VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
Faculty of Computer Science and Engineering

Natural Language Processing (CO3086)

NLP Assignment

Supervisor:		Bui Khanh Vinh	
Group 5 - CC01			
Student:	2252898	Huynh Ngoc	Van
	2252434	Tran Gia	Linh
	2252757	Le Thi Phuong	Thao
	2252654	Huynh Lan	Phuong
	2252803	Nguyen Ngoc Song	Thuong

Ho Chi Minh City, June 1, 2025

No.	Fullname	Student ID	Tasks	Completion
1	Huynh Ngoc Van	2252898	GPT - NER; Contributed to writing Sections 3, 4, 6	100%
2	Tran Gia Linh	2252434	GPT - Sentiment, GPT - Text Generation; Wrote Sections 2, 4, 6	100%
3	Le Thi Phuong Thao	2252757	T5 - Sentiment, T5 - Question Answering; Wrote Sections 3, 4, 5, 6	100%
4	Huynh Lan Phuong	2252654	BERT - Sentence Similarity, T5 - Summarization; Wrote Sections 4,6	100%
5	Nguyen Ngoc Song Thuong	2252803	BERT - Sentiment, BERT - NER; Wrote Sections 4, 5, 6	100%

Workload Distribution for the assignment

GitHub: NLP_Assignment_Group05_CC01

Contents

1 Introduction

The field of **Natural Language Processing (NLP)** has experienced a profound transformation, driven by a paradigm shift in how computational models process and interpret human language. A landmark development in this progression was the introduction of the **Transformer architecture** by Vaswani et al. [18] in their influential paper, *"Attention Is All You Need"*. This architectural innovation redefined neural network design for sequential data by replacing traditional recurrent and convolutional components with **self-attention mechanisms**. This shift greatly enhanced **parallelization**, resulting in significant improvements in **training efficiency** and overall model performance.

The emergence of the Transformer architecture paved the way for the rapid scaling and advancement of **Large Language Models (LLMs)**. Notable examples include **OpenAI's GPT series** (e.g., **GPT-3, GPT-4**) and **Google's PaLM models**, which feature **billions to trillions of parameters** and exhibit exceptional capabilities across a wide range of NLP tasks, such as advanced text generation and machine translation. These models mark a substantial leap forward in artificial intelligence, demonstrating an unprecedented level of **fluency** and **contextual understanding** in handling complex linguistic challenges.

However, the enormous size of these LLMs introduces significant challenges, particularly in terms of the **computational demands** associated with traditional **full fine-tuning** approaches. Fully updating all parameters of such large pre-trained models for downstream tasks often requires **substantial computational resources, memory, and time**. This constraint has driven the development of **parameter-efficient fine-tuning methods**, which adapt pre-trained models to new tasks or domains by updating only a **small subset of parameters**, thereby significantly reducing computational overhead while striving to preserve strong performance.

This report provides a comprehensive overview of the **Transformer architecture**, emphasizing its foundational principles and its transformative effect on NLP. We also examine the architectures of three prominent Transformer-based LLMs—**BERT**, **GPT**, and **T5**—and explore **fine-tuning techniques** applicable to these models for task-specific applications. Through **experimental evaluations** involving these models, we discuss the results and **comparative effectiveness** of different fine-tuning strategies across various NLP tasks.

2 Evolution of large language models (LLMs)

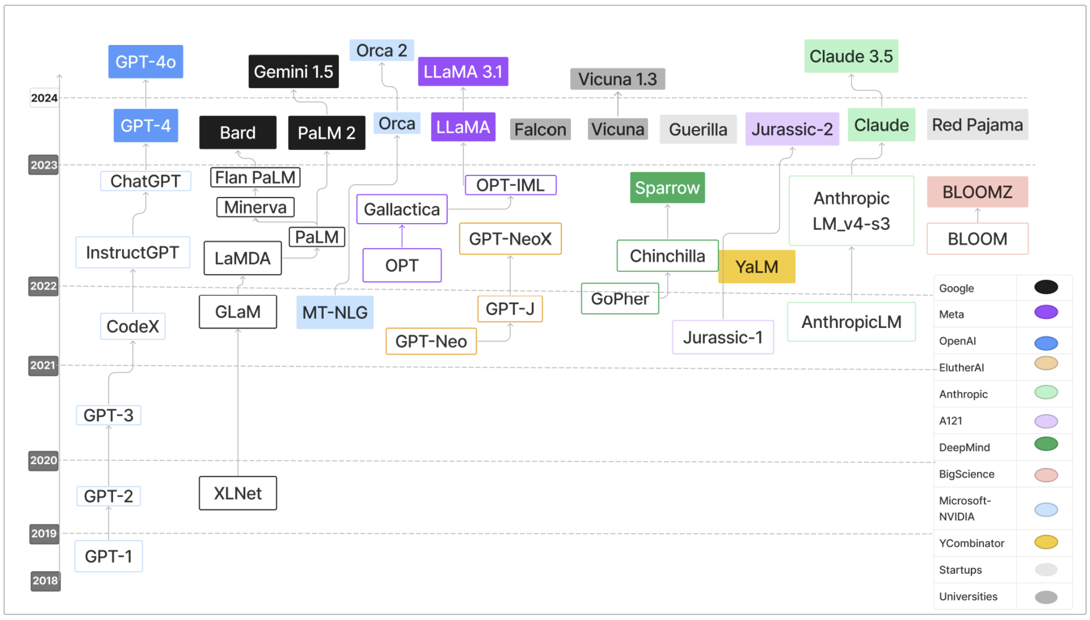


Figure 1: *The evolution of LLMs overtime*

Over the past decade, **large language models (LLMs)** have undergone a dramatic evolution, reshaping the landscape of **Natural Language Processing (NLP)**. This evolution can be traced through several key milestones and innovations that have progressively expanded model capabilities, scalability, and practical applications.

- Early NLP systems relied heavily on **static word embeddings** such as **Word2Vec** (Mikolov et al., 2013) and **GloVe** (Pennington et al., 2014). These models mapped each word to a single dense vector regardless of its context.
- A significant leap came with **contextual embeddings**, notably with **ELMo (Peters et al., 2018)**, which generated word representations depending on their surrounding context. This marked the transition from shallow embedding models to deep contextual models.

2.1 The Emergence of Transformers

- The introduction of the **Transformer architecture** by Vaswani et al. [18] in 2017 revolutionized NLP. Its core innovation—the **self-attention mechanism**—allowed the model to capture dependencies across sequences **without relying on recurrence**, enabling parallelization and scalability.
- **Transformer-based models** became the foundation for a new generation of powerful LLMs.

2.2 Milestones in Large Language Models

2.2.1 Early Stages (2019–2020)

The initial phase of LLM development introduced foundational models that redefined the NLP landscape:

- Google's T5 (Text-To-Text Transfer Transformer, 2019) unified NLP tasks under a text-to-text framework and demonstrated the versatility of transfer learning using encoder-decoder transformers [16].
- OpenAI released GPT-3, a 175-billion-parameter autoregressive model that showcased remarkable zero-shot and few-shot generalization abilities across diverse tasks without task-specific fine-tuning [2]. This was a pivotal moment, positioning LLMs as capable meta-learners.
- mT5 extended the T5 architecture to over 100 languages, proving that multilingual models could achieve performance parity with their monolingual counterparts [20].

2.2.2 Expansion and Multilingual Capabilities (2021)

The year 2021 marked an era of expansion and architectural innovation. New models such as PanGu-α, CPM-2, ERNIE 3.0, Jurassic-1, HyperCLOVA, and Gopher emerged with parameter sizes ranging from 10 billion to over 200 billion. These models emphasized **few-shot learning, language representation, and scalable pretraining** [15]:

- ERNIE 3.0: focused on knowledge integration and multilingual tuning.
- HyperCLOVA: was trained entirely on Korean data, pushing the boundaries of high-quality, language-specific LLMs.
- Jurassic-1: AI21 Labs introduced controlled text generation through prompt-based interfaces.
- DeepMind's Gopher: explored knowledge retrieval and factual accuracy in large-scale generative models

2.2.3 Scaling and Efficiency (2022)

In 2022, research shifted toward improving efficiency and optimizing training:

- ERNIE 3.0 Titan introduced **adversarial loss mechanisms** to enhance robustness and generalization.
- GPT-NeoX-20B, developed by EleutherAI, demonstrated how community-driven open-source efforts could produce performant large models using parallel transformer layers [1].
- Google's GLaM (Generalist Language Model) pioneered the use of **Mixture-of-Experts (MoE)** within the transformer architecture, activating only subsets of model parameters per input to significantly reduce computational costs [4].
- DeepMind's Chinchilla challenged previous scaling laws by showing that training on more

tokens—rather than increasing parameters—led to better performance, thereby **initiating a new paradigm in model scaling strategies**. [5].

2.2.4 Specialization and Cost Efficiency (2023)

In 2023, LLMs became more practical and application-specific. LLM technology becomes more accessible and lightweight, highlighting a move toward efficient deployment in both cloud and edge scenarios.

- GPT-4 was released with improved reasoning capabilities and support for multimodal tasks (text and image input), though its technical details remain undisclosed.
- Claude Instant and GPT-4 Turbo introduced optimized versions for real-time applications, emphasizing reduced latency and lower operational costs.
- Domain-specific models gained prominence—StarCoder was designed for programming and code synthesis tasks, while Mistral Tiny targeted low-resource environments.

2.2.5 Innovations in Instruction Tuning and Safety (2024)

The developments in 2024 have emphasized safety, factual grounding, and instruction following. These advancements reflect an increasing awareness of AI safety and the societal impact of LLMs, paving the way for responsible and reliable AI systems.

- Gemini 1.5 and LLaMA-2-Chat **incorporated instruction-tuned architectures and reinforcement learning from human feedback (RLHF)** to align outputs with user expectations and ethical guidelines.
- WebGPT focused on producing verifiable, reference-backed answers through web browsing [12].
- BLOOMZ and mT0 continued pushing multilingual generalization.

2.2.6 Toward Generalization and Agentic Behavior (2025)

In 2025, LLM research has increasingly focused on general-purpose intelligence, autonomy, and integration with external tools. A defining characteristic of this phase is the emergence of agent-like LLMs capable of **planning, reasoning, and interacting with dynamic environments**:

- GPT-4.5 and Claude 3 Opus demonstrated improved grounding, persistent memory, and long-context reasoning across modalities. These models can process tens or even hundreds of thousands of tokens, enabling document-level comprehension and multi-turn task execution.
- OpenAI's Assistants API and Google's Gemini 1.5 Pro began incorporating tool use, allowing LLMs to invoke APIs, browse the web, generate charts, write code, or call external plugins. This shift toward tool-augmented LLMs reflects a movement from pure language models to **"AI agents"** that can act autonomously to complete goals.
- Instruction-following, alignment, and safety also saw improvements with advanced reinforcement learning from human feedback (RLHF), direct preference optimization (DPO), and constitutional AI techniques.

Moreover, language models are becoming increasingly specialized—smaller models like Mistral 7B, Gemma 2B, and Phi-3 are optimized for edge deployment, democratizing access to powerful LLMs on local devices. As open-source initiatives such as LLaMA 3 and DeepSeek-VL expand the frontier in multilingual and multimodal learning, the focus in 2025 is clearly on building efficient, controllable, and extensible systems that can serve as robust foundations for general-purpose AI.

2.2.7 Summary of evolution of LLMs

The trajectory of LLM development from 2019 to 2025 reflects a dynamic and multifaceted evolution marked by **innovations in model scaling, multilingual capabilities, training efficiency, specialization, and safety**.

- Early breakthroughs such as GPT-3 and T5 emphasized large model size and few-shot learning capabilities. Subsequent years saw improvements in multilingual performance, architectural innovations, and cost-effective deployment strategies.
- By 2023, LLMs were increasingly specialized for practical applications and multimodal integration.
- By 2024, the community focused on instruction tuning, alignment with human intent, and safety in model behavior.
- By 2025, attention shifted to generalization, long-context reasoning, agentic behavior, and tool augmentation, signaling a move from static models to interactive, goal-driven AI agents.

This evolution not only underscores the centrality of LLMs in modern NLP but also highlights their growing role as foundational components for general-purpose, controllable, and extensible AI systems.

3 Overview of Transformer models

The introduction of the **Transformer architecture** by Vaswani et al. [18] in 2017 was a major turning point in Natural Language Processing (NLP). This model replaced Recurrent Neural Networks (RNNs) and Convolutional Neural Networks (CNNs) with a new mechanism called self-attention, removing the need for recurrence and convolution entirely. This change was not just an improvement—it redefined how models process sequences. It allowed faster training through parallel computation and significantly improved model performance [18].

The key innovation in the Transformer is **the self-attention mechanism**. This lets the model determine which words in a sentence are most relevant when interpreting each word [18]. Unlike RNNs that read tokens one at a time, self-attention enables the model to consider all tokens at once, making it easier to understand long-range relationships between words. Self-attention works by creating Query (Q), Key (K), and Value (V) vectors from input embeddings. The attention scores are calculated from Q and K, and these scores are used to weight the V vectors [18].

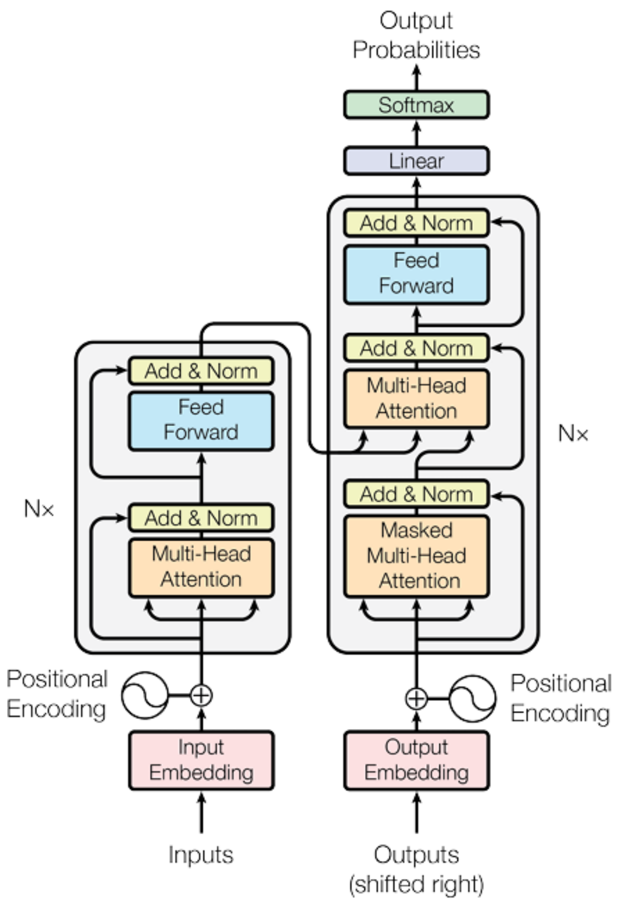


Figure 2: *Transformer Architecture*

The Transformer model has several important parts:

- **Encoder-Decoder Architecture:** The original Transformer uses an encoder-decoder structure [18]. The encoder takes the input and creates a contextualized representation. The decoder uses this, along with its own inputs, to produce the output. Both encoder and decoder consist of repeated identical layers.

- **Positional Encoding:** Because the model does not process sequences in order, positional encodings are added to the input to give it information about token positions. These encodings are based on sine and cosine functions and help the model understand word order [18].

- **Multi-Head Attention:** This component allows the model to attend to different parts of the input in multiple ways. It projects Q, K, and V vectors into different spaces, runs self-attention in parallel, then combines the results. This helps the model learn different types of relationships in the data [18].

- **Feed-Forward Networks:** Each encoder and decoder layer contains a fully connected feed-forward network that applies the same transformation to each position separately [18].

- **Residual Connections and Layer Normalization:** To help train deep models, residual connections are used around each sub-layer, followed by layer normalization. This improves gradient flow and training stability [18].

The Transformer greatly influenced modern NLP. Its design enabled models like BERT, GPT, and T5 [18]. The ability to process sequences in parallel and understand long-range context led to state-of-the-art results in tasks like machine translation and text generation.

The most important change the Transformer introduced was switching from sequential to parallel processing. Earlier models like RNNs processed inputs step-by-step, which limited their ability to handle long-range dependencies and made training slow. The Transformer reads the full input sequence at once and relies only on self-attention to understand context [18]. This makes training faster and helps the model learn better representations. Without this change, it would not be possible to train large language models like GPT and BERT on the scale we see today.

Overall, the Transformer using stacked self-attention and point-wise operations consists of two main components: the **Encoder** and the **Decoder**, designed to process sequential input data in parallel.

3.0.0.1 Encoder

Processes the input and generates a hidden representation. It consists of several identical layers, each with two main sub-components:

- **Self-Attention Mechanism:** Allows each word in the input to “attend to” other words, capturing their relationships.

- **Feed-Forward Neural Network (FFN):** A simple neural network applied independently to each position in the sequence.

3.0.0.2 Decoder

Processes the hidden representation from the Encoder and generates the output sequence. It also consists of several identical layers, each with three main sub-components:

- **Masked Self-Attention:** Ensures the Decoder only "attends to" previous words, avoiding future information.
- **Encoder-Decoder Attention:** Focuses on the relevant parts of the Encoder's output.
- **Feed-Forward Neural Network (FFN)**

3.1 Attention Mechanism

Attention Mechanism is a technique used in machine learning and artificial intelligence to improve the performance of models by focusing on relevant information. It allows models to attend to different specific parts of the input data, assigning varying degrees of weight to different elements based on their relevance to the current task.

The attention mechanism typically involves three key components:

- **Query:** Represents the current context or focus of the model.
- **Key:** Represents the elements or features of the input data.
- **Value:** Represents the values associated with the elements or features.

The attention mechanism computes the attention weights by measuring the similarity between the query and the keys. The values are then weighted by the attention weights and combined to produce the final output of the attention mechanism.

3.2 Self-Attention (Scaled-dot Attention)

In comparison with the general attention mechanism, self-attention is a more general mechanism for capturing relationships within a sequence. Self-attention (also called intra-attention) is an attention mechanism relating different positions of a single sequence in order to compute a representation of the sequence.

3.3 Multi-Head Attention

In the Transformer architecture, multi-head attention is used in both the encoder and decoder, but with a key difference in the decoder: the use of masked multi-head attention.

3.3.0.1 Multi-Head Attention

Multi-Head Attention is a key component of the Transformer architecture, enabling the model to focus on different parts of the input sequence simultaneously. It extends the basic scaled dot-product attention mechanism by applying it multiple times in parallel, with each "head" learning to capture distinct aspects of the input.

Instead of using just one self-attention mechanism, multi-head attention divides the initial dataset into N heads, allowing multiple self-attention mechanisms to run in parallel. The idea

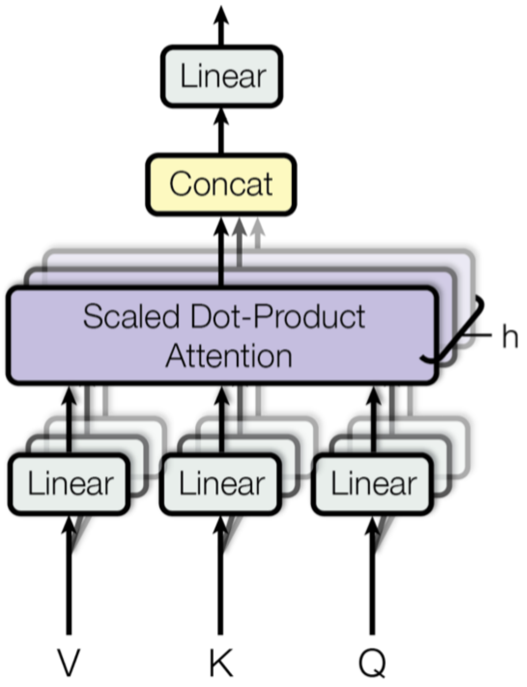


Figure 3: *Multihead Attention*

behind this approach is to let each head independently explore local information, which helps the model generalize better and "collect more diverse information." This enables the Transformer to capture a broader range of dependencies and relationships within the input data, enhancing its overall learning capacity.

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_h)W^O$$

where

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

The projections are parameter matrices:

$$W_i^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}, \quad W_i^K \in \mathbb{R}^{d_{\text{model}} \times d_k}, \quad W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_v}, \quad W^O \in \mathbb{R}^{hd_v \times d_{\text{model}}}$$

In the original paper, they employ $h = 8$ parallel attention layers, or heads. For each of these, we use:

$$d_k = d_v = \frac{d_{\text{model}}}{h}$$

Due to the reduced dimension of each head, the total computational cost is similar to that of single-head attention with full dimensionality.

- **Divide into Heads:** Split the input embedding into NUM_HEADS heads, where each head has a dimension of NUM_TOKENS $\times \frac{\text{DIM}}{\text{NUM\_HEADS}}$.
- **Initialize Weight Matrices:** Define and initialize three weight matrices: W^Q, W^K, and W^V with dimensions DIM $\times \frac{\text{DIM}}{\text{NUM\_HEADS}}$.
- **Compute Q, K, and V:** Obtain Q, K, and V by multiplying the input embedding with their respective weight matrices, then calculate the attention scores.

3.3.0.2 Masked Multi-Head Attention

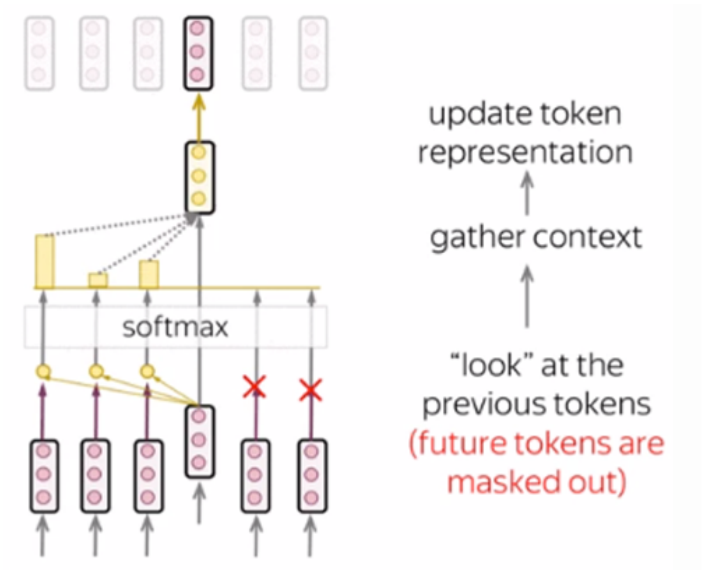

update token representation
gather context
softmax
"look" at the previous tokens
(future tokens are masked out)

Figure 4: *Masked Self-Attention*

In the decoder, masked multi-head attention is introduced to enforce the autoregressive property. The masking ensures that, during training and inference, the decoder cannot "see" future tokens in the sequence it is generating.

This is critical for tasks like text generation (e.g., machine translation), where the model generates the output sequence one token at a time.

The mask is applied to the attention scores before the softmax operation.

For each position i, only tokens from positions 1 to i are visible. All future token positions are masked by setting their scores to $-\infty$, effectively assigning them a probability of zero in the softmax.

To apply this, we create **a causal mask matrix** (a square matrix of size NUM_TOKENS $\times$ NUM_TOKENS), ensuring that each token only attends to preceding tokens by assigning a value of $-\infty$ to future positions. Next, we calculate the scaled dot-product attention, apply the causal mask matrix, and compute the softmax.

3.4 Cross Attention

While self-attention captures relationships between elements within a single input sequence, cross-attention captures relationships between elements of two different input sequences.

Cross-attention computes attention scores based on Query (Q), Key (K) and Value (V) vectors.

In cross-attention, these vectors are derived from different sequences: Q from the target sequence (decoder input) and K and V from the source sequence (encoder output). In machine translation problems, it ensures the translation output attends to the original sentence.

3.5 Input Embedding

Input embeddings are vector representations of discrete tokens such as words, sub-words, or characters. These vectors capture the semantic meaning of tokens, enabling the model to understand and manipulate textual data effectively.

The input embedding layer in the Transformer architecture is the first step in processing input sequences. Its purpose is to convert discrete tokens into continuous vector representations that the model can process.

The role of the input embedding sub-layer is to map input tokens into a high-dimensional space, typically with $d_{\text{model}} = 512$.

3.5.0.1 Word Embedding

Each token in the input sequence is mapped to a fixed-dimensional dense vector using a pre-learned embedding matrix. These embeddings capture the semantic meaning of tokens and allow the model to generalize across similar words.

Given a vocabulary size N and an embedding dimension d_{model}, the embedding matrix $W \in \mathbb{R}^{N \times d_{\text{model}}}$ is used to transform token indices into vectors.

3.5.0.2 Positional Embedding

Since the Transformer model lacks sequential inductive bias like RNNs, positional embeddings are added to retain the order of words in a sentence. These embeddings represent the relative or absolute position of tokens in a sequence.

Positional encodings have the same dimension d_{model} as word embeddings, allowing them to be summed:

$$\text{Input Representation} = \text{Word Embedding} + \text{Positional Embedding}$$

The positional encodings use sine and cosine functions of different frequencies:

$$\text{PE}_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right), \quad \text{PE}_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right)$$

where *pos* is the position and i is the dimension index.

3.6 Feed Forward Network

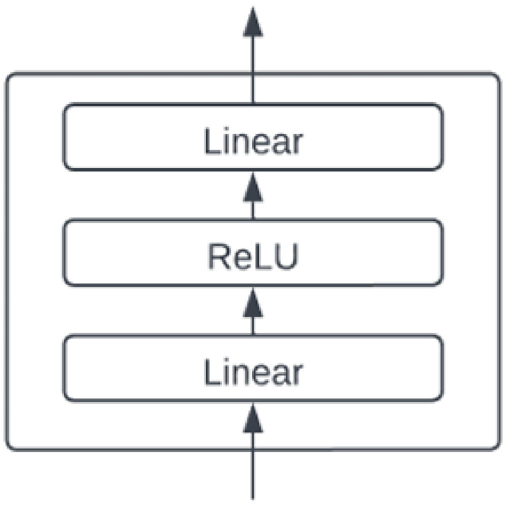


Figure 5: *Feed Forward Network*

After the attention mechanism computes weighted context for each token, a feed-forward layer is applied independently to each position. It consists of two linear transformations with a ReLU activation in between.

In the encoder, after attention:

1. The first fully connected layer projects the input to a higher-dimensional space using weight matrix W_1 and bias b_1.
2. A ReLU activation function is applied.
3. The second fully connected layer projects the output back to d_{model} using weight matrix W_2 and bias b_2.

Mathematically, the Feed Forward Network (FFN) is defined as:

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

where x is the input vector at a particular position.

3.7 Add & Norm

The Add & Norm layer normalizes the output of either the Multi-Head Attention or the Feed Forward sub-layers. This component improves training stability and model convergence by:

- **Add:** Applying residual connections by adding the layer input back to the layer output.
  - After the attention layer: Attention Output + Input
  - After the feed forward layer: FFN Output + Input
- **Norm:** Applying Layer Normalization to the resulting sum.

This combination of residual connection and layer normalization helps to prevent vanishing gradients, maintain consistent distribution of activations, and make optimization more efficient.

4 Finetune techniques

Large Language Models (LLMs), such as GPT, BERT, and their derivatives, have shown remarkable success in a wide range of natural language processing (NLP) tasks. These models, pre-trained on massive corpora, can generalize well to downstream tasks with fine-tuning. However, the sheer size of these models—often with billions of parameters—poses significant challenges in terms of computational resources, time, and energy consumption when performing full fine-tuning. As a result, the research community has turned to Parameter-Efficient Fine-Tuning (PEFT) techniques to adapt LLMs to specific tasks more efficiently.

4.1 Standard fine-tuning

Standard fine-tuning refers to the process of updating all the parameters of a pre-trained language model, including both the base transformer layers and any task-specific heads, for a downstream task. This approach contrasts with parameter-efficient fine-tuning (PEFT) methods that aim to adapt only a small number of additional parameters while keeping the base model frozen.

In standard fine-tuning, the entire model is treated as trainable. For instance, in a classification task using a BERT-based model, the classification head (typically a feedforward layer) and the transformer encoder are both updated during training. This allows the model to better specialize to the target task at the cost of increased computational resources and risk of overfitting, especially when the training data is limited.

This strategy was widely adopted after the success of models such as BERT [3], where pre-training on large corpora followed by full fine-tuning on a downstream task demonstrated state-of-the-art performance across multiple benchmarks. However, full fine-tuning requires storing a copy of the entire model for each task, which can be impractical for resource-constrained settings.

In the context of large language models like T5, standard fine-tuning remains a strong baseline due to its ability to leverage the full model capacity. It is particularly effective when the downstream dataset is large enough to avoid overfitting and justify the cost of training all parameters. Moreover, it enables the model to learn task-specific patterns deeply, especially in tasks that deviate significantly from the pre-training objective (e.g., abstractive summarization or complex question answering).

Advantages:

- **Maximum Adaptability:** Updates all model weights, allowing deep specialization to the target task.
- **High Performance:** Often achieves strong or state-of-the-art results, especially on large or complex datasets.
- **No Architectural Changes Needed:** Directly fine-tunes the pre-trained model without the need to introduce additional modules or techniques.

Disadvantages:

- **High Computational Cost:** Requires significantly more memory and compute resources for training and inference.
- **Storage Inefficiency:** A separate copy of the model must be stored for each downstream task.
- **Overfitting Risk:** Especially likely when training on small datasets, due to the large number of tunable parameters.
- **Slow Adaptation:** Training all parameters from scratch for each task can be slow and inefficient in multi-task or few-shot settings.

4.2 LoRA (Low-Rank Adaptation)

LoRA (Low-Rank Adaptation) is a parameter-efficient fine-tuning technique designed to adapt large pre-trained models to downstream tasks without updating all of their parameters.

Instead of modifying the original weight matrices of the model, LoRA introduces a pair of trainable low-rank matrices that are added to the existing weights during training. By decomposing the weight updates into the product of two smaller matrices, LoRA significantly reduces the number of parameters that need to be learned, making the fine-tuning process more memory- and compute-efficient.

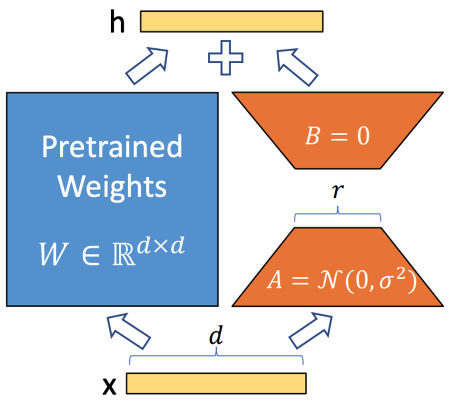
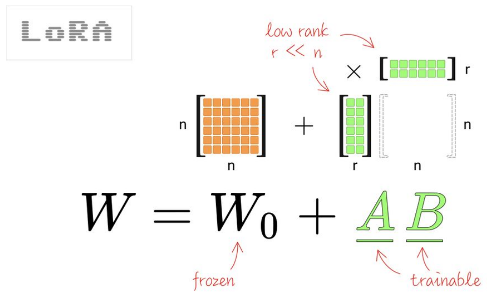

h
Pretrained Weights
$W \in \mathbb{R}^{d\times d}$
$B = 0$
r
$A = \mathcal{N}(0, \sigma^2)$
d
x

Figure 6: *LoRA (Low-Rank Adaptation)*

During training, only these low-rank matrices are updated, while the original model weights remain frozen. This enables rapid adaptation to new tasks with minimal computational overhead and storage requirements.

At inference time, the low-rank updates are merged with the original weights, resulting in a model that performs task-specific predictions without any additional complexity. This approach not only reduces training time and resource usage, but also helps preserve the general knowledge encoded in the original pre-trained model, leading to improved performance especially in low-resource or multi-task learning scenarios.



$$W = W_0 + \underline{A}\,\underline{B}$$

Figure 7: *LoRA (Low-Rank Adaptation)*

In LoRA, **the hyperparameter** r determines the rank of the low-rank matrices A and B that are injected into the weight update process. It controls how much the original weight matrix is decomposed and, consequently, how compact the adaptation will be.

- A smaller r results in fewer trainable parameters, thereby reducing training time and GPU memory usage. This is especially beneficial for large-scale models or resource-constrained environments.
- However, if r is too small, the model may not have enough capacity to effectively learn the task-specific transformations. This can lead to a loss of important information and ultimately degrade model performance.
- Thus, r is a critical hyperparameter that balances efficiency and expressiveness. Proper selection of r is essential to harness the benefits of LoRA without sacrificing accuracy.

Advantages
In addition to introducing no inference latency, as discussed earlier, **LoRA (Low-Rank Adaptation)** offers several other key advantages:

1. **Reduction in Trainable Parameters:** One of the most significant benefits of LoRA is its ability to dramatically reduce the number of trainable parameters. Compared to full fine-tuning of GPT-3 175B with Adam, LoRA can reduce the number of trainable parameters by a factor of **10,000**, and the **GPU memory requirement by 3 times**. This efficiency stems from its low-rank design: LoRA injects trainable rank-decomposition matrices into existing weights, focusing optimization only on the introduced low-rank components.
2. **Low-Level Design Efficiency:** By targeting only low-rank updates to weight matrices, LoRA maintains the original model architecture, making it compatible with a wide range of models and allowing efficient integration without significant architectural modifications or retraining.
3. **Modularity and Reusability:** It allows building multiple lightweight LoRA modules tailored to specific tasks. For example, one can fine-tune a LoRA module on a base LLM for text summarization, and another module on the same base model for question answering. During real-time inference, both modules can share the same frozen backbone model, which needs to be loaded only once. Given the large physical size of modern LLMs (often exceeding 100 GB), this shared architecture significantly reduces memory usage and deployment complexity.

Disadvantages
Despite its many benefits, LoRA also has several limitations to consider:

- **Requires careful hyperparameter tuning:** The performance of LoRA depends significantly on selecting appropriate ranks and scaling factors. Poor choices can lead to suboptimal adaptation or increased computational costs.
- **Moderate increase in model size:** Although efficient, LoRA adds additional adapter parameters, which increases the overall model size compared to purely frozen models.
- **Potential inference overhead:** The inserted low-rank adapter layers may introduce some latency or complexity during inference, depending on implementation and hardware.
- **Performance variability:** LoRA's effectiveness can vary based on the underlying model size and the specific downstream task; it may not always outperform other parameter-efficient fine-tuning methods.
- **Less effective for small models:** For very small models or highly constrained environments, the benefits of LoRA may be limited due to overhead or reduced flexibility.

4.3 AdaLoRA (Adaptive Low-Rank Adaptation)

AdaLoRA (Adaptive Low-Rank Adaptation) [21] is a parameter-efficient fine-tuning method designed to improve upon LoRA by dynamically adjusting how much each part of a pre-trained language model is fine-tuned. Instead of using a fixed low-rank decomposition across all layers like standard LoRA, AdaLoRA learns where and how much adaptation is needed in different layers during training. This adaptive mechanism allows AdaLoRA to allocate model capacity more intelligently, leading to better performance while training fewer parameters.

Advantages:

- ***Adaptive rank allocation***: Dynamically assigns more capacity (higher rank) to important layers and prunes unimportant ones.
- ***Reduced parameter count***: Trains fewer parameters than fixed-rank LoRA while maintaining or improving performance.
- ***Improved efficiency***: Achieves better trade-offs between compute cost and task accuracy.
- ***Better generalization***: By focusing on crucial parts of the model, AdaLoRA may generalize better on unseen data.
- ***Task-specific flexibility***: Allows the fine-tuning process to adjust to the unique structure of each task.

Disadvantages:

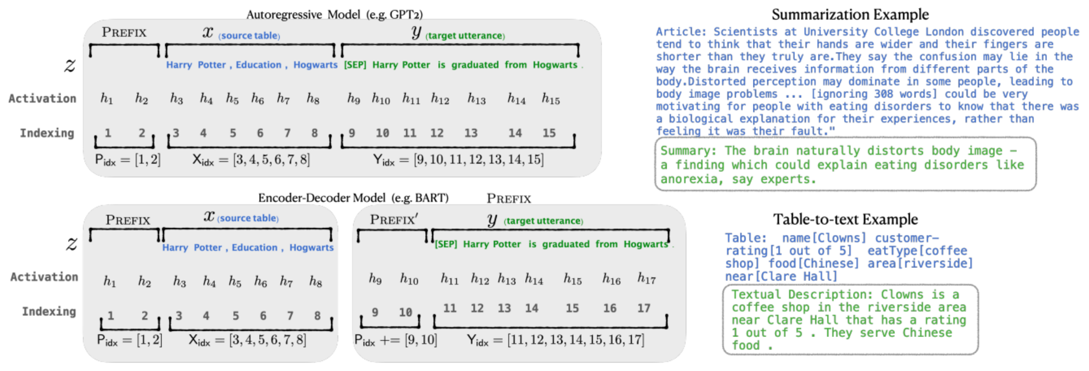
- ***More complex training***: Requires additional mechanisms like sparsity-inducing regularization and rank scheduling.
- ***Harder to implement***: Compared to LoRA, it involves more code changes and param-

eter tuning.

- ***Less transparent***: Dynamic rank pruning makes it harder to interpret which parts of the model are contributing most.
- ***Requires model internals***: Like LoRA, it needs access to and modification of the model's architecture—not ideal for closed-source APIs.

4.4 Prefix tuning

Prefix Tuning is a Parameter-Efficient Fine-Tuning (PEFT) method designed to adapt large pre-trained language models (like GPT-2, GPT-3) to downstream tasks without modifying the model's internal parameters. Instead of fine-tuning all model weights (which can be billions), prefix tuning learns a small task-specific set of vectors called a "***prefix***" that is prepended to the key and value vectors of the attention mechanism at every layer of the transformer.


Autoregressive Model (e.g. GPT2)
PREFIX
x (source table)
y (target utterance)
z
Harry Potter , Education , Hogwarts [SEP] Harry Potter is graduated from Hogwarts .
Activation
Indexing
$P_{idx} = [1,2]$
$X_{idx} = [3,4,5,6,7,8]$
$Y_{idx} = [9,10,11,12,13,14,15]$
Encoder-Decoder Model (e.g. BART)
PREFIX'
$P_{idx} += [9,10]$
$Y_{idx} = [11,12,13,14,15,16,17]$
Summarization Example
Table-to-text Example

Article: Scientists at University College London discovered people tend to think that their hands are wider and their fingers are shorter than they truly are.They say the confusion may lie in the way the brain receives information from different parts of the body.Distorted perception may dominate in some people, leading to body image problems ... [ignoring 308 words] could be very motivating for people with eating disorders to know that there was a biological explanation for their experiences, rather than feeling it was their fault."

Summary: The brain naturally distorts body image – a finding which could explain eating disorders like anorexia, say experts.

Table: name[Clowns] customer-rating[1 out of 5] eatType[coffee shop] food[Chinese] area[riverside] near[Clare Hall]

Textual Description: Clowns is a coffee shop in the riverside area near Clare Hall that has a rating 1 out of 5 . They serve Chinese food .

Figure 8: *Prefix-tuning using autoregressive LM (top) and encoder-decoder model (bottom) [9]*

The prefix vectors are initialized either randomly or from a predetermined starting point. These vectors are new, untrained parameters rather than pre-defined or manually designed elements. The core language model remains unmodified (its weights are frozen), preserving all previously acquired language understanding.

The training process involves:

1. Using a task-specific dataset
2. Training only the prefix vectors
3. Prepending the current prefix (a sequence of embeddings) to the tokenized input
4. Processing the combined input through the model
5. Measuring output quality against targets using a loss function (typically cross-entropy)

The system uses backpropagation to update only the prefix vectors, not the model itself. Through iteration, these vectors are optimized to guide the model toward producing the desired output.

After training completion, the learned prefix becomes fixed. To perform a specific task, the system prepends this trained prefix to the input. The frozen model processes this combined input, and the optimized prefix guides it to generate task-appropriate outputs.

Since the base model remains unchanged, different prefixes can be trained for various tasks and interchanged as needed. This enables a single model to handle multiple tasks by utilizing the appropriate prefix.

Advantanges:

- ***Parameter efficiency***: Trains only a small number of parameters (e.g., ¡1
- ***No model weight change***: Keeps the base model unchanged, which is critical for scenarios where the same model is shared across multiple tasks.
- ***Modularity***: You can easily switch between tasks by swapping the prefix vector — useful in multi-task settings or when serving multiple clients.
- ***Training efficiency***: Since only a small number of parameters are updated, training is much faster and more stable than full fine-tuning.

Disadvantages:

- ***Task-specific overhead***: Requires storing a separate prefix for each task, which might grow large in multi-task or multi-user deployments.
- ***Limited flexibility compared to full fine-tuning***: Can't match full fine-tuning performance in highly complex or very domain-specific tasks where deeper transformations are needed.
- ***Still memory-intensive for very deep models***: Although more efficient than full fine-tuning, prefix tuning still needs to store $2LPd\_model$ parameters.
- ***Requires architectural access***: Unlike prompt tuning which only modifies input tokens, prefix tuning needs low-level access to attention layers—so it's harder to apply in some APIs or closed-source models.

4.5 Prompt tuning

Prompt tuning [7] is a technique designed to enhance the performance of a pre-trained language model without altering its core architecture. Instead of modifying the deep structural weights of the model, prompt tuning adjusts the prompts that guide the model's response. This method is based on the introduction of "soft prompts," a set of tunable parameters inserted at the beginning of the input sequence.

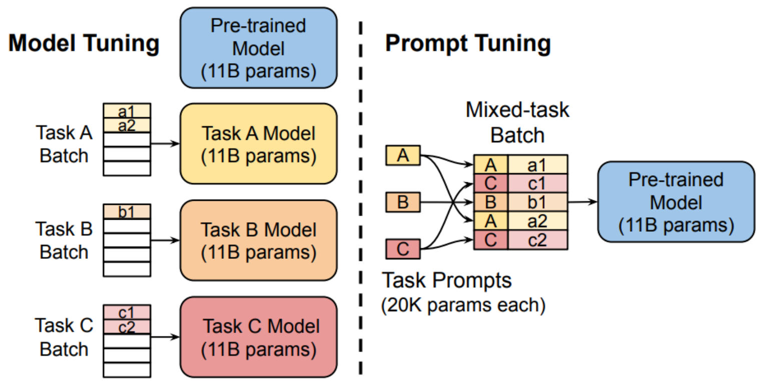

Model Tuning
Pre-trained Model (11B params)
Task A Batch
a1
a2
Task A Model (11B params)
Task B Batch
b1
Task B Model (11B params)
Task C Batch
c1
c2
Task C Model (11B params)
Prompt Tuning
Mixed-task Batch
A
B
C
A a1
C c1
B b1
A a2
C c2
Pre-trained Model (11B params)
Task Prompts (20K params each)

Figure 9: *Compare the traditional approach of model tuning with prompt tuning*

As the observation in figure 9, Model tuning requires making a taskspecific copy of the entire pre-trained model for each downstream task and inference must be performed in separate batches. Prompt tuning only requires storing a small task-specific prompt for each task, and enables mixed-task inference using the original pretrained model.

The workflow of prompt tuning:

1. **Initialization of soft prompts**: Soft prompts are artificially constructed tokens that are added to the model's input sequence. These prompts can be initialized in several ways. Random initialization is common, however they can also be initialized based on certain heuristics. Once initialized, soft prompts are attached to the start of the input data. When the model processes this data, it considers both the soft prompts and the actual input.

2. **Forward Pass and loss evaluation**: The training process is usually similar to that of training a standard deep neural network. It starts with a forward pass where the model processes the combined input through its layers, producing an output. This output is then evaluated against the desired outcome using a loss function, which measures the discrepancy between the model's output and the actual expected value.

3. **Backpropagation**: During backpropagation, the errors are propagated back through the network. However, instead of adjusting the network's weights, we only modify the soft prompt parameters. This process repeats across multiple epochs, with the soft prompts gradually learning to shape the model's processing of inputs in such a way that minimizes the error for the given task.

4. **Iteration**: The process of forward pass, loss evaluation, and backpropagation is repeated over multiple epochs. Over time, these soft prompts learn to shape the input in ways that consistently reduce the loss score, thereby improving the model's task-specific performance without compromising its underlying capabilities.

Advantages:

- ***Resource efficiency***: Prompt tuning maintains the pre-trained model's parameters unchanged, significantly reducing the computational power needed. This efficiency is especially crucial in resource-constrained environments, allowing for sophisticated model use without the high cost. As the average size of foundational models increases, "freezing" model parameters becomes even more appealing since a separate model doesn't need to be deployed for each task.

- ***Rapid deployment***: Unlike comprehensive fine-tuning, prompt tuning requires adjustments only to a small set of soft prompt parameters. This speeds up the adaptation process, enabling quicker transitions between different tasks and reducing downtime.
- ***Model integrity and knowledge retention***: By keeping the core architecture and weights of the model intact, prompt tuning preserves the original capabilities and knowledge embedded in the pre-trained model. This is critical for maintaining the reliability and generalizability of the model across various applications.
- ***Task flexibility***: Prompt tuning facilitates using a single foundational model for multiple tasks by simply changing the soft prompts. This approach reduces the need to train and maintain separate models for each specific task, enhancing scalability and simplicity in model management.
- ***Reduced human involvement***: Prompt tuning requires significantly less human intervention than prompt engineering, where carefully crafting prompts to suit a particular task can be error-prone and time-consuming. Instead, the automated optimization of soft prompts during training minimizes human error and maximizes efficiency.
- ***Competitive performance***: Research indicates that for large models, prompt tuning can achieve performance levels comparable to those of fine tuning. This benefit becomes increasingly significant as model sizes grow, combining high efficiency with strong results.

Disadvantages:

- ***Lower performance on complex tasks***: Prompt tuning may not match the accuracy or robustness of full fine-tuning or techniques like LoRA, especially for tasks that require deep adaptation of model behavior.
- ***Requires large models***: Its effectiveness largely depends on the size of the base model. Prompt tuning often underperforms when used with smaller language models, as they may not have enough internal knowledge to be guided effectively by soft prompts alone.
- ***Uninterpretable prompt vectors***: The learned soft prompts are continuous vectors that don't correspond to actual words, making them hard to understand or debug. This makes prompt tuning less transparent compared to manual or textual prompt engineering.
- ***One prompt per task***: Each task typically requires a separately trained set of soft prompts. This means that while the base model is reused, you still need to manage and store multiple sets of task-specific prompts.
- ***Limited to certain architectures***: Prompt tuning is mainly compatible with models that expose token embeddings (e.g., decoder-only or encoder-decoder Transformers). It's less flexible with black-box models or APIs that don't allow embedding manipulation.

4.6 P-tuning v2

P-Tuning v2 is a parameter-efficient fine-tuning method designed to adapt large language models (LLMs) to downstream tasks while updating only a tiny subset of the model parameters. Instead of modifying the entire model, it learns a small set of trainable prompt embeddings, also known as *soft prompts*, while keeping the original model weights frozen. These soft prompts are not discrete tokens, but continuous vectors that are optimized during training.

Unlike traditional prompt tuning, which appends learned embeddings only at the input layer, P-Tuning v2 improves upon this by injecting soft prompts into every Transformer layer of the model. Specifically, these prompts are inserted into the key and value matrices of the self-attention mechanism in each layer. This deeper integration allows the prompts to influence internal computations, making the method more expressive and effective.

During fine-tuning, only the soft prompts are trained, which significantly reduces memory and computational costs. The rest of the model remains unchanged, making this approach highly efficient and scalable, especially for very large models (e.g., with 100 billion+ parameters). Despite training only a small number of parameters, P-Tuning v2 can achieve performance that is close to full fine-tuning in many NLP tasks such as classification, question answering, and sentiment analysis.

Comparison between P-Tuning v1 and P-Tuning v2

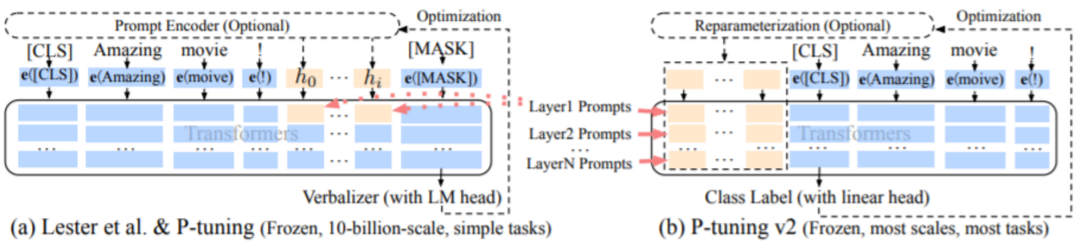

Prompt Encoder (Optional)
Optimization
[CLS]
Amazing
movie
!
[MASK]
e([CLS])
e(Amazing)
e(moive)
e(!)
h_0
h_i
e([MASK])
Transformers
Verbalizer (with LM head)
(a) Lester et al. & P-tuning (Frozen, 10-billion-scale, simple tasks)
Layer1 Prompts
Layer2 Prompts
LayerN Prompts
Reparameterization (Optional)
Optimization
[CLS]
Amazing
movie
!
e([CLS])
e(Amazing)
e(moive)
e(!)
Transformers
Class Label (with linear head)
(b) P-tuning v2 (Frozen, most scales, most tasks)

Figure 10: *Comparison between P-Tuning v1 and P-Tuning v2*

- **Implementation:**
  - **P-Tuning v1:** Adds trainable soft prompts only at the *input layer* of the model, adjusting just the input embeddings of the Transformer.
  - **P-Tuning v2:** Injects soft prompts into *every Transformer layer*, specifically into the key and value matrices of the self-attention mechanism in each layer, enabling deeper influence on internal computations.
- **Expressiveness and Effectiveness:**
  - **P-Tuning v1:** Effective in learning downstream tasks but limited by prompt insertion only at the input.
  - **P-Tuning v2:** More expressive and powerful due to deeper integration within the model, improving performance close to full fine-tuning.

- **Stability and Optimization:**
  - **P-Tuning v1:** May face optimization challenges and instability during training.
  - **P-Tuning v2:** More stable and easier to optimize, especially in low-resource scenarios (limited data or computation).
- **Resource Efficiency:**
  - Both methods train only a small subset of parameters (soft prompts) without changing the original model weights, significantly reducing memory and computational costs compared to full fine-tuning.
- **Applications:**
  - **P-Tuning v2** is particularly suitable for efficiently adapting the same large base model to multiple tasks by swapping different prompt embeddings, especially for very large models (100 billion+ parameters).
- **Advantages**
  - Highly parameter-efficient, only fine-tunes a small prompt vector.
  - Competitive performance, close to or better than full fine-tuning on many tasks.
  - Lower computational cost, requires less memory and training time.
  - Easily applicable to various pretrained models.
  - Prompts can be reused and shared independently from the base model.
- **Disadvantages**
  - Works best with large models; less effective for smaller ones.
  - Sensitive to prompt length and initialization.
  - Learned prompts are hard to interpret semantically.
  - Not always superior to other parameter-efficient methods (e.g., Adapter, LoRA).

In addition to its strong performance, P-Tuning v2 has proven to be more stable and easier to optimize compared to earlier prompt-based methods like P-Tuning v1. It is particularly beneficial in low-resource scenarios where training data or computing power is limited, and is well-suited for scenarios where the same base model needs to be adapted efficiently to a variety of tasks by switching out different prompt embeddings.

4.7 Freeze fine-tuning

Freeze fine-tuning is a technique in which certain layers of a pre-trained language model—typically the lower or middle layers—are kept frozen (i.e., their parameters are not updated), while only a subset of layers, usually the top layers or task-specific heads, are fine-tuned on the downstream task. This approach serves as a middle ground between standard fine-tuning and parameter-efficient methods, aiming to reduce computational cost while maintaining strong performance.

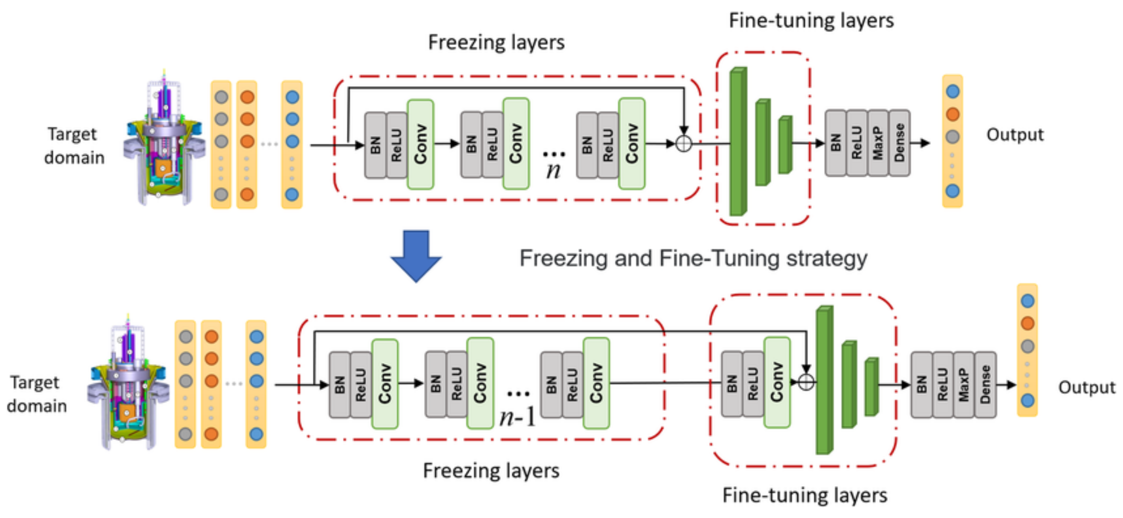


Figure 11: *Freeze fine-tuning*

The rationale behind this method is that lower layers of large transformer models tend to learn general language representations that are transferable across tasks. Therefore, updating only the higher layers allows the model to adapt to task-specific features without significantly altering the core language understanding capabilities. This is particularly beneficial in scenarios with limited training data or compute resources.

Freeze fine-tuning is often used as a baseline or starting point when experimenting with large models such as T5 or BERT. For example, researchers may freeze the encoder and fine-tune only the decoder for sequence generation tasks, or freeze all layers except for the last few transformer blocks and the output head.

Advantages:

- **Reduced Training Cost:** Only a subset of parameters are updated, lowering memory usage and speeding up training.
- **Lower Risk of Overfitting:** By not modifying foundational language layers, the model retains strong generalization capabilities.
- **Simplicity:** Does not require architectural modifications or external components like adapters or prompts.
- **Good Trade-Off:** Offers a balance between full fine-tuning performance and parameter efficiency.

Disadvantages:

- **Suboptimal Adaptation:** May not capture all task-specific nuances if the frozen layers contain limiting inductive biases.
- **Manual Layer Selection:** Requires heuristic or empirical decisions about which layers to freeze and which to fine-tune.
- **Less Flexible Than PEFT:** Compared to LoRA or prompt tuning, it may not scale as well across many tasks or domains.

4.8 Classifier-head Fine-tuning

Classifier-head fine-tuning is an approach in which only the final classification layer (task-specific head) is trained, while the entire backbone of the pre-trained language model remains frozen. This method assumes that the underlying model already encodes rich semantic representations and that a lightweight linear classifier is sufficient to map these features to task-specific outputs.

This technique is especially useful in scenarios with extremely limited computational resources or where very fast experimentation is needed. It is also commonly used as a baseline to assess the effectiveness of more sophisticated adaptation techniques. However, it typically underperforms compared to methods that allow deeper model adaptation, particularly on tasks that require strong contextual understanding or domain-specific reasoning.

In implementation, classifier-head fine-tuning involves adding a task-specific layer (e.g., a linear layer for classification or tagging) on top of the frozen model and updating only its parameters during training. The pre-trained backbone acts as a static feature extractor throughout.

Advantages:

- **Extremely Low Cost:** Only a small number of parameters are updated, drastically reducing training time and memory usage.
- **Simple and Fast Baseline:** Provides a quick benchmark for task feasibility using pre-trained features.
- **Prevents Catastrophic Forgetting:** The base model's general knowledge remains intact since it is not modified.

Disadvantages:

- **Limited Expressiveness:** The classifier head may be too shallow to capture complex task-specific patterns or dependencies.
- **Poor Performance on Complex Tasks:** Tasks requiring deep contextual understanding often suffer due to lack of adaptation in the encoder.
- **Insensitive to Domain Shifts:** Since the base model is frozen, it cannot adapt to new domains or specialized vocabularies.

4.9 Adapter Fine-tuning

Adapter fine-tuning is a parameter-efficient transfer learning method that introduces small, trainable neural modules—called *adapters*—into each layer of a pre-trained Transformer model [13]. Instead of updating all the parameters of the large backbone model, only the adapter modules and task-specific output head are fine-tuned, while the rest of the model is kept frozen. This significantly reduces the number of trainable parameters, making the approach more efficient and modular. Each adapter consists of a small bottleneck architecture (typically a

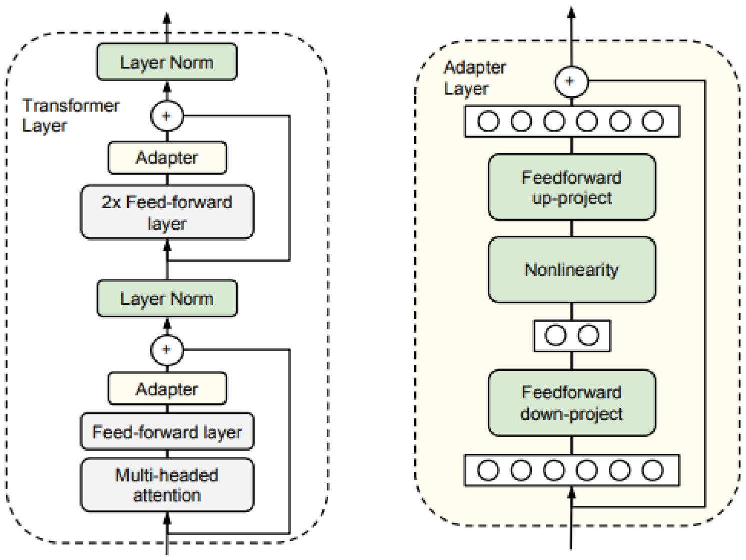


Figure 12: *Architecture of the adapter module and its integration with the Transformer*

down-projection, nonlinearity, and an up-projection), allowing for minimal computational overhead. During inference or multi-task scenarios, adapters can be dynamically loaded and swapped, enabling task-specific specialization without duplicating the entire model.

This technique is especially useful in scenarios where storage, compute, or deployment constraints make full fine-tuning impractical. Moreover, adapter modules promote modularity and reusability across tasks, which is valuable in continual and multitask learning.

Advantages:

- **Efficiency:** Only a small fraction (typically ¡3%) of parameters are trained.
- **Modularity:** Task-specific adapters can be saved and reused without altering the base model.
- **Scalability:** Suitable for multi-task and continual learning with low storage costs.

Disadvantages:

- **Increased Latency:** Additional forward passes through adapter layers may slightly increase inference time.

- **Architecture Modification:** Requires structural changes to the Transformer architecture.
- **Not Always Optimal:** May underperform compared to full fine-tuning in highly specialized or data-rich tasks.

5 Chosen LLM and Tasks

This section introduces the architecture, training objectives, and core applications of two key Transformer-based large language models (LLMs): BERT and GPT. These models represent distinct design choices optimized for different types of natural language processing (NLP) tasks.

5.1 BERT

Bidirectional Encoder Representations from Transformers (BERT), proposed by Devlin et al. [3], is an LLM built using only the encoder component of the Transformer architecture [18]. This encoder-based design allows BERT to process input text bidirectionally, incorporating context from both preceding and following tokens when encoding each word.

BERT is pre-trained using two self-supervised learning objectives:

- **Masked Language Modeling (MLM):** Random tokens (about 15%) in the input are masked, and the model is trained to predict them using the surrounding context. This encourages the model to learn deep bidirectional representations.
- **Next Sentence Prediction (NSP):** The model is given two sentences and must predict whether the second sentence logically follows the first. This task helps BERT learn relationships between sentences.

BERT can be adapted for various downstream tasks using task-specific layers:

- **Sequence Classification:** A classification layer is added on top of the `[CLS]` token to perform tasks like sentiment analysis.
- **Token Classification:** A token-level classifier is used for tasks such as Named Entity Recognition (NER), applying labels to each token.
- **Sentence Similarity:** Sentence embeddings from BERT can be compared using cosine similarity to measure semantic closeness.

5.2 GPT

Generative Pre-trained Transformer (GPT), introduced by OpenAI, follows a different architectural strategy. Unlike BERT, GPT uses only the decoder portion of the Transformer model [14]. This decoder-only architecture is autoregressive, meaning it generates text by predicting the next token based only on previous tokens in the sequence.

GPT uses masked self-attention to prevent the model from accessing future tokens during training. This ensures that each token is predicted using only the past context.

GPT models are pre-trained on large unlabelled text corpora using a language modeling objective: predicting the next word in a sequence. This process enables the model to learn grammar, facts, and general language understanding. Fine-tuning on smaller labeled datasets allows GPT to perform a range of tasks, including summarization, question answering, and dialogue generation.

In this assignment, we apply GPT to three different NLP tasks using three distinct PEFT fine-tuning techniques and compare them to evaluate the performance and adaptation capabilities

- **Sentiment Analysis**: is a key text classification task that involves determining the emotional tone expressed in a given text. In this task, we use **LoRA**, **Prompt-tuning**, and **Classifier Head**.
- **Text Generation**: model is required to generate coherent and contextually appropriate text based on an input prompt. It assesses the model's language modeling and creative text construction abilities. We use **LoRA**, **Prefix-tuning**, and **Adapter**.
- **Named Entity Recognition (NER)**: is a fundamental task in natural language processing (NLP) that involves identifying and classifying key elements in text into predefined categories such as names of people (PER), organizations (ORG), locations (LOC), dates, and other entities. In this task, we use **LoRA**, **AdaLoRA** and **Prefix tuning**.

5.3 T5 (Text-to-Text Transfer Transformer)

Text-to-Text Transfer Transformer (T5), introduced by Google Research, presents a unified framework for a wide variety of natural language processing tasks [16]. Unlike BERT and GPT, which utilize only the encoder or decoder respectively, T5 employs the full encoder-decoder Transformer architecture, aligning with the original design of the Transformer.

T5's most innovative contribution is its adoption of a **text-to-text paradigm**, where all NLP tasks are framed as text generation problems. Whether the task is translation, summarization, question answering, or classification, the input and output are treated as text strings. Task-specific prefixes (e.g., `"summarize:"`, `"translate English to French:"`, `"question: context:"`) are prepended to the input, guiding the model to interpret and execute the task correctly. This unifying format allows T5 to use the same architecture, loss function, and hyperparameters across diverse tasks, streamlining the training process.

The model is pre-trained on a combination of unsupervised and supervised tasks. A core unsupervised pre-training objective is **span corruption**, a denoising task in which spans of contiguous tokens are replaced with sentinel tokens. The model is trained to generate the missing spans in the correct order, encouraging it to learn deep contextual representations and long-range dependencies.

T5 Model Variants and Parameter Sizes

Model Variant	**Number of Parameters**	**Description**
T5-Small	∼60 million	Lightweight version, suitable for resource-constrained environments and fast experimentation.
T5-Base	∼220 million	Balanced size offering strong performance on a wide range of tasks.
T5-Large	∼770 million	Large-scale model with enhanced capacity for complex tasks and higher accuracy.

Table 1: *Summary of T5 model variants and their parameter sizes.*

In addition to unsupervised training, T5 is fine-tuned on **multiple supervised NLP tasks**, all reformulated in the text-to-text schema. This includes tasks such as sentiment analysis, natural language inference, and question answering. The uniformity in task formulation allows T5 to demonstrate strong performance across a wide spectrum of NLP benchmarks while simplifying the task interface.

In this work, we focus on the **T5-small** variant for fine-tuning experiments due to its efficiency and accessibility, especially for rapid prototyping and low-resource settings.

We apply the T5 model to three different NLP tasks, each with its own distinct characteristics and fine-tuning strategies. For each task, we explore and compare suitable fine-tuning methods to evaluate performance and adaptation capabilities.

- **Summarization**: This task requires the model to generate a concise and coherent summary from a longer input text. It emphasizes content selection, abstraction, and fluency. We fine-tune the T5 model using three methods: **Standard fine-tuning**, **LoRA** (Low-Rank Adaptation), and **Freeze tuning** to assess how well each approach enables the model to capture key ideas from lengthy documents.

- **Sentiment Analysis**: This classification task involves determining the sentiment polarity (positive, negative, or neutral) of a given text. For this task as well, we apply **Standard fine-tuning**, **LoRA**, and **Freeze tuning** to investigate how these methods impact the model's ability to learn task-specific features with varying parameter efficiency.

- **Question Answering (QA)**: This task involves generating an accurate answer based on a given question and contextual passage. It requires the model to understand and reason over the input. Unlike the other two tasks, we evaluate three different fine-tuning techniques for QA: **Freeze tuning**, **LoRA**, and **Prefix Tuning v2 (PTv2)**. These approaches focus on parameter-efficient adaptation and task-specific prompting.

This setup allows us to analyze and compare the effectiveness of different fine-tuning techniques across a range of language understanding and generation tasks.
These experiments allow us to compare performance, generalization, and parameter efficiency across tasks with varied linguistic demands.

6 Experimental results & Insights

6.1 BERT

6.1.1 Sentiment Analysis

We evaluate three fine-tuning strategies for emotion classification using the `BERT` model on the multi-class Emotions dataset from Hugging Face. This dataset contains short English texts labeled with one of six emotion categories: `joy`, `sadness`, `anger`, `fear`, `love`, and `surprise`. The goal is to classify each utterance into the correct emotional category.

6.1.1.1 The Emotions Dataset

This dataset consists of thousands of English-language utterances collected from social media, annotated with six discrete emotion labels. It is widely used for benchmarking multi-class emotion classification systems in NLP.

6.1.1.2 Data Preprocessing

We use the `BertTokenizer` from the Hugging Face Transformers library to tokenize the text. Sequences are padded and truncated to a fixed maximum length. The input includes token IDs, attention masks, and corresponding emotion labels. Preprocessing ensures compatibility with BERT's input format.

6.1.1.3 Evaluation Metric

The model performance is assessed using the following metrics:

- **Accuracy**: Proportion of correctly predicted labels.
- **F1 Score**: Harmonic mean of precision and recall.

6.1.1.4 Fine-tuning Techniques & Settings

We compare three different fine-tuning strategies:

- **Full Fine-tuning:** All BERT parameters and classifier head are trainable.
- **Classifier Head Only:** Only the final classification layer is trained, while all BERT layers are frozen.
- **LoRA:** Injects trainable low-rank adapters into attention layers of BERT; the base model is frozen.

Training Setup: All models were trained using the same configuration:

- Batch size: 16 (for both training and evaluation)
- Best model selected based on: `eval_f1`
- Optimizer: AdamW

- Early stopping: patience of 3 epochs

6.1.1.5 Performance

Table 2: *Performance Comparison of BERT Fine-tuning Strategies on the Emotions Dataset*

Strategy	Trainable Params	Training Time (s)	Accuracy	F1 Score
Full Fine-tuning	109,486,854	1099.90	**0.9260**	**0.9264**
Classifier Only	4,614	2679.10	0.4775	0.3716
LoRA	451,596	2858.53	0.9220	0.9232

- **Full fine-tuning** achieves the highest F1 score and accuracy, showing strong performance when the entire model is optimized.
- **LoRA** achieves nearly identical results while training only 0.41% of the parameters, offering an excellent trade-off between efficiency and performance.
- **Classifier-only tuning** performs poorly, with much lower accuracy and F1 score despite longer training. This indicates that deeper model adaptation is required for effective emotion classification on this dataset.

Overall, both full fine-tuning and LoRA are effective for sentiment classification using BERT, with LoRA being the more efficient option.

6.1.2 Named Entity Recognition (NER)

We evaluate three fine-tuning strategies for Named Entity Recognition using the `BERT` model on the CoNLL-2003 dataset from Hugging Face. The goal is to identify and classify named entities in text, such as persons, organizations, locations, and miscellaneous entities.

6.1.2.1 The CoNLL-2003 Dataset

This dataset consists of English newswire articles with annotated entities in the IOB format. It includes four entity types: `PER`, `ORG`, `LOC`, and `MISC`. It is a standard benchmark for sequence labeling tasks in NLP.

6.1.2.2 Data Preprocessing

Although the dataset is already tokenized into words, the `BertTokenizer` from Hugging Face splits words into subwords. To maintain proper alignment between input tokens and entity labels, we propagate the original token's label to all its subword components. Specifically, we assign the label of the first subword to all subsequent subwords.

6.1.2.3 Evaluation Metric

Model performance is assessed using entity-level metrics provided by the `seqeval` library. These metrics focus on the correct identification and classification of entire named entities, rather than individual tokens. Specifically, we report:

- **Accuracy**: Overall proportion of correctly predicted entity tags.
- **F1 Score**: Harmonic mean of precision and recall across all entity types.

Metric Computation: During evaluation, only the first subword token of each original word is considered when computing metrics. This avoids double-counting and ensures that predictions align with the original CoNLL-style annotations.

6.1.2.4 Fine-tuning Techniques & Settings

We compare the following strategies:

- **Full Fine-tuning:** All BERT and classifier weights are updated.
- **Classifier Head Only:** Only the top classification layer is trained, with BERT frozen.
- **LoRA:** Trainable adapters are inserted into BERT's attention layers, while the base model remains frozen.

Training Setup: All models use the following configuration:

- Batch size: 16 (training and evaluation)
- Optimizer: AdamW
- Epochs: Up to 30 with early stopping (patience = 3)
- Evaluation metric: `eval_f1`
- Logging and saving per epoch

6.1.2.5 Performance

Table 3: *Performance Comparison of BERT Fine-tuning Strategies on the CoNLL-2003 Dataset*

Strategy	Trainable Params	Training acTime (s)	Accuracy	F1 Score
Full Fine-tuning	107,726,601	1212.65	**0.9706**	**0.8988**
Classifier Only	6,921	2201.11	0.9170	0.6565
LoRA	456,210	2609.15	0.9660	0.8788

- **Full fine-tuning** delivers the best overall performance, achieving the highest accuracy and F1 score.
- **LoRA** provides a strong trade-off, reaching near-optimal F1 with only 0.42% of trainable parameters.
- **Classifier-only tuning** is computationally light but substantially underperforms on entity recognition tasks.

In summary, full fine-tuning is most effective for NER out of the three fine-tune techniques, but LoRA is a competitive alternative when resource efficiency is a priority.

6.1.3 Sentence Similarity

The dataset used in this study is the **Semantic Textual Similarity Benchmark (STS-B)** from the GLUE benchmark suite. This dataset consists of pairs of sentences annotated with a similarity score ranging from 0 to 5, where 0 indicates no semantic similarity and 5 indicates semantic equivalence. It is commonly used for evaluating models on their ability to understand and quantify sentence similarity.

Given the nature of the Semantic Textual Similarity Benchmark (STS-B) dataset—where fine-grained, continuous similarity judgments are required—choosing the right fine-tuning strategy for BERT is critical. BERT's deep architecture captures layered linguistic knowledge, and how this knowledge is updated during training directly affects both performance and generalization. The three techniques explored in this work were selected to strike different balances between training efficiency, representational stability, and adaptation to task-specific nuances. Standard full fine-tuning provides a performance baseline, Layer-wise Learning Rate Decay (LLRD) is known for improving convergence and preserving useful pre-trained features, and freezing lower layers reduces computational cost while leveraging the general-purpose language understanding already present in the model.

- **A. Standard Full Fine-Tuning**: In this baseline approach, all parameters of the BERT model, including its embedding and encoder layers, are fine-tuned on the STS-B dataset. This strategy allows maximum flexibility for the model to adapt to the task but is computationally demanding and prone to overfitting on small datasets.

- **B. Layer-wise Learning Rate Decay (LLRD)**: This technique applies different learning rates to different layers of the BERT model, with lower learning rates assigned to bottom layers and higher rates to top layers. The rationale is that lower layers capture general language knowledge and require smaller updates, while higher layers can be more aggressively fine-tuned for task-specific information. LLRD helps stabilize training and improve generalization.

- **C. Freezing Lower BERT Layers** In this method, the bottom N (in this experiment: $N = 6$) layers of the BERT model are frozen (not updated during training), while only the top layers and the task-specific head are fine-tuned. This reduces computational cost and may preserve useful general linguistic features learned in earlier layers.

Technique	Pearson Correlation	Training Time	Remarks
Standard	~0.87	High	Best performance
LLRD	~0.86	High	Stable training
Freeze Lower Layers	~0.85	Moderate	Efficient, lower accuracy

Table 4: *Comparison of fine-tuning strategies on the STS-B dataset.*

The experimental results demonstrate the following:

- **Standard full fine-tuning** achieves the highest performance but requires the most resources and is more sensitive to overfitting.

- **LLRD fine-tuning** offers nearly equivalent performance with improved training stability and better generalization.

- **Freezing lower layers** provides a computationally efficient alternative, with slightly reduced accuracy, suitable for resource-constrained settings.

Overall, **Standard full fine-tuning** remains the best choice when computational resources are not a limitation.

6.1.4 Performance Analysis Across Tasks

Technique	**Sentiment Analysis**	**NER**	**Sentence Similarity**	**Training Efficiency**
Full Fine-tuning	**Best**	**Best**	**Best**	Low
LoRA	Near-best	Near-best	–	Moderate
LLRD	–	–	Good	Moderate
Freeze Lower Layers	–	–	Fair	High
Classifier Head Only	Poor	Poor	–	Low

Table 5: *Comparison of Fine-tuning Techniques on BERT Across NLP Tasks*

Overall remark:

Across the three evaluated tasks using BERT, full fine-tuning consistently delivers the highest performance in both classification and sequence labeling, though at the cost of high training time and compute resources. LoRA demonstrates strong performance close to full fine-tuning, particularly in sentiment analysis and NER, while updating fewer than 0.5% of the parameters—making it a compelling choice for resource-efficient training.

For the sentence similarity task, techniques such as LLRD and freezing lower layers provide decent accuracy with reduced training demands, striking a balance between performance and efficiency. However, classifier-only fine-tuning fails to adequately adapt BERT for both token-level and classification tasks, showing significant drops in accuracy and F1 scores despite minimal training overhead.

In conclusion, LoRA is a highly effective and efficient fine-tuning approach for BERT, while layer freezing or LLRD may be considered when moderate performance suffices. Full fine-tuning remains the gold standard when maximum task performance is required, particularly in well-resourced environments.

6.2 GPT

6.2.1 Sentiment Analysis

We evaluate three fine-tuning strategies for emotion classification using the `GPT-2` model on the multi-class Emotions dataset. This dataset includes six emotion categories: `joy`, `sadness`, `anger`, `fear`, `love`, and `surprise`. Each approach adapts the GPT-2 language model for classification, either through architectural modifications or prompt-based conditioning.

The Emotions Dataset consists of short English utterances labeled with one of the six emotions. It provides a challenging benchmark for evaluating generative models adapted to classification tasks. The dataset is moderately sized, suitable for experiments on medium-scale GPUs like the P100.

6.2.1.1 Data Preprocessing

The dataset is tokenized using the GPT-2 tokenizer. Inputs are either the raw text (for prompt tuning and LoRA) or transformed representations (for classifier head training). Padding and truncation are applied as needed. For prompt tuning and LoRA, prompts or adapters are prepended or injected into the input, respectively. Labels are encoded as class indices for evaluation.

6.2.1.2 Evaluation Metric

We use two metrics:

- **Accuracy**: Measures the proportion of correct predictions.
- **Macro F1 Score**: Computes the harmonic mean of precision and recall across all classes, giving equal weight to each emotion category.

Rationale: Since the dataset is balanced across classes, macro F1 provides a robust measure of model performance on underrepresented emotions, while accuracy offers an overall performance view.

6.2.1.3 Fine-tune techniques & Settings

We compare three strategies:

- **LoRA (Low-Rank Adaptation)**: Injects trainable low-rank matrices into GPT-2's attention layers.
- **Prompt Tuning**: Learns task-specific prompt embeddings while keeping the base model frozen.
- **Classifier Head**: Adds a trainable classification head on top of the GPT-2 encoder and fine-tunes the entire model.

Training Setup:

- **Epochs:** All three methods use AdamW optimizer.
  - LoRA: 5 epochs
  - Prompt Tuning: 15 epochs
  - Classifier Head: 5 epochs
- **Batch size:** All three methods use batch size 16.
- **Parameter freezing strategy:**
  - **LoRA:** Base model frozen, trainable low-rank matrices are inserted into the attention layers and optimized during training.
  - **Prompt-tuning:** Base model frozen, learnable prompt embeddings is prepended to the input and trained.
  - **Classifier Head:** A classification layer is added on top of frozen GPT-2.

6.2.1.4 Performance

Table 6: *Performance of GPT-2 with Different Fine-tuning Techniques on the Emotions Dataset*

Model	**Epochs**	**Accuracy**	**F1 Score (Macro)**
Basic GPT-2 (no fine-tuning)	–	0.0000	0.0000
LoRA	5	**1.0000**	**1.0000**
Prompt Tuning	15	**1.0000**	**1.0000**
Classifier Head	5	0.9295	0.9300

- **LoRA**: Achieved perfect accuracy and macro F1 in just 5 epochs, indicating highly efficient adaptation using minimal trainable parameters. Its rapid convergence and strong performance make it ideal for low-resource settings. However, such high scores may signal overfitting, especially in the absence of a more diverse evaluation set.

- **Prompt Tuning**: Also reached perfect scores, though it required 15 epochs to converge. By freezing the base model and learning only soft prompt embeddings, it offers a lightweight and modular approach. This makes it suitable for scenarios with model access restrictions or when multiple tasks are deployed over a shared base model.

- **Classifier Head**: Achieved strong but slightly lower results (92.95% accuracy, 0.93 F1) within 5 epochs. It involves training more parameters than LoRA or Prompt Tuning, which may lead to better generalization on unseen data but also increases the risk of overfitting and computational cost.

Overall, all three fine-tuning techniques significantly outperform the baseline (unmodified GPT-2). LoRA and Prompt Tuning demonstrate remarkable task adaptation with high efficiency, while the Classifier Head approach remains a solid and generalizable baseline.

All three methods significantly outperform the baseline (unmodified GPT-2), with LoRA and Prompt Tuning showing exceptional effectiveness for this task.

6.2.2 Name Entity Recognition (NER)

We evaluate three Parameter-Efficient Fine-Tuning (PEFT) techniques for the Named Entity Recognition (NER) task using the GPT-2 model family on the CoNLL-2003 dataset. The objective is to identify and classify named entities such as persons (PER), organizations (ORG), locations (LOC), and miscellaneous (MISC) within raw text.

6.2.2.1 The CoNLL-2003 Dataset

The CoNLL-2003 dataset is a benchmark corpus for sequence labeling in English newswire text. Each sentence is annotated in the Inside–Outside–Beginning (IOB) format with four types of named entities:

- PER: Person names
- ORG: Organizations
- LOC: Locations
- MISC: Miscellaneous entities

We used the pre-tokenized form of the dataset and mapped each token to its corresponding label, ensuring correct alignment between the tokenizer outputs and the ground truth labels, including support for virtual tokens used in prompt-based PEFT methods.

6.2.2.2 Data Preprocessing

We used the Hugging Face `datasets` library to load the CoNLL-2003 dataset and tokenized it using the GPT-2 tokenizer. To accommodate prompt-based methods like Prefix and Prompt Tuning, we introduced a configurable number of virtual tokens (`num_virtual_tokens`). The label alignment ensures that only the first subword token of each word receives a label, while others are masked with `-100` to avoid contributing to the loss.

We padded sequences to a maximum length of `128` tokens and added handling for virtual tokens during preprocessing. The labels were aligned appropriately for each token sequence to maintain consistency in training across all methods.

6.2.2.3 Model Setup

We used distilgpt2 as the base model for all experiments. We applied three PEFT strategies using the peft library:

- LoRA: Low-Rank Adaptation with rank `r=16`, targeting the attention (`c_attn`) and feed-forward (`c_fc`) modules.
- AdaLoRA: An adaptive version of LoRA that dynamically adjusts the rank during training. It starts with `r=8` and targets r=16.
- Prefix Tuning: Introduces a fixed set of virtual tokens prepended to the input sequence. We used 20 virtual tokens with `encoder_hidden_size=768`.

Each method was configured for the `TOKEN_CLS` (token classification) task and adapted to GPT-2's architecture.

6.2.2.4 Training

All models were trained using the Hugging Face `Trainer` API with the following configuration:

- Learning rate: 5e-5 (LoRA), 1e-3 (Prefix Tuning), and adjusted as needed
- Batch size: 8 (train), 16 (eval)
- Epochs: 30
- Weight decay: 0.1
- Early stopping after 3 epochs with no improvement in F1 score

We used the `seqeval` metric for evaluating sequence labeling, reporting precision, recall, F1 score, and accuracy. During training, the best model (based on F1) was automatically loaded.

Strategy	Trainable Params	Accuracy	F1 Score
LoRA	670,473	0.9579	0.7319
AdaLoRA	338,793	**0.9591**	**0.7409**
Prefix tuning	191,241	0.9576	0.7264

Table 7: *Performance Comparison of GPT2 Fine-tuning Strategies on the CoNLL-2003 Dataset*

6.2.2.5 Evaluation

- F1 Score: AdaLoRA (0.7409) outperforms LoRA (0.7319) and Prefix tuning (0.7264), indicating better handling of entity recognition challenges, such as class imbalance or rare entities.
- Accuracy: All methods achieve high accuracy (¿0.957), with AdaLoRA slightly leading (0.9591). The small differences suggest that all models are effective for general token classification, but F1 score differences highlight varying abilities to handle entity-specific challenges.
- Efficiency: LoRA and AdaLoRA are parameter-efficient fine-tuning methods, making them more computationally efficient than full fine-tuning. Prefix tuning, while also efficient, appears less effective for NER based on the F1 score.

To enhance GPT-2's performance on the NER task, the following strategies are recommended: Hyperparameter optimization, Perform additional preprocessing, Model architecture enhancements, Training on larger dataset.

6.2.3 Text Generation

We fine-tune the `GPT-2` model for text generation using the **Alpaca dataset**. The goal is to generate coherent and instruction-following responses given a prompt. GPT-2, a generative language model developed by OpenAI, is well-suited for auto-regressive generation tasks and has been widely adopted for applications requiring free-form output generation.

The Alpaca dataset is an instruction-following dataset that provides prompt-response pairs. It is designed to simulate real-world natural language instructions, making it an ideal choice for evaluating generative models in instruction tuning tasks. The dataset helps benchmark how well the model can generate aligned and relevant responses to diverse prompts.

6.2.3.1 Data Preprocessing

Each prompt-response pair is preprocessed into a single input string by concatenating the instruction and response with a task-specific format (e.g., `"Instruction: <prompt> Response: <answer>"`). The entire sequence is tokenized using the GPT-2 tokenizer. During training, causal language modeling is applied by shifting the inputs so that the model predicts the next token. Padding is applied where necessary, and attention masks are used to ignore padded positions.

6.2.3.2 Evaluation Metric

We use `perplexity` as the primary evaluation metric. Perplexity is computed using the negative log-likelihood of the predicted tokens. A lower perplexity indicates better predictive performance and model fluency.

Rationale: Since GPT-2 is a generative language model trained with a causal language modeling objective, perplexity is the most suitable metric for measuring the model's ability to generate fluent and coherent responses.

6.2.3.3 Fine-tune techniques & Settings

We evaluate three fine-tuning approaches:

- **LoRA (Low-Rank Adaptation)**: Low-rank trainable matrices are added to attention layers while keeping the base model frozen.
- **Prefix Tuning**: Prepends trainable continuous prefix vectors to each layer's input; only these vectors are updated.
- **Adapter:** Inserts small bottleneck modules in transformer layers; only adapter parameters are trained.

Training Setup:

- **Common settings:** - 10 epochs, learning rate: 5e-5. - Evaluation after every epoch. - Early stopping with patience of 3 epochs. - Best model is selected based on lowest perplexity on validation set.
- **Batch size:** - Uniform batch size of 8 for all techniques due to memory constraints of GPT-2.
- **Precision:** - Mixed-precision training (fp16) enabled when GPU is available.

6.2.3.4 Performance

Table 8: *Perplexity of GPT-2 on Alpaca Dataset Using Different Fine-tuning Methods*

Fine-tuning Method	**Perplexity**
Before Fine-tuning	1375.56
LoRA	1373.36
Prefix Tuning	**375.32**
Adapter	1487.18

- **Prefix Tuning** achieves the best performance with a significant drop in perplexity to 375.32. Prefix-tuning was originally designed for tasks like generation, translation, and QA, and instruction-following fits this mold:
  - Prefix acts as a soft prompt to steer generation.
  - On Alpaca-like data, where instructions have patterns, prefix vectors may capture useful signal.
- **LoRA** yields slightly better performance than the base model, demonstrating modest gains with high efficiency.
- **Adapter Tuning** underperforms in this setup, possibly due to overfitting or suboptimal adapter configuration. Adapter tuning often needs more careful architecture choice.

These results suggest that **Prefix Tuning** is the most effective approach in this context, offering a strong balance between efficiency and performance. However, this doesn't mean that this method has better generation since prefix-Tuning can improve perplexity by adding predictable structure.

6.2.4 Performance Analysis Across Tasks

Technique	**Sentiment Analysis**	**NER**	**Text Generation**	**Training Efficiency**
LoRA	Best	Good	Moderate	High
Prompt/Prefix Tuning	Best	Fair	**Best**	Very High
Adapter	–	–	Poor	Medium
Classifier Head	Good	–	–	Low

Table 9: *Comparison of Fine-tuning Techniques on GPT-2 Across NLP Tasks*

Overall remark: Across all tasks using GPT-2, Prompt/Prefix Tuning shows excellent performance for generation tasks like instruction-following (Alpaca) while being highly parameter-efficient. LoRA emerges as a strong general-purpose technique, excelling in classification tasks such as sentiment analysis and maintaining competitive results in NER and generation. Although the Classifier Head is effective in sentiment classification, its training cost and limited applicability to generative or token-level tasks reduce its overall utility. Adapter Tuning underperforms in this setting, possibly due to suboptimal configuration or dataset mismatch. In summary, PEFT methods like LoRA and Prefix Tuning offer attractive trade-offs between performance and efficiency for adapting GPT-2 across diverse NLP tasks.

6.3 T5

6.3.1 Text Summarization

The dataset used for this task is the **CNN/DailyMail** summarization dataset. It consists of news articles paired with human-written summaries. The task is to generate concise and informative summaries from longer article inputs. This dataset is a standard benchmark for evaluating abstractive summarization models and emphasizes the generation of grammatically coherent and factually consistent summaries.

The summarization task requires the model to generalize well to diverse news content while efficiently learning to produce high-quality outputs. The **T5-small** model is a lightweight encoder-decoder transformer, well-suited for this task, particularly when resources are limited. The three fine-tuning techniques explored were chosen to investigate trade-offs between performance, training efficiency, and parameter efficiency:

- **Prompt tuning** attaches trainable prompts to guide the model's generation without modifying the base parameters. Prompt tuning involves prepending a set of trainable tokens (soft prompts) to the input, while keeping the rest of the model frozen. This method is parameter-efficient, making it ideal for low-resource scenarios or when multiple tasks are served by a single model backbone. However, it may not always match full fine-tuning performance on complex generative tasks like summarization.

- **Layer freezing** selectively prevents updates to earlier layers, preserving general linguistic knowledge while fine-tuning task-specific components. In this technique, the lower layers of the T5 model are frozen during training, and only the higher (more task-specific) layers are updated. This reduces training time and memory usage while allowing the model to specialize its higher-level representations for summarization.

- **LoRA (Low-Rank Adaptation)** introduces low-rank trainable adapters into the attention weights, enabling efficient fine-tuning with fewer trainable parameters. LoRA adds low-rank matrices into the self-attention modules of the model, allowing only these adapters to be updated during training. This approach dramatically reduces the number of trainable parameters while maintaining or exceeding the performance of full fine-tuning in many cases. It is especially effective in resource-constrained or multi-task settings.

Technique	**ROUGE-1**	**ROUGE-2**	**ROUGE-L**	**Training Time**
Prompt Tuning	39.4	17.4	27.2	Low
Layer Freezing	43.6	20.1	30.5	Higher
LoRA	41.9	19.4	29.1	Moderate

Table 10: *Comparison of fine-tuning techniques on CNN/DailyMail using T5-small.*

The summarization results highlight key trade-offs among the fine-tuning strategies:

- **Prompt tuning** is the most parameter-efficient approach, requiring minimal memory and training time, though it yields slightly lower performance.

- **Layer freezing** offers a good compromise, maintaining strong performance while reducing training cost.

- **LoRA** achieves the best overall ROUGE scores while significantly reducing the number of trainable parameters, making it a compelling option for scalable and modular fine-tuning.

In conclusion, **LoRA** emerges as the most effective strategy for abstractive summarization with T5-small on CNN/DailyMail, offering a balance between efficiency and output quality. However, **prompt tuning** may still be preferable when ultra-lightweight deployment is prioritized, or **layer freezing** can be chosen if the resources are available.

6.3.2 Sentiment Analysis

We explore the effectiveness of fine-tuning the `t5-small` model for sentiment analysis using the IMDB dataset.
T5-small is a lightweight variant of the Text-to-Text Transfer Transformer (T5) model proposed by Raffel et al. It retains the text-to-text formulation of the original T5, enabling it to handle various NLP tasks by framing them as generation problems. Due to its reduced number of parameters, T5-small offers a more computationally efficient alternative, making it suitable for low-resource settings. In this task, we cast sentiment classification as a generation task, where the model takes a movie review as input and generates a sentiment label, either `positive` or `negative`.

The IMDB dataset is a widely-used benchmark for binary sentiment classification. It consists of 50,000 movie reviews, evenly split into 25,000 training and 25,000 test examples, with an equal distribution of positive and negative labels. The dataset provides a challenging and balanced setting for evaluating language models. In this experiment, each review is formatted as input text, and the model is expected to generate either `positive` or `negative`, following the text-to-text paradigm.

6.3.2.1 Data Preprocessing

Each movie review is prefixed with `"sentiment:"` and tokenized as input, while the label (`positive` or `negative`) is tokenized as the target. Padding and truncation are applied to both inputs and targets. Padding tokens in the labels are replaced with `-100` to be ignored during loss computation.

6.3.2.2 Evaluation Metric

: We use `accuracy` as the primary evaluation metric, which measures the proportion of correct predictions. Model outputs and reference labels are decoded from token IDs and mapped to binary values (`positive` = 1, `negative` = 0). Predictions outside the valid label set are excluded.

Rationale: Accuracy is well-suited for the balanced binary classification task on the IMDB dataset, providing a clear and interpretable measure of overall performance.

6.3.2.3 Fine-tune techniques & Settings

We compare three different fine-tuning strategies:

- **Standard Fine-tuning**: Updates all parameters of the model during training.
- **Freeze Tuning**: Freezes most of the model parameters and updates only the final classification head.
- **LoRA (Low-Rank Adaptation)**: Adds trainable low-rank matrices to the attention layers, enabling parameter-efficient fine-tuning.

Training Setup:

- **Common settings:** - 10 epochs, learning rate 5e-5, evaluation and checkpoint saving at each epoch. - Early stopping callback with patience 3 epochs. - Load best model at the end based on accuracy metric. - Mixed precision (fp16) enabled if GPU is available.
- **Batch size:** - Standard and LoRA fine-tuning use batch size 32. - Freeze fine-tuning uses batch size 32 as well (can be adjusted if needed).
- **Beam Search during evaluation:** - Beam width set to 5 with early stopping, max generation length 10 tokens. - Used in a custom `BeamSearchTrainer` class overriding the prediction step to generate outputs with beam search instead of greedy decoding. - Improves prediction quality and accuracy by exploring multiple candidate outputs.
- **Parameter freezing strategy:**
  - **Standard**: All model parameters updated.
  - **LoRA**: Base model frozen, train low-rank adapters in attention layers.
  - **Freeze**: Encoder and shared embeddings frozen, only decoder head updated.

6.3.2.4 Performance

Table 11: *Evaluation Results of T5-small with Different Fine-tuning Techniques on Sentiment Analysis with IMDB dataset*

Fine-tuning Method	**Accuracy (%)**	**Eval Loss**
Baseline (e.g., traditional model)	88.00	–
Standard	**92.94**	**0.1009**
Freeze	91.45	0.1087
LoRA	90.26	0.1231

We evaluate the T5-small model on a sentiment analysis task using three fine-tuning strategies. Below is the summary of the results compared to the **baseline accuracy (e.g., pretrained or traditional method)**.

- **Standard Fine-tuning** (Accuracy: **92.94%**, Loss: **0.1009**):
  - Achieved the best performance among all methods.
  - Outperforms the baseline accuracy (e.g., **baseline**) by a significant margin.
  - However, it is the most resource-intensive approach, requiring full model updates.
- **Freeze** (Accuracy: **91.45%**, Loss: **0.1087**):
  - Slightly lower accuracy than Standard, but still competitive.
  - More efficient in terms of training cost, since some parameters are frozen.
  - A good compromise if resources are limited or for faster convergence.
- **LoRA (Low-Rank Adaptation)** (Accuracy: **90.26%**, Loss: **0.1231**):
  - Provides the lowest performance among the three techniques.
  - Still comparable to or better than baseline accuracy in many scenarios.
  - Very lightweight and suitable for low-resource or multi-task settings.

Standard fine-tuning yields the highest accuracy, but at the cost of computational efficiency. Freeze and LoRA offer strong trade-offs, with LoRA being the most efficient, though with slightly reduced performance.

6.3.3 Question Answering (QA)

We fine-tune a **T5-small** model to perform extractive **Question Answering (QA)** using the **SQuAD v1.1** dataset. The QA task is formulated in the text-to-text paradigm, where the input is a concatenation of the question and its corresponding context, and the output is the answer span directly extracted from the context.

SQuAD v1.1 (Stanford Question Answering Dataset) is a widely-used benchmark for evaluating reading comprehension systems. It contains over 100,000 question-answer pairs created by crowdworkers on a set of Wikipedia articles. Each answer is a contiguous span of text found within the given context paragraph, making it a suitable benchmark for extractive QA models.

T5-small is a scaled-down version of **the Text-to-Text Transfer Transformer (T5)** with approximately **60 million parameters**. It is designed to perform a wide range of NLP tasks by framing them as text generation problems. Due to its efficiency and flexibility, T5-small is a practical choice for fine-tuning in environments with limited computational resources.

6.3.3.1 Data Preprocessing

The SQuAD v1.1 dataset consists of *questions*, corresponding *contexts* (passages), and *answers*. The preprocessing steps to prepare the data for a sequence-to-sequence model are as follows:

1. **Combine Question and Context:** Each example is converted into a single input string by concatenating the question and its context in the format:

   `"question: <question> context: <context>"`

   This allows the model to attend to both question and context simultaneously.

2. **Select Target Answer:** For each example, only the first answer text from the `answers` field is selected as the target output.

3. **Tokenization:**

   - The input strings are tokenized with truncation and padding to a maximum length (e.g., 512 tokens).

   - The target answers are tokenized separately, also with truncation and padding to a shorter max length (e.g., 64 tokens).

4. **Label Preparation:** Padding tokens in the target sequences are replaced with -100 so that the loss function ignores these positions during training.

5. **Output Formatting:** The final preprocessed data includes:

   `input_ids, attention_mask, labels, decoder_attention_mask`

   ready for input into the seq2seq model for training.

6.3.3.2 Fine-tune techniques & Settings

In this QA setting, we evaluate how effectively T5-small can learn to extract precise answer spans from contextual information using three different fine-tuning strategies:

- **LoRA (Low-Rank Adaptation)**: Injects trainable low-rank matrices into attention layers, enabling efficient fine-tuning with frozen base model and significantly fewer trainable parameters.

- **P-Tuning v2**: Extends prompt tuning by inserting deep virtual prompt tokens across multiple layers, while freezing the original model weights.

- **Freeze tuning**: The base model weights remain frozen and only newly introduced prompt or adapter parameters are trained.

Training Setup:

- **Common settings**: 10 epochs, learning rate 3e-4, eval and save each epoch, load best model, logging enabled.
- **Batch size**: LoRA uses larger batch size (32); P-Tuning v2 and Freeze use smaller batch sizes (16).
- **fp16** enabled for LoRA if GPU available; disabled for P-Tuning v2 and Freeze (usually on CPU).
- **Beam Search**: Beam search with **width 4** improves accuracy by considering multiple likely output sequences during generation, reducing the chance of suboptimal predictions compared to greedy decoding.
- **Early stopping callback with patience** of 2 to 3 epochs used in all.

6.3.3.3 Evaluation Metric

The `compute_metrics` function uses the SQuAD evaluation metric to measure model performance on question answering. This metric calculates **Exact Match (EM)** and **F1 scores**, which are standard for SQuAD tasks. EM checks for exact answer matches, while F1 measures token overlap between predicted and true answers, providing a balanced assessment of accuracy and partial correctness.

6.3.3.4 Performance

These methods aim to balance performance and parameter efficiency, enabling effective fine-tuning without updating the entire model.

Fine-tune Method	Eval Loss	Eval Exact Match	Eval F1
P-Tuning v2	0.4916	54.06%	68.53%
Freeze	0.4432	61.21%	75.77%
LoRA	**0.4098**	**62.32%**	**76.46%**

Table 12: *Evaluation metrics of different fine-tuning methods on SQuAD v1.1 using T5-small.*

Among the three fine-tuning methods, **LoRA** outperforms both **P-Tuning v2** and **Freeze Tuning**, achieving the lowest evaluation loss (0.4098), and the highest Exact Match (62.32%) and F1 score (76.46%). In contrast, **P-Tuning v2** results in the weakest performance, while **Freeze Tuning** shows noticeable improvement over P-Tuning v2.

While these results are *not state-of-the-art* compared to larger models (e.g., BERT-base or T5-base), they are **reasonable for a lightweight model like T5-small** (approximately 60M parameters). LoRA demonstrates clear advantages in both learning efficiency and performance for extractive QA on the SQuAD v1.1 dataset, making it a promising approach in low-resource or fast-adaptation settings.

6.3.4 Performance Analysis Across Tasks

Technique	Summarization	Sentiment Analysis	QA	Training Efficiency
Layer Freezing	Best	Good	Good	High Cost
LoRA	Good	Moderate	Best	Efficient
Prompt Tuning	Lowest	—	Lowest	Very Efficient
Standard	—	Best	—	Full Fine-tuning

Table 13: *Comparison of Fine-tuning Techniques on T5-small Across NLP Tasks*

Overall remark: Across all tasks with T5 small, we observe that standard fine-tuning consistently produces the best performance, but at the cost of high resource usage. Lightweight methods such as LoRA and Layer Freezing offer promising trade-offs, especially for tasks like summarization and QA, while prompt tuning remains insufficient for tasks requiring deeper language understanding or generation.

7 Conclusions

This report presented an overview of **Transformer models**, their rapid scaling into large language models (LLMs), and the development of various fine-tuning techniques. Through a series of experiments on multiple NLP tasks - including text summarization, sentiment analysis, question answering, and others—we evaluated fine-tuning methods such as **Layer Freezing**, **LoRA**, and **Prompt Tuning** using the **T5-small** model.

The results demonstrate that no single fine-tuning method is universally optimal: while **full fine-tuning** and **Layer Freezing** offer high performance, methods like **LoRA** strike a strong balance between efficiency and accuracy. **Prompt-based methods**, although highly parameter-efficient, generally underperform on complex tasks.

Overall, the choice of fine-tuning strategy should be guided by **task type**, **performance requirements**, and **computational constraints**.

These insights underscore the flexibility of Transformer-based models and emphasize the importance of tailoring fine-tuning techniques to specific application needs in modern NLP.

References

[1] Sidney Black and et al. Gpt-neox-20b: An open-source autoregressive language model. *EleutherAI*, 2022.

[2] Tom Brown and et al. Language models are few-shot learners. *Advances in Neural Information Processing Systems*, 2020.

[3] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *NAACL*, 2019.

[4] Nan Du and et al. Glam: Efficient scaling of language models with mixture-of-experts. *arXiv preprint arXiv:2112.06905*, 2022.

[5] Jordan Hoffmann and et al. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 2022.

[6] Jeremy Howard and Sebastian Ruder. Universal language model fine-tuning for text classification. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 328–339, 2018.

[7] Brian Lester, Rami Al-Rfou, and Noah Constant. The power of scale for parameter-efficient prompt tuning. *arXiv preprint arXiv:2104.08691*, 2021.

[8] Jiapeng Li, Aixin Sun, Jialong Han, and Chenliang Li. A survey on named entity recognition: Tasks, challenges and opportunities. *Information Fusion*, 63:195–211, 2020.

[9] Xiang Lisa Li and Percy Liang. Prefix-tuning: Optimizing continuous prompts for generation. *arXiv preprint arXiv:2101.00190*, 2021.

[10] Xiao Liu, Kaixuan Ji, Yicheng Fu, Denis Tam, Zhengxiao Du, Zhilin Yang, and Jie Tang. P-tuning v2: Prompt tuning can be comparable to fine-tuning universally across scales and tasks. *arXiv preprint arXiv:2110.07602*, 2021.

[11] Eric Mitchell, Kevin Lin, Zi Lin Wu, Caleb Behnke, Chelsea Finn, and Christopher D. Manning. Model editing by standard fine-tuning. *arXiv preprint arXiv:2202.05262*, 2022.

[12] Rewon Nakano and et al. Webgpt: Browser-assisted question-answering with human feedback. *arXiv preprint arXiv:2112.09332*, 2021.

[13] Jonas Pfeiffer, Andreas Rücklé, Clifton Poth, Akhilesh Kamath, Ivan Vulić, Sebastian Ruder, and Iryna Gurevych. Adapterhub: A framework for adapting transformers. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 46–55. Association for Computational Linguistics, 2020.

[14] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. Technical report, OpenAI, 2018.

[15] Jack Rae and et al. Scaling language models: Methods, analysis & insights from training gopher. *arXiv preprint arXiv:2112.11446*, 2021.

[16] Colin Raffel and et al. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 2020.

[17] Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*, pages 3982–3992, 2019.

[18] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[19] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. Transformers: State-of-the-art natural language processing. *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, 2020.

[20] Linting Xue and et al. mt5: A massively multilingual pre-trained text-to-text transformer. *NAACL*, 2021.

[21] Qingru Zhang, Minshuo Chen, Alexander Bukharin, Nikos Karampatziakis, Pengcheng He, Yu Cheng, Weizhu Chen, and Tuo Zhao. Adalora: Adaptive budget allocation for parameter-efficient fine-tuning. *arXiv preprint arXiv:2303.10512*, 2023.

[22] Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q Weinberger, and Yoav Artzi. Bertscore: Evaluating text generation with bert. *arXiv preprint arXiv:1904.09675*, 2019.