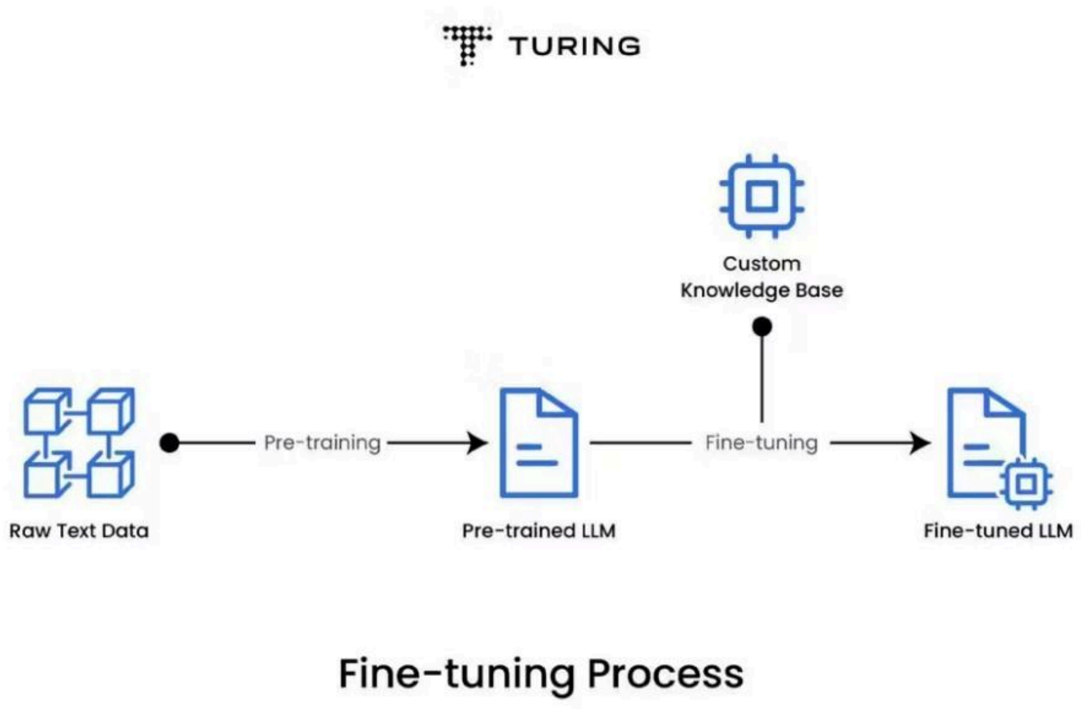
Fine tuning working principle

Fine-tuning allows users to adapt pre-trained LLMs to more specialized tasks. By fine-tuning a model on a small dataset of task-specific data, you can improve its performance on that task while preserving its general language knowledge. For example, a Google study found that fine-tuning a pre-trained LLM for sentiment analysis improved its accuracy by 10 percent.



Fine-tuning is the process of adjusting the parameters of a pre-trained large language model to a specific task or domain. Although pre-trained language models like GPT possess vast language knowledge, they lack specialization in specific areas. Fine-tuning addresses this limitation by allowing the model to learn from domain-specific data to make it more accurate and effective for targeted applications.

1. customization

By fine-tuning a pre-trained LLM, you can customize it to better understand these unique aspects and generate content specific to your domain. This approach allows you to tailor the model's responses to align with your specific requirements, ensuring that it produces accurate and contextually relevant outputs.

Whether it’s legal documents, medical reports, business analytics, or internal company data, LLMs offer nuanced expertise in these domains when trained on specialized datasets. Customization through fine-tuning empowers you to leverage the power of LLMs while maintaining the accuracy necessary for your specific use case.

Fine tuning process

Fine-tuning a pre-trained model for your specific use case or application requires a well-defined process to ensure an optimized outcome.

1.Data Preparation

Data preparation involves curating and preprocessing the dataset to ensure its relevance and quality for the specific task. This may include tasks such as cleaning the data, handling missing values, and formatting the text to align with the model's input requirements.
Additionally, data augmentation techniques can be employed to expand the training dataset and improve the model's robustness. Proper data preparation is essential for fine-tuning as it directly impacts the model's ability to learn and generalize effectively, ultimately leading to improved performance and accuracy in generating task-specific outputs.

2. Choosing the Right pre-trained model

It’s crucial to select a pre-trained model that aligns with the specific requirements of the target task or domain. Understanding the architecture, input/output specifications, and layers of the pre-trained model is essential for seamless integration into the fine-tuning workflow.
Factors such as the model size, training data, and performance on relevant tasks should be considered when making this choice. By selecting a pre-trained model that closely matches the characteristics of the target task, you can streamline the fine-tuning process and maximize the model's adaptability and effectiveness for the intended application.

3. Identifying the right parameters for fine tuning

Configuring the fine-tuning parameters is crucial for achieving optimal performance in the fine-tuning process. Parameters such as the learning rate, number of training epochs, and batch size play a significant role in determining how the model adapts to the new task-specific data. Additionally, selectively freezing certain layers (typically the earlier ones) while training the final layers is a common practice to prevent overfitting.
By freezing early layers, the model retains the general knowledge gained during pre-training while allowing the final layers to adapt specifically to the new task. This approach helps maintain the model's ability to generalize while ensuring that it learns task-specific features effectively, striking a balance between leveraging pre-existing knowledge and adapting to the new task.

4.Validation

Validation involves evaluating a fine-tuned model’s performance using a validation set. Monitoring metrics such as accuracy, loss, precision, and recall provide insights into the model's effectiveness and generalization capabilities.
By assessing these metrics, you can gauge how well the fine-tuned model is performing on the task-specific data and identify potential areas for improvement. This validation process allows for the refinement of fine-tuning parameters and model architecture, ultimately leading to an optimized model that excels in generating accurate outputs for the intended application.

5.Model iteration

Model iteration allows you to refine the model based on evaluation results. Upon assessing the model's performance, adjustments to fine-tuning parameters, such as learning rate, batch size, or the extent of layer freezing, can be made to enhance the model's effectiveness.
Additionally, exploring different strategies, such as employing regularization techniques or adjusting the model architecture, enables you to improve the model's performance iteratively. This empowers engineers to fine-tune the model in a targeted manner, gradually refining its capabilities until the desired level of performance is achieved.

6.Model deployment

Model deployment marks the transition from development to practical application, and it involves the integration of the fine-tuned model into the specific environment. This process encompasses considerations such as the hardware and software requirements of the deployment environment and model integration into existing systems or applications.
Additionally, aspects like scalability, real-time performance, and security measures must be addressed to ensure a seamless and reliable deployment. By successfully deploying the fine-tuned model into the specific environment, you can leverage its enhanced capabilities to address real-world challenges.

2. Data Compliance

In many industries, such as healthcare, finance, and law, strict regulations govern the use and handling of sensitive information. Organizations can ensure their model adheres to data compliance standards by fine-tuning the LLM on proprietary or regulated data.
This process allows for the development of LLMs trained specifically on in-house or industry-specific data, mitigating the risk of exposing sensitive information to external models while enhancing the security and privacy of your data.

3.Limited Labeled data

In many real-world scenarios, obtaining large quantities of labeled data for a specific task or domain can be challenging and costly. Fine-tuning allows organizations to leverage pre-existing labeled data more effectively by adapting a pre-trained LLM to the available labeled dataset, maximizing its utility and performance.

Primary fine tuning Approaches

Fine-tuning involves adjusting LLM parameters, and the scale of this adjustment depends on the specific task that you want to fulfill.

1.Feature Extraction

Feature extraction, also known as repurposing, is a primary approach to fine-tuning LLMs. In this method, the pre-trained LLM is treated as a fixed feature extractor. The model, having been trained on a vast dataset, has already learned significant language features that can be repurposed for the specific task at hand.

The final layers of the model are then trained on the task-specific data while the rest of the model remains frozen. This approach leverages the rich representations learned by the LLM and adapts them to the specific task, offering a cost-effective and efficient way to fine-tune LLMs.

2.Full Fine tuning

Full fine-tuning is another primary approach to fine-tuning LLMs for specific purposes. Unlike feature extraction, where only the final layers are adjusted, full fine-tuning involves training the entire model on the task-specific data. This means all the model layers are adjusted during the training process.
This approach is particularly beneficial when the task-specific dataset is large and significantly different from the pre-training data. By allowing the whole model to learn from the task-specific data, full fine-tuning can lead to a more profound adaptation of the model to the specific task, potentially resulting in superior performance. It is worth noting that full fine-tuning requires more computational resources and time compared to feature extraction.

Fine tuning Methods

There are several fine-tuning methods and techniques used to adjust the model parameters to a given requirement.

1.Supervised Fine tuning

2.Reinforcement Learning from Human Feedback

Supervised Fine tuning

The model is trained on a task-specific labeled dataset, where each input data point is associated with a correct answer or label. The model learns to adjust its parameters to predict these labels as accurately as possible. This process guides the model to apply its pre-existing knowledge, gained from pre-training on a large dataset, to the specific task at hand. Supervised fine-tuning can significantly improve the model's performance on the task, making it an effective and efficient method for customizing LLMs.

Basic Hyperparameter tuning

Basic hyperparameter tuning is a simple approach that involves manually adjusting the model hyperparameters, such as the learning rate, batch size, and the number of epochs, until you achieve the desired performance.
The goal is to find the set of hyperparameters that allows the model to learn most effectively from the data, balancing the trade-off between learning speed and the risk of overfitting. Optimal hyperparameters can significantly enhance the model's performance on the specific task.

Transfer Learning

Transfer LearningTransfer learning is a powerful technique that's particularly beneficial when dealing with limited task-specific data. In this approach, a model pre-trained on a large, general dataset is used as a starting point.

The model is then fine-tuned on the task-specific data, allowing it to adapt its pre-existing knowledge to the new task. This process significantly reduces the amount of data and training time required and often leads to superior performance compared to training a model from scratch.

Multi Tasking Learning

In multi-task learning, the model is fine-tuned on multiple related tasks simultaneously. The idea is to leverage the commonalities and differences across these tasks to improve the model's performance. The model can develop a more robust and generalized understanding of the data by learning to perform multiple tasks simultaneously.
This approach leads to improved performance, especially when the tasks it will perform are closely related or when there is limited data for individual tasks. Multi-task learning requires a labeled dataset for each task, making it an inherent component of supervised fine-tuning.

Few shot Learning

Few-shot learning enables a model to adapt to a new task with little task-specific data. The idea is to leverage the vast knowledge model has already gained from pre-training to learn effectively from just a few examples of the new task. This approach is beneficial when the task-specific labeled data is scarce or expensive.
In this technique, the model is given a few examples or "shots” during inference time to learn a new task. The idea behind few-shot learning is to guide the model's predictions by providing context and examples directly in the prompt.
Few-shot learning can also be integrated into the reinforcement learning from human feedback (RLHF) approach if the small amount of task-specific data includes human feedback that guides the model's learning process.

Task specific Fine tuning

This method allows the model to adapt its parameters to the nuances and requirements of the targeted task, thereby enhancing its performance and relevance to that particular domain.
Task-specific fine-tuning is particularly valuable when you want to optimize the model's performance for a single, well-defined task, ensuring that the model excels in generating task-specific content with precision and accuracy.
Task-specific fine-tuning is closely related to transfer learning, but transfer learning is more about leveraging the general features learned by the model, whereas task-specific fine-tuning is about adapting the model to the specific requirements of the new task.

Supervised Fine-Tuning Steps

1. **Pre-trained LLM:** Begin with a pre-trained Large Language Model (LLM) on a diverse dataset for general language understanding.
2. **Labeled Data Selection:** Curate a dataset with labeled examples relevant to the specific task, such as question-answer pairs or labeled documents.
3. **Fine-Tuning Process:** Feed the labeled data into the pre-trained LLM, adjusting its internal parameters based on the provided outputs. This process refines the model for the targeted task.
4. **Evaluation and Iteration:** Assess the performance on validation data, iterate as needed, and fine-tune further until desired task-specific proficiency is achieved.

SFT when to apply

- SFT offers a structured approach to AI training, ideal for problems with clear answers and abundant data, it can be limited by the scope and quality of the training data.
- In SFT, the risk lies in the potential biases in the training data. For instance, if an image recognition model is trained primarily on pictures of animals taken during the day, it might struggle to recognize the same animals in night-time images.
- SFT is suitable for scenarios where the environment is predictable and the data is plentiful, like in handwriting recognition, where the model learns from a vast database of handwritten texts and their transcriptions.

Advantages of Supervised Fine-Tuning

1. **Efficiency**: It reduces the time and resources required for model development by building upon pre-existing knowledge.
2. **Adaptability**: The technique allows models to be tailored to specific tasks, leading to enhanced performance in real-world scenarios.
3. **Generalization**: Supervised fine-tuning enables models to generalize better and adapt to diverse data distributions.

Limitations and Challenges Associated with Supervised Fine-Tuning

1. **Overfitting**: Fine-tuning on small datasets can lead to overfitting, compromising the model’s generalization ability.
2. **Data Dependency**: The effectiveness of fine-tuning is highly reliant on the availability and quality of labeled data for the target task.
3. **Model Degradation**: Repeated fine-tuning cycles can potentially degrade the original knowledge encoded in the pre-trained model.

2.Reinforcement Learning from Human Feedback:

Reinforcement learning from human feedback (RLHF) is an innovative approach that involves training language models through interactions with human feedback. By incorporating human feedback into the learning process, RLHF facilitates the continuous enhancement of language models so they produce more accurate and contextually appropriate responses.
This approach not only leverages the expertise of human evaluators but also enables the model to adapt and evolve based on real-world feedback, ultimately leading to more effective and refined capabilities.

Reward modelling

In this technique, the model generates several possible outputs or actions, and human evaluators rank or rate these outputs based on their quality. The model then learns to predict these human-provided rewards and adjusts its behavior to maximize the predicted rewards. Reward modeling provides a practical way to incorporate human judgment into the learning process, allowing the model to learn complex tasks that are difficult to define with a simple function. This method enables the model to learn and adapt based on human-provided incentives, ultimately enhancing its capabilities.

proximal policy optimization

Proximal policy optimization (PPO) is an iterative algorithm that updates the language model's policy to maximize the expected reward. The core idea of PPO is to take actions that improve the policy while ensuring the changes are not too drastic from the previous policy. This balance is achieved by introducing a constraint on the policy update that prevents harmful large updates while still allowing beneficial small updates.
This constraint is enforced by introducing a surrogate objective function with a clipped probability ratio that serves as a constraint. This approach makes the algorithm more stable and efficient compared to other reinforcement learning methods.

Comparative learning

Comparative ranking is similar to reward modeling, but in comparative ranking, the model learns from relative rankings of multiple outputs provided by human evaluators, focusing more on the comparison between different outputs.
In this approach, the model generates multiple outputs or actions, and human evaluators rank these outputs based on their quality or appropriateness. The model then learns to adjust its behavior to produce outputs that are ranked higher by the evaluators.
By comparing and ranking multiple outputs rather than evaluating each output in isolation, comparative ranking provides more nuanced and relative feedback to the model. This method helps the model understand the task subtleties better, leading to improved results.

Preference Learning

Preference learning, also known as reinforcement learning with preference feedback, focuses on training models to learn from human feedback in the form of preferences between states, actions, or trajectories. In this approach, the model generates multiple outputs, and human evaluators indicate their preference between pairs of outputs.
The model then learns to adjust its behavior to produce outputs that align with the human evaluators' preferences. This method is useful when it is difficult to quantify the output quality with a numerical reward but easier to express a preference between two outputs. Preference learning allows the model to learn complex tasks based on nuanced human judgment, making it an effective technique for fine-tuning the model on real-life applications.

Parameter efficient fine tuning

Parameter-efficient fine-tuning (PEFT) is a technique used to improve the performance of pre-trained LLMs on specific downstream tasks while minimizing the number of trainable parameters. It offers a more efficient approach by updating only a minor fraction of the model parameters during fine-tuning.
PEFT selectively modifies only a small subset of the LLM's parameters, typically by adding new layers or modifying existing ones in a task-specific manner. This approach significantly reduces the computational and storage requirements while maintaining comparable performance to full fine-tuning.

RLHF Steps

1. **Pre-trained and Supervised Fine-Tuning:** Start with a pre-trained LLM and fine-tune it using labeled data as explained in the supervised fine-tuning steps.
2. **Human Interaction:** The LLM generates task-specific completions, which are then presented to humans for evaluation.
3. **Feedback Collection:** Gather human feedback, often in the form of comparisons, ratings, or annotations, to create a reward model.
4. **Reinforcement Learning Process:** Utilize the reward model to drive reinforcement learning, adjusting the LLM’s internal parameters to maximize future expected rewards.
5. **Policy Improvement:** The LLM refines its policy through reinforcement learning, improving its performance on the specific task based on the human feedback received.

RLHF when to apply

1. RLHF though more flexible and adaptable to complex, unpredictable scenarios, demands a careful design of the reward system and significant human involvement.
1. RLHF the challenge is in the reward design; for example, if an AI trained to play a game is rewarded more for defensive moves than offensive ones, it might overly focus on defense, neglecting other aspects of the game.

1. RLHF is more apt for dynamic, unpredictable environments, like in robotic navigation, where the robot must learn to navigate through different terrains and obstacles, a scenario where pre-labeled data is insufficient.

Advantages of RLHF

1. **Aligns LLMs with human values:** RLHF helps LLMs understand and act according to human preferences, promoting responsible and trustworthy AI.
2. **Improves LLM performance:** Specific feedback targets desired outcomes, leading to more accurate and relevant outputs in specific tasks.
3. **Reduces dependence on large datasets:** Human feedback can supplement limited training data, making LLM training more efficient.
4. **Adapts to diverse settings:** Feedback can be tailored to specific contexts and user needs, leading to personalized and relevant LLM responses.

Challenges and Limitations of RLHF

1. **Scalability:** Providing enough human feedback for complex tasks can be resource-intensive.
2. **Bias and noise:** Human evaluators can introduce bias, affecting the LLM's learning. Careful selection and training of evaluators is crucial.
3. **Security vulnerabilities:** Malicious feedback could manipulate the LLM for harmful purposes. Robust security measures are necessary.

Fine-tuning applications

Fine-tuning pre-trained models is an efficient way to leverage the power of large-scale models for a specific task without the need to train a model from scratch. Some of the prominent use cases where fine-tuning LLMs offer significant benefits are as follows.

Sentiment Analysis

Fine-tuning models on specific company data, unique domains, or unique tasks helps with the accurate analysis and understanding of the sentiment expressed in textual content, enabling businesses to gain valuable insights from customer feedback, social media posts, and product reviews. These insights can inform decision-making processes, marketing strategies, and product development efforts.

For instance, sentiment analysis can help businesses identify trends, gauge customer satisfaction, and pinpoint areas for improvement. In social media, fine-tuned models enable organizations to track public sentiment towards their brand, products, or services, allowing for proactive reputation management and targeted engagement with customers. Overall, fine-tuned large language models are a powerful tool for sentiment analysis that can provide businesses with valuable insights into customer sentiment.

Chatbots

Fine-tuning allows chatbots to generate more contextually relevant and engaging conversations, improving customer interactions and providing personalized assistance in various industries such as customer service, healthcare, e-commerce, and finance. For instance, in healthcare, chatbots can answer detailed medical queries, and offer support, thereby augmenting patient care and accessibility to healthcare information.
In e-commerce, fine-tuned chatbots can assist customers with product inquiries, recommend items based on preferences, and facilitate seamless transactions. In the finance industry, chatbots can provide personalized financial advice, assist with account management, and address customer inquiries with a high degree of accuracy and relevance. Overall, fine-tuning language models for chatbot applications enhances their conversational abilities, making them valuable assets across a wide range of industries.

Summarization

Fine-tuned models automatically generate concise and informative summaries of long documents, articles, or conversations, facilitating efficient information retrieval and knowledge management. This capability is invaluable for professionals who need to analyze vast amounts of data to extract key insights.
In academic and research, fine-tuned summarization models can condense extensive research papers, enabling scholars to grasp key findings and insights more quickly. In a corporate environment, fine-tuned summarization models can assist in distilling lengthy reports, emails, and business documents, facilitating efficient decision-making and knowledge understanding. Overall, the application of fine-tuned language models for summarization enhances information accessibility and comprehension, making it a valuable tool across various domains.
Fine-tuned models deliver optimized outcomes in various use cases, demonstrating the versatility and impact of fine-tuning in augmenting the capabilities of LLMs for unique business solutions.

Optimizing pre-trained models

Parameter efficient fine tuning:

Parameter-efficient Fine-tuning (PEFT) is a technique used in Natural Language Processing (NLP) to improve the performance of pre-trained language models on specific downstream tasks. It involves reusing the pre-trained model’s parameters and fine-tuning them on a smaller

dataset, which saves computational resources and time compared to training the entire model from scratch.
PEFT achieves this efficiency by freezing some of the layers of the pre-trained model and only fine-tuning the last few layers that are specific to the downstream task. This way, the model can be adapted to new tasks with less computational overhead and fewer labeled examples.
Although PEFT has been a relatively novel concept, updating the last layer of models has been in practice in the field of computer vision since the introduction of transfer learning. Even in NLP, experiments with static and non-static word embeddings were carried out early on.
Parameter-efficient fine-tuning aims to improve the performance of pre-trained models, such as BERT and RoBERTa, on various downstream tasks, including sentiment analysis, named entity recognition, and question-answering. It achieves this in low-resource settings with limited data and computational resources. It modifies only a small subset of model parameters and is less prone to overfitting.

When to USE OF PEFT:

Parameter-efficient fine-tuning can be particularly useful in scenarios where computational resources are limited or where large pre-trained models are involved. In such cases, PEFT can provide a more efficient way of fine-tuning the model without sacrificing performance. However, it’s important to note that PEFT may sometimes achieve a different level of performance than full fine-tuning, especially in cases where the pre-trained model requires significant modification to perform well on the new task.

Benefits of PEFT

Here we will discuss the benefits of PEFT in relation to traditional fine-tuning. So, let us understand why parameter-efficient fine-tuning is more beneficial than fine-tuning.

1. Decreased computational and storage costs:

PEFT involves fine-tuning only a small number of extra model parameters while freezing most parameters of the pre-trained LLMs, thereby reducing computational and storage costs significantly.

2.Overcoming catastrophic forgetting:

During full fine-tuning of LLMs, catastrophic forgetting can occur where the model forgets the knowledge it learned during pretraining. PEFT stands to overcome this issue by only updating a few parameters.

3.Better performance in low-data regimes:

PEFT approaches have been shown to perform better than full fine-tuning in low-data regimes and generalize better to out-of-domain scenarios.

4.Portability:

PEFT methods enable users to obtain tiny checkpoints worth a few MBs compared to the large checkpoints of full fine-tuning. This makes the trained weights from PEFT approaches easy to deploy and use for multiple tasks without replacing the entire model.

5.Performance comparable to full fine-tuning:

PEFT enables achieving comparable performance to full fine-tuning with only small number of trainable parameters.

Standard fine tuning:

A standard fine-tuning process involves adjusting the *hidden representations (h)*extracted by transformer models to enhance their performance in downstream tasks. These hidden representations refer to any features the transformer architecture extracts, such as the output of a transformer layer or a self-attention layer.

PEFT TECHNIQUES:

1.ADAPTER:

Adapters are a special type of submodule that can be added to pre-trained language models to modify their hidden representation during fine-tuning. By inserting adapters after the multi-head attention and feed-forward layers in the transformer architecture, we can update only the parameters in the adapters during fine-tuning while keeping the rest of the model parameters frozen.
Adopting adapters can be a straightforward process. All that is required is to add adapters into each transformer layer and place a classifier layer on top of the pre-trained model. By updating the parameters of the adapters and the classifier head, we can improve the performance of the pre-trained model on a particular task without updating the entire model. This approach can save time and computational resources while still producing impressive results.

How it works:

The adapter module comprises two feed-forward projection layers connected with a non-linear activation layer. There is also a skip connection that bypasses the feed-forward layers.

If we take the adapter placed right after the multi-head attention layer, then the input to the adapter layer is the hidden representation h calculated by the multi-head attention layer. Here, h takes two different paths in the adapter layer; one is the skip-connection, which leaves the input unchanged, and the other way involves the feed-forward layers.
Initially, the first feed-forward layer projects h into a low-dimension space. This space has a dimension less than h. Following this, the input is passed through a non-linear activation function, and the second feed-forward layer then projects it back up to the dimensionality of h. The results obtained from the two ways are summed together to obtain the final output of the adapter module.
The skip-connection preserves the original input h of the adapter, while the feed-forward path generates an incremental change, represented as Δh, based on the original h. By adding the incremental change Δh, obtained from the feed-forward layer with the original h from the previous layer, the adapter modifies the hidden representation calculated by the pre-trained model. This allows the adapter to alter the hidden representation of the pre-trained model, thereby changing its output for a specific task.

2.LORA:

Low-Rank Adaptation (LoRA) of large language models is another approach in the area of fine-tuning models for specific tasks or domains. Similar to the adapters, LoRA is also a small trainable submodule that can be inserted into the transformer architecture. It involves freezing the pre-trained model weights and injecting trainable rank decomposition matrices into each layer of the transformer architecture, greatly diminishing the number of trainable parameters for downstream tasks. This method can minimize the number of trainable parameters by up to 10,000 times and the GPU memory necessity by 3 times while still performing on par or better than fine-tuning model quality on various tasks. LoRA also allows for more efficient task-switching, lowering the hardware barrier to entry, and has no additional inference latency compared to other methods.

How it works:

LoRA is inserted in parallel to the modules in the pre-trained transformer model, specifically in parallel to the feed-forward layers. A feed-forward layer has two projection layers and a non-linear layer in between them, where the input vector is projected into an output vector with a different dimensionality using an affine transformation. The LoRA layers are inserted next to each of the two feed-forward layers.
Now, let us consider the feed-forward up-project layer and the LoRA next to it. The original parameters of the feed-forward layer take the output from the previous layer with the dimension dmodel and projects it into dFFW. Here, FFW is the abbreviation for feed-forward. The LoRA module placed next to it consists of two feed-forward layers. The LoRA's first feed-forward layer takes the same input as the feed-forward up-project layer and projects it into an r-dimensional

vector, which is far less than the dmodel. Then, the second feed-forward layer projects the vector into another vector with a dimensionality of dFFW. Finally, the two vectors are added together to form the final representation.
As we have discussed earlier, fine-tuning is changing the hidden representation hcalculated by the original transformer model. Hence, in this case, the hidden representation calculated by the feed-forward up-project layer of the original transformer is h. Meanwhile, the vector calculated by LoRA is the incremental change Δh that is used to modify the original h. Thus, the sum of the original representation and the incremental change is the updated hidden representation h'.
By inserting LoRA modules next to the feed-forward layers and a classifier head on top of the pre-trained model, task-specific parameters for each task are kept to a minimum.

3.Pre-fix tuning:

Prefix-tuning is a lightweight alternative to fine-tuning large pre-trained language models for natural language generation tasks. Fine-tuning requires updating and storing all the model parameters for each task, which can be very expensive given the large size of current models. Prefix-tuning keeps the language model parameters frozen and optimizes a small continuous task-specific vector called the prefix. In prefix-tuning, the prefix is a set of free parameters that are trained along with the language model. The goal of prefix-tuning is to find a context that steers the language model toward generating text that solves a particular task.

How it works:

The prefix can be seen as a sequence of "virtual tokens" that subsequent tokens can attend to. By learning only 0.1% of the parameters, prefix-tuning obtains comparable performance to fine-tuning in the full data setting, outperforms fine-tuning in low-data settings, and extrapolates better to examples with topics unseen during training.
Similar to all previously mentioned PEFT techniques, the end goal of prefix tuning is to reach h'. Prefix tuning uses prefixes to modify the hidden representations extracted by the original pre-trained language models. When the incremental change Δh is added to the original hidden representation h, we get the modified representation, i.e., h'.
When using prefix tuning, only the prefixes are updated, while the rest of the layers are fixed and not updated.

4.Prompt tuning:

Prompt tuning is another PEFT technique for adapting pre-trained language models to specific downstream tasks. Unlike the traditional "model tuning" approach, where all the pre-trained model parameters are tuned for each task, prompt tuning involves learning soft prompts through backpropagation that can be fine-tuned for specific tasks by incorporating labeled examples. Prompt tuning outperforms the few-shot learning of GPT-3 and becomes more competitive as the model size increases. It also benefits domain transfer's robustness and enables efficient prompt ensembling. It requires storing a small task-specific prompt for each task, making it easier to reuse a single frozen model for multiple downstream tasks, unlike model tuning, which requires making a task-specific copy of the entire pre-trained model for each task

.How it works:

Prompt tuning is a simpler variant of prefix tuning. In it, some vectors are prepended at the beginning of a sequence at the input layer. When presented with an input sentence, the embedding layer converts each token into its corresponding word embedding, and the prefix embeddings are prepended to the sequence of token embeddings. Next, the pre-trained transformer layers will process the embedding sequence like a transformer model does to a normal sequence. Only the prefix embeddings are adjusted during the fine-tuning process, while the rest of the transformer model is kept frozen and unchanged.
This technique has several advantages over traditional fine-tuning methods, including improved efficiency and reduced computational overhead. Additionally, the fact that only the prefix embeddings are fine-tuned means that there is a lower risk of overfitting to the training data, thereby producing more robust and generalizable models.

P-tuning:

P-tuning can improve the performance of language models such as GPTs in Natural Language Understanding (NLU) tasks. Traditional fine-tuning techniques have not been effective for GPTs, but P-tuning uses trainable continuous prompt embeddings to improve their performance. This method has been tested on two NLU benchmarks, LAMA and SuperGLUE, and has shown significant improvements in precision and world knowledge recovery. P-tuning also reduces the need for prompt engineering and outperforms state-of-the-art approaches on the few-shot SuperGLUE benchmark.
P-tuning can be used to improve pre-trained language models for various tasks, including sentence classification and predicting a country's capital. The technique involves modifying the input embeddings of the pre-trained language model with differential output embeddings generated using a prompt. The continuous prompts can be optimized using a downstream loss function and a prompt encoder, which helps solve discreteness and association challenges.

IA3:

IA3, short for Infused Adapter by Inhibiting and Amplifying Inner Activations, is another parameter-efficient fine-tuning technique designed to improve upon the LoRA technique. It focuses on making the fine-tuning process more efficient by reducing the number of trainable parameters in a model.
Both LoRA and IA3 share some similarities in their core objective of improving fine-tuning efficiency. They achieve this by introducing learned components, reducing the number of trainable parameters, and keeping the original pre-trained weights frozen. These shared characteristics make both techniques valuable tools for adapting large pre-trained models to specific tasks while minimizing computational demands. Additionally, both LoRA and IA3

prioritize maintaining model performance, ensuring that fine-tuned models remain competitive with fully fine-tuned ones. Furthermore, their capacity to merge adapter weights without adding inference latency contributes to their versatility and practicality for real-time applications and various downstream tasks.

How it works:

IA3 optimizes the fine-tuning process by rescaling the inner activations of a pre-trained model using learned vectors. These learned vectors are incorporated into the attention and feedforward modules within a standard transformer-based architecture. The key innovation of IA3 is that it freezes the original pre-trained weights of the model, making only the introduced learned vectors trainable during fine-tuning. This drastic reduction in the number of trainable parameters significantly improves the efficiency of fine-tuning without compromising model performance. IA3 is compatible with various downstream tasks, maintains inference speed, and can be applied to specific layers of a neural network, making it a valuable tool for efficient model adaptation and deployment.

Process of PEFT:

The steps involved in parameter-efficient fine-tuning can vary depending on the specific implementation and the pre-trained model being used. However, here is a general outline of the steps involved in PEFT:

Pre-training:

Initially, a large-scale model is pre-trained on a large dataset using a general task such as image classification or language modeling. This pre-training phase helps the model learn meaningful representations and features from the data.

Task-specific dataset:

Gather or create a dataset that is specific to the target task you want to fine-tune the pre-trained model for. This dataset should be labeled and representative of the target task.

Parameter identification: Identify or estimate the importance or relevance of parameters in the pre-trained model for the target task. This step helps in determining which parameters should be prioritized during fine-tuning. Various techniques, such as importance estimation, sensitivity analysis, or gradient-based methods, can be used to identify important parameters.

Subset selection:

Select a subset of the pre-trained model's parameters based on their importance or relevance to the target task. The subset can be determined by setting certain criteria, such as a threshold on the importance scores or selecting the top-k most important parameters.

Fine-tuning:

Initialize the selected subset of parameters with the values from the pre-trained model and freeze the remaining parameters. Fine-tune the selected parameters using the task-specific dataset. This involves training the model on the target task data, typically using techniques like Stochastic Gradient Descent (SGD) or Adam optimization.

Evaluation:

Evaluate the performance of the fine-tuned model on a validation set or through other evaluation metrics relevant to the target task. This step helps assess the effectiveness of PEFT in achieving the desired performance while using fewer parameters.

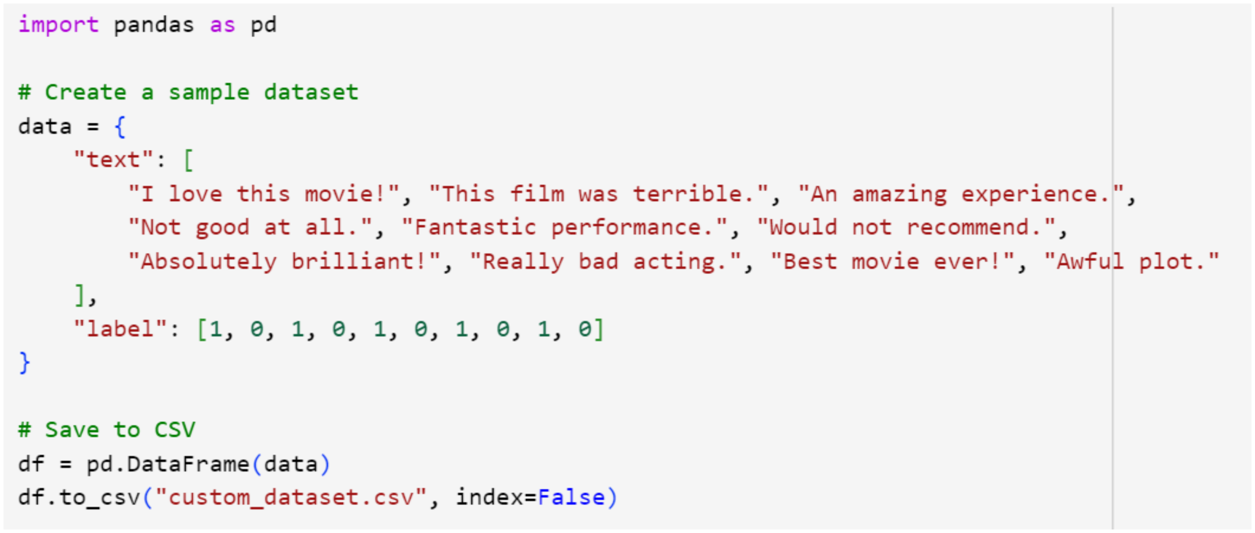
Iterative refinement (optional):

Depending on the performance and requirements, you may choose to iterate and refine the PEFT process by adjusting the criteria for parameter selection, exploring different subsets, or fine-tuning for additional epochs to optimize the model's performance further.
However, it's important to note that the specific implementation details and techniques used in PEFT can vary across research papers as well as applications.

Fine tuning Procedure

Prepare the Dataset:

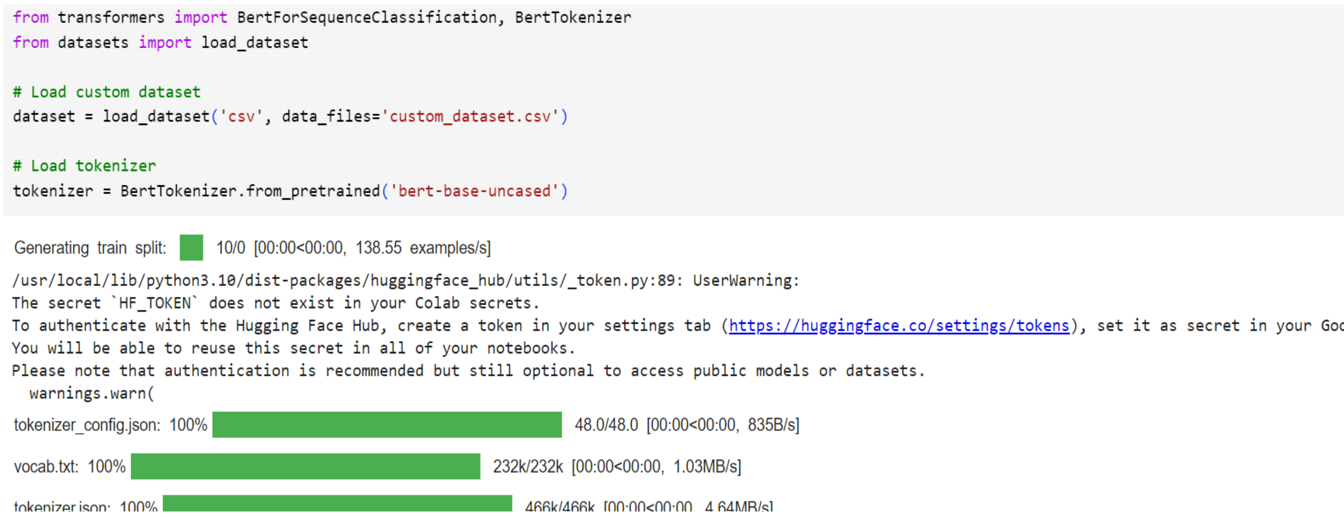
- Assemble a dataset containing text documents and corresponding labels.
- Organize the data into a format suitable for model training, such as a CSV file.



```python
import pandas as pd

# Create a sample dataset
data = {
    "text": [
        "I love this movie!", "This film was terrible.", "An amazing experience.",
        "Not good at all.", "Fantastic performance.", "Would not recommend.",
        "Absolutely brilliant!", "Really bad acting.", "Best movie ever!", "Awful plot."
    ],
    "label": [1, 0, 1, 0, 1, 0, 1, 0, 1, 0]
}

# Save to CSV
df = pd.DataFrame(data)
df.to_csv("custom_dataset.csv", index=False)
```

Load the Pre-Trained Model and Tokenizer:

- Utilize a pre-trained BERT model and tokenizer, which have learned representations of the language.
- The tokenizer breaks down the text into tokens, while the model serves as the backbone for the classification task.

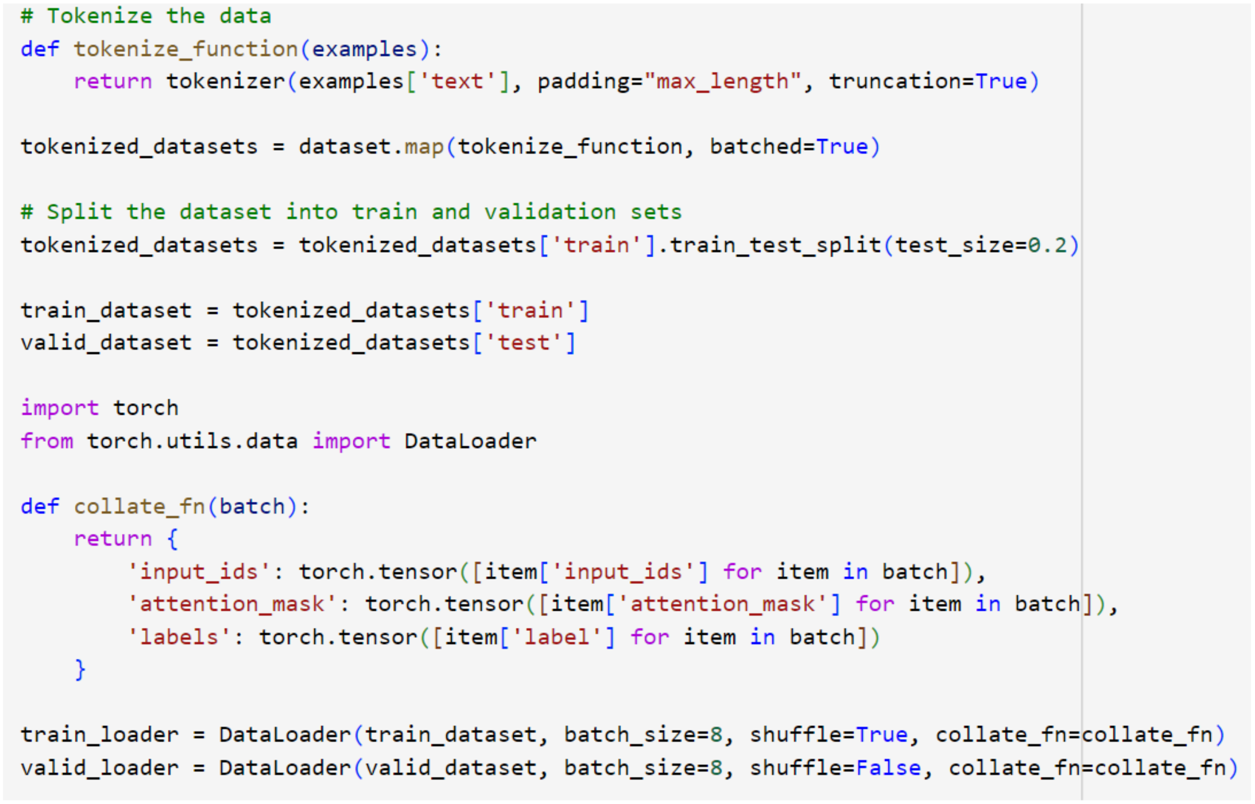


```python
from transformers import BertForSequenceClassification, BertTokenizer
from datasets import load_dataset

# Load custom dataset
dataset = load_dataset('csv', data_files='custom_dataset.csv')

# Load tokenizer
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
```

```
Generating train split:  10/0 [00:00<00:00, 138.55 examples/s]
/usr/local/lib/python3.10/dist-packages/huggingface_hub/utils/_token.py:89: UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (https://huggingface.co/settings/tokens), set it as secret in your Goo
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access public models or datasets.
  warnings.warn(
tokenizer_config.json: 100%  48.0/48.0 [00:00<00:00, 835B/s]
vocab.txt: 100%  232k/232k [00:00<00:00, 1.03MB/s]
tokenizer.json: 100%  466k/466k [00:00<00:00, 4.64MB/s]
```

Preprocess the Data:

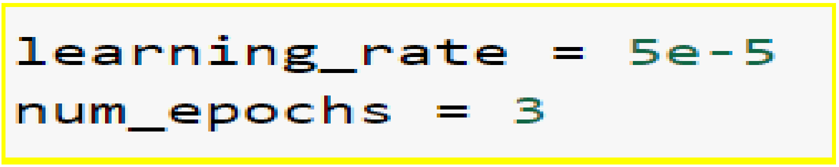
- Tokenize the text data using the loaded tokenizer, converting it into numerical tokens that the model can understand.
- Split the dataset into training and validation sets to facilitate model training and evaluation.



```
# Tokenize the data
def tokenize_function(examples):
    return tokenizer(examples['text'], padding="max_length", truncation=True)

tokenized_datasets = dataset.map(tokenize_function, batched=True)

# Split the dataset into train and validation sets
tokenized_datasets = tokenized_datasets['train'].train_test_split(test_size=0.2)

train_dataset = tokenized_datasets['train']
valid_dataset = tokenized_datasets['test']

import torch
from torch.utils.data import DataLoader

def collate_fn(batch):
    return {
        'input_ids': torch.tensor([item['input_ids'] for item in batch]),
        'attention_mask': torch.tensor([item['attention_mask'] for item in batch]),
        'labels': torch.tensor([item['label'] for item in batch])
    }

train_loader = DataLoader(train_dataset, batch_size=8, shuffle=True, collate_fn=collate_fn)
valid_loader = DataLoader(valid_dataset, batch_size=8, shuffle=False, collate_fn=collate_fn)
```

Set Hyperparameters:

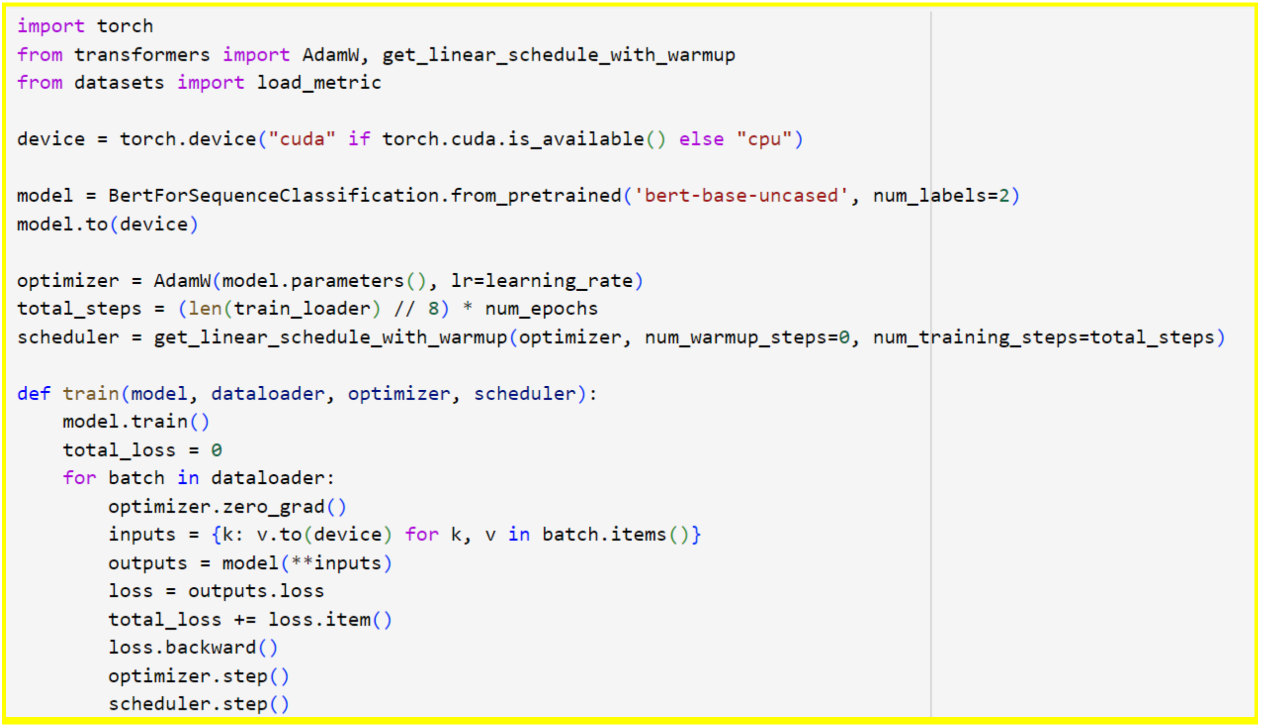
- Define hyperparameters such as the learning rate, number of epochs, and batch size.
- These settings govern the training process and directly impact the model's performance.



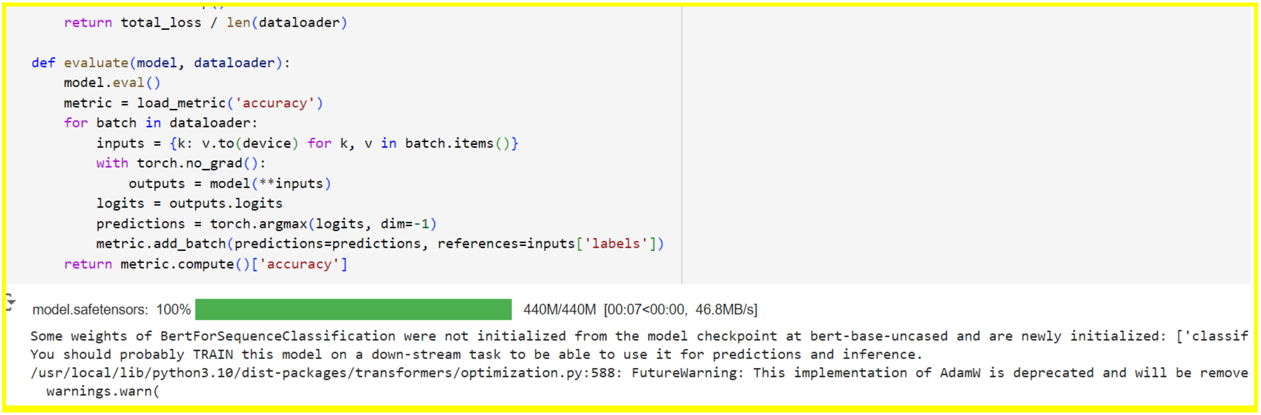
```
learning_rate = 5e-5
num_epochs = 3
```

Define Training and Evaluation Functions:

- Implement functions for training and evaluating the model.
- The training function iterates over the training dataset, adjusting the model's parameters to minimize a chosen loss function.
- The evaluation function assesses the model's performance on a separate validation dataset, calculating metrics like accuracy.



```python
import torch
from transformers import AdamW, get_linear_schedule_with_warmup
from datasets import load_metric

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

model = BertForSequenceClassification.from_pretrained('bert-base-uncased', num_labels=2)
model.to(device)

optimizer = AdamW(model.parameters(), lr=learning_rate)
total_steps = (len(train_loader) // 8) * num_epochs
scheduler = get_linear_schedule_with_warmup(optimizer, num_warmup_steps=0, num_training_steps=total_steps)

def train(model, dataloader, optimizer, scheduler):
    model.train()
    total_loss = 0
    for batch in dataloader:
        optimizer.zero_grad()
        inputs = {k: v.to(device) for k, v in batch.items()}
        outputs = model(**inputs)
        loss = outputs.loss
        total_loss += loss.item()
        loss.backward()
        optimizer.step()
        scheduler.step()
```

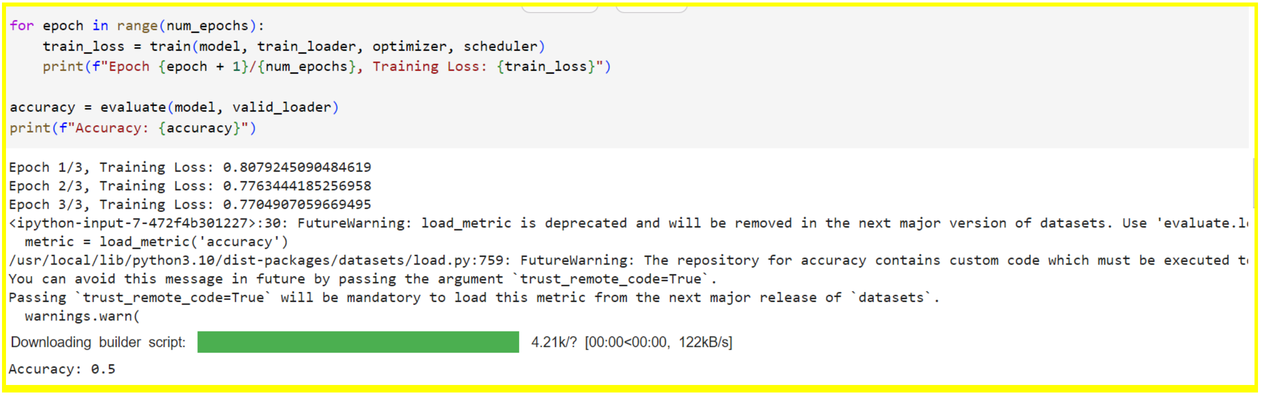


```python
    return total_loss / len(dataloader)

def evaluate(model, dataloader):
    model.eval()
    metric = load_metric('accuracy')
    for batch in dataloader:
        inputs = {k: v.to(device) for k, v in batch.items()}
        with torch.no_grad():
            outputs = model(**inputs)
        logits = outputs.logits
        predictions = torch.argmax(logits, dim=-1)
        metric.add_batch(predictions=predictions, references=inputs['labels'])
    return metric.compute()['accuracy']
```

```
model.safetensors: 100%  440M/440M [00:07<00:00, 46.8MB/s]
Some weights of BertForSequenceClassification were not initialized from the model checkpoint at bert-base-uncased and are newly initialized: ['classif
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
/usr/local/lib/python3.10/dist-packages/transformers/optimization.py:588: FutureWarning: This implementation of AdamW is deprecated and will be remove
  warnings.warn(
```

Fine-Tune the Model:

- Train the model on the training dataset, allowing it to adapt to the specific characteristics of the document classification task.
- Monitor the model's performance on the validation dataset during training to ensure it generalizes well to unseen data.

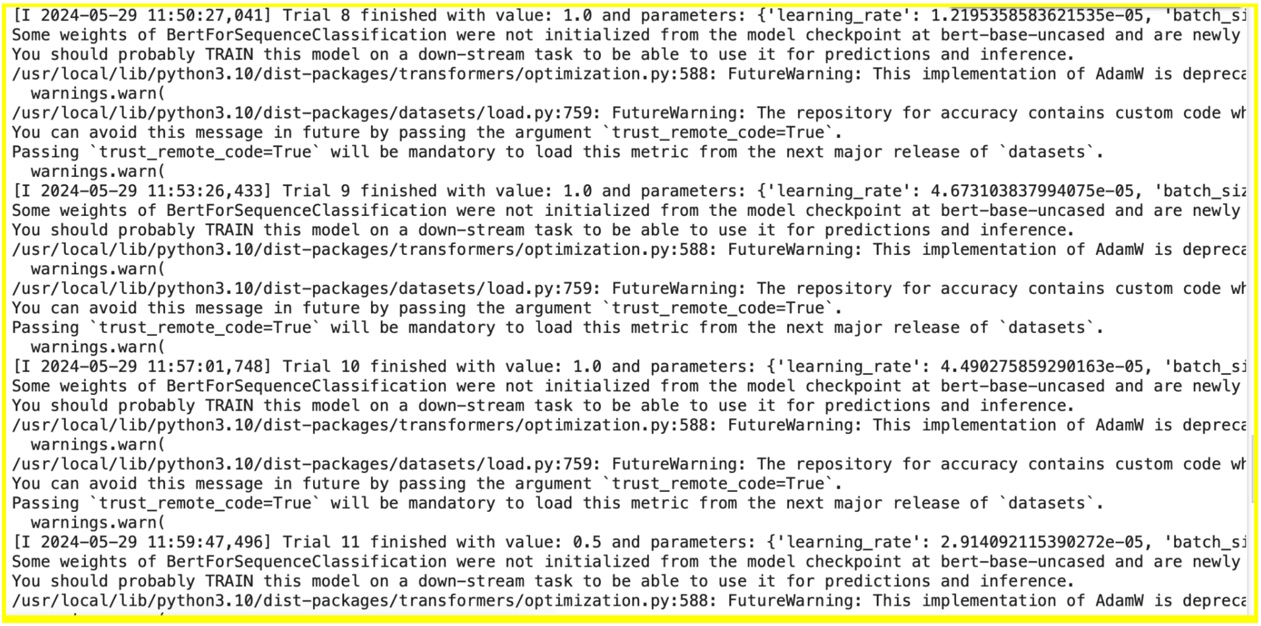


```
for epoch in range(num_epochs):
    train_loss = train(model, train_loader, optimizer, scheduler)
    print(f"Epoch {epoch + 1}/{num_epochs}, Training Loss: {train_loss}")

accuracy = evaluate(model, valid_loader)
print(f"Accuracy: {accuracy}")
```

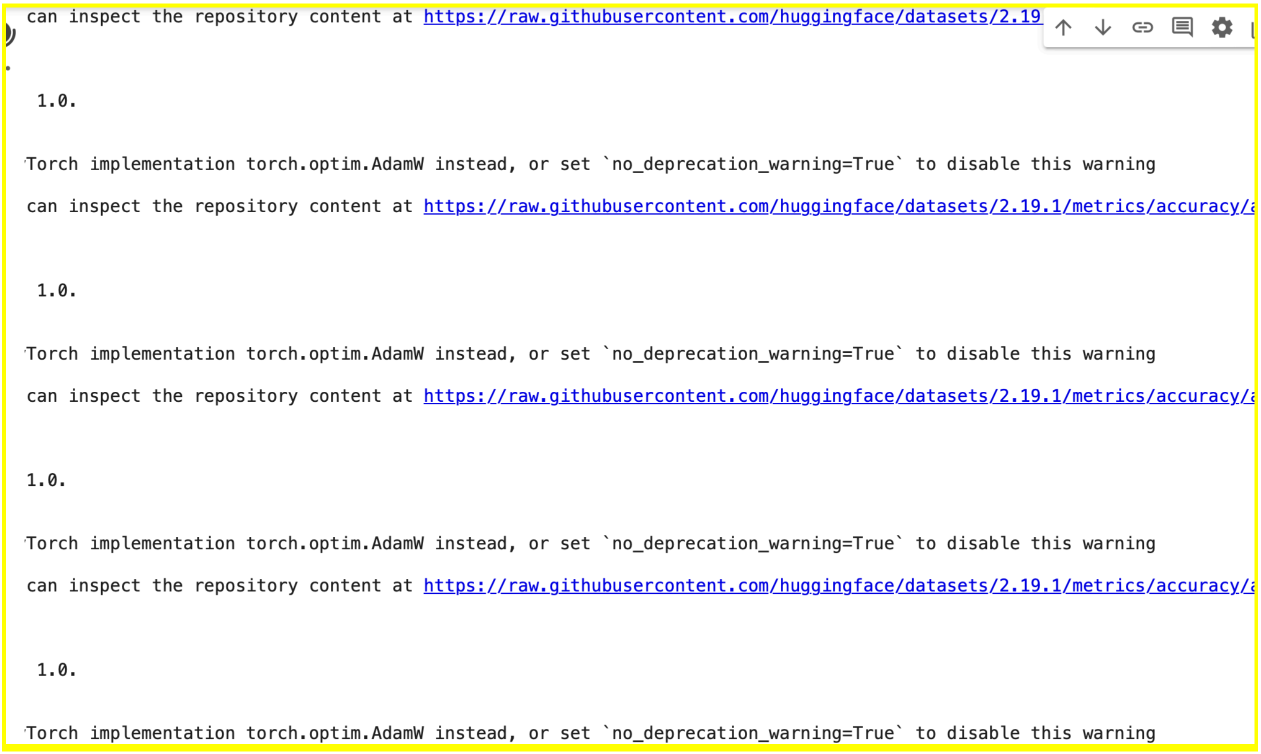
```
Epoch 1/3, Training Loss: 0.8079245090484619
Epoch 2/3, Training Loss: 0.7763444185256958
Epoch 3/3, Training Loss: 0.7704907059669495
<ipython-input-7-472f4b301227>:30: FutureWarning: load_metric is deprecated and will be removed in the next major version of datasets. Use 'evaluate.l
  metric = load_metric('accuracy')
/usr/local/lib/python3.10/dist-packages/datasets/load.py:759: FutureWarning: The repository for accuracy contains custom code which must be executed t
You can avoid this message in future by passing the argument `trust_remote_code=True`.
Passing `trust_remote_code=True` will be mandatory to load this metric from the next major release of `datasets`.
  warnings.warn(
Downloading builder script: 4.21k/? [00:00<00:00, 122kB/s]
Accuracy: 0.5
```

Optimize Hyperparameters (Optional):

- Explore techniques like hyperparameter optimization to find the combination of hyperparameters that maximizes the model's performance.
- Tools like Optuna can systematically search the hyperparameter space and identify optimal settings.



```
[I 2024-05-29 11:50:27,041] Trial 8 finished with value: 1.0 and parameters: {'learning_rate': 1.2195358583621535e-05, 'batch_si
Some weights of BertForSequenceClassification were not initialized from the model checkpoint at bert-base-uncased and are newly
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
/usr/local/lib/python3.10/dist-packages/transformers/optimization.py:588: FutureWarning: This implementation of AdamW is depreca
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/datasets/load.py:759: FutureWarning: The repository for accuracy contains custom code wh
You can avoid this message in future by passing the argument `trust_remote_code=True`.
Passing `trust_remote_code=True` will be mandatory to load this metric from the next major release of `datasets`.
  warnings.warn(
[I 2024-05-29 11:53:26,433] Trial 9 finished with value: 1.0 and parameters: {'learning_rate': 4.673103837994075e-05, 'batch_siz
Some weights of BertForSequenceClassification were not initialized from the model checkpoint at bert-base-uncased and are newly
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
/usr/local/lib/python3.10/dist-packages/transformers/optimization.py:588: FutureWarning: This implementation of AdamW is depreca
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/datasets/load.py:759: FutureWarning: The repository for accuracy contains custom code wh
You can avoid this message in future by passing the argument `trust_remote_code=True`.
Passing `trust_remote_code=True` will be mandatory to load this metric from the next major release of `datasets`.
  warnings.warn(
[I 2024-05-29 11:57:01,748] Trial 10 finished with value: 1.0 and parameters: {'learning_rate': 4.490275859290163e-05, 'batch_si
Some weights of BertForSequenceClassification were not initialized from the model checkpoint at bert-base-uncased and are newly
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
/usr/local/lib/python3.10/dist-packages/transformers/optimization.py:588: FutureWarning: This implementation of AdamW is depreca
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/datasets/load.py:759: FutureWarning: The repository for accuracy contains custom code wh
You can avoid this message in future by passing the argument `trust_remote_code=True`.
Passing `trust_remote_code=True` will be mandatory to load this metric from the next major release of `datasets`.
  warnings.warn(
[I 2024-05-29 11:59:47,496] Trial 11 finished with value: 0.5 and parameters: {'learning_rate': 2.914092115390272e-05, 'batch_si
Some weights of BertForSequenceClassification were not initialized from the model checkpoint at bert-base-uncased and are newly
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
/usr/local/lib/python3.10/dist-packages/transformers/optimization.py:588: FutureWarning: This implementation of AdamW is depreca
```



```
can inspect the repository content at https://raw.githubusercontent.com/huggingface/datasets/2.19

 1.0.

Torch implementation torch.optim.AdamW instead, or set `no_deprecation_warning=True` to disable this warning
can inspect the repository content at https://raw.githubusercontent.com/huggingface/datasets/2.19.1/metrics/accuracy/a

 1.0.

Torch implementation torch.optim.AdamW instead, or set `no_deprecation_warning=True` to disable this warning
can inspect the repository content at https://raw.githubusercontent.com/huggingface/datasets/2.19.1/metrics/accuracy/a

1.0.

Torch implementation torch.optim.AdamW instead, or set `no_deprecation_warning=True` to disable this warning
can inspect the repository content at https://raw.githubusercontent.com/huggingface/datasets/2.19.1/metrics/accuracy/a

 1.0.

Torch implementation torch.optim.AdamW instead, or set `no_deprecation_warning=True` to disable this warning
```

Finalize the Model:

- Once the model achieves satisfactory performance, save it along with the tokenizer for future use.
- This allows you to deploy the model in real-world applications and easily load it for inference

```python
best_params = study.best_params
model = BertForSequenceClassification.from_pretrained('bert-base-uncased', num_labels=2)
model.to(device)
optimizer = AdamW(model.parameters(), lr=best_params['learning_rate'])
total_steps = (len(train_loader) // best_params['batch_size']) * best_params['num_epochs']
scheduler = get_linear_schedule_with_warmup(optimizer, num_warmup_steps=0, num_training_steps=total_steps)
train_loader = DataLoader(train_dataset, batch_size=best_params['batch_size'], shuffle=True, collate_fn=collate_fn)
valid_loader = DataLoader(valid_dataset, batch_size=best_params['batch_size'], shuffle=False, collate_fn=collate_fn)
for epoch in range(best_params['num_epochs']):
    train_loss = train(model, train_loader, optimizer, scheduler)
    print(f"Epoch {epoch + 1}/{best_params['num_epochs']}, Training Loss: {train_loss}")
accuracy = evaluate(model, valid_loader)
print(f"Final Accuracy: {accuracy}")
model.save_pretrained('fine_tuned_model')
tokenizer.save_pretrained('fine_tuned_model')
```