Deploying Switch Transformer with Triton Inference Server on Memory-Constrained Machines

Michal Pitr

4th Year Project Report
Computer Science and Mathematics
School of Informatics
University of Edinburgh

2023

Abstract

Scaling Transformer-based large language models (LLMs) has resulted in massive improvements in natural language processing tasks. However, this scale has resulted in a corresponding increase in computational complexity. Sparse models, inspired by the Mixture of Experts method, offer a promising way to keep scaling language models without affecting their computational cost. Improving their inference-time speed and making them deployable on GPU-memory-constrained machines would significantly reduce the barrier to entry for the adoption of sparse LLMs. This project aims to deploy Switch Transformer, a sparse LLM, on any memory-constrained machine, understand the effects of reduced GPU memory on throughput, and propose a batching algorithm to reduce the impact.

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.

(*Michal Pitr*)

Acknowledgements

I would like to express my gratitude to Luo Mai for supervising this project. His guidance and insight were invaluable for the successful completion of this thesis. Additionally, I would like to extend my thanks to Ph.D. students Leyang Xu and Yao Fu for their assistance and deep expertise. This project was inspired and made possible by their research projects.

Table of Contents

Chapter 1

Introduction

In recent years, there has been a proliferation of large language models (LLMs) that have demonstrated remarkable performance in various natural language processing (NLP) tasks. Very recently, LLM-powered services like ChatGPT[20] and Github Copilot[14] have made LLMs accessible to millions of users. However, the success of these models has come at the cost of high computational requirements during inference, leading to significant resource demand. As these models become more complex, the need for improvements in inference-time efficiency becomes increasingly critical. One promising method to keep scaling LLMs without significantly affecting their computational complexity is through sparsity - Google AI's Switch Transformer achieves comparable downstream task performance to significantly larger dense models.

Sparse models present many challenges when it comes to deployment and typically lag behind dense models in inference performance. This thesis aims to improve the inference throughput of Switch Transformer's sparse layer when deployed on Nvidia's state-of-the-art inferencing server. Additionally, LLMs have grown so much that they commonly do not fit in a single GPU's memory, which presents further inferencing challenges. Improving inference throughput on GPU-memory-constrained machines would reduce the barrier to entry for on-premise and edge deployments of LLMs.

Specifically, this paper proposes a fetching engine to enable the deployment of Switch Transformer on GPU-memory-constrained machines. Moreover, the thesis proposes a batching algorithm to reduce the impact of loading and unloading parts of the model to the GPU.

Chapter 2

Background

2.1 Mixture of Experts and Switch Transformer

Mixture of Experts (MoE) is a divide-and-conquer machine learning method where the problem space is divided and each subset of the problem space is served by a different machine learning model - an expert. When an input is provided to the system, a gating model decides which expert (or experts) should be used [18]. An example application of this would be a multilingual sentiment analysis language where a gating model first decides which language the input corresponds to and then dispatches it to the expert specializing in that particular language [7]. A recent application of the MoE method is Google's general-purpose language model: the Switch Transformer [13]

In the following subsections, I introduce the Switch Transformer and simultaneously explain some of the details of the MoE method.

2.1.1 Motivation

Many of the recently successful large language models (LLMs), such as BERT, T5, GPT3, or LLaMA are all based on the transformer architecture which was first proposed in Vaswani's famous paper *Attention Is All You Need* [26]. Improvements in downstream task performance have mainly been achieved by using more data and larger models [6]. Although scaling the parameter count of models has been an effective approach, the associated computational costs have also risen drastically [24]. Switch Transformer authors proposed using MoE to achieve the same performance as previous LLMs, but at a reduced computational cost.

2.1.2 Architecture

I will first introduce the transformer block, the fundamental building block of transformer models, and then contrast it with the Switch Transformer's MoE-inspired transformer block.

2.1.2.1 Representing input

I think it is helpful to understand how transformer-based LLMs represent inputs. Similar approaches extend to non-language tasks such as computer vision or even biomedical applications, but for brevity's sake, I will focus on representing written text.

Since text is symbolic in nature, it is required to convert it to a representation that a machine learning model can operate with - real-valued vectors. Firstly, the input text is tokenized, which means that each word is potentially split into substrings called tokens. Next, each token has a 1:1 mapping to a real-valued vector of a fixed dimension - 768-dimensional embeddings are common. These vectors are subject to multiple transformations throughout a neural network. They are commonly called hidden representations, latent representations, and feature vectors.

Hence, given a textual input of N words, the embedding forms an (M, d) matrix, where d is the embedding dimension and $M \geq N$. [26]. As mentioned, it is common for transformer tokenizers to split a single word into sub-word tokens so the resulting matrix is bigger than the length of the original sentence. This is a technicality so to simplify examples, in this paper, I will assume each word corresponds to one token which maps to one feature vector.

2.1.2.2 Transformer Block

At a high level, the transformer block is a sequence of matrix multiplications interleaved with non-linear functions. What makes it a block is the fact that this sequence of operations is often sequentially repeated in a typical transformer model - for instance, BERT is composed of 12 transformer blocks [12] while the more recent GPT3 is composed of 96 transformer blocks [6].

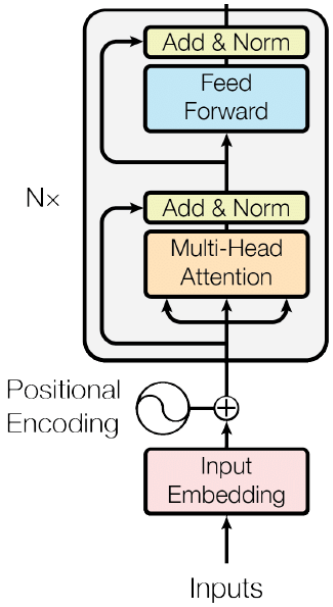

Figure 2.1: Transformer Block diagram adopted from *Attention is All You Need* paper[26]

Describing the operations of a transformer block in detail is beyond the scope of this

project, but I think it is illustrative to at least provide a high-level explanation.

We can think of the block as a function f_i that accepts as its input the matrix representation of a natural language query[1]. This input matrix is of dimensions (N, d), where N is the input length and d is the embedding dimension. f_i's output has the same dimension as its input.

What each f_i does is it takes some *latent* representation of the original input text and processes it to create a more refined representation. This process has two main parts: multi-headed self-attention and a feed-forward neural network.

The self-attention mechanism contextualizes every token in the input text by updating each token as a weighted average of all of the other tokens in the text. Concretely, given the sentences "I want to eat an apple" and "I work at Apple" (let's assume cases are not distinguished as that tends to be the case with language modeling), the meaning of the word "apple" is heavily dependent on context. The self-attention mechanism will imbue the representation of the token "apple" with the representation of the token "work", where this new representation will be closer in meaning to the company Apple than to the fruit.

The Feed-forward neural network takes as input each of the individual tokens and updates their representation. This is done through equation 2.1, where W_k is a weight matrix, b_k is a bias vector, and x is the latent representation of a token - a d-dimensional vector [26]. Crucially, the same weight matrices are used for all tokens in a given block and each block has its own set of weight matrices. This will be the key difference between a normal transformer block and Switch Transformer's approach.

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2 \tag{2.1}$$

I have skipped over some parts like the residual connections [15] and the layer-wise normalization [5]. These methods are employed to stabilize the training process and allow deeper models to be trained.

2.1.2.3 Combining transformers and MoE

The main difference between Switch Transformer and the generic transformer block introduced in section 2.1.2.2 lies in how the feed-forward network (FFN) layer is implemented. In the default transformer block, the FFN layer applies the function given in 2.1 to every token in the sequence provided to it. Notably, the same weight matrices are shared for all inputs in a given transformer block. In contrast, Switch Transformer's switching layer makes the following change:

Instead of a single FFN layer shared by all inputs in a given transformer block, train N different FFN layers (experts) and introduce a router to route individual tokens to specific experts.

[1] Originally, Transformers were mostly used for language modeling but the architecture has been successfully used for other modalities.

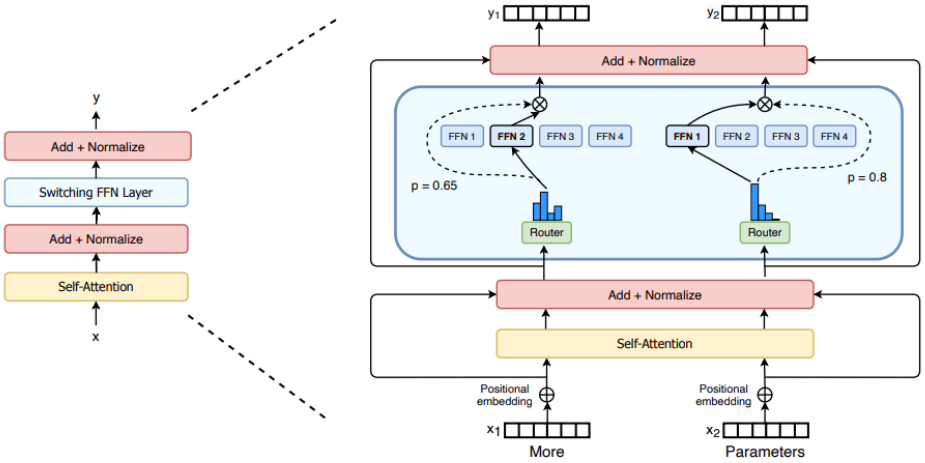

Figure 2.2: Switch Transformer's transformer block. FFN layer is replaced by a switching FFN layer. Graphic taken from [13]

Previously, the model would apply the same FFN layer to every token. In Switch Transformer, the router function takes in the latent representation of the input and for each token, calculates its probability of being routed to expert$_i, i \in \{1...N\}$. For each token, it chooses the expert with the highest routing probability using *argmax*. This is illustrated in figure 2.2. The routing process is illustrated in figure 3.1.

Earlier works [23] conjectured that routing each token to multiple experts was necessary to properly train the routers, but Switch Transformer authors proposed routing each token to at most one expert. This reduces computational complexity and communication overhead in distributed settings.

Replacing the FFN layer with the sparse switching layer increases the model's capacity without significantly increasing its computational complexity. During inference, only one expert in each layer lies on the critical path. In contrast, the traditional scaling approach with dense models increases the number of transformer layers, which adds computations to the critical path - affecting training and inference times. Switch Transformer authors show that their approach yields comparable or better performance with faster training times at the same computation budget (measured in FLOPS).

While the Switch Transformer paper does not conduct an ablation study to interpret the kinds of specializations achieved by individual experts, others have started exploring this area in relation to MoE neural networks. Chen et. al. in their *Towards Understanding Mixture of Experts in Deep Learning* [7] formally prove in their setup that each expert specializes in a specific portion of the data (a cluster) while the router learns to route to individual clusters. They support their findings empirically in both vision and language tasks.

2.1.3 Training

The focus of this project is on the inference of machine learning models, that is, making predictions with an already trained model. While understanding how training of neural networks works is not critical for our purposes, to make the work accessible to readers without a background in machine learning, this section briefly introduces how the weights (parameters) of a neural network are obtained during training. Additionally, I will introduce the most common pre-training task used for training LLMs.

The algorithm used to train neural networks is called back-propagation and it relies on gradient descent. Let us illustrate how gradient descent works by using it to do linear regression.

Suppose we are given some training data and want to fit a linear regression model using gradient descent. The model f is shown in 2.2 and we initialize the weights (also sometimes called parameters) a, b randomly, for instance, $a = 1$ and $b = 1$.

$$f(x) = ax + b \tag{2.2}$$

Our training data is given by pairs $(x, \hat{y})$, where x is the input value and $\hat{y}$ is the corresponding output value. Suppose our training set is $\{(0,3),(1,5),(2,7),(3,9)\}$. We choose squared error as our loss function $L(y, \hat{y}) = (y - \hat{y})^2$.

Let us compute the forward pass of gradient descent using the input $(1,5)$.

$$f(1) = \hat{y} = 1 \times 1 + 1 = 2 \tag{2.3}$$

$$L(2,5) = (2-5)^2 = 3^2 = 9 \tag{2.4}$$

Next, in the backward pass, we determine how much the weights a, b should be modified given the loss from the forward pass.

$$\frac{dL}{dy} = 2(\hat{y} - y)\big|_{y=5,\hat{y}=2} = 2(-3) = -6 \tag{2.5}$$

By the chain rule,

$$\frac{dL}{da} = \frac{dL}{dy}\frac{dy}{da}, \quad \frac{dL}{db} = \frac{dL}{dy}\frac{dy}{db} \tag{2.6}$$

$$\frac{dy}{da} = \frac{d}{da}(ax + b) = x, \quad \frac{dy}{db} = \frac{d}{db}(ax + b) = 1 \tag{2.7}$$

Finally, evaluating the derivatives,

$$\frac{dL}{da} = \frac{dL}{dy}\frac{dy}{da} = -6 \times 2 = -12 \tag{2.8}$$

$$\frac{dL}{db} = \frac{dL}{dy}\frac{dy}{db} = -6 \times 1 = -6 \tag{2.9}$$

To update the weights a, b we use the update equation in 2.10, where W represents any parameter (weight) of the network. The step size α is a hyper-parameter, a parameter not optimized through gradient descent.

$$W = W - \alpha \frac{dL}{dW} \alpha \in \mathbb{R}^+ \tag{2.10}$$

Note, we subtract the gradient because $\frac{dL}{dW}$ is the direction maximizing the increase in L, so we negate it to minimize it.

Let us pick a commonly used $\alpha = 0.003$ and apply equation 2.10 with the partial derivatives computed earlier.

$$a = a - 0.003 \times -12 = a + 0.036 = 1.036 \tag{2.11}$$

$$b = b - 0.003 \times -6 = a + 0.018 = 1.018 \tag{2.12}$$

When we look at our training data, we notice that all the points lie on the function $g(x) = 2x + 3$, while we initialized $f(x)$ as $f(x) = x + 1$, so the increase in both a, b in equations 2.11 and 2.12 corresponds to the data pushing our weights towards the function $g(x)$. With enough iterations and training data, a, b would closely approach 2 and 3 respectively.

An interesting observation is that we computed the value $\frac{dL}{dy}$ only once, but used it to compute the updates for both a and b. Indeed, this is one of the core ideas of the backpropagation algorithm and it is what allows gradients to be computed efficiently even in large networks. [21]

2.1.3.1 Training language models

To extend this approach to language modeling, we would need to design a training task and a corresponding loss function to guide the weight updates. A common NLP pre-training task is masked language modeling (MLM): given some text, randomly mask some of the words, and have the model predict which word from the vocabulary was masked [12]. This can be modeled as a classification task, where the model is trying to maximize the probability of the correct word out of all the possible options in the model's vocabulary.

To compute the loss, we take the outputs of the model, a vector of numbers, normalize them so that each entry is in the range $[0, 1]$, and then compute the negative logarithm of the probabilities. If the model is very wrong and assigns a low probability p to the correct word, the corresponding error is $\lim_{p \to 0} -\log(p) \to \infty$. Given a large volume of data and a model with sufficient parameter count, the model will learn language rules to improve its performance on the MLM task. It has been shown that models trained with MLM can be quickly fine-tuned (a process called transfer learning [4]) on more specific language tasks, such as sentiment analysis.

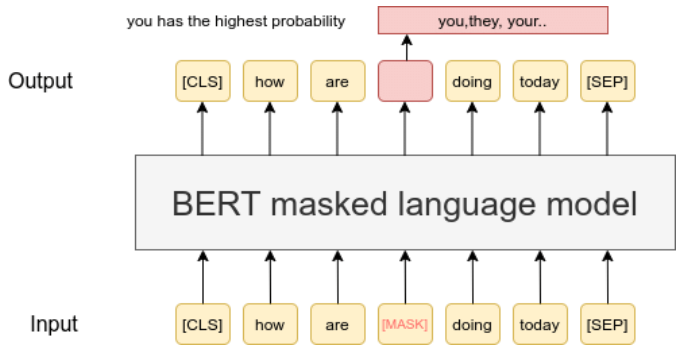

Figure 2.3: Illustration of the masked language modeling pre-training task. `[CLS]` and `[SEP]` are extra tokens used by BERT to indicate the start and end of a sequence. The figure is taken from *Finnish Language Modeling with Deep Transformer Models*[16].

2.2 Deployment Methods

In some settings, deploying a model can be as simple as making a Jupyter notebook[17] available to colleagues. In other settings, it can mean exposing a trained model on an API endpoint on a simple web server, such as Flask. Yet another option is to use optimized inferencing servers such as TorchServe[2], TensorFlow Serving[19], or Nvidia Triton Inference Server[10].

In the next subsection, I show how to deploy a model with Flask and in doing so highlight this approach's limitations to motivate the use of specialized inferencing solutions, such as Nvidia's Triton Inference Server (TRTIS) which I later use in the practical section of this project

2.2.1 Simple web server

To make a trained ML model available to end-users, I can write, for instance, a Flask server that receives POST requests at `/api/<model_name>/predict` and responds with the model's output.

Listing 2.1 shows an illustrative example of a cat and dog image classification API, that I wrote. User sends a base64 encoded image through a POST request to the `/api/cat-or-dog/predict` route. The `catOrDog()` function decodes the image, loads the corresponding model, calls the model with the decoded image, and responds with the model's output. In this case, the output is a `probabilities` array of length two where each index is the probability of the image being of a cat or a dog. I skipped some technicalities, like error handling or image pre-processing for brevity.

Listing 2.1: Deploying a model with Flask

```
1 from flask import Flask, jsonify, request
2 import torch
3 import torchvision.transforms as transforms
4 import base64
5 from PIL import Image
6
```

```
7  app = Flask(__name__)
8
9  @app.route('/api/cat-or-dog/predict', methods=['POST'])
10 def catOrDog():
11     b64_image = request.data
12     decoded_image = base64.b64decode(b64_image)
13     image = Image.open(io.BytesIO(decoded_image))
14     image_tensor = transforms.ToTensor()(image).unsqueeze_(0)
15
16     model = torch.load("/models/cat_or_dog.pt")
17     predictions = model(image_tensor).tolist()
18     return jsonify({'probabilities': predictions})
```

While the above approach can work, it has some limitations worth discussing.

2.2.1.1 Productivity

If a developer wants to deploy a new model, they need to upload the model to the `\models\` folder, create a new route, and write its controller. On top of that, in a larger company, we would not expect a data scientist to also do backend development, so they would either need to divert resources to learn new tooling or they would delegate deployment to a different team - leading to communication overheads.

It would be more productive if all the data scientist had to do was upload the trained model to a model repository along with a configuration file specifying details like model format. The deployment server could then automatically generate an API endpoint.

2.2.1.2 Efficiency

The second main concern relates to speed and efficient use of resources. Looking again at the Flask example, we can notice that for every request the function loads a model from memory into ram (or to GPU memory with a slight modification). Since this is a fairly slow operation, even a handful of requests could make the application feel unresponsive.

To alleviate this issue, we could cache popular models so that the loading speed is reduced. Additionally, we might queue multiple requests for the same model and then run a single inference to exploit parallelism.

2.2.1.3 This is getting pretty complicated

What started off as a simple solution would quickly turn complicated if we tried to address the above-mentioned shortcomings. Luckily, there is a solid selection of ML inference servers out there whose developers have already done this work. Let's look more closely at how Nvidia's Triton Inference Server works and the user experience.

2.2.2 Triton Inference Server

Triton Inference Server (TRTIS) is Nvidia's open-source ML inference server written in C++[10]. It allows users to deploy AI models trained using many of the popular ML and DL frameworks, such as PyTorch, TensorFlow, ONNX, Nvidia's own TensorRT, and more. TRTIS can be deployed in the cloud, on-prem, and on edge devices. It can execute models on Nvidia GPUs, x86 and ARM CPUs, and some hardware accelerators, such as AWS Inferentia.

TRTIS supports communication through HTTP and gRPC, where gRPC is generally the more performant option, largely due to its use of Protocol Buffers[1].

The server also supports pipelines, where the output of one model is forwarded as input to another model. More complicated routing logic can be defined using custom scripts. Triton also provides dynamic batching, automatic performance tuning, and a variety of diagnostic tools.

2.2.2.1 Deploying model to Triton

The recommended way to run TRTIS is as a docker container. One of the required start-up arguments is a path to the folder containing your models. To deploy a new model, we simply need to add the model and its configuration file to this directory. Figure 2.4 shows the structure of a model repository with two models: a PyTorch model `model_A` and an ONNX model `model_B`.

```
model_repository
├── model_A
│   ├── config.pbtx
│   └── 1
│       └── model.pt
└── model_B
    ├── config.pbtx
    └── 1
        └── model.onnx
```

Figure 2.4: Structure of model repository

A minimal configuration must specify the platform, the maximum batch size, and the input and output tensors of the model. For the PyTorch model from figure 2.4, an example minimal configuration file is shown in listing 2.2.

The platform option is guided by the format of our saved model, which is PyTorch in this case. The input option specifies the number of inputs for the model, their names,

types, and dimensions. In this case, the model expects two inputs `input0` and `input1`, which are both 16-dimensional vectors of 32-bit floating point numbers. The model outputs `output0`, which is again a 16-dimensional FP32 vector. For the full list of options, refer to TRTIS's documentation.

Listing 2.2: Triton configuration file example

```
platform: "pytorch_libtorch"
max_batch_size: 8
input [
    {
      name: "input0"
      data_type: TYPE_FP32
      dims: [ 16 ]
    },
    {
      name: "input1"
      data_type: TYPE_FP32
      dims: [ 16 ]
    }
]
output [{
      name: "output0"
      data_type: TYPE_FP32
      dims: [ 16 ]
    }]
```

2.2.2.2 Inferencing on deployed models

When starting TRTIS, we can choose at which ports it will be listening for HTTP and gRPC requests. The most convenient way to make requests to the server is through Triton's client library, which provides a simple API for interacting with the server. This library is available in multiple languages, including Python.

Listing 2.3 shows an example of using the Python library to make a single request to the server.

Listing 2.3: Triton client example

```
import numpy as np
import tritonclient.grpc as grpcclient

triton_client = grpcclient.InferenceServerClient(
           url="localhost:8001")

inputs = []
outputs = []
inputs.append(grpcclient.InferInput('input0', [1, 16], "FP32"))
inputs.append(grpcclient.InferInput('input1', [1, 16], "FP32"))

# Randomly generate data for the two input tensors
```

```
input0_data = np.random.random(shape=(1, 16))
input1_data = np.random.random(shape=(1, 16))

# Initialize the data
inputs[0].set_data_from_numpy(input0_data)
inputs[1].set_data_from_numpy(input1_data)

outputs.append(grpcclient.InferRequestedOutput('output0'))

# Test with outputs
results = triton_client.infer(
    model_name="model_A",
    inputs=inputs,
    outputs=outputs)

# Get the output arrays from the results
output0_data = results.as_numpy('output0')
```

Chapter 3

Experimental setup

In this chapter, I describe the setup used for experiments. This setup aims to measure the overhead of dynamically loading and unloading Switch Transformer experts to/from Triton. For this reason, I only work with the first switching layer which was introduced in section 2.1.2.3.

3.1 Deploying Switch Transformer on Triton Inference Server

Switch Transformer was built around Mesh TensorFlow, Google's open-source library for distributed deep learning [22]. Its main use case is for training large models that cannot fit on a single machine. While Mesh TensorFlow can be used for inference, we would like to run Switch Transformer on Nvidia's Triton Inference Server to benefit from a fully-fledged inferencing solution.

3.1.1 Splitting Switch Transformer into PyTorch models

To deploy Switch Transformer on Triton, the Mesh TF model needs to be converted to a supported format. A Ph.D. student in the same lab converted the Switch Transformer experts into standalone PyTorch files. After this conversion, each expert is a standalone PyTorch model that can be loaded in Triton as shown in section 2.2.2.1. Each expert expects a $(k, 768)$ real-valued matrix, where k is the batch size. This is so that if in a single input sentence, multiple tokens are dispatched to the same expert, we can stack the corresponding feature vectors and make a single inference request. The return shape is identical.

3.1.2 Triton Custom Backend Dispatcher

Now that experts can be deployed, we need a way to receive requests, dispatch them to corresponding experts, and return the results back to the caller. We can achieve this with Triton custom backend [11]. These can be deployed just like any other model but

can run arbitrary code. Notably, the backend can be configured to forward requests to other models according to a custom routing function.

Leyang Xue, a PhD student working in the same lab, wrote a Python backend that serves as a dispatcher to experts. In this project, I extend Leyang's dispatcher. The dispatcher requires each request to provide three inputs:

- `hidden_states` with shape (128, 768)
- `routes` with shape (128, 128)
- `route_prob_max` with shape (128, 1)

Figure 3.1 illustrates how the dispatcher receives requests and dispatches them to experts. When the dispatcher receives input, it inspects the `routes` matrix, which specifies for each token to which expert (if any) it is to be dispatched. If multiple tokens are dispatched to the same expert, their feature vectors are stacked into an $(k, 768)$ matrix so there is only one inference request. Once all experts return, the `hidden_states` matrix is updated and the size of the update is scaled down by the `route_prob_max` scalar. The updated matrix is then returned back to the original caller of the dispatcher.

In a full deployment, `routes` and `route_prob_max` would be determined by a *router* as explained in section 2.1.2.3. Instead, we are working without a router and specify the routes manually when making a request. To be able to run experiments with realistic routing decisions, a PhD student in the same lab traced the routing decisions made by the routers in a full Switch Transformer.

3.2 Adding Fetching Engine

Triton supports three control modes for determining which models from the model repository are loaded in memory.

None control mode attempts to load all models in the model repository at startup. If a model fails to load (due to lack of memory, incorrect configuration, or otherwise) Triton will not re-attempt to reload the model. Any updates to the model repository will require an explicit restart to take effect.

Explicit control mode requires the user to specify which models from the repository should be loaded at startup. Further loading and unloading of models has to be done through the model control protocol. Triton provides a Model Repository Extension API with support for both gRPC and REST/HTTP. To load a new model with the REST/HTTP API, it is sufficient to make a POST request to `v2/repository/models/${MODEL_NAME}/load`. Similarly, one can unload a model this way.

Poll control mode periodically checks the model repository for changes and tries to load newly added models. This mode does not allow the use of the loading/unloading API.

Since our focus is on deploying Switch Transformer on systems without sufficient memory to fit the whole model in GPU memory, we will need to use the explicit

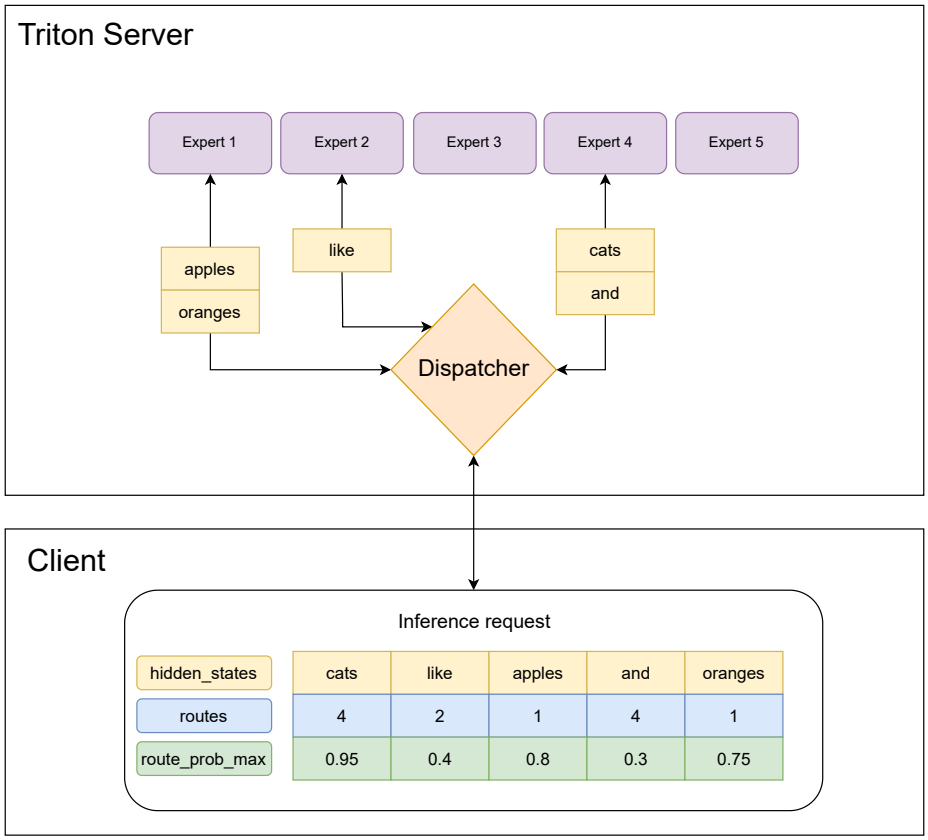

Figure 3.1: This figure illustrates how the dispatcher handles an inference request. `hidden_states` is shown as an array of words (tokens), but this is just to simplify the notation. In reality, each word would be a real-valued 768-dimensional feature vector representing the given word. Notice how the dispatcher only makes three inference requests to individual experts by stacking together feature vectors dispatched to the same experts. The route probabilities are used when updating hidden states with the outputs of individual experts but it is not shown on the diagram. In this project, I work with a Switch Transformer with 128 experts in its switching layer.

control mode and manage which models are loaded in GPU memory through the Model Repository Extension API. Because experts need to be loaded and unloaded dynamically depending on the routing decision, it is necessary to implement a fetching engine to handle it.

3.2.1 Designing the Fetching Engine

The role of a fetch engine is to dynamically load and unload experts on demand while ensuring that the number of experts loaded in memory does not exceed the allowed memory capacity. Additionally, we want it to collect fetching metrics so that we can measure the impact of fetching on performance.

The key parts of the fetch engine I designed are shown in listing 3.1. The key method is `load_model()`, which is called by the router before a request can be dispatched to a particular expert. The method does one of three things depending on the GPU state:

- Model is already loaded in GPU memory and so no further work needs to be done.
- Model is not in memory and there is free capacity so the model can be immediately loaded.
- Model is not in memory and memory capacity is full. First, a different model must be evicted after which the requested model can be loaded. The eviction policy can have a significant impact on performance. I take a FIFO approach, where the oldest loaded model will be evicted first.

Depending on which case a given load falls under, the engine correspondingly updates the metrics it tracks. The `load_model()` method is called by the dispatcher before it forwards a request to the corresponding expert. During experiments, the fetch engine allows me to control how many experts can be loaded in GPU memory at any given time, which will effectively simulate running this setup on a memory-constrained machine.

I chose to use the First In, First Out eviction policy. Other viable options include LRU or even random replacement. Future studies could investigate the impact of different fetch engines on memory hit rates.

Listing 3.1: Fetch Engine Implementation

```
1  class FetchEngine:
2      def __init__(self, mem_capacity, num_experts, client):
3          #<--redacted-->
4
5      def load_model(self, expert_num):
6          self.num_requests += 1
7          if self.is_model_loaded(expert_num):
8              self.num_hits += 1
9              return True
10
11         while self.get_model_count() >= self.mem_capacity:
12             model = self.load_history.popleft()
```

```
13              self.client.unload_model(f"switch-base-128
                    _encoder_expert_1_{model}")
14              self.num_evictions += 1
15
16          self.client.load_model(f"switch-base-128_encoder_expert_1_{
                expert_num}")
17          self.num_fetches += 1
18          self.load_history.append(expert_num)
19          return True
20
21      def is_model_loaded(self, model):
22          model_name = f"switch-base-128_encoder_expert_1_{model}"
23          return self.client.is_model_ready(model_name)
```

3.3 Data

The dataset I used for this project was collected by tracing the routing decisions of a full Switch Transformer Base with 128 experts per switching layer. The inputs consisted of English queries. This dataset was generated by a Ph.D. student from the same research lab.

3.4 Setting Triton parameters

The bash script in Listing 3.2 shows the configuration used for running the server when running experiments.

Listing 3.2: Triton start-up script

```
1 docker run --rm --gpus='"device=2"' \
2 --name triton_michal -p 8000:8000 -p 8001:8001 -p 8002:8002 \
3 --shm-size=10gb --ulimit memlock=-1 \
4 -v /mnt/raid0nvme1/michal:/models \
5     futurexy/tritonserver_batcher:0120 tritonserver \
6 --model-repository=/models/model_repo_switch-base-128 \
7 --model-control-mode=explicit \
8 --load-model dispatcher \
9 --log-error=1
```

3.5 Hardware and software

All work and experiments have been done on the `gala1` server running Ubuntu 20.04 LTS. The machine is equipped with 8 NVIDIA RTX A5000 each with 24GB of VRAM, a dual-socket motherboard housing two AMD EPYC 7453 28-Core processors, and 1TB of RAM. For experiments, only one GPU card is used. The available memory is controlled through the fetch engine.

Since the server is shared by multiple researchers, it is impossible to completely eliminate the effect of other users on the testing performance. To mitigate this, a completely unused GPU was chosen to run experiments, and experiments were repeated multiple times to account for random variations.

3.6 Building a strong baseline system

To evaluate whether my proposed changes improve the system performance, let us start with a strong baseline. In this case, the only difference will be the batching algorithm used. For the strong baseline, we can use Triton's dynamic batcher[9]. This can be enabled as an option in a model's configuration file. For our purposes, it makes sense to enable batching in front of the Python dispatcher.

3.6.1 Dynamic Batcher

Dynamic Batcher by default waits until it receives
`max_batch_size` requests or until a default timeout period expires and then sends a batched request to the model. Optionally, we can specify `preferred_batch_size`, a list of integers representing which batch sizes are preferred. This might make a difference when working with specific hardware, i.e. it might be advantageous to use powers of two batch sizes.

We can also specify `max_queue_delay_microseconds`, which will queue requests until either `max_batch_size` is met or the oldest request's queue time exceeds this delay. This is especially useful for latency-sensitive systems, such as content recommender systems on websites like YouTube, where the latency of the model affects the page load time. Akamai Technologies, Inc. in their *The State of Online Retail Performance* 2017 report showed that optimal load times for peak conversion rates ranged from 1.8 to 2.7 seconds while increases in load time positively correlated with bounce rates and reduced conversion rates [3].

At the time of writing, there is an open issue on Triton's GitHub mentioning that Triton does not combine batched inference requests into a single tensor for Python backends but rather provides the batch as an array of inference requests. Because of this, a dedicated experiment will measure the effect of stacking batched inference requests into a single tensor as opposed to executing each one sequentially. Figure 3.6.1 illustrates how the dynamic batcher with stacking implemented interacts with the dispatcher. Notice how *apples, berries*, and *oranges* are all routed to expert 1 and executed as a single inference request.

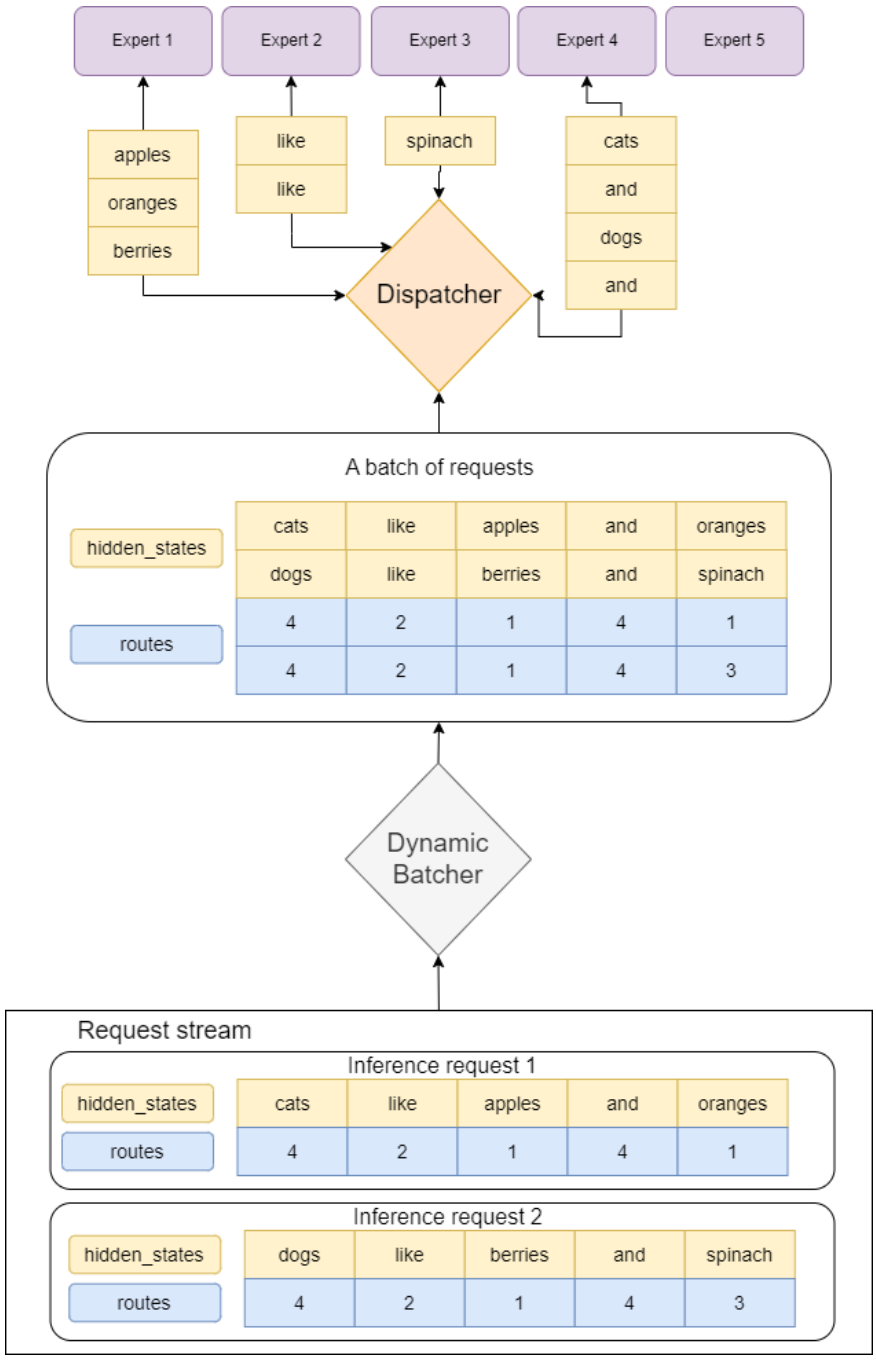

Figure 3.2: Dynamic Batcher with request stacking implemented.

3.7 Code and Reproducibility

All code and testing scripts are stored in a private GitHub repository. Because the repository is shared by other research students, there is no current plan to make the whole repository public. However, access can be given on request.

Chapter 4

Experiments

4.1 Baseline

To measure the default throughput of the switching layer, let us run baseline experiments without any batching. For throughput experiments, we can use the `triton_client.async_infer()` method to quickly send the entire dataset. As mentioned in the previous chapter, I will be running experiments with just a single layer of the Switch Transformer, as this will allow me to run more experiments and results should be extrapolate-able to the full model.

In this section, I conduct experiments with the following settings:

- Dispatcher without batching
- Dispatcher with dynamic batching (batch size 64)
- Dispatcher with dynamic batching and stacking (batch size 64)

For each of these settings, the allocated memory capacity is varied which limits how many experts fit in memory at a given time. This will let us see the effect of decreasing memory on throughput. The expected decrease in performance is caused by the additional IO overhead of loading and unloading experts to and from the GPU. Reduced memory also affects how many experts can be executing in parallel. For each setting, the same 2000 requests from the routes dataset are used. Three runs are used and the routes are shuffled before each run to get a representative behavior. The random seed is fixed at 42 so that different experiments use the same route orderings.

4.1.1 Discussion of baseline results

Figures 4.1 and 4.2 show the effect of reducing memory capacity on the system's throughput and the number of resulting expert load operations (unloads are not included, but the numbers are strongly correlated). The time for throughput is measured as the duration from the first inference request until the final inference response returns.

It is notable how quickly throughput degrades for the dispatchers without batching and the default dynamic batcher. When the memory capacity is reduced from 128 to

100, where 128 is the total number of experts in our system, the throughput drops from 53.3 to 5.9 for no batching and from 57.5 to 6.1 with dynamic batching. Subsequent reductions in memory capacity still reduce performance, but the relative decrease is considerably smaller in comparison.

On the other hand, the dispatcher with stacking implemented not only has a much higher throughput at full memory capacity but its performance degradation as memory capacity decreases is considerably less severe. As explained under figure 4.1, this performance increase is both due to increased parallelism and due to a reduction in the number of load requests.

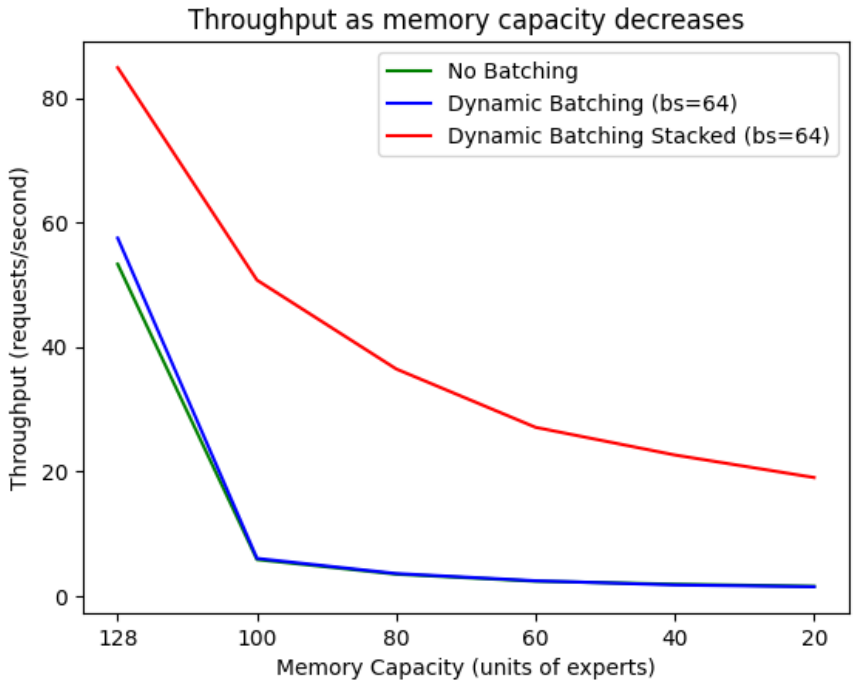

Figure 4.1: As mentioned in section 3.6.1, the dynamic batcher does not automatically stack batched requests into a single tensor, but rather simply passes batched requests as an array. This explains why `No Batching` and `Dynamic Batching` exhibit the same performance degradation as this type of batching does not improve parallelism. `Dynamic Batching Stacked` is a version of the dispatcher that takes the provided array of requests and stacks these into a single NumPy tensor. This allows the dispatcher to group together tokens destined for the same expert and only make a single inference call for each group, hence vastly improving parallelism. Simultaneously, this reduces the number of expert loads and unloads.

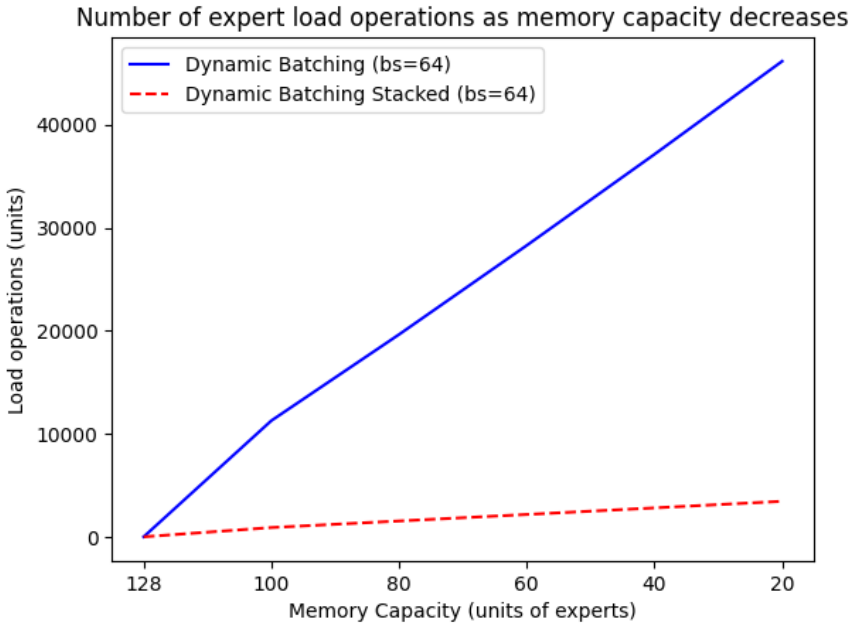

Figure 4.2: This figure shows the reduction in load operations resulting from stacking batched inference requests into a single tensor. Without stacking, when the dispatcher received a batch of requests where some of them activated the same experts, the dispatcher would sequentially go through each request and load/unload experts as necessary. This meant that in any given batch, the same expert might be unloaded only to be immediately reloaded for the next request. With the stacked implementation, all tokens corresponding to the same expert can be grouped together, resulting in at most one load per expert for the whole batch.

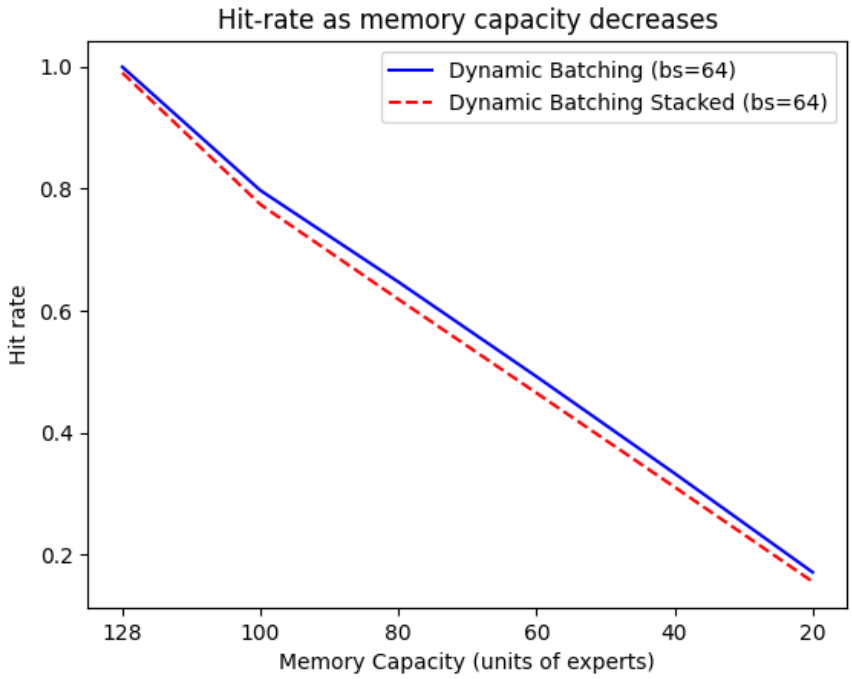

Figure 4.3: This figure shows how the memory hit rate of the fetch engine varies with memory capacity. Although the absolute number of memory loads is significantly higher for the dispatcher without stacking implemented, as shown in figure 4.2, the hit rates are comparable for both dispatchers. Hit rate might not be the best metric to use as it is hard to keep it representative when comparing sequential and parallel implementations. Hit rate can be useful for comparing the same type of dispatchers but with a different batching algorithm used, for instance.

4.2 Proposed batching algorithm

Having seen the massive uplift in performance from stacking the batches provided by the dynamic batcher into a single tensor to exploit parallelism, this section attempts to modify the behavior of the default dynamic batcher to further improve throughput.

4.2.1 Understanding performance bottlenecks

The section 4.1.1 clearly shows that reduced memory capacity significantly affects throughput. This section dives deeper into how exactly memory capacity causes reduced throughput.

When an inference request arrives at a dispatcher running with high memory capacity, it is more likely that the activated experts are already loaded in the GPU memory. The top part of figure 4.4 illustrates that in such a case, the dispatcher can asynchronously forward feature vectors to the specified experts, resulting in high expert-level parallelism.

With low memory capacity, the likelihood that all of the experts for a given request are in memory is small. If an expert is not in memory, the fetch engine has to load it, which blocks the dispatcher until the expert is loaded. Because each expert only executes for

a short amount of time, the loading overhead is significant and reduces the effective expert-level parallelism. This is visualized in the bottom section of figure 4.4.

Figure 4.5 illustrates a more direct way in which memory capacity hampers performance. The fetch engine's memory capacity limits how many experts can be loaded in GPU memory at any given time. If a request activates more experts than fit in memory, the dispatcher has to dispatch parts of the request to only those that do fit, wait for those to return, and only at this point can these experts be unloaded from GPU memory in favor of the remaining ones. This process might have to be repeated multiple times. This directly limits the system's maximum expert-level parallelism as at most `max_memory_capacity` experts can be executing in parallel.

As it stands, reduced memory capacity can severely limit the system's parallelism. At the same time, we have observed massive performance uplift from batching requests by improving token-level parallelism. This is illustrated in figure 3.2. However, simply increasing batch size does not necessarily improve performance. In my tests, going beyond 64 in fact degraded it. If increasing the batch size does not improve performance, can we instead improve the batching algorithm to group requests that activate the same experts? This idea is discussed in the next section.

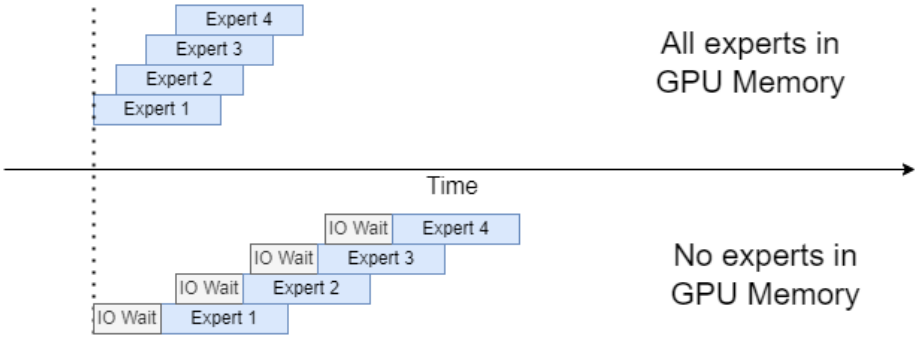

Figure 4.4: The top part of this figure shows expert-level parallelism when all involved experts are already loaded in GPU memory. When this is the case, the dispatcher can asynchronously dispatch requests to the experts and multiple experts run in parallel. The bottom part shows the same situation, but when none of the involved experts are in GPU memory. The fetching engine has to load each expert before the dispatcher can dispatch a request to it. This significantly affects the achievable expert-level parallelism and hence the end-to-end latency and throughput.

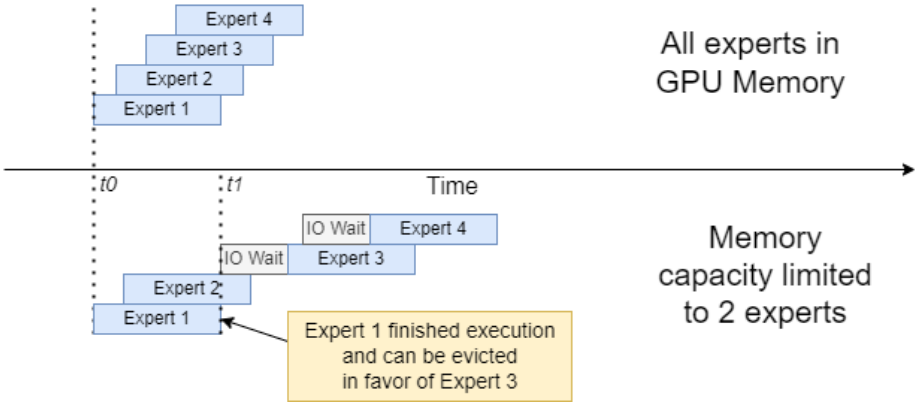

Figure 4.5: Memory capacity limits the maximum number of experts executing in parallel.

4.2.2 System description

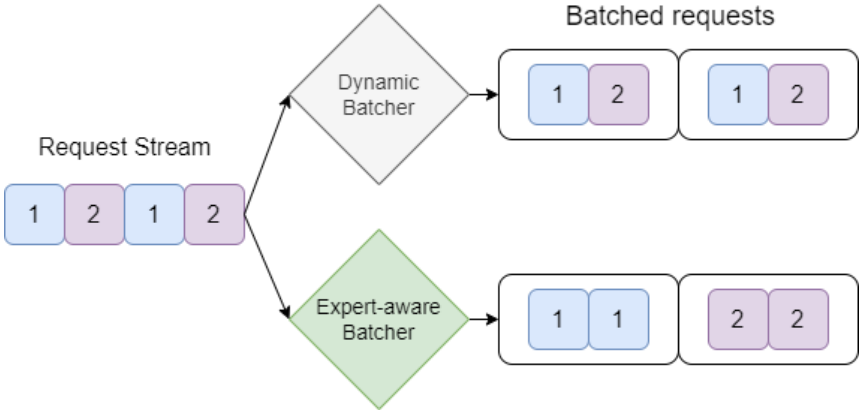

Figure 4.6: This figure depicts a simplified stream of requests in which each request is illustrated as the expert it activates. In practice, each request activates multiple experts. For the sake of the example, assume the memory capacity is 1 and that the maximum batch size is 2. The dynamic batcher batches requests in what is essentially the arrival order. However, the arrival order is not always optimal. As seen in the figure, the dynamic batcher groups together two requests each activating a different expert. The expert-aware batcher instead groups together requests which activate the same experts - reducing the total number of model IO operations necessary and improving parallelism.

This section discusses a proposal for an expert-aware batcher. An illustration of this idea is shown in figure 4.6. In essence, instead of just batching requests by arrival order, it might be beneficial to cluster requests by the experts they activate.

The key is to find a suitable distance function to use for clustering such that the number of additional expert load operations is minimized. We can represent which experts are loaded by the current batch as an N-dimensional binary vector $\vec{v}_b$, where N is the total number of experts[1]. Activations of a single request can be similarly expressed by $\vec{v}_r$.

[1] In our experiments we use Switch Transformer Base with 128 experts.

Summing the vector gives the total number of unique activated experts, as shown in equation 4.1.

$$\text{Unique activated experts by a batch} = \sum_{n=1}^{N} \vec{v_n} \tag{4.1}$$

A simple approach to clustering is to use the Manhattan distance. Given a batch and a set of candidates, add the candidate with the smallest Manhattan distance from the batch. However, this has a major weakness as it also penalizes candidates for not activating all of the experts already activated by the batch. Equation 4.2 shows the Manhattan distance between a batch activating experts 1 and 2 and a request activating just expert 2. The Manhattan distance is 1 in this case, as the vectors only differ in the first entry. Equation 4.3 shows that a request activating experts 1, 2, and 3 also has a distance of 1 from the same batch. This is problematic, because in 4.2 the request is a subset of the batch and results in no additional expert loads, whereas 4.3 demands an additional load for expert 3.

$$Manhattan\left(\vec{v_b} = \begin{bmatrix}1\\1\\0\end{bmatrix}, \vec{v_r} \begin{bmatrix}0\\1\\0\end{bmatrix}\right) = 1 \tag{4.2}$$

$$Manhattan\left(\vec{v_b} = \begin{bmatrix}1\\1\\0\end{bmatrix}, \vec{v_r} \begin{bmatrix}1\\1\\1\end{bmatrix}\right) = 1 \tag{4.3}$$

Clearly, we need a better metric that only penalizes additional loads. In equation 4.4 I introduce a custom metric I call the *PositiveManhattan*. It differs from the normal Manhattan distance in that a request is only penalized for additional loads not already included in a batch. Because of this, the order of arguments matters. Equations 4.5 and 4.6 show how the new metric can differentiate between the requests that simple Manhattan distance could not.

$$PositiveManhattan(\vec{v_b}, \vec{v_r}) = \sum_{i=1}^{N} \max(\vec{v_{ri}} - \vec{v_{bi}}, 0) \tag{4.4}$$

$$PositiveManhattan\left(\vec{v_b} = \begin{bmatrix}1\\1\\0\end{bmatrix}, \vec{v_r} \begin{bmatrix}0\\1\\0\end{bmatrix}\right) = 0 \tag{4.5}$$

$$PositiveManhattan\left(\vec{v_b} = \begin{bmatrix}1\\1\\0\end{bmatrix}, \vec{v_r} \begin{bmatrix}1\\1\\1\end{bmatrix}\right) = 1 \tag{4.6}$$

4.2.3 Implementation

This subsesection provides implementational details for my expert-aware batching algorithm. Listing 4.1 shows how the batcher operates with pseudocode. The worst-case time-complexity of the algorithm is $O(N^2)$, where N is the number of queued requests.

This paragraph provides some clarifications around the algorithm. The `calculateDistances()` function calculates the *PositiveManhattan* distance for every candidate in the queue from the current batch. The `initializeBatch()` randomly selects one queued request as the origin of the batch. I also tried other implementations for `initializeBatch()`, one where the first element of a batch is determined as the request with the smallest *PositiveManhattan* distance from the experts currently loaded in GPU memory. Another approach simply picked the expert activating the fewest experts. Regardless, all approaches behaved almost identically so only results from random initialization are reported.

I considered implementing my batcher as an extension to Triton's Dynamic Batcher. Triton allows programmers to extend the functionality of the Dynamic Batcher by implementing a set of predetermined methods in a header file and linking it as a shared library [8]. However, this only allows for relatively limited modifications to Dynamic Batcher's behavior - mostly related to when should a batch be ended but does not allow any re-ordering of requests.

I could have implemented my batcher by forking the Triton core repository and overwriting their definition of Dynamic Batcher for full control, but this would be quite a challenging undertaking under the time-limited conditions. As a proof of concept, I decided to implement my batcher inside of the dispatcher and use NumPy to achieve reasonable performance. This approach gives up some performance and hence throughput but should allow me to determine the merits of this batching strategy by analyzing any reduction in memory load operations.

In my implementation, I rely on the Dynamic Batcher to queue requests but increase the batch size. Inside my batcher, I read a large batch provided by the Dynamic Batcher and reorder it into smaller batches (bs=64 for a fair comparison with previous approaches) before passing them onto the dispatching logic. While doing this, I need to keep a mapping of the original order of requests so that when responses are sent back, I send them in the correct order.

Listing 4.1: Pseudocode for expert-aware batching

```
queue = getQueuedRequests()
batch = initializeBatch()
batches = []
while queue:
    distances = calculateDistances(batch, queue)
    idx = argmin(distances)
    batch.add(queue[idx])
    queue.remove(idx))
    if batch.isFull():
        batches.add(batch)
```

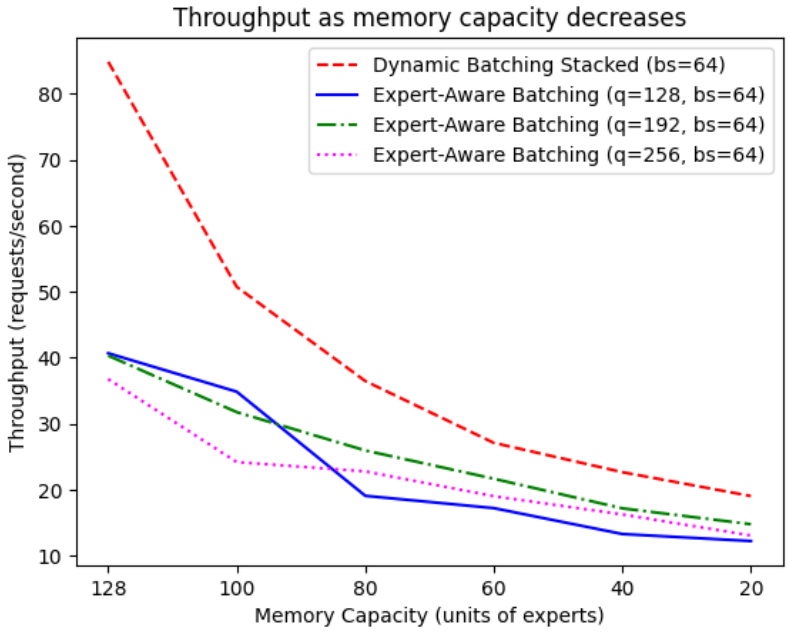

Figure 4.7: This figure reports the throughput of expert-aware batching strategies with different queue sizes and compares them against the strong baseline.

```
11          batch = initializeBatch()
12  if batch:
13      batches.add(batch)
14  return batches
```

4.2.4 Results

This section discusses the experimental results of my expert-aware batcher. Figure 4.7 shows that my expert-aware batcher did not improve the end-to-end throughput of the system, regardless of the queue size. There are two main reasons why changing the batcher might negatively affect performance: new batches result in more expert load operations or the time it takes to execute the batcher itself adds to the total execution time.

To determine whether the expert-aware batcher produces batches that result in more GPU memory load operations, we can examine figure 4.8. We notice that the expert-aware batcher produces batches that result in the same or marginally fewer load operations. The degree of improvement is proportional to the queue size. The largest effect is observed at the memory capacity of 20 experts, where a 4% reduction in loads for the queue size of 256 is observed. Although the improvement is marginal, it is consistent across multiple experiments and different memory capacities. This shows that the batcher is at least as good as the dynamic batcher which relies purely on arrival order. However, the computational complexity of the expert-aware batcher is $O(n^2)$, and any gains from reduced memory loads are nullified by the higher computational cost.

The hit rate is unchanged by the change in a batching algorithm.

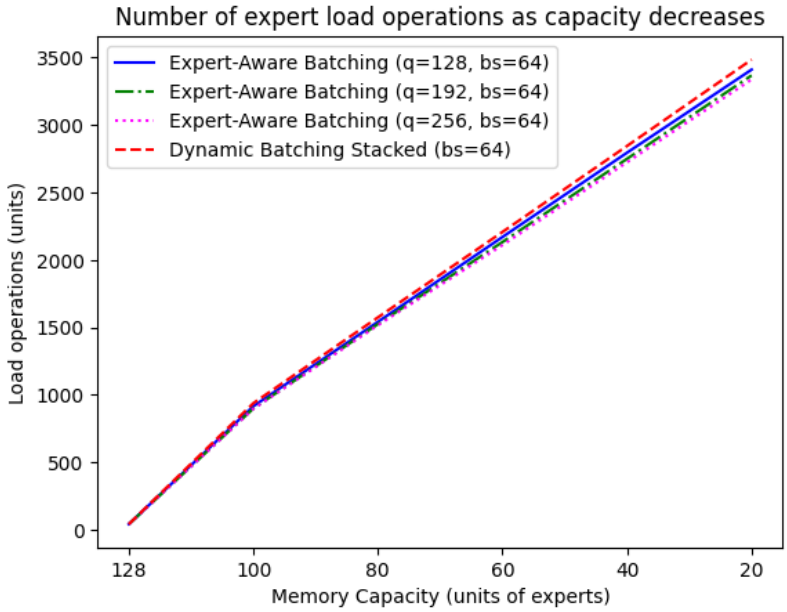

Figure 4.8: This figure highlights how expert-ware batching can achieve a lower number of total expert load operations but the reduction is marginal. The largest observed reduction is around 400 loads at 20 memory capacity, a 4% improvement.

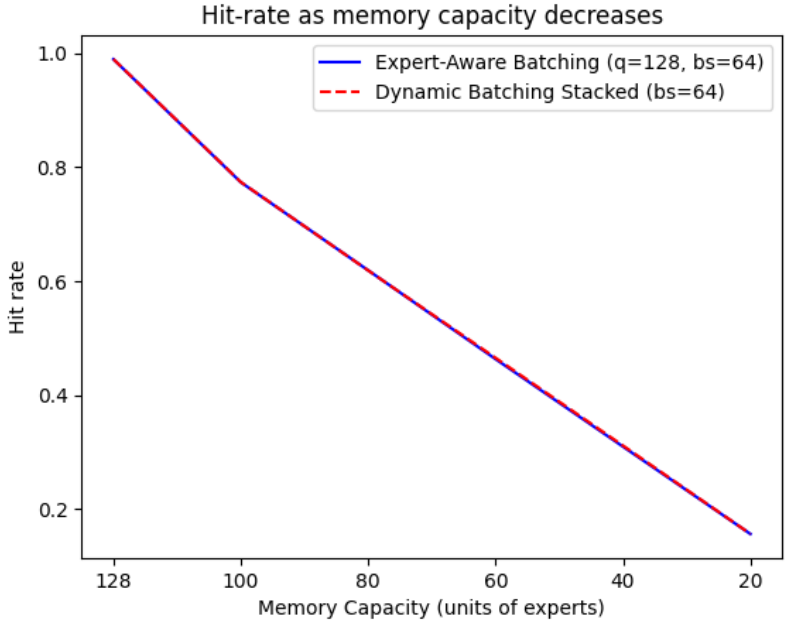

Figure 4.9: There are effectively no changes in hit rate as a result of the batching algorithm. I only report for one queue size as larger queue sizes behaved comparably. Any improvements as a result of expert-aware batching are already accounted for as a decrease in the total number of loads as shown in figure 4.8.

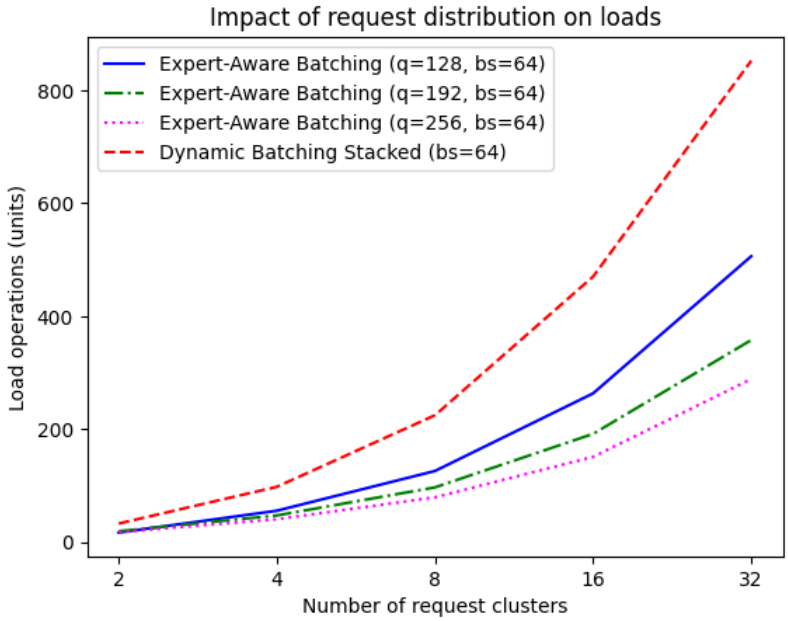

Figure 4.10: Evaluating the impact of expert-aware batching on the number of load operations by the fetch engine using synthetic requests. Memory capacity is limited to 1 expert.

4.3 Analysis

In this section, I analyze why the expert-aware batcher does not improve performance as expected. I focus the analysis on changes in the number of expert load operations. My starting hypothesis is that expert activations (routes) do not group into nicely separable clusters. First, I test whether the expert-aware batcher can improve performance for synthetically generated routes which do separate into non-overlapping clusters. Next, I explore the distribution of routes in the dataset used for testing.

4.3.1 Analyzing synthetically generated routes

This subsection explores the performance of the expert-aware batcher under idealized conditions on synthetically generated routes. These routes were generated such that each request only activates one expert. I vary the number of experts from which requests are uniformly sampled.

Figure 4.10 shows the impact of the distribution of requests on the average number of load operations. Using the expert-aware batcher significantly reduces the number of expert loads compared to the baseline dynamic batcher. The large reduction in expert loads reinforces the starting hypothesis that the expert-aware batcher can significantly reduce IO overhead but the original dataset does not separate into non-overlapping clusters.

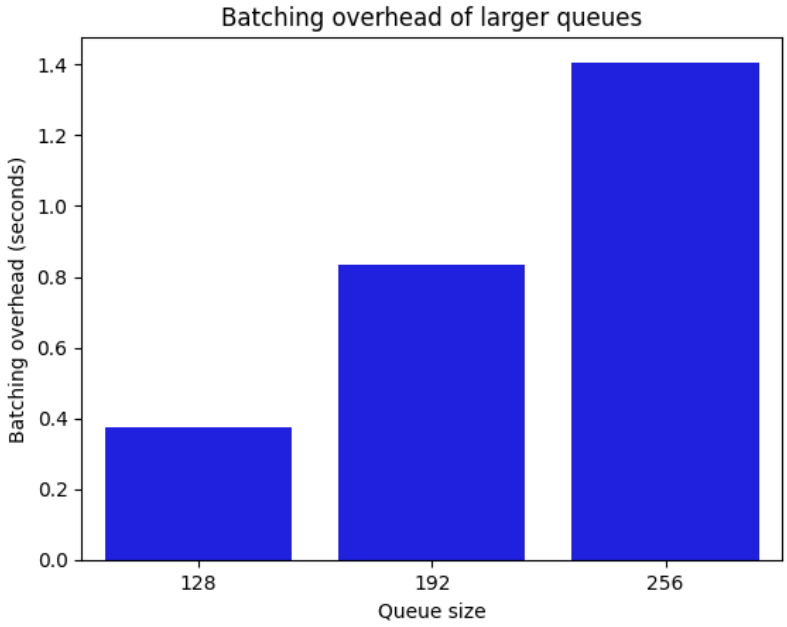

Figure 4.11: The average time it takes to prepare batches given a queue size. As expected given the quadratic time complexity of the expert-aware batcher, doubling the queue size roughly quadruples the batching time. This does not include the queuing time itself which also grows as queue size increases.

4.3.2 Overhead of expert-aware batching

Figure 4.11 highlights the computational overhead of the expert-aware batcher. Although the batcher was written to take advantage of NumPy's vectorization, likely, implementing the batcher in a more performant language could significantly reduce these batching times.

4.3.3 Exploring routes distribution

This subsection analyzes the distribution of routes in the original dataset and attempts to determine if the dataset is separable into clusters. The activation patterns produced by Switch-Transformer's routers are fairly complex. Each request can activate up to 128 different experts at the same time.

An average request activates 27.8 unique experts. The probability of any individual expert being activated by a given request ranges from 10% up to 45% with the mean and median sitting at 20%.

To visualize how easily the data would cluster, I used a dimensionality reduction method t-SNE[25] to project the routes dataset down to two dimensions. Additionally, I used the same technique to visualize the synthetically generated routes for comparison. The plots are shown in figure 4.12. Dimensionality reduction methods can be misleading so the plots serve mainly an illustrative purpose. However, notice how noisy the original dataset is compared to the synthetically generated data. This would limit the expert-aware batcher's ability to group requests into clusters that activate similar experts.

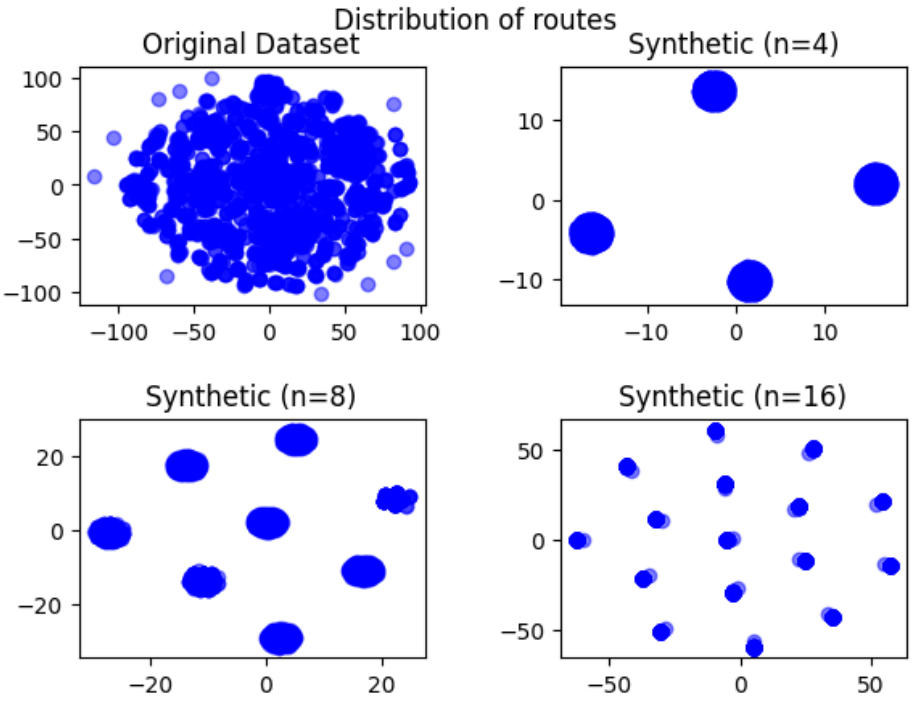

Figure 4.12: t-SNE visualization of routes in the original dataset and synthetically generated routes. The variable n controls how many clusters the synthetic dataset should generate. All datasets are of the same size. In the synthetic plots, each cluster corresponds to roughly $\frac{2000}{n}$ routes, because the datasets were constructed to cluster well. In comparison, the original dataset is a lot noisier.

Chapter 5

Conclusions

This thesis has focused on reducing the IO overhead of fetching expert models for the Switch Transformer architecture and more broadly for large Mixture of Experts neural networks. The main contributions of the work are:

- Making deployment of Switch Transformer possible on memory-constrained devices using a fetching engine
- Showing massive performance improvement through request stacking, a feature, at the time of writing, missing in Triton's Python custom backends
- Analysis of reduced memory on Switch Transformer's parallelism
- Proposal of expert-aware batcher and showing its limited impact on IO overhead
- Analysis of expert-aware batcher using synthetic data

Although the proposed expert-aware batcher failed to improve real-world throughput, it lead to an informative investigation into the distribution of request routes. Future work could investigate replacing Triton's dynamic batcher entirely. The fetch engine allows the deployment of sparse models on memory-constrained devices. It would be interesting if this functionality was incorporated into Triton's core feature set.

It appears to be the case that the current distribution of expert activations is noisy and not conducive to request-level batching. However, the router making routing decisions is a learned neural network so it might be interesting to consider constraining the router at training time to generate specific activation patterns that would be more conducive to batching.

In conclusion, this has been an interesting exploration under time-constrained circumstances. The current explosion of large language models and their rise to the mainstream requires advances on the inference side. This project proposed a fetching engine to make the deployment of sparse LLMs possible on low-resource machines and analyzed the performance penalty incurred for reducing the available memory.

Bibliography

[1] Protocol buffers.

[2] Pytorch/serve: Serve, optimize and scale pytorch models in production.

[3] The state of online retail performance, Apr 2017.

[4] Zaid Alyafeai, Maged Saeed AlShaibani, and Irfan Ahmad. A survey on transfer learning in natural language processing. *CoRR*, abs/2007.04239, 2020.

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

[6] 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. *CoRR*, abs/2005.14165, 2020.

[7] Zixiang Chen, Yihe Deng, Yue Wu, Quanquan Gu, and Yuanzhi Li. Towards understanding mixture of experts in deep learning, 2022.

[8] Nvidia Corporation. Triton custom batching strategy.

[9] Nvidia Corporation. Triton dynamic batcher.

[10] Nvidia Corporation. Triton inference server.

[11] Nvidia Corporation. Triton python backend.

[12] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[13] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *CoRR*, abs/2101.03961, 2021.

[14] Github. Github copilot — your ai pair programmer, 2021.

[15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[16] Abhilash Jain. Finnish language modeling with deep transformer models. *arXiv*, 03 2020.

[17] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, Carol Willing, and Jupyter development team. Jupyter notebooks - a publishing format for reproducible computational workflows. In Fernando Loizides and Birgit Scmidt, editors, *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, pages 87–90, Netherlands, 2016. IOS Press.

[18] Saeed Masoudnia and Reza Ebrahimpour. Mixture of experts: a literature survey. *The Artificial Intelligence Review*, 42(2):275, 2014.

[19] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. Tensorflow-serving: Flexible, high-performance ML serving. *CoRR*, abs/1712.06139, 2017.

[20] OpenAI. Introducing chatgpt, 11 2022.

[21] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning Representations by Back-propagating Errors. *Nature*, 323(6088):533–536, 1986.

[22] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, Ryan Sepassi, and Blake A. Hechtman. Mesh-tensorflow: Deep learning for supercomputers. *CoRR*, abs/1811.02084, 2018.

[23] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc V. Le, Geoffrey E. Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *CoRR*, abs/1701.06538, 2017.

[24] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for deep learning in NLP. *CoRR*, abs/1906.02243, 2019.

[25] Laurens van der Maaten and Geoffrey Hinton. Viualizing data using t-sne. *Journal of Machine Learning Research*, 9:2579–2605, 11 2008.

[26] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.