Exploring Efficient Learning of Small BERT Networks with LoRA and DoRA

Stanford CS224N Default Project

Aditri Bhagirath
Department of Computer Science
Stanford University
`aditri@stanford.edu`

Moritz Bolling
Department of Statistics
Stanford University
`mbolling@stanford.edu`

Daniel Frees
Department of Statistics
Stanford University
`dfrees@stanford.edu`

Abstract

While Large Language Models (LLMs) have revolutionized artificial intelligence, fine-tuning LLMs is extraordinarily computationally expensive, preventing smaller businesses and research teams with limited GPU resources from engaging with new research. Hu et al. (2021) and Liu et al. (2024) introduce Low-Rank Adaptation (LoRA) and Weight-Decomposed Low-Rank Adaptation (DoRA) as highly efficient and performant solutions to the computational challenges of LLM fine-tuning, demonstrating huge speedups and memory usage savings for models such as GPT-3 and RoBERTa. We seek to expand upon the original LoRA and DoRA papers by benchmarking efficiency and performance of LoRA and DoRA when applied to a much smaller scale of language model: our case study here is the compact minBERT model. Our findings reveal that optimal custom configurations of LoRA and DoRA, coupled with Automatic Mixed Precision (AMP), significantly enhance training efficiency without compromising performance. Furthermore, while the parameterization of minBERT is significantly smaller than GPT-3, our results validate the observation that gradient updates to language models are inherently low-rank even in small model space, observing that rank 1 decompositions yield negligible performance deficits. Furthermore, aided by our highly efficient minBERT implementation, we investigate numerous architectures, custom loss functions, and hyperparameters to ultimately train an optimal ensembled multitask minBERT model to simultaneously perform sentiment analysis, paraphrase detection, and similarity scoring.

1 Key Information to include

- Mentor: Kaylee Burns; External Collaborators (if you have any): N/A; Sharing project: No
- Team Contributions:
  Aditri: Base minBERT implementation (excluding Adam), baseline models for all subtasks with enhanced features, data preprocessing, contrastive experiments.
  Moritz: Debugged Adam, implement LoRA and DoRA, benchmark/grid search LoRA and DoRA, ablation studies
  Daniel: Implemented Adam, multi-task training regimes, Clf layers, AMP, TensorBoard, LR expts, EMA, pearson & hybrid losses, architecture expts, ensembling, nltk cleaning, iterative unfreezing.

2 Introduction

In recent years, large language models have increased massively in terms of both performance and size. Current best estimations of large language model performance scaling laws, such as Kaplan et al. (2020) suggest that this trend will continue for the foreseeable future, with model performance scaling consistently as a power law relative to model size, dataset size, and the amount of compute used for training. As a consequence of the massive scale of modern large language models, the accessibility of language model fine-tuning has significantly decreased.

The traditional approach for fine-tuning involves updating all model parameters, which can quickly become infeasible as the number of model parameters grows. However, this methodology came into question when Li et al. (2018) uncovered the promising finding that many models could be massively compressed into smaller parameter subspaces without compromising performance. Aghajanyan et al. (2020) extended this finding to demonstrate that fine-tuning updates to pre-trained large language models had low intrinsic rank. Subsequently, Hu et al. (2021) and Liu et al. (2024) proposed LoRA and DoRA, respectively, as solutions towards parameter efficient fine-tuning of pre-trained large language models, achieving a $10,000x$ reduction in trainable parameters and $3x$ reduction on GPU memory requirement when fine-tuning GPT-3 175B, without sacrificing performance.

To the best of our knowledge, no prior research has benchmarked efficiency gains or implementation best practices when leveraging LoRA and DoRA for the fine-tuning of small BERT models. As an extension to the work of Hu et al. (2021) and Liu et al. (2024), we seek to understand the intrinsic rank of fine-tuning updates, benchmark expected efficiency gains and performance reductions, and propose optimal selection of layers for low-rank adaptation of small BERT models. Our case study is our $\approx 226M$ (Appendix O) parameter minBERT model (built off of Stanford's CS224N minBERT: Stanford University (2024)). To achieve this, we design custom implementations of LoRA and DoRA for minBERT and quantify efficiency gains and performance reduction for a multi-task minBERT model. Our model is designed to perform sentiment analysis, paraphrase detection, and semantic textual similarity scoring simultaneously. As a secondary goal, empowered by our efficient fine-tuning capabilities, we conduct extensive experiments towards deriving an optimal minBERT multi-task model.

3 Related Work

3.1 Efficient Fine-Tuning

Li et al. (2018) uncovered the fact that many models can be re-parameterized in much lower dimensional space without losing performance. Building upon this, Aghajanyan et al. (2020) set out to investigate the intrinsic rank of fine-tuning updates for pre-trained large language models, finding that Li et al.'s results held true in the language modeling domain. Subsequently, Hu et al. (2021) and Liu et al. (2024) proposed parameter efficient fine-tuning techniques to take advantage of the low intrinsic rank in large language model fine-tuning updates. The novel contribution of Hu et al. (2021), Low-Rank Adaptation for Large Language Models (LoRA), has gained massive popularity in both academia and industry, enabling research teams with limited resources to explore fine-tuning use cases that might not otherwise be approachable. Liu et al. (2024) was released earlier this year and suggests that LoRA tine-tuned model performance might be massively improved by a simple decomposition of weight updates into separate magnitude and direction components.

LoRA's original benchmark results demonstrated a striking reduction in VRAM usag ——for instance, it reduced memory requirements from 1.2TB to 350GB during training on GPT-3 175B Hu et al. (2021). Hu et al. (2021) also benchmarked LoRA with impressive results on GPT-2, RoBERTa, and DeBERTa and evaluated fine-tuned models across a variety of benchmarks, including GLUE, WikiSQL, SAMSum, and the E2E NLG Challenge. RoBERTa and DeBERTa were fine-tuned with LoRA and performance tested against the GLUE benchmark, containing several subtasks such as MNLI (Multi-Genre Natural Language Inference) [Williams et al. (2018)], SST-2 (Stanford Sentiment Treebank) [Socher et al. (2013)], MRPC (Microsoft Research Paraphrase Corpus) [Dolan and Brockett (2005),] and QQP (Quora Question Pairs) [Quora (2017)]. On these subtasks, LoRA achieved scores close to full fine-tuning, and even outperformed on some tasks. The LoRA adaptations of GPT-2 and GPT-3, tested on WikiSQL (natural language to SQL conversion) and E2E NLG (generating descriptions of restaurants from structured data), respectively, performed better than full fine-tuning while drastically reducing the number of parameters by up to 10,000 times Hu et al. (2021).

3.2 Performance Optimizations

Several additional works guided our multi-task performance optimization experiments. The Adamax optimizer was proposed alongside the Adam optimizer by Kingma and Ba (2015), where moment updates rely on the infinity norm, $L^p, p \to \infty$, rather than the L^2 norm. Given that we investigate fine-tuning of a small BERT model, our gradient updates involved high (768) dimensional embedding updates and potentially low intrinsic-rank attention layer updates. As such, the infinity norm may be a suitable measurement of gradient magnitude since our gradient updates are likely sparse.

Recent works such as Howard and Ruder (2018) have demonstrated the efficacy of iterative backbone unfreezing throughout fine-tuning of pre-trained large language models, whereby more and more of the large language model backbone is unfrozen as training progresses. Iterative unfreezing may yield superior generalization model performance and increase robustness of transfer learning to the phenomenon of catastrophic forgetting Kirkpatrick et al. (2017).

As recently confirmed by Mohammed and Kora (2023), ensembling methods provide consistent performance gains and generally yield flatter, more generalizable optima. This fundamental principle applies across traditional regression/classification methods, tree-based methods, and deep learning, though in deep learning there is still some debate as to whether ensembling provides gains that can't be equivalently accomplished by larger deep neural network architectures.

A portion of our experiments are dedicated to elucidating whether Adamax, iterative unfreezing, and ensembling are suitable approaches to increase the multi-task performance of small fine-tuned BERT models.

4 Approach

We implement all of the following model architectures and fine-tuning approaches ourselves using PyTorch (Paszke et al. (2019)).

4.1 Multi-task Baseline

As a baseline, we implemented a minBERT model as described in Stanford's CS224N default project handout (Stanford University (2024)), with $1D$ convolutions and 1-layer fully-connected-network (FCN) classifier layers for each of our three tasks (sentiment, paraphrase detection, similarity scoring). Raw BERT embeddings from the minBERT backbone were fed as input into the SST classifier, while absolute BERT embedding differences between sentence pairs were computed and fed as input into the Quora and STS classifiers. Baseline training utilized a training routine where SST, Quora, and STS were each trained in turn, once per epoch, using Cross-Entropy (CE) loss for SST sentiment classification, Binary Cross-Entropy (BCE) loss for Quora paraphrase binary classification, and mean squared error (MSE) loss for similarity score prediction.

4.2 Optimized Multi-task Model

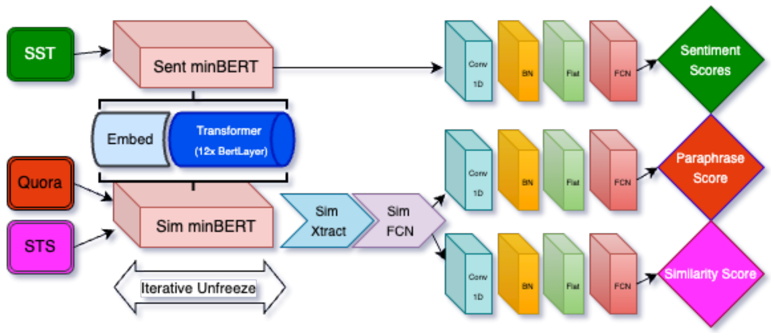

Figure 1: Architecture of our optimized minBERT model.

Following the results of our performance optimization experiments (subsection 5.3), our optimized model architecture was constructed as depicted in Figure 1. Our BERT backbone consists of two identical but separate "Sent" and "Sim" pre-trained minBERT instances. A similarity feature extraction layer extracts features from sentence pair embeddings, concatenating the raw embeddings, e_1, e_2 with their elementwise product, $e_1 \odot e_2$, cosine similarity, $\frac{e_1 \cdot e_2}{\|e_1\|\|e_2\|}$, and absolute difference $\|e_1 - e_2\|$. All classifier layers consist of $1D$ convoltuions with kernel size 3 and 4 filters. BatchNorms are applied after convolutional and fully-connected layers, and we use ReLU non-linearity. Iterative unfreezing is applied to the backbone such that only a subset of parameters, $\theta + \omega$, are trainable at any given time. We denote classifier parameters as $\theta = \theta_{SST} + \theta_{Para} + \theta_{STS}$ and BERT backbone parameters as Ω, with trainable backbone parameters $\omega \subseteq \Omega$.

4.3 Multi-task Learning Problem Formulation

Based on the results of our performance optimization experiments around loss functions (Appendix F, we utilize Cross-Entropy loss to optimize for the SST sentiment task, Binary Cross-Entropy loss to optimize for the Quora paraphrase detection task, and a custom Pearson correlation loss to optimize for the STS similarity task. Thus we can consider our learning procedure to be an optimization problem where we seek to minimize the following composite loss

$$L(\theta) = L_{\text{CE}}(f_{\theta_{\text{SST}}+\Omega}, y_{\text{SST}}) + L_{\text{BCE}}(f_{\theta_{\text{Para}}+\Omega}, y_{\text{Para}}) + L_{\text{Pearson}}(f_{\theta_{\text{STS}}+\Omega}, y_{\text{STS}})$$

where $f_{\theta_{\text{SST}}+\Omega}, f_{\theta_{\text{Para}}+\Omega}, f_{\theta_{\text{STS}}+\Omega}$ represent the model logits for SST, Quora paraphrase, and STS tasks respectively. We note also that our optimization problem has increasing degrees of freedom at each epoch depending how many of the backbone parameters $\omega \subseteq \Omega$ are unfrozen and trainable.

4.4 Fine-Tuning Efficiency

4.4.1 Baselines

Our baseline reference point for memory usage, time and accuracy consisted of the training baseline described in subsection 4.1 with all layers unfrozen, and without any automatic mixed-precision (AMP) or efficient fine-tuning (LoRA and DoRA) applied. We derived a lower bound for accuracy and an upper bound for efficiency by comparing against fine-tuning of only the classifier layers.

4.4.2 AMP

With automatic mixed precision (AMP) training, we computed forward passes and loss during training in half-precision format (`float16`), then cast back to full `float32` to backpropagate gradients, using a gradient scaler to minimize loss of gradient information due to `float16` underflow. This can substantially improve training throughput by reducing memory usage (NVIDIA (2023)). For an overview of the mathematical principles behind AMP, see Appendix C.

4.4.3 LoRA

Various Python libraries, such as `peft` (Mangrulkar et al. (2022)), provide implementations of LoRA and DoRA for common model architectures. However, we built a custom implementation tailored to the minBERT model, granting us precise control over the layers subjected to LoRA modifications. LoRA involves freezing the pre-trained weights of a linear layer (W_0) and integrating trainable low-rank matrices B and A. This adaptation is represented by the equation:

$$W' = W_0 + BA$$

where $W_0 \in \mathbb{R}^{d \times k}$, $B \in \mathbb{R}^{d \times r}$, and $A \in \mathbb{R}^{r \times k}$ with $r \ll \min(d, k)$. This methodology reduces the number of parameters that need to be updated, thereby enhancing the efficiency of fine-tuning. While Hu et al. (2021) introduced an additional parameter α that scales the weight update by $\frac{\alpha}{r}$, they acknowledged that using the Adam optimizer makes tuning α nearly equivalent to adjusting the learning rate. Consequently, we maintained α at a constant value equal to r since we utilized Adam variants for efficiency experiments.

To implement LoRA, we froze all original parameters of each linear layer and initialize matrices A and B according to the layer's dimensions and the rank hyperparameter r, with A receiving Gaussian initialization and B being zero-initialized as described by Hu et al. (2021). During the forward pass, outputs from the original layer and the BA product were computed and summed.

4.4.4 DoRA

DoRA was proposed recently by Liu et al. (2024) Liu et al. (2024) as a more performant version of LoRA, with weight updates decomposed into directional and magnitude components, offering more precise control over updates. The weights are updated as follows:

$$W' = \frac{m}{\|W_0 + BA\|_c}(W_0 + BA)$$

where ΔV represents the incremental directional update, $m = \|W_0\|_c$, and $\| \cdot \|_c$ denotes the vector norm of a matrix across its columns. Note that m is a trainable magnitude parameter (m) initialized to the column norms of the original weights. While `peft` recently released initial implementations of DoRA, we implemented DoRA from scratch based loosely on the work of Taylor et al. (2023).

4.4.5 LoRA and DoRA Modes

Our custom implementations of LoRA and DoRA enabled four distinct matrix decomposition modes:

1. **attn-only**: This mode aligns with the recommendations from Hu et al. (2021), applying LoRA only to the attention mechanism's query, key, and value matrices while freezing all other parameters except classifier layers. Especially powerful for transformer architectures.
2. **attn**: Similar to the *attn-only* mode, this setting replaces the attention matrices (query, key, and value) but does not freeze the remaining layers.
3. **all-lin-only**: In addition to query, key, and value matrices, this mode extends LoRA to all linear layers within the minBERT model. All other layers except classifiers are frozen.
4. **all-lin**: Operates similarly to *all-lin-only* by applying LoRA across all linear layers, but does not freeze the other layers, permitting full model dynamics during training.

These modes offer varying degrees of parameter adaptability and training dynamics, enabling the tailored application of LoRA/DoRA depending on specific model requirements and targeted results. In terms of the number of parameters being updated, *attn-only* modifies the fewest, while *attn* adjusts the most. As a reference point, the entire minBERT model has 110M parameters. 21M of these are contained in linear layers within the attention blocks, a further 64M are contained in linear layers outside of the attention blocks.

4.5 Performance Optimization

Our efficient fine-tuning techniques enabled us to perform numerous experiments towards deriving the most performant multi-task minBERT model possible. We experimented with Linear, NonLinear, and Convolutional classifiers, cosine annealing and multiplicative learning rate scheduling, exponential moving average (EMA) weight updates, custom loss functions (Pearson loss, hybrid MSE/BCE loss, ordinal CE loss, SimCSE contrastive loss and others), various data cleaning pipelines, various similarity task feature extraction layers, iterative BERT backbone unfreezing, various optimizers (SGD, AdamW, RAdam, Adamax, SparseAdam) and task-specific optimizers, weight decay strength, initial learning rate selection, sub-task learning rate multipliers, multiple interleaving training schedules, and finally model ensembling and optimal ensemble mixtures.

5 Experiments

All experiments were conducted on NVIDIA A100 Tensor Core GPU with 40GB of VRAM. We conducted 3 main experiments to evaluate the impact of our efficiency improvements.

5.1 Data

We experiment with Stanford Sentiment Treebank (SST), Quora Question Pairs (QQP), and Semantic Textual Similarity Benchmark (STS) datasets for our three model sub-tasks. Sub-tasks and corresponding dataset details are included in Table 1.

Task	Input	Output
Sentiment Analysis (SST)	$11,855$ **sentences** from movie reviews.	A **sentiment score** ranging from 0 (Negative) to 4 (Positive)
Paraphrase Detection (QQP)	$404,298$ **question pairs** with binary labels indicating whether the questions are paraphrases.	A **binary label (Yes or No)**, indicating whether the question pairs are paraphrases.
Semantic Textual Similarity (STS)	$8,628$ **sentence pairs** with similarity scores.	A **semantic similarity score** on a scale from 0 (unrelated) to 5 (equivalent meaning)

Table 1: Overview of Data and Tasks

We perform some basic data preprocessing before passing inputs into our multi-task BERT model (see Appendix N for details).

5.2 Evaluation methods

We evaluate our models using multi-class acurracy for sentiment analysis (SST), binary accuracy for paraphrase detection (Quora) and Pearson correlation for semantic textual similarity (STS). For a total multi-task performance score (and for the purposes of checkpointing our best model weights during training), we adopt the same formula as the CS224N project competition leaderboard, normalizing the Pearson correlation to a $[0, 1]$ range and taking a simple average across the three tasks.

Additionally, to assess the efficiency of our models, we record the average time and average peak memory usage per training epoch for each task. Due to significant variability in computational demands across tasks——with the Quora paraphrase detection task requiring approximately 15 times more resources than the SST and STS tasks——these measures are averaged on a per-task basis.

5.3 Performance Experiments and Results

After initial experiments on optimizers (Appendix A), learning rate, weight decay, architecture (Appendix G), iterative unfreezing (Appendix B), EMA weight updates (Appendix E), custom loss functions (Appendix F), ensembling (Appendix D) and optimal ensemble mixtures of models with varied sub-task bias (Appendix L, our optimal model architecture was implemented as reported in subsection 4.2, and our best final model consists of 23 different models ensembled together, across our "G6 Duality of Man" (Appendix H), "G6 Yin Yang" (Appendix I), and "G6 Interleaver" (Appendix K model architectures/ learning routines. We dub this best ensemble "G6 Lisan Al Gaib", and report its results on the test dataset in Table 2. We also report our "G6 Oligarch v1" ensemble results.

Model	Sentiment	Paraphrase	Similarity	Overall Score
G6 Oligarch v1	**0.551**	0.881	0.853	0.786
G6 Lisan al Gaib	0.543	**0.889**	**0.861**	**0.788**

Table 2: Best Ensemble Model Test Results

5.4 Efficiency Experimental Details and Results

5.4.1 Large Grid Search

To evaluate the impact of LoRA and DoRA on our multitask classifier, we conducted a comprehensive grid search over the parameters summarized in Table subsection M.1.

The results of this grid search are presented in Figure 2, where values are averaged over all parameters and for each decomposition rank r. We observe a decrease in total accuracy with fewer fine-tunable parameters, as expected. Moreover, examining the results by batch size, memory consumption exhibits the anticipated trend: greater numbers of parameters correspond to higher memory usage, while fewer parameters result in reduced memory consumption. However, we also make two unexpected observations: 1) accuracy does not improve with increasing rank; and 2) DoRA does not outperform LoRA. These findings suggest that the intrinsic dimensionality of the weight updates for our small minBERT model are sufficiently low such that most information can be captured by a rank-one approximation. Higher ranks or distinctions between directionality and magnitude may capture only minimal additional nuances, overshadowed by the variability introduced. This hypothesis was substantiated through a comprehensive series of experiments comparing LoRA and DoRA across various ranks, which demonstrated that while memory usage and computational time escalate with rank, accuracy does not. Detailed results and analysis are provided in subsection M.2.

5.4.2 Ablation Study

To gauge the memory, time, and performance impact of various combinations of LoRA, DoRA, and AMP, we ran an ablation experiment, collecting metrics across all configurations. The hyperparameters for the experiment setup and the detailed results can be found in Figure M.3. In summary, the key findings are that AMP led to a reduction of memory usage by 26%, combining it with LoRA *attn-only* brought this to a 50% reduction, while using LoRA *attn* brought it to 28%. We noticed that in one case, using AMP caused the accuracy to drop significantly. This is likely due to our baseline efficiency experiment model being unstable (this model did not yet use Adamax and multiple minBERT backbones). That said, even though we apply a gradient scaler to avoid underflow, AMP can increase model instability in that we lose some precision in our gradient update space. In general AMP left accuracy unaffected. AMP yielded significant average time speedups of 60%, whereas LoRA and DoRA caused negligible time improvements. These results are as expected as LoRA and DoRA's main advantage is the memory savings they bring.

5.4.3 Applying Improvements to Final Model

Finally, we evaluated the performance of our best (non-ensemble) model architecture (subsection 4.2) with and without efficiency optimizations, employing AMP with either LoRA or DoRA in the *all-lin*

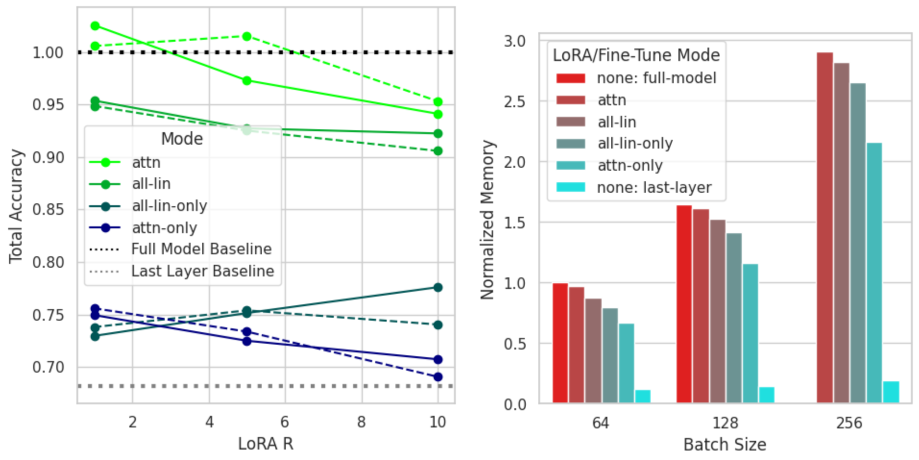

Figure 2: Left: LoRA and DoRA Accuracy per Mode across Rank. Dashed lines represent DoRA, solid lines LoRA. Normalized to Full Model baseline. Right: Normalized Peak Memory Consumption compared to Batch Size.

mode, across three seeds and seven epochs of training. The configurations are detailed in Table 7 and the accuracy results in Figure 3. The LoRA + AMP combination slightly outperformed the baseline, while DoRA + AMP marginally underperformed, with variances within a 1% range suggesting no statistical significance. Despite these minor performance differences, LoRA + AMP was more resource-efficient, showing a 1% improvement in performance, 33% faster execution, and 12% less memory usage than the baseline. Conversely, DoRA + AMP, despite a slight performance dip, was 24% faster and used 7% less memory. The memory savings were somewhat muted due to the iterative unfreezing approach and averaging our measurements over all epochs.

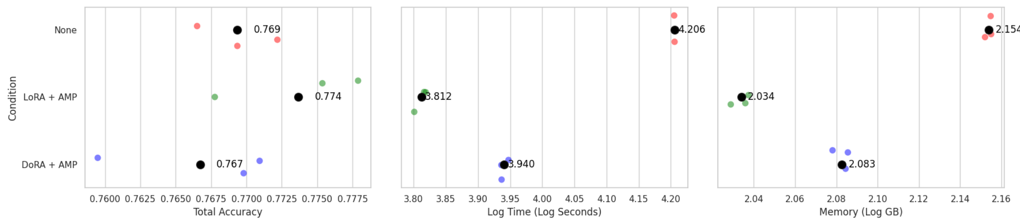

Figure 3: Comparison of Total Accuracy, Average Log Time and Average Log Memory across all tasks. The black dots represent the average over all 3 runs.

6 Analysis

Our findings suggest that with optimal hyperparameters, the integration of LoRA and DoRA with AMP can significantly enhance memory and time efficiency with minimal impact on model accuracy. We observed memory reductions ranging from 7% to 50% and time efficiency improvements up to 60%. Notably, the effect on accuracy does not correlate with LoRA rank but is highly dependent on its operational mode, with observed variations from a decrease of 24% to an increase of 3%. Low-rank adaptations, due to their reduced memory demands and relatively modest performance degradation, are particularly apt for scenarios requiring frequent model updates, such as real-time language processing or environments constrained by hardware capabilities. Furthermore, our study underscores the necessity for a nuanced understanding and thorough evaluation of potential failure cases induced by these adaptations, particularly in high-stakes scenarios. Before deployment, it is crucial to assess whether these adaptations introduce undesirable biases or failure patterns. Additionally, we found that STS tasks suffer an average performance decrease of 4% more compared to sentiment classification tasks, suggesting task-specific changes in the intrinsic dimension of the weight updates.

For sentiment analysis, outputs from LoRA and DoRA closely matched those of our best model. In scenarios where DoRA predictions were incorrect, they deviated from our optimal model's predictions by only one point in 94% of the cases. This variance is considered acceptable given the subjective nature of sentiment analysis. Examples where DoRA's sentiment predictions significantly diverged from our best model are presented in Table Table 3, typically involving sentences with mixed emotional content, which seem to have posed challenges for DoRA in capturing nuanced sentiment.

Table 3: Example Discrepancies in Sentiment Score Prediction: DoRA vs. Our Best Model

Sentence	**True Sentiment**	**DoRA Score**	**Best Model Score**
"It's one pussy-ass world when even killer-thrillers have no edge."	0	3	1
"It's like watching a nightmare made flesh."	0	3	1
"Harland Williams is so funny in drag he should have his own TV series."	2	3	1

In terms of paraphrase detection, our model attained a peak accuracy of 0.888, with the DoRA version achieving a comparable 0.858. The performance of DoRA on the STS task was also close to that on sentiment classification, mostly differing by minor points.

Regarding our secondary goal of deriving an optimal minBERT-based model for our multi-task reasoning benchmarks, we found that using `Adamax` as an optimizer (Appendix A), strong multiplicative learning rate decay ($0.5^{\text{epoch}} \times LR$), separate Sentiment and Similarity minBERT backbones (Appendix H), shared Quora and STS non-linear feature extraction layer, $1D$ convolutional classifiers, Pearson correlation loss (Appendix F), and ensembling of as many models as possible (subsection D.1) were critical to maximizing performance.

7 Conclusion

We extend the work of Hu et al. (2021) and Liu et al. (2024) by customizing LoRA and DoRA for minBERT, and benchmarking efficiency and accuracy effects of low-rank adapation on small BERT models. Our results confirm that even for small BERT models the intrinsic dimension of the weight updates, ΔW, is 1. Increasing the rank beyond this point does not yield significant performance gains. The fact that `Adamax` proved to be our best optimizer may be further evidence that the intrinsic rank of our fine-tuning updates was very low, given that `Adamax` is well-suited to sparse gradient updates.

We find that for our small BERT model, LoRA provides sufficient granularity for performant fine-tuning, with LoRA fine-tuning exceeding vanilla training performance. Our DoRA fine-tuning approach achieves slightly lower performance, suggesting that we do not benefit from separately decomposing magnitude and direction for our weight updates. Moreover, while the efficiency gains demonstrated here on minBERT are more modest compared to those observed in larger models such as GPT-3, our findings endorse the adoption of AMP combined with LoRA, applied to all linear transformer layers, even within smaller language model frameworks. These efficiency optimizations facilitate broader architectural explorations and the utilization of computationally expensive techniques like ensembling, enhancing the model's performance and feasibility for extensive deployment. While we make qualitative observations about model errors when using parameter efficient fine-tuning versus full fine-tuning (see section 6), further research is necessary to elucidate the exact relationship between low-rank weight updates to small BERT models and the specific types of errors made by these models on downstream tasks.

8 Ethics Statement

This work analyzes minBERT, a compressed version of BERT. As a result of its decreased parameterization, our minBERT model may not capture the full richness and complexity of the data it was trained on. For instance, minBERT may not capture the appropriate use of terms based on social context, varied dialects or regional differences, and may generate responses that are too reductive based on general patterns observed in the data. To mitigate this, we propose conducting comprehensive bias audits [dwt (2024)] to identify any problematic patterns in task predictions, and involving community stakeholders in an iterative model refinement process prior to deployment in any high-risk use cases, such as chat applications, social media, loan approval, etc. Incorporating feedback from diverse user groups and integrating domain-expert rules can help address specific biases and ensure more equitable model performance.

Using compression techniques such as LoRA and DoRA may also result in a loss of interpretability of our model. By introducing low-rank matrices which interact with existing high-dimensional parameter spaces, we further obfuscate the model's decision-making techniques. While deep-learning models are already challenging to interpret, techniques such as attention heat-mapping can be used to gain qualitative insight into model predictions. No studies to date have specifically investigated the impact of low-rank fine-tuning on interpretive visualizations, so it is entirely possible that we lose attention granularity. To combat this loss in interpretabality, future research should explore designing visual tools to understand how adjusting intermediate matrices influences model outputs, using high-rank update counterfactuals. This may offer a clearer picture of the impact of these compression techniques on model decision-making. Furthermore, establishing new interpretability benchmarks specific to models using LoRA and DoRA, combined with controlled A/B testing in deployment, can provide insights into the practical impacts of these techniques and safeguard against unforeseen adverse outcomes. Interpretability is absolutely critical to any applications where our model might be deployed as a decision-maker, or incorporated into a decision making pipeline (e.g. loan applications, mortgage pricing).

References

2015. Tensorboard: Visualize your tensorflow model. Software available from tensorflow.org.

2023. Polyak averaging. `https://paperswithcode.com/method/polyak-averaging`. Accessed: 2024-06-08.

2024. Ai bias audits: Using ai for hiring & employment. `https://www.dwt.com/insights/2024/05/ai-bias-audits-using-ai-for-hiring-employment`.

Armen Aghajanyan, Luke Zettlemoyer, and Sonal Gupta. 2020. Intrinsic dimensionality explains the effectiveness of language model fine-tuning. *arXiv:2012.13255*.

Steven Bird, Edward Loper, and Ewan Klein. 2009. Nltk: The natural language toolkit. URL: `http://www.nltk.org`.

Leo Breiman. 2001. *Random Forests*, volume 45. Springer.

Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 785–794. ACM.

William B. Dolan and Chris Brockett. 2005. Automatically constructing a corpus of sentential paraphrases. In *Proceedings of the Third International Workshop on Paraphrasing (IWP)*.

Jeremy Howard and Sebastian Ruder. 2018. Universal language model fine-tuning for text classification. *fast.ai and University of San Francisco; Insight Centre, NUI Galway; Aylien Ltd., Dublin*. Available at ArXiv:1801.06146.

Edward J. Hu, Yelong Shen, Phillip Williams, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*.

Pavel Izmailov, Dmitrii Podoprikhin, Timur Garipov, Dmitry Vetrov, and Andrew Gordon Wilson. 2019. Averaging weights leads to wider optima and better generalization.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*.

Divya Khyani and Siddhartha B S. 2021. An interpretation of lemmatization and stemming in natural language processing. *Shanghai Ligong Daxue Xuebao/Journal of University of Shanghai for Science and Technology*, 22:350–357.

Diederik P. Kingma and Jimmy Lei Ba. 2015. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A. Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, et al. 2017. Overcoming catastrophic forgetting in neural networks. *Proceedings of the National Academy of Sciences*, 114(13):3521–3526. Edited by James L. McClelland, Stanford University, Stanford, CA, approved February 13, 2017 (received for review July 19, 2016).

D. J. Ladani and N. P. Desai. 2020. Stopword identification and removal techniques on tc and ir applications: A survey. In *2020 6th International Conference on Advanced Computing and Communication Systems (ICACCS)*, pages 466–472, Coimbatore, India.

Chunyuan Li, Heerad Farkhoor, Rosanne Liu, and Jason Yosinski. 2018. Measuring the intrinsic dimension of objective landscapes. *arXiv preprint arXiv:1804.08838*.

Shih-Yang Liu, Chien-Yi Wang, Hongxu Yin, Pavlo Molchanov, Yu-Chiang Frank Wang, Kwang-Ting Cheng, and Min-Hung Chen. 2024. Dora: Weight-decomposed low-rank adaptation. *arXiv preprint arXiv:2402.09353*.

Sourab Mangrulkar, Sylvain Gugger, Lysandre Debut, Younes Belkada, Sayak Paul, and Benjamin Bossan. 2022. Peft: State-of-the-art parameter-efficient fine-tuning methods. `https://github.com/huggingface/peft`.

Ammar Mohammed and Rania Kora. 2023. A comprehensive review on ensemble deep learning: Opportunities and challenges. *Journal of King Saud University - Computer and Information Sciences*.

NVIDIA. 2023. Mixed precision training. `https://docs.nvidia.com/deeplearning/performance/mixed-precision-training/index.html`.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library.

Quora. 2017. Quora question pairs. Dataset released for the Kaggle "Quora Question Pairs" competition.

Richard Socher, Alex Perelygin, Jean Y Wu, Jason Chuang, Christopher D Manning, Andrew Y Ng, and Christopher Potts. 2013. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1631–1642.

Stanford University. 2024. Default final project handout: Minbert. Stanford CS224n: Natural Language Processing with Deep Learning.

Chris Taylor, Sebastian Raschka, and Abdou Aziz Diop. 2023. Implementation of dora. `https://github.com/catid/dora`.

Adina Williams, Nikita Nangia, and Samuel R Bowman. 2018. A broad-coverage challenge corpus for sentence understanding through inference. In *Proceedings of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*, pages 1112–1122.

A Adamax Optimizer

The Adamax optimizer was proposed alongside the Adam optimizer by Kingma and Ba (2015), where moment updates rely on the infinity norm, $L^p, p \to \infty$, rather than the L^2 norm. For our use-case whereby we seek to propagate gradient updates to embeddings when training in `iterative` and `full-model` fine-tuning modes, the infinity norm may be a more suitable measurement of gradient magnitude, since our embeddings are high dimensional (size 768) and gradient updates are likely sparse. Empirically, we found a significant performance increase when using Adamax as opposed to Adam and SGD (see Figure 7). Adam and RAdam were also inferior to Adamax for our training process.

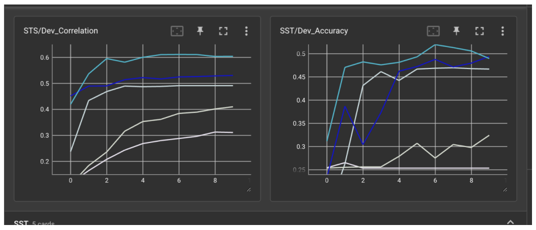

Figure 4: Adamax yields smoother performance increase and better convergence (light blue) as opposed to Adam (grey) and SGD (dark blue).

B Iterative BERT Backbone Unfreezing

Recent works such as Howard and Ruder (2018) have demonstrated the efficacy of iterative backbone unfreezing throughout fine-tuning of pre-trained large language models, whereby more and more of the large language model backbone is unfrozen as training progresses. Iterative unfreezing may yield superior generalization model performance and increase robustness of transfer learning to the phenomenon of catastrophic forgetting Kirkpatrick et al. (2017). Motivated by the promising results of progressive unfreezing in the literature, we implement iterative unfreezing (see subsection B.1) of the BERT backbones in our multit-task BERT model. Our early experiments comparing training convergence between full unfreezing and iterative unfreezing clearly demonstrated better convergence using the iterative strategy, and as such we sought to derive an optimal unfreezing schedule. Ultimately, our optimal schedule required fairly rapid unfreezing. We unfreeze 3 additional BERT backbone attention layers (there are 12 in total) each epoch until the entire backbone is unfrozen. Note also that when we unfreeze the first backbone layers, we make sure to unfreeze all post-attention BERT layers. Similarly, when all BERT backbone layers are unfrozen, we unfreeze not only through the first attention layer, but all the way back to the original Embedding layer. For a closer look at the iterative unfreezing controller, see subsection B.2.

B.1 Flexible Unfreezing MultitaskBERT Class Method

```python
def manage_freezing(self, structure: str):
    """
    Manages freezing of BERT Attention layers

    Adjusts the trainable status of layers in a model based on a specified
    structure command. This function can freeze all layers, unfreeze all layers,
    or unfreeze only the top N layers of the model.

    Args:
        structure (str): A command string that dictates how layers should be
                         frozen or unfrozen. It can be 'freezeall', 'unfreezeall',
                         or 'unfreezetopN' where N is an integer indicating the
                         number of top layers to unfreeze.

    Raises:
        ValueError: If the structure parameter does not follow the expected
                    format or specifies an invalid option.
    """
    for bert in (self.bert_sentiment, self.bert_sim):
        children = list(bert.children())
        # The ModuleList containing the 12 self-attention layers is at index 5
        attn_idx = 5
        attention_layers = children[attn_idx]
        total_attn_layers = len(attention_layers)

        if structure == "freezeall":
            # Freeze all layers
            for param in bert.parameters():
                param.requires_grad = False

        elif structure == "unfreezeall":
            # Unfreeze all layers
            for param in bert.parameters():
                param.requires_grad = True

        elif structure.startswith("unfreezetop"):
            # Attempt to extract the number of attn. layers to
            # unfreeze from the structure string
            try:
                n_layers = int(structure[len("unfreezetop") :])
            except ValueError:
                raise ValueError(
                    (
                        "Invalid layer specification. Ensure it "
                        "follows 'unfreezetopN' "
                        "format where N is a number."
                    )
                )

            # Freeze all layers first
            for param in bert.parameters():
                param.requires_grad = False

            # Unfreeze children after the attention layers
            for i in range(attn_idx+1, len(children)):
                for param in children[i].parameters():
                    param.requires_grad = True
            # Unfreeze the last n attention_layers
            for i in range(total_attn_layers - n_layers, total_attn_layers):
                for param in attention_layers[i].parameters():
                    param.requires_grad = True
        else:
            raise ValueError(
                (
                    "Invalid structure parameter. Use 'freezeall',"
                    "'unfreezeall', or "
                    "'unfreezetopN' where N is a number."
                )
            )
    return None
```

Listing 1: Manage Freezing Function

B.2 Iterative Unfreezing Controller for Training

```
for epoch in range(args.epochs):
    if args.fine_tune_mode == 'iterative':
        checkpoint_slice = args.epochs // 5   # no longer using this,
                            #but can be used for 'even-slice' unfreezing
        if epoch < 1:
            model.manage_freezing('freezeall')
            print(f"\n====== All BERT Layers Frozen ======\n")
        elif epoch < 2:
            model.manage_freezing('unfreezetop3')
            print(f"\n====== Top 3 BERT Layers Unfrozen ======\n")
        elif epoch < 3:
            model.manage_freezing('unfreezetop6')
            print(f"\n====== Top 6 BERT Layers Unfrozen ======\n")
        elif epoch < 4:
            model.manage_freezing('unfreezetop9')
            print(f"\n====== Top 9 BERT Layers Unfrozen ======\n")
        else:
            model.manage_freezing('unfreezeall')
            print(f"\n====== All BERT Layers Unfrozen ======\n")
            # 12 total attn layers and early layers including Embeddings
```

C AMP

There are a few key strategies that AMP employs to reduce computational load and memory utilization, primarily by using FP16 instead of FP32 wherever feasible, without sacrificing numerical stability and accuracy too significantly. This leads to performance improvements on hardware designed to effectively exploit lower precision arithmetic (such as our NVIDIA A100s).

Loss Scaling (LS)

To prevent numerical underflow in lower precision (FP16), gradients are scaled up by a factor S during computation:

$$L' = S \cdot L$$

After backpropagation, gradients are scaled down to update weights. This ensures that no significant digits are lost.

$$\frac{\partial L}{\partial w} = \frac{1}{S} \cdot \frac{\partial L'}{\partial w}$$

Precision Casting

Critical operations such as gradient accumulations and updates are performed in higher FP32 precision. To account for mixed precision operations, we cast appropriately to prevent information loss.

Master Weight Copy

A high precision (FP32) copy of the weights is maintained. This ensures accurate update steps:

$$w_{\text{new}} = w - \eta \cdot \text{cast}(\frac{\partial L}{\partial w}, \text{FP32})$$

D Ensembling

Towards achieving optimal multi-task model performance, we ensemble various mixtures of high-performance models together. As recently confirmed by Mohammed and Kora (2023) Mohammed and Kora (2023), ensembling methods provide consistent performance gains and generally yield flatter, more generalizable optima. This fundamental principle applies across traditional regression/-classification methods, tree-based methods, and deep learning, though in deep learning there is still some debate as to whether ensembling provides gains that can't be equivalently accomplished by larger deep neural network architectures. In general, the two overarching ensemble approaches are to concatenate the predictions of many weak models, such that each learns from each others mistakes (e.g. XGBoost Chen and Guestrin (2016)) or to average the predictions of many strong models (e.g. RandomForest Breiman (2001)).

In the present work, we apply the latter strategy to average the strengths and weaknesses of our various strong learners. We find that ensembling consistently increases performance across all three sub-tasks, and propose that despite the debate in deep learning literature as to whether ensembling provides gains inachievable with better architecture, the ease of ensembling versus spending days searching for more optimal architecture is well worth it when computational resources are sufficient to perform the ensembling. Given our general approach in this paper towards deriving more efficient, yet performant multi-task minBERT models, our ensembling approach can be regarded as a case study in the performance benefits that efficient fine-tuning can provide, agnostic of resource shortages.

Our ensembling approach is as follows. We initialize an EnsembledModel class with a list of pre-loaded, trained MultitaskBERT models (or MultitaskBERTDualityOfMan for the older Duality of Man architecture). Then, we aggregate sentiment, paraphrase and similarity predictions by stacking predictions from all EnsembledModel sub-models into a tensor, and taking the mean of these predictions. In this way, we hope to approximate a better optimum point by averaging the predictions of a number of sub-models converged to less optimal local minima.

Towards creating an ensemble which comprises of sub-models that excel at different sub-tasks, we implement separate optimizers with varying weight-decay and learning rate for each sub-task. We found that within a reasonable learning rate range ($\approx 1e-4$), variations in learning rate of up to $10x$ primarily had the effect of biasing model performance towards the task with larger learning rate. As such, we apply various learning rate scales to different sub-task across our ensemble sub-models (for examples, see the training invocations in Appendix H and Appendix I).

Our EnsembledModel class implementation is as follows:

```python
import torch

class EnsembledModel(nn.Module):
    def __init__(self, models):
        super(EnsembledModel, self).__init__()
        self.models = nn.ModuleList(models)

    def forward(self, input_ids, attention_mask):
        raise NotImplementedError("EnsembledModel does not support the
     forward "
            "method directly. Use predict_sentiment,
    predict_paraphrase, "
            "or predict_similarity methods.")

    def predict_sentiment(self, input_ids, attention_mask):
        with torch.no_grad():
            logits = torch.stack([model.predict_sentiment(input_ids,
                                        attention_mask) \
                                for model in self.models], dim=0)
            return torch.mean(logits, dim=0).squeeze()

    def predict_paraphrase(self, input_ids_1, attention_mask_1,
                           input_ids_2, attention_mask_2):
        with torch.no_grad():
            logits = torch.stack([model.predict_paraphrase(input_ids_1
    ,
                                        attention_mask_1,
                                        input_ids_2,
                                        attention_mask_2) \
                                        for model in self.models], dim
    =0)
            return torch.mean(logits, dim=0).squeeze()

    def predict_similarity(self, input_ids_1, attention_mask_1,
                        input_ids_2, attention_mask_2):
        with torch.no_grad():
            logits = torch.stack([model.predict_similarity(input_ids_1
    ,
                                            attention_mask_1,
                                            input_ids_2,
    attention_mask_2) \
                                            for model in self.models],
     dim=0)
            return torch.mean(logits, dim=0).squeeze()
```

Listing 2: Ensembled Model and Model Loading in PyTorch

To load models into an EnsembledModel (with the desired filepaths in args.filepaths), a minimal example is as follows. For example, we call our overall ensembling script with $python `ensemble.py model1.pt model2.pt`.

```
from multitask_bert import MultitaskBERT, MultitaskBERTDualityOfMan

def load_models(filepaths, device):
    models = []
    for filepath in filepaths:
        print(f"====== Loading Model: {filepath} ======")
        saved = torch.load(filepath, map_location=device)
        config = saved['model_config']
        if "duality" in filepath:
            model = MultitaskBERTDualityOfMan(config)
        else:
            model = MultitaskBERT(config)
        model.load_state_dict(saved['model'])
        model = model.to(device)
        model.eval()
        models.append(model)
    return models

def main(args):
    device = torch.device('cpu')
    if torch.cuda.is_available():
        device = torch.device('cuda')
    elif torch.backends.mps.is_available():
        device = torch.device('mps')

    model_list = load_models(args.filepaths, device)
    ensemble_model = EnsembledModel(model_list).to(device)

    test_multitask(args, model=ensemble_model)
```

D.1 Ensemble Results

We found ensembling to consistently boost performance (see Appendix H, Appendix I, Appendix J, Appendix K), and in general more models yielded better performance (Appendix L). Notably adding more "votes" to better models had a very minor postive impact on performance (see Oligarchy v1 in Appendix L), but generally the effect of adding more models was stronger. Our best model, "G6 Lisan al Gaib" outperforms the "G6 The Sage" model, with the difference between them being that the "Lisan al Gaib" variant includes more relatively low-performance models. Even in the case of weaker sub-models, we note very good ensembled performance (e.g. Appendix K).

E Exponential Moving Average (EMA) Model

E.1 Motivation

We implement a multitask-compatible version of the exponential moving average (EMA) model, a form of Polyak averaging Pol (2023). This decision was motivated by the success of stochastic weight averaging (SWA) introduced by Izmailov et al. (2019) Izmailov et al. (2019), who propose SWA as a solution to achieve more optimal convergence using exponential moving averages and cyclical (or constant) learning rates as compared to traditional SGD with learning rate decay. The general proposal of SWA is that SGD solutions are often inoptimal, but an average of solution points yields a more generalizable optimum point that yields superior performance on unseen data. EMA is similar to SWA, except that a running average of weights is maintained and updated every epoch, rather than updating weights to be an average of weights from the 'converged' weights at the end of various cycles.

E.2 Adopted Method and Extensions

We use a custom cyclical learning rate for our EMA experiments, motivated by the fact that Izmailov et al. (2019) find cyclically decreasing learning rates to better approximate solutions each cycle, despite the fact that the loss space is overall better explored by constant learning rate SGD due to its larger updates and greater movement. Given our compute limitations, we opted for cyclical learning rate scheduling, towards producing an average of fewer better weights versus the average of many weaker weights.

We implement a custom cyclical learning rate scheduler which decreases learning rate by `8x` each cycle, with cycles each running for 2 epochs. We further match these 2 epoch cycles up with our iterative unfreezing strategy, whereby each successive deepening of the unfreezing corresponds perfectly to a learning rate cycle. Because of the demonstrably superior performance of `Adamax` as an optimizer (see Appendix A), we use `Adamax` as opposed to `SGD`, which is typically utilized for SWA methods. Thirdly, as an extension to the official PyTorch implementation of EMA, we design a custom implementation of the EMA model wrapper which performs weight-averaging each epoch as usual, but at evaluation-time adopts the multiple task output methods of our superclass `MultitaskBERT` model. We update BatchNorms at the end of training in line with the original work, using the official PyTorch implementation from `torch.optim.swa_utils`.

```python
import copy

class EMA:
    def __init__(self, model, decay):
        self.model = model
        self.decay = decay
        self.shadow = copy.deepcopy(model.state_dict())
        self.ema_model = copy.deepcopy(model)

    def update(self):
        for name, param in self.model.state_dict().items():
            if name in self.shadow:
                self.shadow[name] = self.decay * self.shadow[name] + \
                    (1.0 - self.decay) * param

    def apply_shadow(self):
        self.ema_model.load_state_dict(self.shadow)
        return self.ema_model
```

Listing 3: EMA Class

E.3 EMA Results

Unfortunately, our EMA results were very poor. We propose that this is likely a result of (1) the relatively few epochs we train for and (2) our non-traditional implementation choices. With a traditional decay value of 0.999, we retain far too much of our initial (and low performing) weights due to the very few epochs we train for (see medium blue curve below), and with a lower decay of 0.5 we see instability (see light blue curve below), especially as training progresses (Figure 5). Furthermore, our use of extremely short cycles (2 epochs), cyclical learning rate with `Adamax` and EMA (rather than `SGD` and SWA) are non-traditional and seem to significantly detract from model training performance.

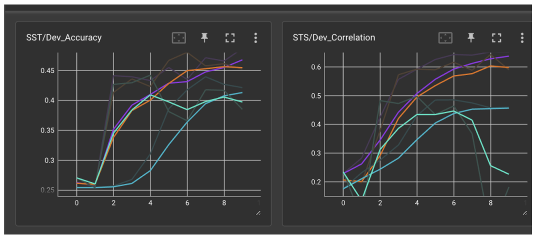

Figure 5: Impact of cyclic learning rate and EMA on performance. Medium Blue is EMA with decay 0.999, Light Blue is EMA with decay 0.5, reference curves are traditional training with cyclic LR scheduling.

F Custom Loss Functions

F.1 Optimal Loss Functions

$$L_{\text{CE}}(p, y) = -\sum_{i=1}^{C} y_i \log\left(\frac{e^{p_i}}{\sum_{j=1}^{C} e^{p_j}}\right)$$

where C is the number of classes, p are the predicted logits, and y is the one-hot encoded true label vector.

$$L_{\text{BCE}}(p, y) = -\frac{1}{N}\sum_{i=1}^{N} [y_i \log(\sigma(p_i)) + (1 - y_i)\log(1 - \sigma(p_i))]$$

where $\sigma(p_i) = \frac{1}{1+e^{-p_i}}$ denotes the sigmoid function, p represents the logits from the model, y is the binary label, and N is the number of samples.

$$L_{\text{Pearson}}(p, y) = 1 - \frac{\sum_{i=1}^{n}(p_i - \bar{p})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(p_i - \bar{p})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

where $\bar{p}$ and $\bar{y}$ are the means of the predicted and true values, respectively.

The optimization objective is to find the parameter set $\theta = \{\theta_{\text{SST}}, \theta_{\text{Para}}, \theta_{\text{STS}}, \Omega\}$ that minimizes the total loss L:

$$\theta^* = \arg\min_{\theta} L(\theta)$$

F.2 Other Loss Function Experiments

Kullback-Leibler Divergence Loss (KL)

This loss measures how one probability distribution diverges from a second, expected probability distribution:

$$L = \sum_{i=1}^{n} y_i \cdot (\log(y_i) - \mathbf{z}_i)$$

where y_i represents the target probability for the i-th instance, and $\mathbf{z}_i$ denotes the predicted log-probabilities for the i-th instance.

Mean Squared Error Loss (MSE)

Typically used for regression:

$$L = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$$

where $\hat{y}_i$ represents the predicted value for the i-th instance and y_i denotes the actual target value for the i-th instance.

Log-Cosh Loss

Log-cosh is a smooth loss that behaves much like the mean squared error but does not yield gradients as steep as MSE:

$$L = \frac{1}{n}\sum_{i=1}^{n} \log(\cosh(\hat{y}_i - y_i))$$

where $\hat{y}_i$represents the predicted value for the ith instance. y_i denotes the actual target value for the ith instance.

BCE/MSE Mixed Loss

A combination of BCE and MSE, useful for tasks that benefit from both probabilistic and regression-based approaches. We tested this mixed loss on both the SST and STS tasks, with some improvements in early epochs, but ultimately worse convergence.

$$L = \lambda_{BCE} \cdot L_{BCE} + \lambda_{MSE} \cdot L_{MSE}$$

Ordinal Cross Entropy Loss

This loss function is used for ordinal regression. It involves calculating the cumulative probabilities for each class and then applying the cross entropy loss to these cumulative probabilities. The loss is defined as:

$$L = -\sum_{i=1}^{n} [\text{one\_hot}(y_i, k) \cdot \log(\text{cumulative\_probs}_i) + (1 - \text{one\_hot}(y_i, k)) \cdot \log(1 - \text{cumulative\_probs}_i)]$$

where $\text{one\_hot}(y_i, k)$ is the one-hot encoded vector of the target class label y_i with k classes. The cumulative probabilities $\text{cumulative\_probs}_i$ for a predicted class label j are calculated as:

$$\text{cumulative\_probs}_i(j) = \sum_{l=1}^{j} \text{softmax}(\mathbf{z}_i)(l)$$

where $\mathbf{z}_i$ are the logits predicted by the model for the i-th instance, and $\text{softmax}(\mathbf{z}_i)(l) = \frac{e^{z_{il}}}{\sum_{m=1}^{k} e^{z_{im}}}$ is the softmax function applied to these logits.

Contrastive Loss

Often used in learning embeddings or in siamese networks, contrastive loss helps to ensure that similar items are closer to each other than they are to dissimilar items. The loss is defined as:

$$L = \sum_{i=1}^{n} \left[(1 - y_i) \frac{1}{2} (\text{dist}_i)^2 + y_i \frac{1}{2} \max(0, \text{margin} - \text{dist}_i)^2 \right]$$

where y_i is the binary label indicating whether the pair of instances are similar (1) or dissimilar (0), dist_i is the Euclidean distance between the embeddings of the pair, computed as:

$$\text{dist}_i = \sqrt{\sum_{j=1}^{d} (x_{i1j} - x_{i2j})^2}$$

where x_{i1j} and x_{i2j} represent the j-th dimension of the embeddings for the first and second instance of the i-th pair, respectively, and d is the dimensionality of the embeddings. The margin is a predefined threshold that defines how far dissimilar pairs should be pushed apart.

```python
def pearson_correlation_loss(preds, target):
    vx = preds - torch.mean(preds)
    vy = target - torch.mean(target)
    cost = torch.sum(vx * vy) / (torch.sqrt(torch.sum(vx ** 2)) *
    torch.sqrt(torch.sum(vy ** 2)))
    return 1 - cost  # return 1 - correlation to make it a loss

def log_cosh_loss(preds, target):
    def log_cosh(x):
        return x + torch.nn.functional.softplus(-2. * x) - torch.log(
    torch.tensor(2.0))
    return torch.mean(log_cosh(preds - target))

class OrdinalCrossEntropyLoss(nn.Module):
    def __init__(self):
        super(OrdinalCrossEntropyLoss, self).__init__()

    def forward(self, logits, targets):
        targets_one_hot = F.one_hot(targets, num_classes=logits.size
    (1)).float()
        cumulative_probs = torch.cumsum(F.softmax(logits, dim=1), dim
    =1)
        loss = - torch.sum(targets_one_hot * torch.log(
    cumulative_probs) + (1 - targets_one_hot) * torch.log(1 -
    cumulative_probs))
        return loss
```

Listing 4: Python Code for Various Loss Functions and Training Loops

In the training loop for SST, we tried CE, BCE, KL loss (which can be equivalent to CE but has different convergence properties which may make it less favorable), ordinal CE, and finally a "mixed" version which was a hybrid between MSE and BCE. We found that the mixed loss led to the best performance for SST.

```
# ======= Various Criterion Options ==========
criterion_ce = nn.CrossEntropyLoss(reduction='mean')
criterion_bce = nn.BCEWithLogitsLoss(reduction='mean')
criterion_kl = nn.KLDivLoss(reduction='batchmean')
criterion_ord = OrdinalCrossEntropyLoss()

if criterion == criterion_ce:
    loss = criterion(logits, b_labels.view(-1))
elif criterion == criterion_bce:
    labels_one_hot = F.one_hot(b_labels, num_classes=5).float()
    loss = criterion(logits, labels_one_hot)
elif criterion == criterion_kl:
    labels_one_hot = F.one_hot(b_labels, num_classes=5).float()
    log_probs = F.log_softmax(logits, dim=-1)
    loss = criterion(log_probs, labels_one_hot)
elif criterion == "mixed":
    lambda_bce = 1
    lambda_mse = 0.1
    labels_one_hot = F.one_hot(b_labels, num_classes=5).float()
    loss_bce = nn.BCEWithLogitsLoss()(logits, labels_one_hot)
    probabilities = F.softmax(logits, dim=1)
    scores = torch.tensor([1, 2, 3, 4, 5], dtype=torch.float32,
        device=device).unsqueeze(0)
    expected_scores = torch.matmul(probabilities, scores.T).squeeze(1)
    loss_mse = nn.MSELoss()(expected_scores, b_labels.float())
    loss = lambda_bce * loss_bce + lambda_mse * loss_mse
```

In the training loop for STS, we tried MSE, Pearson, constrastive loss, and a hybrid between MSE and Pearson. We found that Pearson loss works best. Contrastive loss led to a slight decrease in performance. This may be because contrastive loss is fundamentally binary, optimizing to categorize pairs as either similar or dissimilar without accounting for degrees of similarity (which is what the STS task includes). The dichotomous nature of contrastive loss doesn't support the precise quantification of semantic closeness required by STS.

```
criterion_mse = nn.MSELoss()  # MSE loss since labels are 0-5
    criterion_pearson = pearson_correlation_loss

        if criterion in [criterion_mse, criterion_pearson]:
            loss = criterion(logits, b_labels)
        elif criterion == "mixed":
            lambda_mse = 0.5
            lambda_pears = 1.0
            mse_loss = criterion_mse(logits, b_labels)
            pears_loss = pearson_correlation_loss(logits, b_labels)
            loss = lambda_mse * mse_loss + lambda_pears * pears_loss
```

G Chronological Performance Experiments

Below we detail the chronological and qualitative experiments performed towards deriving our strongest model architectures and training routines. In early experiments, we worked iteratively and used learning curves obtained via TensorBoard ten (2015) as our measure of model training stability and performance. We performed all early experiments using only SST and STS training, due to the massive compute requirements for training the Quora paraphrase task. In occasional interleaved full-training experiments, we found that bench-marking optimal architecture and learning rates using these two tasks served as a fairly reliable proxy full-training performance. Experiments were run for 10 epochs with an initial learning rate of $2e-4$ and multiplicative learning rate decay of $0.5^{\text{epoch}} \times LR$ unless indicated otherwise.

First, we tested several different optimizers. SGD (green and purple below) performed much worse than `Adam` variants (light blue), as shown in Figure 6.

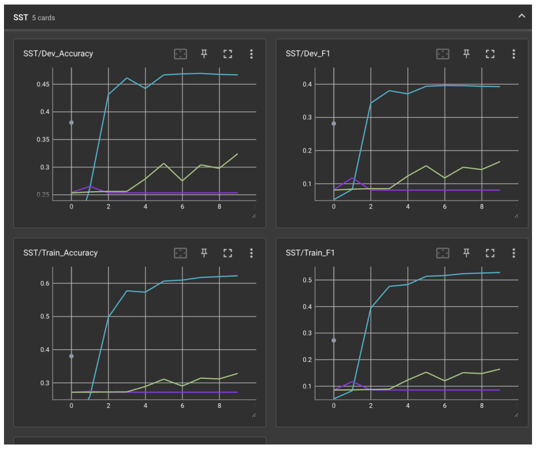

Figure 6: Performance comparison of SGD and Adam variants.

`Adamax`, which has been proposed as a suitable optimizer for embeddings, performed better than vanilla `Adam` and `SGD` tests on `SST` and `STS`. Adding a shared Linear Similarity Feature Extraction layer shared between the Quora and STS classifiers, after the BERT backbone, enabled more improvement (see Figure 7). We hypothesized, and later confirmed, that this shared sub-layer would further improve performance when training for `Quora`, since `Quora` and `STS` are both similarity-based tasks. The shared Similarity Feature Extraction layer might enable these two tasks to learn more from each other even with a frozen backbone.

Result (no Quora training):

1. **Dev sentiment accuracy:** 0.520
2. **Dev paraphrase accuracy:** 0.368
3. **Dev STS correlation:** 0.611

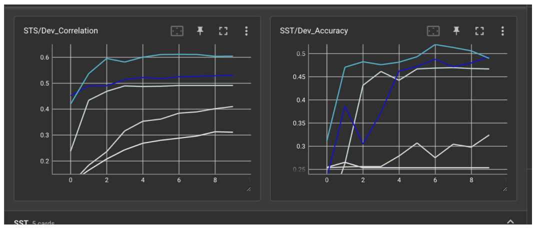

Figure 7: Adamax (dark blue) yields smoother performance increase and better convergence as opposed to Adam and SGD (grey). Adamax + Linear Similarity Feature Extraction layer between minBERT and Quora/STS tasks further improves performance (light blue)

We then implemented a number custom loss functions, including a Pearson Correlation Loss designed to directly optimize STS for Pearson Correlation (see Appendix F). The Pearson Correlation Loss Optimizer improved `STS` performance, but somehow decreased `SST` performance even though it wasn't applied to `SST`, as shown in Figure 8. This impact on SST performance was later resolved by the "Duality of Man" architecture (see Appendix H).

Pearson Correlation Loss result (no Quora training):

1. **Dev sentiment accuracy:** 0.396
2. **Dev paraphrase accuracy:** 0.368
3. **Dev STS correlation:** 0.688

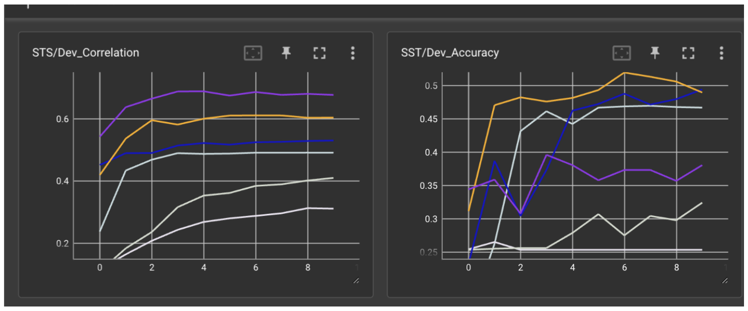

Figure 8: Effect of Pearson Correlation Loss Optimizer (Purple) on STS and SST performance.

We then upgraded the Similarity Feature Extraction layer to be non-linear instead of linear and noted further STS performance improvements, as shown in Figure 9.

Result (no Quora training):

1. **Dev sentiment accuracy:** 0.384
2. **Dev paraphrase accuracy:** 0.632
3. **Dev STS correlation:** 0.737

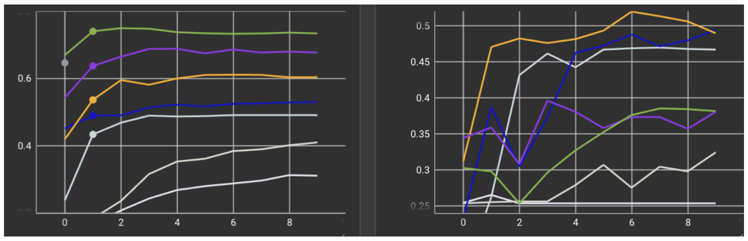

Figure 9: NonLinear Similarity Feature Layer improves STS (light green).

Towards gaining a rough-and-dirty approximation of Quora performance, we then implemented partial Quora training whereby we would randomly select approximately 5% (200 batches at batch size 96) of `Quora` data to get a rough and dirty train and benchmark. Unfortunately, this approximation did not work. Performance definitely decreased on the other tasks, as shown in Figure 10. Notably, with full Quora training experiments, we did not see the same performance decrease.

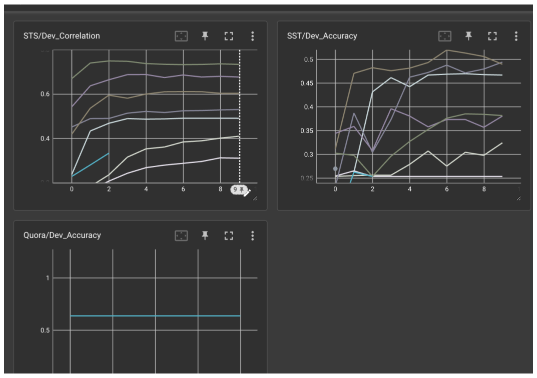

Figure 10: Performance impact of using a small portion of Quora data for training and benchmarking.

We then implemented iterative unfreezing (see Appendix B for details). Iterative unfreezing worked well for `STS`. However, we noticed some continued instability on `SST` with an initial learning rate of `2e-4`, as shown in Figure 11.

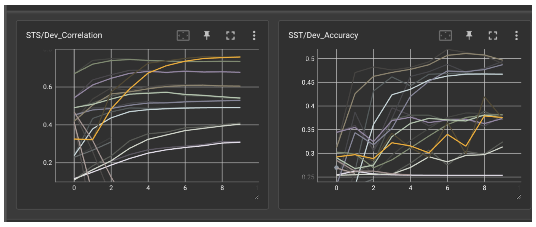

Figure 11: Iterative Unfreezing (Orange) benefits STS convergence. SST suffers instability.

To explore the SST learning instability, we modified initial learning rate, and noticed (unexpectedly, given the SST instability) that faster learning rates were very important to `SST`, but detrimental to `STS`. We also found that larger regularization (`weight_decay`) hurt `SST`. See Figure 12 below.

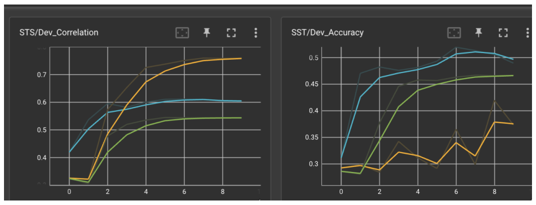

Figure 12: Orange (low learning rate) benefits STS performance while green (high learning rate) benefits SST performance. Large regularization (weight decay of $1e-3$) decreases performance on both SST and STS.

Towards optimizing learning rate for each sub-task, we then introduced 3 *separate* `Adamax` optimizers and 3 *separate* learning rate schedulers towards obtaining more degrees of freedom on training optimization for each task. In the first experiment, we multiply `SST` by `10` (purple). This did help `SST` do well and stabilized the learning curves, but `STS` performance decreased, possibly due to being overwhelmed by `STS`. This finding that increased sub-task learning rate within a reasonable range ($\approx 1e-4$ to $2e-4$) leads to increased bias towards learning that task held for future experiments as well. See Figure 13 below.

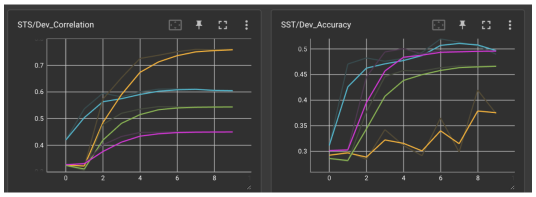

Figure 13: Effect of increasing the initial learning rate for SST by `10` on training stability and performance.

To alleviate the sensitivity in balancing SST and STS performance, we separated the minBERT backbone into two separate "Sent" (`SST`) and "Sim" (`Quora` + `STS`) backbones, with moderate performance improvements (see Appendix H for more dualities on this architecture, dubbed "Duality of Man" for the two backbones). This architecture prevented `SST` from blowing up the `STS` performance when increasing the learning rate. Furthermore, introducing some `nltk` preprocessing (lemmatization, stopword removal, special character cleanup) to the input data for all three tasks yielded minor gains. See Figure 14 below.

Duality of Man + nltk Initial Result (No Quora training):

1. **Dev sentiment accuracy:** 0.510
2. **Dev paraphrase accuracy:** 0.368
3. **Dev STS correlation:** 0.760

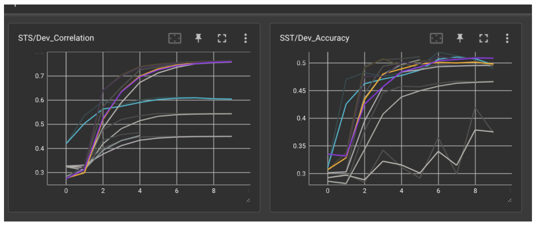

Figure 14: Effect of separating `minBERT` backbones (orange) and introducing `nltk` preprocessing (purple) on performance.

We ran a massive 50 epoch (10 epoch cycles for each unfreezing of `1/4` of the backbone) experiment for 35 epochs overnight, and noticed that the `Quora` dataset did not appreciably improve in performance until the backbone was somewhat unfrozen (epoch 10 here), as shown in Figure 15. As such, we implemented a dynamic Quora training option whereby Quora training does not begin until a later stage of unfreezing (though we did not experiment with this much).

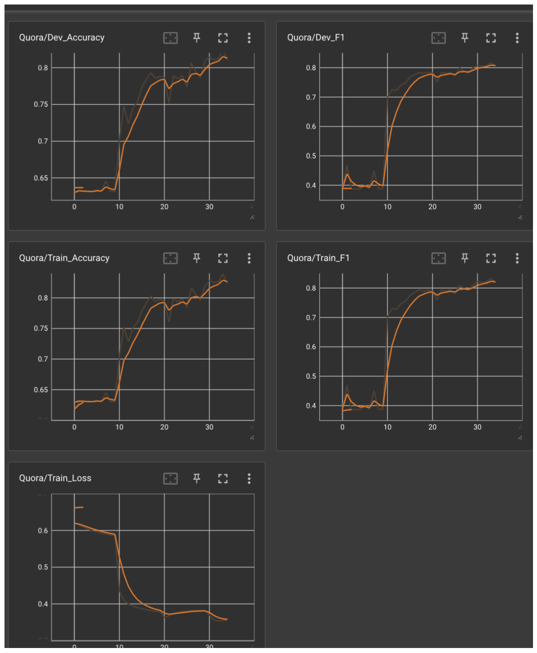

Figure 15: Performance improvement of the Quora dataset after unfreezing the backbone.

Our next experiment used a lower initial learning rate (`1e-4`) with a shallower learning rate decay ($0.7^{\text{epoch}} * LR$). This led to worse performance. In particular, `STS` suffered, as shown in Figure 16.

Result:

1. **Sentiment classification accuracy:** 0.487
2. **Paraphrase detection accuracy:** 0.368
3. **Semantic Textual Similarity correlation:** 0.651

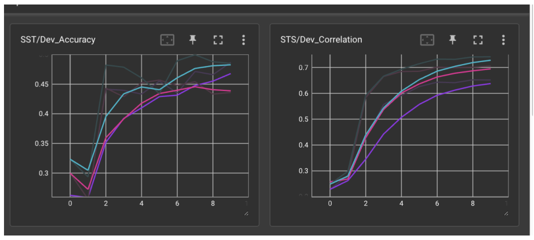

Figure 16: Impact of lower learning rate and `lambda = 0.7**epoch` scheduler on performance (purple).

We then tried a triangular cyclical learning rate schedule from $1e-4$ to $2e-4$, seeing very little difference. With the cylical learning rate scheduler still in place, we experiment with an exponential moving average (EMA) wrapper for our model, which decreased performance, especially in later epochs. See Appendix E for details on the EMA experiment. The learning curve is also below (Figure 17).

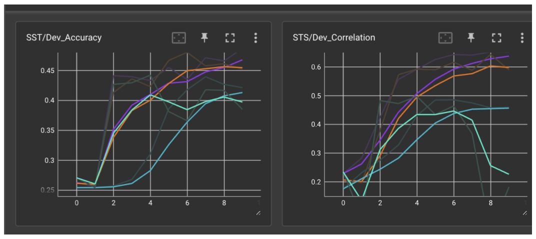

Figure 17: Impact of cyclic learning rate and EMA on performance.

We then scrapped EMA, and continue exploring optimal multiplicative learning rate scheduling, noting that more aggressive decay (multiplicative factor 0.7) was better than shallower decay (0.85), especially for the SST task. See Figure 18.

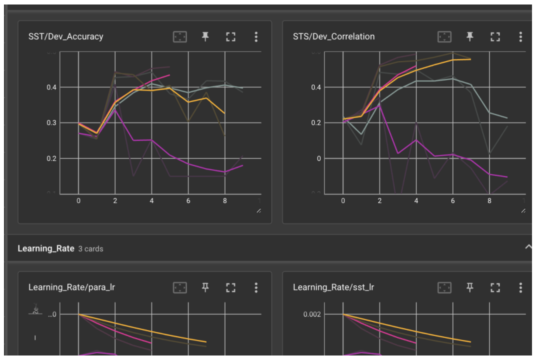

Figure 18: Comparison of Lambda LR schedulers: `0.7` (pink) vs `0.85` (orange).

Reducing learning rate even more (multiplicative factor of 0.5) and reducing regularization helped the overall learning trajectory, as shown in Figure 19.

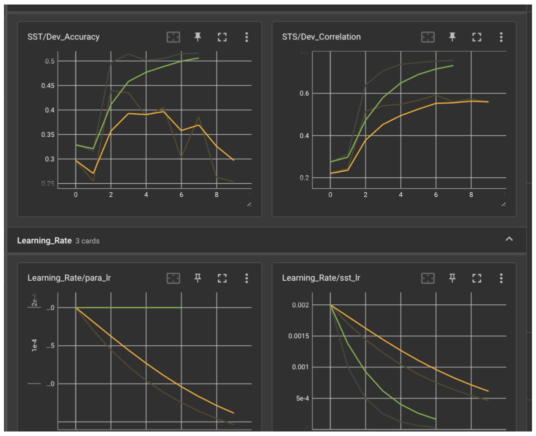

Figure 19: Impact of further reduced learning rate (0.5) and less weight decay on learning trajectory.

Many further architecture and learning routine experiments were performed, but are not detailed here. See subsection 4.2 for our ultimately best architecture and Appendix D for our ensembling strategy to achieve maximal multi-task performance.

H "Duality of Man" Model Variants and Ensemble

H.1 "Duality of Man" Architecture

With "Duality of Man" model variants we introduced a separate BERT backbone for the SST task. In experiments, we found that Quora and STS training benefitted one another via shared sub-layers and backbone, likely the result of the inherent similarity between the two tasks. Both the paraphrase detection (Quora) and similarity score (STS) tasks seek to understand how similar two sentences are. On the other hand, the SST task is focused on sentiment, which is largely orthogonal to sentence similarity (especially since we derive sentiment based on the input of one sentence at a time). Attempting to train all three models on the same BERT backbone led to instability in tuning learning rates appropriately so as not to compromise SST performance. By enforcing our prior knowledge that the SST task differs significantly from the other two and partitioning its backbone separately, we saw hugely increased training stability.

Compared to our best "Yin Yang" model architecture, the earlier "Duality of Man" architecture lacks a few important features. The "Duality of Man" Similarity Feature Extraction sub-layer shared between the paraphrase and STS tasks lacks the unadulterated embeddings of the two sentences. The feature extraction includes only the element-wise product, element-wise difference, and cosine similarity of input sentence embeddings for the Para and STS classifier layers (see below). There are also no BatchNorms in the "Duality of Man" classifier layers. The latter made the more significant difference in boosting "Yin Yang" model variant performance over this earlier "Duality of Man" architecture.

```python
def extract_comparison_features(self,
                                input_ids_1, attention_mask_1,
                                input_ids_2, attention_mask_2):
        """
        Given a batch of pairs of sentences, extract comparison features
        for Quora and STS tasks.

        Useful for both Sim tasks.
        """

        embeds_1 = self.forward(input_ids_1, attention_mask_1, which_bert = "sim")
        embeds_2 = self.forward(input_ids_2, attention_mask_2, which_bert = "sim")
        cosine_sim = F.cosine_similarity(embeds_1, embeds_2).unsqueeze(-1)
        elem_prods = embeds_1 * embeds_2
        diff = torch.abs(embeds_1 - embeds_2)
        features = torch.cat([diff, elem_prods, cosine_sim], dim=1)
        features = self.comparison_features_fcn(features)

        return features
```

G6 "Duality of Robust Men" (`RAdam`, 5x instead of 10x LR for `SST`), `1e-4` weight decay for `para`

```
$python multitask_classifier.py -fine-tune-mode iterative -optim RAdam
-sst_lr_multiplier 5 -para_weight_decay 1e-4 -lr 1e-4 -use_gpu -amp
-batch_size 64 -train_sst -train_quora -train_sts -clf conv -epochs 10
```

G6 "Duality of Faster Men" - same as above, higher LR

```
$python multitask_classifier.py -fine-tune-mode iterative
-sst_lr_multiplier 5 -para_weight_decay 1e-4 -lr 2.1e-4 -use_gpu -amp
-batch_size 64 -train_sst -train_quora -train_sts -clf conv
```

G6 "Duality of Slow Men"

```
$python multitask_classifier.py -fine-tune-mode full-model
-sst_lr_multiplier 5 -para_weight_decay 1e-4 -lr 5e-5 -use_gpu -amp
-batch_size 64 -train_sst -train_quora -train_sts -clf conv
```

G6 "Duality of Nonlinear Men"

```
$python multitask_classifier.py -fine-tune-mode iterative
-sst_lr_multiplier 5 -para_weight_decay 1e-4 -lr 1e-4 -use_gpu -amp
-batch_size 64 -train_sst -train_quora -train_sts -clf nonlinear
```

G6 "Mega Duality" (20 epochs)

```
$python multitask_classifier.py -fine-tune-mode iterative
-sst_lr_multiplier 5 -para_weight_decay 1e-4 -lr 1e-4 -use_gpu -amp
-batch_size 64 -train_sst -train_quora -train_sts -clf conv -epochs 20
```

H.2 Ensemble Command

We ensemble all of the Duality of Man variants using the following command.

```
$python ensemble.py -filepaths g6-duality-of-man.pt duality-of-fast-man.pt
duality-of-long-distance-runners-20.pt duality-of-man-nonlinear.pt
duality-of-slow-man.pt duality-of-robust-men.pt
```

Result:

1. **Sentiment classification accuracy:** 0.533
2. **Paraphrase detection accuracy:** 0.858
3. **Semantic Textual Similarity correlation:** 0.853

I "Yin Yang" Model Variants and Ensemble

Towards improving the "Duality of Man" variants architecture, the Yin Yang model architecture (our best multi-task model architecture) includes both pure sentence embeddings and the extracted elementwise product, cosine similarity, and embedding difference features. We also add more BatchNorms to the classifier layers, with the latter providing a significant performance boost. See below for the new shared similarity task feature extraction layer, and the updated classifier layers.

```python

def __init__(self, config):
    ...
    # ===== sentiment =====
    ...
    self.sentiment_conv = nn.Sequential(
        nn.Conv1d(in_channels=1, out_channels=4, kernel_size=3), # note: input needs to be unsqueezed
        nn.BatchNorm1d(4),
        nn.Flatten(),  # Flatten layer to convert (N, 4, D-2) to (N, 4*(D-2))
        nn.Linear(4 * (BERT_HIDDEN_SIZE - 2), BERT_HIDDEN_SIZE // 2),
        nn.BatchNorm1d(BERT_HIDDEN_SIZE // 2),
        nn.ReLU(),
        nn.Linear(BERT_HIDDEN_SIZE // 2, N_SENTIMENT_CLASSES)
    )

    # ====== comparison task shared layer ======
    # learn linear mapping which can be shared among STS and Quora tasks for comparison features
    self.comparison_features_fcn = nn.Sequential(
        nn.Linear(FEATURE_SIZE, FEATURE_PROJ_SIZE),
        nn.BatchNorm1d(FEATURE_PROJ_SIZE),
        nn.ReLU(),
        nn.Linear(FEATURE_PROJ_SIZE, FEATURE_PROJ_SIZE),
        nn.BatchNorm1d(FEATURE_PROJ_SIZE)
    )

    # ===== paraphrase ======
    ...
    self.paraphrase_conv =  nn.Sequential(
        nn.Conv1d(in_channels=1, out_channels=4, kernel_size=3), # note: input needs to be unsqueezed
        nn.BatchNorm1d(4),
        nn.Flatten(),  # Flatten layer to convert (N, 4, D-2) to (N, 4*(D-2))
        nn.Linear(4 * (FEATURE_PROJ_SIZE - 2), BERT_HIDDEN_SIZE // 2),
        nn.BatchNorm1d(BERT_HIDDEN_SIZE // 2),
        nn.ReLU(),
        nn.Linear(BERT_HIDDEN_SIZE // 2, 1)
    )
    ...
    # ===== similarity ========
    ...

    self.similarity_conv = nn.Sequential(
        nn.Conv1d(in_channels=1, out_channels=4, kernel_size=3), # note: input needs to be unsqueezed
        nn.BatchNorm1d(4),
        nn.Flatten(),  # Flatten layer to convert (N, 4, D-2) to (N, 4*(D-2))
        nn.Linear(4 * (FEATURE_PROJ_SIZE - 2), BERT_HIDDEN_SIZE // 2),
        nn.BatchNorm1d(BERT_HIDDEN_SIZE // 2),
        nn.ReLU(),
        nn.Linear(BERT_HIDDEN_SIZE // 2, 1)
    )
```

```python

...

def extract_comparison_features(self,
                                input_ids_1, attention_mask_1,
                                input_ids_2, attention_mask_2):
    """
    Given a batch of pairs of sentences, extract comparison features
    for Quora and STS tasks.

    Useful for both Sim tasks.
    """

    embeds_1 = self.forward(input_ids_1, attention_mask_1, which_bert
    = "sim")
    embeds_2 = self.forward(input_ids_2, attention_mask_2, which_bert
    = "sim")
    cosine_sim = F.cosine_similarity(embeds_1, embeds_2).unsqueeze(-1)
    elem_prods = embeds_1 * embeds_2
    diff = torch.abs(embeds_1 - embeds_2)
    features = torch.cat([embeds_1, embeds_2, diff, elem_prods,
    cosine_sim], dim=1)
    features = self.comparison_features_fcn(features)

    return features
```

I.1 G6 "Yin Yang"

```
$python multitask_classifier.py -fine-tune-mode iterative -lr 1e-4
-use_gpu -amp -batch_size 64 -train_sst -train_quora -train_sts -clf conv
-sst_weight_decay 8e-3 -para_weight_decay 1e-5 -sts_weight_decay 9e-3
-lr_lambda 0.55 -optim Adamax -sst_lr_multiplier 7 -para_lr_multiplier 1.0
-sts_lr_multiplier 3 -epochs 7
```

Result:

1. **Dev sentiment accuracy:** 0.516
2. **Dev paraphrase accuracy:** 0.851
3. **Dev STS correlation:** 0.861

I.2 G6 "Yang Yin"

```
$python multitask_classifier.py -fine-tune-mode iterative -lr 1e-4
-use_gpu -amp -batch_size 64 -train_sst -train_quora -train_sts -clf conv
-sst_weight_decay 8e-3 -para_weight_decay 1e-5 -sts_weight_decay 9e-3
-lr_lambda 0.55 -optim Adamax -sst_lr_multiplier 3 -para_lr_multiplier 4
-sts_lr_multiplier 1 -epochs 7
```

Result:

1. **Dev sentiment accuracy:** 0.515
2. **Dev paraphrase accuracy:** 0.888
3. **Dev STS correlation:** 0.835

I.3 G6 "Yin Yang Fast"

```
$python multitask_classifier.py -fine-tune-mode iterative -lr 1.5e-4
-use_gpu -amp -batch_size 64 -train_sst -train_quora -train_sts -clf conv
-sst_weight_decay 8e-3 -para_weight_decay 1e-5 -sts_weight_decay 9e-3
```

```
-lr_lambda 0.55 -optim Adamax -sst_lr_multiplier 5 -para_lr_multiplier 5
-sts_lr_multiplier 2 -epochs 7
```

Result:

1. **Dev sentiment accuracy:** 0.513
2. **Dev paraphrase accuracy:** 0.887
3. **Dev STS correlation:** 0.816

I.4 G6 "Yin Yang Steady Hand"

```
$python multitask_classifier.py -fine-tune-mode iterative -lr 1e-4
-use_gpu -amp -batch_size 64 -train_sst -train_quora -train_sts -clf conv
-sst_weight_decay 9e-3 -para_weight_decay 1e-5 -sts_weight_decay 1e-2
-lr_lambda 0.5 -optim Adamax -sst_lr_multiplier 4 -para_lr_multiplier 5
-sts_lr_multiplier 3 -epochs 7
```

Result:

1. **Sentiment classification accuracy:** 0.522
2. **Paraphrase detection accuracy:** 0.888
3. **Semantic Textual Similarity correlation:** 0.848

I.5 G6 "Yang Yin The Parabola"

```
$python multitask_classifier.py -fine-tune-mode iterative -lr 1e-4
-use_gpu -amp -batch_size 64 -train_sst -train_quora -train_sts -clf conv
-sst_weight_decay 9e-3 -para_weight_decay 1e-5 -sts_weight_decay 1e-2
-lr_lambda 0.5 -optim Adamax -sst_lr_multiplier 3 -para_lr_multiplier 8
-sts_lr_multiplier 1 -epochs 7
```

Result:

1. **Sentiment classification accuracy:** 0.515
2. **Paraphrase detection accuracy:** 0.892
3. **Semantic Textual Similarity correlation:** 0.801

I.6 G6 "Yin Yang" 2

Minor adjustments to the original to better time the SST optimal point towards the 7th epoch.

```
$python multitask_classifier.py -fine-tune-mode iterative -lr 1e-4
-use_gpu -amp -batch_size 64 -train_sst -train_quora -train_sts -clf conv
-sst_weight_decay 9e-3 -para_weight_decay 1e-5 -sts_weight_decay 1e-2
-lr_lambda 0.5 -optim Adamax -sst_lr_multiplier 4 -para_lr_multiplier 1.0
-sts_lr_multiplier 3 -epochs 7
```

Result:

1. **Sentiment classification accuracy:** 0.520
2. **Paraphrase detection accuracy:** 0.842
3. **Semantic Textual Similarity correlation:** 0.858

I.7 G6 "Yue the Moon Spirit" - "Yin Yang" Ensemble

We ensemble all of the Yin Yang variants (other than Steady Hand 2,3,4,5,6 which were not yet available) via:

```
$python ensemble.py -filepaths g6-yin-yang.pt g6-yang-yin.pt
g6-yin-yang-fast.pt g6-yin-yang-steady-hand.pt g6-yang-yin-the-parabola.pt
g6-yin-yang-2.pt
```

Result:

1. **Sentiment classification accuracy:** 0.522
2. **Paraphrase detection accuracy:** 0.889
3. **Semantic Textual Similarity correlation:** 0.858

Based on the success of the Steady Hand variant, we later made more to add to our final ensembles.

I.8 G6 "Yin Yang Steady Hand" 2

```
$python multitask_classifier.py -fine-tune-mode iterative -lr 1e-4
-use_gpu -amp -batch_size 64 -train_sst -train_quora -train_sts -clf conv
-sst_weight_decay 9e-3 -para_weight_decay 1e-5 -sts_weight_decay 1e-2
-lr_lambda 0.5 -optim Adamax -sst_lr_multiplier 3 -para_lr_multiplier 4
-sts_lr_multiplier 2 -epochs 7
```

Individual results not recorded.

I.9 G6 "Yin Yang Steady Hand" 3

```
$python multitask_classifier.py -fine-tune-mode iterative -lr 1e-4
-use_gpu -amp -batch_size 64 -train_sst -train_quora -train_sts -clf conv
-sst_weight_decay 9e-3 -para_weight_decay 1e-5 -sts_weight_decay 1e-2
-lr_lambda 0.55 -optim Adamax -sst_lr_multiplier 4 -para_lr_multiplier 5
-sts_lr_multiplier 3 -epochs 7
```

Individual results not recorded.

I.10 G6 "Yin Yang Steady Hand" 4

```
$python multitask_classifier.py -fine-tune-mode iterative -lr 1e-4
-use_gpu -amp -batch_size 64 -train_sst -train_quora -train_sts -clf conv
-sst_weight_decay 9e-3 -para_weight_decay 1e-5 -sts_weight_decay 1e-2
-lr_lambda 0.6 -optim Adamax -sst_lr_multiplier 2 -para_lr_multiplier 3
-sts_lr_multiplier 2 -epochs 7
```

1. **Sentiment classification accuracy:** 0.510
2. **Paraphrase detection accuracy:** 0.887
3. **Semantic Textual Similarity correlation:** 0.845

I.11 G6 "Yin Yang Steady Hand" 5

```
$python multitask_classifier.py -fine-tune-mode iterative -lr 1e-4
-use_gpu -amp -batch_size 64 -train_sst -train_quora -train_sts -clf conv
-sst_weight_decay 9e-3 -para_weight_decay 1e-5 -sts_weight_decay 1e-2
-lr_lambda 0.5 -optim Adamax -sst_lr_multiplier 5 -para_lr_multiplier 6
-sts_lr_multiplier 4 -epochs 7
```

1. **Sentiment classification accuracy:** 0.520
2. **Paraphrase detection accuracy:** 0.888
3. **Semantic Textual Similarity correlation:** 0.841

I.12 G6 "Yin Yang Steady Hand" 6

```
$python multitask_classifier.py -fine-tune-mode iterative -lr 1e-4
-use_gpu -amp -batch_size 64 -train_sst -train_quora -train_sts -clf conv
-sst_weight_decay 9e-3 -para_weight_decay 1e-5 -sts_weight_decay 1e-2
-lr_lambda 0.5 -optim Adamax -sst_lr_multiplier 6 -para_lr_multiplier 7
-sts_lr_multiplier 5 -epochs 7
```

1. **Sentiment classification accuracy:** 0.518
2. **Paraphrase detection accuracy:** 0.891
3. **Semantic Textual Similarity correlation:** 0.840

J G6 "Run it Back" Model Variant

Given the massively larger compute requirements of training full epochs on the Quora dataset, we proposed that we might get easy value with minimal added compute by re-shuffling and re-training the SST and STS tasks multiple times per epoch. As such, we implemented simple training loops to allow multiple rounds of training for each sub-task per-epoch when requested. Unfortunately, this experiment proved largely unsuccessful across all three tasks, with dramatic decrease in paraphrase accuracy (the task we did not multiplicatively train per epoch). The main model architecture here was derived from the "Yin Yang" architecture, with differences limited to training routine.

Command:

```
$python multitask_classifier.py -fine-tune-mode iterative -lr 1e-4
-use_gpu -amp -batch_size 64 -train_sst -train_quora -train_sts -clf conv
-sst_weight_decay 9e-3 -para_weight_decay 1e-5 -sts_weight_decay 1e-2
-lr_lambda 0.55 -optim Adamax -sst_lr_multiplier 4 -para_lr_multiplier
1.0 -sts_lr_multiplier 3 -epochs 7 -num_sst_trains 2 -num_quora_trains 1
-num_sts_trains 5
```

Results:

1. **Dev sentiment accuracy:** 0.494
2. **Dev paraphrase accuracy:** 0.653
3. **Dev STS correlation:** 0.839

K G6 "Interleaver" Model Variants and Ensemble

As opposed to the Duality of Man, Yin Yang, and Run it Back architectures, the Interleaver models derive their name from the fact that we interleave model training at a finer granularity. Whereas all other models are trained by iterating through the full SST Dataloader, then the Quora Dataloader, then the STS Dataloader, this model iterates through SST, Quora, and STS batch by batch, calculating each loss function and backpropagating in turn in a single batch iteration. While we observed inferior performance for the individual models (particularly on SST), the overall ensemble yields surprisingly good results. The main model architecture here was derived from the "Yin Yang" architecture, with differences limited to training routine.

```
# G6 interleaver
$python multitask_classifier.py --fine-tune-mode iterative --lr 1e-4
    --use_gpu --amp --batch_size 64 --train_sst --train_quora --
    train_sts --clf conv --sst_weight_decay 8e-3 --para_weight_decay 1
    e-5 --sts_weight_decay 9e-3 --lr_lambda 0.55 --optim Adamax --
    sst_lr_multiplier 4 --para_lr_multiplier 1.0 --sts_lr_multiplier 3
     --epochs 7 --num_sst_trains 5 --num_quora_trains 1 --
    num_sts_trains 5

# G6 interleaver more para
$python multitask_classifier.py --fine-tune-mode iterative --lr 1e-4
    --use_gpu --amp --batch_size 64 --train_sst --train_quora --
    train_sts --clf conv --sst_weight_decay 8e-3 --para_weight_decay 1
    e-5 --sts_weight_decay 9e-3 --lr_lambda 0.55 --optim Adamax --
    sst_lr_multiplier 3 --para_lr_multiplier 4 --sts_lr_multiplier 2
    --epochs 7 --num_sst_trains 4 --num_quora_trains 1 --
    num_sts_trains 4

# G6 interleaver fast
$python multitask_classifier.py --fine-tune-mode iterative --lr 1.5e-4
     --use_gpu --amp --batch_size 64 --train_sst --train_quora --
    train_sts --clf conv --sst_weight_decay 8e-3 --para_weight_decay 1
    e-5 --sts_weight_decay 9e-3 --lr_lambda 0.55 --optim Adamax --
    sst_lr_multiplier 5 --para_lr_multiplier 5 --sts_lr_multiplier 2
    --epochs 7 --num_sst_trains 3 --num_quora_trains 1 --
    num_sts_trains 3

# G6 interleaver steady hand
$python multitask_classifier.py --fine-tune-mode iterative --lr 1e-4
    --use_gpu --amp --batch_size 64 --train_sst --train_quora --
    train_sts --clf conv --sst_weight_decay 9e-3 --para_weight_decay 1
    e-5 --sts_weight_decay 1e-2 --lr_lambda 0.5 --optim Adamax --
    sst_lr_multiplier 4 --para_lr_multiplier 5 --sts_lr_multiplier 3
    --epochs 7 --num_sst_trains 5 --num_quora_trains 1 --
    num_sts_trains 5

# G6 interleaver the parabola
$python multitask_classifier.py --fine-tune-mode iterative --lr 1e-4
    --use_gpu --amp --batch_size 64 --train_sst --train_quora --
    train_sts --clf conv --sst_weight_decay 9e-3 --para_weight_decay 1
    e-5 --sts_weight_decay 1e-2 --lr_lambda 0.5 --optim Adamax --
    sst_lr_multiplier 3 --para_lr_multiplier 8 --sts_lr_multiplier 2
    --epochs 7 --num_sst_trains 2 --num_quora_trains 2 --
    num_sts_trains 2

# G6 interleaver 2
$python multitask_classifier.py --fine-tune-mode iterative --lr 1e-4
    --use_gpu --amp --batch_size 64 --train_sst --train_quora --
    train_sts --clf conv --sst_weight_decay 9e-3 --para_weight_decay 1
    e-5 --sts_weight_decay 1e-2 --lr_lambda 0.5 --optim Adamax --
    sst_lr_multiplier 4 --para_lr_multiplier 2.0 --sts_lr_multiplier 4
     --epochs 7 --num_sst_trains 10 --num_quora_trains 1 --
    num_sts_trains 10
```

```

# ensemble
$python ensemble.py --filepaths g6-interleaver.pt g6-interleaver-more-
    para.pt g6-interleaver-fast.pt g6-interleaver-steady-hand.pt g6-
    interleaver-the-parabola.pt g6-interleaver-2.pt
```

Listing 5: G6 Interleaver Shell Commands

G6 "Interleaver" Ensemble Results

Result:

1. **Sentiment classification accuracy:** 0.500
2. **Paraphrase detection accuracy:** 0.897
3. **Semantic Textual Similarity correlation:** 0.866

L Final Ensembles

L.1 G6 "Ménage à Onze" - Duality of Man + Yin Yang Ensemble

We ensemble all Yin Yang and Duality of Man variants with:

```
$python ensemble.py -filepaths g6-yin-yang.pt g6-yang-yin.pt
g6-yin-yang-fast.pt g6-yin-yang-steady-hand.pt g6-yang-yin-the-parabola.pt
g6-yin-yang-2.pt g6-duality-of-man.pt duality-of-fast-man.pt
duality-of-long-distance-runners-20.pt duality-of-man-nonlinear.pt
duality-of-slow-man.pt duality-of-robust-men.pt
```

Result:

1. **Sentiment classification accuracy:** 0.538
2. **Paraphrase detection accuracy:** 0.879
3. **Semantic Textual Similarity correlation:** 0.860

Other Ensembles Tried:

G6 "Oligarchy" v1 [First CS224N Class Test Leaderboard Submission]

This model is the same as G6 "Ménage à Onze" except that we now give Yin Yang Steady Hand (our most performant single sub-model) 3 votes.

```
$python ensemble.py -filepaths g6-yin-yang.pt g6-yang-yin.pt
g6-yin-yang-fast.pt g6-yin-yang-steady-hand.pt g6-yang-yin-the-parabola.pt
g6-yin-yang-2.pt g6-duality-of-man.pt duality-of-fast-man.pt
duality-of-long-distance-runners-20.pt duality-of-man-nonlinear.pt
duality-of-slow-man.pt duality-of-robust-men.pt g6-yin-yang-steady-hand.pt
g6-yin-yang-steady-hand.pt
```

Dev Result:

1. **Sentiment classification accuracy:** 0.536
2. **Paraphrase detection accuracy:** 0.882
3. **Semantic Textual Similarity correlation:** 0.860

Test Result:

1. **SST test accuracy:** 0.551
2. **Paraphrase test accuracy:** 0.881
3. **STS test correlation:** 0.853
4. **Overall test score:** 0.786

G6 "Oligarchy" v2

This ensemble is the same as Oligarchy v1 except we give Yin Yang, Yang Yin, Yang Yin the Parabola, and Duality of Robust Men models additional votes as well.

```
$python ensemble.py -filepaths g6-yin-yang.pt g6-yang-yin.pt
g6-yin-yang-fast.pt g6-yin-yang-steady-hand.pt g6-yang-yin-the-parabola.pt
g6-yin-yang-2.pt g6-duality-of-man.pt duality-of-fast-man.pt
duality-of-long-distance-runners-20.pt duality-of-man-nonlinear.pt
duality-of-slow-man.pt duality-of-robust-men.pt g6-yin-yang-steady-hand.pt
g6-yin-yang-steady-hand.pt g6-yin-yang.pt g6-yang-yin.pt
g6-yang-yin-the-parabola.pt duality-of-robust-men.pt
```

Result:

1. **Sentiment classification accuracy:** 0.530
2. **Paraphrase detection accuracy:** 0.883

3. **Semantic Textual Similarity correlation:** 0.860

G6 "Lisan al Gaib"

This ensemble includes one of (almost) everything, including additional slightly modified versions of Yin Yang Steady Hand. Kitchen-sink style ensemble. It sees across the dunes.

```
$python ensemble.py -filepaths g6-yin-yang.pt g6-yang-yin.pt
g6-yin-yang-fast.pt g6-yin-yang-steady-hand.pt g6-yang-yin-the-parabola.pt
g6-yin-yang-2.pt g6-duality-of-man.pt duality-of-fast-man.pt
duality-of-long-distance-runners-20.pt duality-of-man-nonlinear.pt
duality-of-slow-man.pt duality-of-robust-men.pt g6-interleaver.pt
g6-interleaver-more-para.pt g6-interleaver-fast.pt g6-interleaver-steady-hand.pt
g6-interleaver-the-parabola.pt g6-interleaver-2.pt g6-yin-yang-steady-hand-2.pt
g6-yin-yang-steady-hand-3.pt g6-yin-yang-steady-hand-4.pt
g6-yin-yang-steady-hand-5.pt g6-yin-yang-steady-hand-6.pt
```

Result:

1. **Sentiment classification accuracy:** 0.530
2. **Paraphrase detection accuracy:** 0.891
3. **Semantic Textual Similarity correlation:** 0.864

G6 "The Sage"

This ensemble combines primarily known high-quality sub-models, with a handful of weaker performers. It does not perform better than the kitchen sink model, suggesting that simply including more models is more optimal.

```
$python ensemble.py -filepaths g6-yin-yang.pt g6-yang-yin.pt
g6-yin-yang-fast.pt g6-yin-yang-steady-hand.pt g6-yang-yin-the-parabola.pt
g6-yin-yang-2.pt g6-duality-of-man.pt duality-of-man-nonlinear.pt
duality-of-slow-man.pt duality-of-robust-men.pt g6-interleaver-steady-hand.pt
g6-yin-yang-steady-hand-2.pt g6-yin-yang-steady-hand-3.pt
g6-yin-yang-steady-hand-4.pt g6-yin-yang-steady-hand-5.pt
g6-yin-yang-steady-hand-6.pt
```

Result:

1. **Sentiment classification accuracy:** 0.529
2. **Paraphrase detection accuracy:** 0.889
3. **Semantic Textual Similarity correlation:** 0.860

M Efficiency Experiments Details

M.1 Grid Search Details

Parameter	**Values**
AMP	On
Learning Rate	1×10^{-4}, 5×10^{-5}, 1×10^{-5}, 5×10^{-6}
Batch Size	64, 128, 256, 384
LoRA Rank	1, 5, 10
LoRA Mode	'none', 'all-lin', 'attn', 'all-lin-only', 'attn-only'
DoRA	Off, On
Fine-Tune Mode	'full-model', 'last-layer'
Epochs	3

Table 4: Grid search parameters for evaluating the impact of LoRA and DoRA.

M.2 LoRA and DoRA Over Many Ranks

This grid search generated a total of 960 possible combinations. However, only 416 of these combinations are feasible, as certain parameter configurations are logically incompatible (e.g., setting 'LoRA Mode' to 'none' while varying 'LoRA Ranks' is illogical since LoRA is deactivated). Out of these, only 270 could be executed to completion due to memory constraints. For instance, configurations with 'Fine-Tune Mode' set to 'full-model' and 'LoRA Mode' set to 'none' could not support batch sizes above 128. We restricted our training and evaluation to the SST and STS tasks, as the Paraphrasing task required significantly more computational resources.

We ran both LoRA and DoRA for ranks 1,2,5,10,20,50,80,100 with the hyperparameters outlined in Table 5. The resulting normalized Memory, Time and Accuracy values are presented in Figure 20.

Table 5: Hyperparameters

Hyperparameter	Value
Batch Size	192
Learning Rate	0.0002
Epochs	10
AMP	On
Dropout Probability	0.3

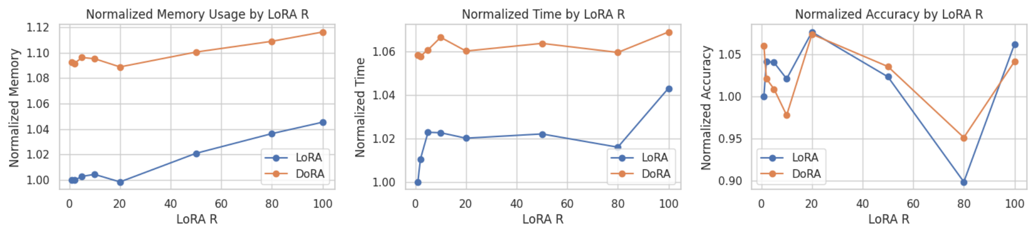

Figure 20: All values normalized to the value for LoRA and Rank 1

M.3 Ablation Experiment Details

Parameter	Values
AMP	Off, On
Learning Rate	$1 \times 2 \times 10^{-4}$
Batch Size	96
LoRA Rank	1
LoRA Mode	'none', 'attn', 'attn-only'
DoRA	Off, On
Fine-Tune Mode	'full-model', 'last-layer'
Epochs	10

Table 6: Grid search parameters for evaluating the impact of LoRA and DoRA.

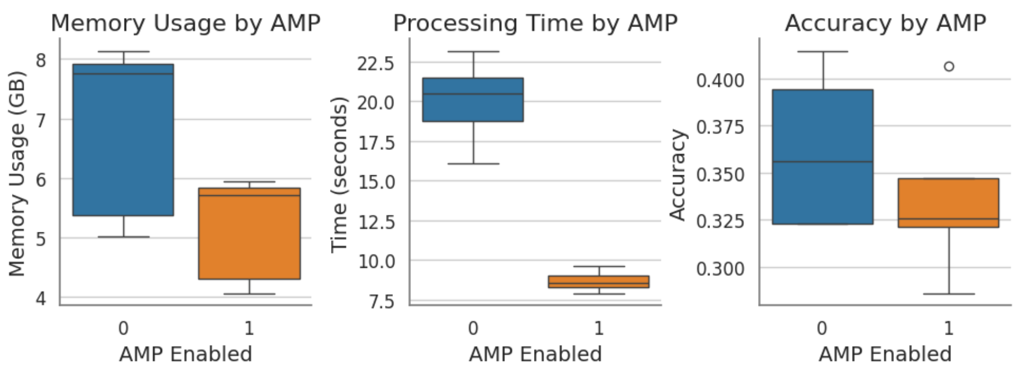

Figure 21: Comparison of Memory Usage, Processing Time, and Accuracy With and Without AMP

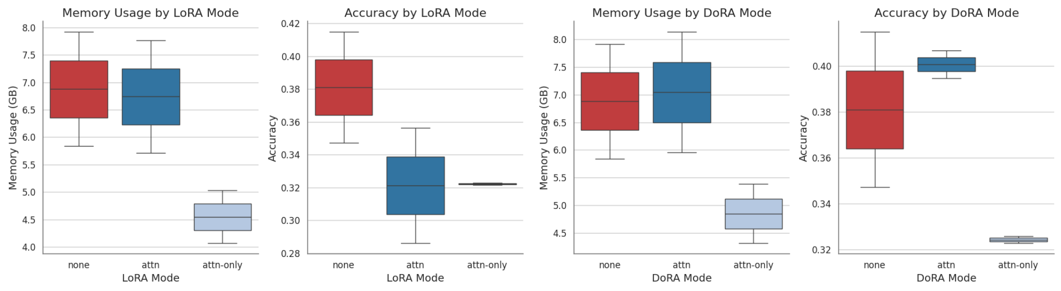

Figure 22: Comparison of Memory Usage and Accuracy Across LoRA/DoRA Modes

M.4 Final Model Hyperparameters

Table 7: Hyperparameters used for benchmarking the final model configurations.

Hyperparameter	Value
Fine-tune mode	Iterative
Learning rate (lr)	1×10^{-4}
Batch size	64
Tasks	SST, Quora, STS
Classifier (clf)	Convolutional
SST weight decay	9×10^{-3}
Paraphrase weight decay	1×10^{-5}
STS weight decay	1×10^{-2}
Learning rate lambda (lr_lambda)	0.5
Optimizer (optim)	Adamax
SST learning rate multiplier	4
Paraphrase learning rate multiplier	5
STS learning rate multiplier	3
Epochs	7

N Data Preprocessing Experiments

In addition to the default string processing included in the handout, we experimented with a modified `datasets.py` using the Bird et al. (2009) library to remove stopwords, lemmatize, remove all punctuation, and replaced contractions such as "n't" with "not" and "'ll" with "will". Stopwords are words with low discrimination power, such as 'a', 'an', 'the', 'and', etc and their removal has been shown to increase performance on text classification tasks Ladani and Desai (2020). Lemmatization involves breaking a word down into its root meaning (for instance, "best" to "good") and has been shown to improve performance in sentiment analysis tasks Khyani and B S (2021). We hypothesized also that contraction removal could help with text standardization and potentially improve subtask performance. Overall, we did not observe significant differences between our various preprocessing pipelines when applied to model training with our final model architecture (subsection 4.2). Thus, we stuck with only basic white-space removal and special character removal for the models included in our final ensembles (Appendix L).

O Model Parameter Count

We computed each of our minBERT backbones to contain $\approx 110M$ trainable parameters, and our full set of convolution classifiers to contain $\approx 6M$ trainable parameters. Thus, for the architecture described in subsection 4.2, our model consists of $\approx 226M$ parameters.