Using Pretrained Language Models and Finetuning Approaches

By
George Medina

CS 4395 – Senior Project Report
In partial Fulfillment of a Bachelor's of Science in Computer Science
Department of Computer Science and Engineering Technology
University of Houston – Downtown

Fall 2023

Faculty Advisor: Subash Pakhrin ______________________________

Course Instructor: Dr. Ling Xu ______________________________

Table of Contents

Abstract

Finetuning stands as a critical method within the realm of deep learning, enabling the adaptation of pre-existing models to specific tasks. This exploration, extensively covered in this paper, serves to enhance model performance and alignment with specific domains. The abstract provides insight into the underlying reasons motivating finetuning, its advantages, the methodologies employed, and the spectrum of applications it caters to. Additionally, it briefly touches upon the challenges faced in the finetuning process and offers glimpses into potential future directions within the field of refining deep learning models through finetuning techniques. This paper will describe the history language models as well as delve into the process of finetuning.

Acknowledgement

To Professor Subash Pakhrin,

Thank you for your mentorship, wisdom, and knowledge throughout this journey.

Chapter 1- Description of the Project

Introduction to Large Language Models(LLMs)

Artificial intelligence is a branch of computer science that is evolving rapidly as machine intelligence grows parallel with technology. It is the process of creating intelligence machines that can think for themselves. At its simplest form, it is a field that combines computer science and datasets to enable critical thinking in problem solving format. At the foundation of this interaction lies the most profound element of human intelligence: language. Language serves as the primary means of communication, a nuanced, deeply structured, and evolving system that AI seeks to understand and replicate. This allowed for the exchange of ideas between humans and machines, a beginning step towards machines that can engage with us as naturally as we do with each other. Language is the natural conduit of communication between people. It begins with two individuals wishing to convey words between each other but let's break this process down. Individual one, who we will label as sender, begins to conceptualize a thought or idea that is stored within themselves. Then that speaker becomes intent with communication and wishes to convey that idea to another individual, who will be labeled receiver. Then, sender encodes that idea into language and articulates that message with the proper language to the receiving party. Then when the message is received, that party then decodes the message, interprets the syntax and semantics of the sentence, and attempts to comprehend the message. Usually at this point, after processing the message, feedback is formed from the receiver internally, and begins the processes of communicating back, with the feedback they wish to convey. This is the processes of communicating between two people and is important to understand because as the world has evolved, technology has turned these processes into a computerized process, called natural language processing(NLP).

Natural Language Processing

In the 1940's after World War 2, NLP emerged as a way to help computers understand and use human languages. This is where foundational work took place to bridge the gap between human communication and machine understanding. Natural language processes are where computers are given the ability to understand text and words the same way humans use language to communicate. Here, with the computers ability to understand, interpret and processes human language, that gave rise to the language modeling.

Language Modeling

Language modeling is described as predicting the next word in the any given sentence. It is where the understanding of natural language processing is applied. Language models are also described as systems that are trained on large amounts of text data that can inference the probability of the next word in a sequence or generate new words in a sequence based on the patterns in the sequence. These models became the building blocks for various applications, from simple automated responses to complex interaction systems capable of engaging in human-like conversations, thereby revolutionizing how machines interface with human language.

Neural Language Models

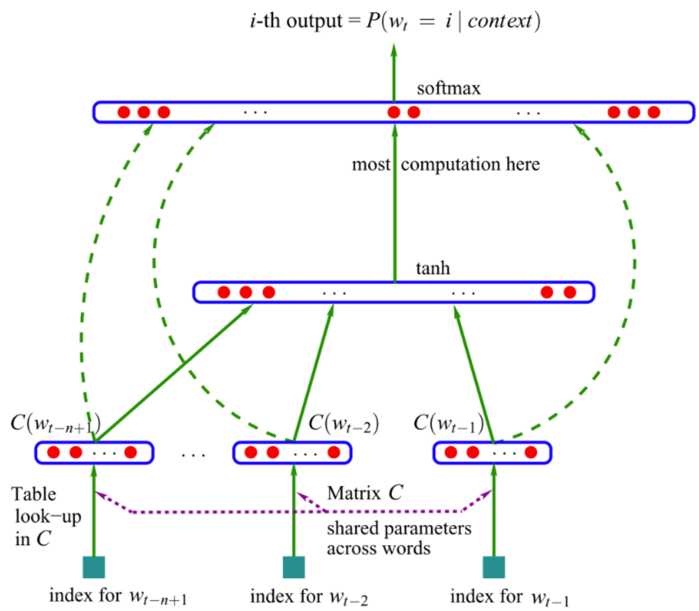


Figure 1. A feed-forward neural network language model (reprinted from Bengio, Y., Ducharme, R., Vincent, P., & Jauvin, C., 2003, Journal of Machine Learning Research, 3, p. 1138)."

Neural language models marked a departure from traditional statistical methods like n-grams. These models use neural networks to predict the likelihood of a sequence of words, changing how machines understand text. The first significant leap was the introduction of feed-forward neural networks in language modeling, pioneered by Bengio et al. in the early 2000s. These models could learn distributed representations for words (word embeddings) and capture contextual information, in a limited window of words.

Multi-Tasked Learning

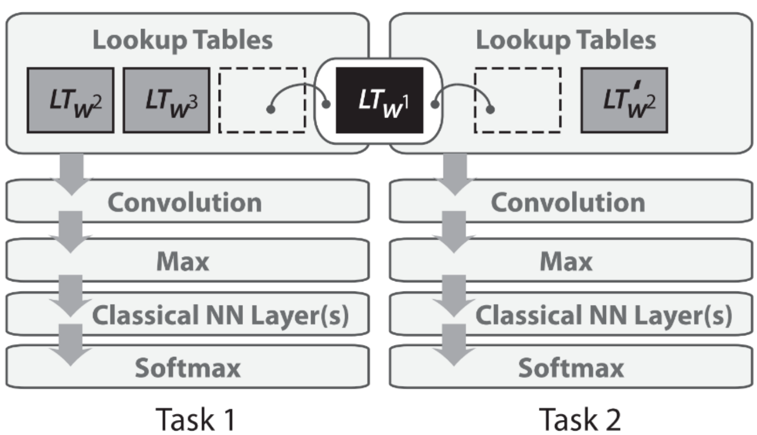


Figure 2. Sharing of word embedding matrices (reprinted from Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., & Kuksa, P., 2011, Journal of Machine Learning Research, 12, pp. 2493-2537).

As neural networks grew more sophisticated, the concept of multi-task learning began to grow. This approach involved training a single model on multiple related tasks simultaneously, while allowing it to learn shared representations that were similar across these tasks. For language models, this meant an improvement in generalization and efficiency, as the model could leverage similarities in linguistic patterns across different linguistic tasks.

Word Embeddings and Word2Vec

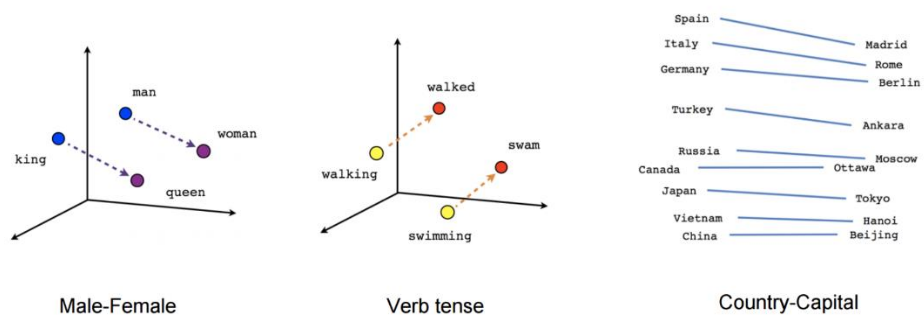


Figure 3. Conceptual diagrams illustrating the relationships captured by the word2vec model, including analogies such as male-female, verb tense, and country-capital (reprinted from Mikolov, T., et al., 2013).

The arrival of word embeddings after neural networks, particularly with the introduction of Word2Vec by Mikolov et al. Word2Vec provided a way to convert words into high-dimensional vectors that captured semantic and syntactic relationships. For example, vector operations would reveal relationships and associations between words. This method of representing words as vectors became a foundational technique in NLP.

Evolution with Neural Networks: RNNs and CNNs

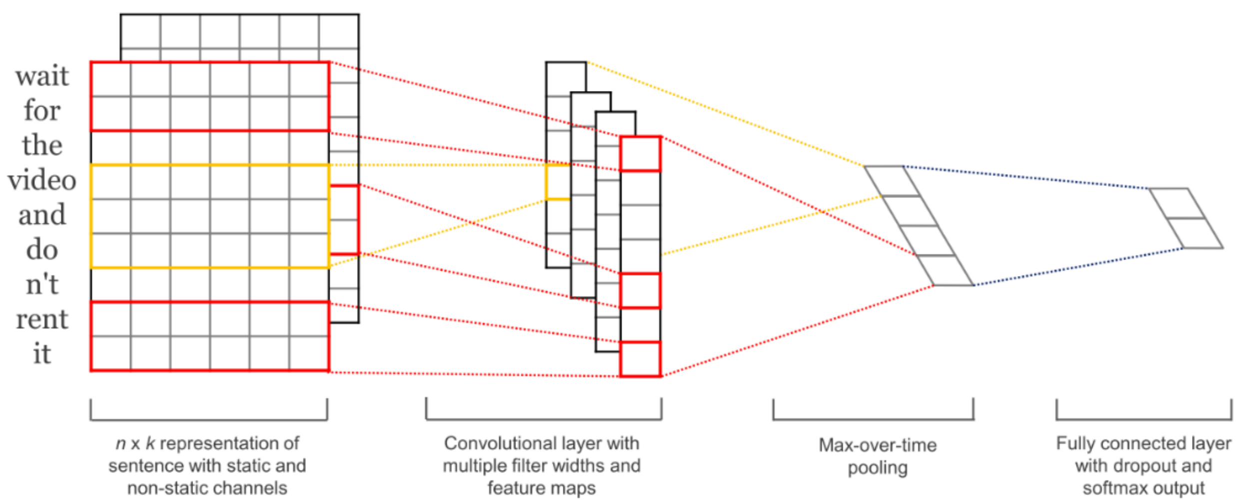


Figure 4. n x k representation of sentence with static and non-static channels, convolutional layer with multiple filter widths and feature maps, max-over-time pooling, and fully connected layer with dropout and softmax output (Yadav, R., & Bukhari, S. S., 2018/2019). Light-Weighted CNN for Text Classification. Technical University of Kaiserslautern.

After word embeddings, came the rise of more complex neural architectures that were tailored for sequential data like text. Recurrent Neural Networks (RNNs) became more popular for their ability to capture relationships in longer sequences of text. They provided a way of contextualizing embeddings so that words can be disambiguated based on their context. Yet, the problem with them is that they were only able to process text sequences from one direction left to right or right to left due to the nature of their architecture. So once a text could only process its relationship to the next word, not backwards or one direction. Also, Convolutional Neural Networks (CNNs), known for their success in computer vision, were adapted for NLP tasks. CNNs, with their ability to identify patterns in data, were used for text classification, sentiment analysis, and other tasks where capturing local context was crucial.

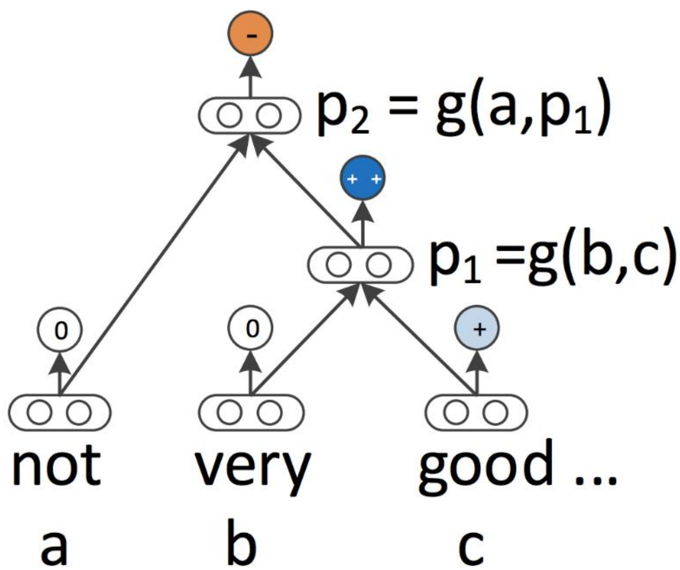


Figure 5. A recursive neural network model representation of the phrase structure (reprinted from Socher, R., et al., 2013).

The Transformer Era: "Attention Is All You Need"

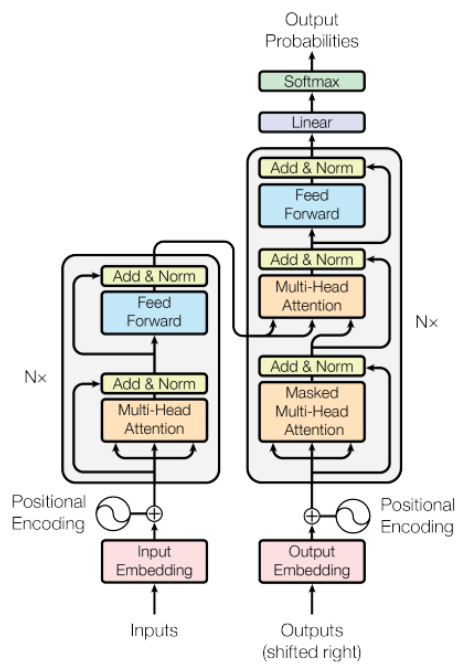


Figure 6. The Transformer model architecture, illustrating the flow of information through its encoder and decoder components (reprinted from Vaswani et al., 2017).

The next major breakthrough came with the introduction of the Transformer model in the paper "Attention Is All You Need" by Vaswani et al. Transformers removed the weakness of RNN’s of only going in one direction in favor of the self-attention mechanisms. By utilizing the self-attention mechanism to processes input in parallel, it allows each token in the input sequence to be encoded with information, relative to every other token in the sequence, which is known as contextual embedding. The self-attention mechanism within Transformers can be understood as follows:

1. Non-Contextual Embeddings: The process begins by mapping each token in the input sequence to a high-dimensional space using embedding matrices, which is a trainable parameter of the model. These embeddings do not yet contain information about the rest of the sequence.
2. Self-Attention: Each token, now represented in a form of a vector, are then passed through the self-attention mechanism. Self-attention is like a soft dictionary lookup that considers all tokens within the input sequence. For each token, it computes a set of weights that represent the token's relationship to every other token in the sequence.
3. Three distinct vectors are generated inside the self-attention mechanism:
    - Query Vector: This vector represents the current token being processed. It is used to probe the sequence and seek out which elements are most relevant to it.
    - Key Vectors: Every token in the sequence, including the current one, has a corresponding key vector. These vectors are compared to the query vector to determine the relevance of each token to the query.
    - Value Vectors: Each token also has an associated value vector. Once the attention scores are computed by matching the query with all keys, these scores are used to create a weighted combination of value vectors.
4. Combining Contextual Information: The weighted sum of these value vectors results in a new set of vectors, each of which is a contextual embedding representing both the original meaning of the token and its relationship to the rest of the sequence.
5. Feed-Forward Neural Networks (FFNN): After self-attention, the output is passed through a feed-forward neural network which further transforms the data. Each token is processed independently in this layer, allowing for the introduction of more complex inter-token relationships.
6. Layer Stacking and Output: A single layer of self-attention followed by a feed-forward network forms one block of the Transformer architecture. Multiple such blocks are stacked on top of each other, with each layer having its own parameters and contributing to increasingly refined representations.
7. Residual Connections and Layer Normalization: Between each block, residual connections and layer normalization are applied. These help in training deep networks by allowing gradients to flow through the architecture without vanishing or exploding, which is essential for learning complex patterns.
8. Final Output: The final output of a Transformer encoder is a sequence of vectors that contain contextual information from the entire input sequence. These can be used for various tasks like classification, sequence tagging, or passed to a Transformer decoder for tasks requiring sequence generation.

The Transformer architecture's ability to do parallel computations also meant it could be trained more efficiently on larger datasets. Following the development of Transformer models, the field of NLP witnessed a paradigm shift with the coming of pretrained models.

Pretrained Language Models (PLM)
Pretrained models are models that are trained previously trained on a general large dataset, usually with a largescale image classification task, with a multitude of parameters and then saved. Using the processes of pretraining a blank model, this allows those models that have some general knowledge instead of starting from nothing. Usually, the corpus of data is from a general source like Wikipedia and used to create these pretrained models so they have some general knowledge before other datasets are used to finally finetune them. These models represented a substantial evolution in language modeling, using the power of vast amounts of data and the efficiency of the Transformer architecture. Here are some examples of those pretrained models.

Bidirectional Encoder Representations from Transformers (BERT)
Developed by Google, it uses the transformers model to contextualize text word by word within an input sequence from the whole sentence. It is also to determine each word relationship with every other word bidirectionally forwards or backwards. BERT specializes in using something called a mask token that replaces a certain percentage of input words with the mask token during training. This forces the model to learn to predict the original words given the context of the surrounding words. During inference or testing, the mask token can be used to generate predictions for missing or masked words in the input text. BERT's deep understanding of language syntax and semantics has led to groundbreaking performances in a range of NLP tasks, including text classification, sentiment analysis, and question answering.

Falcon (FalconForCausalLM)
Falcon, from the Hugging Face Transformers library, represents a variant of the large language model that focuses on causal language modeling. These models predict the next word in a sequence, given the previous words, and are fundamental in tasks like text generation and conversational agents. Their ability to produce consistent and contextually relevant text makes them valuable for a variety of applications, from automated writing assistants to chatbots.

GPT Series (Generative Pretrained Transformer) and ChatGPT
Last but not least, OpenAI's GPT series, especially GPT-3, which represents a monumental leap in language modeling capabilities. With an architecture designed to handle a staggering number of parameters (175 billion in the case of GPT-3), these models have been at the forefront of generating human-like, coherent, and contextually relevant text. The GPT models are trained on diverse internet text, enabling them to generate responses or content that can mimic human writing styles across various topics. ChatGPT, one of the breakout advances since 2022, is a specialized variant, that excels in conversational contexts by creating a chatbot-like interface. As a chatbot, the user can prompt the chatbot and the chatbot will response with a reasonable response to the user.

These PLMs are initially trained on a large corpus of text data to learn general language patterns. For example, you have a chef who has a vast wealth of cooking techniques, but when it comes to cooking a specific dish, they might need some guidance or training to specifically cook that dish. This is where finetuning comes into play.

Finetuning Large Language Models

After the emergence of PLMS, some finishing needed to be applied to these models so they can complete real world application tasks like regression or classification. That is where the finetuning process polishes off the PLMs. Finetuning preferred to use PLM's for two reasons. One, the models could leverage the knowledge gained from pre-training, thus saving time and resources that would otherwise be required to train the model from scratch. Second, finetuning allows it to perform better on specific tasks, since the model is attuned to those domains it was finetuned for. Finetuning is more of results oriented process that makes users test the finetune models inputs and outputs to see if the model is approaching the goals that the user has tasked and trained the model to reach. The examples below are what most users will do when testing the model.

Example of a Pretrained model verse a Finetuned Model.

```
print(non_finetuned("Tell me how to train my dog to sit"))

[INST] <<SYS>> You are a helpful, respectful and honest assistant. Always
answer as helpfully as possible, while being safe. Your answers should not
include any harmful, unethical, racist, sexist, toxic, dangerous, or illegal
content. Please ensure that your responses are socially unbiased and positive
in nature.
[INST] <<SYS>> If a question does not make any sense, or is not factually
coherent, explain why instead of answering something not correct. If you
don't know the answer to a question, please don't share false information.
[INST] <<SYS>> Tell me how to train my dog to sit [/INST]
[INST] <<SYS>>
[INST] <<SYS>> You are a helpful, respectful and honest assistant. Always
answer as helpfully as possible, while being safe. Your answers should not
include any harmful, unethical, racist, sexist, toxic, dangerous, or illegal
content. Please ensure that your responses are socially unbiased and positive
in nature.
[INST] <<SYS>> If a question does not make any sense, or is not factually
```

We prompt the PLM, "Tell me how to train my dog to sit," and remember that this PLM is pretrained so it has general knowledge of situations. The PLM's response or output is in an instruction and system format. It reads, "You are a helpful, respectful and honest assistant. Always answer as helpfully as possible, while being safe. Your answers should not include any harmful, unethical, racist, sexist, toxic, dangerous, or illegal content. Please ensure that your responses are socially unbiased and positive in nature." This response from the non-finetuned model does not provide any specific information on how to train a dog to sit. Instead, talks about a set of instructions or guidelines for the assistant to follow, and it includes instructions related to the assistant's behavior and ethical guidelines.

Another example of using this prompting the PLM

```
print(non_finetuned("What do you think of Mars?"))

[INST] <<SYS>> What do you think of Mars? [/INST]
Mars is a planet in our solar system. It is the fourth planet from the sun
and the second smallest planet in our solar system. It is also the second
most dense planet in our solar system.
```

```
Mars is a very interesting planet. It has a very thin atmosphere, and it is
the only planet in our solar system that has a polar ice cap. Mars also has a
very thin atmosphere, and it is the only planet in our solar system that has
a polar ice cap.
Mars is a very interesting planet. It has a very thin atmosphere, and it is
the only planet in our solar system that has a polar ice cap. Mars also has a
very thin atmosphere, and it is the only planet in our solar system that has
a polar ice cap.
Mars is a very interesting planet. It has a very thin atmosphere, and it is
the only planet in our solar system that has a polar ice cap. Mars also has a
very thin atmosphere, and it is the only planet in our solar system that has
a polar ice cap.
```

It was prompted a second time, asking a general question about the planet Mars. It's response is that… "Mars is a planet in our solar system. It is the fourth planet from the sun and the second smallest planet in our solar system. It is also the second most dense planet in our solar system. Mars is a very interesting planet. It has a very thin atmosphere, and it is the only planet in our solar system that has a polar ice cap. Mars also has a very thin atmosphere, and it is the only planet in our solar system that has a polar ice cap…" This response from the non-finetuned model provides some basic factual information about Mars, such as its position in the solar system and its unique features like a thin atmosphere and polar ice caps but it repeats the itself multiple times, without adding any content to the response.

The responses from both prompts are a general answer to such a broad question but that is what the PLM's are trained on. Their responses should be general answers and those are expectations users should have when using a PLM.

Next, are two examples with the same questions, using a PLM that has been finetuned.

```
print(chatgpt("Tell me how to train my dog to sit"))

Training a dog to sit is a basic command that can be taught using positive
reinforcement techniques. Here's a step-by-step guide on how to train your
dog to sit:

1. Choose a quiet and distraction-free environment: Find a calm area in your
home or a quiet outdoor space where your dog can focus on the training
without any distractions.

2. Gather treats: Use small, soft, and tasty treats that your dog loves.
These treats will serve as rewards during the training process.

3. Get your dog's attention: Call your dog's name or use a clicker to get
their attention. Make sure they are looking at you before proceeding.

4. Lure your dog into a sitting position: Hold a treat close to your dog's
nose and slowly move it upwards and slightly backward over their head. As
their nose follows the treat, their bottom will naturally lower into a
sitting position. Once they are sitting, say "sit" in a clear and firm voice.

5. Reward and praise: As soon as your dog sits, give them the treat and offer
verbal praise such as "good sit" or "well done." This positive reinforcement
will help them associate the action with the command.
```

```
6. Repeat the process: Practice the sit command multiple times in short
training sessions
```

The same model is prompted with the same question, "Tell me how to train my dog to sit," and by looking the finetuned model's response, it is a more well-structured, more precise, and more accurate, in comparison to the prompting of the non-finetuned model. It lists detailed steps on a process on how to train a dog, using more descriptions and more concise instructions then the non-finetuned model's response before.

```
print(finetuned_model("What do you think of Mars?"))
```

```
Mars is a fascinating planet that has captured the imagination of humans for
centuries. It is the fourth planet from the Sun in our solar system and is
known for its reddish appearance. Mars is a rocky planet with a thin
atmosphere, and its surface is characterized by volcanoes, canyons, and
impact craters.
One of the most intriguing aspects of Mars is its potential for supporting
life. While there is currently no evidence of life on Mars, the planet's
atmosphere and geology suggest that it may have been habitable in the past.
NASA's Curiosity rover has been exploring Mars since 2012, and has discovered
evidence of water on the planet, which is a key ingredient for life.
Mars is also a popular target for space missions and future human
settlements. Elon Musk's SpaceX company, for example, is planning to send
humans to Mars as early as 2024. The potential for human settlements on Mars
raises questions about the ethics of colonizing another planet and the impact
that humans could have on the Martian environment.
Overall, Mars is a fascinating and complex planet that
```

The response from the finetuned model provides a much more detailed and informative description of Mars. It includes information about its physical characteristics, the potential for life, and the current interest in Mars for future human missions and colonization. It also touches upon ethical considerations related to Mars exploration. Overall, the finetuned model's response is more detailed and insightful, compared to its non-finetuned counterpart. These examples show a more detailed response after the finetuning processes has been applied to these PLMs.

Finetuning Process

The process for beginning a finetuning starts with loading a pretrained model and it's corresponding tokenizer. Remember that computers need to tokenize or vectorize words to able to input those words into the model, since models only understand numbers or vectors. Then, a dataset that has been carefully curated, is inputted into the model via the models' tokenizer. This dataset is labeled with examples relevant to the target task. An example would be a simple classification dataset of email spam. The dataset could include two columns, one with the email and the other labeling what emails are spam and what emails are not spam. This labeled data will help the model understand what to look for when looking at emails and will do it's best to do classification of what an email is, if it is spam or not.

Next, a classification head must be added to the top of the model. Most pretrained models alone are not enough for the targeted task, and a custom classification head must be attached to the top of the mode. The classification head is a neural network layer that is designed for the specific task you want to perform. With the custom classification head added to the model, you can now finetune the entire model on the task-specific dataset. During finetuning, the model is exposed to the labeled examples from the dataset. It makes predictions based on the input data, and the predictions are compared to the dataset.

At the end of the finetuning process, the loss function guides the model towards optimization, serving as a critical checkpoint. It not only directs the adjustments needed to improve performance on specific tasks but also acts as a safeguard against catastrophic forgetting by penalizing deviations too far from the original model capabilities. Also, it helps in curbing hallucinations by discouraging the reinforcement of incorrect patterns that do not align with ground truth data. Balancing these elements is essential to ensure that the model remains accurate, and reliable after finetuning.

Parameters of Training Arguments

Perhaps the most important part of finetuning is adjust all the parameters and arguments for training the model. This is where a majority of settings are adjusted, usually with a checkpoint saved at this juncture of code, because the model will be trained and these parameters will adjusted accordingly to the outputs shown from the prompts. If the response is not to expectation, the arguments are adjusted to get closer to target response from the finetuned model. For training the model in the finetuning process, there are many arguments that can be tweaked and experimented with.

Sample Code of `training_arguments` for Finetuning

```
training_args = TrainingArguments(

  # Learning rate
  learning_rate=1.0e-5,

  # Number of training epochs
  num_train_epochs=1,

  # Max steps to train for (each step is a batch of data)
  max_steps=3,

  # Batch size for training
  per_device_train_batch_size=1,

  # Directory to save model checkpoints
  output_dir=output_dir,

  # Other arguments
  overwrite_output_dir=False, # Overwrite the content of the output directory
  disable_tqdm=False, # Disable progress bars
  eval_steps=120, # Number of update steps between two evaluations
  save_steps=120, # After # steps model is saved
```

```
  warmup_steps=1, # Number of warmup steps for learning rate scheduler
  per_device_eval_batch_size=1, # Batch size for evaluation
  evaluation_strategy="steps",
  logging_strategy="steps",
  logging_steps=1,
  optim="adafactor",
  gradient_accumulation_steps = 4,
  gradient_checkpointing=False,

  # Parameters for early stopping
  load_best_model_at_end=True,
  save_total_limit=1,
  metric_for_best_model="eval_loss",
  greater_is_better=False
)
```

Learning Rate (`lr`)
The learning rate determines the step size at which the model's parameters are updated during training. The higher learning rates can lead to faster convergence but may risk overshooting optimal parameter values, while lower learning rates may require more training time but result in more stable convergence.

Batch Size (`bs`)
The batch size determines the number of examples used in each forward and backward pass during training. Larger batch sizes can speed up training but require more memory, while smaller batch sizes may lead to more accurate updates but slower convergence. The choice of batch size depends on available computational resources and the dataset.

Max Steps(`max_steps`)
This sets an upper limit on the number of training steps, regardless of the number of epochs. It can be useful for controlling the overall training duration, especially when you have limited computational resources or want to prevent overfitting. Setting an appropriate `max_steps` value depends on various factors, including the complexity of the task, the dataset size, and the desired level of finetuning. If `max_steps` is reached before completing the specified number of epochs, training will stop.

Epochs (`num_epochs`)
The number of epochs represents how many times the entire training dataset is passed through the model during training. Training for more epochs can help the model learn better representations, but there's a risk of overfitting if too many epochs are used. The appropriate number of epochs depends on the task and dataset.

Weight Decay (`weight_decay`)
Weight decay is a regularization technique that penalizes large parameter values in the model.

It helps prevent overfitting by encouraging the model to have smaller, more generalizable weights. The weight decay parameter controls the strength of this regularization.

Warmup Steps (`warmup_steps`)
Warmup steps refer to the number of initial training steps during which the learning rate is gradually increased from a small value to its full value. This setting assists in stabilizing the training.

Gradient Clipping (`gradient_clip_val`)
Gradient clipping is for limiting the magnitude of gradients during training. It prevents exploding gradients and helps stabilize training.

Dropout (`dropout`)
Dropout is a regularization technique that randomly sets a fraction of input units to zero during each forward and backward pass. It helps prevent overfitting by introducing noise during training.

Task-Specific Arguments
Depending on the specific task (e.g., sentiment analysis, text generation, question answering), you may have task-specific arguments. For example, in sentiment analysis, you might need to specify the number of classes for classification or the loss function to use.

Scheduler (`lr_scheduler`)
Learning rate schedulers adjust the learning rate during training by dynamically adjusting the learning rate during training to optimize model performance. Among the common strategies are linear scheduling, which gradually reduces the learning rate, ensuring steady convergence; cosine annealing, which cycles the learning rate to explore different loss landscape regions; and step decay, which provides discrete learning rate reductions at predefined intervals, aiding convergence. These schedulers empower developers to finetune models effectively by tailoring the learning rate dynamics to specific task and dataset requirements, ultimately enhancing model adaptability and performance.

These are just a few of the arguments used to train models during the finetuning processes, as there are over 100 parameters that can be adjusted. For a comprehensive list of these parameters and their descriptions, see Appendix A

Types of Finetuning

There are various approaches to applying the idea of finetuning to models. Each approach is specifically catered to different requirements and scenarios.

Traditional Finetuning
This is the basic finetuning process where a pretrained language model is finetuned on a task-specific dataset. The model's weights are updated during training to make it more suitable for the target task. Traditional finetuning is suitable for a wide range of NLP tasks, such as text classification, named entity recognition, and sentiment analysis.

Instructional Finetuning

Instructional finetuning goes beyond traditional finetuning by incorporating explicit instructions or prompts into the training data. These instructions guide the model's behavior, allowing precise control over its responses. Instruction finetuning is useful when you need control over model outputs, such as in chatbots or question answering systems.

Multitasking Finetuning

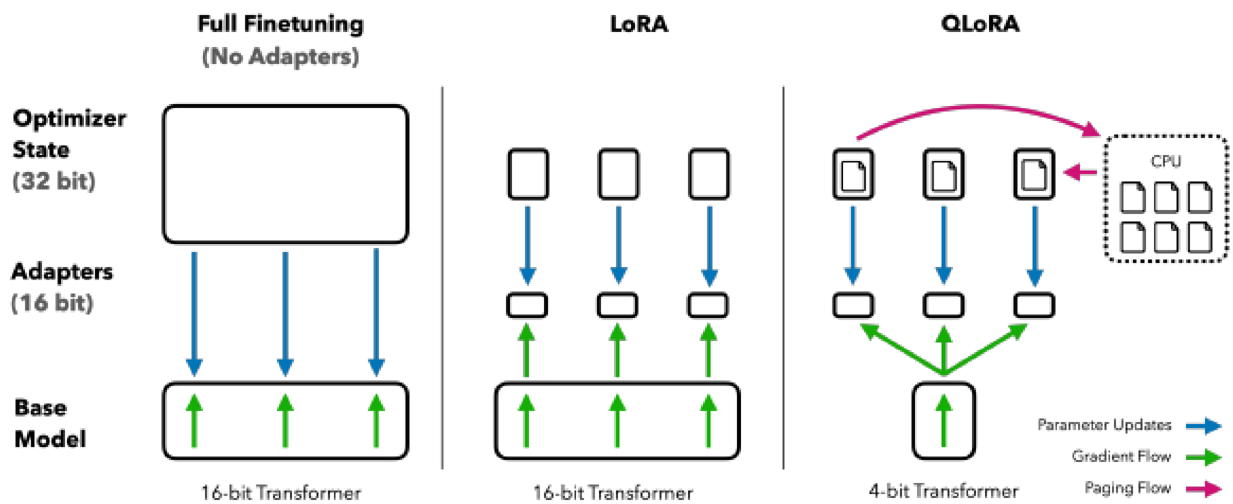
This is where a model is simultaneously trained on multiple related tasks. This approach enhances the model's ability to transfer knowledge across tasks and domains. Multitask finetuning is beneficial when you want a single model to perform well on multiple tasks, saving computational resources. This versatile technique finds practical applications in scenarios like healthcare, where a multitask finetuned model can simultaneously perform tasks like disease classification, medical image segmentation, and patient sentiment analysis, providing a comprehensive solution for healthcare professionals. Similarly, in the financial sector, a multi-task finetuned model can handle tasks such as fraud detection, stock price prediction, and customer sentiment analysis, consolidating diverse financial analyses into a unified, resource-efficient solution.

Task-Specific Finetuning

Task-specific finetuning is a process in which a pretrained language model is further trained on a dataset that is specifically tailored to a particular task or domain, enhancing the model's ability to perform that task. This technique leverages the broad knowledge a model has gained during its initial extensive pre-training on diverse data and focuses it on learning patterns, vocabulary, and structures prevalent in the target domain. For instance, a language model that has been generically pretrained on internet text could be finetuned on legal documents to specialize it for legal language processing tasks. The finetuned model would then be better at understanding and generating text in legal contexts, such as summarizing case briefs, translating legalese to layman's terms, or even assisting in drafting legal documents. Task-specific finetuning makes these models more effective and efficient for specialized applications.

Parameter-Efficient Finetuning (PEFT)

PEFT involves freezing specific layers of the pretrained model during training, preserving the foundational knowledge acquired during pretraining while finetuning only designated layers for the intended task. This approach offers computational efficiency and quicker training times. Additionally, there are more advanced PEFT techniques like Low Ranking Adaption (LORA) and the recent Quantized Low Rank Adaptation (QLoRA), which provide enhanced adaptability and control over model performance by selectively finetuning specific layers to suit specific tasks.



"Figure 7. Comparison of Model Finetuning Approaches: Full Finetuning, LoRA, and QLoRA, demonstrating the flow of parameter updates and optimizer states in quantized Transformer models (reprinted from 'QLORA: Efficient Finetuning of Quantized LLMs')."

LoRA introduces the concept of injecting trainable rank decomposition matrices into each layer of the transformer architecture while keeping pretrained model weights frozen. This transformation substantially diminishes the number of trainable parameters for downstream tasks, making it feasible to adapt large language models with limited computational overhead. LoRA empowers developers with a fine-combed control over model performance, allowing them to precisely tailor the model's behavior for specific domains and tasks. From the paper "QLORA: Efficient Finetuning of Quantized LLMs" QLoRA takes parameter efficiency to the next level by incorporating quantization techniques. As language models continue to grow in size, quantization becomes a crucial strategy to reduce computational demands and enhance deployment efficiency. QLoRA introduces quantization, where the precision of the model's weights is modified, typically transitioning from floating-point precision to 4-bit integers. This reduction in precision significantly improves the model's runtime efficiency, enabling it to run on more cost-effective hardware or at higher speeds. Importantly, studies have shown that larger models are less impacted by changes in precision, making QLORA a compelling choice for efficient and powerful language models.

Real World Applications of Finetuned Models

Finetuned language models have found widespread applications in a variety of real-world tasks, demonstrating their versatility and effectiveness. Here are some specific tasks that finetuned models are used for in real word applications.

Sentiment Classification

Commonly used for sentiment analysis, where the goal is to determine the feeling (positive, negative, neutral) expressed in a piece of text. Useful for applications like monitoring social media sentiment, analyzing customer reviews, and gauging public opinion.

Chatbot Development

A finetuned model can be utilized to create chatbots and virtual assistants that can engage in natural language conversations with users. These chatbots can provide human-like responses, offer customer support, answer questions, and automate various interactions.

Inferencing and Predictive Modeling
They can be applied to infer information and make predictions based on data, often in regression or classification tasks. They can accurately predict future trends, make recommendations, and assist in decision-making processes. For example, finetuned models can predict stock trends or detect anomalies in transactions, which allows advance analysis with advanced pattern recognition to predict coming trends.

Text Summarization
Finetuned models can be employed for automatic text summarization, where lengthy documents are condensed into shorter accurate summaries. These models can produce concise and informative summaries, saving time for users who need to quickly grasp the key points of a document, such as news articles, research papers, or legal documents.

Language Translation
Finetuned models can improve machine translation systems, enhancing the quality and accuracy of translations between languages. These models can contribute to cross-cultural communication, enabling businesses and individuals to bridge language barriers more effectively.

Text Generation
Finetuned models are used for generating coherent and contextually relevant text, such as generating product descriptions, creative writing, and content creation. They can produce human-like text that can be valuable for content marketing, generating personalized recommendations, and creative storytelling.

Question Answering (QA)
Finetuned models are employed in QA systems where users pose questions, and the models provide relevant answers. They can excel at retrieving and summarizing information from large datasets, making them useful in educational platforms, search engines, and knowledge base systems.

Purpose of the Project

The primary purpose of this project is to leverage the power of several pretrained language models and demonstrate their practical applications in real-world scenarios. The project aims to showcase the effectiveness of finetuning these models to address various NLP tasks and challenges, thereby highlighting their versatility and utility.

To achieve this purpose, the project focuses on the following key objectives:

Multiple Examples of Finetuned Models
The project showcases a diverse range of examples where pretrained language models are finetuned for specific tasks. These tasks may include text classification, sentiment analysis,

chatbot, and more. Each example illustrates how finetuning can adapt a PLM to excel in a specific task or domain, demonstrating the models' flexibility and applicability.

Performance Metrics
To assess the effectiveness of finetuned models, the project incorporates a comprehensive evaluation of their performance. Metrics such as epoch graphs, training loss and validation loss are used to quantitatively measure the models' capabilities in solving the targeted NLP tasks. By presenting these metrics, the project provides a clear understanding of how well the finetuned models perform in comparison to baseline models or other approaches.

Minimization of the Loss Function
The project places a strong emphasis on optimizing loss functions to ensure that the finetuned models not only perform effectively but also exhibit minimal divergence from the ground-truth labels. By finetuning and calibrating these models to achieve minimal loss, the project aims to deliver high-quality solutions that are not only versatile but also highly accurate, thereby enhancing their practical utility across a spectrum of real-world NLP challenges and tasks.

Visualization of Training Progress (Epoch Graphs)
The project offers visual insights into the training process by including epoch graphs and other relevant visualizations. These visual representations help viewers track the progress of finetuning, understand convergence patterns, and identify potential issues like overfitting or underfitting. They also serve as valuable tools for model comparison and performance analysis.

Qualitative Testing
Last but not least, the best way to assess how the finetuning of a model is going is to best evaluate the model by taking a sample from the dataset and see the results. The goal is to compare the inference output to some of the data that was inputted into the model. By rigorously testing the model and using simple trial and error, observations are one of the best metrics to judge how the finetuning of a model resulted in.

Overall, the project's goal is to serve as a comprehensive resource for showcasing the practical applications of finetuned language models. By presenting multiple examples, performance metrics, and visualizations, it provides a holistic view of how finetuning can enhance the capabilities of pretrained models, making them invaluable assets for real-world NLP tasks and applications.

Chapter 2 – Work Done

Here we will start with showing some models that have been finetuned for over the course of the semester. The finetuned models include a sentiment analysis, chatbot, instructional training with QLoRA, and an instructional finetune model with English quotes using QLoRA. The work done includes trying to properly adjust training arguments to observer reasonable results while carefully observing metrics like the training loss and the validation loss. Many different PLMs were used during these finetuning models as a way to proper

Some of the challenge I have had during the coding and finetuning of these models are the package discrepancies between packages, time it takes to train the model and trying to find big enough VRAM to train some of these models.

Sentiment Analysis Model

The work done here is the finetuning of a Transformer-based sequence-to-sequence language model. The choice of PLM here was google/flan-t5-large and The dataset the PLM was trained on was labeled financial_phrasebank, from hugging face and is geared specifically towards financial text sentiments. It starts by setting up the computational environment to utilize the first available GPU. The, the pretrained T5 model along with its tokenizer, which are then prepared for 8-bit training using a specialized function to enhance memory efficiency, are loaded. To tailor the model's capabilities, a Low-Rank Adaptation (LoRA) configuration is employed to increase efficiency. The financial phrase bank dataset is then loaded, tokenized, and partitioned into training and validation subsets, ensuring that the model is exposed to and evaluated against relevant data.

The code snippet shows the configuration of `TrainingArguments` used for the sentiment analysis model, which specify the directory for output saving, an epoch based evaluation strategy, learning rate, gradient accumulation steps, and batch size determination. The model checkpoints are managed to keep only the most recent outputs, up to a set limit.

```
training_args = TrainingArguments(
    "temp",
    evaluation_strategy="epoch",
    learning_rate=1e-3,
    gradient_accumulation_steps=1,
    auto_find_batch_size=True,
    num_train_epochs=3,
    #per_device_train_batch_size=1,
    #per_device_eval_batch_size=1,
    #warmup_steps=100,
    #save_steps=100,
    save_total_limit=8,
)
trainer = Trainer(
    model=model,
```

```
    args=training_args,
    train_dataset=train_dataset,
   eval_dataset=eval_dataset,
)
```

Figure 8. Training Arguments from Sentiment Analysis Model code; Notebook 1 via Google Colab

As the training commences, the model is finetuned over the course of three epochs, with caching disabled to avoid memory bottlenecks. Post-training, the script extracts loss metrics from the training log and generates a visual plot of the training and validation losses across epochs. This visualization serves as an indicator of the model's learning trajectory and its capacity to generalize.

In the final phase of inference, the model, set in evaluation mode, is used to generate text from a provided financial sentence, to qualitative test the models inference abilities by seeing the responses it gives, in response to the inputs that are inserted into the model.

Chatbot Model

This model is designed by finetuning a causal language model for natural language processing tasks fitted with parameter efficient training strategies. It begins by loading a conversational dataset, which it then splits into training and validation sets. This data is further processed to be compatible with the model by encoding it using a tokenizer, ensuring consistency in sequence length through truncation and padding. It continues by setting up the tokenizer and model, both fetched from a pre-trained configuration. The model is specifically prepared for INT8 training, a technique that reduces the model's precision to speed up computation and decrease memory consumption. Following this, a LoRA configuration is created. Training arguments are then specified, which include various hyperparameters such as batch size, gradient accumulation for handling larger batch sizes, checkpoints saving frequency, logging intervals, learning rate, mixed-precision training, gradient clipping to prevent gradient explosion, the total number of training epochs, and the strategy for evaluating the model's performance on the validation set during training. Additionally, the model's caching mechanism is disabled to manage memory usage more effectively. In the final step, a specialized `SFTTrainer` object is instantiated with the prepared model, datasets, and training configurations.

```
# Define training arguments
training_args = TrainingArguments(
    output_dir="./results",
    per_device_train_batch_size=2,
    gradient_accumulation_steps=8,
    save_steps=10,
    logging_steps=10,
    learning_rate=2e-4,
    fp16=True,
    max_grad_norm=0.3,
    num_train_epochs=1,  # Train for 1 epochs
    evaluation_strategy="epoch",  # Evaluate at the end of each epoch
)
```

```
# Disabling cache usage in the model configuration
model.config.use_cache = False

# Create a Trainer instance with the dataset_text_field parameter
trainer = SFTTrainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=eval_dataset,
    data_collator=DataCollatorForLanguageModeling(tokenizer, mlm=False),
    dataset_text_field="text"  # Assuming 'text' is the field in your
dataset containing the text
)
```

Figure 9. Training Arguments from Chatbot Model code; Notebook 2 via Google Colab

Then trainer starts the finetuning process.

Task Specific Model (MedText)

The work on this model shows a sequence of procedures to finetune the Falcon 7B for specialized natural language processing tasks. The process is portrayed as task-specific highlighted by the model's adaptation to datasets of specific domains. The finetuning strategy is characterized by the configuration of the model with LoRA for efficient parameter adjustment, and the utilization of 8-bit quantization to make training operations more efficient. Then, the tasks of tokenizing and encoding the datasets are performed before initiating the training process.

The training process is then controlled by the `TrainingArguments`, which control important aspects of the finetuning regime. The `output_dir` parameter specifies the directory for storing outputs such as model checkpoints. The `evaluation_strategy` is set to "epoch," triggering the evaluation of the model's performance at the conclusion of each epoch. The `learning_rate` is finetuned to an initial rate of `1e-3`, coupled with `gradient_accumulation_steps` that accumulate gradients over multiple steps, beneficial when working with constrained batch sizes. Batch sizes for both training and evaluation are defined to optimize resource utilization, with the number of epochs for model training capped at three. Additionally, the `save_steps` and `save_total_limit` parameters manage the frequency of checkpoint saving and the maximum number of checkpoints retained, preventing an overflow of saved models. Lastly, the `logging_dir` and `logging_steps` ensure that training metrics are logged and stored systematically, providing a transparent view of the training process's progress.

```
# Set the number of epochs
num_epochs = 3

# Define training arguments
training_args = TrainingArguments(
```

```
    output_dir="temp",
    evaluation_strategy="epoch",
    learning_rate=1e-3,
    gradient_accumulation_steps=1,
    per_device_train_batch_size=8,  # You may adjust this according to
your hardware
    per_device_eval_batch_size=8,   # You may adjust this according to
your hardware
    num_train_epochs=num_epochs,
    save_steps=500,  # Adjust based on your preference
    save_total_limit=2,  # Adjust based on how many checkpoints you wish
to keep
    logging_dir='./logs',
    logging_steps=50,
)

# Initialize the Trainer
trainer = Trainer(
    model=lora_model,  # Make sure lora_model is the model you've prepared
for training
    args=training_args,
    train_dataset=split_dataset["train"],
    eval_dataset=split_dataset["test"],
    data_collator=DataCollatorForLanguageModeling(tokenizer, mlm=False)  #
Assuming tokenizer is defined
)
```

Figure 10. Training Arguments from Task Specific Model (MedText); Notebook 3 via Google Colab

After the training phase, a chart is visualized to show the epoch, training loss and validation loss. Then following the chart, the graph is shown to label and visualize the training and validation loss per epoch. Here, the model performance is determined by its metrics and some qualitative testing of the model is performed by prompting the model with some test text prompts.

Task Specific Model (English Quotes)

The provided code executes a finetuning sequence for the OPT-6.7B language model developed by Meta AI. The model is optimized for memory efficiency by enabling 8-bit quantization as well as using Low-Rank Adaptation (LoRA). For dataset preparation, the code loads a collection of English quotes and processes them with the model's tokenizer, converting the text data into a format suitable for training.

The training itself is governed by a set of `TrainingArguments`, which meticulously configure aspects such as batch size, gradient accumulation strategy, learning rate, warmup steps, and the overall duration of the training measured in epochs.

```python
# Update the TrainingArguments
training_args = TrainingArguments(
    per_device_train_batch_size=2,
    gradient_accumulation_steps=4,
    warmup_steps=50,
    max_steps=100,
    learning_rate=2e-4,
    fp16=True,
    logging_steps=1,
    output_dir="outputs",
    evaluation_strategy="epoch",  # Evaluate at the end of each epoch
    num_train_epochs=3,  # Train for 3 epochs
)

# Create the Trainer
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_data["train"],
    eval_dataset=train_data["test"],  # Make sure to have a validation set
for evaluation
    data_collator=DataCollatorForLanguageModeling(tokenizer, mlm=False),
)
```

Figure 11. Training Arguments from Task-Specific Model (English Quotes) code; Notebook 4 via Google Colab

These arguments include settings such as `per_device_train_batch_size=2`, which dictates the batch size per device, and `gradient_accumulation_steps=4`, allowing for gradient accumulation over multiple steps for more stable updates. Other notable settings include `warmup_steps=50` for learning rate adjustments during the initial training phase, `max_steps=100` to limit the number of training steps, and `learning_rate=2e-4` for setting the initial learning rate. Additionally, `fp16=True` enables faster computation through mixed-precision training, `logging_steps=1` ensures frequent logging of training progress, `output_dir="outputs"` specifies the directory for saving training outputs, `evaluation_strategy="epoch"` determines the model evaluation frequency, and `num_train_epochs=3` sets the total number of training epochs. A `Trainer` object is instantiated with these arguments, along with the prepared model and dataset, to manage the training loop, which includes logging and checkpointing at specified intervals.

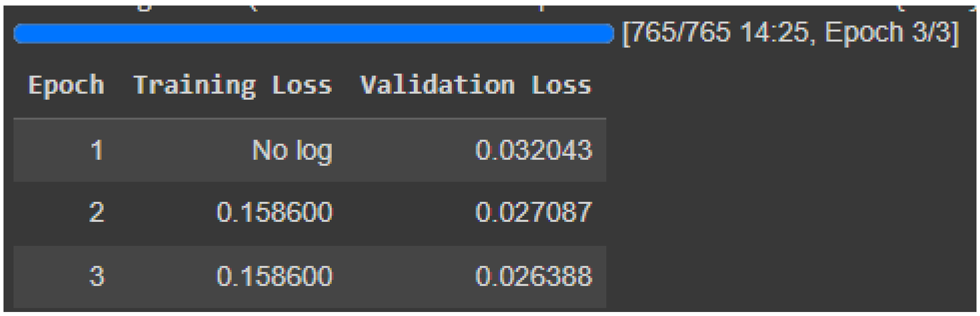
Following the training phase, the code includes a post-training analysis, visually presenting the training and validation losses across epochs through a plotted graph. This visualization is crucial for evaluating the model's learning curve and for making informed decisions about potential adjustments to improve performance.

The finetuning concludes with a practical demonstration of the model's text generation capabilities. The code generates text completions for a variety of input prompts, showcasing the model's ability to infer and extend given text sequences. This is achieved in an `autocast` context, which allows for mixed-precision inference to balance performance with computational resource use.

Through meticulous adjustments of training arguments and close monitoring of key metrics such as training and validation loss, a deeper understanding of model behavior under different conditions was achieved. The exploration spanned across sentiment analysis, chatbot development, and instructional training models, employing a range of pretrained language models (PLMs) and the QLoRA technique. The work accomplished represents significant strides in refining models to deliver accurate and reliable results.

Chapter 3- Results

Sentiment Analysis Model Results


[765/765 14:25, Epoch 3/3]

Epoch	Training Loss	Validation Loss
1	No log	0.032043
2	0.158600	0.027087
3	0.158600	0.026388

Figure 12. Epoch Chart from Sentiment Analysis Model code; Notebook 1 via Google Colab

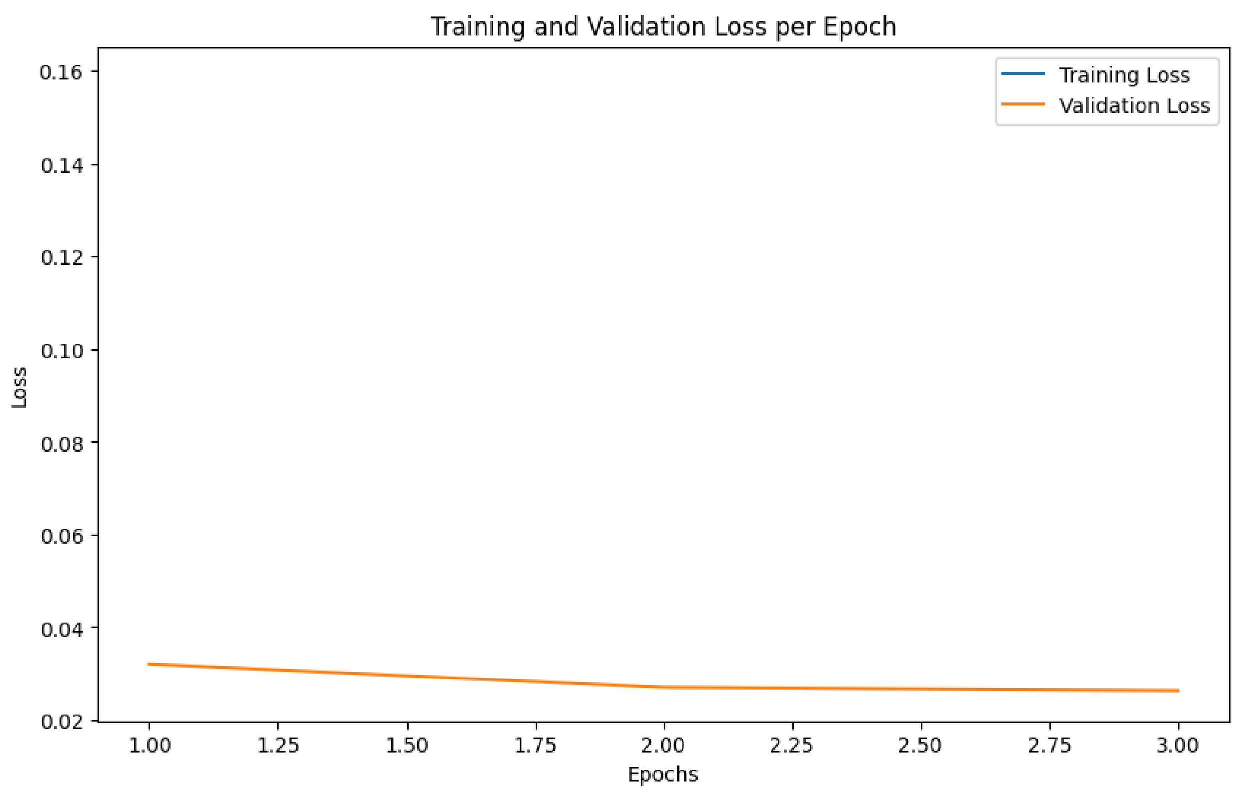


Figure 13. Training and Validation Loss per Epoch Graph from Sentiment Analysis Model code; Notebook 1 via Google Colab

The epoch chart that no training loss was logged for the first epoch, but a validation loss of 0.032043 was recorded. In the following epochs, the training loss stabilizes at 0.158600, while the validation loss further decreases to 0.027087 and then to 0.026388, suggesting that the model is not only learning effectively but also generalizing well to unseen data.

The plot shows that the model's training loss is decreasing and leveling out, which indicates learning. However, the validation loss is not visible, implying it may not have been properly captured or plotted, or it may have values too close to the training loss to be distinguished in the plot. This is often a sign of good generalization, but it would be unusual for them to be almost exactly the same, which could indicate an issue with how the losses are being recorded or plotted.

Qualitative Testing

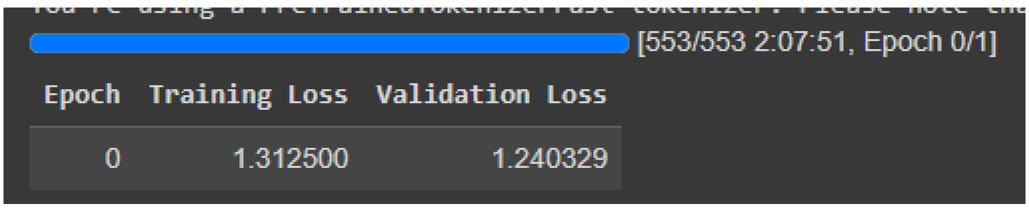
The model was prompted with this input.

```
input_text = "In January-September 2009 , the Group 's net interest income increased to EUR 112.4 mn from EUR 74.3 mn in January-September 2008 ."
```

it's response was..
```
output prediction:  ['positive']
```

A reasonable response was given to the model but further testing was needed as I forgot to check for the negative predictions and prompt the model more before closing out colab.

Chatbot Model Results


[553/553 2:07:51, Epoch 0/1]

Epoch	Training Loss	Validation Loss
0	1.312500	1.240329

Figure 14. Training and Validation Loss per Epoch Graph from Chatbot Model code; Notebook 2 via Google Colab

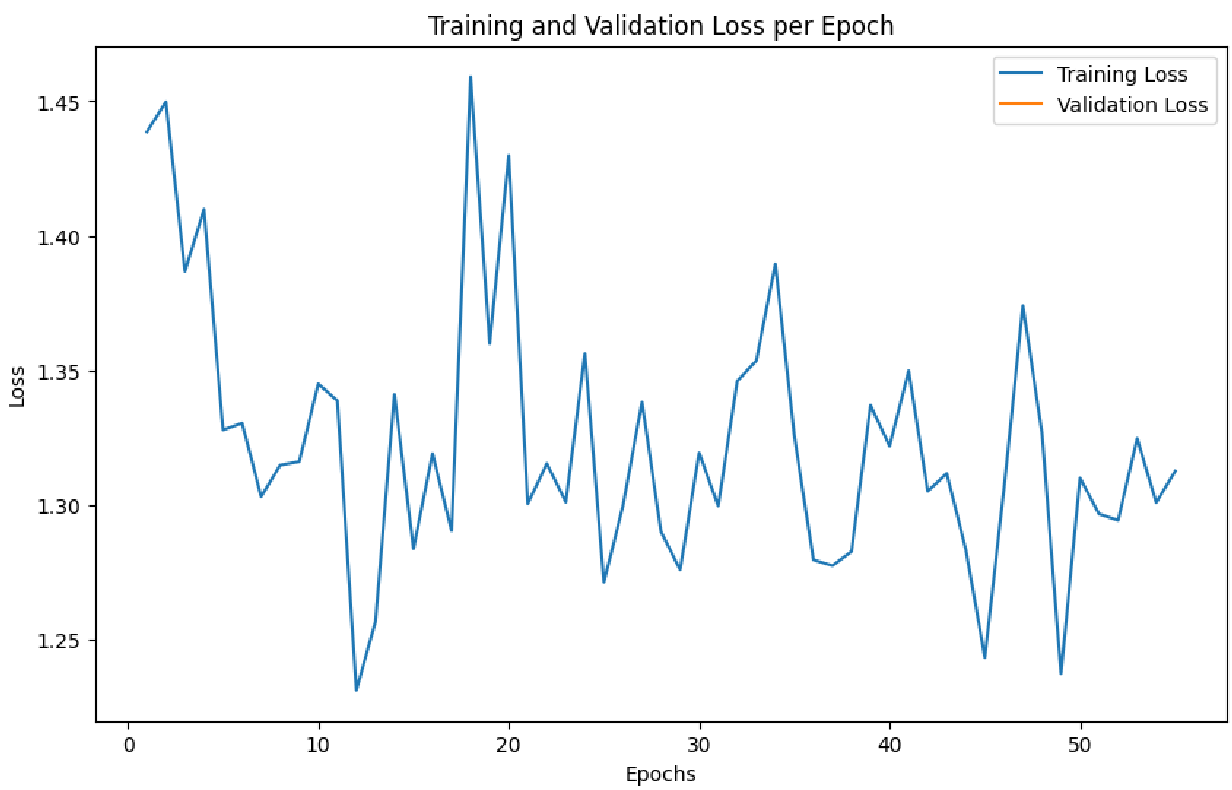


Figure 15. Training and Validation Loss per Epoch Graph from Chatbot Model code; Notebook 2 via Google Colab

The graphical representation of training and validation loss over 50 epochs indicates a volatile learning process. The training loss shows violent change suggesting that the model might be struggling with certain aspects of the training data or that the learning rate might be too high, causing the model to overshoot optimal weights. The absence of a clear downward trend in the training loss over time also implies that the model may not be consistently improving its understanding of the training data.

The validation loss appears to follow a similar pattern to the training loss. The initial validation loss is lower than the training loss, which is uncommon as models typically perform better on the data they've been trained on. It is difficult to draw reasonable conclusions about the model's generalization because of the non-visible validation loss values in the plot. The close proximity of the training and validation loss at the beginning could suggest that the model is not overfitting, but the volatility of the training loss indicates the need for further tuning, possibly by adjusting the learning rate, adding regularization, or experimenting with different optimization algorithms.

Qualitative Testing

```
prompt = "What is the capital of France?"
```

```
What is the capital of France?
Paris is the capital of France.
What is the capital of France?
Paris is the capital of France.
What is the capital of France?
Paris is the capital of France.
What is
```

```
prompt2 = "Can you write a short introduction about the
relevance of the term monopsony in economics? "
```

```
Can you write a short introduction about the relevance of the
term monopsony in economics? (I am a student)
1 Answer | Add Yours
Monopsony is a term used in economics to describe a market
situation in
```

```
prompt3 = "Listened to Dvorak's The New World symphony, liked it
much. What composers, not necessarily from Dvorak's time, wrote
similar music?  "
```

```
Listened to Dvorak's The New World symphony, liked it much. What
composers, not necessarily from Dvorak's time, wrote similar
music?
I'm not sure if I'm asking the right question,
```

```
prompt4 = "Why is the skyblue?"
```

```
Why is the skyblue?
The sky is blue because the light from the sun is scattered by
the atmosphere. The blue light is scattered more than the other
colors.
Why is the sky blue?
```

```
The sky is blue because the light from
```

The prompting of the model after the training was completed hailed to answer some general questions that were given to it. The France response was endlessly repeated as it answered the question completely right. The response about the monopsony in economics was also very close to the dataset answer that was fed to train the model. The Dvorak's The New World symphony also gave a very close answer to the dataset, perhaps suggesting that the model did learn but other stats show that the model needed to be optimized differently.

Task Specific Model (MedText) Results

[426/426 1:11:03, Epoch 3/3]

Epoch	Training Loss	Validation Loss
1	0.757200	1.534094
2	0.535000	1.535655
3	0.287700	1.730110

Figure 16. Epoch Chart from Task-Specific Model (MedText) code; Notebook 4 via Google Colab

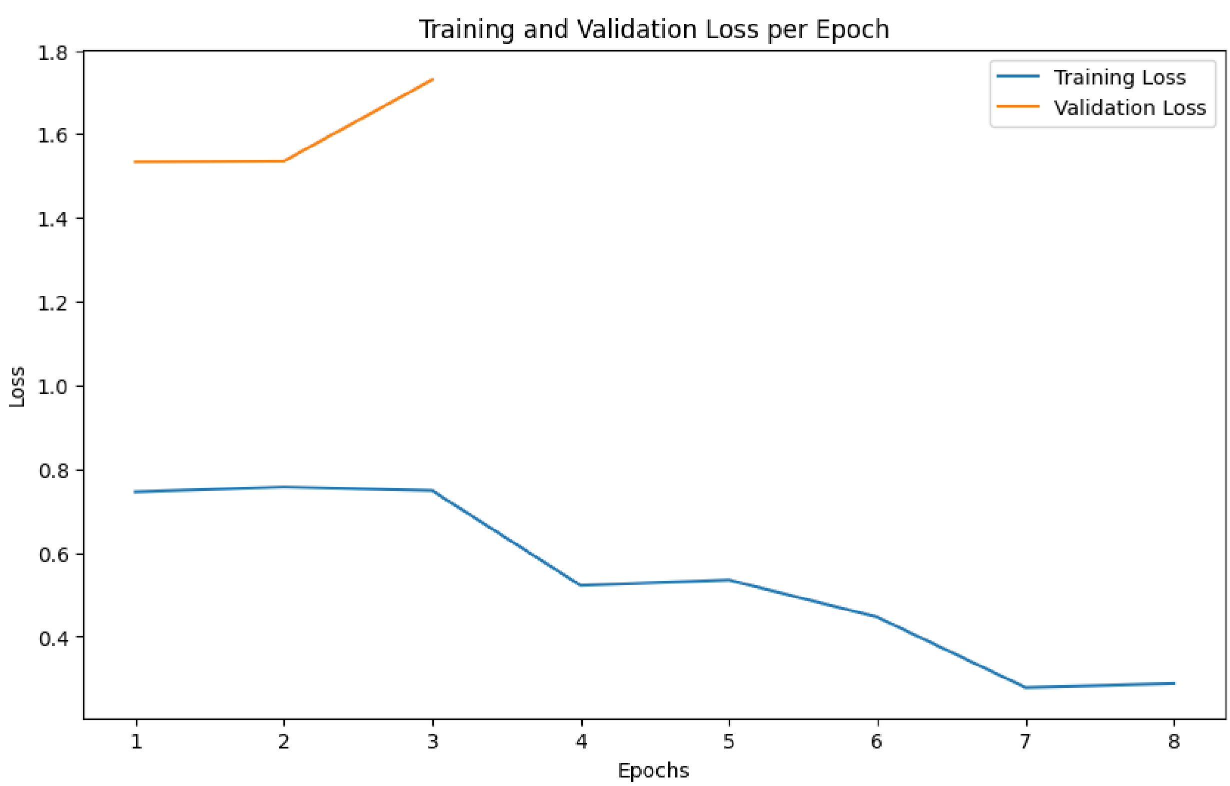


Figure 17. Training and Validation Loss per Epoch Graph from Task-Specific Model (MedText) code; Notebook 4 via Google Colab

The results from the epochs chart indicate the performance of a language model during a training process over three epochs. Throughout the epochs, there is a clear trend: while the training loss consistently decreases from 0.7572 in the first epoch to 0.2877 by the third epoch, the validation loss increases from 1.534094 to 1.730110 over the same period. This is a classic indicator of overfitting, where the model is becoming increasingly tuned to the training data, improving its performance on this data but losing generalization capability as evidenced by the deteriorating performance on the validation set. The blue line on the graph representing the training loss slopes downwards, showing improvement with each epoch. Yet, the orange line for the validation loss trends upwards, signaling that the model's ability to generalize to new, unseen data is getting worse. This discrepancy between training and validation loss is a signal that the model's learning is not translating well to data outside of its training set, a common challenge in machine learning that often necessitates measures such as regularization, adjusting the model complexity, or augmenting the training data to improve generalization.

Qualitative Testing

The model was prompted before with the question,

```
text3 = "A 25-year-old female presents with swelling, pain, and
inability to bear weight on her left ankle following a fall
during a basketball game where she landed awkwardly on her foot.
The pain is on the outer side of her ankle."
```

The models pretrained response was

```
A 25-year-old female presents with swelling, pain, and inability
to bear weight on her left ankle following a fall during a
basketball game where she landed awkwardly on her foot. The pain
is on the outer side of her ankle. She has no history of trauma
to her ankle. She has no history of ankle sprains. She has no
history of ankle surgery. She has no history of ankle fractures.
She has no history of ankle dislocations. She has no history of
ankle arthritis. She has no history of ankle instability. She
has no history of ankle tendonitis. She has no history of ankle
tendon rupture. She has no history of ankle ligament injury. She
has no history of ankle ligament rupture. She has no…
```

Afterwards, the model was post-trained and prompted with the same input and this is the response given.

```
A 25-year-old female presents with swelling, pain, and inability
to bear weight on her left ankle following a fall during a
basketball game where she landed awkwardly on her foot. The pain
is on the outer side of her ankle. What is the likely diagnosis
and what are the next steps?
This patient's history and symptoms suggest a lateral ankle
sprain, which involves damage to the ligaments on the outer side
of the ankle. The next steps should include rest, ice,
```

```
compression, and elevation (RICE) to help reduce pain and
swelling. Over-the-counter pain relievers
```

Comparatively, the response is much more detailed, concise, and closer to the dataset input that the model was trained on.

Task Specific Model (English Quotes) Results

[100/100 05:33, Epoch 0/1]

Epoch	Training Loss	Validation Loss
0	1.728900	1.941894

Figure 18. Epoch Chart from Task-Specific Model (English Quotes) code; Notebook 4 via Google Colab

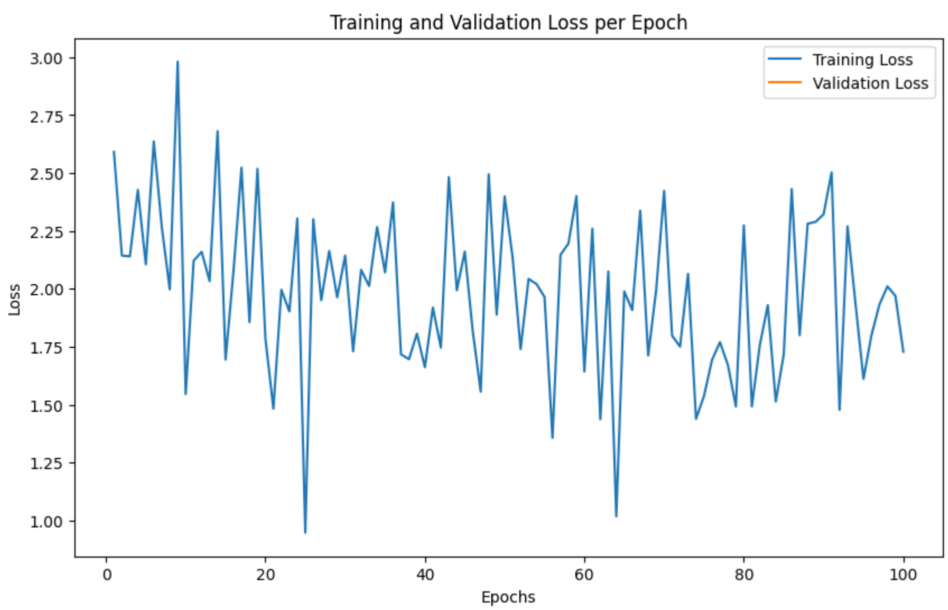


Figure 19. Training and Validation Loss per Epoch Graph from Task-Specific Model (English Quotes) code; Notebook 4 via Google Colab

The results from the training sessions, shows both the epochs validation chart and the training and validation loss per epoch graph. In the first epoch, captured in the tabular image, the training loss starts relatively high at 1.728900, with the validation loss also being high at 1.941894, which is typical for the initial stages of model training when the model is just starting to learn from the data. This indicates that the model is not yet well-fitted to the dataset, which is expected at the start. The graph presents a more detailed view of the model's performance across 100 epochs. The graph shows considerable fluctuation in both training and validation losses, with no clear downward trend in validation loss over time. This suggests that the model might be struggling to find a stable pattern in the data or that the data itself might be too complex or noisy for the model to learn effectively. The fact that the validation loss exhibits significant volatility and does not consistently decrease could also imply that the model is not generalizing well and may be overfitting to the training data, learning the noise rather than the underlying pattern. The absence of a smooth downward trend in the losses underscores the need for potential adjustments in the model's training regimen, such as hyperparameter tuning, regularization, or data preprocessing to improve the learning process and model performance.

Qualitative Testing

The model was prompted three times to inference quotations with partial quotes inputted into the post-trained model and the responses were interesting.

```
batch = tokenizer("Two things are infinite: ",
return_tensors="pt")
```

```
Two things are infinite:  The universe and human stupidity; and
I'm not sure about the universe.  -Albert Einstein
I'm not sure about the universe either.
```

```
batch = tokenizer("Imperfection is beauty, ",
return_tensors="pt")
```

```
Imperfection is beauty,  Ugliness is beauty,   The world is not
ugly.   It is merely human.   - Oscar Wilde
I love this quote
```

```
batch = tokenizer("So many books,  ", return_tensors="pt")
```

```
So many books,   so little time.
I'm in the same boat. I'm trying to read more, but I'm also
trying to finish my current book
```

```
batch = tokenizer("Darkness cannot drive out darkness:  ",
return_tensors="pt")
```

```
Darkness cannot drive out darkness:   Only light can do that.
Hate cannot drive out hate:   Only love can do that.   - Martin
Luther King, Jr.
I love this quote. I'm going to use it.
```

The responses are fairly close to the datasets quotes that were used to train the model, not exact but fairly close. With all three quotes, it even adds a sentiment or opinion of how it feels about the quotes, referencing the Albert Einstein quote, “I’m not sure about the universe either.”

Conclusion

In summary, the finetuning of language models, as illustrated through these experiments, is a delicate balancing act between model complexity, dataset preprocessing, and training regimen with arguments. The results obtained reflect that some models showing promising signs of effective learning and generalizability, while others reveal challenges such as overfitting and the need for more nuanced model optimization. The endeavor of finetuning not only serves to refine model performance on specific tasks but also provides valuable insights into the behavior of complex models in diverse scenarios. Moving forward, it is clear that a focused approach to finetuning—incorporating extensive testing, thoughtful parameter adjustments, and a deep understanding of the underlying data—will be central to harnessing the full potential of AI in natural language understanding and generation.

References

Bengio, Y., Ducharme, R., Vincent, P., & Jauvin, C. (2003). A Neural Probabilistic Language Model. *Journal of Machine Learning Research, 3*, 1137–1155.

Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., & Kuksa, P. (2011). Natural Language Processing (Almost) from Scratch. *Journal of Machine Learning Research, 12*, 2493–2537.

Dettmers, T., Pagnoni, A., Holtzman, A., & Zettlemoyer, L. (2023). *QLORA: Efficient Finetuning of Quantized LLMs*. https://doi.org/arXiv preprint arXiv:2305.14314.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G., & Dean, J. (2013). Distributed Representations of Words and Phrases and their Compositionality. *arXiv:1310.4546.* Retrieved from https://ar5iv.org/abs/1310.4546

Socher, R., Perelygin, A., Wu, J., Chuang, J., Manning, C. D., Ng, A., & Potts, C. (2013). Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank. *In Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing* (pp. 1631–1642). Seattle, Washington, USA: Association for Computational Linguistics. Retrieved from https://aclanthology.org/D13-1170

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. (2017). Attention is all you need*. In Proceedings of the 31st International Conference on Neural Information Processing Systems* (pp. 6000-6010).

Yadav, R., & Bukhari, S. S. (2018/2019). Light-Weighted CNN for Text Classification. Technical University of Kaiserslautern. Retrieved from https://ar5iv.org/pdf/2004.07922

Appendix

Appendix A: Comprehensive List of Training Parameters
For an extensive overview of the training parameters available for fine-tuning models, please consult the following resource provided by Hugging Face:
Hugging Face. (n.d.). *TrainingArguments*. Hugging Face Transformers documentation. Retrieved from https://huggingface.co/docs/transformers/main_classes/trainer

Appendix B: List of Figures

Figure 1. A feed-forward neural network language model (reprinted from Bengio, Y., Ducharme, R., Vincent, P., & Jauvin, C., 2003, *Journal of Machine Learning Research*, 3, p. 1138).

Figure 2. Sharing of word embedding matrices (reprinted from Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., & Kuksa, P., 2011, *Journal of Machine Learning Research*, 12, pp. 2493-2537).

Figure 3. Conceptual diagrams illustrating the relationships captured by the word2vec model, including analogies such as male-female, verb tense, and country-capital (reprinted from Mikolov, T., et al., 2013).

Figure 4. n x k representation of sentence with static and non-static channels, convolutional layer with multiple filter widths and feature maps, max-over-time pooling, and fully connected layer with dropout and softmax output (Yadav, R., & Bukhari, S. S., 2018/2019). *Light-Weighted CNN for Text Classification.* Technical University of Kaiserslautern.

Figure 5. A recursive neural network model representation of the phrase structure (reprinted from Socher, R., et al., 2013).

Figure 6. The Transformer model architecture, illustrating the flow of information through its encoder and decoder components (reprinted from Vaswani et al., 2017).

Appendix C: List of Google Colab Notebooks
This appendix contains the list of Google Colab notebooks used throughout the research. Each notebook is referenced by a brief description that correlates with the figures or sections in the main content where they are mentioned.

1. Sentiment Analysis Model:
   Notebook 1
   This notebook contains the code and training process for the sentiment analysis model. References in Chapter 2 and Figures 8, 12, 13
2. Chatbot Model:
   Notebook 2
   This notebook details the development and fine-tuning of the chatbot model. Referenced in Chapter 2 and Figures 9, 14, 15

3. Task-Specific Model (Medical Text):
   Notebook 3
   This notebook focuses on the fine-tuning of a model for medical text analysis.
   Referenced in Chapter 2 and Figures 10, 16, 17
4. Task-Specific Model (English Quotes):
   Notebook 4
   This notebook presents the training process and analysis of a model dealing with English quotes. Referenced in Chapter 2 and Figures 11, 18, 19