University of Sheffield

Implementing an LLM on Phone

Mokshesh Jain

Supervisor: Dr. Nikolaos Aletras

A report submitted in fulfilment of the requirements for the degree of MSc in Adv. Computer Science

in the

Department of Computer Science

September 12, 2024

Declaration

All sentences or passages quoted in this report from other people's work have been specifically acknowledged by clear cross-referencing to author, work and page(s). Any illustrations that are not the work of the author of this report have been used with the explicit permission of the originator and are specifically acknowledged. I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this project and the degree examination as a whole.

Name:Mokshesh Jain

Signature:

Date: 12-09-2024

Abstract

This project involves the development of an iOS application that integrates a large language model (LLM) to deliver AI-powered text generation and interactive capabilities directly running on a mobile device. The projects main focus lies on optimizing a pre-trained GPT-2 model to be compatible with apple devices, converting it to the Core ML format for on-device inference. This includes custom tokenization, model optimization techniques, and the implementation of sampling strategies like greedy selection and top-k to enhance text generation quality. The result is an efficient, responsive application that showcases the feasibility of running advanced AI models on consumer-grade hardware, offering a unique, personalized experience in natural language processing on mobile platforms.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Powerful language models like GPT [**?**] were developed as a result of noteworthy advancements in the field of artificial intelligence, particularly, in the realm of natural language processing (NLP). Although these models have showcased a remarkable capacity to understand and produce human-like text, we still face significant challenges when it comes to their application on edge devices, such as limitations in computational capacities and memory constraints among other obstacles. Addressing these challenges is vital, especially as concerns about environmental impact and data privacy grow. This paper is a scholarly endeavor to address and bridge this divide by implementing a large language model (LLM) on an iOS platform, enabling on-device text generation without relying on cloud-based resources.This approach allows for high-quality text generation while enhancing privacy by processing data locally, without transmitting sensitive information to external servers.

This project focuses on optimizing large language models (LLM) for iOS, maintaining accuracy while converting the GPT-2 model to Apple's Core ML framework. Key features include custom tokenization, quantization, and advanced text generation techniques like greedy and top-k sampling. The application provides real-time interaction with the model, delivering a smooth AI experience on mobile devices. This project demonstrates the feasibility of deploying sophisticated models on consumer-grade hardware, paving the way for mobile applications in fields like education, productivity, and entertainment without sacrificing privacy or environmental considerations.

1.1 Aims and Objectives

1.1.1 Aims

The primary aim of this project is to successfully implement a large language model (LLM) on an iOS device, enabling real-time, on-device text generation without the need for cloud-based resources. This will demonstrate the feasibility of running sophisticated AI models on mobile devices, thus enhancing the accessibility and portability of AI-powered applications.

1.1.2 Objectives

1. Model Conversion and Optimization: Convert the GPT-2 model to Core ML format and optimize it for iOS devices to ensure efficient performance and memory usage.
2. On-Device Tokenization: Develop or adapt a tokenizer that can efficiently process and manage text data directly on the device.
3. Text Generation Techniques: Implement advanced text generation methods, such as greedy and top-k sampling, to enhance the quality and relevance of the generated text.
4. Performance Testing and Validation: Evaluate the model's performance in terms of speed, accuracy, and memory usage on various iOS devices to ensure a smooth user experience.
5. User Interface Integration: Integrate the model with a user-friendly iOS application, providing a seamless experience for text generation tasks.
6. Documentation and User Guidelines: Create comprehensive documentation and guidelines to assist users in understanding and effectively utilizing the application.

1.2 Overview of the Report

This report details the process of developing an iOS application that leverages a large language model (LLM) for on-device text generation. It begins with an introduction to the significance of LLMs and the challenges of deploying them on mobile devices. The report then covers the steps taken to convert and optimize the GPT-2 model for iOS, including handling tokenization, model compression, and inference. Following this, the report discusses the integration of the model into the app's user interface, ensuring a seamless user experience. Performance evaluation and testing results are presented, highlighting the model's effectiveness on different iOS devices. The report concludes with a summary of the project's outcomes and recommendations for future work, including potential improvements and further applications of on-device LLMs.

Chapter 2

Literature Survey

2.1 Large Language Models (LLMs), Transformer Architecture, and Tokenization

Large Language Models (LLMs) represent a significant breakthrough in natural language processing (NLP) and machine learning. The development of LLMs took a major leap forward in 2017 when Google introduced the Transformer architecture, which became foundational to most modern LLMs. Before the introduction of Transformers, models like Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTM) networks were the most commonly used architectures for NLP tasks. However, both RNNs and LSTMs struggled with long-term dependencies and context retention, particularly in lengthy sequences, due to the sequential nature of their computations .

The Transformer architecture, introduced by Vaswani et al. in their 2017 paper "Attention is All You Need" , revolutionized NLP by eliminating the need for recurrence and convolutions. The key innovation of Transformers is the self-attention mechanism, which allows the model to capture dependencies between words in a sequence regardless of their distance. Self-attention works by comparing each token in a sequence with every other token to calculate the importance, or "attention," of one token in relation to others. This process involves using query, key, and value vectors, where the resulting attention scores help the model focus on important parts of the sequence while downplaying less relevant parts. This enables LLMs to build rich contextual representations of the text, making Transformers highly effective for tasks requiring long-range contextual understanding.

Before self-attention can be applied, the input text must be transformed into a higher - dimensional vector space through tokenization. Tokenization involves breaking the text into smaller units, or tokens, which are then converted into numerical representations suitable for the model. These tokens are encoded into embeddings, dense vectors in a high-dimensional space, enabling the model to grasp semantic meaning before applying multi-head self-attention.

Multi-head self-attention extends the self-attention mechanism by running multiple self-attention processes, called "heads," in parallel. In this mechanism, relationships between tokens in a sequence are computed using query, key, and value vectors. Each head operates

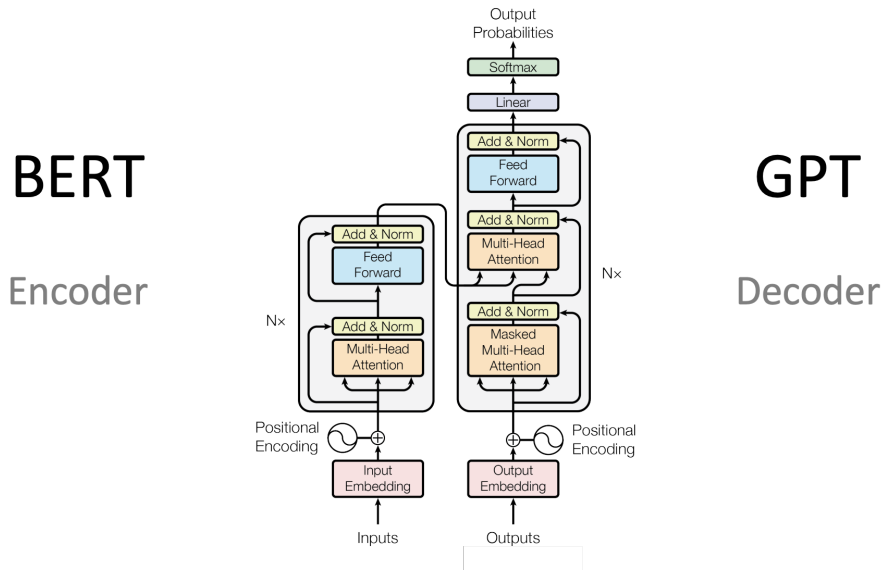

Figure 2.1: Transformer Architecture

independently, focusing on different parts of the sequence or capturing different types of relationships. The dot product between the query and key vectors is normalized, followed by a softmax operation to determine attention weights, which highlight the importance of one token relative to others. By computing multiple sets of attention weights in parallel, multi-head attention captures complex dependencies within the input sequence.

The results from all heads are then concatenated and linearly transformed to produce the final output, enabling the model to build a richer, more diverse contextual understanding of the input by attending to different aspects of the token relationships simultaneously.

Figure : Transformer architecture

One significant advantage of Transformer models over RNNs is their ability to process sequences in parallel, bypassing the sequential bottleneck inherent to RNNs. This parallel processing is crucial for scaling up models to handle more complex tasks such as machine translation and summarization.

2.1.1 Types of Language Modeling in LLMs

LLMs can be categorized based on their pre-training objectives, which shape how they learn and generate text.

1. Autoregressive Language Modeling (ALM) involves training models to predict each subsequent token based on the sequence of preceding tokens. This pretraining task is employed in decoder-only architectures such as GPT-2[9] and GPT-3[3] , where the model learns one token at a time in a left-to-right fashion.

2. Prefix Language Modeling (PLM) involves training models to predict subsequent tokens based on a partial prefix in the input text. This objective is particularly relevant to

encoder-decoder architectures like BART or T5 where PLM is useful for generating text based on a known prefix

3. Masked Language Modeling (MLM) operates by randomly masking tokens within the input sequence and training the model to predict these masked tokens using the surrounding unmasked context. This approach is central to BERT-like models and provides a more comprehensive understanding of context while enabling parallelization during training . Unlike ALM and PLM, MLM allows the model to learn bidirectional context.

2.1.2 Advancement and Growth of LLMs

Since the introduction of the Transformer architecture, Large Language Models (LLMs) have experienced exponential growth in both size and complexity. Early models such as GPT-2 featured around 1.5 billion parameters, but more recent models like GPT-3 and GPT-4 have scaled up to 175 billion parameters. Similarly, BERT (Bidirectional Encoder Representations from Transformers) has models ranging from 110 million parameters (BERT-Base) to 340 million parameters (BERT-Large), while T5 (Text-to-Text Transfer Transformer) ranges from 60 million parameters (T5-Small) to 11 billion parameters (T5-XXL).[11]

Other significant models like PaLM (Pathways Language Model) have taken the scaling trend even further, featuring a massive 540 billion parameters, while LaMDA (Language Model for Dialogue Applications), designed for conversational AI, incorporates 137 billion parameters. This scaling has led to significant improvements in various tasks, including zero-shot learning, question-answering, and language translation, making LLMs increasingly versatile across a wide array of applications.

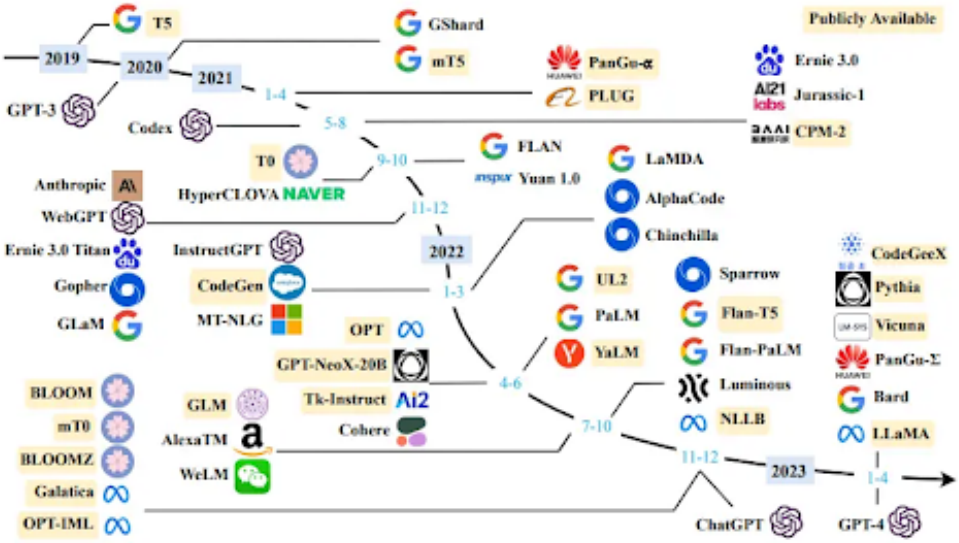

Figure 2.2: Developement of LLM throughout the years

figure : A timeline of existing large language models.[11]

LLMs are pre-trained on vast and diverse datasets, enabling them to learn grammar, world facts, and even reasoning abilities. However, this rapid scaling comes with several drawbacks. One major issue is the increased computational demands, as training and deploying these models require enormous computational resources, leading to higher energy consumption and operational costs. Another challenge is the generation of hallucinations—outputs that are factually incorrect or nonsensical, often resulting from overfitting on large training datasets or biased data. Mitigating hallucinations is an ongoing area of research .

2.1.3 Environmental Impact of LLMs

The growing size of LLMs has had a profound environmental impact. The training of large models like GPT-3 consumes substantial amounts of energy, with estimates suggesting that training GPT-3 consumed approximately 1,287,000 kWh, equivalent to 552 tons of CO2 emissions.[2] This high energy demand arises from both dynamic computing—when the hardware is fully utilized—and idle computing—when the hardware is waiting for data or processes.

Inference, or the ongoing use of the trained model, also contributes significantly to energy consumption, especially when running large models in cloud environments. To address these issues, several approaches, including model compression, quantization, and distillation, have been proposed to reduce the environmental impact by making LLMs more energy-efficient .

2.1.4 Running LLMs on Edge Devices

Deploying LLMs on mobile and edge devices presents several challenges due to limited processing power, memory constraints and battery capacity.Various hybrid approaches have been developed to enable efficient deployment. Model compression techniques, such as pruning and quantization, reduce the model size by approximating parameters, making it feasible to run LLMs on edge devices. Knowledge distillation involves training smaller "student" models to mimic the behavior of larger "teacher" models, significantly reducing computational requirements while maintaining performance .[4]

Edge-cloud hybrid systems offer another solution, where heavy computation tasks are offloaded to the cloud while lighter tasks are processed on the edge device. This approach minimizes latency and reduces the burden on mobile hardware .

2.2 GPT Models and it's evolution

The development of GPT models has marked a significant progression in the field of large language models (LLMs). GPT-2, with 1.5 billion parameters, represents a pivotal moment in this evolution. Developed by OpenAI, GPT-2 is a generative pre-trained Transformer designed to generate coherent text and perform complex tasks with minimal fine-tuning. Its ability to scale up parameter counts demonstrated how increasing model size could lead to significant

improvements in performance. Despite these advances, GPT-2 also highlighted concerns about the misuse of large-scale language models, such as their potential for generating fake news and other forms of malicious content.

Following the success of GPT-2, ChatGPT has generated considerable excitement within the AI community. Built on the powerful GPT model, ChatGPT is specially optimized for conversational tasks.

The core principle underlying GPT models involves compressing world knowledge into a decoder-only Transformer model through language modeling. This approach enables the model to recover or "memorize" semantic knowledge and function as a general-purpose task solver. Two key elements contribute to the success of GPT models: first, the ability to accurately predict the next token in a sequence, and second, the scaling up of model size. OpenAI's research on LLMs can be roughly divided into distinct stages, each representing a step in the evolution of these models.

2.2.1 Early Explorations and GPT-1

In the early days of OpenAI, the idea of using language models for intelligent systems was explored, initially through recurrent neural networks (RNNs). The advent of the Transformer architecture, introduced by Google in 2017, led OpenAI to adapt their language modeling efforts to this new architecture. This adaptation resulted in the release of GPT-1 in 2018. GPT-1, the first model in the GPT series, utilized a generative, decoder-only Transformer architecture and employed a hybrid approach of unsupervised pretraining followed by supervised fine-tuning. This model established the foundational architecture for subsequent GPT models and demonstrated the principle of predicting the next word in a sequence.[11]

2.2.2 The Rise of GPT-2

GPT-2, which expanded the parameter scale to 1.5 billion, was trained on a large dataset known as WebText. Unlike its predecessors, GPT-2 aimed to perform tasks through unsupervised language modeling without explicit fine-tuning using labeled data. The model's approach was to predict outputs based on given inputs and task information, framing task-solving as a word prediction problem. This approach is supported by the claim that the unsupervised language modeling objective aligns with the supervised task-specific objectives. Essentially, GPT-2's unsupervised learning capabilities enable it to tackle various tasks by predicting text in a manner that mimics supervised learning outcomes.

2.2.3 Model Components and Training

GPT-2's architecture employs a decoder-only structure that processes text in a unidirectional, left-to-right manner, distinguishing it from bidirectional models like BERT that utilize both past and future context. It features a stack of decoder layers incorporating multi-head self-attention and feedforward networks, which effectively capture long-range dependencies across large text sequences. The model uses Byte Pair Encoding (BPE) for tokenization,

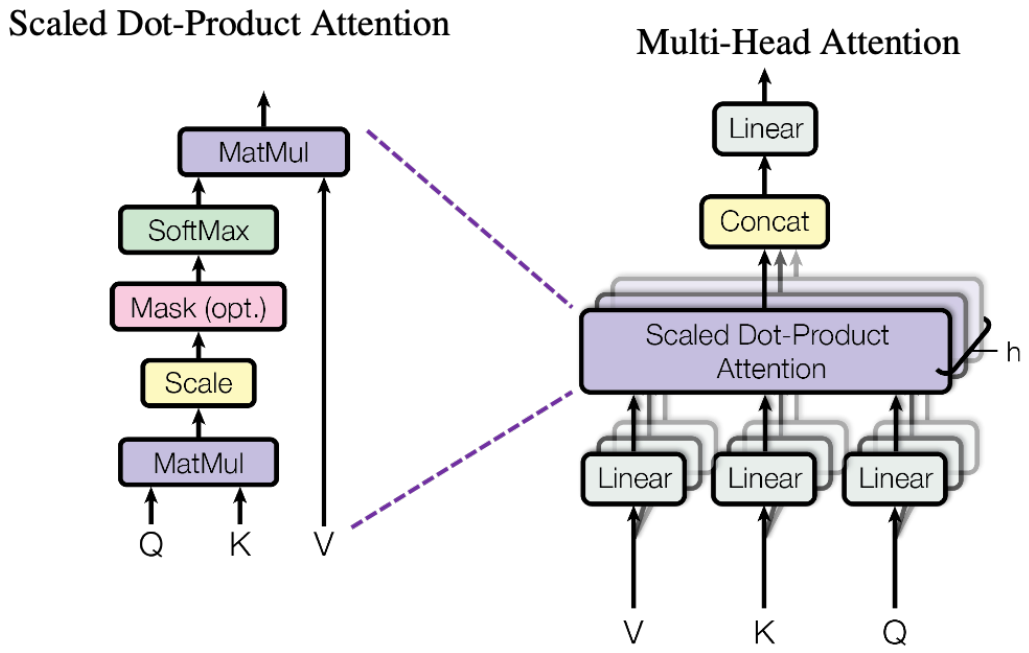

Figure 2.3: Self Attention

breaking text into subword units that are then converted into high-dimensional embeddings. Positional encodings are added to maintain word order, which is crucial for context-sensitive tasks.

GPT-2's model components include self-attention mechanisms that allow the model to weigh the importance of each token relative to others, thereby capturing long-distance dependencies. Feedforward layers follow the self-attention mechanisms to further refine the information, enhancing the model's text comprehension. Positional encoding ensures that the sequence order of tokens is preserved, which is critical for maintaining contextual relationships.

The model generates text by sampling from the output distribution at each step. Various sampling techniques are employed, including greedy decoding, top-k sampling, and nucleus (top-p) sampling. Greedy sampling selects the most probable token at each step, which can result in repetitive text. Top-k sampling limits the selection to the top-k candidates, increasing diversity, while nucleus sampling chooses from tokens whose cumulative probability exceeds a certain threshold, balancing diversity with coherence.

GPT-2 was trained on WebText, a dataset of 8 million documents, which provided the model with a broad exposure to different writing styles, topics, and structures. However, the model's large size presents significant challenges, particularly for deployment on resource-constrained devices such as smartphones. High-performance GPUs are required for both training and inference, due to the computational demands of processing and generating text at this scale.

2.3 Model Optimization Techniques

Implementing large language models (LLMs) on edge devices poses significant challenges due to limited computational and memory resources. However, several optimization techniques, including quantization, pruning, knowledge distillation, and newer methods like palettization[8], have been developed to make these models more efficient.

2.3.1 Quantization

Quantization[7] reduces the precision of a model's weights and activations, commonly from 32-bit floating-point values to lower precision formats such as 8-bit integers. This drastically reduces the model size and computational cost. Quantization can be performed either statically (during model training) or dynamically (during inference). The **Core ML** framework, commonly used in edge devices like iPhones, supports several quantization techniques. Among these, **linear quantization** is particularly effective, where model weights are scaled and rounded to the nearest integer, thus preserving accuracy while reducing complexity. This method allows for efficient execution without significant loss in model accuracy.

2.3.2 Palettization (Weight Clustering)

Palettization, also referred to as weight clustering, is another powerful technique used to compress machine learning models by reducing redundancy in their parameters[1]. Instead of storing each weight individually, palettization clusters weights with similar values and represents them using a lookup table (LUT) of centroids. This method compresses a model by mapping original weights to the nearest cluster centroid, which drastically reduces the number of unique weights the model must store. The weight matrix is converted into an index table, where each element points to a cluster center in the LUT. This is particularly useful in edge environments, as it significantly lowers the memory footprint without major losses in performance.

2.3.3 Pruning

Pruning removes less important weights from the model, reducing the number of parameters and overall model size. It can be done globally (across the whole model) or layer-wise. Research by Han et al. (2015) shows that pruning can reduce the size of neural networks substantially while maintaining their accuracy, thus making models more suitable for edge devices.

2.3.4 Knowledge Distillation

Knowledge distillation involves training a smaller model (student) to mimic the behavior of a larger, more complex model (teacher). The student model, which has fewer parameters, learns to replicate the predictions of the teacher model, making it significantly more efficient. Hinton et al. (2015) demonstrated that this technique could produce smaller models that

maintain the performance levels of their larger counterparts, making it a useful approach for deploying LLMs on mobile and edge devices.

These optimization methods, along with Core ML's support for post-training quantization and weight clustering, enable efficient deployment of LLMs on resource-constrained devices.

2.4 Hugging Face

Hugging Face is widely recognized for its contributions to the NLP community, particularly through its Transformers library, which provides pre-trained models such as GPT, BERT, and others. The library simplifies the use of state-of-the-art models with APIs that enable tasks like text generation, translation, and more. A key feature of Hugging Face is its Model Hub, a vast repository of thousands of community-contributed models, all maintained in a standardized format. This consistency allows users to easily download and run models from the hub using just a few lines of code, either locally or via Hugging Face's APIs. The platform provides comprehensive tools for fine-tuning, training, and deploying models, making it particularly valuable for developers who want to integrate AI into their applications without requiring deep machine learning expertise. Additionally, Hugging Face integrates seamlessly with popular frameworks like PyTorch and TensorFlow, supporting a wide variety of AI applications.

For this project, Hugging Face significantly simplified the acquisition of GPT-2 models, tokenizers, and APIs. It also streamlined the management of essential components like tokenization and configuration files, making the development of AI-powered mobile applications more efficient and accessible.

2.5 Core ML

Introduction to Core ML Core ML[1] is a machine learning framework introduced by Apple in 2017, designed to bring machine learning to iOS, macOS, watchOS, and tvOS applications by running models directly on Apple devices. According to the official Apple documentation, Core ML is highly optimized to take advantage of the hardware on Apple devices, such as the CPU, GPU, and Neural Engine, enabling efficient on-device inference with low latency. The framework supports a wide variety of model types, including neural networks, decision trees, support vector machines, and general machine learning models like ensembles, making it a versatile tool for mobile machine learning development.

Performance and Efficiency Core ML excels at enabling fast, on-device machine learning by utilizing hardware acceleration, particularly with Apple's A-series chips and the dedicated Neural Engine. Running models locally not only improves performance but also addresses privacy concerns, as sensitive data never needs to leave the device. The Core ML framework takes advantage of Metal Performance Shaders (MPS) for GPU-based model execution, while the Apple Neural Engine (ANE) is optimized for deep learning tasks, enhancing the processing speed of complex models like image classification, natural language processing, and object

detection.

In comparison to cloud-based inference, Core ML delivers faster performance and lower energy consumption, which is especially important for mobile applications where power efficiency is critical. For example, Apple notes that Core ML's on-device processing ensures quick responses in real-time applications, such as augmented reality (AR) and natural language processing (NLP) features integrated into iOS apps like Siri and Photos.

Model Compatibility and Conversion Core ML supports a variety of model formats and allows developers to convert models trained in frameworks such as TensorFlow, PyTorch, scikit-learn, and XGBoost into the Core ML model format using Core ML Tools. According to the Apple documentation, the conversion process is seamless for most model architectures, though models with custom layers or unsupported operators may require additional optimization before they can be used in Core ML.

The introduction of Core ML 3 further enhanced the framework's flexibility by adding support for on-device training and dynamic models such as recurrent neural networks (RNNs), making it suitable for tasks like speech recognition and language translation. Apple's tools also support techniques like quantization, which reduce model size to fit on mobile devices without sacrificing much accuracy, making it possible to run large neural networks on resource-constrained devices.

Applications in Industry Core ML is widely adopted in industries where on-device machine learning offers distinct advantages. For instance, healthcare applications can leverage Core ML for real-time diagnostics using image recognition. Apple's own applications like Photos and Siri rely heavily on Core ML for tasks like facial recognition, object detection, and natural language understanding. The Apple documentation notes that Core ML's seamless integration with frameworks like Vision (for image analysis) and Natural Language (for text processing) makes it ideal for developers looking to build AI-powered mobile applications.

Despite its advantages, Core ML faces certain limitations, particularly in managing large, complex models on resource-constrained mobile devices. Apple suggests model optimization techniques such as model pruning and quantization to reduce the size and computational demand of models. However, developers still face challenges when working with models that contain a large number of parameters, such as those used in NLP or deep learning tasks. Additionally, while Core ML excels in the Apple ecosystem, its lack of cross-platform support limits its use to iOS and macOS devices.

Moreover, while Core ML supports on-device inference and, with Core ML 3, on-device training, models requiring continuous updates or highly dynamic data may still benefit from cloud-based solutions. Apple continues to enhance the framework to meet the demands of complex AI models, but scaling such models on mobile devices remains a challenge, especially for applications requiring large neural networks.

As mobile machine learning and edge AI continue to evolve, the future of Core ML will likely involve deeper integration with other Apple frameworks and more sophisticated support for training and inference on-device. Enhanced support for larger, more complex models, better tools for optimizing and compressing models, and further integration with hardware

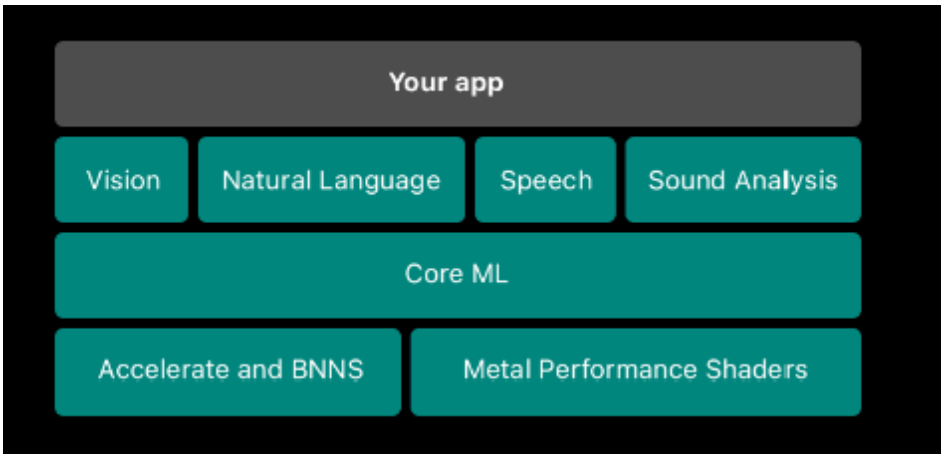

Figure 2.4: Core ML

accelerators such as the Neural Engine will make Core ML increasingly central to AI-driven iOS and macOS applications.

2.6 Swift

Swift, introduced by Apple in 2014, is a powerful and efficient language designed for iOS, macOS, watchOS, and tvOS development. It emphasizes safety, performance, and clean syntax, offering modern features like type safety, optionals, and functional programming paradigms. Swift's memory management is optimized through Automatic Reference Counting (ARC), and its concise syntax improves code readability. Swift's compatibility with Objective-C and integration with Xcode make it ideal for Apple ecosystems. SwiftUI, introduced in 2019, allows for declarative user interface design, making it easier to build responsive apps across Apple devices. Swift also supports Core ML, Apple's machine learning framework, allowing developers to integrate on-device machine learning models in their apps. Through this, machine learning models, including complex language models like GPT-2, can be efficiently run on iPhones and iPads without needing external servers, ensuring enhanced privacy and performance. Despite its youth compared to languages like Java or Python, Swift's open-source nature and robust community support ensure its continued evolution in application development. Key challenges include its frequent updates and relatively limited cross-platform tooling. However, its ease of use, performance advantages, and strong Apple integration make Swift a pivotal language for the future of mobile app development.

Chapter 3

Methodology

The project aims to integrate a GPT-2-based language model into an iOS application for natural language processing tasks. This involves multiple stages, including model development, conversion for mobile use, and optimization for efficiency.

3.1 model selection

Selecting GPT-2 for this project was a deliberate choice based on several factors that make it well-suited for mobile deployment:

1. Lightweight and Scalable: GPT-2 comes in multiple sizes, ranging from 117M to 1.5B parameters. For edge devices, the smaller models offer a balance between performance and computational resource demands, making them ideal for mobile platforms. The availability of different model sizes also enables experimentation with optimization techniques, providing valuable insights into the effects of various optimizations on models of varying scales, allowing developers to fine-tune for both performance and efficiency.

2. Flexibility: GPT-2 is versatile, supporting various NLP tasks like text generation, summarization, and translation.Its ability to generalize across different domains without fine-tuning makes it particularly attractive.

3. Interpretability: GPT-2's architecture, with its clear input-output relationship in sequence generation, allows for greater interpretability compared to more opaque models. The attention mechanisms provide insights into which words or phrases the model deems important.

4. Tokenization: GPT-2 uses Byte-Pair Encoding (BPE), which ensures efficient handling of rare and common words, contributing to high performance even with constrained input sizes. This supports an efficient representation of text and reduces overhead during input processing.

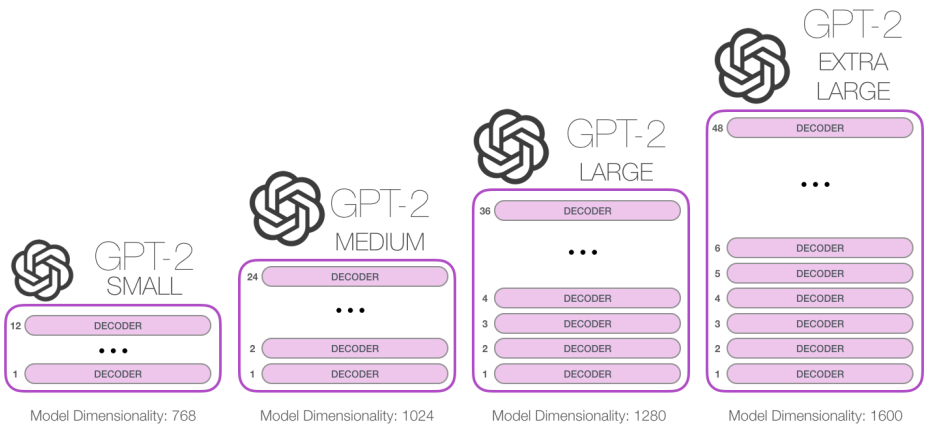

Figure 3.1: Different versions of GPT-2

5. Model Provenance: GPT-2 is well-documented, open-sourced, and widely used, making it easier to integrate into various systems. This transparency and community support make it more reliable and easier to troubleshoot.

6. Optimization Potential: The model architecture supports efficient optimizations such as quantization, pruning, and other techniques for reducing inference time and model size, allowing it to be deployed on devices with limited computational power, such as iPhones. In sum, GPT-2's modularity, lightweight structure, strong interpretability, and ease of optimization make it an excellent choice for running on resource-constrained devices like mobile phones.

3.2 Conversion and Inference

3.2.1 TorchScript Overview

TorchScript is an intermediate representation of a PyTorch model that enables the model to be serialized and optimized for deployment in production environments, including mobile and embedded devices. PyTorch models typically run in a dynamic computational graph, but for deployment, they need to be converted into a static graph that can be exported. TorchScript facilitates this by capturing the model's operations in a more portable form.

Scripting vs. Tracing

TorchScript offers two ways to convert PyTorch models:

1. Scripting:Scripting[5] converts a PyTorch model's entire Python code into TorchScript. It preserves control flow operations like loops and conditionals, making it suitable for models with dynamic computation graphs. PyTorch code that uses control structures

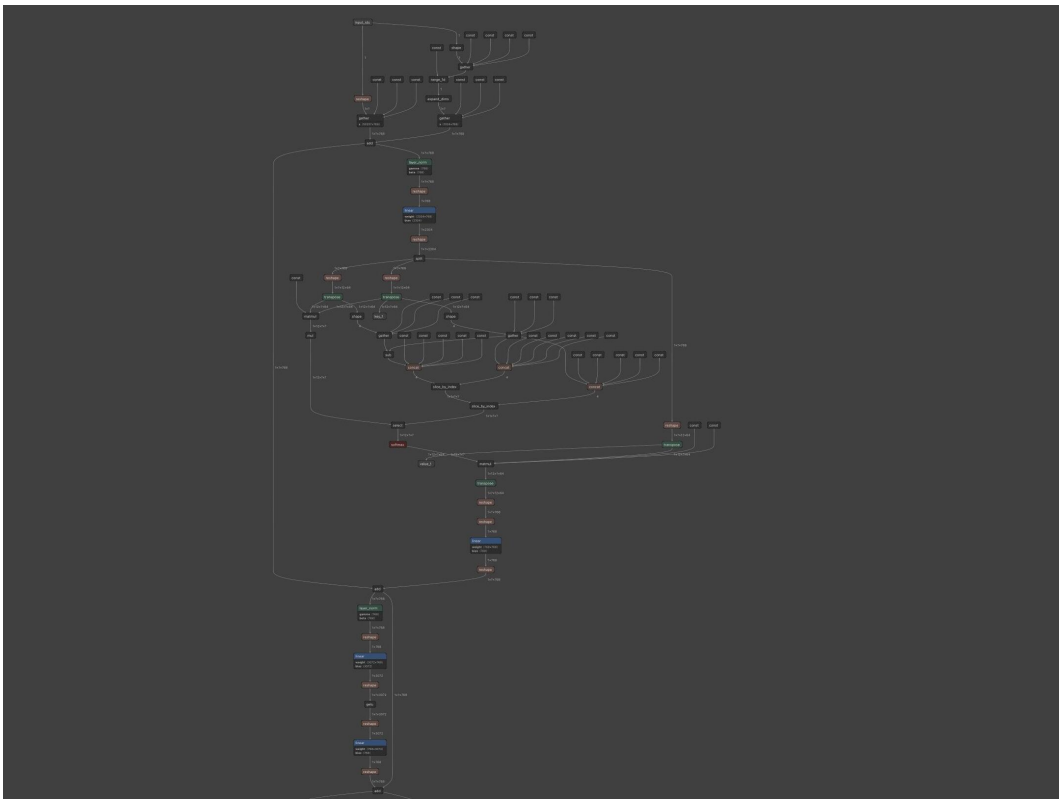

Figure 3.2: Visual representation of converted model

such as 'if' statements or loops is converted into static code. This provides greater flexibility since the model's behavior can depend on the input.

2. Tracing: Tracing[6] records the operations performed during a forward pass with a sample input, capturing the flow of operations statically. It works well with models that have a fixed computation graph. However, it does not capture control flow structures like loops or conditional branching, making it ideal for models with static execution paths.

3.2.2 Inference and Text Generation

Once the model is converted using TorchScript and deployed in an application, it is ready for inference, which refers to the process of generating predictions or outputs from the model. For text generation tasks, GPT-2 typically relies on two common sampling techniques:

1. Greedy Sampling: Greedy sampling selects the token with the highest probability at each step of generation. This method is deterministic, as it always chooses the most probable token, but it can result in repetitive and predictable text. Example: At each time step, the model computes the probabilities of all possible next tokens. Greedy sampling selects the token with the maximum probability.

2. Top-K Sampling: In top-K sampling, only the top-K most probable tokens are considered at each generation step, introducing more diversity. A token is randomly selected from this restricted pool of K candidates, allowing for a balance between determinism and randomness.

   Example: In top-K sampling with $K = 50$, the model computes the probabilities for all possible next tokens, sorts them, and only the top 50 tokens are considered for the next step.

These techniques are critical for controlling the quality of generated text, allowing developers to tune the balance between creativity and predictability during inference. For real-time applications, such as on-device text generation in iOS, these methods play an essential role in delivering coherent and contextually accurate outputs.

3.3 Optimizations

3.3.1 Linear quantization

Linear quantization is also known as affine quantization, achieves this process by mapping the range of float values to a quantized range, such as the range for 8-bit integers [-127, 128], and interpolating linearly. This mapping is expressed by the following mathematical equations:

```
# process of dequantizing weights:
w_unquantized = scale * (w_quantized - zero_point)

# process of quantizing weights:
w_quantized = clip(round(w_unquantized/scale) + zero_point)
```

In the above equations, w_unquantized and scale are of type float, and w_quantized and zero_point (also called quantization bias, or offset) are of the quantized data type.

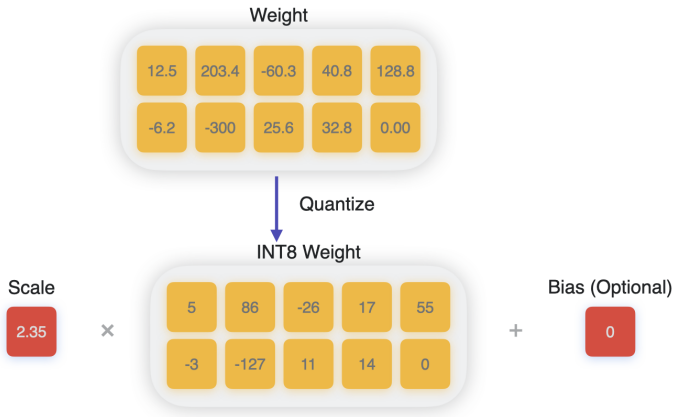

Figure 3.3: Weight Quantization

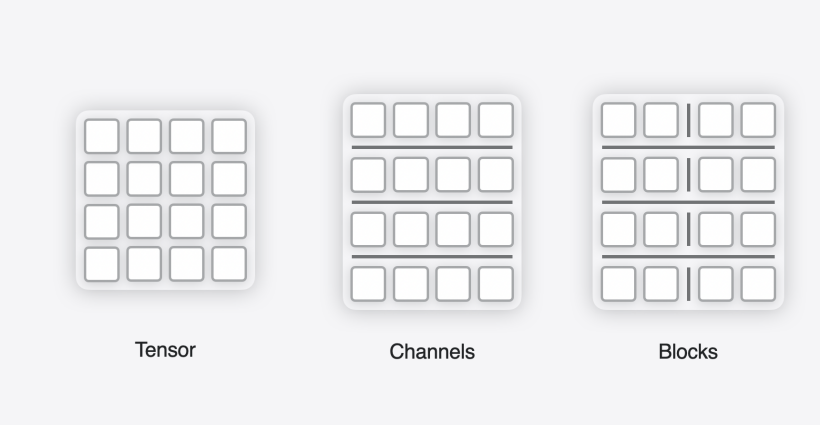

Figure 3.4: Enter Caption

Quantization Granularity

There are three modes supported for QuantizationGranularity: per_tensor, per_channel, per_block. per_tensor granularity computes a single float scale value (and zero point, in the case of symmetric quantization) for the whole tensor. per_channel granularity uses a scale factor for each outer dimension (also referred to as the output channel) of the weight tensor. The per_block granularity shares scale factors across blocks of values in the weight tensor which helps provide more fine-grained control of quantizing the weight values which contributes to improving the accuracy of the model.

3.3.2 Palletization

also referred to as weight clustering, compresses a model by clustering the model's float weights, and creating a lookup table (LUT) of centroids, and then storing the original weight values with indices pointing to the entries in the LUT.

Weights with similar values are grouped together and represented using the value of the cluster centroid they belong to, as shown in the following figure. The original weight matrix is converted to an index table in which each element points to the corresponding cluster center.

N=1,2,3,4,6,8 are supported, where N is the number of bits used for palettization.

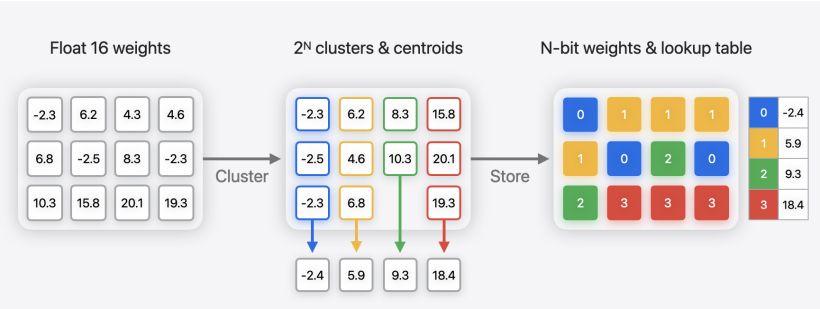

Figure 3.5: Palletization

Granularity Figure above shows what is referred to as per_tensor granularity, where the entire tensor shares a single LUT. This can lead to high approximation error for large matrices. Starting iOS18/macOS15, a mode called per_grouped_channel is available , which allows a group of channels (specified by the parameter group_size) to share a single LUT, thereby having multiple LUTs for the whole weight matrix. For example, a weight of shape (1024, 1024), with group_size=16, will have 64 LUTs.

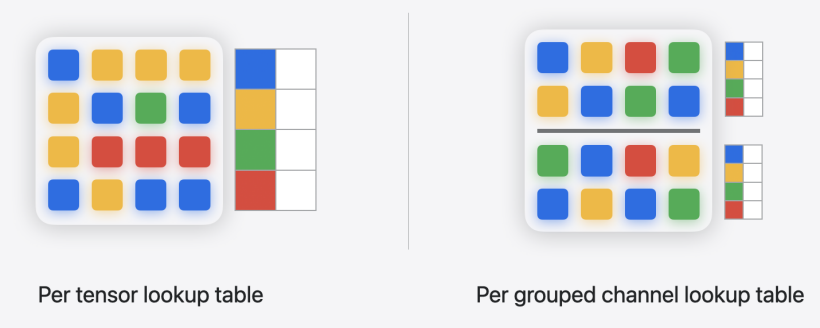

Figure 3.6: Palletization Granularity

3.4 Analysis

1. Inference Speed: A key metric for evaluation is the time taken by the model to generate responses on the edge device. This involves assessing the speed of inference using CPU-only processing in iOS.

2. Storage Used: The model's storage footprint will be analyzed, especially after applying optimization techniques like quantization and pruning to ensure it fits within the constraints of mobile devices.

3. CPU Power Consumption: An analysis of how much CPU power is consumed during inference will help determine the model's efficiency on mobile devices. This ensures that the app remains power-efficient and doesn't drain battery excessively.

Evaluating these factors will be essential to balancing performance, usability, and resource efficiency when deploying the GPT-2 model in an iOS environment.

Chapter 4

Setup and Implementation

The following section outlines the approach used to implement a large language model (LLM) on an iOS platform, focusing on the conversion and optimization of a GPT-2 model. The process involved tracing the model using TorchScript, converting it into a Core ML format using coremltools, and applying various optimization techniques to ensure efficient on-device inference. There are four major steps to this process: model conversion, model optimization, model inference, and app development. Where model multiple techniques for next token selection were implemented in the Application. This section details the step-by-step techniques, tools, and frameworks used to achieve real-time text generation on consumer-grade hardware while maintaining performance and accuracy.

4.1 Project Requirements

1. Hardware Requirements
   (a) iOS Device: The project targets iOS devices, so you will need an iPhone or iPad for testing and deployment.
   (b) Mac Computer: A Mac with macOS installed is necessary for development, as Xcode, the primary development environment for iOS apps, is only available on macOS.
   (c) Processor and RAM: A machine with at least an Intel Core i5 processor and 8GB of RAM is recommended for smooth operation, especially during model training, optimization, and testing phases.
2. Software Requirements
   (a) Xcode: The Integrated Development Environment (IDE) for macOS, used for developing iOS apps. The latest version of Xcode is required to support the Core ML framework and Swift programming.

(b) Python Environment: Python 3.7 or later with virtual environments enabled. Python is used for model training, optimization, and conversion into Core ML format.

(c) Core ML Tools: ‘coremltools‘, a Python library for converting machine learning models into Core ML format, is required. Version 4.0 or later is recommended.

(d) Transformers Library: The Hugging Face Transformers library is required to load and work with the GPT-2 model in Python. Ensure that you have a compatible version that supports the model you intend to use.

(e) Torch and TorchScript: PyTorch and TorchScript are needed to handle the model operations and tracing before conversion into Core ML format.

3. Development Tools

(a) Swift Programming Language: The iOS app is developed in Swift, so familiarity with Swift is necessary.

(b) UIKit: Apple’s UI framework for developing iOS interfaces. This project uses UIKit for creating the user interface.

(c) Core ML: Apple’s framework for on-device machine learning. Core ML is utilized to run the GPT-2 model on iOS devices.

(d) Terminal/Command Line Interface: Used for running Python scripts, installing dependencies, and managing virtual environments.

4. Model Requirements

(a) Pre-trained GPT-2 Model: Download the four sizes of pre-trained GPT-2 models, the gpt-small, gpt-medium, gpt-large, gpt-xl.

(b) Tokenization Support: Tokenization is required to process input text into a format the GPT-2 model can understand. Download the tokenizer. A single tokenizer work for all the variations of GPT-2

5. Optimization Tools

(a) Core ML Optimization Tools: The ‘coremltools.optimize‘ module is used to compress and optimize the model. This is crucial for reducing the model size and improving its performance on mobile devices.

(b) Quantization and Palettization Techniques : Implementing these techniques reduces the model’s memory footprint and computational load during inference.

6. User Interface (UI) Requirements

(a) Text Input Field: For the user to input prompts.

(b) Text Output Field: To display the model’s generated responses.

   (c) Buttons: For triggering text generation and stopping generation

7. Testing and Debugging Tools

   (a) Simulator: Xcode's built-in iOS Simulator is used to test the app on different iOS versions and devices.

   (b) Device Logs and Debugging : Tools within Xcode for logging and debugging app behavior during testing.

8. Deployment Requirements

   (a) Apple Developer Account: Necessary for deploying the app on an actual iOS device and for distribution through the App Store.

   (b) App Store Guidelines: Ensure the app complies with Apple's guidelines for App Store submission, particularly regarding model performance, resource usage, and user privacy.

These requirements outline the tools and technologies necessary to develop, optimize, and deploy an iOS app that uses a GPT-2 model for text generation. Each component is essential for ensuring that the project runs efficiently and effectively on target devices.

4.2 Technical stack and Versions

1. Python 3.10.14: The primary scripting language for model handling and processing.
2. NumPy 1.21.1: Used for numerical computations.
3. Scikit-learn 1.1.2: Applied for model evaluation.
4. Transformers 4.44.0: Provides access to pre-trained models like GPT-2.
5. PyTorch 2.0.0: Core deep learning framework for GPT-2 manipulation.
6. Swift Xcode: Essential for iOS development and UI integration.
7. CoreMLTools 8.0b1: Converts models for iOS inference.
8. Conda: Manages virtual environments and dependencies.
9. macOS 15 or greater: Required for development with Xcode.
10. iOS 18 or greater: Target operating system for the iOS app.

4.3 Model Conversion

The GPT-2 model and it's different variation were selected for this project due to their advanced capabilities in generating coherent and contextually relevant text. The choice is based on its balance between performance and size, making it an optimal candidate for deployment on mobile devices. The model, known for its capacity to generate sophisticated language, is loaded using the 'transformers' library. Specifically, the 'GPT2LMHeadModel' and 'GPT2Tokenizer' classes are utilized to handle the model's loading and text tokenization processes, respectively.

```
mlmodel = GPT2LMHeadModel.from_pretrained("gpt2-xl", TorchScript = True).eval()
tokenizer = GPT2Tokenizer.from_pretrained("gpt2-xl")
```

This step ensures that the model is correctly instantiated and prepared for further conversion and deployment. An alternative approach to this is to download the files to a local path and load the model and tokenizer from the local path.

```
local_path = "/Users/username/documents/huggingface/gpt2"
mlmodel = GPT2LMHeadModel.from_pretrained(local_path)
```

Once the model is loaded, A input text is tokenized using the GPT-2 tokenizer, converting it into a format that the model can process effectively. This step is critical as it breaks down the input text into tokens that the GPT-2 model uses to predict the next word in the sequence.

```
input_text = "The quick brown fox jumps over the lazy"
input_ids = tokenizer.encode(input_text, return_tensors="pt")
```

These "input_ids" act as a dummy input, which is fed to the model for the downstream task of tracing.

To optimize the model for deployment, it is traced using TorchScript. TorchScript allows the model to be converted into a format that can run independently of the Python environment, making it suitable for deployment on iOS devices. The tracing process captures the model's static computational graph, ensuring that the operations performed during inference are preserved and optimized for mobile execution.

```
traced_token_predictor = torch.jit.trace(token_predictor, input_ids[-1])
```

The traced model is then converted to the Core ML format of the model using the 'coremltools' library. This conversion is necessary to enable the model to run natively on iOS devices using Apple's Core ML framework. During the conversion process, the input tensor types are specified, and the model is set to meet the precision and deployment requirements for iOS.

```
mlmodel = ct.convert(
traced_token_predictor,
inputs=[ct.TensorType(name="input_ids", shape=(ct.RangeDim(1, 64),), dtype=np.int32)
compute_precision=ct.precision.FLOAT32,
minimum_deployment_target=ct.target.iOS18
)
mlmodel.save("traced.mlpackage")
```

1. mlmodel: This variable holds the converted Core ML model. After the conversion process completes, mlmodel will contain the Core ML version of the PyTorch or TensorFlow model (traced_token_predictor in this case), ready for deployment in iOS apps

2. ct.convert: This is the conversion function from the Core ML Tools (ct) library, which converts a PyTorch, TensorFlow, or other machine learning model into a format compatible with Core ML. In this case, it's converting the traced_token_predictor model to Core ML.

3. traced_token_predictor: This is the model that is being converted. It's typically a PyTorch model that has been traced (i.e., recorded while running to capture its computational graph), allowing it to be converted to a Core ML model

4. inputs=[ct.TensorType(...)]: This argument specifies the input type and shape that the Core ML model expects. Here, it's telling Core ML the nature of the input that the model will receive:

   (a) ct.TensorType: Specifies the type of tensor (multidimensional array) that will be used as input to the model. It describes the shape and data type of the input.

   (b) name="input_ids": This gives a name to the input tensor, which is useful when referring to this input in the Core ML model. In this case, the input tensor is named input_ids, which represents token IDs fed to the model during inference.

   (c) shape=(ct.RangeDim(1, 64),): Specifies the shape of the input tensor. Here, it's using a range dimension: ct.RangeDim(1, 64): This indicates that the first dimension of the tensor can vary between 1 and 64. This is useful when the input sequence length might vary between different inputs.

   (d) dtype=np.int32: Specifies the data type of the input tensor as int32. In this case, the input consists of integers, which is common for token IDs in NLP models.

5. compute_precision=ct.precision.FLOAT32 : This specifies the precision or data type for the model's computations. In this case, the model is set to use 32-bit floating-point precision (FLOAT32), which is standard for many machine learning models. This can be changed to other precisions (e.g., FLOAT16 for lower precision) to trade off between accuracy and speed.

6. minimum_deployment_target=ct.target.iOS18: This specifies the minimum iOS version that the model should support for deployment. In this case, the model is targeted for devices running iOS 18 or higher which is necessary for certain downstream optimization that were applied to the model. Core ML will optimize the model based on the capabilities available in iOS 18, such as taking advantage of newer hardware or software features

The model was then saved as an '.mlpackage', which is the format required for deploying models within iOS applications.

4.4 Model Optimization

Given the computational constraints of mobile devices, the model underwent several optimization steps to reduce its size and improve inference speed without significantly compromising performance. The first optimization involved linear quantization, which reduces the precision of the weights to int8, thereby decreasing the model's size and improving execution speed.

```
op_config = cto.coreml.OpLinearQuantizerConfig(
mode="linear_symmetric",
dtype="int8",
granularity="per_channel",
block_size=32,
)
model_config = cto.coreml.OptimizationConfig(global_config=op_config)
compressed_mlmodel = cto.coreml.linear_quantize_weights(mlmodel, config=model_config)
```

'OpLinearQuantizerConfig' is a configuration used in the linear quantization process, which reduces the precision of the model's weights from floating-point (typically 32-bit) to a lower precision, such as 8-bit integers ('int8'). The purpose of this quantization is to compress the model size, reduce memory usage, and increase the inference speed on resource-constrained devices like mobile phones, while attempting to retain most of the model's accuracy.

1. Mode: linear_symmetric means that the quantization is symmetric around zero, meaning both positive and negative values are scaled equally. This helps in reducing quantization errors, especially for models that have weights symmetrically distributed around zero.

2. dtype: The data type ('dtype') used is 'int8', which indicates that each weight will be represented as an 8-bit integer instead of a 32-bit floating-point number. This reduces the storage requirements by a factor of four.

3. Granularity: 'per_channel' quantization applies different scales to different channels of the model's weights, providing finer control and often better accuracy retention compared to per-tensor quantization, which applies the same scale across all channels.

4. Block Size: ‘block_size=32‘ specifies that the quantization will be performed in blocks of 32 weights. This approach helps in balancing the quantization error and computational efficiency.

Further optimization was achieved through Palettization

```
op2_config = cto.coreml.OpPalettizerConfig(
mode="kmeans",
nbits=4,
granularity="per_grouped_channel",
group_size=16
)
model_config = cto.coreml.OptimizationConfig(global_config=op2_config)
palletized_mlmodel = cto.coreml.palettize_weights(compressed_mlmodel, model_config)
palletized_mlmodel.save("PalletGPT2.mlpackage")
```

‘OpPalettizerConfig‘ is a configuration used in the weight palettization process, which further reduces the model size by clustering weights into a smaller set of representative values (a palette). This process significantly reduces the number of unique values in the weight matrices, leading to compression in model size and faster inference.

1. Mode: ‘kmeans‘ palettization applies the k-means clustering algorithm to the weights. This algorithm groups similar weights together and replaces them with the centroid of the cluster. This step can significantly reduce the number of unique values in the model’s weights.

2. nbits: ‘nbits=4‘ means that the weights are reduced to 4-bit representations. This further compresses the model compared to standard 8-bit quantization. However, there is a trade-off, as using fewer bits can lead to greater loss of precision.

3. Granularity: ‘per_grouped_channel‘ applies the palettization across grouped channels, meaning that the clustering is done within each group of channels rather than globally. This approach allows for more tailored optimization, reducing the impact on model accuracy. The following choice of Granularity is only available from IOS 18/MacOs 15 onwards, hence the requirements mentioned are high.

4. Group Size: ‘group_size=16‘ specifies that weights are clustered into groups of 16 channels. This grouping enables the model to retain more context-specific information, which can help in preserving accuracy despite the aggressive reduction in precision.

The combination of ‘OpLinearQuantizerConfig‘ and ‘OpPalettizerConfig‘ provides a robust method to significantly reduce the size of the GPT-2 models while maintaining a balance between computational efficiency and model accuracy. By first reducing the precision of the weights and then clustering these weights into representative values, the model becomes much

more suitable for deployment on mobile devices, where computational resources and memory are limited. These optimizations ensure that the model can deliver high-quality language generation performance in a mobile environment. These optimizations ensured that the model is suitable for deployment on resource-constrained devices like smartphones.

4.5 App Development and Inference

This section introduces the application, which implements a large language model (GPT-2) on an iOS platform using Swift, CoreML, and PyTorch. The application allows users to input text, which is processed using GPT-2 to generate text in real-time on the device. The entire process, from user input to text generation, occurs on-device without relying on cloud services, ensuring privacy and efficient performance.

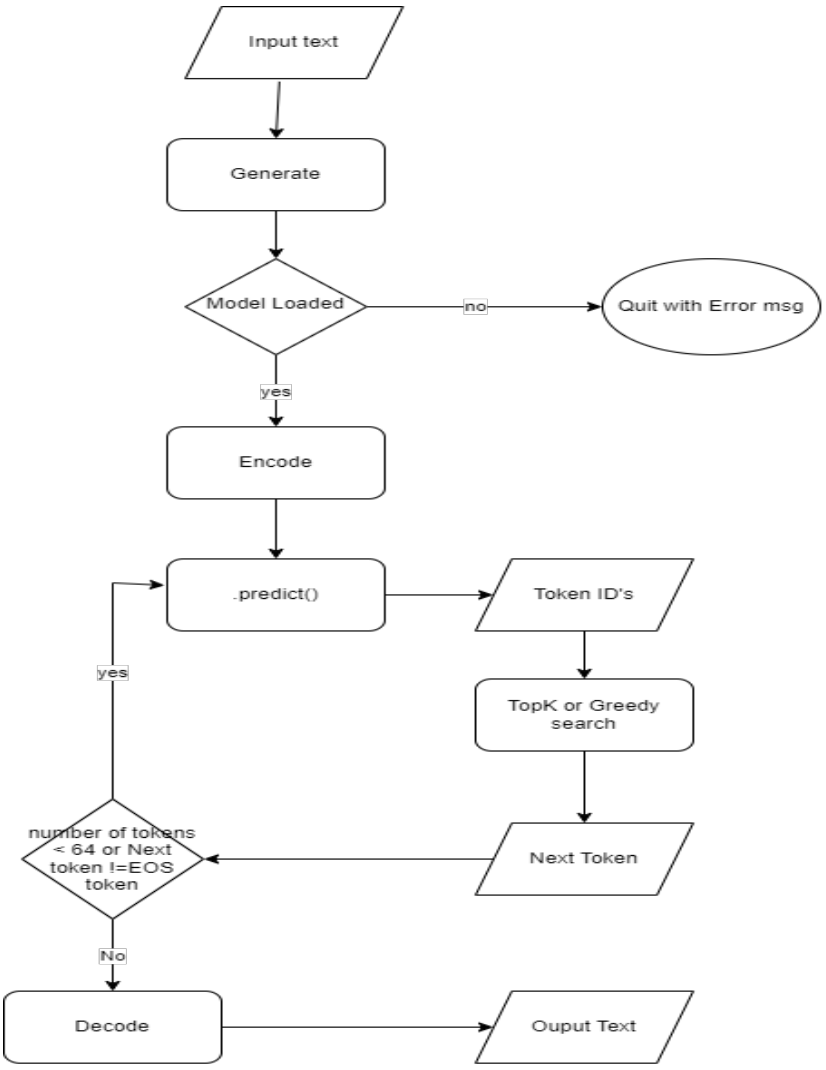

Figure 4.1: Flow structure of the application

4.5.1 App Structure Overview

The app consists of the following primary components:

1. App Entry Point (chatgpt2App.swift)
2. User Interface (ContentView.swift)
3. ViewModel for Model Interaction (GPT2ViewModel.swift)
4. Text Generation Logic (GPT2.swift)
5. Tokenization and Processing (GPT2Tokenizer.swift)

Summary of Flow

1. chatgpt2App.swift: Initializes the app with ContentView as the main screen.
2. ContentView.swift: Sets up the UI for user input and output, calling the GPT2ViewModel to manage the text generation process.
3. GPT2ViewModel.swift: Loads the model, manages input/output, and handles generation and stopping via interaction with the GPT-2 model.
4. GPT2.swift: Implements the GPT-2 model's text generation logic, token prediction, and sampling strategies.
5. GPT2Tokenizer.swift: Encodes and decodes text to allow the model to process it and return readable outputs.[10]

These components work together to take user input, pass it through the GPT-2 model, and return generated text.

chatgpt2App.swift

This file is the entry point for the iOS app, responsible for launching the interface. It initializes the app and sets ContentView as the main screen that the user interacts with. This file ensures that when the app is launched, the UI is correctly rendered.

Key Functions:

1. Initializes the app with ContentView as the first screen the user sees.
2. The file doesn't handle any complex logic but acts as a central initialization point for the app's interface.

ContentView.swift

This file creates the user interface of the app using SwiftUI. It allows the user to enter text, generate an output using the GPT-2 model, and stop text generation mid-process. The file manages two buttons for generating and stopping the process, displays the input, output, and time taken for the generation, and manages the layout and styling of these UI elements. Key Functions:

1. Input Field: Allows users to enter text for generation.
2. Generate/Stop Buttons: Start and stop the text generation process by calling functions in the ViewModel.
3. Output Display: Shows the generated text and time taken for the process.
4. UI Updates: The app uses @StateObject to connect the user interface to the GPT2ViewModel, ensuring that changes in data reflect on the screen.

GPT2ViewModel.swift

This file acts as the intermediary between the UI and the underlying GPT-2 model. It manages loading the model, processing user input, and handling responses from the model. The ViewModel also handles asynchronous tasks like generating the text in the background and stopping the generation mid-way.

Key Functions:

1. loadModel(): Loads the GPT-2 model with a specific decoding strategy, such as topK(40).
2. generateResponse(): Generates text based on user input by calling the generate function in the GPT2 class. The result is then displayed on the UI.
3. stopResponseGeneration(): Stops the text generation process mid-way by setting a flag in the GPT-2 model.

The GPT2ViewModel interacts directly with the GPT2.swift file to process inputs and generate outputs, managing UI responsiveness and ensuring smooth user interaction.

GPT2.swift

This file contains the core logic of the GPT-2 model. It loads a pre-trained GPT-2 model and manages the process of text generation using different strategies, such as greedy sampling or top-K sampling. The file controls token prediction and handles the input and output sequences.

Key Functions:

1. init(): Initializes the GPT-2 model with a selected strategy (greedy or topK) and sets up the tokenizer.

2. generate(): The main function for text generation. It processes the input tokens, generates text, and appends tokens until the required number of tokens is generated or the process is stopped.

3. predict(): This function predicts the next token based on the input tokens using the model's current state. It applies different strategies to select the most appropriate token for the next step. it is a part of the coremltools, used to run predictions on the .mlpackage

4. stopGeneration(): Halts the text generation process, used to prevent further generation if the user presses the stop button.

GPT2Tokenizer.swift

This file handles the tokenization of input text and decoding of generated tokens. It converts the input text into tokens that the GPT-2 model can process and decodes the output tokens back into readable text.[10]

Key Functions:

1. encode(): Converts input text into tokens using Byte Pair Encoding (BPE).

2. decode(): Converts a sequence of tokens back into human-readable text.

This outlines the process of selecting, preparing, optimizing, and deploying a large language model on an iOS device. By leveraging PyTorch for model preparation, TorchScript for tracing, Core ML for deployment, and Core ML Tools for optimization, the project successfully brings advanced language generation capabilities to mobile devices. This approach ensures that the GPT-2 model can be used efficiently on iOS, providing a seamless and responsive user experience.

4.6 Guide

Once the app ChatGPT2 is installed on an iPhone or iPad running iOS 18 or later, follow these steps to use it:

1. Open the app.

2. Enter an incomplete sentence in the text field.

3. Press Generate to start text generation. The app generates the sentence one token at a time, displaying the time taken for each token above the input field.

4. The generation stops if: The end-of-sentence token is reached or The token limit of 64 is hit or The Stop button is pressed.

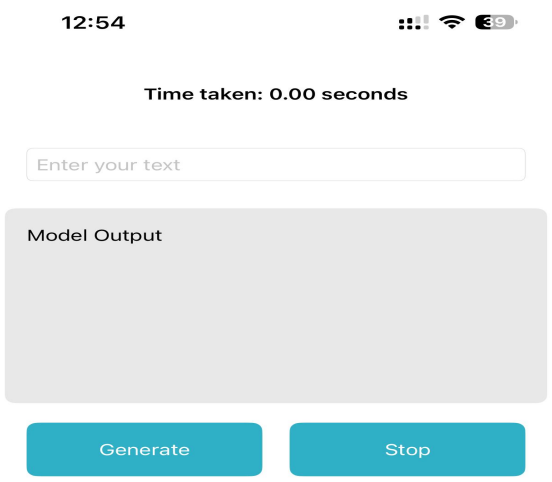

Figure 4.2: Application UI

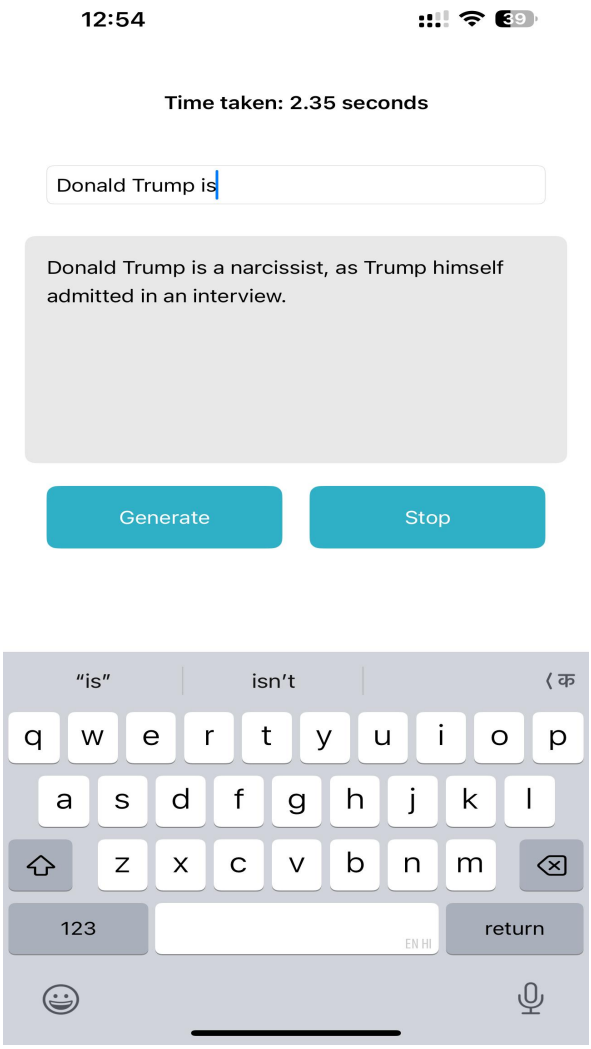

Figure 4.3: Text generation in progress

4.7 Summary of Implementation

The implementation of the project involved converting GPT-2, a large language model, into an iOS application for real-time, on-device text generation. The model was first optimized by reducing its size through techniques like token sampling and converting it into Core ML format using 'coremltools'. Afterward, it was integrated into an iOS app using Swift, ensuring it runs efficiently on iPhones or iPads with iOS 18 or above. The app features a user interface that allows users to input incomplete sentences and generate text predictions using the GPT-2 model. The implementation process focused on performance optimizations to enable smooth inference on mobile devices, avoiding reliance on cloud-based servers while maintaining the model's accuracy and real-time response.

Chapter 5

Evaluations and Discussions

This section delves into the core findings derived from the implementation of the GPT-2 model. This section will discuss the performance metrics, evaluation techniques, and the overall effectiveness of the model on the iOS platform. It will also highlight the computational efficiencies gained through model optimizations and provide a comparative analysis of different text generation methods. The results will serve as the foundation for assessing the model's accuracy, efficiency, and its practical applicability in real-world scenarios.

5.1 Storage space comparison

Comparison between compressed and uncompressed GPT models

–	Uncompressed	Compressed
gpt2-small	499.2 mb	126.8 mb
gpt2-medium	1.42gb	358.5 mb
gpt2-large	3.1 gb	779.5 mb
gpt2-xl	6.23 gb	1.57 gb

Table 5.1: Size Comparisons between compressed and uncompressed models

5.1.1 Analysis of Size Comparison Table

The table compares the disk storage size between uncompressed and compressed GPT-2 models of different sizes. For example, the GPT-2 small model reduces from 499.2 MB to 126.8 MB after compression, and the GPT-2 XL model shrinks from 6.23 GB to 1.57 GB.

5.1.2 Results and Insights

The compression achieves a significant reduction in size, with reductions between 60-75% across all models. This reduction in size makes the models more portable and easier to deploy

on devices with limited storage. However, there is a trade-off as this reduction in size typically increases computational demands during inference, which affects memory (RAM) usage.

5.2 RAM Usage

RAM Usage comparison between compressed and uncompressed GPT models

–	Uncompressed	Compressed
gpt2-small	145 mb	820 mb
gpt2-medium	190 mb	1,7 gb
gpt2-large	340 mb	3.5 gb
gpt2-xl	520 mb	6.6 gb

Table 5.2: RAM Usage Comparisons between compressed and uncompressed models

5.2.1 Description for RAM Usage Table and Analysis

The table showcases the RAM usage comparison between compressed and uncompressed GPT-2 models of varying sizes. Notably, compressed models exhibit a significant increase in RAM usage across all sizes. For instance, GPT-2 small increases from 145 MB to 820 MB, and the GPT-2 XL model rises from 520 MB to 6.6 GB.

This increase in RAM usage can be attributed to palettization. While palettization reduces the model's disk space, it introduces additional computational complexity during inference. Instead of directly using stored weights, the model must perform palette lookups for compressed weights. This increased operation leads to more memory consumption, especially for larger models.

5.2.2 Analysis of Results

The table reveals a clear trade-off between reduced model size and increased RAM consumption. Palettization efficiently minimizes disk storage, making the models more portable and suitable for deployment on devices with limited storage. However, the increase in RAM usage makes it less ideal for memory-constrained environments like mobile or edge devices.

This highlights a key challenge in model compression: While storage footprint is reduced, maintaining or improving inference performance—without overwhelming RAM usage—remains a technical hurdle. Efficient RAM management techniques or other optimizations would be necessary to balance both memory and storage needs when deploying such compressed models.

5.3 Inference Speed

Per-token speed comparison comparison between compressed and uncompressed GPT models

–	Uncompressed	Compressed
gpt2-small	0.22 seconds	0.23 seconds
gpt2-medium	0.37 seconds	0.34 seconds
gpt2-large	0.66 seconds	0.56 seconds
gpt2-xl	1.33 seconds	0.88 seconds

Table 5.3: Per-token speed comparison comparison between compressed and uncompressed models

5.3.1 Analysis of Per-Token Speed Comparison Table

The table presents the inference speed per token for both compressed and uncompressed GPT-2 models. In most cases, compressed models exhibit faster per-token generation times compared to their uncompressed counterparts, particularly in larger models such as GPT-2 XL, where the compressed version speeds up token generation from 1.33 seconds to 0.88 seconds.

Insights:

- **Larger models benefit more from compression:** GPT-2 XL sees a significant reduction in inference time, improving efficiency by approximately 34
- **Smaller models show minimal speed differences**, with GPT-2 small seeing almost no change.
- **Compression techniques** not only reduce storage size but can also lead to faster inference times, particularly for large models. This highlights the potential for optimization techniques like quantization and pruning to improve both memory usage and computational speed.

5.4 Evaluations and Discussions

This chapter presents a comprehensive analysis of the GPT-2 model's performance on iOS devices, focusing on storage space, RAM usage, and inference speed. Compression significantly reduces model size, with reductions between 60-75%, making deployment on mobile platforms more feasible. However, compression increases RAM usage, particularly for larger models, due to techniques like palettization. Inference speed also improves with compression, especially for larger models, highlighting the benefits of compression for both storage and speed. These findings emphasize the need to balance storage efficiency with memory management for optimal deployment on mobile devices.

Chapter 6

Discussion

The project successfully demonstrated the feasibility of deploying a large language model (LLM) such as GPT-2 on iOS, addressing key challenges related to resource efficiency, privacy, and real-time performance on mobile devices. Several core elements emerged during the project, such as the importance of model compression, inference optimization, and the integration of advanced sampling techniques like top-k sampling. These factors are critical to balancing performance with the computational limitations inherent in mobile hardware.

6.1 On-Device Inference vs. Cloud-Based Solutions

A significant advantage of this project was the decision to run the GPT-2 model locally on the device without relying on cloud-based services. This ensures that the data remains private and secure, as no information is transmitted to external servers. On-device inference also reduces latency, resulting in faster response times and a better user experience. By processing the text locally, the app improves real-time interaction, which is essential for mobile applications requiring instant feedback.

The challenge lies in the inherent limitations of mobile devices. Mobile hardware, especially in older devices, lacks the computational power of cloud servers or high-performance desktops. However, the project leveraged optimizations to reduce the memory footprint and computational complexity of the GPT-2 model while preserving its performance.

6.2 Model Compression and Optimization Techniques

One of the main techniques used in the project was model compression through quantization, which helped reduce the size of the model. Compressing the model was critical to running it efficiently on iOS devices with limited RAM. Techniques like linear quantization helped reduce the precision of model weights from 32-bit floating-point to 8-bit integers, drastically lowering the size of the model without significant loss of accuracy. The model sizes were reduced by up to 75

While model compression improved memory usage and allowed the app to run smoothly on iOS, the technique of palletization revealed some challenges. Palletization, while effective in shrinking the model's disk size, resulted in higher RAM usage, as evidenced by the analysis of RAM usage in both compressed and uncompressed models. This outcome highlighted the trade-offs between storage efficiency and real-time memory allocation, which is a key insight for future optimization work. Further exploration into dynamic memory management techniques or partitioning methods could help address these challenges.

6.3 Sampling Techniques for Text Generation

Text generation relied on top-k sampling, which allowed the model to select from the most probable tokens at each step, thereby ensuring coherent and contextually relevant text outputs. The option of greedy sampling, while simpler, could lead to more deterministic and predictable outputs, sometimes limiting the diversity of generated text.

One insight from the project is that more advanced sampling methods like top-p (nucleus) sampling could further improve the diversity and quality of generated text by selecting tokens from the smallest group whose cumulative probability exceeds a certain threshold. This technique balances randomness and coherence better than top-k sampling, allowing for more creative text generation.

6.4 Hardware-Specific Optimizations

The project highlights the potential of further optimizing the model for Apple's Neural Engine (ANE). The ANE is designed for efficient neural network processing, and future work could focus on adapting the Core ML model to take full advantage of these hardware accelerators. This would allow for faster inferences with less energy consumption, making the app more practical for real-world use cases.

6.5 Real-Time Performance and User Interaction

An important achievement of the project was real-time interaction with the model. By displaying the time taken to generate each token, users are provided with valuable feedback on the model's performance, offering transparency in the app's functionality. Despite the optimizations, larger versions of GPT-2 still exhibit slower token generation speeds compared to smaller models, as shown in the inference speed comparison. Nonetheless, the project shows that even larger models like GPT-2 XL can generate text at speeds that are acceptable for mobile users, thanks to compression and careful management of computational resources.

6.6 Challenges and Lessons Learned

One of the main challenges was balancing model size and speed. Smaller models are faster and easier to run on mobile devices, but they may generate less coherent text due to reduced parameter sizes. Conversely, larger models provide more accurate text generation but require more aggressive optimizations to fit within the resource constraints of mobile hardware.

The project also encountered memory management issues related to palletization, as compressing the model for storage resulted in significantly higher RAM usage. This suggests that memory optimization requires a delicate balance between minimizing the storage footprint and ensuring efficient runtime execution.

6.7 Future Directions

Future iterations of the project could explore several avenues for improvement:

1. **Larger Model Deployment:** Implementing larger models such as GPT-3 or GPT-2 XL with optimized quantization and hardware-specific configurations could provide more sophisticated text generation.

2. **Hybrid Cloud-Edge Processing:**A hybrid model that leverages both on-device and cloud processing could provide an optimal balance between real-time interaction and computational efficiency.

3. **Task-Specific Fine-Tuning:**Fine-tuning GPT-2 on specific tasks or datasets could improve its performance in niche applications, making it more versatile.

4. **Apple ANE Integration:**Further optimizations targeting Apple's Neural Engine could unlock faster inferences and better battery efficiency, making the app more practical for day-to-day use.

5. **Exploring Other Sampling Methods:**Implementing top-p sampling could improve the variety and quality of generated text, offering a better user experience for applications that rely on creative or exploratory text generation.

In conclusion, this chapter highlights the successful adaptation of a large language model like GPT-2 for iOS devices, emphasizing the critical balance between model complexity and mobile device constraints. Through model compression, efficient sampling, and hardware-specific optimizations, the project illustrates the potential of advanced NLP applications on mobile platforms. While challenges related to memory management and speed were addressed, further exploration into hybrid approaches, task-specific fine-tuning, and better utilization of Apple's Neural Engine are essential for unlocking the full capabilities of LLMs in mobile applications.

Chapter 7

Conclusion

The successful deployment of GPT-2 on iOS showcases the growing feasibility of running sophisticated LLMs on mobile devices, driven by optimizations in model compression and hardware advancements like the Apple Neural Engine. The current implementation, while efficient, still leaves room for improvements in terms of model size, speed, and task-specific accuracy. By integrating more advanced sampling techniques, optimizing for Apple's ANE, and exploring hybrid approaches, the app could reach new levels of performance, enabling a wide range of AI-driven applications in mobile environments.

This project also opens the door for the use of LLMs in privacy-sensitive applications, where on-device processing is critical. The balance between computational efficiency and accuracy achieved here can serve as a foundation for future mobile AI applications, pushing the boundaries of what can be done on consumer-grade devices. As mobile hardware continues to improve, the integration of more complex models, coupled with optimized inference strategies, will be key to unlocking the full potential of LLMs in everyday mobile use cases.

Bibliography

[1] Apple core ml optimization guide. https://apple.github.io/coremltools/docs-guides/source/opt-quantization-

[2] Brown, B. The environmental impact of large language models. *Cutter Consortium* (2023).

[3] Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. Language models are few-shot learners, 2020.

[4] Chen, D., Liu, Y., Zhou, M., Zhao, Y., Wang, H., Wang, S., Chen, X., Bissyandé, T. F., Klein, J., and Li, L. Llm for mobile: An initial roadmap, 2024.

[5] Model scripting,core ml. https://apple.github.io/coremltools/docs-guides/source/model-scripting.html.

[6] Model tracing,core ml. https://apple.github.io/coremltools/docs-guides/source/model-tracing.html.

[7] Optimization workflow. https://apple.github.io/coremltools/docs-guides/source/opt-workflow.html.

[8] Palettization overview. https://apple.github.io/coremltools/docs-guides/source/opt-palettization-overview.h

[9] Visualizing transformer language models. https://jalammar.github.io/illustrated-gpt2/.

[10] Swift-transformers. https://huggingface.co/blog/swift-coreml-llm.

[11] Zhao, W. X., Zhou, K., Li, J., Tang, T., Wang, X., Hou, Y., Min, Y., Zhang, B., Zhang, J., Dong, Z., Du, Y., Yang, C., Chen, Y., Chen, Z., Jiang, J., Ren, R., Li, Y., Tang, X., Liu, Z., Liu, P., Nie, J.-Y., and Wen, J.-R. A survey of large language models, 2023.