Benchmarking Large Language AI Models for Machine Translation

Wassim Jabrane

Master of Science
Artificial Intelligence
School of Informatics
University of Edinburgh
2023

Abstract

Machine translation, a fundamental task in natural language processing (NLP), holds exceptional significance as it bridges communication gaps across diverse languages and cultures. As large language models increase in size, this brings additional computational costs. Libraries like Archer, a high-performance inference engine, aim to optimise the deployment of these models to reduce the resource requirements needed to utilise their predictive capabilities. This research aims to support these tools by developing a benchmarking suite for machine translation to aid in addressing the bottleneck in performance, both in terms of the quality of output and the efficiency of model execution. Through constant refinement, the suite's architecture embodies a systematic approach to evaluating machine translation models that can be later expanded to support other metrics. In addition, the work done provides a building block for benchmarking not only trained models' inference runtime, and could be used to help the development of inference libraries to further the development of assisting with serving machine learning models. Finally, using the framework, we analyse the results reported from the dense models T5 and NLLB and then on sparse, larger models such as SwitchTransformer and NLLB-MoE deployed using Archer.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Wassim Jabrane*)

Acknowledgements

I would like to thank my dear family and close ones as well as my friends for their support. I would also like to mention my gratitude towards my supervisor Luo Mai and Leyang Xue for their guidance, mentorship, and continuous support. I owe a lot of my success to those around me, and I am forever grateful for the kindness, knowledge acquired, and skills developed in my last five years at the University of Edinburgh.

Table of Contents

Chapter 1

Introduction

Benchmarking AI models is an essential component of model development to inform engineers and researchers about their performances. Recently, Large Language Models (LLMs) have demonstrated significant improvements over the years in research with the success and accessibility of services like ChatGPT [1]. Large models in the current stage of research, generally tend to scale up to trillions of parameters: 1.76 trillion for GPT-4 [2]. Standardising these models and making them more feasible to run in smaller-scaled hardware is the ultimate goal of this research aims to explore.

Deployment of large AI models which would require substantial memory capacity means higher performance infrastructure requirements as the model gets larger. Our focus will be exclusively on the task of machine translation, the field of utilising machine learning to translate text from one language to another. Navigating the domain of machine translation is difficult due to the need to capture and handle textual data which is high-dimensional and sparse, though this is a problem for all natural language processing tasks. In addition, it presents a challenging task when working to translate to low-resource languages: languages with a limited amount of dataset available, and when working with extremely high-dimensional data.

The sheer scale of these models has led to new avenues of research devising strategies and optimisation techniques for a more resource-efficient operation. These involve building optimised training and inference libraries to speed up model computation, for example Deepspeed, [3] developed by Microsoft, and Archer, an internal project run at the University of Edinburgh. For instance, with the development of Mixture of Expert [4] models characterised by their sparsity, these tools leverage this characteristic to their advantage. One such technique is disk offloading, which is the premise of utilising additional hardware resources to store model weights. Despite the additional

I/O overhead of copying memory, the main goal is to develop strategies to move these weights when needed to deploy the sparse models and push for minimal latency and throughput.

For both reasons of deploying models with quality and speed of performance in mind, the development of robust benchmarking tools to accurately assess the performance is essential. Work like MLPerf Inference Benchmark Suite [5] aims to address the gap of ensuring a standardised benchmark system to measure the efficiency of the model inference time. The main contribution of this project is to research and aim to work on providing a support benchmarking tool that can be used for the development of more powerful AI models and the development of inference engine libraries to address bottlenecks in designing the system.

Specifically, we will aim to use five models to assess the benchmarking of translating from English to French from two datasets: WMT14 [6] and FLORES-200 [7]. The models being assessed in these experiments are NLLB [7], T5 [8], and their sparse variations NLLB-MoE [7] and SwitchTransformer [9] inspired by Mixture of Expert architecture computed on NVIDIA GPU (see Table 2.1 for the hardware specification). The following thesis will be broken down into these main chapters:

- Chapter 2 - Background: This chapter briefly describes the NVIDIA GPU architecture and its hardware components that are exclusively designed to support AI model inference. We then extensively explain the dense models NLLB and T5 and their sparse versions NLLB-MoE and Switch Transformers, and the metrics used for benchmarking, and the related work on inference engines.

- Chapter 3 - Benchmark Design and Implementation: This chapter describes the software framework of the benchmarking suite and the components of its architecture.

- Chapter 4 - Experiments: Using the developed suite, we perform extensive experimentations via different hyperparameter configurations. We then assess the produced results via the reported analytics from the models, with and without using Archer.

Finally, we will conclude this paper in chapter 5 by recommending future enhancements which could expand metrics contributing to the evolving landscape of machine translation technologies and running large-scale models in devices with limited computing resources.

Chapter 2

Background & Related Work

2.1 Introduction to NVIDIA Ampere Architecture

The graphics processing unit, GPU, comprises numerous processing units that are capable of executing arithmetic and logic operations concurrently. As such, it is the main foundational electronic circuit that enables running AI models, which involves running matrix calculations from the model weights along with the model input.

Ampere Architecture, GPU, is a computing architecture that is developed by NVIDIA [10]. The server-side GPU that we utilise for our experiments, NVIDIA Geforce RTX 3090, employs this architecture as part of the Ga10x lineup [10]. Additional specification on the server-side machine used on the experiment is found in Table 2.1. It promises improvements over its predecessors, with the introduction of third-generation tensor cores. They are specialized hardware units designed to accelerate matrix operations and are optimized to perform mixed-precision matrix multiplications. This dynamic versatility of calculations helps accelerate throughput while preserving accuracy. Another example is structured sparsity, which is a concept that refers to imposing deliberate patterns of zero elements in data or models to achieve efficiency and reduce redundancy, which could help cut down on model size. This can be useful for natural language tasks, as data sparsity is common due to the large vocabulary sizes that lead to high-dimensional data. By strategically removing unnecessary components, a model's size can be reduced, making it faster to train, requiring less memory, and being more suitable for deployment on resource-constrained devices or in scenarios with limited computational power. Other features include NVLink technology, promising fast communication between other GPUs if available, and PCIe 4.0, enabling faster data transfer rates between the GPU and other hardware components.

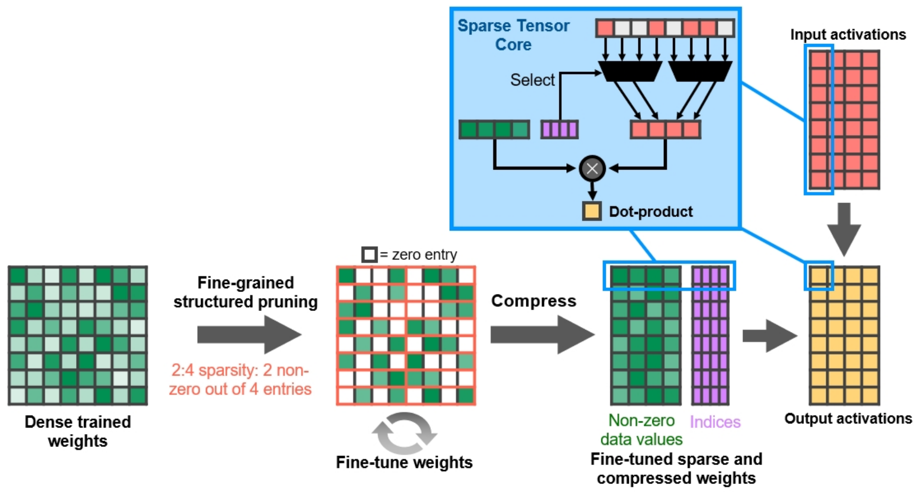


Figure 2.1: Visual representation of the Structured Sparsity technique, from the Ampere GA102 paper [10]. The original trained weights get pruned with a 2-out-of-4 non-zero pattern before being compressed. The result minimises data footprint and bandwidth, which fastens the operation of passing through a layer.

To serve large language models on GPUs, it is vital to utilise all components of hardware architecture such as the CPU, GPU, and SSD. Frameworks like Deepspeed, which will be discussed in section 2.4, help with this through different techniques, one of which is offloading memory to storage devices like SSD.

GPU Count	1
GPU Name(s)	NVIDIA GeForce RTX 3090
CPU Model	AMD Ryzen 9 5950X 16-Core Processor
GPU Driver Version	525.105.17
Framework	PyTorch 2.0.1

Table 2.1: The specifications of the architecture used to benchmark the models

2.2 Transformers

Machine learning is a field that uses algorithms that learn from input data to generate predictions on unseen inputs. Deep learning, being a subset of that field, performs the same task - with the only major distinction being that it adopts artificial neural networks. At their core, transformers are neural networks that process and generate sequences of data, making them particularly suited for tasks involving string(s) of text, such as language translation, text generation, sentiment analysis, and more [11]. The baseline transformer model contains 65 million parameters. Their structure is characterised by having an encoder block and a decoder block as the two main components. The encoder layer processes the input, which is then routed to the decoder layer which generates the output. In this section, we will briefly describe the inner workings of the dense transformer model before diving into the different variations of the transformer models that are used for our experiments.

The encoder and decoder components both comprise multi-head attention and feed-forward neural networks. The feedforward layer converts the output produced by the self-attention sublayer through a non-linear transformation, which aids in increasing the complexity of the model to improve its predictions. The multi-head self-attention mechanism calculates attention scores for different words within the sequence simultaneously, allowing the model to capture various types of relationships. To further illustrate, the concept of attention is a process of assigning weights to elements of input data, enabling models to emphasize relevant information and disregard irrelevant aspects. This allows the model to capture complex relationships and be able to formulate a representation of the input. In the case of self-attention, the attention mechanism is applied within a single sequence, allowing the model to weigh the importance of different positions relative to each other. By assigning different attention weights, the model can capture short and long-range dependencies between words, enhancing its understanding of the sequence. This is useful for language modelling, predicting the likelihood of a sequence of words in a given context, which makes it a fundamental basis behind many domains in natural language processing including machine translation. Self-attention can have variations, such as multi-head attention, which allows the model to capture different types of relationships in parallel, enhancing its capability to understand complex patterns within the input sequence. One of the significant advantages of transformers is their ability to process sequences in parallel rather than sequentially, which significantly speeds up training and inference unlike other sequence-to-sequence models like Recurrent Neural Networks (RNN) [12]. Additionally, the attention computations make transformers highly interpretable, as they provide insights into which parts of the input sequence contribute more to specific parts of the output sequence.

The self-attention sublayer consists of query, key, and value vectors to the attention weights, which determine the relevance of each input embedding. Afterwards, the final output of the self-attention sublayer is obtained after calculating the weighted sum of the value vectors. Other relevant components of the model include Residual

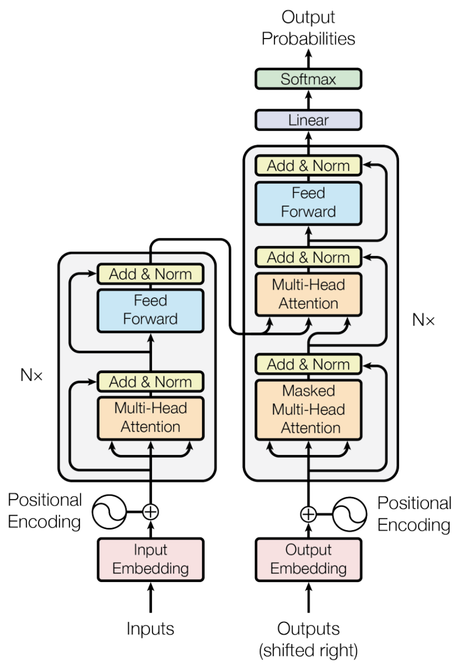


Figure 2.2: The Transformer Architecture [11]

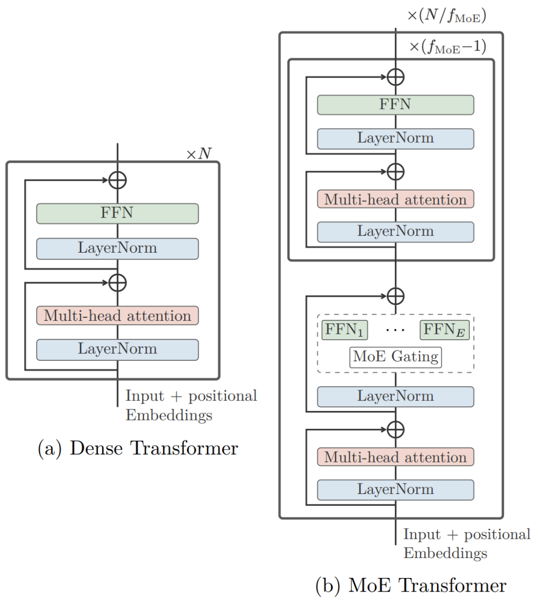


Figure 2.3: Mixture of Experts (MoE) Design Architecture. Note the similarities and differences between the layers in a Dense Transformer and MoE Transformer. Mainly, the FFN layer is replaced by the MoE Gating Extension of Transformer [7].

connections [13] to mitigate vanishing gradients, layer normalisation [14] to stabilise training, and positional encoding [11] to capture information on the order of the tokens in the sequence. They each serve their purpose in improving the performance of the model training and output.

As a result, transformers have inspired various architectures and models, such as BERT (Bidirectional Encoder Representations from Transformers) [15], GPT (Generative Pre-trained Transformer) [16], and T5 (Text-to-Text Transfer Transformer) [8]. They are often trained on large amounts of text data and fine-tuned for specific tasks, resulting in state-of-the-art performance.

2.2.1 T5

T5, or Text-to-Text Transfer Transformer, is a neural network model that is inspired by the Transformer design [8]. T5 extends the Transformer's capabilities by approaching most NLP tasks. It reformulates all tasks (during both pre-training and fine-tuning) with a text-to-text format, meaning that the model receives textual input and produces

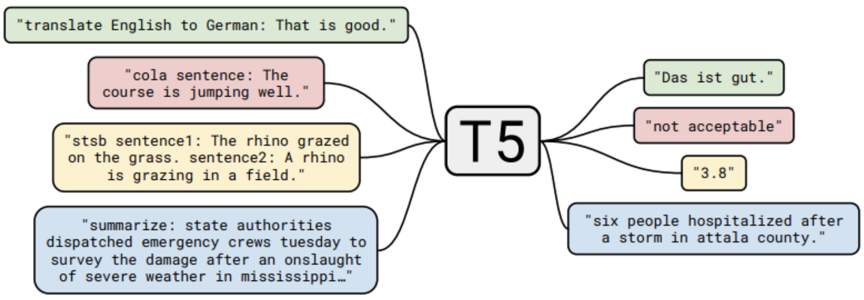

"translate English to German: That is good."
"cola sentence: The course is jumping well."
"stsb sentence1: The rhino grazed on the grass. sentence2: A rhino is grazing in a field."
"summarize: state authorities dispatched emergency crews tuesday to survey the damage after an onslaught of severe weather in mississippi…"
T5
"Das ist gut."
"not acceptable"
"3.8"
"six people hospitalized after a storm in attala county."

Figure 2.4: Example of T5 Prefix Guiding in different NLP tasks [8].

textual output. Its main characteristic is its versatility, as it was designed such that it encompasses a unified framework to handle many tasks. In addition, it uses the same basic architecture across all tasks, with task-specific information provided as prefixes in the input data. For example, to perform machine translation, the prefix would be ”translate English to French:”. These prefixes are conditioning mechanisms used to guide the model's behaviour during fine-tuning (Figure 2.4). This reduces the need for task-specific architectural changes from the Transformer design, and so the pre-trained model could be easily fine-tuned with training for the specific problem that needs to be tackled. Hence, it is an adequate model to use for our benchmarking suite.

2.2.2 NLLB

No Language Left Behind (NLLB-200) is a neural network model developed with the intention of enhancing translation quality and coverage for languages with limited resources [7]. The model can translate a total of 200 languages. Its underlying architecture is based on the transformer design but incorporates some modifications to achieve its objective. For example, the model introduces layer normalization at the start of each sub-layer instead of after the residual connection, leading to more stable training [17].

2.2.3 NLLB-MoE & Switch Transformers

The MoE (Mixture of Experts) variant of the NLLB model extends the Dense NLLB model by introducing a gating mechanism that selects from a range of ”experts” or sub-models specialized in various input segments [4]. Each expert has its own Dense MLP model, and the gating mechanism determines the appropriate experts based on the input. In Figure 2.3, each MoE layer consists of E experts and a gating network responsible for directing tokens. Within the MoE sublayer, E feedforward networks

(FFNs) are employed. The MoE layer is an additional component present in both the encoder and decoder, replacing the feed-forward network sublayer with N feed-forward networks. When compared to dense models, increasing performance by scaling model complexity meant adding more layers, which increases the forward propagation time. Hence, due to this gating design, MoE's generative performance can be increased substantially without adding additional computational cost and latency for inference, but at the expense of being extremely memory-intensive.

A Switch Transformer [9] is a variant of T5 with the properties of a modified version of Mixture of Experts. Its sparsity is achieved by also introducing a ”switch” gating mechanism that selects a subset of tokens based on their relevance to the current input. The difference is that the routing function for Switch Transformer determines only one expert to send each token to, whereas for MoE, it can be sent to more than one.

2.3 Performance Evaluation on MT models

In deep learning, there is a necessity to validate model performance for the quality of outputs. Aside from that, it is also highly important to take into consideration the inference time and the size of the model. While this may be fixed by having larger memory and several GPUs (to enable parallelism across different devices), there is still the need to optimise these models to make them available for many more users. This means generating high-quality output in real-time, without sacrificing such quality with constraining it to smaller resources.

2.3.1 Quality Metrics

There are many existing automatic machine translation metrics, some of which are neural-based. We aim to reproduce the machine translation metrics in order to ensure the reproducibility of pre-trained models to track which could be helpful in future use cases where we could analyse any changes in performance. For the purpose of this research and given time constraints, we limit the scope to using BLEU [18], chrF [19], and METEOR [20]. BLEU score is used extensively in machine translation in the literature, used in T5. Whereas for chrF, the motivation for using this metric along with BLEU is to compare against the translation metric scores found in NLLB, and we attempt to reproduce the results in order to ensure the repeatability and consistency of our experiments. Finally, METEOR score was added to incorporate an additional metric, which through literature suggested that it was more suited to be used for single sentences, unlike BLEU which was designed for corpus measurement.

2.3.1.1 BLEU

BLEU, or Bilingual Evaluation Understudy Score, is a metric used for automatic evaluation of machine translation task [18] to quantify the quality of machine-generated translations against human reference translations, and they are measured from 0 to 1. In literature, they are expressed as percentages implicitly, between 0 to 100. Higher BLEU scores generated from the model signify better quality and are more similar to a dataset of high-quality reference translations, as with all the other metrics.

N-grams are contiguous sequences of ”n” elements, where ”n” can represent individual words or characters depending on the input design. Hence, bigrams for a sentence like ”This is amazing” would be ”This is” and ”is amazing”. BLEU scores involve counting matching n-grams in the candidate translation to n-grams in the reference text, where a unigram would be each token and a bigram comparison would be each word pair. This comparison is made regardless of word order. The calculation for BLEU would be as follows:

$$\text{BLEU} = \text{BP} \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right) \tag{2.1}$$

where p_n are the modified n-gram precisions, w_n are the corresponding weights to the precisions, which are usually uniform in practice ($w_n = \frac{1}{N}$), and BP is the brevity penalty term. N-gram precision in machine translation measures how well the n-grams in the generated output translation align with the n-grams in the reference translation, evaluating alignment at the n-gram level.

While BLEU is shown to be correlated with human ratings of Machine Translation, it is less correlated in NLG task [21]. It is not interpretable across different datasets and is a poor proxy of fluency and adequacy [21]. Fluency refers to the quality of the generated output in the source language, assessing whether the text maintains grammatical correctness and coherence. Meanwhile, adequacy evaluates whether the output text accurately conveys a comparable meaning to the input. It examines whether any information from the original input has been modified, omitted, distorted, or inaccurately added to the translated output. Hence, it is vital to use other metrics, as usage of BLEU alone has recently been found to lead to worse-performing deployment decisions [22] due to these shortcomings.

Also, since BLEU is tokenization-dependent, which might not lead to standardised results, we will report spBLEU as well [7]. spBLEU is the BLEU score applied to text that has been tokenized using the SentencePiece - a tokenizer system that learns subword units from training data of text from any language [23].

2.3.1.2 METEOR

METEOR (Metric for Evaluation of Translation with Explicit ORdering) also applies exact word matching through the calculation of unigram precision and recall to retrieve the harmonic mean [20]. The score's simplified calculation would be

$$score = (1 - P_{en}) * F_{mean} \tag{2.2}$$

, where P_{en} is the penalty and F_{mean} would be the score The difference with METEOR is that it incorporates a synonym matching step to account for variations of the lexicon. It also stems the input - reduces words to their base or root linguistic form (eg. ”consoled” $\rightarrow$ ”console”).

2.3.1.3 chrF

Used in NLLB-200 paper [7], and being the most recently developed metric out of the three, chrF (character n-gram F-score), works by using precision and recall of character n-grams, which are sequences of characters of length ”n.” [19]

Researchers explored neural metrics that utilize neural networks to capture semantic similarity and fluency, addressing some of BLEU's limitations [7]. Metrics, like ChrF and SacreBLEU [24], incorporate character-level matching and handle capitalization and punctuation issues [19]. Hence, its focus on character-level analysis allows the score to be more robust to minor variations in word order and word choice. The formula for the chrF score involves computing precision, recall, and applying the length penalty:

$$chrF = \frac{(1 + \beta^2) * ngrP * ngrR}{\beta^2 * ngrP + ngrR} \tag{2.3}$$

where β is a parameter that balances the weight of precision and recall which can be configured, ngrP is the n-gram precision and ngrR is the n-gram recall.

However, chrF has some limitations. It may not fully capture higher-level linguistic aspects, such as syntax and semantics, that other metrics like BLEU or METEOR attempt to address. Additionally, chrF's sensitivity to character-level n-grams might indicate that it is sensitive to misspellings errors. We also complement chrF with reporting chrF++, which adds n-gram words to the score [25]. chrF++ is thus referred to as having word_order of 2. The altered score was shown to correlate more strongly using Pearson correlation with direct human assessments.

2.3.2 Model Inference Time

To evaluate the speed inference of the model, we can use two metrics: latency and throughput.

Latency refers to the time it takes for a single inference i.e) translation to be generated by a deep learning model after submitting the source input text. In other words, it is the delay between sending an input to the model and receiving the corresponding output. Latency is usually measured in units of time, such as microseconds or seconds. Minimizing latency is crucial since low latency means that the model can respond quickly to input, providing timely and interactive results, such as in the scenario where the chatbot provides multilingual support which is necessary to maintain a smooth conversational experience with the client. It can be affected by various factors, including the model architecture's complexity, the size of the input data, hardware resources, software optimizations, and the deployment environment. Techniques like model quantization, model pruning, and efficient hardware accelerators (such as GPUs and TPUs) are often employed to reduce latency in deep learning inference [26]. We can express latency L simply as $L = T_{end} - T_{start}$, where T_{end} is the time when the model has successfully produced the full output T_{start} is when the input is sent to the model.

Throughput, on the other hand, refers to the number of inferences (translations) a deep learning model can perform within a given time frame. It is also critical to consider it as we might find the need to deploy large language models to process a large number of inferences simultaneously within a short period of time. Optimizing throughput involves a combination of hardware resources, parallelism, and software optimizations. Techniques like batch processing, multi-threading, and distributed computing can help increase throughput by enabling the model to process multiple inferences simultaneously. We can simplify the expression of throughput Tp as $Tp = \frac{N}{\Delta t}$, where N is the number of inferences processed in the time interval Δt.

There is often a trade-off between latency and throughput, as they are inversely proportional. One common way to manage this trade-off is by adjusting the batch size (B) used for inference. A larger batch size can increase throughput by processing multiple inferences in parallel, but it might also increase latency for individual inferences due to the time required to accumulate enough samples in the batch.

2.4 Disk offloading

Memory demands can arise in various contexts and domains for tasks and applications that are memory intensive. As a result, there is ongoing research to address these without purchasing more computing power and memory. In data-centre applications, one such system proposed is Transparent Memory Offloading (TMO) which improves server memory utilisation by transparently offloading it to remote memory pools [27] Similarly, the field of deep learning also suffers such constraints. As such, there are several publications investigating how to enable efficient and effective deployment of large models. Running inference and training on large language models that cannot fit entirely into the GPU requires the usage of optimisation to reduce the memory footprint. Examples include knowledge distillation[28], training a smaller model to replicate the outputs of a larger one, mixed-precision training[29], combining lower and higher precision numerical representations which can be used to enable tensor cores, or disk offloading, the process of redistributing some of the memory requirements in heterogeneous devices from GPU to other resources such as SSDs and CPU. This section discusses the related work done to serve large language models. Since some of the key related work has been mentioned in the Informatics Project Proposal (IPP) report, the section on Deepspeed has been partly taken from it.

2.4.1 Related work: Deepspeed

Deepspeed is an open-source deep learning optimization library developed by Microsoft [3]. It provides a set of tools for training large deep learning models efficiently as well as inference. Features such as gradient checkpointing, memory optimization, and pipeline parallelism (interleaved pipeline) enable the training and inference of models with billions of parameters on a single or multiple machines.

In terms of offloading memory, Deepspeed developed ZeRO infinity to offload memory from GPU computations only, these utilise GPU, CPU, and NonVolatile Memory Express (NVMe) memory to allow for huge upscale on model training with limited resources [30]. For inference, ZeRO-Inference, which is adapted from ZeRO-infinity, leverages memory components like GPU memory, DRAM, and NVMe to meet the substantial memory demands of accommodating large-scale models. This technique stems from observing that limited GPU resources could have different types of memory that could add up to terabytes in scale, which shows that there is a possibility to accommodate large language models that scale up to billions of parameters. According to the paper, the framework is suitable for inference tasks that are ”throughput-oriented and allow large batch sizes” instead of low latency [31].

Due to the huge scale of memory requirements used by MoE models, there is also research done to advance the acceleration of MoE models. The inference is bound by memory bandwidth, the rate at which data is moved between pools of memory, and reading model weights of large MoE models could result in memory bottleneck. Deepspeed-MoE hence was developed as a system optimization strategy on existing multi-GPU inference systems by combining the memory bandwidth of multiple distributed GPUs to enhance memory bandwidth utilization [32].

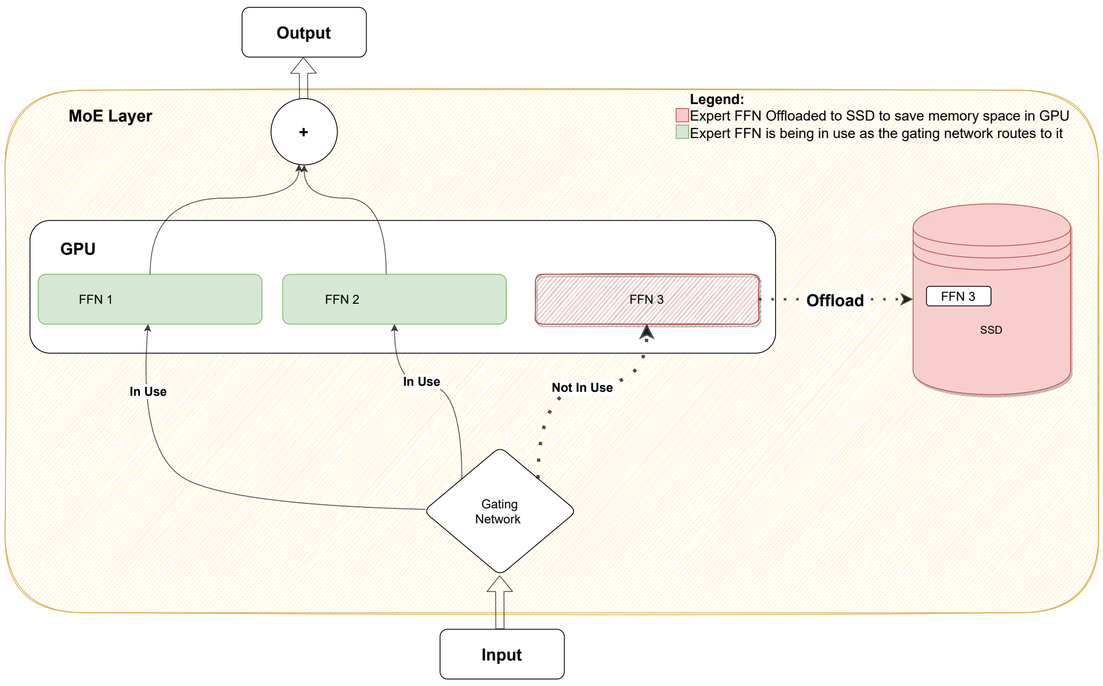

Output
MoE Layer
+
Legend:
🟥 Expert FFN Offloaded to SSD to save memory space in GPU
🟩 Expert FFN is being in use as the gating network routes to it
GPU
FFN 1
FFN 2
FFN 3
Offload
FFN 3
SSD
In Use
In Use
Not In Use
Gating Network
Input

Figure 2.5: The Methodology of Disk Offloading on Mixture of Experts. Experts, denoted as FFN here, that reside in the GPU are offloaded to other resources like the SSD to save memory. The gating network for instance, scores high probabilities in choosing the experts of FFN1 and FFN2 to calculate the inputs of this task.

2.4.2 Archer

Archer, an inference engine, is an internal University project currently under development that serves to optimise the efficiency and throughput of machine learning models, in order to allow large language models to be deployed in various settings, including resource-constrained devices. It aims to be an extendable engine, that can be used as a plugin to other open-source inference servers like Triton from Nvidia. The library also includes disk offloading which tests and works on by utilising SSD to save weights.

The premise of Archer's improvements comes from the observation that a small portion of the experts in a large MoE model is used. Unlike dense models where layers are all given equal importance, the sparsity of MoE models can be leveraged to improve the disk-offloading technique compared to that of Deepspeed. By pre-emptively prefetching the experts required in GPU in time for forward propagation to the correct experts, this caching mechanism would increase latency and decrease throughput. To guess which expert it needs in time, as otherwise, this would result in slower performance, it tackles to address this through exploiting the sparse activation patterns from the expert. The methodology used, called Expert Activation Predictor, utilises the skewed probability distribution observed where the activation of an expert will indicate with higher probability which next expert will be triggered. These likelihoods serve to guide the inference engine in determining which experts to prefetch ahead of time.

We utilise the Archer engine to test our benchmark against the current iterations of the library to investigate briefly the efficiency of the models run on it.

Chapter 3

Benchmark Design and Implementation

The developed benchmarking suite represents a comprehensive framework designed to evaluate the performance of large language models on the metrics reported in the Background chapter 2. It is designed to provide a qualitative and quantitative understanding of model behaviour.

There are three main components of the framework to support the main class BenchmarkingSuite: Archer, ConfigurationSetup, and Analytics. We will first start describing the major class that calls upon the external modules Pytorch, Hugging Face, and Archer, to help with the inference of the model to generate the results described in Analytics. The ConfigurationSetup is a helper module written to support the execution of BenchmarkingSuite. It provides the utility to prepare, preprocess, and instantiate the experiments to be run. The diagram in 3.1 provides the encapsulated software architecture of the framework.

3.1 BenchmarkingSuite

This is the main class of the suite. The class first starts by checking whether we run a single or multiple experiments, and checks the other settings. Following that, depending on the model, we import its appropriate Tokenizer and Model class from HuggingFace and also download the model using it. The inputs and datasets will be downloaded and transformed by the appropriate classes Datasets and Preprocessing while also corresponding to the hyperparameters provided, such as dataset size to be used for machine translation or by filtering input sequence length. Afterwards, BenchmarkingSuite is responsible for running the model. The input is brought into the tokenizer, which prepares and encodes the text strings into vectors of numerical

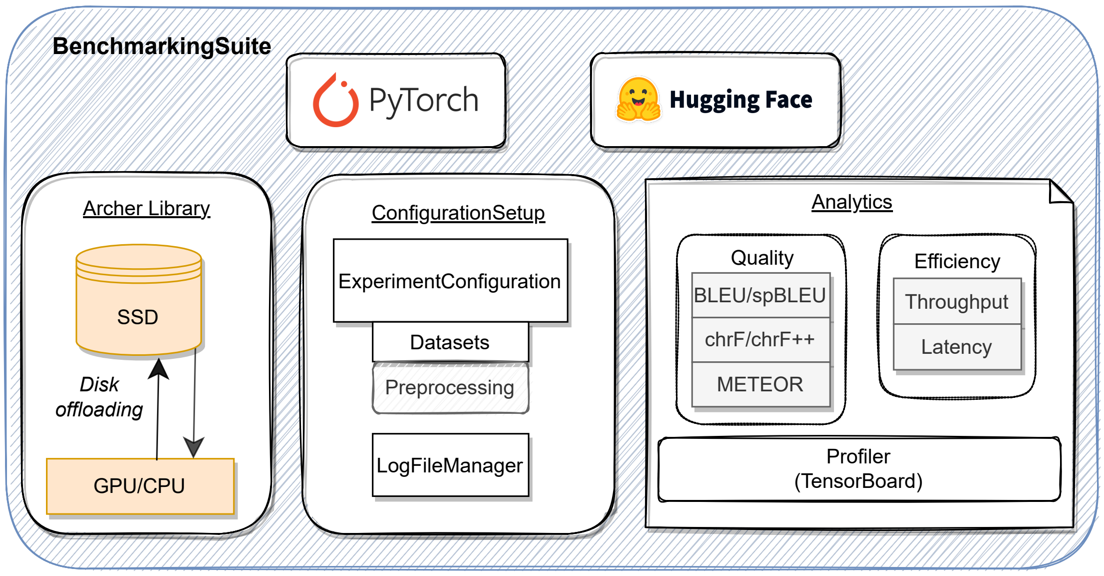


Figure 3.1: Benchmark Software Architecture. A high-level overview of the components of the benchmarking module, which makes it capable of running experiments in the main BenchmarkingSuite class to report a variety of metrics.

representations signifying their word ID in the vocabulary on which the tokenizer and the model were trained on. This transformed representation allows the model to process that data. Once that is done, and once we filter the input by the maximum sequence length, we run the generation process. We use DataLoader from HuggingFace to split the dataset samples in batches of a specified batch size in order to run the inference The library also helps in managing the efficient copying of data from CPU to GPU, for the model weights to run on. The output predictions are then decoded by the tokenizer from numerical vectors to strings of text that can be read by the user. We save the results and continue generating them across the other batches. Once that is done, we compute the metric scores through the evaluate library from HuggingFace, which is an interface that brings together different libraries to compute the scores (See Quality Metrics in Analytics section). We save the results in a CSV file to be later analysed. For multiple experiments, we rerun the whole procedure of
tokenizer.encode() → model.generate() → tokenizer.decode() → compute_metrics()
until all permutations of hyperparameters are gone through.

If the process fails, we log the failed experiment and continue on to the next one. This saves development time instead of rerunning the whole experiment from the beginning when the problem is specific to one set of configurations. Also, through the logging system, we can identify the cause of the error by examining it, instead of it being lost. This allowed for more experiments to be run as a result.

3.2 Configuration Setup

The Configuration setup is a written module that represents four classes to help execute and run the main class to perform its generation.

3.2.1 ExperimentConfiguration

This class is responsible for setting up the choices of hyperparameters needed to run the experiments. It also gives the ability for the main class to add the hyperparameters onto the command line. The arguments passed onto the file make it easy to run a model on a dataset and the hyperparameters of choice. The different hyperparameters will be discussed in the Experiments section.

Currently, there is no support for other language codes other than *”en-fr”* which is what we will aim to benchmark in the scope of our projects. However, it should be easy to support new language codes and integrate them so long as the dataset and model support such translation. The most problematic part comes from using new datasets from HuggingFace as each dataset has its own formatting. As a solution, the Dataset Class was designed in order to tackle this issue of formatting from new datasets into one standardised way which BenchmarkingSuite can use to run machine translation.

3.3 Dataset Class

Each dataset sometimes has varying formatting to how it organises its training and test data. The current prototype is built with that in mind. Currently, it supports only the two datasets WMT14 [6] and FLORES-200 [7], but this interface is built so adding datasets means ease of extending the implementation to include the new dataset and custom format it in the right way. The formatting will be tested to ensure it adheres to the formatting needed for the inputs to benchmarkingSuite. The data ingestion step ensures uniformity in input data The benchmarking framework is designed in a way to enable ease of integration of new metrics and datasets into reporting and the ability to perform multiple experiments against the models that need to be evaluated. In-depth detail of the dataset, hyperparameters and the datasets used to report the metrics on will be explained in Section 4.1.

Metric	Overview	Measurement
Latency	The time it takes to run a single inference in seconds or milliseconds. For each encoder and decoder, it is normalised per token to compute the latency per token for more standardised comparisons.	$L = T_{end} - T_{start}$
Throughput	Number of translations that can be done within a timeframe	$Tp = \frac{N}{\Delta t}$

Table 3.1: Summary of the Efficiency Analytics

3.3.1 Archer

We call the Archer inference engine's API to deploy sparse expert models that cannot fit into the GPU's memory. The methodology of it is abstracted for the user, as it can be used immediately by calling ArcherEngine and then initialising the inference engine with the right HuggingFace Tokenizer and Model Class to help it load its model architecture and saved weights. These will be used to generate the guided caching mechanism to help it optimise the deployment of the model.

It is worth noting that most of the benchmarks and experiments are done without the usage of Archer, This is for a variety of reasons, mainly because Archer is still under constant changes at the time of researching and writing this paper. Although large models like NLLB-MoE can still be run quite effectively, there is significant uncertainty in the quality metrics reported. However, we can still run experiments using a large language model to report on efficiency metrics, such as SwitchTransformers [9]. The objective of the support of running Archer is to serve as a proof of concept that can lead to the development of a system to benchmark uses of inference and optimisation libraries.

3.4 Analytics

The metrics in the benchmarking suite that are used to measure the performance of the models are described in this chapter. Before introducing them, this table provides an overview of what these metrics are and their measurement formulas:

Metric	Description	Measurement
BLEU	The most widely used MT metric for the ease of its simplicity. Count matching n-grams in the candidate translation to n-grams in the reference text.	$\text{BLEU} = \text{BP} \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right)$
spBLEU	BLEU score applied to text that has been tokenized using the Sentence-Piece, aimed to mitigate BLEU's tokenization-dependence	$\text{BLEU} = \text{BP} \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right)$
chrF	Using precision and recall of character n-grams, which are sequences of characters of length "n".	$chrF = \frac{(1+\beta^2)*ngrP*ngrR}{\beta^2*ngrP+ngrR}$
chrF++	chrF score with the inclusion of n-gram words	$chrF{++} = \frac{(1+\beta^2)*ngrP*ngrR}{\beta^2*ngrP+ngrR}$
METEOR	Exact word matching through the calculation of unigram precision and recall to retrieve the harmonic mean	$METEOR = (1-P_{en}) * F_{mean}$

Table 3.2: Summary of the Quality Analytics

3.4.1 Quality Metrics

We report on quality metrics using SacreBLEU [24], which is a standardised library to automatically compute evaluation metrics for machine translation provided reference data and the data to evaluate against. Its development was necessary to lead to consistent, and error-prone metrics. It supports a variety of metrics, like BLEU and chrF. For METEOR, we use NLTK [33] which computes the measurement, a popular natural language processing library used for a variety of tasks including text analysis. Both libraries are supported in HuggingFace [34], as it provides an interface easily to access these different libraries. This is integrated with HuggingFace [34].

3.4.2 Efficiency Metrics

The metrics reported are latency and throughput. During the first iteration of the suite, the latency was recorded as the runtime execution of the entire model pipeline. However, further iterations of the framework realised that this may not lead to normalised and comparative benchmarks. Instead, we thought it would interesting to decompose this into recording latency and throughput for the encoder block and decoder block separately. This was achieved by writing a forward_decorator used to wrap the original forward function in HuggingFace transformer function to modify and introduce extra functionality to it: decompose the latencies into encoder and decoder. The time taken to run the forward propagation of one block is then normalised to get latency per token instead, since every sentence is of variable length and so we want to make sure the latency metric is reflected and can be compared against in a more standardised way. The throughput is then computed from the latency and total number of tokens processed to generate this.

3.4.3 Profiling

The framework supports the utility of viewing and profiling memory and GPU resource utilisation of the model, through the usage of Pytorch Profiler [35] and Tensorboard [36]. These are tools that aid developers in tracking experiment metrics during training and inference, including trace viewers. This can help investigate the CPU and GPU usages, allowing developers to diagnose any bottlenecks such as I/O wait time to optimise the runtime of model execution. For the scope of the project, we do not extensively analyse the results returned from the trace viewer. However, for a demonstrative example and to support our analysis in the Experiment Section 4, we will show the GPU utilisation for the different parameters configured for the runtime Archer on SwitchTransformers.

3.5 Implementation

The implementation was written in Pytorch using HuggingFace API to execute our models. We undergo rigorous testing by examining the outputs of the model and the metrics returned. Originally, we aimed to examine these analytics returned by the models against the results in the T5 paper. However, despite not replicating the results of the paper itself, we cross-referenced against HuggingFace Github repository's results [37] and reported identical BLEU scores for the pre-trained T5 model. We suspect

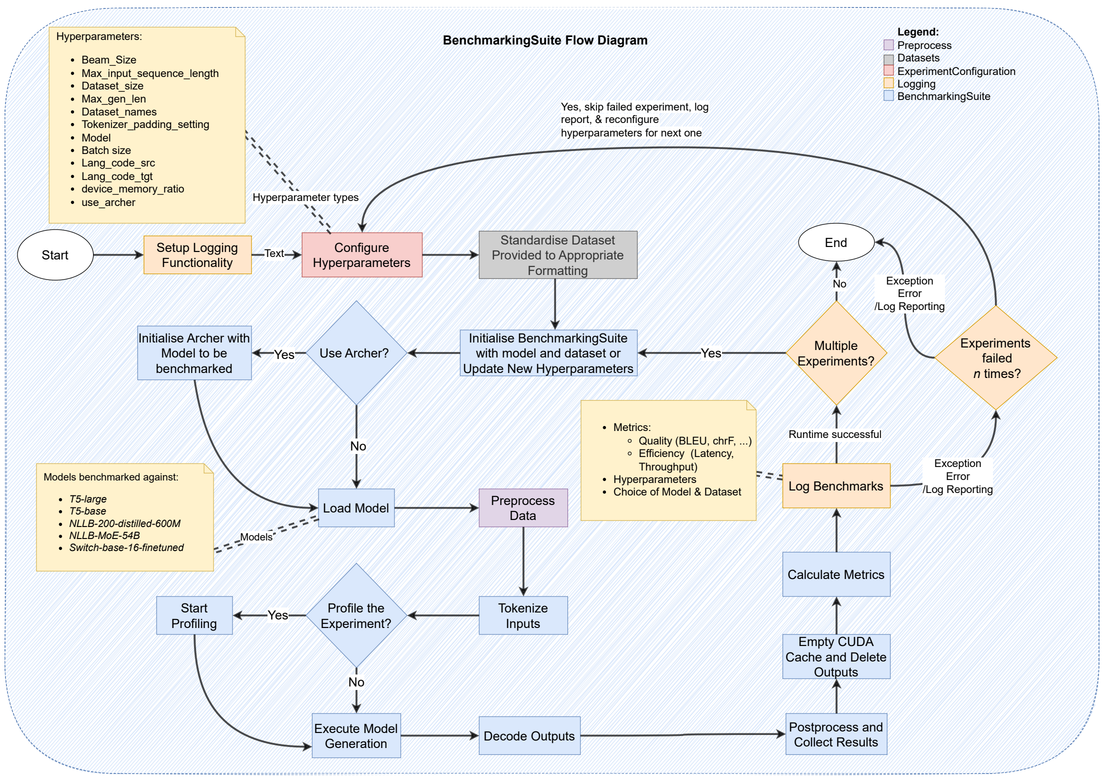

BenchmarkingSuite Flow Diagram
Legend:
Preprocess
Datasets
ExperimentConfiguration
Logging
BenchmarkingSuite
Hyperparameters:
Beam_Size
Max_input_sequence_length
Dataset_size
Max_gen_len
Dataset_names
Tokenizer_padding_setting
Model
Batch size
Lang_code_src
Lang_code_tgt
device_memory_ratio
use_archer
Hyperparameter types
Start
Setup Logging Functionality
Text
Configure Hyperparameters
Standardise Dataset Provided to Appropriate Formatting
Yes, skip failed experiment, log report, & reconfigure hyperparameters for next one
End
No
Exception Error /Log Reporting
Initialise Archer with Model to be benchmarked
Yes
Use Archer?
Initialise BenchmarkingSuite with model and dataset or Update New Hyperparameters
Yes
Multiple Experiments?
Experiments failed n times?
No
Metrics:
Quality (BLEU, chrF, ...)
Efficiency (Latency, Throughput)
Hyperparameters
Choice of Model & Dataset
Runtime successful
Models benchmarked against:
T5-large
T5-base
NLLB-200-distilled-600M
NLLB-MoE-54B
Switch-base-16-finetuned
Models
Load Model
Preprocess Data
Log Benchmarks
Exception Error /Log Reporting
Calculate Metrics
Start Profiling
Yes
Profile the Experiment?
Tokenize Inputs
Empty CUDA Cache and Delete Outputs
No
Execute Model Generation
Decode Outputs
Postprocess and Collect Results

Figure 3.2: The Workflow of the Benchmarking System

that the BLEU scores reported on their baseline might have had slightly different hyperparameter configurations for its decoder output. As mentioned, the benchmarking suite consisted of several iterations and refinements to optimise its performance, in order to run as memory efficient as possible. This included optimisation of code such as deleting variables when not in use or emptying the cache after the runtime of the model. The figure 3.2 contains the workflow diagram of the benchmarking steps to complete the successful execution of computing analytics.

Chapter 4

Experiments

In this section, we will showcase some of the experiments done to benchmark and analyse the results of the models. First, we will briefly describe the datasets and the hyperparameters which were used for our experiments. Then, we explain the methodology for running these experiments using the benchmarking suite software and also state what combinations of parameters are applied. Lastly, we will report and analyse our findings for dense and sparse models.

4.1 Datasets & Models

We will examine the benchmarks reported for one language-pair for the scope of our project: French-English translation pairs sourced from the two datasets, WMT14 and FLORES-200.

The annual event for Conference on Machine Translation, which was previously called Workshop on Machine Translation before 2016, aims to advance the evaluation of machine translation [6]. The dataset WMT14 created in the year 2014 was emerged since the inception of the conference as part of a shared effort from researchers and participants to tackle specific NLP tasks. Examples for that year included five objectives including a news translation task, a quality estimation task, a metrics task, and a medical text translation task. For our purposes, we will use the news translation dataset with the ”French-English” language pair. To ensure consistency of results, it is worth noting that the dataset has been updated by the same members twice [38] after finding some errors. For these experiments, we are using the latest updated version provided by SacreBLEU library.

FLORES-200 is a collection of datasets that contains 200 languages in total, in-

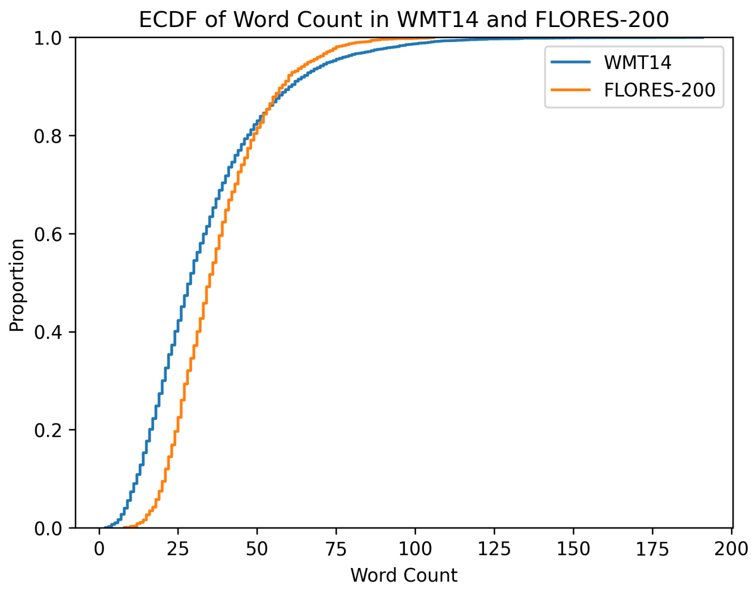


Figure 4.1: Empirical Cumulative Distribution Function (ECDF) plot of all inputs, for French and English text to visualise the differences in the text length between WMT14 and FLORES-200. The two languages are combined together in this plot, but, while not presented here, there are similar trend lines for the ECDF of English only and the ECDF of French only.

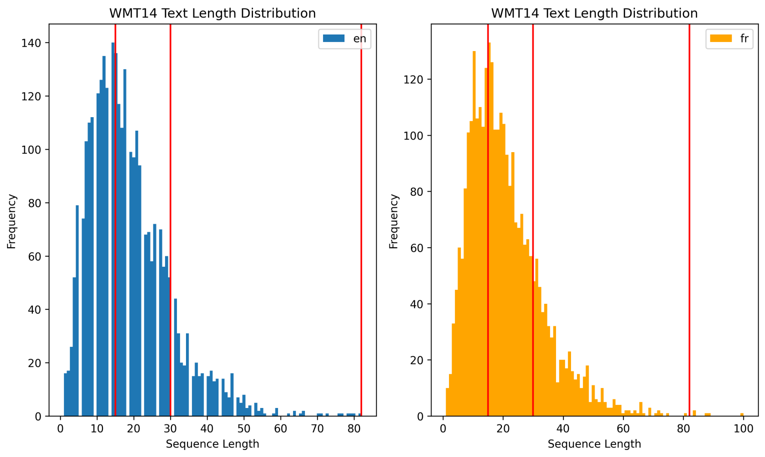


Figure 4.2: WMT14 English Sample Text Length Distribution Count for English and French. Red vertical lines denote the three configurations of the hyperparameter max_input_sequence_length: 15 word sentences, 30 sentences, -1, or in other words no maximum limit is applied.

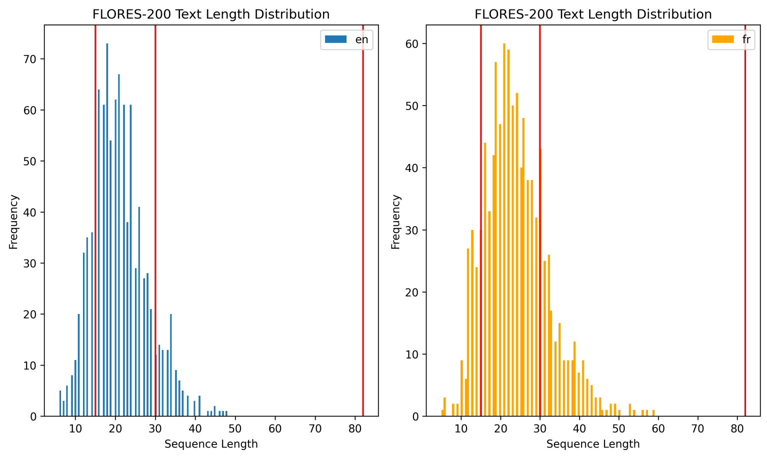


Figure 4.3: FLORES-200 Sample Text Length Distribution for English and French. Red vertical lines denote the three configurations of the hyperparameter max_input_sequence_length: 15 word sentences, 30 sentences, -1, or in other words no maximum limit is applied.

cluding low-resource languages [7], or in other words, languages with limited available translations. FLORES-200 is an extension of the previous FLORES-101 dataset, with double the range of languages [23]. The aim is to expand and standardise even further the availability of more data to use for benchmarking for the task of machine translation.

The choice of these specific datasets was motivated by the interest in developing a benchmarking tool that could be later used for the extremely large language models that utilise sparsity: SwitchTransformers and NLLB-MoE (which contains 54 billion parameters). However, as a start, we will use their original dense versions to build and benchmark our suite against, which would be T5 and the (knowledge) distilled version of NLLB respectively, with different parameter sizes [7]. We will use the respective datasets trained and validated. Only the validation dataset is required, as it is necessary to benchmark the pre-trained models on the specific data used to evaluate against and to ensure consistency by cross-referencing the results.

We investigated the datasets briefly to learn more about the data to help with the design of the experiments. From the plots, we concluded that it may be interesting to see how the benchmark outputs of smaller sentences compares to larger ones in terms of quality and runtime during inference. This led us to create the hyperparameter max_input_sequence_len, which is the maximum input sequence length of the samples to be used for running inference. The results are illustrated using an Empirical Cumulative Distribution Function to demonstrate how data values are spread out in a dataset. The results of the ECDF provided insights into the hyperparameters of max_input_sequence_len to be considered.

The proportions ratio of WMT14 and FLORES-200 provided insights into the values of the hyperparameter max_input_sequence_len to consider for the experiments, respectively 15, 30, -1 (which is a notation used to signify that no maximum limit is used to filter the dataset) - See figure 4.1. Further explanations of this parameter along with others will be provided in the following section.

4.2 Hyperparameters

The experiments conducted are a combination of the parameter values that were selected for benchmarking. The section will provide an overview of these parameters:

- *Beam Size:* Beam size refers to a parameter used to inform the decoding strategy, particularly in sequence-to-sequence models like neural machine translation (NMT), for generating the most likely output sequence from a trained model [39]. When a sequence-to-sequence model generates an output sequence, it does so step by step, predicting one token at a time. At each step, the model considers a set of possible next tokens based on the previously generated tokens. The beam size determines the number of candidates the model keeps track of during this generation process. When set to 1, the decoder performs a greedy search. When the beam size is greater than 1, it performs the beam search. The beam size parameter is one of the following in the list [1,2,4].

- *Max_input_sequence_len:* The input sequence length of the samples in the dataset. It is used to filter out the sentences to include only those that are of the word count of *max* size parameter or below. The motivation was to examine the latency and throughput of sentences of varying sizes. Tables 4.1 and 4.2 demonstrate the percentages of dataset size after applying the filter. The parameters are set to one of [15, 30, -1], where -1 simply represents there no limit/filter is applied i.e use all of the dataset.

- *Dataset size*: To customise manually how many samples we want to run for inference. The experiments were conducted on dataset sizes of [1, ”all”] to investigate the latency in correlation to throughput. The value 1 signifies a single sentence inference, and ”all” means to utilise all of the dataset for evaluation. The

max_input_sequence_length	Transformed Dataset size	Percentage of data
15,	199	20%
30	898	90%
-1 (all)	997	100%

Table 4.1: Dataset Size After Filter - FLORES200

max_input_sequence_length	Transformed Dataset size	Percentage of data
15,	1370	46%
30	2627	88%
-1 (all)	3000	100%

Table 4.2: Dataset Size After Filter - WMT14

two-sentence English-French pairs from both datasets WMT14 and FLORES-200 were used uniformly for all experiments to ensure consistency. These sentences are presented in the Appendix 6.1 and 6.2.

- *Max_gen_length*: Limits the maximum generation sequence length returned as the model output. It was incorporated initially after the identification of an outlier in the metrics returned during the first iteration of the benchmarking suite for some early experiments. The results indicated that lower generation lengths returned had slightly higher values in the quality metrics. The experiment used a combination of [32,64,128,256,512] together against the other combinations of hyperparameters. However, the results did not signify any correlation and were not an indication of a better-performing model. We later note that for the task of machine translation and from our experiments is that it is best for the model to not have restrictions on the maximum output of translation, given that the same restriction is not applied to the input.

- *Dataset Names*: To evaluate the experiments on the two datasets described earlier: [WMT14, FLORES-200]

- *Tokenizer padding setting:* Whether the tokens encoded by the tokenizer are padded to the maximum length denoted in the model configurations. The addition of this parameter was done after inspiration to the maximum generation length parameter, as we decided it might be worth investigating whether this setting has any effect on the output of the model. Possible values are [”no_pad_limit”, ”pad_to_max_length”].

- *Model*: The model to benchmark on for the run. The pre-trained models chosen are [*”t5-base”, ”t5-large”, ”nllb-200-distilled-600M”*] for dense model experiments, then for the sparse ones, the model *”nllb-moe-54b”* and a fine-tuned model of *”switch-base-16”* are used. The parameter sizes of the models are as follows:
  - *”t5-base”*: 220 million,
  - ”t5-large”: 770 million,
  - *”nllb-200-distilled-600M”*: 600 million ,
  - *”nllb-moe-54b”*: 54 Billion.

  The ”switch-base-16”’s parameter size is inconclusive despite calculating the parameter size of each of the other models.
- *Batch size*: It is the size of batched samples to feed into the forward propagation of the model inference for one cycle. It is usually fixed to 32 but has been configured to other batch sizes depending on the model size. Large batch sizes would provide high throughput but low latency.
- *lang_code_src*: Source language to translate from. Currently set constant to ”en” for the benchmark results.
- *lang_code_tgt*: Target Language to translate to. Currently set constant to ”fr” for the benchmark results.
- *device_memory_ratio*: Used by Archer to limit the max GPU usage for fetching and storing the model weights. It was added to benchmark the memory resource utilisation and efficiency for the different set of ratios: [0.1, 0.3, 0.6, 0.9].
- *use_archer*: An additional parameter was added to toggle whether Archer is used for inference.

4.3 Methodology

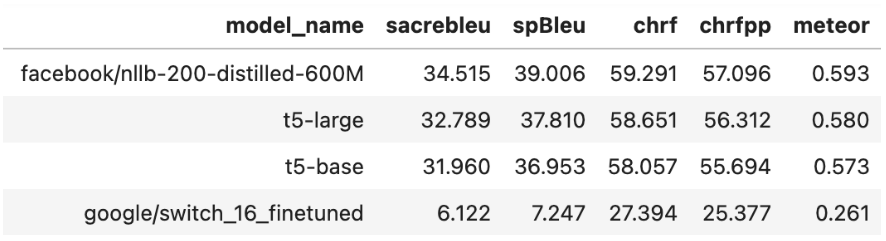
As mentioned earlier, the experiments run are a combination of hyperparameters mentioned in the previous section 4.2 by the benchmarking toolkit, on both the validation datasets of ”WMT14” and ”FLORES-200”. The following results section presents the most notable findings of the analysis on the speed and generative performance summarised of different hyperparameters, first by leading the discussion and focusing on dense models, and finally reporting on the results by the sparse models that are deployed with the help of Archer.

For some models, like NLLB-MoE, there were restrictions on fully benchmarking the output sequences returned from it. While the framework completes the runtime of the model successfully in one of the latest iterations with the aid of Archer, the translations were of extremely poor quality. Also, due to the large scale of NLLB-MoE, it was limited to testing one sentence, with the configuration that was found to be the best for the smaller dense models including the knowledge-distilled NLLB model.

Therefore, to address these limitations as a contingency, additional experiments were conducted on the SwitchTransformer to be able to continue the objective of the research to investigate the performances of the models running on Archer as well for the purpose of evaluating the efficiency of Archer.

Since the pre-trained SwitchTransformer models were designed for Masked Language Modeling, the task to fill in missing words in a sentence, there was an additional component of fine-tuning the model to the task of Machine Translation. It is worth noting that the model is able to run without the necessity of utilising Archer to execute it. However, investigating the smaller, fine-tuned model still holds value as it served as a sufficient step to our initial objective of developing this benchmarking suite. Since Archer can use the sparsity characteristic found in SwitchTransforme, the framework can be used to evaluate against different configurations of Archer, including an additional Archer-specific related configuration: device_memory_ratio. The motivation behind training the model was to investigate Archer’s influence on the efficiency of runtime, and the effects of the parameter device_memory_ratio on the performance. We also briefly investigate the profiling feature to investigate more about memory utilisation.

The ”switch-base-16” model was fine-tuned on the English-Language pair on the OPUS dataset[40], a collection of translated text from the web, after being split into a ratio of 0.80 for training and 0.20 for validation. The details and explanation of the training hyperparameters and the training process will be kept brief as it was not finetuned extensively since that is not the main scope of the dissertation. Appendix 6.3 contains the hyperparameters extracted from the code using HuggingFace API.



model_name	sacrebleu	spBleu	chrf	chrfpp	meteor
facebook/nllb-200-distilled-600M	34.515	39.006	59.291	57.096	0.593
t5-large	32.789	37.810	58.651	56.312	0.580
t5-base	31.960	36.953	58.057	55.694	0.573
google/switch_16_finetuned	6.122	7.247	27.394	25.377	0.261

Figure 4.4: Quality Metrics on the best hyperparameter configurations on WMT14 for the entire dataset.

4.4 Results & Analysis

4.4.1 Dense Models

4.4.1.1 Optimal Hyperparameters

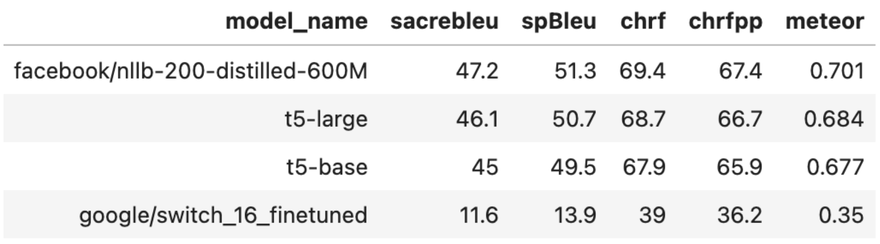
From the different combinations of experiments performed, the Tables in Figure 4.4, 4.5, 4.6, and 4.7 show the quality and efficiency metrics returned by the configuration that returned the best scores for each of the model for each set of values. The set of configurations for the experiments was consistent across all models: beam_size of 4, max_generation_length of 128, and batch_size = 32 regardless of what the other parameters contained.

The configuration was shown to be the most optimal one for all metrics, in terms of quality, the higher scores returned in each efficiency, latency is desired to be as minimal as possible while throughput is desired to be as maximal as possible. In addition, we found that out of the four models benchmarked against, the distilled model of NLLB tended to provide the highest performance in terms of quality for both datasets WMT14 (Table in Figure 4.5) and FLORES-200 (Table in Figure 4.4)

Despite the higher parameter size found in the T5-large model of 770 million, the distilled model showed better performance, offering a glimpse that optimised distilled knowledge models may offer promising results and are an effective tool to reduce the memory requirement of a model while delivering faster and accurate results.

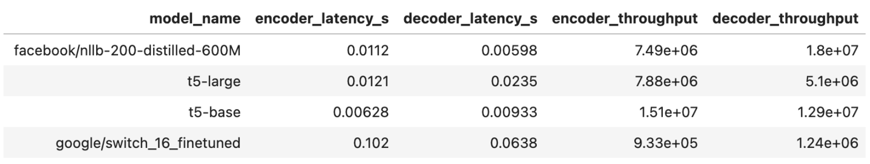
In terms of runtime, t5-base being a smaller model than the three of them, executes slightly faster with the least encoder latency and higher encoder throughput on average

The tables also show the fine-tuned SwitchTransformer model's performances. Since it was trained quickly, it provides low translation results. Further analysis of the model will be done in the later sections.



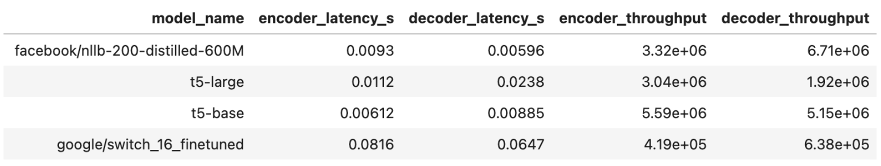
model_name	sacrebleu	spBleu	chrf	chrfpp	meteor
facebook/nllb-200-distilled-600M	47.2	51.3	69.4	67.4	0.701
t5-large	46.1	50.7	68.7	66.7	0.684
t5-base	45	49.5	67.9	65.9	0.677
google/switch_16_finetuned	11.6	13.9	39	36.2	0.35

Figure 4.5: Quality Metrics on the best hyperparameter configurations on FLORES-200 for the entire dataset.



model_name	encoder_latency_s	decoder_latency_s	encoder_throughput	decoder_throughput
facebook/nllb-200-distilled-600M	0.0112	0.00598	7.49e+06	1.8e+07
t5-large	0.0121	0.0235	7.88e+06	5.1e+06
t5-base	0.00628	0.00933	1.51e+07	1.29e+07
google/switch_16_finetuned	0.102	0.0638	9.33e+05	1.24e+06

Figure 4.6: Efficiency Metrics on the best hyperparameter configurations on WMT14 for the entire dataset.



model_name	encoder_latency_s	decoder_latency_s	encoder_throughput	decoder_throughput
facebook/nllb-200-distilled-600M	0.0093	0.00596	3.32e+06	6.71e+06
t5-large	0.0112	0.0238	3.04e+06	1.92e+06
t5-base	0.00612	0.00885	5.59e+06	5.15e+06
google/switch_16_finetuned	0.0816	0.0647	4.19e+05	6.38e+05

Figure 4.7: Efficiency Metrics on the best hyperparameter configurations on FLORES-200 for the entire dataset.

4.4.1.2 Beam Size

The experimentations done for beam sizes were done to investigate if there was any significant improvement or reduction of the values. Figures 4.8 and 4.9 aim to summarise the decoding latency and throughput. On average, there is a slight improvement in terms of quality of outputs, where each of the metrics BLEU, chrF, meteor, and their modified versions provide a boost of about 0.01 - 0.1 in increase. In theory, performing beam search as opposed to greedy search would be an increase in computational cost which would affect efficiency. However, in terms of latency and throughput of the layer responsible for the decoding strategy, the results are not affected for different beam sizes of t5-base and nllb-200-distilled-600M, but there is a significant increase in the latency reported by the bar chart, which is averaged across all other settings. Additional test runs would be necessary to further investigate the specific increase found in one model over the other. Future experiments could also include looking at the length penalty to benchmark against it, which is an additional hyperparameter to penalise words that generate short sentences compared to their references.

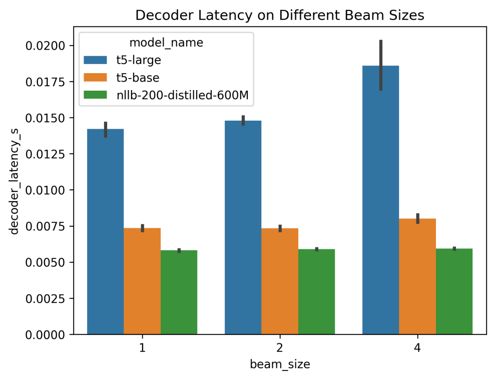


Figure 4.8: The effects of beam size on decoder latency

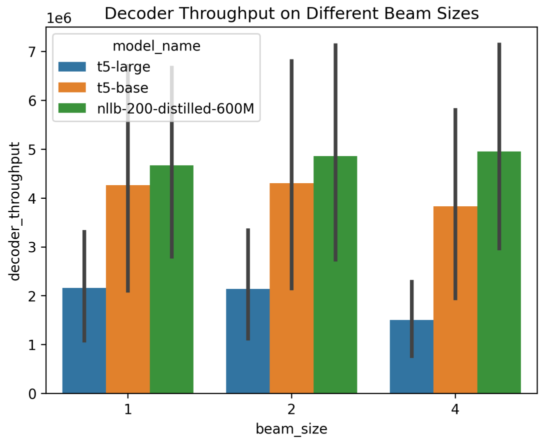
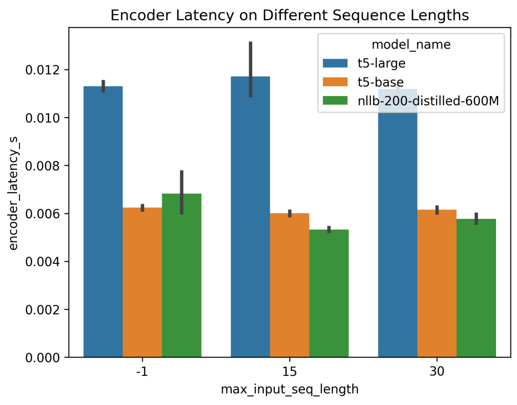


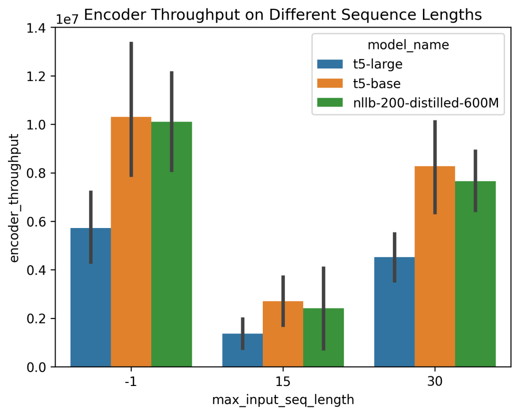
Figure 4.9: The effects of beam size on decoder throughput.

4.4.1.3 Maximum Input Sequence Length

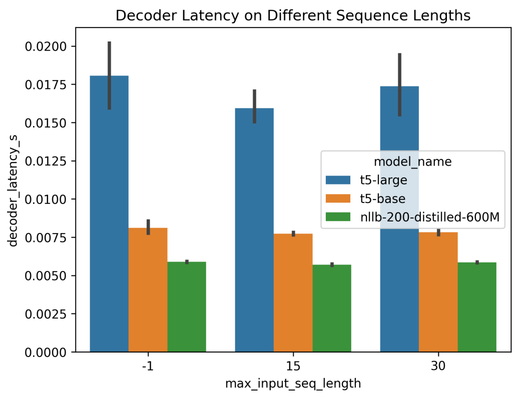
The maximum Input Sequence Length size parameter was incorporated to assess the inference quality of different sequence lengths on the output of the model. The speed analytics are averaged and summarised in bar plots in figure 4.10. For all models, T5-large consumes the most process results per token for encoding and decoding 4.10a, producing the lowest throughputs among the other benchmarked models. Also, the



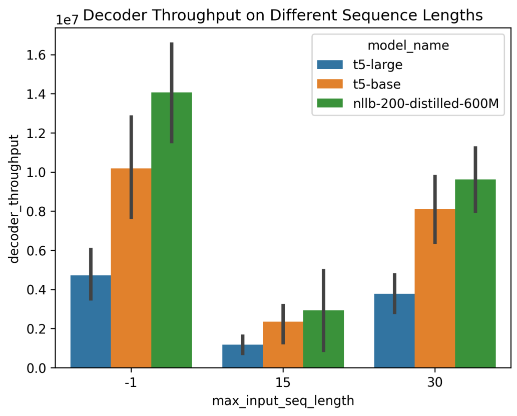
(a) Encoder Latency



(b) Encoder throughput



(c) Decoder Latency



(d) Decoder throughput

Figure 4.10: Efficiency metrics of all experiments done averaged and summarised in the bar chart. The error bars at the top of each bar signify the variability of data to indicate the uncertainty measurement (standard deviation). This is measured against the two datasets in full

encoder throughput is the highest for T5-base (Figure 4.10b) for all sequence lengths. In addition, for seq len set to no limit, -1, the encoder (Figure 4.10a) for T5-base has the lowest latency. However, for decoding, the NLLB model overtakes both latency and throughput. This result builds up on our previous analysis of the efficiency of models, where t5-base on average is as efficient if not slightly more efficient than NLLB-200-distilled-600M, but the decoder for NLLB is more efficient in translating.

Also, on the max input sequence length, we see that T5-large is not affected. However, we do find that for the other two models, it is slightly average on latency, but its throughput is drastically decreased for both the encoder and decoder. It may signify that the short sequence length still leads to activations of multiple weights, as the weights need to be activated regardless in order to compute and decode the model. However, as input sequence length increases, we see higher performances of throughput, which supports our claim.

Finally, the higher cost of T5 models may be due to the additional computational overhead and model weights required to model a variety of tasks, as T5 models are versatile and trained on many tasks. In terms of NLLB, its fast performance may be due to the knowledge distillation technique. It would be interesting to compare two runs of distilled and non-distilled models in future research.

The maximum Input Sequence Length size parameter was incorporated to assess the inference quality of different sequence lengths on the output of the model. The results of speed analytics have been averaged and visualized in the bar plots presented in Figure 4.10.

Across all models, T5-Large exhibits the highest resource consumption per token for both encoding and decoding, resulting in lower throughputs when compared to the other benchmarked models (Figure 4.10a). Conversely, T5-Base consistently demonstrates the highest encoder throughput, as depicted in Figure 4.10b, across all tested sequence lengths. Notably, when sequence length is set to ”no limit” (-1), T5-Base achieves the lowest encoder latency (Figure 4.10a). However, for decoding, the NLLB model surpasses T5-Base in both latency and throughput. This observation aligns with our earlier analysis of model efficiency in finding the optimal hyperparameter configuration, where T5-Base proved to be as efficient as, if not slightly more efficient than, NLLB-200-distilled-600M on average. Nevertheless, the NLLB decoder outperforms in translation efficiency.

Concerning the ”maximum input sequence length,” T5-Large appears to be unaffected, but the other two models exhibit slightly increased latency for shorter sequence lengths, while their throughput experiences a significant decline for both encoder and decoder tasks. This phenomenon suggests that even with shorter sequence lengths, multiple weights are activated due to the inherent computation and decoding requirements of the models. However, as the input sequence length increases, the throughput performance is improved, supporting our claim.

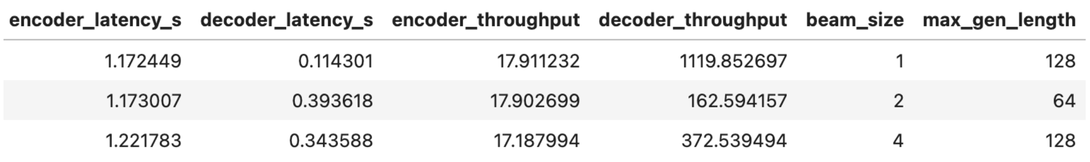
The higher computational cost associated with T5 models can be attributed to the additional computational overhead and model weights required to help exhibit their versatility in handling a wide range of tasks. T5 models are versatile and trained on multiple tasks. In contrast, NLLB’s rapid performance may be attributed to knowledge distillation techniques. Future research may benefit from comparing distilled and non-distilled models in different experimental runs.

4.4.2 Sparse Models

4.4.2.1 MoE - Efficiency Analysis

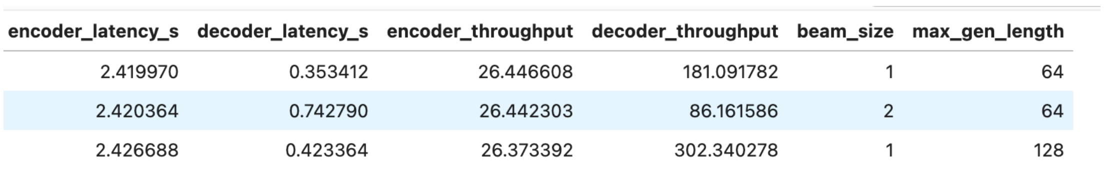
While the translation results for NLLB-MoE-54B, the largest language model of our experiments, are suboptimal to perform any deeper analysis on the quality analytics at the moment, the results for efficiency are detailed in the following tables: Figure 4.11 and Figure 4.12 which sorts by having the best running encoder latencies recorded at the top; and the 4.14 and 4.13 figure tables denote the slow efficiency performance results to be the top result. These results were benchmarked against single sentences from the datasets of WMT14 and FLORES200, and set to device_memory_ratio of 0.3 and 0.6 for some of them. For a single sentence, the MoE performs quite efficiently using Archer given its large parameter size. While the prefetching and loading weights at the initialisation of the engine take approximately 15-20 minutes to load, the inference time is remarkable for a single sentence in contrast to WMT14.

However, in FLORES200, we encountered two instances where the model exhibited a very high encoder latency of 122.380 and 121.689, and scored low on the other efficiency metrics as well (Table 4.13). We suspect that it would be a result of putting a low memory device ratio of 0.3. Further testing needs to be done on this aspect to test the hypothesis and the cause of this as other experiments can run with less than 3 latency per token. the utilization of a low memory device ratio of 0.3 as the potential cause. Due to the time and development constraints, future experiments could validate and further test the hypothesis, as other experiments have successfully achieved latencies of less than 3 per token. Otherwise, no visible trend appeared through the analysis and plotting of these figures, and hence further and various experiments may be required. In fact, the results from the table 4.14 seem to suggest that beam size and increasing generation length sequence lead to a more efficient high decoder throughput, which is contradictory based on our understanding. The other last cue to the randomness with data would be related to the Archer engine's prefetching strategy of experts from an external device, which is probabilistic in nature. Directly analysing Archer's true positive rate on prefetching could shed further light on this.



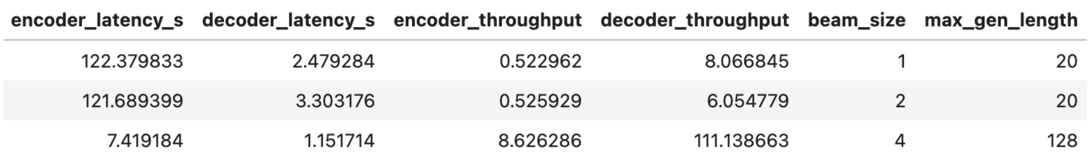
encoder_latency_s	decoder_latency_s	encoder_throughput	decoder_throughput	beam_size	max_gen_length
1.172449	0.114301	17.911232	1119.852697	1	128
1.173007	0.393618	17.902699	162.594157	2	64
1.221783	0.343588	17.187994	372.539494	4	128

Figure 4.11: Efficiency Analytics Sorted by Ascending Least Encoder Latency in WMT



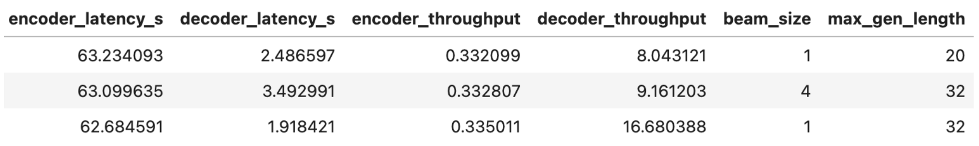
encoder_latency_s	decoder_latency_s	encoder_throughput	decoder_throughput	beam_size	max_gen_length
2.419970	0.353412	26.446608	181.091782	1	64
2.420364	0.742790	26.442303	86.161586	2	64
2.426688	0.423364	26.373392	302.340278	1	128

Figure 4.12: Efficiency Analytics Sorted by Ascending Encoder Latency in FLORES



encoder_latency_s	decoder_latency_s	encoder_throughput	decoder_throughput	beam_size	max_gen_length
122.379833	2.479284	0.522962	8.066845	1	20
121.689399	3.303176	0.525929	6.054779	2	20
7.419184	1.151714	8.626286	111.138663	4	128

Figure 4.13: Efficiency Analytics Sorted by Decreasing Decoder Latency in FLORES



encoder_latency_s	decoder_latency_s	encoder_throughput	decoder_throughput	beam_size	max_gen_length
63.234093	2.486597	0.332099	8.043121	1	20
63.099635	3.492991	0.332807	9.161203	4	32
62.684591	1.918421	0.335011	16.680388	1	32

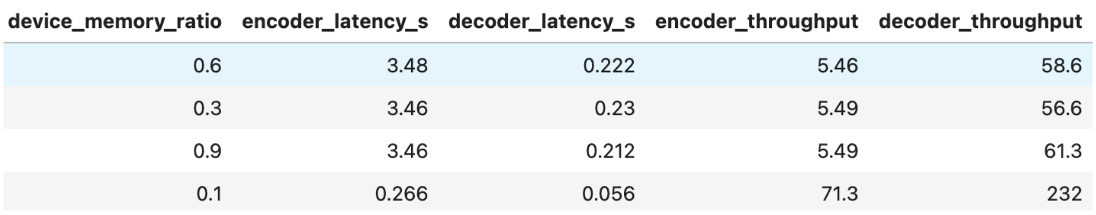
Figure 4.14: Efficiency Analytics Sorted by Decreasing Decoder Latency in WMT

4.4.2.2 SwitchTransformer - Device Memory Ratio

This section benchmarks the results of the fine-tuned SwitchTransformer model, for the purpose of assessing the configuration of device_memory_ratio and its effects on memory utilisation and speed. While the trained neural network can be run without the usage of Archer, having a smaller model than NLLB-MoE to measure performance helps in testing and analysing before potentially scaling up to larger models. This alternative model possesses comparable capabilities to MoE, and so Archer can utilise its prefetching mechanisms.

The device_memory_ratio should not have any effects on the generative performance of its translation, as it is merely responsible for Archer to resourcefully manage the utility of GPU memory and the data movement of the model weights. The testing has been limited to a batch size of 1 for the purpose of consistency with the NLLB-MoE-54B experiments. The results of experimenting with different memory ratio parameters are displayed in the table in Figure 4.15.

Surprisingly, the model was much more efficient when running on a very low memory ratio of 0.1. There is not a clear indication as to why that could be from the results itself, as the expectation was the model would have deteriorated in runtime performance. The other experiments have stable and somewhat identical efficiency, as most of the model weights may be already loaded in GPU to run its inference. In order to have a more in-depth look at the result, As a result, this presents a great opportunity to test the profiling methodology built into the framework.



device_memory_ratio	encoder_latency_s	decoder_latency_s	encoder_throughput	decoder_throughput
0.6	3.48	0.222	5.46	58.6
0.3	3.46	0.23	5.49	56.6
0.9	3.46	0.212	5.49	61.3
0.1	0.266	0.056	71.3	232

Figure 4.15: Finetuned Switch Transformer Performances on Different Device Memory Ratios. The Hyperparameter configurations run for these are batch size of 32, dataset size of 1, maximum generation length of 32, and beam size of 1.

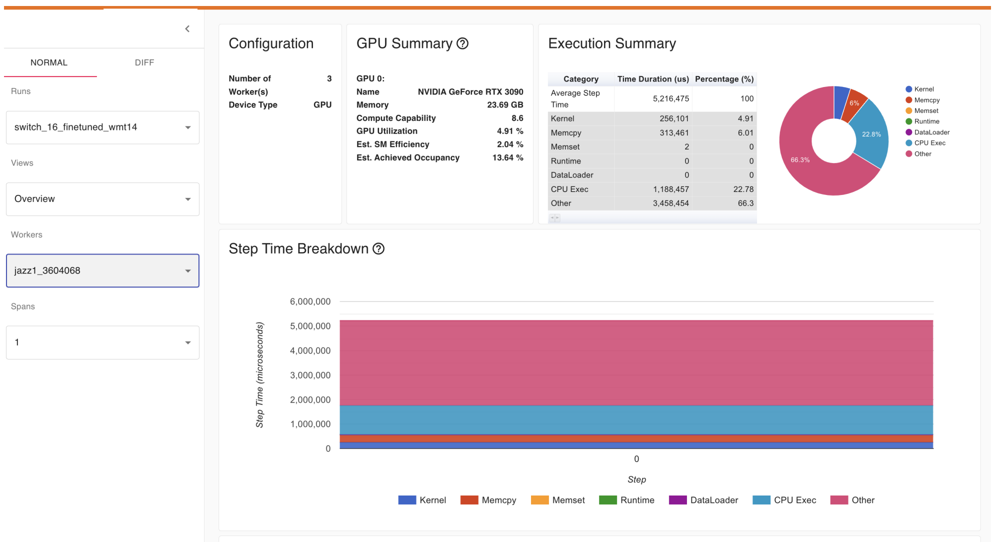
4.4.2.3 Profiling

In this section, the SwitchTransformer models that ran earlier are profiled and saved into JSON files with information regarding in-depth performance metrics about the operations running in the background and memory utilisation. The profiler starts when the inputs are passed to model generation and ends after the outputs generated are decoded, post-processed, and stored later for computing the analytics for quality performance. The profiling results for the experiments done are visualised in Tensorboard and demonstrated in Figures 4.16, 4.17, 4.18 to analyse the phenomena found earlier in section 4.4.2.2.

Observing the runtime can be found in Tensorboard's Overview dashboard, which contains information about the memory utilisation and the description of the runtimes during the execution of the model. The analysis will be shortened to investigate the two device ratio parameters [0,1,0.9] in addition to not utilising Archer. There are two reasons for this: in order to examine the performances with and without Archer and to analyse the question on how a low memory_device_ratio of 0.1 can achieve low latency to denote efficient runtime.

Firstly, the results demonstrate that the model without an archer runs the fastest in terms of its step time breakdown (Figure 4.18). It also encompasses a large majority of its runtime on CPU execution. This may be because the model under-utilises the GPU, as the batch size is small. In comparison, the other experiments showcase the majority of its execution done on ”Other”, labelled as pink, which is time spent that does not include any of the categories labelled as dark blue for ”Kernel” (Kernel execution time on GPU), red for ”Memcpy” (GPU involved memory copy), light blue for ”CPU Exec” (Execution time of CPU), and so on. Also, when comparing the step times of the ratio of 0.1 (Section 4.16) against the ratio of 0.9 (Section 4.17), the total step time in the experiment with 0.1 is higher than 5,000,000 microseconds, which is larger than the step time found the 0.9 ratio. This, however, contradicts the calculations returned for latency in the previous section, since this means that the ratio of 0.9 should be the more efficient one (which is originally expected).

When further inspecting the differences between runtime breakdown percentages, the category called ”Memcpy” takes a larger proportion of the breakdown in the memory device ratio of 0.1. Using the two observations we made, we can infer that the I/O operation may be what skewed our results if perhaps the I/O scheduling call is made pre-emptively to call the model which obscures the attempted logging results. Hence, the next iterations of this benchmarking suite need to accommodate and improve the logging system. This may be done by potentially modifying the program to either start logging when a model is flagged to be fetched into the GPU and/or to make the benchmarking suite more I/O aware.



Configuration

Number of Worker(s)	3
Device Type	GPU

GPU Summary

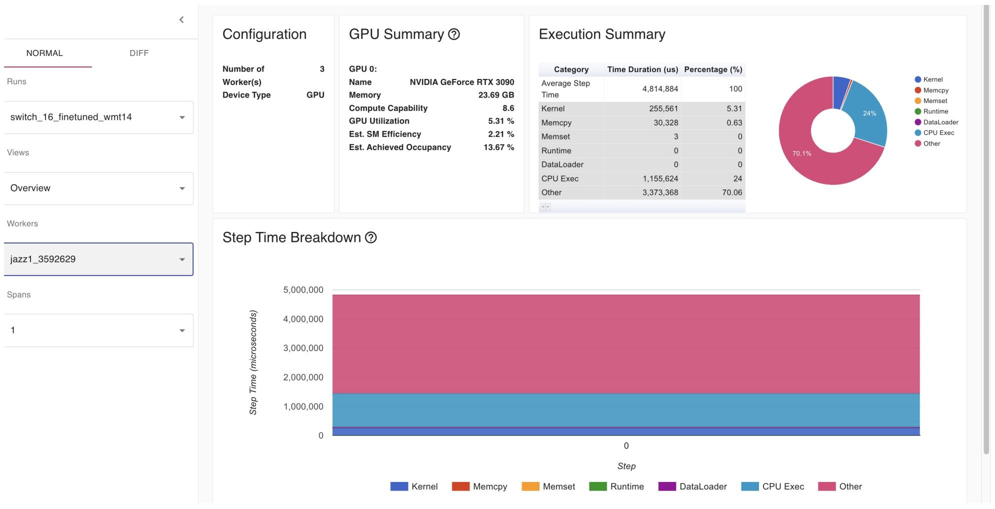
GPU 0:	
Name	NVIDIA GeForce RTX 3090
Memory	23.69 GB
Compute Capability	8.6
GPU Utilization	4.91 %
Est. SM Efficiency	2.04 %
Est. Achieved Occupancy	13.64 %

Execution Summary

Category	Time Duration (us)	Percentage (%)
Average Step Time	5,216,475	100
Kernel	256,101	4.91
Memcpy	313,461	6.01
Memset	2	0
Runtime	0	0
DataLoader	0	0
CPU Exec	1,188,457	22.78
Other	3,458,454	66.3

Step Time Breakdown

Figure 4.16: Tensorboard Overview: Profiling the runtime of fine-tuned switch model on Archer with the memory device ratio parameter set to 0.1



Configuration

Number of Worker(s)	3
Device Type	GPU

GPU Summary

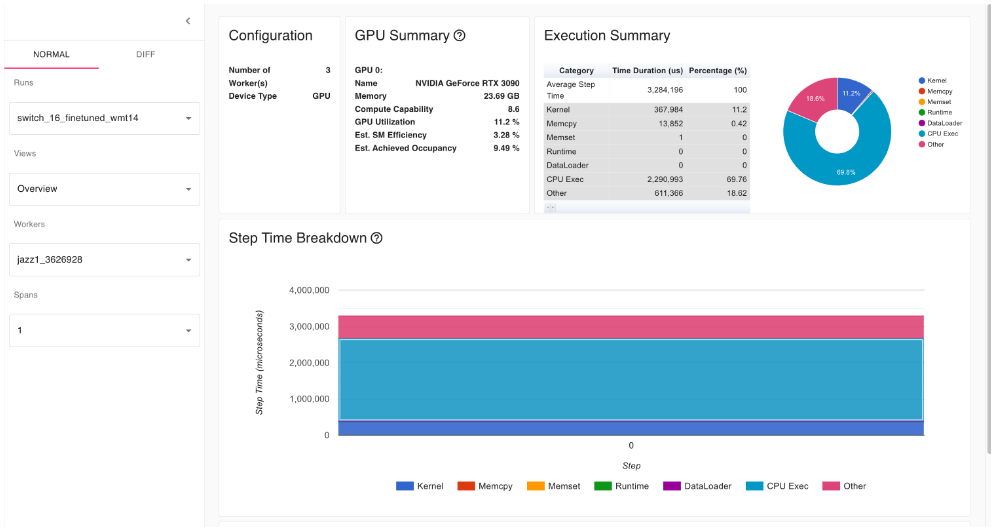
GPU 0:	
Name	NVIDIA GeForce RTX 3090
Memory	23.69 GB
Compute Capability	8.6
GPU Utilization	5.31 %
Est. SM Efficiency	2.21 %
Est. Achieved Occupancy	13.67 %

Execution Summary

Category	Time Duration (us)	Percentage (%)
Average Step Time	4,814,884	100
Kernel	255,561	5.31
Memcpy	30,328	0.63
Memset	3	0
Runtime	0	0
DataLoader	0	0
CPU Exec	1,155,624	24
Other	3,373,368	70.06

Step Time Breakdown

Figure 4.17: Tensorboard Overview: Profiling the runtime of fine-tuned switch model on Archer with the memory device ratio parameter set to 0.9



Configuration

Number of Worker(s)	3
Device Type	GPU

GPU Summary

GPU 0:	
Name	NVIDIA GeForce RTX 3090
Memory	23.69 GB
Compute Capability	8.6
GPU Utilization	11.2 %
Est. SM Efficiency	3.28 %
Est. Achieved Occupancy	9.49 %

Execution Summary

Category	Time Duration (us)	Percentage (%)
Average Step Time	3,284,196	100
Kernel	367,984	11.2
Memcpy	13,852	0.42
Memset	1	0
Runtime	0	0
DataLoader	0	0
CPU Exec	2,290,993	69.76
Other	611,366	18.62

Step Time Breakdown

Figure 4.18: Tensorboard Overview: Profiling the runtime of fine-tuned switch model without Archer

Chapter 5

Conclusions

This dissertation has focused on developing a support tool that could facilitate the automatic evaluation against large language models, starting with machine translation metrics, to assess the quality and efficiency of the model. It also serves as a main foundational block to support the analysis and development of inference engines as well. After continuous refinements, we were able to test the benchmarking suite against dense and sparse models, with and without Archer, to find out more about how different settings of the model - whether it be testing against a corpus of data, a single sentence, or the decoding strategy - led to different results and understanding of it.

The benchmarking framework developed would hopefully serve as an invaluable resource that can be extended after this dissertation to encompass more analytics for tasks beyond machine translation in the natural language processing sub-field, granting us deeper insights into performance and aiding in meticulous evaluation and comprehensive performance analysis. For quality metrics, this would include more recent neural-based ones like COMET [41]. For runtime inference, this includes factors like weight loading times, speed of disk offloading, and more in-depth analysis of engines like Archer and Deepspeed. We could explore the analysis of other avenues to help run models optimally, including benchmarking against the usage of vector databases, such as RETRO [42], potentially providing valuable insights into the model inference behaviour of the system. Another value that this project could have is to assess the models that run on more resource-constrained devices, such as edge GPUs. One final highlight to conclude the research with, is the reported latency of the switch transformer model being extremely efficient despite setting a low maximal GPU ratio. The latency which inconclusively could be a result of the I/O scheduling call while still having an overall slightly larger run-time compared to other experiments. This insightful finding provokes the necessity to conduct further experimentation to narrow down the causality of this outcome.

In summary, our work in benchmarking large language models has paved the way for future advancements, offering a platform for assessing sparse models and embracing the continuously evolving landscape of machine translation technologies.

Bibliography

[1] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[2] OpenAI. Gpt-4 technical report, 2023.

[3] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery Data Mining*, KDD '20, page 3505–3506, New York, NY, USA, 2020. Association for Computing Machinery.

[4] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer, 2017.

[5] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, Ramesh Chukka, Cody Coleman, Sam Davis, Pan Deng, Greg Diamos, Jared Duke, Dave Fick, J. Scott Gardner, Itay Hubara, Sachin Idgunji, Thomas B. Jablin, Jeff Jiao, Tom St. John, Pankaj Kanwar, David Lee, Jeffery Liao, Anton Lokhmotov, Francisco Massa, Peng Meng, Paulius Micikevicius, Colin Osborne, Gennady Pekhimenko, Arun Tejusve Raghunath Rajan, Dilip Sequeira, Ashish Sirasao, Fei Sun, Hanlin Tang, Michael Thomson, Frank Wei,

Ephrem Wu, Lingjie Xu, Koichi Yamada, Bing Yu, George Yuan, Aaron Zhong, Peizhao Zhang, and Yuchen Zhou. Mlperf inference benchmark, 2019.

[6] Ondřej Bojar, Christian Buck, Christian Federmann, Barry Haddow, Philipp Koehn, Johannes Leveling, Christof Monz, Pavel Pecina, Matt Post, Herve Saint-Amand, Radu Soricut, Lucia Specia, and Aleš Tamchyna. Findings of the 2014 Workshop on Statistical Machine Translation. In *Proceedings of the Ninth Workshop on Statistical Machine Translation*, pages 12–58, Baltimore, Maryland, USA, June 2014. Association for Computational Linguistics.

[7] NLLB Team, Marta R. Costa-jussà, James Cross, Onur Çelebi, Maha Elbayad, Kenneth Heafield, Kevin Heffernan, Elahe Kalbassi, Janice Lam, Daniel Licht, Jean Maillard, Anna Sun, Skyler Wang, Guillaume Wenzek, Al Youngblood, Bapi Akula, Loic Barrault, Gabriel Mejia Gonzalez, Prangthip Hansanti, John Hoffman, Semarley Jarrett, Kaushik Ram Sadagopan, Dirk Rowe, Shannon Spruit, Chau Tran, Pierre Andrews, Necip Fazil Ayan, Shruti Bhosale, Sergey Edunov, Angela Fan, Cynthia Gao, Vedanuj Goswami, Francisco Guzmán, Philipp Koehn, Alexandre Mourachko, Christophe Ropers, Safiyyah Saleem, Holger Schwenk, and Jeff Wang. No language left behind: Scaling human-centered machine translation, 2022.

[8] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer, July 2020. arXiv:1910.10683 [cs, stat].

[9] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity, 2022.

[10] Artificial intelligence computing leadership from nvidia, 2021.

[11] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

[12] Alex Sherstinsky. Fundamentals of recurrent neural network (RNN) and long short-term memory (LSTM) network. *Physica D: Nonlinear Phenomena*, 404:132306, mar 2020.

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[14] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization, 2016.

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

[16] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving Language Understanding by Generative Pre-Training.

[17] Ruibin Xiong, Yunchang Yang, Di He, Kai Zheng, Shuxin Zheng, Chen Xing, Huishuai Zhang, Yanyan Lan, Liwei Wang, and Tieyan Liu. On layer normalization in the transformer architecture. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 10524–10533. PMLR, 07 2020.

[18] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a Method for Automatic Evaluation of Machine Translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA, July 2002. Association for Computational Linguistics.

[19] Maja Popović. chrF: character n-gram F-score for automatic MT evaluation. In *Proceedings of the Tenth Workshop on Statistical Machine Translation*, pages 392–395, Lisbon, Portugal, September 2015. Association for Computational Linguistics.

[20] Satanjeev Banerjee and Alon Lavie. METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. In *Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization*, pages 65–72, Ann Arbor, Michigan, June 2005. Association for Computational Linguistics.

[21] Ehud Reiter. A Structured Review of the Validity of BLEU. *Computational Linguistics*, 44(3):393–401, September 2018. Place: Cambridge, MA Publisher: MIT Press.

[22] Tom Kocmi, Christian Federmann, Roman Grundkiewicz, Marcin Junczys-Dowmunt, Hitokazu Matsushita, and Arul Menezes. To Ship or Not to Ship: An Extensive Evaluation of Automatic Metrics for Machine Translation. In *Proceedings of the Sixth Conference on Machine Translation*, pages 478–494, Online, November 2021. Association for Computational Linguistics.

[23] Naman Goyal, Cynthia Gao, Vishrav Chaudhary, Peng-Jen Chen, Guillaume Wenzek, Da Ju, Sanjana Krishnan, Marc'Aurelio Ranzato, Francisco Guzman, and Angela Fan. The flores-101 evaluation benchmark for low-resource and multilingual machine translation, 2021.

[24] Matt Post. A call for clarity in reporting BLEU scores. In *Proceedings of the Third Conference on Machine Translation: Research Papers*, pages 186–191, Belgium, Brussels, October 2018. Association for Computational Linguistics.

[25] Maja Popović. chrF++: words helping character n-grams. In *Proceedings of the Second Conference on Machine Translation*, pages 612–618, Copenhagen, Denmark, 2017. Association for Computational Linguistics.

[26] Gaurav Menghani. Efficient deep learning: A survey on making deep learning models smaller, faster, and better. *ACM Comput. Surv.*, 55(12), mar 2023.

[27] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. Tmo: Transparent memory offloading in datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 609–621, New York, NY, USA, 2022. Association for Computing Machinery.

[28] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network, 2015.

[29] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training, 2018.

[30] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning, 2021.

[31] Reza Yazdani Aminabadi, Samyam Rajbhandari, Minjia Zhang, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Jeff Rasley, Shaden Smith, Olatunji Ruwase, and Yuxiong He. DeepSpeed Inference: Enabling Efficient Inference of Transformer Models at Unprecedented Scale, June 2022. arXiv:2207.00032 [cs].

[32] Samyam Rajbhandari, Conglong Li, Zhewei Yao, Minjia Zhang, Reza Yazdani Aminabadi, Ammar Ahmad Awan, Jeff Rasley, and Yuxiong He. DeepSpeed-MoE: Advancing Mixture-of-Experts Inference and Training to Power Next-Generation AI Scale, July 2022. arXiv:2201.05596 [cs].

[33] Steven Bird, Ewan Klein, and Edward Loper. *Natural language processing with Python: analyzing text with the natural language toolkit.* " O'Reilly Media, Inc.", 2009.

[34] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Huggingface's transformers: State-of-the-art natural language processing, 2020.

[35] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019.

[36] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[37] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Perric Cistac, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-Art Natural Language Processing. pages 38–45. Association for Computational Linguistics, October 2020.

[38] Matt Post. A Call for Clarity in Reporting BLEU Scores. In *Proceedings of the Third Conference on Machine Translation: Research Papers*, pages 186–191, Belgium, Brussels, 2018. Association for Computational Linguistics.

[39] Alex Graves. Sequence transduction with recurrent neural networks, 2012.

[40] Biao Zhang, Philip Williams, Ivan Titov, and Rico Sennrich. Improving massively multilingual neural machine translation and zero-shot translation, 2020.

[41] Ricardo Rei, Craig Stewart, Ana C Farinha, and Alon Lavie. Comet: A neural framework for mt evaluation, 2020.

[42] Sebastian Borgeaud, Arthur Mensch, Jordan Hoffmann, Trevor Cai, Eliza Rutherford, Katie Millican, George van den Driessche, Jean-Baptiste Lespiau, Bogdan Damoc, Aidan Clark, Diego de Las Casas, Aurelia Guy, Jacob Menick, Roman Ring, Tom Hennigan, Saffron Huang, Loren Maggiore, Chris Jones, Albin Cassirer, Andy Brock, Michela Paganini, Geoffrey Irving, Oriol Vinyals, Simon Osindero, Karen Simonyan, Jack W. Rae, Erich Elsen, and Laurent Sifre. Improving language models by retrieving from trillions of tokens, 2022.

Chapter 6

Appendix

6.1 Single sentence dataset from FLORES-200 for Examining Efficiency of Models

English: On Monday, scientists from the Stanford University School of Medicine announced the invention of a new diagnostic tool that can sort cells by type: a tiny printable chip that can be manufactured using standard inkjet printers for possibly about one U.S. cent each.

French: Des scientifiques de l'école de médecine de l'université de Stanford ont annoncé ce lundi la création d'un nouvel outil de diagnostic, qui permettrait de différencier les cellules en fonction de leur type. Il s'agit d'une petit puce imprimable, qui peut être produite au moyen d'une imprimante à jet d'encre standard, pour un coût d'environ un cent de dollar pièce.

6.2 Single sentence dataset from WMT14 for Examining Efficiency of Models

English: A Republican strategy to counter the re-election of Obama French: Une stratégie républicaine pour contrer la réélection d'Obama

6.3 Training hyperparameters of the SwitchTransformer Model Fine-Tuned

```
training_args = Seq2SeqTrainingArguments(
    output_dir="opus_switch_model_16",
    evaluation_strategy="epoch",
    learning_rate=2e-5,
    per_device_train_batch_size=4,
    per_device_eval_batch_size=4,
    gradient_accumulation_steps=4,
    gradient_checkpointing=True,
    weight_decay=0.01,
    save_total_limit=3,
    num_train_epochs=2,
    predict_with_generate=True,
    generation_max_length=512
)
training_args = training_args.set_logging(strategy="steps",
                                steps=100, report_to=["tensorboard"])
```