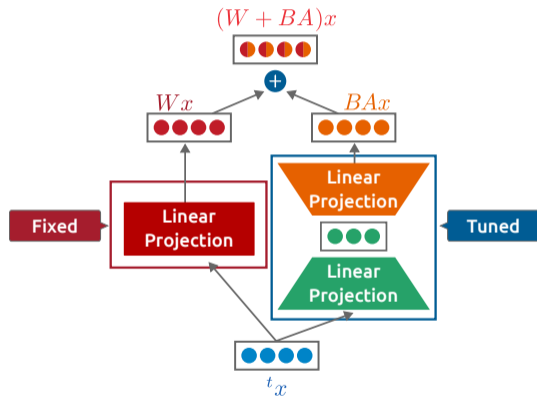
$$Q = W_q^l x + B_q^l (A_q^l x)$$

# PEFT: Adapter approaches - LoRA



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

# PEFT: Adapter approaches - LoRA



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

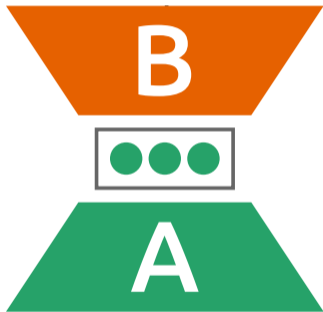
$$Q = (W_q^l + B_q^l A_q^l) x$$

The operation can be factorised !!!

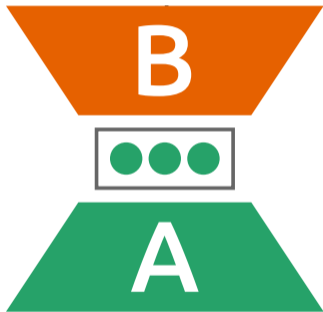
→ At inference change weights of Q linear module, replacing it by :

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

No additional weights at inference !!!

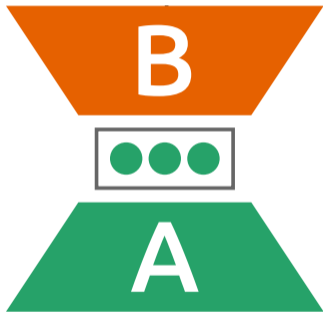


Why low rank?



Why low rank?

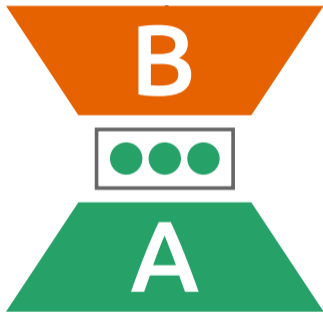
$$\rightarrow \text{rank}(B_q^l A_q^l) \leq k \rightarrow A_q^l \in \mathbb{R}^{k \times n}$$



Why low rank?

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

Why **low** rank?



Why low rank?

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

Why **low** rank?

$$\rightarrow k \ll n$$

# PEFT: Adapter approaches - LoRA (efficiency)

## Efficiency (memory)

- For training → similar to bottleneck adapter (slightly lower memory)
- For inference → similar to base model

→ 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.

# PEFT: Adapter approaches - LoRA (performances)

Model & Method	# Trainable Parameters	MNLI	SST-2	MRPC	CoLA	QNLI	QQP	RTE	STS-B	Avg.
RoB <sub>base</sub> (FT)*	125.0M	<b>87.6</b>	94.8	90.2	<b>63.6</b>	92.8	<b>91.9</b>	78.7	91.2	86.4
RoB <sub>base</sub> (BitFit)*	0.1M	84.7	93.7	<b>92.7</b>	62.0	91.8	84.0	81.5	90.8	85.2
RoB <sub>base</sub> (Adpt <sup>D</sup> )*	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 $\pm$ 27	89.7 $\pm$ 3	84.4
RoB <sub>base</sub> (Adpt <sup>D</sup> )*	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 $\pm$ 22	90.3 $\pm$ 1	85.4
RoB <sub>base</sub> (LoRA)	0.3M	87.5 $\pm$ 3	<b>95.1<math>\pm</math>2</b>	89.7 $\pm$ 7	63.4 $\pm$ 1.2	<b>93.3<math>\pm</math>3</b>	90.8 $\pm$ 1	<b>86.6<math>\pm</math>7</b>	<b>91.5<math>\pm</math>2</b>	<b>87.2</b>
RoBl <sub>arge</sub> (FT)*	355.0M	90.2	<b>96.4</b>	<b>90.9</b>	68.0	94.7	<b>92.2</b>	86.6	92.4	88.9
RoBl <sub>arge</sub> (LoRA)	0.8M	<b>90.6<math>\pm</math>2</b>	96.2 $\pm$ 5	<b>90.9<math>\pm</math>1.2</b>	<b>68.2<math>\pm</math>1.9</b>	<b>94.9<math>\pm</math>3</b>	91.6 $\pm$ 1	<b>87.4<math>\pm</math>2.5</b>	<b>92.6<math>\pm</math>2</b>	<b>89.0</b>
RoBl <sub>arge</sub> (Adpt <sup>P</sup> )†	3.0M	90.2 $\pm$ 3	96.1 $\pm$ 3	90.2 $\pm$ 7	<b>68.3<math>\pm</math>1.0</b>	<b>94.8<math>\pm</math>2</b>	<b>91.9<math>\pm</math>1</b>	83.8 $\pm$ 29	92.1 $\pm$ 7	88.4
RoBl <sub>arge</sub> (Adpt <sup>P</sup> )†	0.8M	<b>90.5<math>\pm</math>3</b>	<b>96.6<math>\pm</math>2</b>	89.7 $\pm$ 1.2	67.8 $\pm$ 2.5	<b>94.8<math>\pm</math>3</b>	91.7 $\pm$ 2	80.1 $\pm$ 29	91.9 $\pm$ 4	87.9
RoBl <sub>arge</sub> (Adpt <sup>H</sup> )†	6.0M	89.9 $\pm$ 5	96.2 $\pm$ 3	88.7 $\pm$ 29	66.5 $\pm$ 44	94.7 $\pm$ 2	92.1 $\pm$ 1	83.4 $\pm$ 1.1	91.0 $\pm$ 17	87.8
RoBl <sub>arge</sub> (Adpt <sup>H</sup> )†	0.8M	90.3 $\pm$ 3	96.3 $\pm$ 5	87.7 $\pm$ 1.7	66.3 $\pm$ 20	94.7 $\pm$ 2	91.5 $\pm$ 1	72.9 $\pm$ 29	91.5 $\pm$ 5	86.4
RoBl <sub>arge</sub> (LoRA)†	0.8M	<b>90.6<math>\pm</math>2</b>	96.2 $\pm$ 5	<b>90.2<math>\pm</math>1.0</b>	68.2 $\pm$ 1.9	<b>94.8<math>\pm</math>3</b>	91.6 $\pm$ 2	<b>85.2<math>\pm</math>1.1</b>	<b>92.3<math>\pm</math>5</b>	<b>88.6</b>
DeB <sub>XXL</sub> (FT)*	1500.0M	91.8	<b>97.2</b>	92.0	72.0	<b>96.0</b>	92.7	93.9	92.9	91.1
DeB <sub>XXL</sub> (LoRA)	4.7M	<b>91.9<math>\pm</math>2</b>	96.9 $\pm$ 2	<b>92.6<math>\pm</math>6</b>	<b>72.4<math>\pm</math>1.1</b>	<b>96.0<math>\pm</math>1</b>	<b>92.9<math>\pm</math>1</b>	<b>94.9<math>\pm</math>4</b>	<b>93.0<math>\pm</math>2</b>	<b>91.3</b>

## Conclusion

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

## 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 ?

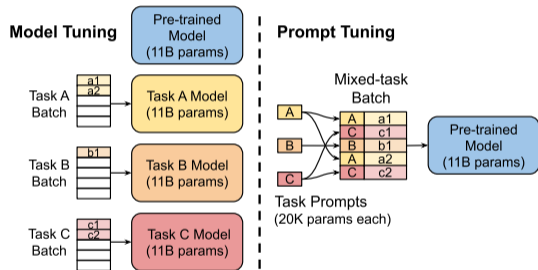
## Continuous prompt tuning

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

---

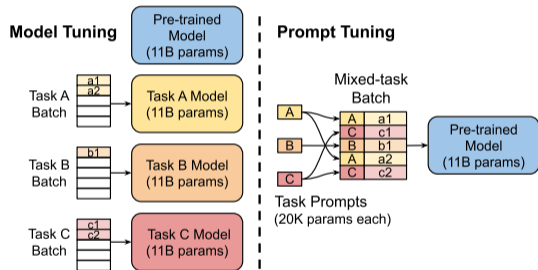
[li-liang-2021-prefix] - li-liang-2021-prefix  
[lester-etal-2021-power] - lester-etal-2021-power

# PEFT: Prompt-tuning



Fine-tuning tokens

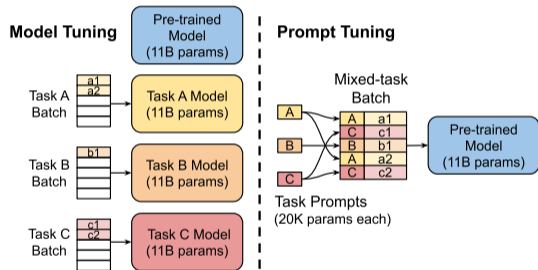
# PEFT: Prompt-tuning



## Fine-tuning tokens

- For each task associate special tokens

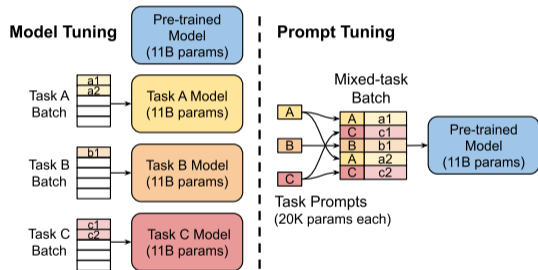
# PEFT: Prompt-tuning



## Fine-tuning tokens

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

# PEFT: Prompt-tuning



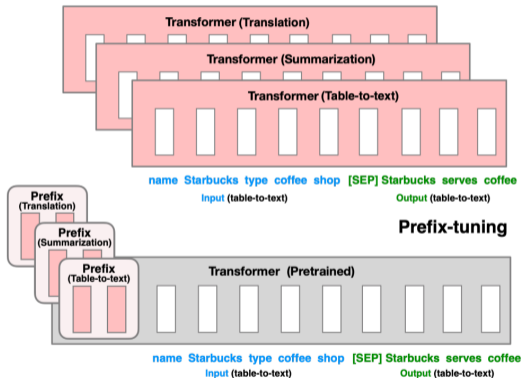
## Fine-tuning tokens

- 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

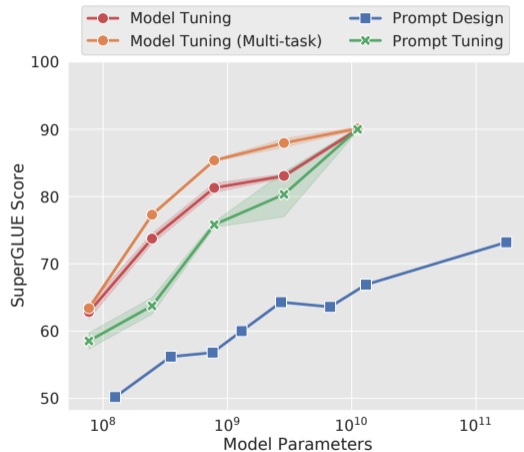


## Fine-tuning tokens

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

→ Only need to retain gradient for those special tokens

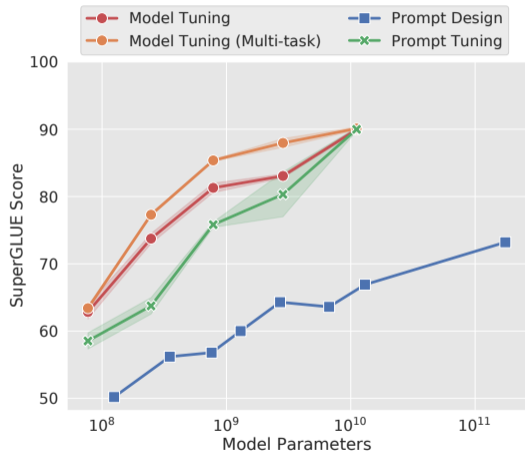
# PEFT: Prompt-tuning



## 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	$\approx 110\text{M}$	$> 418\text{Mo}$
GPT2-medium	$\approx 355\text{M}$	$> 1.2\text{Go}$
Llama-2-7B	$\approx 7\text{B}$	$> 26\text{Go}$
Llama-3-70B	$\approx 70\text{B}$	$> 260\text{Go}$

**Table 2:** 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

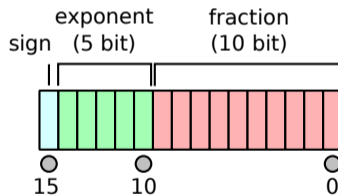
- The model is too large (doesn't fit my GPU)
- Can I compress it with acceptable performance loss?

→ Encode parameters using less bits

Using 16 bits ?

Coding on 16 bits half-precision

→  $\approx 4$  decimal number



<sup>0</sup>image: [https://en.wikipedia.org/wiki/Half-precision\\_floating-point\\_format](https://en.wikipedia.org/wiki/Half-precision_floating-point_format)

### Half-precision issue when training !!

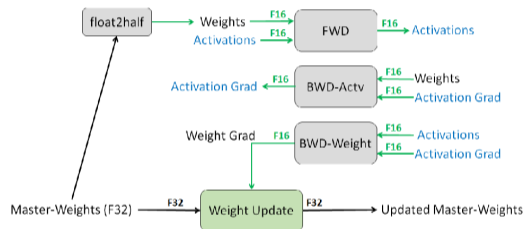
- Computation of gradient, forward, backward is ok
- Parameter update can be unstable

“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

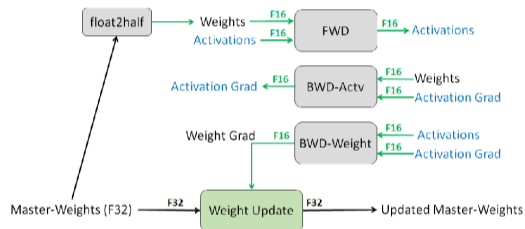


## Mixed Precision

→ “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



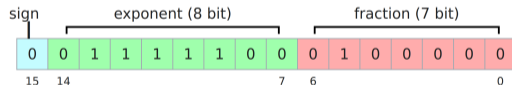
Main issue training in FP16:

**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 exponent
- Using less bits for the inner range values



Can we train/predict using lower precision ?

# Not really PEFT: Quantization

Using 8 bits ?

→ 256 values (-127 to +127)!!!

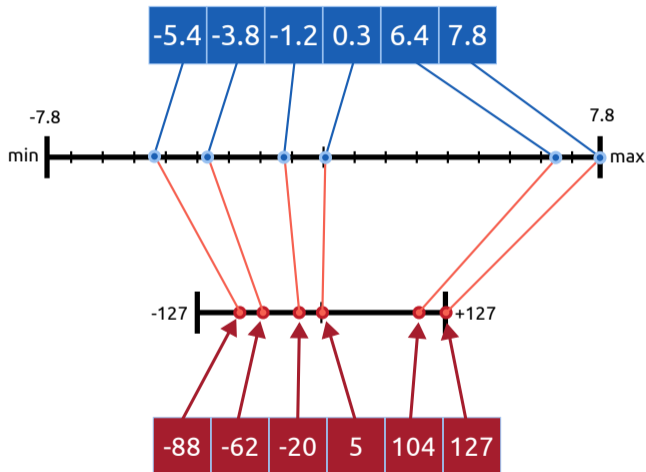
- Mapping FP32 to 8 bits
- Using a scaling factor

## Principle

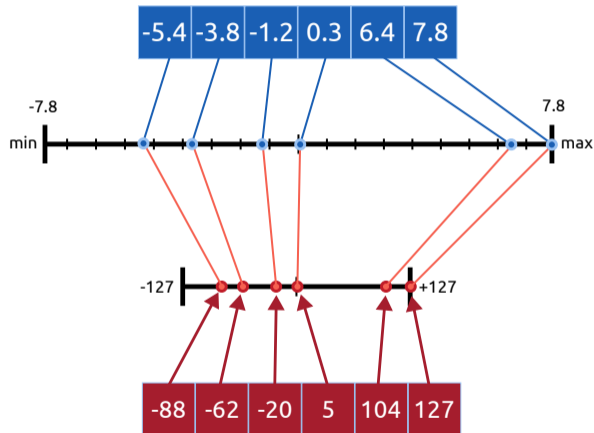
Let have  $W$  a matrix containing as maximum absolute value 7.8. a range of 1 could be represented :

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

values



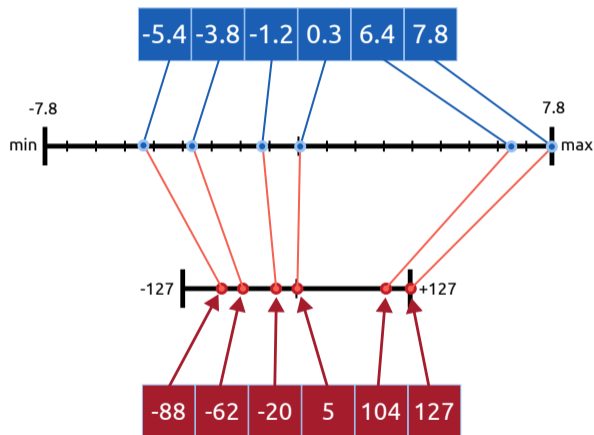
# Not really PEFT: Quantization



Quantization (example)

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

# Not really PEFT: Quantization

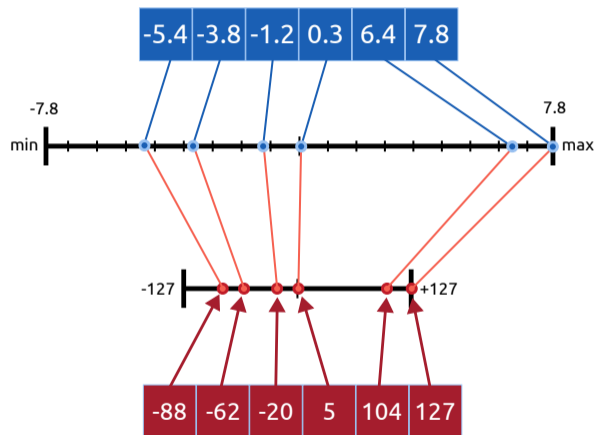


## Quantization (example)

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

$$\cdot -5.4 \times s \rightarrow -\text{round}(87.9) = -88$$

# Not really PEFT: Quantization

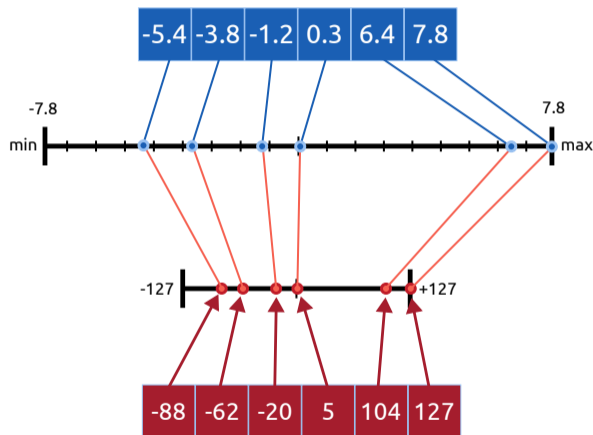


## Quantization (example)

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

- $-5.4 \times s \rightarrow -\text{round}(87.9) = -88$
- $-3.8 \times s \rightarrow -\text{round}(61.8) = -62$

# Not really PEFT: Quantization

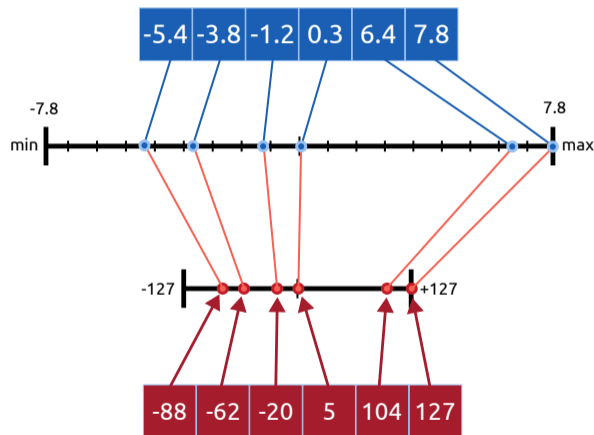


## Quantization (example)

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

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

# Not really PEFT: Quantization

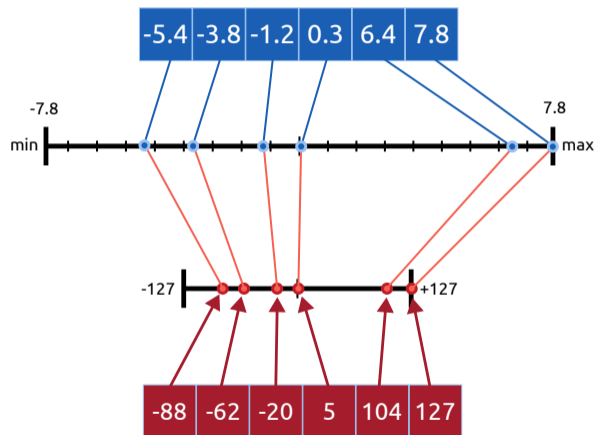


## Quantization (example)

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

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

# Not really PEFT: Quantization

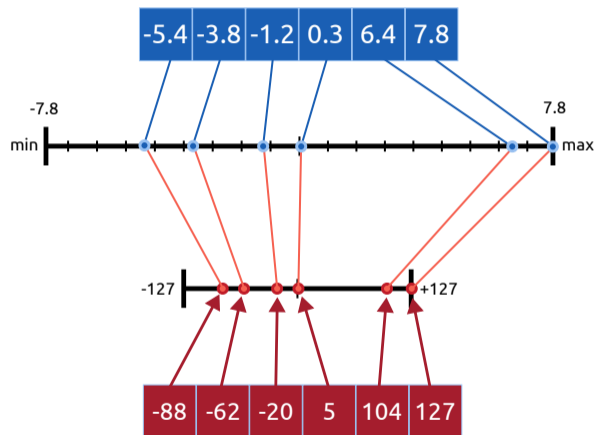


## Quantization (example)

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

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

# Not really PEFT: Quantization

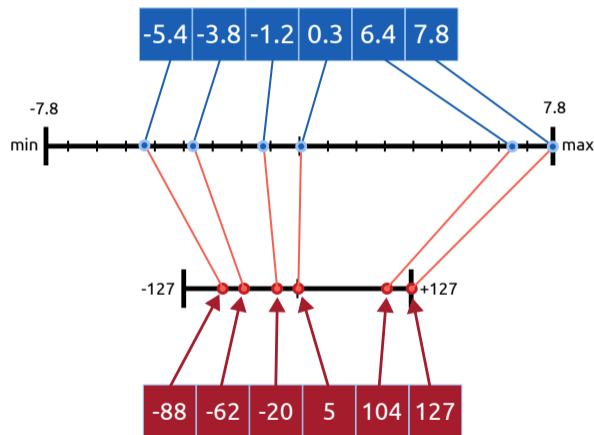


## Quantization (example)

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

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

# Not really PEFT: Quantization



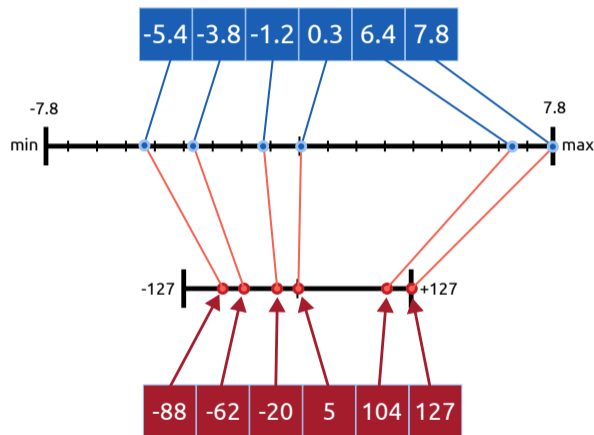
## Quantization (example)

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

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

→ and 7.77 ?

# Not really PEFT: Quantization



## Quantization (example)

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

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

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

## Quantization in neural networks

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

→ Need to have the reverse operation !!

# Not really PEFT: De-Quantization

## De-Quantization (example)

We quantized as following:

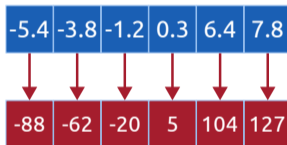


To de-quantize we need to divide by the scaling factor  $s$

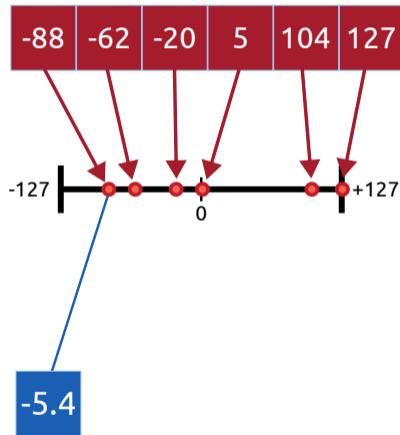
# Not really PEFT: De-Quantization

De-Quantization (example)

We quantized as following:



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



$$s=16.28$$

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



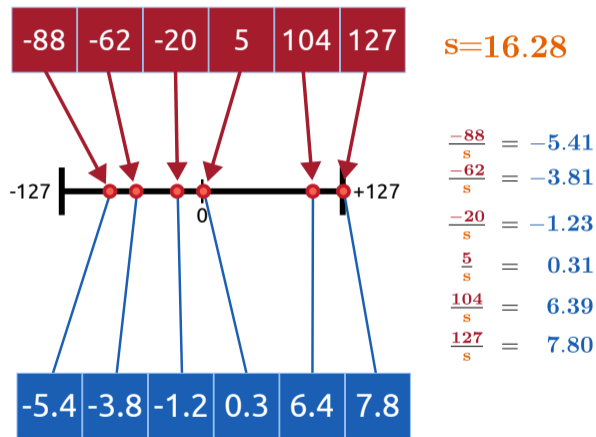
# Not really PEFT: De-Quantization

De-Quantization (example)

We quantized as following:



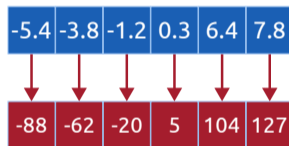
To de-quantize we need to divide by the scaling factor  $s$



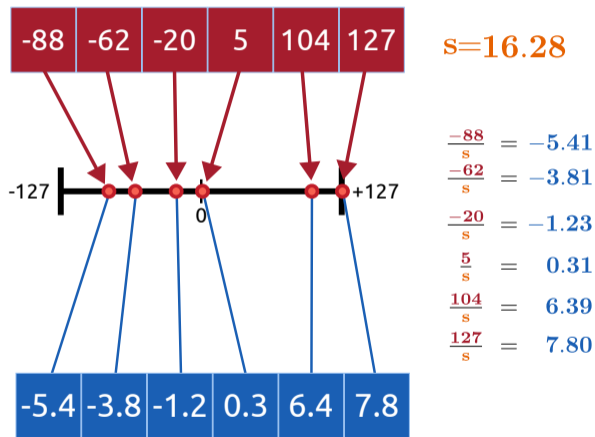
# Not really PEFT: Symetric Quantization

De-Quantization (example)

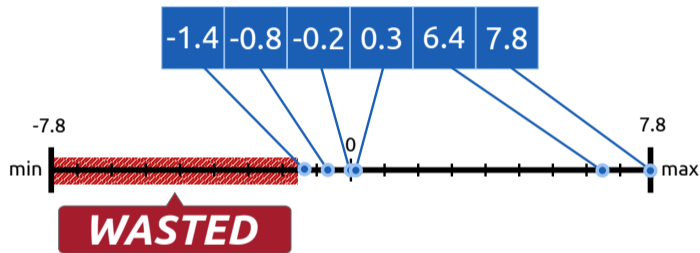
We quantized as following:



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



# Not really PEFT: Asymmetric Quantization



Asymmetric 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

```
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 → Evaluate/testing a large model

## Online ressources ?

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


## Training and quantization

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

## Unstable training ?

→ Mainly due to weight update !!


## Training and quantization

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

## Unstable training ?

- Mainly due to weight update !!
- In **adapter**, only few weights are updated

## Training and quantization

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

## Unstable training ?

- Mainly due to weight update !!
- In **adapter**, only few weights are updated

## Use quantization for non-updated weights ?

## Training and quantization

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

## Unstable training ?

- Mainly due to weight 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

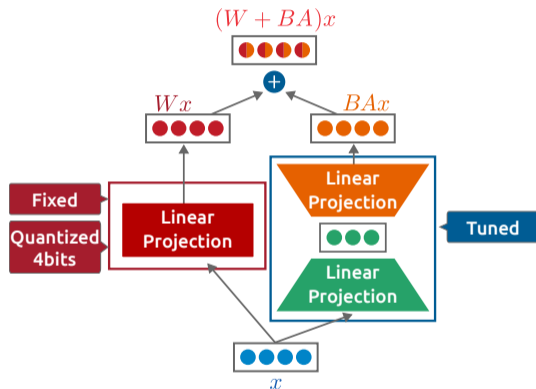
### Abstract

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

# Quantization and PEFT methods : QLoRA

## QLoRA training

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



## Quantization and PEFT methods : QLoRA

Dataset Model	GLUE (Acc.)	Super-NaturalInstructions (RougeL)				
	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 Model	GLUE (Acc.)	Super-NaturalInstructions (RougeL)				
	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

→ 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

→ Do we always need to modify/compress weight (for adaptation) ?

# Adaptation: Need to modify the model ?

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

## One-shot

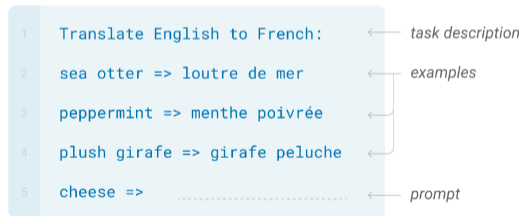
In addition to the task description, the model sees a single example of the task. No gradient updates are performed.



→ In context learning

## Few-shot

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



# Adaptation: Need to modify the model ?

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

## Standard 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 Output

A: The answer is 27. ❌

## 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: 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 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

---

Some performances issues :

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

How to accelerate inference ?

# Optimisation in transformer architecture

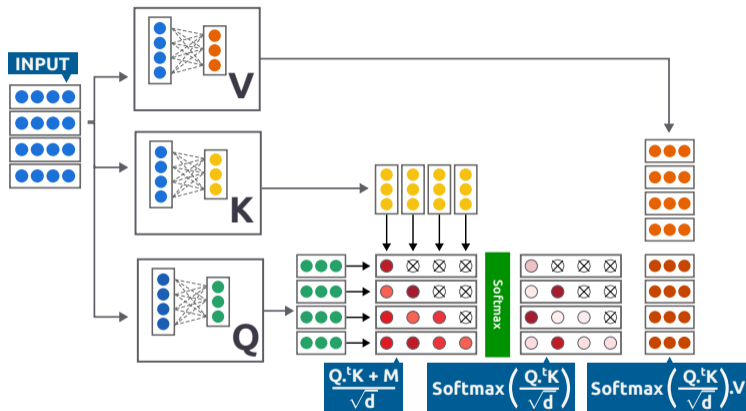


Figure 5: Illustration of the decoder model attention

## Multhead Attention

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



Figure 6: 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

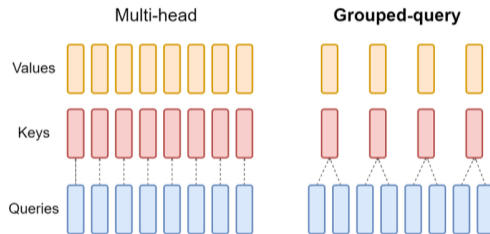
Token	KV produced	cache
1	$k_1, v_1$	$k_1, v_1$
2	$k_2, v_2$	$k_1, v_1, k_2, v_2$
$\vdots$	$\vdots$	$\vdots$
n	$k_n, v_n$	$k_1, v_1, k_2, v_2 \dots, k_n, v_n$

# Optimisation in transformer architecture: Group Queries

## Grouped queries

- Limiting the number of stored parameters
- Restraining the number key/values heads (one per query group) [ainslie2023gqa]

[ainslie2023gqa] - ainslie2023gqa



# Optimisation in transformer architecture: Global Attention

In decoder approaches:

Use self-attention (or global attention)

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

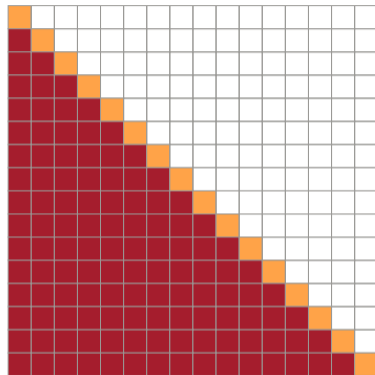


Figure 7: Illustration of the global attention matrix

# 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 [beltagy2020longformer]

---

[beltagy2020longformer] - beltagy2020longformer

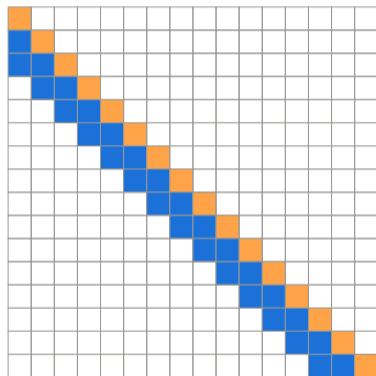


Figure 8: Illustration of the local attention matrix

## Example

Let consider:

- Three layers transformer
- 4 windowed attention

What token for 11<sup>th</sup> output is  
"taken" in consideration ?

# Optimisation in transformer architecture: local Attention

## Example

Let consider:

- Three layers transformer
- 4 windowed attention

What token for 11<sup>th</sup> output is "taken" in consideration ?

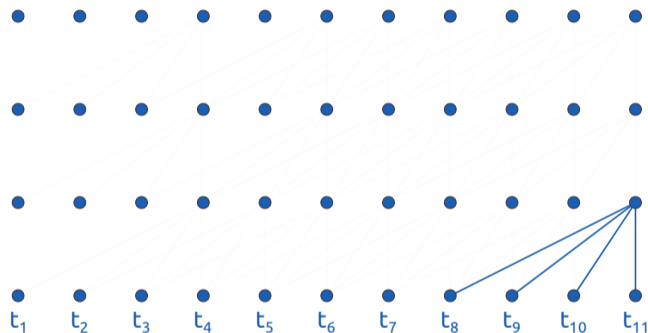


Figure 9: Receptive field in windowed attention

# Optimisation in transformer architecture: local Attention

## Example

Let consider:

- Three layers transformer
- 4 windowed attention

What token for 11<sup>th</sup> output is "taken" in consideration ?

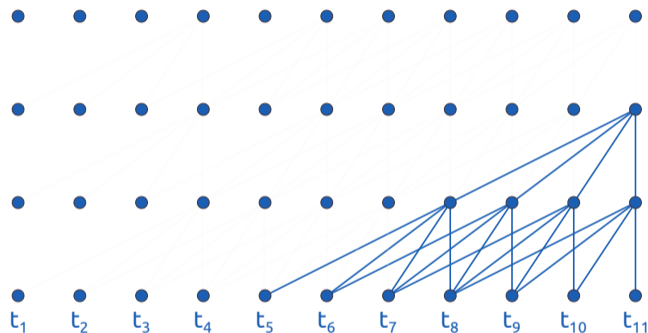


Figure 9: Receptive field in windowed attention

# Optimisation in transformer architecture: local Attention

## Example

Let consider:

- Three layers transformer
- 4 windowed attention

What token for 11<sup>th</sup> output is "taken" in consideration ?

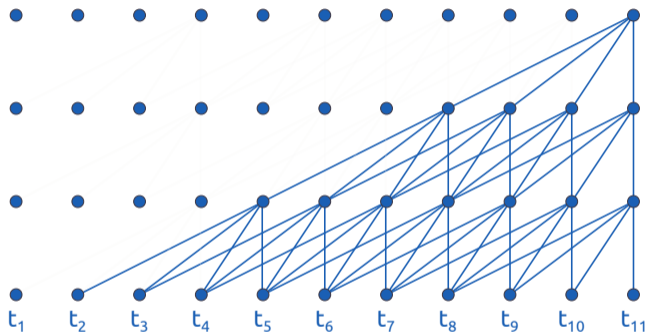


Figure 9: Receptive field in windowed attention

# Optimisation in transformer architecture: local Attention

## Example

Let consider:

- Three layers transformer
- 4 windowed attention

What token for 11<sup>th</sup> output is  
"taken" in consideration ?

→ from 2 to 11

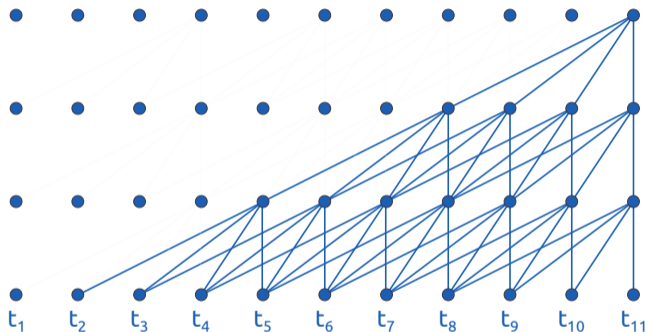
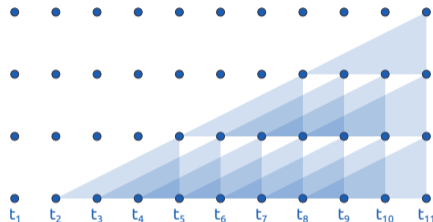


Figure 9: Receptive field in windowed attention

# Optimisation in transformer architecture: local Attention

Receptive field size:

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



Advantages

- Longer sequences for lower computational/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

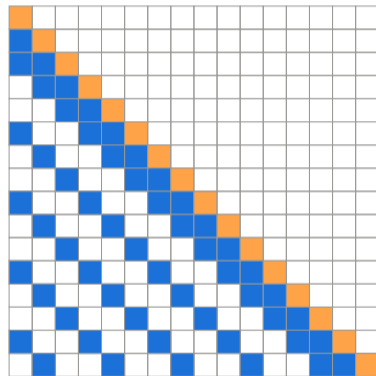


Figure 10: Illustration of the dilated attention matrix

# Optimisation in transformer architecture: Mixed Attention

## Using different sparse attention

- BigBird (Random + Local + Global) [zaheer2020big]
- Longformer (Dilated + Local + Global) [beltagy2020longformer]
- Attention Sink [xiao2023efficient]

→ Few large model use it...

---

[zaheer2020big] - zaheer2020big

[beltagy2020longformer] - beltagy2020longformer

[xiao2023efficient] - xiao2023efficient

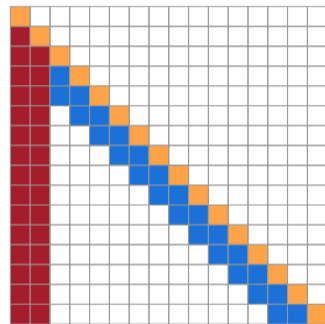


Figure 11: Mxing global and local attention

# Linear attention

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

$$O = \text{softmax}\left(\frac{Q \cdot {}^tK}{\sqrt{D}}\right) \cdot V$$

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

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

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

$$O = \text{softmax}\left(\frac{Q \cdot {}^tK}{\sqrt{D}}\right) \cdot V$$

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

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

# Linear attention

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

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

We can write  $O_l$  as it follows :

$$\begin{aligned} O_l &= \frac{\frac{Q_l \cdot {}^tK}{\sqrt{D}}}{\sum_{j=0}^L \left(\frac{Q_l \cdot {}^tK}{\sqrt{D}}\right)_j} \cdot V \\ &= \frac{\frac{Q_l}{\sqrt{D}} \cdot \frac{{}^tK}{\sqrt{D}} \cdot V}{\sum_{j=0}^L \left(\frac{Q_l}{\sqrt{D}} \cdot \frac{{}^tK}{\sqrt{D}}\right)_j} = \frac{\frac{Q_l}{\sqrt{D}} \cdot \left(\frac{{}^tK}{\sqrt{D}} \cdot V\right)}{\frac{Q_l}{\sqrt{D}} \sum_{j=0}^L \left(\frac{{}^tK}{\sqrt{D}}\right)_j} \end{aligned}$$

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

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

→ We can compute KV once !!!

# Linear attention

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

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

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

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

→ Replace with function close to softmax :

- $\phi, \psi$  being  $1 + \text{elu}(x)$  [katharopoulos2020transformers]
- Many other kernels...

---

<sup>0</sup>[katharopoulos2020transformers] - katharopoulos2020transformers

# Linear attention: in decoder (autoregressive) models

## Linear attention and decoder

In autoregressive models, we can only access previous tokens...

→ For  $O_t$  we must compute  ${}^tK_{:,0:t}V_{0:t}$

→ Cannot compute once for all

Can be done using a loop for similar complexity

→ 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 develop new approaches

→ Pytorch

## Using high level libraries

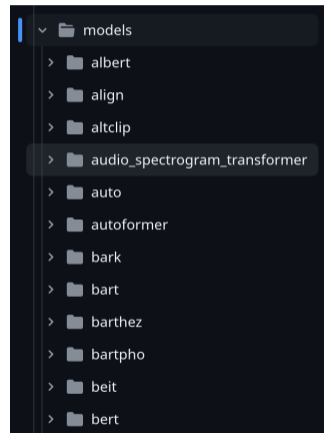
- Easy to implement
- Missing flexibility

→ transformers, PEFT

## Finding pytorch code

Hopfully, most models are implemented in pytorch

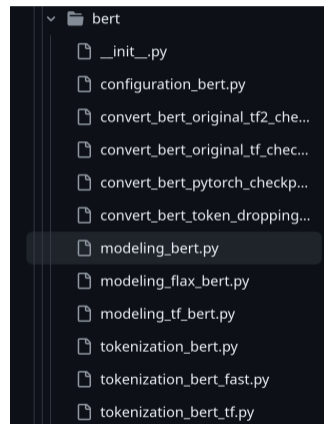
- `https://github.com/huggingface/transformers/tree/main/src/transformers/models`



## 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

```
972  ✓      def __init__(self, config, add_pooling_layer=True):
973          super().__init__(config)
974          self.config = config
975
976          self.embeddings = BertEmbeddings(config)
977          self.encoder = BertEncoder(config)
978
979          self.pooler = BertPooler(config) if add_pooling_layer else None
980
```

What is the architecture of the model ?

→ We can print the model

What is the architecture of the model ?

→ 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)
    )
  )
)
```

## Defining the module

Defining a module that will replace original linear module

```
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

```
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
```

→ Where are the modules we want to replace ?

→ 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 ?

→ We can print the model

What is the architecture of the model ?

→ 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)
              )
            )
          )
        )
      )
    )
  )
)
```

Training the model !!!

Training the model !!!

```
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()
```

## Training the model !!!

```
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

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

→ HuggingFace face proposes the PEFT library !!!

```
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 !!!

```
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 !!!

## To conclude

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

Questions ?



<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)

## 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)

## 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 ?