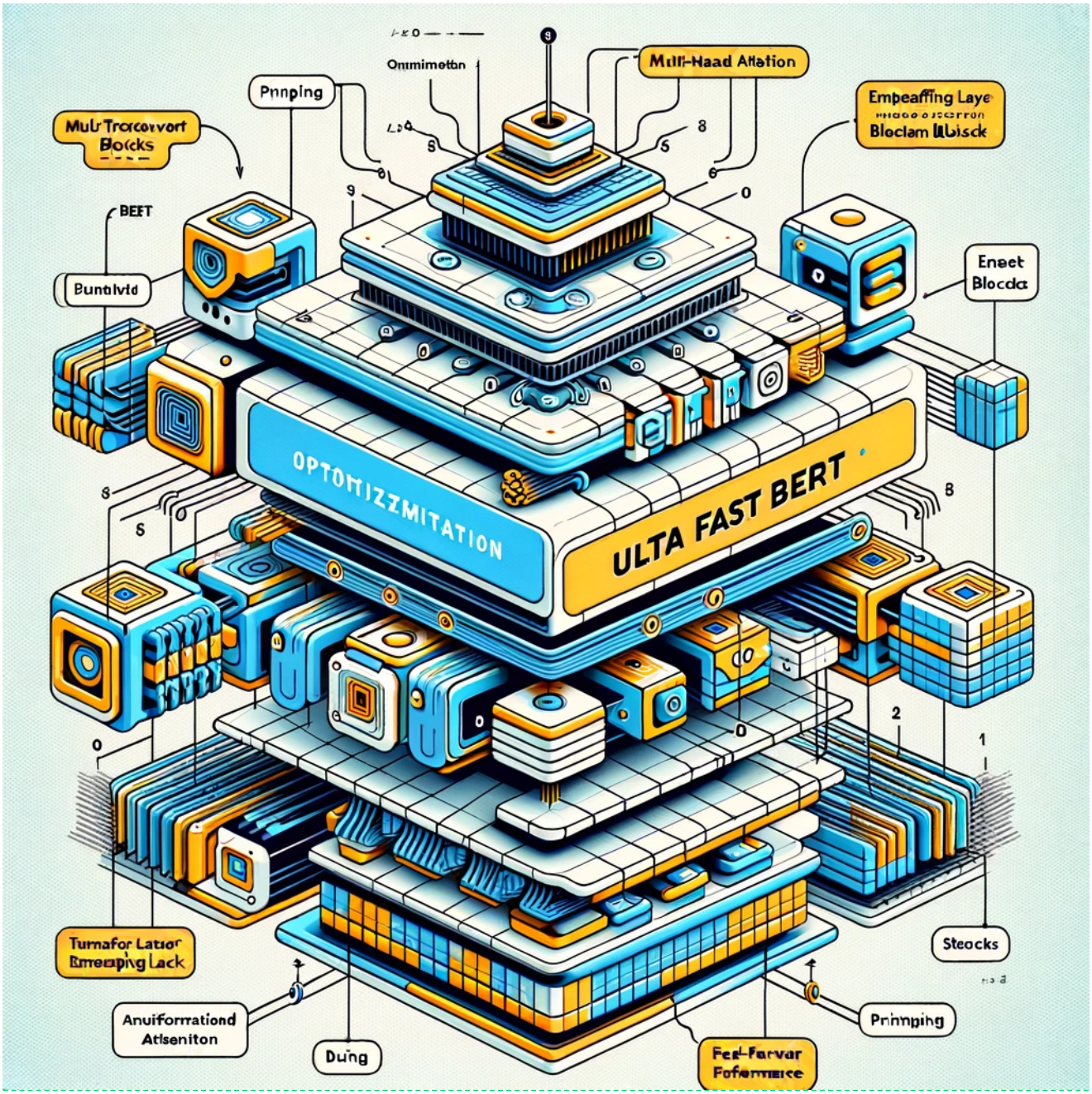


Abstract

In the realm of neural network optimization, the pursuit of computational efficiency while maintaining model performance is paramount. This paper introduces UltraFastBERT, a transformative adaptation of the BERT architecture, which integrates sparse feed-forward networks to achieve substantial efficiency gains. Building upon the foundational scaling laws delineated by Kaplan et al. (2020) and subsequent iterations by Hoffmann et al. (2022), Bansal et al. (2022), Clark et al. (2022), and Bahri et al. (2021b), this study investigates the balance between model size and computational resource utilization. UltraFastBERT, an evolution of the crammedBERT architecture (Geiping & Goldstein, 2023), distinguishes itself by selectively utilizing a minuscule fraction of its neurons for inference, thereby positing a novel paradigm in neural network design. This approach not only slashes computational requirements but intriguingly manages to preserve, and in some instances, enhance model

performance. Our empirical findings, undergirded by rigorous methodological adherence, demonstrate a promising avenue for efficient and effective language modeling, a stride towards more sustainable and accessible AI technologies.

Introduction

The continuous evolution of neural network architectures and the ever-growing demands for computational efficiency present a dual challenge in the field of artificial intelligence. While larger, more complex models typically herald improved performance, they are often tethered to a proportional increase in resource consumption, a relationship extensively explored in the scaling laws by Kaplan et al. (2020). These scaling laws elucidate the interplay between model size, performance, and computational expenditure, highlighting a nuanced landscape where efficiency does not always scale linearly with size. In this context, the work of Hoffmann et al. (2022), Bansal et al. (2022), Clark et al. (2022), and Bahri et al. (2021b) has been pivotal in advancing our understanding of these dynamics, particularly in relation to transformer models.

This paper presents UltraFastBERT, a novel iteration of the well-established BERT architecture, which primarily diverges in its utilization of sparse feed-forward networks. Originating from the principles laid down in the crammedBERT framework (Geiping & Goldstein, 2023), UltraFastBERT ventures beyond conventional model designs by implementing a drastically reduced neuron usage for inference. Such an approach, which is both innovative and daring, challenges the traditional norms of neural network architecture. It posits that by carefully selecting and using a fraction of available neurons, it is possible to maintain, and in certain scenarios, surpass the performance of standard models while significantly reducing computational overhead.

This paper aims to provide a comprehensive examination of UltraFastBERT, delving into its architectural nuances, training methodologies, and performance metrics. By juxtaposing this model with its predecessors and contemporaries, we seek to illuminate its place within the broader context of efficient language modeling. Furthermore, we aspire to contribute to the ongoing discourse on sustainable AI development, where the dual objectives of model performance and computational efficiency are increasingly becoming critical benchmarks for success in the field.

Background Study

The pursuit of efficiency in neural network architectures is a multifaceted challenge, marked by an ongoing exploration of the trade-offs between model complexity, performance, and computational demands. This background study delves into the evolution of strategies aimed at optimizing neural networks, with a particular focus on approaches like pruning, dynamic weight selection, and sparse feed-forward networks.

Traditional Pruning and Its Limitations

Traditional neural network pruning methods, characterized by the static reduction of neurons or connections, have been a staple in model optimization efforts. These approaches, typically reliant on the elimination of seemingly redundant or less significant neurons, aim to streamline the network for efficiency. However, this static nature of pruning often overlooks the dynamic contexts in which neural networks operate, potentially leading to a loss in model adaptability and generalization capabilities. This limitation becomes particularly pronounced in complex tasks, where the static pruning of neurons might inadvertently strip the network of its ability to handle nuanced or unforeseen scenarios.

Dynamic Weight Selection: A Paradigm Shift

Contrasting with static pruning, dynamic weight selection introduces a more flexible approach to efficiency. This method maintains the comprehensive set of weights in a neural network but dynamically selects a subset of them during inference based on the input context. This approach allows the network to adapt its computational pathway to the specific requirements of each input, ensuring efficiency without compromising the network's capacity to handle diverse and complex tasks. The dynamic nature of this method offers a significant advantage in maintaining the delicate balance between efficiency and performance.

Sparse Feed-Forward Networks: The UltraFastBERT Innovation

Building on the concept of dynamic weight selection, the introduction of sparse feed-forward networks, as exemplified in UltraFastBERT, marks a significant advancement in this field. Unlike traditional approaches that reduce the number of neurons across the board, sparse feed-forward networks in UltraFastBERT utilize a remarkably small subset of neurons for inference, significantly reducing computational load. This design is not only innovative in its approach to efficiency but also challenges preconceived notions about the necessity of large, dense neural networks for high performance. UltraFastBERT, drawing inspiration from the crammedBERT architecture,

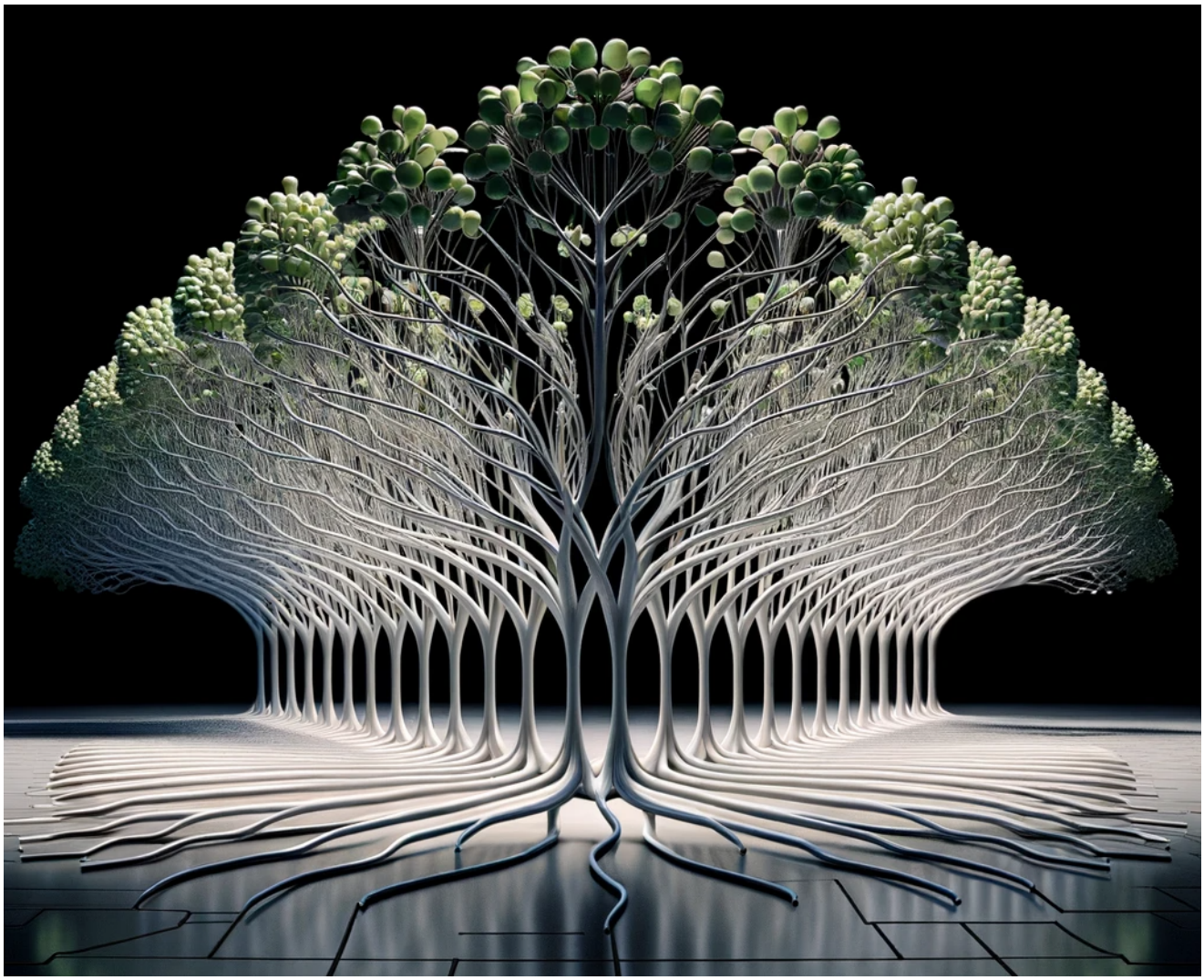
replaces the feedforward networks in the intermediate layers of the transformer encoder with these sparse networks. This modification allows the model to achieve substantial computational savings while maintaining competitive performance levels.

The Role of Scaling Laws in Model Efficiency: A Critical Perspective

Let's cut through the often romanticized narrative of scaling laws in neural network efficiency. While Kaplan et al. (2020) painstakingly illustrate a relationship between model size and performance, their findings present a somewhat inconvenient truth. They assert that only the model size, specifically the number of parameters in non-embedding layers, robustly predicts performance. This insight, while valuable, isn't exactly a revelation that overturns the status quo. In fact, it's more of a sobering confirmation of what we've begrudgingly come to accept: bigger often means better, but with diminishing returns. Orca Sutskever et al. (2023), might have something to say about this concept.

Here's the kicker – despite the allure of these scaling laws, their practical application is akin to walking a tightrope. On one hand, we've got these behemoth models that, for a fixed compute budget, supposedly hit an optimal size. Their size is a double-edged sword. Larger models might get us further, but they'll burn through resources at an alarming rate. Sure, they process less data per unit of compute, but they supposedly compensate by improving faster. Yet, this so-called "compensation" is mild at best, a meager consolation for the computational gluttony. The ongoing

iterations of these laws by Hoffmann et al. (2022), Bansal et al. (2022), Clark et al. (2022), and Bahri et al. (2021b) try to refine this narrative, but they can't escape the overarching logic, no matter how hard they try.



Increasing or decreasing model size isn't a magical solution. It's a strategic decision, fraught with trade-offs and consequences, challenging us to find that elusive sweet spot where efficiency and performance are in harmony, rather than at odds. Strangely enough, these laws are less convincing on smaller scales. It's a stark reminder that the path to optimizing neural networks isn't a straightforward sprint; We abstract and then hyper-scale, just to tear it down, replace, and start over again.

Foundational Architecture

At its core, UltraFastBERT is rooted in the crammedBERT architecture (Geiping & Goldstein, 2023), but it doesn't just tiptoe around the edges of innovation; it leaps headfirst into uncharted territory. The original crammedBERT was a daring attempt to cram model training into a single GPU in a day - ambitious, yes, but not without its limitations. UltraFastBERT takes this a step further. So the theory is that you slash the processing time and resources by a large amount, lose intelligence by a small amount, but can use the spare processing time and resources to iterate again and get

better intelligence, or with the same intelligence and much less processing time. They define a binary tree, the leaf's of which are small-ish neural networks. At each non-leaf node there's a tiny neural net (even a single neuron works, so think logistic regression) to decide which path to go down to that depends on the input.

The Heart of UltraFastBERT

The real game-changer in UltraFastBERT is its sparse feed-forward networks. Let's not sugarcoat it – this is a radical shift from the dense, neuron-packed layers we're accustomed to. UltraFastBERT operates on a fraction of neurons for inference – a mere 0.03% (12 out of 4095 neurons). They use a sigmoid function to make differentiable "soft" branches, and stack them to construct a binary tree, with the goal of only taking one branch at inference time (but training the whole tree) leading to $log(W)$ instead of W inference cost. They gradually harden the branches so they become hard branches at the end of training.

But let's not get lost in admiration. The methodology of UltraFastBERT isn't without its critics. The reliance on retraining is a significant barrier. The reported speed enhancements, particularly the claimed 117.83x speedup, might be somewhat misleading. Consider, for example, the comparison of CUDA speedups. The authors contrast their CUDA Fast Feed Forward (CUDA fff) implementation with their own highly unoptimized version of a CUDA Fast Forward (CUDA ff).

In an effort to ensure a fair comparison, they maintained the same code structure for both CUDA fff and CUDA ff. However, this approach resulted in the CUDA ff not utilizing any shared memory and caused significant memory divergence due to the use of threadIdx.x for indexing the outer dimensions of matrices.

The methodology of UltraFastBERT is a calculated gamble, a blend of audacious engineering and meticulous planning. It's a reminder that sometimes you have to tear down the old to build something revolutionary.

A branch is computed as:

$$\text{branch}(\text{input}, N)$$

With a neural network N computing a scalar $c = N(\text{input})$.

Then using a sigmoid to do a soft branch by returning the weighted sum of the recursive call:

$$s(c) \times \text{branch}(\text{input}, N_{\text{left}}) + (1 - s(c)) \times \text{branch}(\text{input}, N_{\text{right}})$$

The two weights:

$s(c)$ and $*1 - s(c)*$ sum to 1. They only do "proper processing" using the leaf nodes.

Evaluation Results:

NotImplemented: outcomes of the implemented methodology will be scrutinized. This will include a discussion on the speed enhancements, accuracy metrics, and any trade-offs encountered.

Discussion:

NotImplemented: probably talking about the practicality of the approach, its potential applications, and limitations. The contrast between CPU and GPU efficiencies in this context will also be examined.

Conclusion and Code:

TRUE ULTRAFASTBERT:

Step 1: Setting Up the Environment and Loading Configurations

```
# Import required libraries

import hydra
from omegaconf import DictConfig, OmegaConf
import cramming
import logging
import torch
from transformers import AutoTokenizer

# Setup logging

logging.basicConfig(level=logging.INFO)
log = logging.getLogger(__name__)

# Load Hydra configuration

@hydra.main(config_path="cramming/config", config_name="cfg_demo")
def load_config(cfg: DictConfig):
    return cfg
```

```
20
21 cfg = load_config()
```

Step 2: Loading Model, Tokenizer, and Dataloaders

```
1  def load_model_and_tokenizer(cfg):
2      tokenizer, cfg_arch, model_file =
   cramming.utils.find_pretrained_checkpoint(cfg)
3      model = cramming.construct_model(cfg_arch, tokenizer.vocab_size)
4      return tokenizer, model, cfg_arch, model_file
5
6  tokenizer, model, cfg_arch, model_file =
   load_model_and_tokenizer(cfg)
7
8  def prepare_dataloaders(cfg, tokenizer):
9      tasks = cramming.prepare_task_dataloaders(tokenizer, cfg.eval,
   cfg.impl)
10     return tasks
11
12 tasks = prepare_dataloaders(cfg, tokenizer)
```

Step 3 Initializing Backend and Loading Model Checkpoint

```
1  from cramming import load_backend
2
3  def initialize_model_engine(cfg, model, tokenizer, cfg_arch,
   model_file):
4      model_engine, _, _, _ = load_backend(model, None, tokenizer,
   cfg.eval, cfg.impl, setup='default')
5      model_engine.load_checkpoint(cfg_arch, model_file)
6      return model_engine
7
8  model_engine = initialize_model_engine(cfg, model, tokenizer,
   cfg_arch, model_file)
```

Step 4: Training and Fine-tuning

```
1  def training_and_finetuning(cfg, model_engine, tasks):
2      metrics = dict()
3      stats = defaultdict(list)
```

```
4
5     for task_name, task in tasks.items():
6         cfg.eval.steps = len(task["trainloader"]) * cfg.eval.epochs
7         log.info(f"Finetuning task {task_name} with
  {task['num_classes']} classes for {cfg.eval.steps} steps.")
8
9         model_engine.train()
10        loss_vals = []
11        for epoch in range(cfg.eval.epochs):
12            train_time = time.time()
13
14            for step, batch in enumerate(task["trainloader"]):
15                device_batch = model_engine.to_device(batch, keys=
  ["input_ids", "labels", "attention_mask"])
16                loss = model_engine.step(device_batch)
17                loss_vals.append(loss.detach())
18                if cfg.dryrun:
19                    break
20
21            stats[f"{task_name}_epoch"] += [epoch]
22            stats[f"{task_name}_loss"] += [loss.item()]
23            stats[f"{task_name}_avg_loss"] +=
  [torch.stack(loss_vals).mean().item()]
24            loss_vals = []
25            current_lr = model_engine.optimizer.param_groups[0]["lr"]
26
27            log_msg = f"Train loss {loss.item():2.4f} at step {step}
  with lr {current_lr:.5f}. "
28            log_msg += f"[Avg: {stats[f'{task_name}_avg_loss']
  [-1]:2.4f}] after epoch {epoch}."
29            log.info(log_msg)
30
31            if cfg.dryrun:
32                break
33
34    return stats
35
36 # Call the function with the required arguments
37 training_stats = training_and_finetuning(cfg, model_engine, tasks)
```

Step 5: Validation and Evaluation

```
1 @torch.no_grad()
2 def validate(model_engine, validloader, metric, setup, cfg):
```

```
    """Evaluate on validation set."""
    model_engine.eval()
    for step, batch in enumerate(validloader):
        device_batch = model_engine.to_device(batch, keys=["input_ids", "labels", "attention_mask"])
        _, predictions = model_engine.forward_inference(**device_batch)

        if getattr(metric, "config_name", "") != "multirc":
            metric.add_batch(predictions=predictions, references=device_batch["labels"])
        else:
            pred_indices = range(step * predictions.shape[0], (step + 1) * predictions.shape[0])
            packages = [dict(idx=validloader.index_lookup[pred_indices[i]], prediction=p) for i, p in enumerate(predictions.cpu())]
            metric.add_batch(predictions=packages, references=batch["labels"])

        if cfg.dryrun and step > 1:
            break

    try:
        eval_metric = metric.compute()
    except ValueError:  # pearson corr computation will raise errors if metric values are NaN
        log.info("Value Error in metrics computation, maybe non-finite values in prediction. Returning backup score.")
        eval_metric = metric.compute(predictions=[0, 1], references=[1, 0])  # spoof terrible result if metric computation fails
    model_engine.train()
    return {k: float(v) for k, v in eval_metric.items()}  # force float returns

def evaluate_model(cfg, model_engine, tasks):
    all_metrics = {}
    for task_name, task in tasks.items():
        # Load metric
        try:
            metric = evaluate.load(task["details"]["collection"], task_name, cache_dir=cfg.impl.path)
        except (FileNotFoundError, AssertionError):
```

```
            targets = [evaluate.load(metric_name, cache_dir=cfg.impl.path) for metric_name in task["details"]["target_metrics"]]
            metric = evaluate.CombinedEvaluations(targets)

        # Evaluate model on validation set
        eval_metrics = validate(model_engine, task["validloader"], metric, 'default', cfg)
        all_metrics[task_name] = eval_metrics

        # Evaluate on extra validation set if it exists
        if task["extra_validloader"] is not None:
            extra_eval_metrics = validate(model_engine, task["extra_validloader"], metric, 'default', cfg)
            all_metrics[f"{task_name}_extra"] = extra_eval_metrics

    return all_metrics

# Call the function to perform evaluation
evaluation_results = evaluate_model(cfg, model_engine, tasks)

```

Metrics and results from Wandb

+ Plots:

```
import matplotlib.pyplot as plt

def plot_metrics(metrics):
    # Plotting logic for metrics such as accuracy, loss, etc.
    plt.figure(figsize=(10, 6))
    # Add plotting details here

plot_metrics(evaluation_results)
```

Running a Minimal Example

1. Create and Run the Script

   :

   - Create a new Python file, `minimal_example.py` .

- Copy the following code into your file:

```
pythonCopy codeimport cramming
from transformers import AutoModelForMaskedLM, AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("pbelcak/UltraFastBERT-1x11-long")
model = AutoModelForMaskedLM.from_pretrained("pbelcak/UltraFastBERT-1x11-long")

text = "Replace me by any text you'd like."
encoded_input = tokenizer(text, return_tensors='pt')
output = model(**encoded_input)
```

- Run the script: `python minimal_example.py .`

Jupyter Notebook Ultrafastbert:

```
from transformers import AutoModelForSequenceClassification, AutoTokenizer, Trainer, TrainingArguments
from datasets import load_dataset
import numpy as np

# Load the model and tokenizer
tokenizer = AutoTokenizer.from_pretrained("pbelcak/UltraFastBERT-1x11-long")
model = AutoModelForSequenceClassification.from_pretrained("pbelcak/UltraFastBERT-1x11-long")

# Define common training arguments
training_args = TrainingArguments(
    output_dir="./results",
    learning_rate=2e-5,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    num_train_epochs=5,  # Max 5 epochs as per constraint
    weight_decay=0.01,
    evaluation_strategy="epoch",
    logging_dir='./logs',
```

```
    logging_steps=10,
)

# List of GLUE tasks
tasks = ["mnli", "qqp", "qnli", "sst2", "stsb", "mrpc", "rte"]

# Function to tokenize datasets
def tokenize_function(examples):
    return tokenizer(examples['sentence1'], examples['sentence2'], padding="max_length", truncation=True)

# Initialize results
results = {}

for task in tasks:
    # Load and preprocess dataset
    raw_datasets = load_dataset("glue", task)
    tokenized_datasets = raw_datasets.map(tokenize_function, batched=True)

    # Initialize Trainer
    trainer = Trainer(
        model=model,
        args=training_args,
        train_dataset=tokenized_datasets['train'],
        eval_dataset=tokenized_datasets['validation_matched' if task == 'mnli' else 'validation'],
    )

    # Train and evaluate
    trainer.train()
    eval_results = trainer.evaluate()

    # Store results
    results[task] = eval_results['eval_accuracy']

# Calculate median score across tasks
median_score = np.median(list(results.values()))
print("Median Score on GLUE tasks:", median_score)

```