UNIVERSITY
OF HULL

# Efficient Compression Techniques for Large Language Models with Limited Compute Resources

being a dissertation submitted in partial fulfilment of the requirements for the degree of

Master of Science

in Artificial Intelligence

in the University of Hull

by

Kyriakos Topouzis

September 2025

# Acknowledgements

I would like to thank my supervisor, Dr. Muhammad Khalid, for the helpful guidance and practical suggestions he provided during our discussions. Any remaining errors are my own.

I’m grateful to my family and friends for their support, and to the open-source communities behind PyTorch, Hugging Face Transformers, ONNX Runtime, bitsandbytes, and the GLUE/SST-2 dataset; this work also benefited from Google Colab for experimentation.

# Abstract

Transformer models achieve state-of-the-art performance in natural language processing, yet their high computational and memory demands restrict deployment on everyday hardware. This dissertation investigates whether model compression and parameter-efficient fine-tuning can make transformers practical on constrained environments without sacrificing accuracy.

Two architectures were examined: BERT-base and its distilled variant, DistilBERT. The Stanford Sentiment Treebank (SST-2) provided the evaluation task, chosen for its balanced size and clear binary classification format. Experiments were conducted across three hardware settings: a CPU-only Colab backend, an NVIDIA Tesla T4 GPU, and a consumer-grade GTX 1050 Ti. The methods tested included post-training quantization (via PyTorch, ONNX Runtime, and bitsandbytes), unstructured L1 pruning combined with quantization, and parameter-efficient fine-tuning using LoRA and QLoRA. Performance was measured in terms of accuracy, latency, and memory consumption, with all experiments implemented in reproducible Jupyter notebooks.

Results showed that INT8 quantization on CPU consistently reduced latency and, in the case of ONNX Runtime, improved portability with negligible accuracy loss. On GPUs, however, full-precision FP32 remained the fastest mode: quantized variants preserved accuracy but introduced kernel overheads and additional memory usage. On the GTX 1050 Ti, lack of kernel support caused quantized models to fall back to CPU execution, erasing expected gains. Pruning improved latency but not memory, since dense runtimes do not exploit sparsity. LoRA enabled efficient fine-tuning of DistilBERT within a single T4 GPU, while QLoRA made BERT-base adaptation feasible under 16 GB VRAM, halving training time and reducing inference cost.

The dissertation contributes a hardware-aware evaluation of compression and fine-tuning methods, highlights where techniques fail as well as succeed, and provides practical deployment guidance. Findings emphasise that efficiency depends not only on algorithms but equally on hardware support and runtime maturity.

## Contents

# List of Figures

# List of Tables

# 1. Introduction

Over the last decade, the transformer architecture [1] has become the foundation of natural language processing. By replacing recurrent models with attention-based mechanisms, transformers made it possible to handle long-range dependencies and parallelise training more effectively. This innovation quickly displaced earlier approaches and allowed rapid progress in machine translation, sentiment analysis, and a wide range of classification tasks.

As models grew larger, predictable gains in performance followed. Work on scaling laws [2] showed that accuracy improves steadily with more parameters, data, and compute. This has driven the development of models such as BERT and LLaMA [3,4], and more recent systems like GPT-4 [5]. These advances have transformed the field, but they have also introduced serious challenges. Large models demand substantial computational resources for both training and deployment, often beyond the reach of individual researchers or small organisations. Even smaller variants can still be difficult to use in practice when memory or latency constraints are strict.

This tension between capability and cost has motivated research into methods that make transformer models more efficient. More recently, parameter-efficient fine-tuning methods such as LoRA and QLoRA [6,7] have made it possible to adapt large models to specific tasks without updating all their parameters. Together, these approaches form a toolkit for deploying transformers in environments where resources are limited.

Despite their promise, a gap remains between theory and practice. Much of the literature reports results obtained on powerful cloud GPUs or TPUs, with efficiency measured primarily in terms of accuracy. Less attention has been paid to latency, memory, or compatibility with common hardware. In practice, quantization can fall back to CPU execution if kernel support is missing, pruning may not reduce memory use in dense frameworks, and fine-tuning methods may behave inconsistently depending on model scale. These limitations are rarely highlighted in academic reports, which tend to emphasise headline performance rather than deployment realities.

This dissertation addresses that gap by systematically evaluating compression and fine-tuning methods under realistic hardware conditions. The study examines two models: BERT-base, a widely used 12-layer architecture with around 110 million parameters, and DistilBERT, a distilled version with about 66 million parameters that retains most of BERT’s accuracy while being smaller and faster [8]. The experiments focus on the Stanford Sentiment Treebank (SST-2), a binary sentiment classification task from the GLUE benchmark [9,10]. Its simplicity and practical relevance make it well suited to testing the impact of compression and adaptation techniques.

To capture a representative range of environments, the models were evaluated on three hardware setups: a CPU-only backend available in Google Colab, an NVIDIA Tesla T4 GPU with kernel support for low-bit operations, and a local NVIDIA GTX 1050 Ti GPU without Tensor Cores. These settings reflect contexts ranging from free or entry-level resources, through modern cloud accelerators, to older consumer-grade hardware still in wide use.

The methods under study include quantization on CPU and GPU, pruning combined with quantization on CPU, and parameter-efficient fine-tuning through LoRA and QLoRA. Performance was assessed in terms of accuracy, latency, and memory usage. These metrics were chosen to reflect the practical trade-offs of compression: whether reductions in precision or sparsity lead to meaningful improvements in responsiveness and efficiency while maintaining acceptable accuracy.

The aim of the dissertation is to provide a clear and practical comparison of these methods across models and hardware. By highlighting not only their strengths but also their limitations, the study offers evidence to guide deployment decisions outside of large data centres. Results show that INT8 quantization on CPU is highly effective, with ONNX Runtime providing the strongest latency gains [11]. On GPUs, full precision often remains fastest, especially for smaller models, while quantization introduces overheads that can outweigh theoretical savings. On legacy GPUs such as the GTX 1050 Ti, quantized inference is not viable at all. Pruning reduces latency but not memory under dense execution. LoRA enables efficient fine-tuning for smaller models, while QLoRA proves valuable at larger scale, making BERT-base fine-tuning feasible and efficient on a single T4 GPU.

The contributions of this dissertation are threefold. First, it provides a systematic comparison of compression and fine-tuning methods across CPUs, modern cloud GPUs, and older consumer GPUs, showing where each technique is practical and where it fails. Second, it integrates inference-time methods with parameter-efficient fine-tuning, offering a fuller view of how efficiency can be achieved across the lifecycle of transformer models. Third, it translates experimental findings into concrete deployment guidance, including defaults and warning signs that practitioners can use when applying these methods.

The remainder of this dissertation is organised into four main chapters. Chapter 2 reviews the literature on transformers, scaling, compression, and parameter-efficient fine-tuning. Chapter 3 sets out the methodology, including dataset, models, hardware, and evaluation metrics. Chapter 4 presents the results of eight experiments across CPU, GPU, and fine-tuned settings. Chapter 5 discusses the findings, interprets their significance, and offers recommendations for deployment, along with limitations and directions for future work.

# 2. Literature Review

## 2.1 Transformers and Scaling

The transformer architecture changed the direction of natural language processing and, more broadly, machine learning [1]. Before transformers, most sequence models relied on recurrent neural networks (RNNs) and long short-term memory networks (LSTMs). These could capture order and patterns in text but struggled with long-range dependencies. They were also difficult to parallelise, which made scaling slow and inefficient.

Transformers solved these problems with self-attention. Instead of reading tokens one by one, each token could look at every other token at once. This made training easier to parallelise and helped models learn long-distance relationships. With layers of multi-head attention and feed-forward blocks, transformers quickly outperformed older models in translation, summarisation, sentiment analysis, and question answering.

Scaling laws pushed this progress further. Kaplan et al. [2] showed that models get better in a predictable way as they grow: more data, more parameters, more compute. The community followed this path. What was once considered large — BERT with around 110 million parameters — was quickly surpassed. More recent families such as LLaMA and GPT series scale into the tens or hundreds of billions, and this trend continues. With each jump, capability increased, but so did cost.

Training usually requires fleets of GPUs, and serving these models with low latency demands careful engineering. Even the smaller transformer variants can consume enough compute and memory to matter in day-to-day use.

A simple picture helps here. Small models are like delivery vans: fast, flexible, and easy to use in many places. Large models are like heavy trucks: they can carry far more, but they also need wider roads, special handling, and larger depots. Raw power is not the only concern. Access and flexibility also matter when compute, memory, or energy are limited.

This tension has led to a line of work focused on efficiency. The aim is to keep the strengths of transformers while lowering the barriers to use. Research explores methods such as reducing numerical precision, cutting away redundant components, and adapting models with only a few added parameters. All of these point toward the same goal: making state-of-the-art models not only accurate, but also usable outside of large, well-funded data centres.

The next sections review these methods in turn, showing how they build on earlier work and where they are most effective. The focus is on techniques that have already been used in practice and that show promise under real deployment conditions.

## 2.2 Model Compression Techniques

Systems like BERT, GPT, and their successors achieve high levels of accuracy, yet they require large amounts of compute and memory. For groups outside major research labs or cloud providers, these costs can be difficult to manage. For example, BERT-base, with 110 million parameters, uses hundreds of megabytes in FP32 precision, and inference on a normal CPU can be slow. With models in the billions of parameters, the problem only becomes more severe.

To address this, researchers have developed compression techniques that aim to shrink model size, reduce latency, and lower deployment costs, while still preserving as much accuracy as possible. Three families of methods stand out. Quantization reduces the precision of numerical values. Pruning removes parameters or structures that contribute little. Knowledge distillation transfers what a large "teacher" model has learned into a smaller "student." These approaches can also be combined: quantization cuts memory, pruning reduces structure, and distillation produces leaner architectures from the start. Together, they form the foundation of work on efficient transformers.

### 2.2.1 Quantization

Quantization is one of the simplest ways to make models smaller and faster. It works by lowering the precision of the numbers used to store weights and activations. A useful analogy is a library where every book is printed in elaborate calligraphy. The content is the same, but the ornate script takes far more space. Switching to plain print makes the books easier to store and read without changing the meaning. Quantization does the same for models: compact, efficient, and easier to process.

In deep learning, the “ornate script” is the use of 32-bit floating-point (FP32) numbers for everything. This level of detail is more than what many tasks need. Replacing FP32 with lower-precision formats—such as 8-bit integers (INT8), 16-bit floats (FP16), or even 4-bit integers—reduces memory use and speeds up computation. For example, an FP32 weight takes four bytes, while an INT8 weight needs only one, a 75% reduction in storage without changing the number of parameters.

There are several ways to apply quantization. Post-training quantization (PTQ) converts a pre-trained model to lower precision directly. It requires no retraining and is fast to apply, which makes it attractive for deployment. PyTorch’s dynamic quantization is a widely used PTQ method, targeting large linear layers in transformers while leaving activations in FP32 for stability [12]. In practice, PTQ often preserves most of a model’s accuracy.

A more advanced method is quantization-aware training (QAT). Here, quantization effects are simulated during training so the model can adapt to them. This approach usually maintains accuracy better than PTQ, especially in smaller models, but it requires extra training effort. Between PTQ and QAT lies mixed-precision training, where some operations use reduced precision (commonly FP16) while sensitive calculations remain in FP32. Popularised by NVIDIA , this hybrid method has become the standard for training very large models efficiently [13].

The benefits of quantization are especially clear on CPUs. Research shows that INT8 quantization can cut both latency and RAM use by more than half, making transformers practical on laptops or embedded devices [14]. For real-world systems like chatbots or voice assistants, this can mean the difference between an unusable model and one capable of serving millions of queries.

On GPUs, the picture is mixed. Data-centre accelerators such as NVIDIA’s T4 or A100 include optimised low-precision kernels and gain strongly from quantization. Consumer GPUs often lack this support. On older cards, quantized models may even fall back to CPU execution, removing any benefit. Even when kernels exist, overhead can offset the expected speed-ups. This makes hardware awareness crucial when applying quantization.

The latest frontier is ultra-low precision, with weights stored in as few as 4 bits. At this level, each weight is limited to 16 possible values, raising the risk of accuracy loss. Yet careful methods have shown surprising robustness. QLoRA demonstrated that, with proper scaling and parameter-efficient fine-tuning, 4-bit quantization can approach FP32 accuracy. The fact that a 65-billion-parameter model such as LLaMA could be fine-tuned in 4-bit on a single consumer GPU highlights the disruptive potential of this approach.

Portability is another concern. Frameworks like ONNX Runtime now provide standardised INT8 quantization workflows across CPUs and GPUs from different vendors. This ensures models can be deployed in varied environments without major re-engineering. Industry reports show that ONNX INT8 quantization often reduces latency by 30–40% with little loss in accuracy , making it a common choice for production use [11].

Overall, quantization is a simple idea, but one that has become very powerful in practice. It reduces memory use, accelerates inference, and enables deployment beyond specialised data centres. Its effectiveness, however, depends on hardware support and careful calibration, especially when working at very low precisions.

### 2.2.2 Pruning

If quantization reduces the font size of a book, pruning is like removing the pages that add little to the story. The idea is straightforward: not every parameter in a neural network is equally important. By removing the least useful ones, the model becomes smaller and often faster, while still keeping most of its predictive strength.

Pruning has a long history in deep learning. Many networks contain large amounts of redundancy, with up to 90% of parameters removable in some cases with only minor accuracy loss [15]. This result highlighted the potential for smaller storage footprints, lower transmission costs, and faster inference.

In transformers, pruning can be applied at different levels. Unstructured pruning removes individual weights, often those near zero, leaving behind sparse weight matrices. However, because common libraries and hardware are not designed to exploit irregular sparsity, these theoretical gains do not always translate into faster runtime. Structured pruning, on the other hand, removes whole neurons, filters, or attention heads. The matrices stay dense, which makes them easier to optimise on existing hardware. Many attention heads in BERT could be removed with little effect on accuracy, a key example of structured pruning [16].

More adaptive strategies have also been developed. Movement pruning identifies which weights to remove during fine-tuning, allowing the network to adjust to the specific task [17]. This treats pruning not only as a way to compress models but also as a kind of regularisation, which can improve generalisation while reducing computation.

But pruning also has limits. One recurring issue is that memory use often does not shrink unless specialised runtimes are employed. In frameworks such as PyTorch, zeroed-out weights are still stored in dense tensors, so RAM or VRAM consumption may remain unchanged. As a result, pruning tends to improve latency more than memory footprint. Even so, when combined with quantization or distillation, pruning can deliver useful efficiency gains, particularly for CPU-based inference.

### 2.2.3 Knowledge Distillation

While quantization and pruning reshape an existing model, knowledge distillation instead trains a smaller model to imitate a larger one. Hinton et al. [18] formalised the approach, showing that a “student” network trained on the softened probability outputs of a “teacher” can achieve strong performance despite having far fewer parameters. The key insight is that the teacher’s output distribution conveys richer information than raw labels alone.

Transformers have provided some of the best-known applications of distillation. DistilBERT compressed BERT-base by 40% while retaining more than 95% of its accuracy on the GLUE benchmark. It runs 60% faster at inference and requires significantly less memory, making it a widely adopted alternative for sentiment analysis, classification, and related tasks. TinyBERT [19] followed a similar path, distilling BERT into a compact model optimised for mobile devices. These cases show that distillation can be a practical way to make transformer models deployable.

Importantly, distillation often serves as a foundation for other techniques. A student model such as DistilBERT can then be quantized or pruned to push efficiency even further. In this way, distillation is not only a standalone compression method but also a stepping stone toward layered optimisation strategies.

These compression strategies reduce model size and cost, but adapting models for specific downstream tasks raises a different challenge. This is where parameter-efficient fine-tuning techniques such as LoRA and QLoRA come into play.

Table 1: Summary of core model compression techniques

| **Technique** | **Core Idea** | **Benefits** | **Limitations** |
|---|---|---|---|
| **Quantization** | Reduce numerical precision of weights/activations (e.g. FP32 → INT8/4-bit). | Cuts memory footprint (up to 75%), accelerates inference (esp. on CPUs), enables deployment on limited hardware. | Accuracy loss possible at very low precision; dependent on hardware/kernel support; overhead can negate gains. |
| **Pruning** | Remove unimportant weights, neurons, or attention heads (structured/unstructured). | Smaller effective model, reduced computation, potential latency gains. | Sparse matrices often not hardware-friendly; memory footprint may remain unchanged; risk of accuracy loss at high pruning rates. |
| **Knowledge Distillation** | Train a compact "student" model to mimic a large "teacher." | Produces entirely smaller architectures; efficient and fast inference; strong accuracy retention. | Requires access to teacher model; effectiveness depends on design/scale of student. |

## 2.3 Parameter-Efficient Fine-Tuning

Compression methods like quantization, pruning, and distillation improve efficiency during storage and inference, but another challenge appears when models are adapted to specific tasks. Traditional fine-tuning updates every parameter of a pre-trained model, which becomes impractical as models grow. Adjusting all 110 million parameters in BERT-base is already expensive. With several frontier LLMs now reaching hundreds of billions of parameters, full fine-tuning is out of reach for most researchers and organisations.

Parameter-efficient fine-tuning (PEFT) addresses this problem. Instead of retraining the entire network, PEFT introduces small trainable modules or transformations while the bulk of the model remains frozen. This reduces the number of trainable parameters by orders of magnitude, lowering both memory and compute requirements. Another advantage is modularity: one base model can support multiple task-specific variants simply by storing the added adapter weights, which are often less than 1% of the original model size.

### 2.3.1 LoRA

Low-Rank Adaptation (LoRA), proposed by Hu et al. [6], builds on the observation that the updates required during fine-tuning often lie in a much lower-dimensional space than the full parameter matrix. Concretely, given a weight matrix $W \in \mathbb{R}^{d\times k}$, LoRA freezes $W$ and instead learns two smaller matrices $A \in \mathbb{R}^{d\times r}$ and $B \in \mathbb{R}^{r\times k}$, where $r$ is much smaller than $d$ or $k$. The adapted weight is $W + AB$, with the trainable update $\Delta W = AB$. Because $r$ is small (often between 4 and 64), the number of trainable parameters is tiny compared to the original matrix.

In practice, LoRA reduces fine-tuning from hundreds of millions of parameters to only a few million. Empirical studies show that this often preserves accuracy across tasks while requiring a fraction of the resources of full fine-tuning. After training, the LoRA adapters can be merged back into the base model, so inference proceeds without extra overhead.

A useful analogy is music. A large model is like a symphony orchestra: retraining every musician for a new piece is time-consuming. LoRA is like hiring a handful of new players with small, specialised instruments who adjust the harmony while the orchestra plays unchanged. The core ensemble remains the same, but the performance adapts smoothly.

Because it is simple and effective, LoRA has quickly become one of the main tools for adapting large models. It allows faster experimentation, domain-specific fine-tuning, and efficient deployment in limited settings. It has also influenced a wave of related adapter methods, many of which are now included in libraries such as Hugging Face’s PEFT toolkit.

### 2.3.2 QLoRA

Quantized LoRA (QLoRA), introduced by Dettmers et al. [7], extends the idea by combining low-rank adaptation with quantization. The motivation is straightforward: even if only 1% of parameters are trainable, simply loading a base model in FP16 or FP32 can exceed the memory capacity of many GPUs.

QLoRA addresses this by storing the frozen base model in 4-bit precision while training LoRA adapters on top. During computation, the quantized weights are temporarily dequantized in chunks, while the small LoRA modules remain in higher precision for stability. This design drastically reduces memory requirements, making it possible to fine-tune very large models on a single high-memory GPU rather than a distributed cluster.

The impact has been significant. QLoRA enabled fine-tuning of models with tens of billions of parameters on hardware accessible to many research labs, leading to widely used derivatives such as the Guanaco family. These results illustrated a shift in accessibility: state-of-the-art model adaptation was no longer confined to a handful of industrial labs but opened to the broader community.

That said, QLoRA is not without risks. Quantizing to 4-bit can introduce numerical instability, and some models or tasks may be less robust under such constraints. Larger models with more redundancy appear to tolerate the approach better than smaller ones, but the method remains an area of active exploration.

Even with these challenges, QLoRA is still a major contribution to the field. By combining quantization with low-rank adaptation, it has made fine-tuning of extremely large language models feasible on hardware previously considered inadequate. Alongside LoRA, it represents a broader movement in NLP research; shifting from brute-force retraining of massive models to lightweight, accessible methods.

## 2.4 Benchmarks and Deployment

Benchmark studies make clear that efficiency gains from compression and optimisation cannot be evaluated in isolation from hardware. For example, BERT-base requires hundreds of milliseconds per sample on a CPU [3], which is far too slow for interactive use. DistilBERT reduces this latency but still places a significant load on commodity processors. These results underline the importance of systematic benchmarking: models that appear practical in one environment may fail in another.

Hardware differences are often decisive. Cloud GPUs with support for mixed-precision arithmetic can deliver strong throughput, while older or consumer-grade GPUs may lack optimised kernels altogether. In those cases, quantized models can revert to CPU execution, sometimes running slower than their full-precision counterparts. This means it is important to test compression methods on different devices to see their real benefits.

Software runtimes also shape deployment outcomes. Frameworks such as TensorFlow Lite and ONNX Runtime have extended quantization and optimisation support across CPUs, GPUs, and custom accelerators. In production contexts, INT8 ONNX models have been reported to cut inference latency by 30–40% with little loss in accuracy. These results show how software stacks can translate theoretical improvements into practical efficiency.

The most constrained environments are edge and mobile devices, which typically operate with limited memory and energy budgets. Compact architectures such as TinyBERT [19] and MobileBERT [20] are designed to meet these conditions, and further compression through pruning or quantization is often necessary to reach real-time performance.

Finally, deployment efficiency connects to sustainability. Large-scale inference consumes substantial energy, compounding the environmental cost of training [21]. Compression methods reduce computational overhead during serving, contributing to both economic and ecological sustainability.

In short, benchmarks show that hardware, runtime support, and energy budgets matter just as much as accuracy when judging if transformer models are feasible. These considerations frame the motivation for the experimental evaluation presented in the next chapter.

## 2.5 Research Gap

Research into transformer compression has progressed quickly. Methods such as quantization, pruning, knowledge distillation, and more recent approaches like LoRA and QLoRA, all show that it is possible to reduce model size and improve efficiency while keeping much of the original performance. These techniques are widely cited as promising answers to the challenge of scaling.

However, most studies test these methods in isolation and under favourable conditions, usually on high-end cloud GPUs or TPUs, with results reported mainly in terms of accuracy. Far less attention has been paid to deployment on constrained hardware. In practice, users working with laptops, consumer GPUs, or free-tier cloud CPUs face different issues: kernel support can be inconsistent, quantized models may fall back to CPU execution, pruning may not reduce memory use in reality, and fine-tuning methods like QLoRA can behave unpredictably on smaller models. These limitations are rarely discussed, as published work tends to emphasise headline results rather than failure cases.

This dissertation addresses that gap by providing a comparative evaluation of compression and parameter-efficient fine-tuning techniques across a range of hardware conditions. The aim is to present not only their strengths but also their limitations, offering practical evidence for researchers and practitioners who need to deploy transformer models outside of large data centres.

# 3. Methodology

## 3.1 Task and Dataset

The experiments in this dissertation are anchored in the Stanford Sentiment Treebank (SST-2), a binary sentiment classification dataset introduced by Socher et al. [9] and later incorporated into the General Language Understanding Evaluation (GLUE) benchmark suite [10]. As one of the most widely adopted tasks for evaluating transformer models, SST-2 provides an appropriate testbed for the compression and fine-tuning techniques explored in this project.

The task is straightforward: given a sentence extracted from a movie review, the model must classify its sentiment as either positive or negative. Unlike simple polarity datasets, SST-2 is built on expert-annotated sentences with grammatical structure, providing a more rigorous test of language understanding. Its binary format is well suited to this study: performance shifts attributable to compression methods can be detected clearly without the confounding factors of multi-class classification. At the same time, sentiment analysis has clear industrial relevance, with applications in customer feedback monitoring, conversational agents, and lightweight on-device assistant, precisely the scenarios where hardware resources may be limited.

The GLUE version of SST-2 contains around 67,000 training examples, 872 validation samples, and 1,821 test samples. While the original treebank included fine-grained sentiment labels, these are collapsed to binary form in SST-2. For the purposes of this dissertation, the validation set was used as the primary benchmark for reporting accuracy, since test labels are not publicly available. In certain resource-intensive experiments, such as QLoRA fine-tuning of BERT-base, a reduced training subset (e.g., the first 20,000 samples) was employed to balance computational feasibility with representativeness.

To ensure comparability across experiments, a unified preprocessing pipeline was applied. The dataset was accessed through the Hugging Face datasets library, tokenised with the appropriate WordPiece tokenizer for each model (e.g., distilbert-base-uncased or bert-base-uncased), and truncated to a maximum sequence length of 128 tokens. Dynamic padding was used within batches to optimise memory use, and sentiment labels were mapped to binary integers. Batch sizes were set to 16 by default, though adjusted in some fine-tuning runs depending on hardware constraints. The deliberate use of a standardised pipeline ensured that observed differences in accuracy, latency, or memory footprint could be attributed to the compression methods themselves rather than to inconsistencies in preprocessing.

In sum, SST-2 provides a compact, well-established benchmark that balances academic rigour with practical relevance. Its size and binary classification format allow systematic, hardware-conscious comparisons while remaining computationally manageable across CPUs, GPUs, and quantized or pruned variants of transformer models.

## 3.2 Models Under Study

This dissertation evaluates two widely used transformer-based models: BERT-base and DistilBERT [3,8]. They provide a useful contrast between a full-scale architecture and a distilled variant, making them suitable for testing the effects of compression and parameter-efficient fine-tuning.

BERT-base has 12 transformer layers and around 110 million parameters. It offers strong performance across NLP tasks but is relatively large for CPU inference, occupying over 400 MB in full precision. In this study, it serves as the uncompressed reference point against which efficiency methods are compared.

DistilBERT reduces BERT’s size by about 40% through knowledge distillation, using six layers and roughly 66 million parameters. It retains most of BERT’s accuracy while running faster and using less memory, which has made it a practical choice for sentiment analysis and classification tasks. Here it functions both as a baseline and as the main subject of further compression experiments.

Including both models reflects methodological and practical aims. DistilBERT shows how a distilled architecture behaves when compressed further, while BERT-base presents a more challenging case for aggressive methods such as 4-bit quantization and QLoRA. Together, they show how compression strategies work across different model scales.

## 3.3 Hardware and Experimental Environment

A central aim of this dissertation is to test whether transformer models can be deployed efficiently on limited hardware. To capture a realistic range of conditions, experiments were run in three environments: the Colab CPU backend, an NVIDIA T4 GPU provided by Colab, and a local machine with an NVIDIA GTX 1050 Ti GPU paired with an Intel Core i5-7500 CPU. Together these span the spectrum of computing resources available to many researchers and developers outside of specialised labs or industrial-scale infrastructure.

The Colab CPU served as a baseline, representing entry-level or free-tier environments. Hardware specifications vary by session but typically involve Intel Xeon processors with multiple cores at moderate clock speeds. Without GPU acceleration, all operations execute on general-purpose cores. This reflects many practical contexts (educational use, small-scale projects, production systems without GPUs, etc.) where transformer inference must run entirely on CPU.

The NVIDIA T4 GPU was chosen as a representative of cloud accelerators. With 16 GB of VRAM, Tensor Cores for mixed-precision arithmetic, and support for INT8 and FP16, it is widely available through Colab, AWS, and GCP. Positioned between consumer GPUs and flagship accelerators such as the A100 or H100, the T4 supports quantization kernels in libraries like bitsandbytes, making it an ideal testbed for measuring whether compression methods provide meaningful efficiency gains in a cloud setting.

The local environment, built around an NVIDIA GTX 1050 Ti with 4 GB of VRAM and an Intel i5-7500 CPU, represents older consumer-grade hardware. Released in 2016, the 1050 Ti lacks Tensor Cores and has no native support for low-bit operations, yet it remains common in desktops, laptops, and small servers. Including this setup was essential for evaluating deployment constraints often overlooked in academic benchmarks. As later results show, such hardware introduces unique challenges, including CPU fallback during quantized inference and atypical VRAM allocation behaviour.

All experiments were run in Python 3.10+ with PyTorch 2.1+ as the primary deep learning framework. Hugging Face Transformers provided pre-trained models and tokenizers, while Datasets handled SST-2 loading. Compression methods relied on established toolchains: bitsandbytes for 8-bit and 4-bit quantization, ONNX Runtime for framework-agnostic INT8 acceleration, and Hugging Face's PEFT library for LoRA and QLoRA. System monitoring combined Python's time module for latency, psutil for RAM, and nvidia-smi for GPU VRAM.

Memory usage was reported as retained consumption (ΔRSS for RAM, ΔVRAM for GPU), defined as end-state minus baseline, with GPU results also including total VRAM after model load.

Taken together, these hardware and software environments reflect realistic deployment conditions. The Colab CPU captures minimal-resource scenarios, the T4 GPU illustrates a widely available cloud accelerator with quantization support, and the GTX 1050 Ti demonstrates the limitations of older consumer devices. Their contrasting profiles form the basis for the cross-hardware analysis presented in Chapter 4.

Table 2: Hardware specifications used in experiments.

| **Environment** | **Processor / GPU** | **Memory (RAM/VRAM)** | **Notes** |
|---|---|---|---|
| **Colab CPU** | Intel Xeon (varies by session) | 12–16 GB RAM | No GPU acceleration; all inference on general-purpose CPU cores. |
| **Colab T4 GPU** | NVIDIA T4 (Turing architecture) | 12–16 GB RAM, 16 GB VRAM | Supports FP16 and INT8 via Tensor Cores; widely available in the cloud. |
| **Local Machine** | Intel Core i5-7500 + GTX 1050 Ti | 16 GB RAM, 4 GB VRAM | No Tensor Cores; limited or no support for low-bit quantization kernels. |

## 3.4 Compression and Parameter-Efficient Fine-Tuning Techniques

This project implemented three categories of efficiency methods: quantization, pruning, and parameter-efficient fine-tuning (LoRA/QLoRA). Each was chosen to reflect toolchains that are widely used in practice and to cover both inference-time and training-time efficiency.

### 3.4.1 Quantization

Quantization was explored through three different toolchains: PyTorch, bitsandbytes, and ONNX Runtime.

On CPU, the simplest approach was PyTorch’s dynamic INT8 quantization, applied to DistilBERT. In this method, the weights of linear layers are converted to 8-bit integers at runtime while activations remain in FP32. It requires no retraining and can be applied directly to pre-trained checkpoints via torch.quantization.quantize_dynamic. This setup was evaluated on the Colab CPU backend and served as a baseline for measuring the effect of quantization in resource-limited environments.

On GPU, quantization was tested with the bitsandbytes library, which integrates directly with Hugging Face model loading and supports both 8-bit and 4-bit precision. On the T4 GPU, this allowed us to test whether lower precision reduced VRAM usage and sped up inference compared with FP32. Because quantization occurs at model load, weights are stored and operated on in reduced precision from the start. With the T4’s kernel support for INT8 and INT4, direct comparisons with FP32 were possible. The GTX 1050 Ti, by contrast, lacks Tensor Cores and native low-bit arithmetic. Here, quantized models often fell back to CPU execution despite being loaded into GPU memory, leading to slower inference and illustrating the hardware dependence of quantization gains.

A third pathway used ONNX Runtime, which provides a portable, framework-agnostic solution. Models were first exported from PyTorch into ONNX format, then quantized post-training with ONNX’s dynamic quantization tools. Inference ran entirely within ONNX Runtime on the Colab CPU. Unlike PyTorch’s built-in quantization, ONNX added backend optimisations such as operator fusion and graph scheduling. In practice, it delivered a clear CPU-side latency reduction with minor accuracy impact; memory effects were negligible in our measurements.

Taken together, these three implementations cover a broad spectrum of current quantization strategies. PyTorch offers a lightweight option for CPUs, bitsandbytes enables low-bit experimentation on GPUs, and ONNX Runtime provides a cross-platform route geared toward deployment.

### 3.4.2 Pruning

Pruning was tested only on DistilBERT and carried out on CPU using PyTorch's pruning API. The method chosen was unstructured L1-norm pruning, which removes weights with the smallest magnitudes. It was applied to all linear layers at three sparsity levels: 30%, 40%, and 50%. Unstructured pruning was selected for its simplicity and fine-grained control over sparsity, even though it is less hardware-efficient than structured pruning.

A key drawback of PyTorch's framework is that zeroed weights remain stored in dense tensors, and pruning masks add further overhead. As a result, memory usage did not shrink—in fact, it increased slightly at higher sparsity. What did improve was inference speed: pruning consistently reduced latency on CPU. Because of this, latency rather than memory was treated as the main performance measure. To mirror realistic deployment pipelines, all pruning experiments were followed by INT8 dynamic quantization so that sparsity and low precision could be tested in combination.

### 3.4.3 Parameter-Efficient Fine-Tuning (LoRA and QLoRA)

Alongside inference-oriented compression, the project also tested parameter-efficient fine-tuning. The motivation was straightforward: updating all 110 million parameters of BERT-base would exceed the limits of a single T4 GPU. To make adaptation feasible, lightweight adapters were introduced using Low-Rank Adaptation (LoRA) and its quantized variant, QLoRA.

LoRA was applied through the Hugging Face PEFT library to DistilBERT's attention projection matrices ($r = 8$, $\alpha = 16$, dropout = 0.05). The base model was frozen and fewer than 1% of parameters were updated. Training ran for two epochs on a T4 GPU with a batch size of 16, and the adapters were merged back into the base checkpoint for evaluation. We did not run a separate LoRA experiment for BERT-base in this study.

QLoRA quantizes the frozen backbone to 4-bit (nf4 with double quantization) while training LoRA adapters in higher precision. In our runs, QLoRA failed on DistilBERT due to bitsandbytes kernel assertions, but it trained successfully on BERT-base ($r = 16$ on the q/v projections), enabling adaptation within the T4's 16 GB VRAM.

Together, LoRA and QLoRA represent the two main routes in parameter-efficient fine-tuning: low-rank adaptation to cut trainable parameters, and quantization-assisted fine-tuning to make large models trainable within modest hardware budgets.

## 3.5 Metrics and Measurement

To evaluate the impact of compression and fine-tuning techniques, three categories of metrics were used across the experiments: accuracy, latency, and memory consumption. Each captures a different aspect of efficiency, and together they allow meaningful comparison between hardware setups and methods.

Accuracy was measured on the SST-2 validation set. Predictions were obtained by applying a softmax to model logits and comparing the predicted label against the ground truth. For some notebooks, evaluation was limited to 100 validation samples for speed, while others used the full set. In all cases, accuracy was reported as the percentage of correct classifications. (For reproducibility: n3, n4, n5, and n7 evaluate on a 100-sample slice of the SST-2 validation set; n1, n2, n6, and n8 evaluate on the full validation set of 872 samples.)

Latency was measured at the inference level. On CPUs, elapsed time was recorded with Python's time module, either as per-sample averages (n1) or as total time for the full validation set (n2, n6). On GPUs, an additional synchronisation step ensured CUDA operations finished before timing was finalised, giving stable per-sample latency in milliseconds (n3–n5, n7, n8).

Memory consumption was tracked separately for system RAM and GPU VRAM. RAM usage was recorded with psutil as ΔRAM (ΔRSS), computed as end-of-inference minus a baseline taken immediately after model load (and any warm-up), where RSS denotes resident set size. GPU experiments also reported VRAM consumption using nvidia-smi, with both the total VRAM footprint after model load and the change during inference (ΔVRAM).

Not all experiments recorded the same set of metrics. CPU notebooks reported accuracy, latency, and RAM; n1 and n6 report ΔRAM (end − baseline), whereas n2 reports absolute RSS at end of evaluation (not a delta). GPU notebooks (n3–n5, n7, n8) additionally report VRAM after load and ΔVRAM during inference. In the pruning sweep (n2), we emphasise accuracy and latency because RSS does not decline under unstructured pruning with dense execution, even as sparsity increases.

By combining these measures—accuracy for predictive quality, latency for responsiveness, and memory for resource efficiency—the study was able to capture both the benefits and the trade-offs of compression methods in realistic hardware contexts.

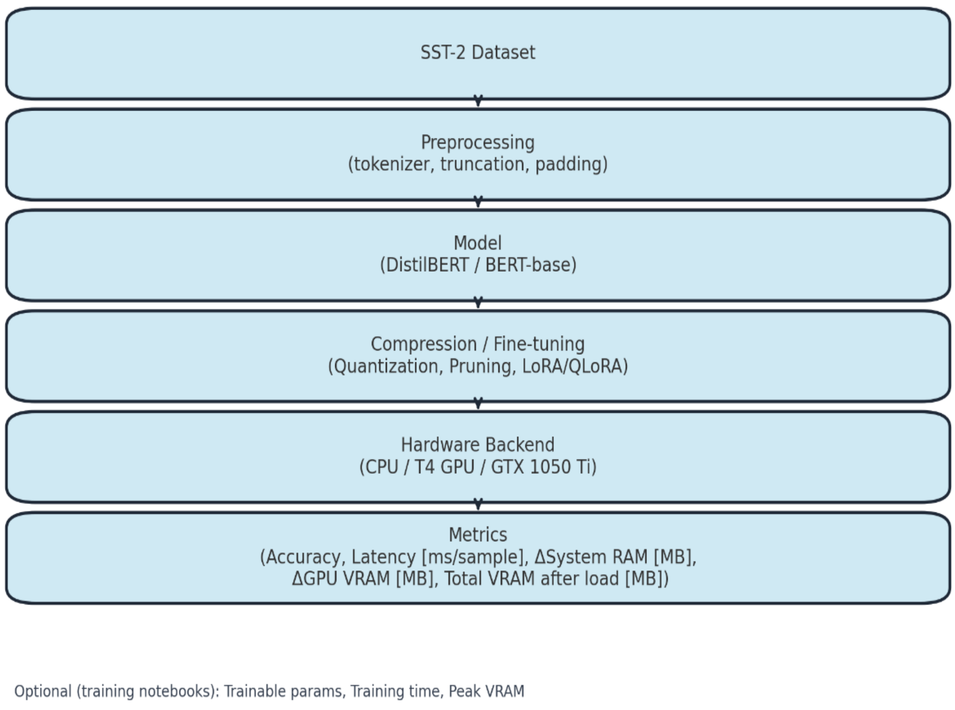
SST-2 Dataset
Preprocessing
(tokenizer, truncation, padding)
Model
(DistilBERT / BERT-base)
Compression / Fine-tuning
(Quantization, Pruning, LoRA/QLoRA)
Hardware Backend
(CPU / T4 GPU / GTX 1050 Ti)
Metrics
(Accuracy, Latency [ms/sample], ΔSystem RAM [MB],
ΔGPU VRAM [MB], Total VRAM after load [MB])
Optional (training notebooks): Trainable params, Training time, Peak VRAM

Figure 1: Overview of the experimental workflow.

## 3.6 Reproducibility and Limitations

Reproducibility was a guiding principle in this study. All experiments were implemented in Python 3.10+ with PyTorch 2.1+ as the core framework. Package versions were fixed within each notebook, and random seeds were set wherever possible, particularly for LoRA and QLoRA fine-tuning, to reduce variation from stochastic initialisation. Data loading and preprocessing were handled exclusively through the Hugging Face *Datasets* library, ensuring that tokenisation and batching were consistent across runs.

To make results transparent, each experiment was encapsulated in a dedicated Jupyter notebook (n1–n8). These notebooks include the full pipeline—from model loading and transformation to evaluation and metric collection—so that outputs can be traced directly back to their configuration. Memory was measured in a retained form: ΔRSS for system RAM and ΔVRAM for GPU memory, both defined as end-of-inference minus baseline usage. This convention was applied consistently across hardware setups.

Despite these measures, certain limitations remain. Google Colab, used for CPU and T4 GPU experiments, runs in virtualised environments where hardware can vary slightly between sessions, leading to fluctuations in absolute timings or RAM usage. Local experiments on the GTX 1050 Ti were more stable, but highlighted another issue: the lack of Tensor Cores meant quantized models often fell back to CPU execution, a behaviour that could not be avoided.

Measurement itself also has boundaries. ΔRSS and ΔVRAM provide reliable indicators of sustained memory consumption but may not capture transient spikes during execution. Similarly, latency was measured at the per-sample or batch level rather than throughput, so it reflects responsiveness but not large-scale serving capacity. Finally, some methods—most notably QLoRA applied to DistilBERT—failed due to kernel-level errors in the quantization backend. These cases were retained in the record to highlight the gap between theoretical promise and actual viability on constrained hardware.

In summary, the methodology prioritised consistency, transparency, and practical reproducibility, while also documenting the limits of the hardware and software used. These constraints frame the interpretation of results in Chapter 4, where both the strengths and weaknesses of each technique are considered.

# 4. Results

## 4.1 DistilBERT on CPU with Dynamic Quantization

The first experiment established a CPU-only baseline for DistilBERT and then applied 8-bit dynamic quantization using PyTorch's built-in functionality. This setting represents a minimal-resource deployment: inference was executed on the Colab CPU backend with no GPU acceleration, evaluating the full SST-2 validation set of 872 samples. Tokenization used a maximum sequence length of 128 tokens with padding and truncation, and models were assessed in FP32 and INT8 precision.

The FP32 baseline achieved 91.06% accuracy, confirming that DistilBERT maintains strong performance on sentiment classification even without GPU support. However, the computational cost was significant: average inference latency was 0.352 seconds per sample, with a system RAM increase of 273 MB. These figures highlight the practical challenges of running full-precision transformer models in CPU-only contexts, where both responsiveness and memory budgets are constrained.

Applying 8-bit dynamic quantization yielded clear efficiency gains. Accuracy dropped modestly to 89.33% (a reduction of  about 2 percentage points), but average latency fell by more than half to 0.151 seconds per sample. Additional RAM usage (ΔRAM) also collapsed to 5.8 MB, representing a reduction of over 95% compared with the FP32 run. While these figures describe retained memory deltas rather than total model footprint, they still illustrate the real efficiency improvements achievable with integer arithmetic on general-purpose CPU cores.

Attempts to apply 4-bit quantization were unsuccessful, as PyTorch's dynamic quantization backend currently supports only INT8 for linear layers. This reflects an important constraint: lower-precision methods are not universally available across frameworks and hardware, and unsupported configurations cannot yet deliver their theoretical benefits in CPU environments.

Overall, this experiment demonstrates that dynamic INT8 quantization is a practical method for CPU deployment of DistilBERT. It substantially reduces both latency and memory usage with only a minimal loss in accuracy, making it suitable for real-world applications where GPU acceleration is unavailable. The lack of 4-bit support underscores the dependency of compression gains on framework maturity, a theme revisited in later experiments.

Table 3: Results for DistilBERT on CPU (n1).

| **Mode** | **Accuracy** | **Latency (s/sample)** | **Δ RAM (MB)** |
|---|---|---|---|
| FP32 | 91.06% | 0.352 | 273 |
| INT8 | 89.33% | 0.151 | 5.8 |
| 4-bit | Unsupported | – | – |

## 4.2 DistilBERT on CPU with Pruning and Quantization

The second experiment extended the CPU baseline by combining dynamic 8-bit quantization with unstructured L1 pruning of DistilBERT. The aim was to test whether removing weights with small magnitudes could provide further efficiency gains on top of quantization, and whether such sparsity translated into measurable improvements in runtime and memory use on CPU.

Unstructured pruning was applied to all linear layers of the model at three sparsity levels: 30%, 40%, and 50%. After pruning, PyTorch’s dynamic quantization converted the pruned weights to INT8 at runtime. Unlike structured pruning, which eliminates whole neurons or heads, unstructured pruning removes individual weights. This gives fine-grained control over sparsity but is less hardware-friendly, since the underlying tensors remain dense and require binary masks for execution.

The results highlighted both advantages and limitations. Accuracy declined steadily with higher pruning: from 90.48% at 30% sparsity to 87.16% at 50%, a drop of just over three percentage points. At the same time, latency improved noticeably. End-to-end inference on the full 872-sample validation set dropped from 60.2 seconds at 30% pruning to 47.5 seconds at 50% pruning, confirming that fewer active weights can translate into faster CPU inference.

Memory behaviour was less encouraging. Despite the removal of nearly half the parameters, RAM usage actually increased, rising from 2330 MB at 30% pruning to 2697 MB at 50% pruning. This counterintuitive result reflects PyTorch's pruning mechanism: pruned weights remain allocated in dense tensors, while additional binary masks and quantization overhead introduce further memory costs. True memory savings would require a sparse-aware backend such as ONNX Runtime or DeepSparse, which were outside the scope of this experiment.

Taken together, the experiment illustrates the trade-offs of combining pruning with quantization on CPU. Latency improves with higher sparsity, but accuracy drops and RAM usage rises. In contexts where responsiveness is critical and memory is less constrained, pruning plus quantization can still provide value. However, for memory-limited deployments, the lack of RAM reduction underlines the limits of unstructured pruning in dense execution frameworks like PyTorch.

Table 4: Results of DistilBERT on CPU (n2) with pruning and 8-bit dynamic quantization.

| **Pruning + Quantization** | **Accuracy** | **Total Latency (s)** | **RAM (MB)** |
|---|---|---|---|
| 30% + INT8 | 90.48% | 60.2 | 2330 |
| 40% + INT8 | 88.53% | 56.3 | 2494 |
| 50% + INT8 | 87.16% | 47.5 | 2697 |

## 4.3 DistilBERT on T4 GPU with Quantization

The third experiment moved inference from CPU to GPU, using an NVIDIA Tesla T4 with 16 GB of VRAM on Google Colab. This setting enabled evaluation of DistilBERT in full precision (FP32) and with 8-bit and 4-bit quantization, using the bitsandbytes library. A subset of 100 validation samples from SST-2 was used to balance runtime feasibility with experimental consistency.

The FP32 baseline reached 94% accuracy, consistent with earlier CPU experiments, but with considerably faster inference on GPU. Average latency was 12.40 ms per sample, with a system RAM increase of 213.52 MB. GPU VRAM usage after model load was 659.9 MB, and inference introduced an additional 30 MB delta. These results confirm the efficiency of the T4 GPU when running uncompressed models, providing both speed and moderate memory consumption.

Quantization to 8-bit precision preserved accuracy at 94%, but latency increased substantially to 77.20 ms per sample—over six times slower than FP32. Memory usage patterns were mixed: system RAM increased by only 130.85 MB, less than the FP32 run, and inference-time VRAM delta fell to 14 MB. However, the total VRAM footprint grew to 843.9 MB, reflecting quantization overhead.

In the 4-bit configuration, accuracy dipped slightly to 93%. Latency averaged 16.50 ms per sample, slower than FP32 but markedly faster than INT8. RAM usage fell further to 5.73 MB, while the VRAM delta decreased to just 2 MB. Yet the total VRAM footprint increased again, reaching 953.9 MB. This indicates that while INT4 reduces activation and intermediate memory deltas, it requires additional allocations at load time, offsetting expected savings.

Taken together, these results highlight the trade-offs of GPU quantization for small transformer models. Accuracy was essentially unchanged, but FP32 remained the fastest and most memory-efficient option. Both INT8 and INT4 added overhead in terms of latency and total VRAM, though INT4 at least offered balanced performance compared to the sharp slowdown in INT8. These findings suggest that on modern GPUs, quantization benefits scale more effectively with larger models, where parameter compression provides more meaningful relative savings.

Table 5: Results for DistilBERT on T4 GPU (n3).

| **Mode** | **Accuracy** | **Latency (ms/sample)** | **RAM Δ (MB)** | **VRAM Δ (MB)** | **Total VRAM (MB)** |
|---|---|---|---|---|---|
| FP32 | 94.00% | 12.4 | 213.52 | 30 | 659.88 |
| INT8 | 94.00% | 77.2 | 130.85 | 14 | 843.88 |
| INT4 | 93.00% | 16.5 | 5.73 | 2 | 953.88 |

## 4.4 DistilBERT on GTX 1050 Ti with Quantization

The fourth experiment examined DistilBERT on an older, consumer-grade GPU: the NVIDIA GTX 1050 Ti with 4 GB of VRAM, paired with a local Intel i5-7500 CPU. Unlike the T4, the 1050 Ti lacks Tensor Cores and modern CUDA kernel support for low-bit operations, making it a useful case study of deployment on constrained, legacy hardware. The model was evaluated in FP32, INT8, and INT4 precision using the bitsandbytes backend, with 100 SST-2 validation samples.

The FP32 baseline produced stable results, achieving 94% accuracy with an average latency of 9.15 ms per sample. System RAM usage increased by 97 MB, while GPU VRAM rose modestly during inference (+20 MB) to a total of 1402.5 MB after model load. These figures confirm that full-precision inference is both functional and reasonably efficient on the 1050 Ti, albeit slower than on the T4.

The 8-bit configuration maintained accuracy at 94% but exhibited clear signs of hardware mismatch. Latency spiked dramatically to 98.4 ms per sample—more than ten times slower than FP32. System RAM usage increase fell to 48.7 MB, but the GPU VRAM delta was negative (−12 MB), signalling that execution had fallen back to the CPU even though the model appeared loaded on the GPU. The total VRAM footprint nevertheless rose to 1574.6 MB, a side-effect of bitsandbytes' internal overhead.

The 4-bit configuration achieved 93% accuracy, with latency averaging 11.3 ms per sample—slower than FP32 but far faster than INT8. RAM usage showed an anomalous negative delta (−4.9 MB), again consistent with CPU fallback. VRAM behaviour matched the INT8 case: a negative delta (−3 MB) during inference but a higher overall footprint (1715.1 MB) after model load. These patterns underline that although quantized models can be instantiated on the GPU, actual execution is offloaded to CPU kernels when hardware support is lacking.

In summary, the GTX 1050 Ti handled FP32 inference correctly but failed to benefit from INT8 or INT4 quantization. Instead of accelerating, quantized modes ran slower and consumed more memory, while negative VRAM deltas revealed CPU fallback. This contrasts with the T4 experiments (Section 4.3), where quantization at least preserved GPU execution despite added overhead. The findings highlight that without modern kernel support (compute capability ≥ 7.5), quantization methods are effectively unusable on older GPUs, reinforcing the need to align software techniques with hardware capabilities.

Table 6: Results for DistilBERT on GTX 1050 Ti (n4).

| **Mode** | **Accuracy** | **Latency (ms/sample)** | **RAM Δ (MB)** | **VRAM Δ (MB)** |
|---|---|---|---|---|
| FP32 | 94.00% | 9.15 | 97.4 | 20.13 |
| INT8 | 94.00% | 98.44 | 48.71 | −12.38 |
| INT4 | 93.00% | 11.27 | −4.93 | −3.00 |

## 4.5 BERT-base on T4 GPU with Quantization

The fifth experiment scaled up from DistilBERT to the full BERT-base model, which has around 110 million parameters—nearly twice the size of its distilled counterpart. The goal was to test whether the behaviour of quantization observed with DistilBERT (Section 4.3) would generalise to a larger architecture. The model was evaluated on a T4 GPU using the first 100 samples of the SST-2 validation set, comparing FP32, INT8, and INT4 inference modes via the bitsandbytes library.

The FP32 baseline achieved 92% accuracy, showing that BERT-base delivers strong sentiment classification performance in line with expectations. Average latency was 12.8 ms per sample, confirming that the larger model is slower than DistilBERT but still responsive in GPU-accelerated settings. System RAM usage increased by 63 MB, while GPU VRAM rose by 32 MB during inference, to a total footprint of about 1.29 GB. These figures illustrate both the higher computational cost of BERT compared to DistilBERT and the efficiency of the T4 in handling large models in full precision.

Moving to INT8 quantization preserved accuracy at 92% but introduced a major performance penalty. Latency jumped to 94.0 ms per sample, more than seven times slower than FP32. System RAM overhead dropped to 21 MB, and VRAM usage after model load fell slightly to 1.08 GB, with only a +12 MB delta during inference. These results suggest that while INT8 reduces memory footprint, the kernel overhead on the T4 makes it impractical for real-time inference.

The INT4 configuration also maintained 92% accuracy but offered a more balanced trade-off. Latency increased to 20.8 ms per sample, slower than FP32 but far better than INT8. System RAM usage was almost negligible at 0.5 MB, and GPU VRAM consumption settled at 1.20 GB, with just a +6 MB delta. This indicates that INT4 delivers meaningful memory efficiency and acceptable latency, though it still cannot match FP32 in speed.

Overall, these results highlight that quantization effects scale differently with larger models. Unlike DistilBERT, where quantization added overhead without clear benefit, BERT-base shows that INT4 quantization can provide a workable compromise between speed and efficiency on the T4 GPU. FP32 remains the fastest option, but INT4 achieves similar accuracy with lower system RAM and only a modest latency increase. By contrast, INT8 on T4 is prohibitively slow, underscoring that not all quantization settings are equally viable in practice.

Table 7: Results for BERT-base on T4 GPU (n5).

| **Mode** | **Accuracy** | **Latency (ms/sample)** | **RAM Δ (MB)** | **VRAM Δ (MB)** | **Total VRAM (MB)** |
|---|---|---|---|---|---|
| FP32 | 92.00% | 12.83 | 63.26 | 32 | 1289.9 |
| INT8 | 92.00% | 94.04 | 20.96 | 12 | 1081.9 |
| INT4 | 92.00% | 20.75 | 0.51 | 6 | 1195.9 |

## 4.6 DistilBERT on CPU with ONNX Runtime Quantization

The sixth experiment evaluated DistilBERT inference on CPU using ONNX Runtime, with the aim of testing whether exporting models to ONNX and applying dynamic INT8 quantization could deliver efficiency gains over PyTorch’s built-in path. Like the earlier CPU experiments, this evaluation used the full SST-2 validation set (872 samples), with tokenized inputs truncated to a maximum length of 128.

The FP32 baseline produced an accuracy of 91.06% and required 191.68 seconds to process the validation set. This confirms the robustness of DistilBERT even without GPU acceleration but highlights the relatively high cost of CPU-only execution.

Quantizing the exported model to INT8 preserved accuracy almost completely at 90.48%, while reducing inference time to 121.08 seconds. This represents roughly a 37% speed-up, showing that ONNX Runtime can substantially accelerate transformer inference on CPUs. In both FP32 and INT8 modes, process-level RAM deltas read as ~0 MB. This likely reflects limitations of the psutil-based measurement and allocator reuse, rather than a literal absence of memory consumption differences. The result underlines that ONNX optimizations primarily impact latency rather than footprint in this configuration.

Unlike in Section 4.2, pruning was not combined with ONNX. PyTorch's pruning masks are not preserved in ONNX export, so the runtime executes dense weights regardless of sparsity applied beforehand. As a result, this experiment isolates quantization effects alone.

In summary, ONNX Runtime provided a practical and portable acceleration path for CPUs. With minimal accuracy loss, it reduced inference time by over one-third compared to PyTorch FP32 execution. While RAM deltas could not be reliably measured, the latency improvements are unambiguous. For deployment on CPUs, particularly where latency is the key bottleneck, ONNX offers a clear advantage over native PyTorch execution.

Table 8: Results for DistilBERT on CPU with ONNX Runtime (n6).

| **Mode** | **Accuracy** | **Total Latency (s)** | **ΔRAM (MB)** |
|---|---|---|---|
| FP32 | 91.06% | 191.68 | ~0 |
| INT8 | 90.48% | 121.08 | ~0 |

## 4.7 DistilBERT with LoRA Fine-Tuning on T4 GPU

The seventh experiment shifted focus from pure inference compression to parameter-efficient fine-tuning, using Low-Rank Adaptation (LoRA) to adapt DistilBERT on the SST-2 dataset. The aim was to evaluate whether small transformer architectures could benefit from lightweight adapters within the memory limits of a single T4 GPU, and to test whether QLoRA (4-bit quantization plus LoRA) was viable for models of this scale.

LoRA adapters were inserted into the attention projection matrices with a low rank (r = 8, α = 16, dropout = 0.05). This setup reduced the number of trainable parameters to fewer than 0.5% of the total model size, while the rest of the model was frozen. Training proceeded for two epochs on SST-2, and the adapters were merged back into the base model afterwards so that evaluation reflected a single unified checkpoint.

The LoRA-adapted model fine-tuned successfully. Validation accuracy reached 90.48% on the full SST-2 dev set (872 samples), only slightly below DistilBERT's standard fine-tuned benchmark. Training required only a few minutes on the T4 GPU, with peak VRAM usage around 1–2 GB, confirming that LoRA dramatically reduces the cost of task adaptation. Inference latency after fine-tuning averaged 10.5 ms per sample across the validation set, with a stable VRAM footprint of ~1 GB. These figures demonstrate that LoRA makes fine-tuning feasible within modest hardware budgets without sacrificing much predictive performance.

Attempts to apply QLoRA were unsuccessful. Multiple training runs failed with a bitsandbytes AssertionError stemming from kernel-level mismatches. This shows that QLoRA, though effective for larger architectures (see Section 4.8), is not yet compatible with smaller models such as DistilBERT.

In summary, this experiment highlights the value and limitations of parameter-efficient fine-tuning for small transformers. LoRA proved to be a practical, lightweight method, delivering ~90% accuracy with minimal memory and runtime cost. QLoRA, however, could not be applied successfully, pointing to implementation constraints in current quantization toolchains.

Table 9: Results of DistilBERT with LoRA and attempted QLoRA fine-tuning on T4 GPU (n7).

| **Model** | **Trainable / Total Params** | **Training Time (min)** | **Peak VRAM (train, MB)** | **Accuracy** | **Latency (ms/sample)** | **VRAM After Load (MB)** |
|---|---|---|---|---|---|---|
| DistilBERT + LoRA (FP16) | 0.295M / 67.25M (0.44%) | ~few minutes | ~1000–2000 | 90.48% | 10.5 | ~1022 |
| DistilBERT + QLoRA (4-bit) | 0.295M / 46.02M (0.64%) | – | – | Failed | – | – |

## 4.8 BERT-base with QLoRA Fine-Tuning on T4 GPU

The final experiment applied Quantized Low-Rank Adaptation (QLoRA) to BERT-base, extending the investigation from DistilBERT to a full 110 million parameter model. QLoRA integrates 4-bit quantization of the frozen backbone with low-rank adapters trained in higher precision, making it possible to fine-tune large-scale transformers within the 16 GB VRAM available on a single T4 GPU. This directly addressed the challenge of adapting BERT-base in resource-limited environments, where conventional full-precision training would typically exceed hardware budgets.

Two training regimes were compared. In the full fine-tuning configuration, all weights were updated for two epochs on SST-2 using a learning rate of 2e-5. This setting represented the traditional approach to adapting BERT-base. In the QLoRA configuration, the backbone was quantized to 4-bit NF4 precision, while LoRA adapters were introduced with rank 16 on the query and value projection matrices ($\alpha = 32$). These adapters comprised just over half a million parameters, or 0.54% of the total model, leaving the rest of the backbone frozen. Training proceeded for two epochs with a learning rate of 2e-4, reflecting the different optimisation dynamics of adapter-based fine-tuning.

The results highlight the efficiency advantages of QLoRA. Full fine-tuning required 14.6 minutes of training and peaked at approximately 2.5 GB of VRAM, updating all 109 million parameters. Validation accuracy reached 91.6%, consistent with strong published baselines for BERT-base on SST-2. Inference was equally demanding: latency averaged 7.6 milliseconds per sample, with VRAM consumption of roughly 421 MB after model load and evaluation peaks of 1.5 GB.

By contrast, QLoRA fine-tuning required only 7.6 minutes of training and reduced peak VRAM during training to 1.6 GB. Validation accuracy remained competitive at 90.6%, less than one percentage point lower than the FP32 baseline. Crucially, inference performance improved markedly. Latency dropped to 3.3 milliseconds per sample, VRAM after model load was reduced to just 99 MB, and evaluation peaks remained well below 1.2 GB. These findings demonstrate that QLoRA not only enabled BERT-base fine-tuning on a T4 GPU, but also delivered significant efficiency gains during inference, reversing the trend observed with earlier quantization experiments that had increased memory usage.

In summary, QLoRA proved to be a practical and effective strategy for adapting large transformer models on constrained hardware. While it introduced a minor accuracy trade-off, it nearly halved training time, reduced memory consumption in both training and inference, and improved latency by more than a factor of two. Unlike the failed DistilBERT QLoRA attempt in Section 4.7, BERT-base provided a scale where QLoRA's benefits could be fully realised, making it an attractive approach when efficiency is paramount.

Table 10: Results of BERT-base with full fine-tuning (FP32) and QLoRA on T4 GPU (n8).

| **Model** | **Trainable / Total Params** | **Training Time (min)** | **Peak VRAM (train, MB)** | **Accuracy** | **Latency (ms/sample)** | **VRAM After Load (MB)** | **Peak VRAM (eval, MB)** |
|---|---|---|---|---|---|---|---|
| BERT FP32 | 109,483,778 / 109,483,778 (100%) | 14.59 | 2526.1 | 0.9163 | 7.6 | 420.99 | 1513.7 |
| BERT QLoRA (4-bit, r=16) | 591,362 / 109,483,778 (0.54%) | 7.55 | 1638.9 | 0.906 | 3.25 | 99 | 1124.7 |

# 5. Discussion of results and recommendations

## 5.1 Interpretation of results

Across all eight experiments the picture is consistent. On CPU, post-training INT8 quantisation delivers practical gains: notably shorter runtimes with minimal accuracy loss. The clearest latency win occurs with ONNX Runtime; PyTorch dynamic INT8 also improved latency and, in our run, reduced measured ΔRAM. On the T4 GPU, full precision remains the fastest mode for DistilBERT. For BERT-base, the FP32 model is still quickest among its FP32/INT8/INT4 inference variants, while the separately trained QLoRA variant delivers even lower latency and memory during inference. Quantised paths preserve accuracy yet often fail to convert theoretical arithmetic savings into wall-clock speed; on DistilBERT they also increased total VRAM, whereas on BERT-base both INT8 and INT4 slightly reduced total VRAM but still ran slower than FP32. On a legacy GPU (GTX 1050 Ti), quantised inference is not a viable serving strategy. Having unsupported kernels can force computation back to the CPU despite the model appearing to load on the device. Unstructured pruning on CPU reduces latency under dense execution but does not reduce RAM, and the accuracy cost increases in a predictable way with sparsity. Parameter-efficient fine-tuning proves valuable for training within budget; LoRA on DistilBERT did not accelerate inference, whereas QLoRA on BERT-base improved inference latency and reduced VRAM.

The CPU experiments show why integer inference remains a dependable path for constrained deployments. Exporting DistilBERT to ONNX and applying INT8 quantisation reduced total processing time on the SST-2 dev set (872 samples) from 191.68 seconds to 121.08 seconds; in that notebook, ΔRAM was approximately 0 MB for both FP32 and INT8 (ΔRAM/ΔVRAM are defined in Chapter 3 §3.5 as end minus baseline). PyTorch's dynamic INT8 path also improved efficiency, lowering the average per-sample latency from 0.352 seconds to 0.151 seconds and reducing ΔRAM from 273 MB to 5.8 MB. Taken together, both routes materially improved CPU inference with minimal accuracy change. ONNX INT8 gave the lowest total-set time in our run and adds portability; PyTorch dynamic INT8 is a reliable drop-in when export is impractical.

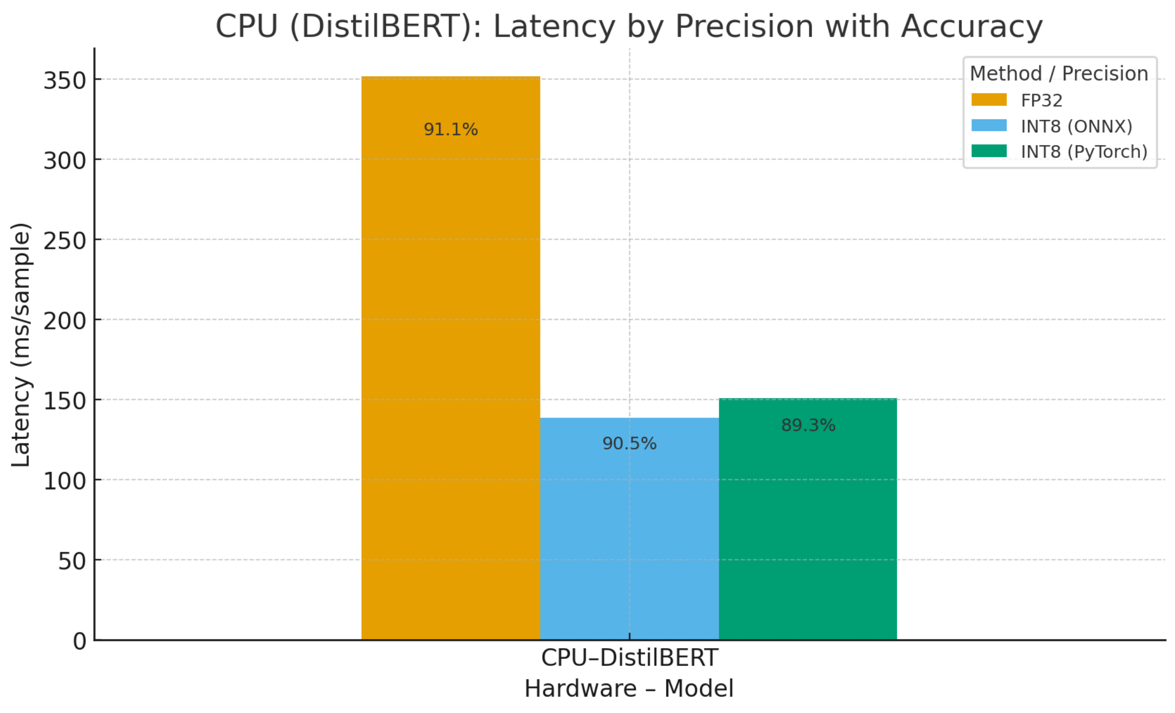

Figure 2: Latency and accuracy of DistilBERT on CPU under different precision settings (FP32, INT8 ONNX, INT8 PyTorch).

Results on the T4 GPU tell a different story. For DistilBERT, FP32 achieved the lowest latency at 12.40 milliseconds per sample, and the total VRAM after model load was around 660 MB. The quantised modes maintained accuracy yet ran slower and used more GPU memory; INT8 measured 77.20 milliseconds per sample and increased total VRAM markedly, with INT4 showing similar behaviour. For BERT-base, FP32 again delivered the best latency at 12.83 milliseconds per sample, while INT8 and INT4 were slower despite preserving task accuracy, and both used slightly less total VRAM than FP32 in our run. Quantised inference on GPUs often requires dequantisation/requantisation at boundaries where fused integer kernels are not available for the full transformer block. Those boundaries introduce memory movement and kernel-launch costs. For encoder-style workloads with many attention and feed-forward substeps, the extra traffic can outweigh the theoretical savings of lower-precision arithmetic. Because FP32 kernels on T4 are mature and these models fit comfortably in memory, full precision is faster. In this stack, inference-only quantisation is a capacity tactic rather than a speed tactic (see Figure 3).

The behaviour on older consumer hardware underscores the need to confirm not only that a quantised model loads on a GPU, but also that the kernels actually execute there. On the GTX 1050 Ti, the quantised DistilBERT runs were slower than the FP32 baseline and presented negative ΔVRAM during inference, which strongly indicates that computation fell back to the

CPU. Concretely, FP32 produced about 9.15 milliseconds per sample. INT8 took 98.44 milliseconds per sample and showed a ΔVRAM of −12.38 MB even though memory was allocated at load time. INT4 followed the same pattern. These figures show the lack of suitable low-bit kernels and the risk that load-time indicators do not reflect where execution happens. In conclusion, on the 1050 Ti it is safer to run FP32 on the GPU or ONNX INT8 on the CPU, since a quantised model file does not guarantee GPU acceleration on older cards (see Figure 3).

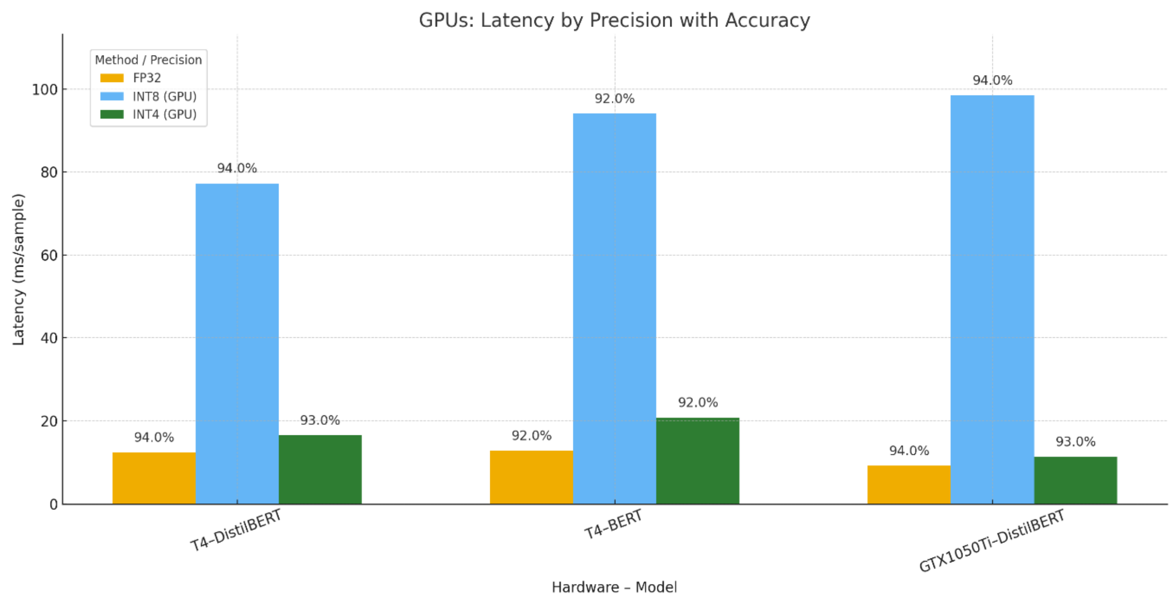

Figure 3: Latency and accuracy of DistilBERT and BERT on GPUs (T4, GTX 1050 Ti) under different precision settings (FP32, INT8, INT4).

Pruning on CPU behaved predictably under dense execution. Increasing unstructured sparsity from 30 percent to 50 percent provided modest reductions in total latency while RAM increased (process RSS), because pruned layers continue to be represented as dense tensors with masks. In the most aggressive setting evaluated, end-to-end time improved to 47.5 seconds but end-of-evaluation RSS rose to 2697 MB. This outcome is expected: compute can skip some multiply-adds but storage remains dense. Under the measurement regime adopted in Chapter 3, compute reduces slightly, yet memory does not contract unless the serving stack can exploit sparsity structurally. The accuracy declines steadily as sparsity increases, with no surprising behaviour at the levels tested. The correct reading is that unstructured pruning is a usable latency lever for CPU inference when a few seconds per full pass matter in practice. However, it is not a memory optimisation unless paired with a sparse-aware runtime or export flow that can take advantage of the pattern (see Chapter 4, Table 4).

Parameter-efficient fine-tuning separates training feasibility from serving efficiency, which is precisely why it is useful in constrained settings. For DistilBERT with LoRA, the fine-tuned model preserved accuracy after adapters were merged and, once evaluated, FP32 remained the fastest inference mode on T4. One concrete comparison makes the point. For DistilBERT + LoRA, inference ran at 10.5 milliseconds per sample on the full validation set with a ~1.0 GB VRAM footprint after load. QLoRA on DistilBERT failed to train due to bitsandbytes errors, but on BERT-base it succeeded and improved inference latency to 3.25 milliseconds per sample with ~99 MB VRAM after load, compared with 7.60 milliseconds and ~421 MB for the FP32 fine-tuned model. In short: LoRA did not change DistilBERT's serving cost, while QLoRA materially improved BERT-base serving cost. After fine-tuning, serving choices should follow hardware logic: FP32 remains fastest for DistilBERT on T4; for BERT-base, the QLoRA model is the most efficient on T4; and on CPU, ONNX INT8 remains the practical path (see Table 9 and Figure 4).

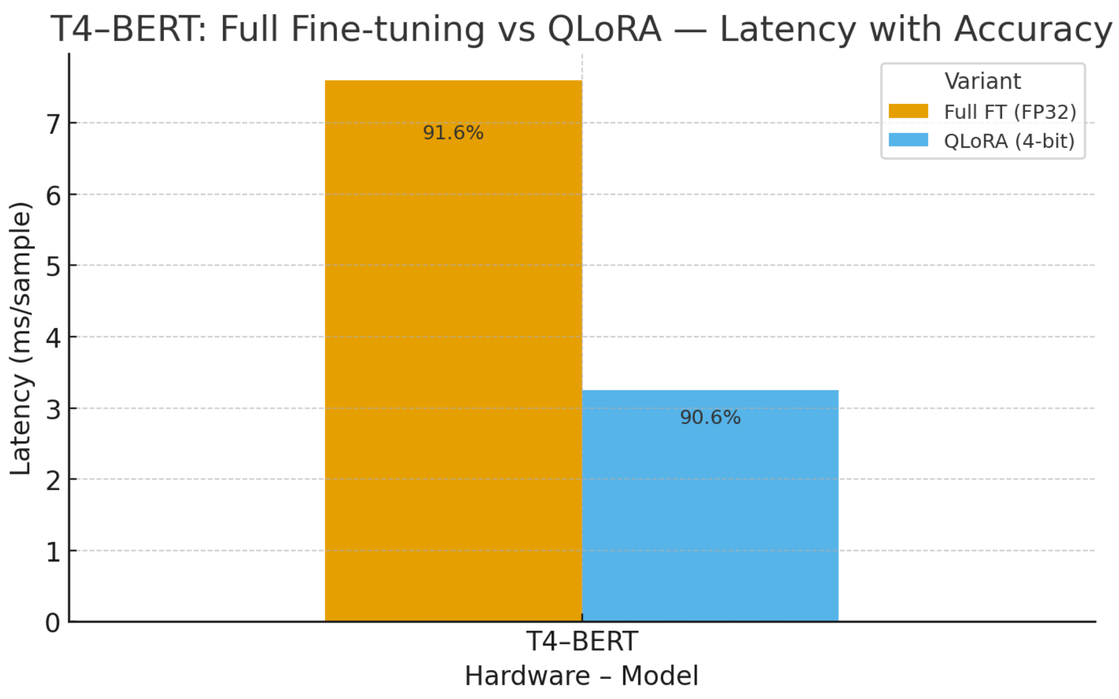

Figure 4: Latency and accuracy of BERT on T4 GPU under full fine-tuning (FP32) and QLoRA (4-bit).

Taken together, the results cohere. CPU cores benefit directly from integer arithmetic and from the graph-level planning that ONNX Runtime applies, so the pathway from lower precision to lower wall-clock time is short and well supported. The T4 sits in a regime where full-precision fused kernels are mature and memory is sufficient for the models under study, so

the extra work that inference-only quantised paths introduce is not repaid in speed; for DistilBERT it also raises total VRAM, while for BERT-base it slightly lowers total VRAM without closing the latency gap. Older GPUs without kernel support simply cannot execute these quantised workloads natively, so even when a model appears to reside on the device, the runtime may push computation back to the CPU with predictable effects on latency and memory measurements. Pruning behaves as theory suggests under dense execution. It trades accuracy for some latency relief and does not shrink memory. LoRA and QLoRA play their intended role by allowing adaptation with limited resources; at inference time, LoRA leaves DistilBERT's cost profile roughly unchanged on T4, whereas QLoRA at the BERT-base scale improves both latency and VRAM.
Two points about methodology matter for interpretation. First, GPU latency was measured per sample with explicit CUDA synchronisation. This means the results show single-request responsiveness, not maximum throughput. Under this setting, extra overheads such as kernel launches and dequantisation/requantisation steps weigh heavily in the T4 quantised runs. Second, all experiments standardised tokenisation and truncated sequences to 128 tokens. The reported gains (CPU INT8) and non-gains (T4 INT8/INT4 for inference-only models) therefore reflect short-sequence encoder inference; the QLoRA-trained BERT-base is the notable exception with improved latency on T4. Much longer sequences could increase arithmetic intensity and could change the absolute timings, but not the overall ordering in this stack.

It is important not to over-generalise any single result. The findings do not claim that GPU quantisation is always slower, only that in the configurations tested here —which include specific models, hardware, and library versions— quantised paths did not beat full precision on latency, and memory effects were mixed: DistilBERT increased total VRAM, whereas BERT-base reduced it slightly. The same caution applies to pruning; there are runtimes that exploit sparsity and they can recover memory along with speed, but the stack used in this study executes pruned weights in dense form, so memory does not decline. Parameter-efficient fine-tuning should not be evaluated solely on serving speed. It was adopted to make fine-tuning feasible on limited resources, and it met that goal; additionally, at the BERT-base scale QLoRA did improve inference latency and memory. Where adapters don't alter serving cost (e.g., DistilBERT + LoRA), their primary gains are in training cost and data-to-quality efficiency (see Chapter 3, §3.6).

## 5.2 Actionable recommendations

This section translates the evidence into deployment guidance with clear defaults, the conditions under which they hold, and concrete validation steps before adopting any configuration in production.

For CPU-only serving (edge, offline, or cost-sensitive VMs), the most reliable default is to export the model to ONNX and run it with INT8 quantisation. In our tests, this consistently cut end-to-end time with only small accuracy loss; ΔRAM improvements were negligible in ONNX (~0 in our runs), while PyTorch dynamic INT8 did reduce ΔRAM substantially (see Chapter 4, Tables 3 and 8). A low-risk adoption process is simple: export the checkpoint, check operator coverage, run a small accuracy test, then measure latency and ΔRAM on the full dev set with the same tokenisation and truncation settings. Keep threading parameters fixed across runs to avoid noise, and repeat measurements a few times to get stable averages. If export is blocked by an operator or integration limit, PyTorch dynamic INT8 is still a useful fallback, though without ONNX's scheduling advantages.

For latency-sensitive serving on T4, the sensible default is FP32 for the BERT-family encoders tested. In our results, quantised modes preserved accuracy but did not improve latency; VRAM effects were mixed (higher for DistilBERT, slightly lower for BERT-base) (see Chapter 4, Tables 5 and 7). If quantisation is attempted, treat it as an experiment with a clear acceptance rule—for example, at least a 10% gain in p50 latency (median) over FP32 with no extra VRAM after load. Measurements should include warm-up and explicit synchronisation. They should report both p50 and p95 latency (median and 95th percentile), since user responsiveness depends on tail as well as average performance. Device utilisation and allocator traces help confirm that kernels run on the GPU and catch regressions that simple metrics may miss. In practice, the main reason to use quantisation on T4 is capacity, not speed: it may let a very large model fit when memory is tight, but if the model already fits, the overhead of dequantisation and limited kernel fusion usually cancels the arithmetic savings. Where responsiveness is the main goal, FP32 remains the default. This guidance concerns inference-only quantisation; PEFT/QLoRA is addressed separately below.

On legacy or consumer GPUs like the GTX 1050 Ti, the safest options are FP32 on the GPU or ONNX INT8 on the CPU; quantised GPU serving should not be assumed viable. In our tests, quantised runs on the 1050 Ti often fell back to CPU execution even though the model appeared to load on the device, leading to slower latency and negative ΔVRAM. Before adopting any quantised setup on such hardware, confirm execution with device-level utilisation and allocator traces, not just load-time indicators. A simple rule applies: if latency gets worse or ΔVRAM turns negative, treat it as a hard failure and revert to the stable defaults. This avoids wasting effort on a kernel path the device cannot support.

For task adaptation under tight budgets, parameter-efficient fine-tuning (PEFT) is the most practical route. In our setting, LoRA and QLoRA met accuracy targets while staying within a single T4's training resources. Merge adapters for LoRA (DistilBERT) before evaluation and deployment; for BERT-base with QLoRA, serve the 4-bit backbone with the LoRA adapters attached (unmerged), mirroring our evaluation setup and common practice. At inference, the hardware defaults remain unchanged. For DistilBERT, LoRA fine-tuning did not materially alter serving cost on the T4, and QLoRA was not compatible in our runs; therefore, FP32 remains the recommended T4 serving mode for DistilBERT, with ONNX INT8 as the CPU default. For BERT-base, QLoRA provided tangible serving benefits relative to the FP32 fine-tuned model, improving latency and reducing VRAM with only a small accuracy decrease. Accordingly, when that trade-off is acceptable, serve the QLoRA-trained BERT-base on T4; otherwise, use FP32 on T4 and ONNX INT8 on CPU.

Two practices help improve reliability on any hardware. First, use a pre-production checklist for precision changes. This should cover version pinning (framework, ONNX Runtime, bitsandbytes if used, CUDA, cuDNN, and the driver), containerised execution for stability, and seed control for reproducibility. Every candidate configuration should be tested with the same preprocessing, sequence settings, batch size, and concurrency as in Chapter 4 and in the target production profile. Second, use a consistent reporting format. For online inference, record p50 and p95 latency after warm-up and synchronisation, plus VRAM after load and ΔVRAM during inference. For CPU runs, report total elapsed time, ΔRAM, RAM total and accuracy.

Capacity planning benefits from clear headroom policies. On GPU, aim to keep VRAM after load below about 80% of total capacity to reduce the risk of fragmentation and sudden OOM failures under bursty traffic. On CPU, track ΔRAM as well as absolute RSS to spot creeping memory use from data-dependent allocations or threading changes. When a model approaches its headroom limit, follow a simple order of options: on T4, choose a smaller encoder that runs comfortably in FP32 before trying quantisation for speed; on CPU, prefer ONNX INT8 before cutting sequence length or batch size, since INT8 delivered the largest improvement without reducing throughput.

Deployment changes should use guarded roll-outs. Start with a short canary phase that runs automated checks on latency, error rates, ΔVRAM/ΔRAM, and a "no negative ΔVRAM on GPU" rule. This gives confidence that a new setup works under real traffic. If the canary fails, automatically roll back to the last stable configuration. Keep logging lean but informative—record model ID, commit hash, precision mode, and runtime versions—so any regression can be traced quickly to its source.

Finally, make observability part of the setup itself. Use lightweight monitoring that tracks latency distributions, error codes, and the Δ metrics for each request group. These signals act as early warnings of performance drift. They can also expose hidden fallbacks—for example, a rise in latency alongside a steady drop in ΔVRAM usually means execution has silently moved back to the CPU. Because our experiments showed clear differences across precision modes and devices, these checks act as guardrails to keep the system within the boundaries validated in Chapter 4.

Overall, these recommendations provide safe defaults for each deployment setting, along with a small set of validation steps to avoid common mistakes. They also offer a clear, repeatable path to roll out or roll back changes. The scope is intentionally narrow: we do not propose new models, losses, or datasets, and we avoid speculative tweaks that our results cannot support. The next section discusses the limits of these conclusions and where future kernels or runtimes could shift the trade-offs observed here.

## 5.3 Limitations and future work

This study is practical by design. It asks what happens to latency and memory when we change precision or pruning settings on real hardware, under a specific toolchain, and with controlled preprocessing. That narrow scope keeps the results reproducible, but it also leaves out cases that matter in other deployments. The points below set the limits clearly and point to follow-ups that would test how far the findings travel.

First, the hardware range is limited. We report two anchors: a T4 GPU common in cloud inference and a legacy GTX 1050 Ti that appears in cost-constrained setups. The T4 results show that in the inference-only quantization experiments (n3–n5), FP32 remained faster than low-bit modes for the BERT encoders tested; the 1050 Ti results show that quantised models can appear to load on-GPU while actually executing on the CPU. Those are important operational signals, but they should not be read as universal. Newer GPUs (e.g., L4/A10/A100/H100) and other vendors ship different kernel sets, memory hierarchies, and compiler stacks. A direct replication on at least one newer datacentre GPU and one newer consumer GPU would test whether the same ordering holds, especially for low-bit paths that may be better fused than on the T4.

Second, the model family is narrow. We evaluated encoder-only transformers (DistilBERT and BERT-base). Decoder-only and encoder–decoder models have different operator mixes and fusion opportunities, which can change how precision interacts with kernels and memory. The conclusions here should therefore be read as characterising short-sequence encoder inference, not generation workloads or long-context models. Extending the grid to include one decoder-only and one seq2seq model would help separate what is “BERT-specific” from what is general.

Third, the sequence length is fixed. All experiments used standardised tokenisation and truncation to 128 tokens. That isolates precision effects but also biases the results toward interactive, low-arithmetic-intensity cases. With much longer sequences, compute dominates overhead, and absolute latencies will rise; the relative ranking between FP32 and low-bit paths could also shift. A simple extension is to repeat the runs at 128/256/512 tokens and report how latency and memory scale for each mode.

Fourth, we measured GPU performance one request at a time. For each sample we timed until CUDA finished, so the numbers reflect interactive response. This makes small overheads stand out—kernel launches and dequantisation/requantisation steps. That is why low-bit modes on the T4 did not reduce wall-clock time. We did not measure throughput with large batches or mixed concurrency. Adding those tests would round out the picture for queued or bulk workloads.

Fifth, the evaluation sets differ in size across some experiments. Many GPU runs use a 100-sample slice of SST-2 for quick, controlled comparisons, while the ONNX CPU experiment processes the full validation set (872 samples). That choice served each experiment’s purpose but reduces comparability at a glance. The results still support the conclusions within each hardware class, yet a fully uniform evaluation pass (same sample size, same reporting fields) would remove this source of confusion.

Sixth, metric consistency can be improved. We report ΔRAM and ΔVRAM as “change relative to baseline,” which captures runtime deltas but not full footprints or allocator fragmentation. We also did not use p50/p95 on CPU, while we emphasised per-request latency on GPU. In future runs, reporting both absolute and delta memory, plus p50/p95 on CPU, would give a clearer view across environments and avoid over-interpreting a single number.

Seventh, pruning was evaluated under dense execution and not through the ONNX path. In PyTorch, unstructured pruning adds masks but keeps dense tensors, so latency can drop slightly while memory rises; those results match what we observed. We did not test structured sparsity (e.g., block patterns) or sparse-aware runtimes, and ONNX exports in our setup did not preserve pruning masks, so we did not combine pruning with the ONNX INT8 flow. Testing structured patterns that align with kernel support, and exporting to runtimes that store and execute sparsely end-to-end, would show whether pruning can deliver both latency and memory wins.

Eighth, quantisation coverage depends on versions and kernels. The low-bit GPU results reflect the specific library versions and kernels available in this stack (PyTorch, bitsandbytes, CUDA/cuDNN/driver). Coverage and fusion change over time. Small upgrades can flip an operator from "dequantise → FP16/FP32 → requantise" to a fully fused INT8 path, changing both speed and memory. While Chapter 4 pins versions, a controlled "one-component-at-a-time" version sweep would quantify how sensitive the conclusions are to stack drift.

Ninth, our PEFT results depend on the chosen ranks, the layers targeted, and whether adapters are merged. For DistilBERT with LoRA, we merged the adapters before evaluation. For BERT-base with QLoRA, we served the 4-bit backbone with adapters unmerged, which reflects typical deployment. These choices isolate serving cost for LoRA and mirror real-world practice for QLoRA, but different ranks or layer targets could shift accuracy or latency slightly. A small hyperparameter sweep (rank, α, target layers) and a direct comparison of merged versus unmerged serving would test robustness without changing the overall method.

Tenth, some choices in the notebooks were pragmatic and leave room to tighten discipline. Examples include mixing small slices and full sets across experiments, relying on ΔRAM/ΔVRAM without always showing absolute footprints, and not carrying the pruning tests through the ONNX export. None of these invalidate the direction of the results, but they do make the story harder to compare line-by-line. A single reporting template and a uniform harness would solve this.

Future work should therefore be incremental and testable. A useful next pass would (i) replicate the grid on a newer datacentre GPU and a newer consumer GPU, using the same per-sample methodology; (ii) try an alternative GPU backend that promises deeper INT8/INT4 fusion and profile dequantise/requantise boundaries; (iii) sweep sequence length to map scaling of latency and memory across modes; (iv) add throughput and concurrency experiments alongside per-sample latency; (v) evaluate structured sparsity with a sparse-aware runtime and report both memory and time; (vi) run a small PEFT hyperparameter grid and compare merged vs. unmerged serving; (vii) add a second CPU backend/OS to test portability; and (viii) ship a lightweight validation script that fails any configuration showing negative ΔVRAM on intended GPU paths or a mismatch between device utilisation and expected execution.

Taken together, these limits explain why the recommendations in this chapter are conservative. They fit the measured stack and the models we studied. The next steps above probe where the findings hold, where they change, and which combinations of hardware and runtime turn theoretical low-bit savings into real improvements in latency and memory.

# 6. Conclusion

This dissertation asked a simple question: can transformer models be made practical on everyday hardware, not just in large data centres? To explore this, the study evaluated three compression families—quantization, pruning, and knowledge distillation—and two parameter-efficient fine-tuning methods, LoRA and QLoRA. The setup was kept consistent across tasks and notebooks, and the same models were run on three different environments: a standard CPU, an NVIDIA T4, and an older GTX 1050 Ti. The aim was to look beyond accuracy charts and focus on what actually determines deployability: latency, memory, kernel support, and runtime behaviour.

The main finding is clear. On CPUs, post-training INT8 quantization works reliably and is often the best first step. Both PyTorch and ONNX Runtime produced substantial latency reductions with minimal accuracy loss, and ONNX delivered the largest and most portable gains. For serving transformers on laptops or small servers, an ONNX INT8 model offers a strong and dependable baseline.

On the T4 GPU, results were more mixed. Quantized inference in INT8 or 4-bit generally preserved accuracy, but it did not consistently beat FP32 in speed once kernel overheads and data movement were included. In practice, FP32 often remained the fastest path. This runs counter to the common expectation that lower precision is always faster on GPU, but it reflects a practical truth: without mature kernels and fusions for the exact workload, theoretical gains may not appear at end-to-end level.

The GTX 1050 Ti made the hardware dependence even more visible. Quantized models frequently fell back to CPU execution because kernel support was incomplete, which erased expected benefits and sometimes made performance worse. In that setting, the safe options were FP32 on the GPU or INT8 on the CPU via ONNX. “GPU” is not a single category; age, architecture, and library support matter as much as the model itself.

Pruning helped, but in a focused way. Removing weights or whole structures reduced computation and improved latency, especially on CPU. Memory use, however, rarely dropped in standard dense runtimes because zeroed weights are still stored and irregular sparsity is not exploited. Accuracy declined as sparsity increased, which is expected. Overall, pruning acted primarily as a latency tool. When memory is the bottleneck, it needs to be combined with quantization or deployed with a sparse-aware runtime to deliver real footprint savings.

Distillation remained a solid foundation. DistilBERT provided a lighter and faster baseline without severe accuracy losses and still benefited from INT8 quantization on CPU. In practice, a distilled model paired with ONNX INT8 travels well across machines and keeps performance in a useful range for interactive use.

LoRA and QLoRA addressed a different challenge. By freezing most parameters—and, in the case of QLoRA, storing the base model in 4-bit during training—they made task adaptation feasible under tight VRAM budgets. This significantly improved accessibility for fine-tuning. These methods are not inference accelerators, though. After training, serving speed still depends on kernels and runtimes. Adapters help obtain the model that is needed; quantization and the execution stack determine how fast it runs.

The experiments also surfaced failure cases that matter in practice: silent CPU fallbacks when GPU kernels are missing, counter-intuitive VRAM behaviour with quantized paths, and sensitivity to library versions. These are part of the result, not footnotes. Reporting only accuracy misses the real obstacles to deployment. A more complete picture includes device residency checks, kernel summaries, and end-to-end latency measured under the exact conditions claimed to be supported.

The contribution of this work is threefold. It offers a hardware-aware view of compression and adapter methods that includes negative results and edge cases, not just best-case accuracy. It provides a reproducible setup—shared data pipeline, comparable batching, and uniform metrics—across eight notebooks that span CPUs, a T4, and a 1050 Ti, using both PyTorch and ONNX. And it turns the evidence into practical guidance: prefer ONNX INT8 on CPU when resources are tight; on datacentre GPUs, measure FP32 against quantized paths rather than assuming speedups; avoid quantized inference on older consumer GPUs unless kernel support is verified; use pruning mainly to reduce compute unless sparse-aware execution is available; and treat LoRA or QLoRA as ways to make fine-tuning affordable, not as substitutes for inference-time optimisation.

There are limits to acknowledge. The hardware range is varied but not exhaustive, and kernel support evolves. The task is SST-2 classification, so conclusions should be re-examined for long-sequence and generative workloads, where attention costs and batching strategies differ. Measurements target interactive, low-batch scenarios; high-throughput serving may change the trade-offs. Even with these caveats, the patterns are consistent across runs and align with practical experience: kernels and runtimes often decide outcomes as much as algorithms and architectures.

Future work should focus on closing the gap between theory and runtime reality. Structured sparsity with vendor-supported kernels could turn pruning’s theoretical memory savings into actual footprint reductions. Operator-level profiling and better fusion on GPUs may unlock the speed promised by low-precision arithmetic. A wider sweep across hardware—including newer datacentre GPUs, ARM CPUs, AMD GPUs, and emerging NPUs—and across workloads such as long-context inference and generation would test how far these conclusions travel. Quantization-aware training for smaller models may also stabilise ultra-low precision where post-training methods are brittle.

In the end, the message is simple. Efficiency is not only about clever algorithms; it is about the fit between those algorithms and the hardware and software that run them. On today’s commodity systems, ONNX-based INT8 on CPU is a dependable win. On GPUs, full precision is often still competitive unless quantized paths have mature kernel support for the exact workload. By focusing on what actually runs fast and fits in memory, this dissertation offers a realistic path for deploying transformers when resources are limited.

## Reference list / Bibliography

[1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All You Need. In *Advances in Neural Information Processing Systems* 30 (NeurIPS 2017), 5998–6008.

[2] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling Laws for Neural Language Models. arXiv:2001.08361 [cs.LG].

[3] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of NAACL-HLT 2019*. ACL, Minneapolis, MN, 4171–4186.

[4] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurélien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971 [cs.CL].

[5] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL].

[6] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. LoRA: Low-Rank Adaptation of Large Language Models. arXiv:2106.09685 [cs.CL].

[7] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2023. QLoRA: Efficient Finetuning of Quantized LLMs. arXiv:2305.14314 [cs.CL].

[8] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2019. DistilBERT, a Distilled Version of BERT: Smaller, Faster, Cheaper and Lighter. arXiv:1910.01108 [cs.CL].

[9] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, Andrew Y. Ng, and Christopher Potts. 2013. Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank. In *Proceedings of EMNLP 2013*. ACL, Seattle, WA, 1631–1642.

[10] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. 2018. GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding. In *Proceedings of the 2018 EMNLP Workshop BlackboxNLP*. ACL, Brussels, 353–355.

[11] Microsoft. 2021. Accelerate and Quantize Pretrained Transformer Models with ONNX Runtime. Technical Report. Microsoft Corporation. https://onnxruntime.ai

[12] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. 2018. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. In *Proceedings of CVPR 2018*. IEEE, 2704–2713.

[13] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and others. 2018. Mixed Precision Training. In *Proceedings of ICLR 2018*.

[14] Shijie Wu, Alexey Romanov, and James Cross. 2020. INT8 Quantization for Transformer Inference. arXiv:2009.08066 [cs.CL].

[15] Song Han, Huizi Mao, and William J. Dally. 2016. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. In *Proceedings of ICLR 2016*.

[16] Paul Michel, Omer Levy, and Graham Neubig. 2019. Are Sixteen Heads Really Better than One? In *Advances in NeurIPS 2019*. 14014–14024.

[17] Victor Sanh, Thomas Wolf, and Alexander M. Rush. 2020. Movement Pruning: Adaptive Sparsity by Fine-Tuning. arXiv:2005.07683 [cs.CL].

[18] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the Knowledge in a Neural Network. arXiv:1503.02531 [cs.CL].

[19] Xiaoqi Jiao, Yichun Yin, Lifeng Shang, Xin Jiang, Xiao Chen, Linlin Li, Fang Wang, and Qun Liu. 2020. TinyBERT: Distilling BERT for Natural Language Understanding. In *Findings of EMNLP 2020*. ACL, Online, 4163–4174.

[20] Zhiqing Sun, Hongkun Yu, Xiaodan Song, Renjie Liu, Yiming Yang, and Denny Zhou. 2020. MobileBERT: A Compact Task-Agnostic BERT for Resource-Limited Devices. In *Proceedings of ACL 2020*. ACL, 2158–2170.

[21] Emma Strubell, Ananya Ganesh, and Andrew McCallum. 2019. Energy and Policy Considerations for Deep Learning in NLP. In *Proceedings of ACL 2019*. ACL, 3645–3650.

## Appendix 1

Repository: https://github.com/KyriakosTop/bert-distilbert-compression

Notebooks (at frozen tag): https://github.com/KyriakosTop/bert-distilbert-compression/tree/v1.0/notebooks

Notebook index (at tag v1.0):

| **ID** | **File (permalink)** | **Hardware / Runtime** | **What it includes (1-liner)** |
|---|---|---|---|
| n1 | https://github.com/KyriakosTop/bert-distilbert-compression/blob/v1.0/notebooks/n1_dbert_quant_cpu.ipynb | **CPU** (Colab) | DistilBERT on CPU: **FP32 vs INT8** (PyTorch dynamic); full SST-2 dev; reports accuracy, **per-sample latency**, ΔRAM. |
| n2 | https://github.com/KyriakosTop/bert-distilbert-compression/blob/v1.0/notebooks/n2_dbert_quant_prun_cpu.ipynb | **CPU** (Colab) | DistilBERT with **unstructured L1 pruning (30/40/50%) + INT8**; full dev; reports accuracy, **total latency**, process **RSS**. |
| n3 | https://github.com/KyriakosTop/bert-distilbert-compression/blob/v1.0/notebooks/n3_dbert_quant_gpu_t4.ipynb | **T4 GPU** (Colab) | DistilBERT on GPU: **FP32 / INT8 / INT4** (bitsandbytes); 100-sample slice; accuracy, **ms/sample**, ΔRAM, **ΔVRAM + VRAM after load**. |
| n4 | https://github.com/KyriakosTop/bert-distilbert-compression/blob/v1.0/notebooks/n4_dbert_quant_gpu_gtx.ipynb | **GTX 1050 Ti** (local) | DistilBERT on legacy GPU: **FP32 / INT8 / INT4**; 100-sample slice; same metrics; shows **CPU fallback** behaviour. |
| n5 | https://github.com/KyriakosTop/bert-distilbert-compression/blob/v1.0/notebooks/n5_bert_quant_gpu_t4.ipynb | **T4 GPU** (Colab) | **BERT-base** on GPU: **FP32 / INT8 / INT4**; 100-sample slice; accuracy, ms/sample, ΔRAM, **ΔVRAM + VRAM after load**. |
| n6 | https://github.com/KyriakosTop/bert-distilbert- | **CPU** (Colab) | DistilBERT with **ONNX Runtime**: **FP32 vs INT8**; full dev; accuracy, **total latency**; ΔRAM noted (~0). |

| | | | |
|---|---|---|---|
| | compression/blob/v1.0/notebooks/n6_dbert_onnx_cpu.ipynb | | |
| n7 | https://github.com/KyriakosTop/bert-distilbert-compression/blob/v1.0/notebooks/n7_dbert_lora_gpu_t4.ipynb | **T4 GPU** (Colab) | **LoRA** fine-tuning for DistilBERT (works) and **QLoRA attempt** (fails); reports params, train time/VRAM, accuracy, **ms/sample**. |
| n8 | https://github.com/KyriakosTop/bert-distilbert-compression/blob/v1.0/notebooks/n8_bert_qlora_gpu_t4.ipynb | **T4 GPU** (Colab) | **BERT-base**: **FP32 full fine-tune vs QLoRA (4-bit)**; reports params, train time/VRAM, accuracy, **ms/sample**, **VRAM after load / peak**. |