ExpLoRA: Exploring LoRA, SMART, and SOAP to Finetune GPT-2 for Downstream Tasks

Stanford CS224N Default Project

Mia Penfold
Department of Computer Science
Stanford University
`penfold@stanford.edu`

Olivia Bruvik
Department of Computer Science
Stanford University
`oliviabb@stanford.edu`

Sidharth Srinivasan
Department of Computer Science
Stanford University
`s1dharth@stanford.edu`

Abstract

We work to implement core components of GPT-2, including masked multi-head attention, transformer layers, and the AdamW optimizer, successfully matching baseline performance on sentiment analysis tasks. Given this implementation is a large pre-trained model, it handles a wide variety of tasks such as detection and sonnet generation. That being said, due to its very large parameter count, it's a less effective method for fine-tuning to specific downstream tasks. In particular, for each downstream task, there would be a large finetuning effort, even though only a small subset of the weights need to be tweaked. To address this, we implemented Low-Rank Adaptation (LoRA), SMART regularization, and SOAP optimization. These three techniques are helpful as they experiment with reducing memory usage and accelerating training while maintaining model performance for downstream applications. This work combines practical implementation of transformer architectures with modern efficiency techniques, providing insights into making large language models more accessible for distributed tasks. Ultimately, we were able to achieve above the baseline for sonnet generation and at par with baseline for paraphrase detection, while achieving lower computational needs.

Key Information to include: Mentor: n/a; External Collaborators: n/a; Sharing project: n/a

1 Introduction

Large language models (LLMs) provide a powerful tool to understand and generate natural language, applicable to an ample range of tasks. However, adapting these LLMs to specialized tasks—and doing so efficiently—remains a challenge: traditional methods are often computationally expensive, prone to overfitting.

In this project, we implement GPT-2 from scratch and apply it to two downstream tasks: sonnet generation and paraphrase detection. We test our baseline implementation with traditional, "vanilla" fine-tuning and attempt to tackle common fine-tuning challenges by adding Low-Rank Adaptation (LoRA), Smoothness-Inducing Adversarial Regularization (SMART), and a SOAP (Shampoo with Adam in the Preconditioner) optimizer. We will pay particular attention to training efficiency, i.e., the number of trained parameters, and model performance, i.e., the model's ability to recognize paraphrases and generate structured, high-quality poetry. We assess model performance using accuracy scores for paraphrase recognition and the chrF metric for poetry generation quality.

We first collected a baseline score for a "vanilla" fine-tuned GPT-2 on paraphrase detection and sonnet generation tasks. We then compared this baseline with fine-tuning using a variety and combination of techniques, including LoRA, SMART, SOAP, and generation techniques.

2 Related Work

Traditional methods for fine-tuning large language models face significant limitations. Training all model parameters results in high computational costs, while the scarcity of domain-specific datasets often leads to overfitting. As massive models like Grok (314 billion parameters), Claude (175 billion), and GPT-4 (1.76 trillion) become the norm, traditional fine-tuning grows increasingly impractical. To address these challenges, researchers are developing parameter-efficient tuning techniques and regularization strategies that enhance generalization. Our project mainly focuses on the following two methods.

Parameter-efficient fine-tuning (PEFT): One important method of parameter-efficient fine-tuning is Low-Rank Adaptation (LoRA), introduced by Hu et al. (2021) [1]. LoRA freezes the pre-trained model weights and introduces low-rank decomposition matrices into the Transformer layers, significantly reducing the number of trainable parameters with a 10,000x reduction achieved. Similarly, LoRA also reduces memory requirements by up to 3× compared to full fine-tuning, while maintaining comparable or superior task performance across various NLP benchmarks. LoRA introduces no additional inference latency, unlike adapter layers and other PEFT methods.

Regularization: The second key challenge is getting a model to generalize well despite small dataset sizes. Smoothness-Inducing Adversarial Regularization for Fine-Tuning (SMART), proposed by Jiang et al. (2021) [2] is a method of preventing overfitting and ensuring robust generalization. The method, meant to help models generalize in low-resource settings, combines adversarial regularization with Bregman proximal point optimization. SMART trains the model to ensure that small perturbations in the input do not cause drastic output changes. This encourages local smoothness and thus explicitly controls model complexity. Additionally, Bregman proximal optimization prevents aggressive parameter updates, leading to more stable fine-tuning and better generalization to unseen data. SMART has demonstrated improved performance on multiple NLP benchmarks, outperforming large-scale models like T5 (Raffel et al., 2020) [3] on the GLUE benchmark.

SOAP Optimization: We also implement Shampoo with Adam in the Preconditioner's eigenbasis (SOAP), which is a computationally efficient variant of Shampoo that maintains its preconditioning benefits while significantly reducing iteration count and the time to train a model. Unlike AdamW, which only updates running averages of first- and second-moment quantities, SOAP operates in a rotated space, leading to over 35% time savings and 20% performance improvements over Shampoo in large-batch training, making it a more effective choice for our language model pre-training. [4]

Top-K Generation: Originally introduced in Hierarchical Neural Story Generation (Fan et al., 2018) [5], top-k generation enhances text fluency and coherence by restricting token selection to the k most probable choices at each step. We use a multinomial sampling of these choices (which is original to our custom implementation, i.e. not in the original paper).

3 Approach

Our approach integrates LoRA fine-tuning, SMART regularization, a SOAP Optimizer and top-k generation to improve paraphrase detection and sonnet generation inspired by the GPT-2-based Transformer architecture, illustrated in Figure 2 in the Appendix, as described in the CS224N Default Final Project Handout [6]. Below, we detail our model design, training techniques, and baselines.

3.1 Baseline Model Architecture: GPT-2 with AdamW

We build upon the decoder-only Transformer architecture introduced in GPT-2 (Radford et al., 2019) [7]. Our implementation uses a trainable embedding layer with byte pair encoding (BPE) tokenization to efficiently handle subword units. Our decoder-only transformer consists of 12 layers with:

- **Masked Multi-Head Self-Attention** to allow the model to jointly attend to information from different representation subspaces at different positions. We compute attention by taking the weighted sum of the value and the softmax of the dot products of the keys and queries.

- **Position-wise Feed-Forward Layers (MLP with ReLU activations)**
- **Layer Normalization & Residual Connections** to stabilize training.

The model is trained to predict a “next token" x_i based on a sequence of input tokens $\{x_1, x_2, ..., x_n\}$, it produces a probability over the vocabulary of all x_is such that :

$$P(x_i|x_1, x_2, ..., x_{i-1}) = \texttt{softmax}(W_o h_i) \tag{1}$$

h_i represents the last hidden state, and W_o is the output projection weight matrix.

Optimizer: We implemented an AdamW optimizer. Like Adam, AdamW only requires first-order gradients for stochastic optimization and uses an adaptive learning rate for different parameters. The algorithm updates exponential moving averages of the gradient m_t and the squared gradient v_t, using hyperparameters $\beta_1, \beta_2 \in [0, 1)$ to control the rate of exponential decay. We calculate $\hat{m}_t$ and $\hat{v}_t$ by using bias correction on these moving averages, which are biased towards zero. Unlike Adam, AdamW applies weight decay after the main gradient-based updates, leading to more effective regularization.

3.2 Vanilla fine-tuning model on domain specific tasks

We used a “vanilla” full-model fine-tuning to adapt the model for sonnet generation and paraphrase detection and achieved a 27.371 chrF score and 0. 896% accuracy, respectively, on the dev sets. We used the default hyper-parameters from the given code. We also added a feature where the model will stop early if it sees performance start to degrade. We knew that the extensions we planned to implement ran the risk of overfitting and decreasing in quality with subsequent epochs.

3.3 Additional methods of fine-tuning model on domain specific tasks

3.3.1 LoRA: Low-Rank Adaptation

LoRA (Hu et al., 2021)[1] enables efficient fine-tuning by freezing the pre-trained pre-trained weight matrix $W_0 \in R^{d\times k}$ and injects trainable rank decomposition matrices into each layer of the Transformer architecture. Instead of updating all parameters, LoRA represents the weight updates ΔW as low-rank decomposition.

$$\Delta W = BA, \quad B \in \mathbb{R}^{d\times r}, \quad A \in \mathbb{R}^{r\times d} \tag{2}$$

Such that the output of a linear transformation h:

$$h = W_0 x + \Delta W x = W_0 x + BAx \tag{3}$$

where $r \ll d$ (common rank values r are 1, 2, 4, 8, 16 and 32). This drastically reduces the number of trainable parameters while preserving pre-trained weights—and therefore the knowledge contained by them. We apply LoRA to the query, key and value projections of the self-attention layers.

3.3.2 SMART Regularization

To improve generalization, we apply SMART [2]. In our model, SMART works by adding a regularization term to the standard cross-entropy loss to improve generalization. We do this by slightly perturbing our embeddings before passing them through or model to measure how much the model’s outputs change under small variations. These perturbations $\tilde{x}_i$ are creating by adding a small amount of noise to the input embeddings x_i. These are then optimized to find the worst-case perturbation, to ensure that $\|\tilde{x}_i - x_i\|_p \leq \epsilon$. The regularization term $R_s(\theta)$ is computed using Bregman Proximal Point Optimization (BPPO) as:

$$R_s(\theta) = \frac{1}{n} \sum_{i=1}^{n} \max_{\|\tilde{x}_i - x_i\|_p \leq \epsilon} \ell_s(f(\tilde{x}_i; \theta), f(x_i; \theta))$$

where ℓ_s is the symmetrized KL-divergence. The final loss combines the SMART loss and cross-entropy loss:

$$\text{Loss} = \text{CrossEntropyLoss} + \lambda_s R_s(\theta)$$

where λ_s controls the strength of the regularization. This method stabilizes fine-tuning, preventing overfitting and making the model more robust to small variations in input. To implement this we build on the smart-pytorch library implementation and correct it to better represent the equations in the paper. Notably, instead of normalizing noise by dividing the step by `step_norm + self.epsilon`, we now divide by `step_norm + 1e-8` the original approach (+ self.epsilon) could overly shrink the perturbation. The updated approach better aligns with the paper since it explicitly ensures controlled perturbation within ϵ. Our custom Pytorch performed better than the smart-pytorch library.

3.3.3 SOAP Optimization:

We also added SOAP, which continually updates the running average of the second momentum like Adam, but in the current coordinate basis [4]. It does so by integrating Adam's adaptive learning rate mechanism with the original Shampoo optimization method. Shampoo approximates the curvature using a preconditioner $P_t = (H_t + \epsilon I)^{\frac{-1}{4}}$, where H_t is a block-diagonal approximation of the curvature and ϵ is a damping factor. SOAP simplifies the Adam updats by aligning Shampoo's preconditioner P_t with Adafactor's factorization eigenbasis and integrates Adam's momentum and variance terms. The update becomes $\theta_{t+1} = \theta_1 - k\frac{P_t m_t}{\sqrt{v_t}+\epsilon}$ instead of the original update rule of $\theta_{t+1} = \theta_1 - k\frac{m_t}{\sqrt{v_t}+\epsilon}$.

We used the original implementation of the SOAP optimizer by the main author Nikhil Vyas with Adam's beta parameters of $b_1, b_2 = 0.95, 0.95$, an Adam's epsilon of $1e-8$, and bias correction in Adam. We did not normalize gradients per layer, and a weight decay coefficient of 0.01, and precondition frequency of 10.

3.3.4 Sonnet generation variations:

Top-k sampling: The base implementation for GPT-2 involves setting a cutoff probability, normalizing the tokens that are above the cutoff, then sampling. We expanded upon this by adding top-k as a sampling choice during sonnet generation. We added a top_k parameter with a default of 3. The implementation is identical to that of top-p, except we keep only the k highest probability tokens, renormalize the sum to 1 within the smaller set, and sample from the k most likely next tokens. We also added a small epsilon of $1e-10$ to avoid division by zero.

Beam search: We also implemented beam search with both top-k and top-p sampling, an algorithm to choose the best generated token based on the cumulative probability of all tokens in each sequence.

Iterative generation: Following subpar results from the beam search, we added another implementation of the sonnet generation where we generated a set of sonnets and chose the sonnet with the highest cumulative probability.

4 Experiments

4.1 Tasks, Data and Evaluation Methods

4.1.1 Task #1: Sonnet Generation

We finetune GPT-2 to generate a Shakespearean sonnet, given the first three lines.

- **Dataset**: We use 154 Shakespearean sonnets, of 14 lines each with an ABAB CDCD EFEF GG rhyme scheme. The Sonnet dataset has the following train/dev/test split: (124, 15, 15 sonnets).
- **Training Objective**: The model is trained using causal language modeling: $P(x_i | x_{<i})$
- **Loss Function**: Cross-entropy loss applied over the entire sequence: $L = -\sum_i y_i \log P(x_i)$
- **Evaluation:** chrF Score, measures character-level n-gram overlap between our generated sonnet and the test sonnet.
- **Generation:** Top-p (nucleus) sampling with temperature scaling is used by our baseline model (we later try to improve upon this with top-k sampling)

4.1.2 Task #2: Paraphrase Detection

The paraphrase detection task is formulated as a cloze-style problem, where GPT-2 generates “yes” or “no” given a pair of sentences.

- **Dataset**: We finetune on the Quora dataset, which contains labeled paraphrase pairs.The Quora dataset has the following train/dev/test split: (141,506, 20,215, 40,431 question pairs)
- **Training Objective**: GPT-2 is trained to generate the correct label, mapping question pairs as paraphrases of each other (“yes" $\rightarrow$ token ID 8505) or not (“not" $\rightarrow$ token ID 3919):
- **Loss Function**: Cross-entropy loss applied over last token: $L = -\sum_i y_i \log P(x_i)$
- **Evaluation:** We used accuracy, the percentage of correctly classified pairs.

4.2 Experimental details

After implementing LoRA and SMART on our baseline model, we tuned our hyperparameters using grid-search to achieve the best possible chrF score on the sonnet task. We focused on tuning on this task rather than on paraphrase detection because it only took 15-20 minutes to finetune, compared to 4-8 hours for paraphrase on NVIDIA T1, M4 Pro, and M3 Max chips, allowing for quicker experiments.

4.2.1 Tuning LoRa and SMART hyperparameters on Sonnet Generation task

SMART: We finetune the model for 10 epochs with a batch size of 8. We used grid search to tune hyperparameters. To reduce the possible permutations we chose to keep the variance `smart_var` constant at $\sigma^2 = 1 \times 10^{-5}$. We use ϵ (`smart_eps`) values of 1×10^{-6} and 1×10^{-5}, and set η (`smart_step`) to 0.01 or 0.001. We experimented with the regularization parameter λ_s (`smart_lambda`) taking on values of 0.0009, 0.001, 0.002, 0.003, 0.004, 0.005, 0.01, 0.02 or 0.05, while exploring learning rates of 0.005 and 0.001.

LoRA: We finetune a model on 10 epochs and a batch size of 8. For experimentation, we tried a variety of rank, alpha, and dropout rate and found the best combination using grid search. Given the quantity of combinations available, we started by running experiements with a fixed rank (r=2) and scale = 0.5, 0.25, 1, 2, dropout = 0.1, 0.05, and learning rate = 0.001, 0.0005. These were chosen from literature review and our own early testing. Once finishing rank=2, we determined that a scale of 1 or 2 and learning rate of 0.001 was consistely best (higher chrF score). Thus, we proceeded to run rank=2, 4, 8, 16, 32, 64 with these set parameters.

LoRA + GPT model size: We also experimented with using different size gpt models, including the standard and gpt-large, holding the lora rank at 2, lora scale at 2, lora dropout at 0.05, epochs at 10, learning rate at 0.0001, and top-p at 0.98.

SOAP + sonnet generation variations: We ran >50 experiments finetuning the model with SOAP with a variety of hyperparameters. We ran a grid search to find the best hyperparameters with epochs = $\{5, 10, 20\}$, batch size = $\{2, 4, 8, 16, 32\}$, and learning rate = $\{0.00001, 0.0001, 0.001, 0.003\}$. We also included variations of the Sonnet generations, including top-p sampling with top_p = $\{0.9, 0.93, 0.94, 0.95, 0.98\}$, top-k sampling with top_k = $\{1, 3, 4, 5, 10, 20\}$, beam search with number of beams = $\{1, 3, 5\}$, and multiple sonnet generations with the number of generations = $\{3, 5\}$.

SMART + LoRA: We use grid search to find the best hyperparameters for using both SMART and LoRA We experiment with λ_s (`smart_lambda`) values of 0.0001, 0.001, 0.01, and 0.02. We set ϵ (`smart_eps`) to 1×10^{-6}, σ^2 (`smart_var`)to 1×10^{-5} andη (`smart_step`) to 0.001. We investigate LoRA ranks of 2, 4, 8, and 32, while keeping the LoRA scale fixed at 1 and LoRA dropout at 0.05. We kept a constant learning rate a learning rate of 0.001 and train the model for 10 epochs with a batch size of 8.

SMART + LoRA + SOAP: We use grid search to find the best hyperparameters for using SMART and LoRA and SOAP in combination. We experimented with learning rates = $\{0.0001, 0.001, 0.003\}$, epochs = $\{10, 20\}$, top-p = $\{0.9, 0.98\}$, batch size = $\{4, 8, 16\}$, while keeping the LoRA scale fixed at 1, LoRA rank at 8, and LoRA dropout at 0.05. We also kept the smart hyperparameters fixed at `smart_lambda` = 0.004, λ_s (`smart_lambda`) = 0.004, ϵ (`smart_eps`) = 1×10^{-5}, σ^2 (`smart_var`) to 1×10^{-5} andη (`smart_step`) to 1×10^{-3}.

4.3 Results

Our qualitative dev chrF scores are below in Table 1. We used this to get an indication of which hyperparameters would perform the best for the leaderboard.

Table 1: Sonnet SMART					**Table 2: Sonnet LoRA**				**Table 3: Sonnet SMART + LoRA**		
λ	step	eps	noise	chrF	rank	scale	dropout	chrF	λ	rank	chrF
4e-3	**1e-3**	**1e-5**	**1e-5**	**41.957**	**2**	**2**	**0.05**	**42.255**	**1e-4**	**8**	**41.586**
1e-3	1e-3	1e-5	1e-5	41.909	32	1	0.05	42.111	1e-3	32	41.367
1e-3	1e-3	1e-6	1e-5	41.643	2	1	0.05	42.094	1e-3	8	41.223
1e-3	1e-2	1e-5	1e-5	41.310	8	2	0.05	41.936	1e-4	32	40.813
9e-4	1e-3	1e-5	1e-5	41.293	16	1	0.05	41.850	1e-4	4	40.763
2e-3	1e-3	1e-5	1e-5	41.233	16	2	0.05	41.844	1e-3	2	40.192
3e-3	1e-3	1e-5	1e-5	41.200	64	1	0.1	41.831	1e-4	2	39.665
5e-3	1e-2	1e-6	1e-5	40.407	4	2	0.1	41.824	1e-3	4	38.990
1e-2	1e-3	1e-5	1e-5	41.003	64	2	0.1	41.776	2e-2	32	36.782
5e-3	1e-3	1e-5	1e-5	40.624	2	2	0.1	41.692	1e-2	32	35.884

Table 1: These are the top 10 best-performing runs for each grid search with the highest performer at the top. The learning rate was 0.001, the batch size was 8, and there were 10 epochs. The rest have been omitted.

Table 2: Ten Best Results with SOAP Enabled

Accuracy (%)	Batch Size	Learning Rate	Sampling Method	Sampling Value
42.48	4	3e-3	Top-p	p = 0.98
42.04	8	3e-3	Top-p	p = 0.90
42.02	4	3e-3	Top-p	p = 0.97
41.97	4	3e-3	Top-p	p = 0.95
41.85	8	3e-3	Top-p	p=0.90
41.83	8	3e-3	Top-k	k = 4
41.80	8	3e-3	Top-k	k = 4
41.80	4	1e-3	Top-p	p = 0.98
41.70	8	3e-3	Top-k	k = 3
41.70	8	3e-3	Top-k	k = 3

After isolating the best hyperparameters, we wanted to great an indication of which combination of extensions on our base model would perform the best. Table 2 gives us an indication of the best performing sonnet generations, and Table 3 gives us the same for paraphrase detection. Note that due to the duration needed for paraphrase detection, we were not able to run the same level of experiementation.

With these dev results in mind, we decided to submit to the leaderboard for sonnet generation with only SOAP and had a chrF score of 42.48. For paraphrase detection, we submitted the vanilla finetuned method and got 89.6% accuracy.

We expected our results to give us significant improvements in performance because in their corresponding papers, these methods often result in considerable improvements on the baseline. We understand, however, that we were fine-tuning models on limited data and did not have the computational resources to tune hyperparameters as much as we would have wanted (especially for paraphrase detection). We were happy to see that despite these limitations, we did manage to improve upon our baseline results. Our many takeaway however, is that these methods can provide important improvements in terms of efficiency. While only training approx. 0.1% of parameters, LoRA improves upon our baseline. On the paraphrase detection task, SMART is able to produce comparable results (within 0.2% of baseline) with only 2 epochs of training compared to 10 for the baseline.

5 Analysis

Our base implementation of GPT-2 provided us with with a strong baseline and gave us an idea of the outpute output should look like, especially for the downstream tasks that wlike to fine-tune.ine-tune.

	Sonnet Generation (chrF score)
“Vanilla” fine-tuning (baseline)	41.584
SMART Library	41.957
SMART Custom	42.212
LoRA	42.255
SOAP	**42.48**
Top-K + "multiple generations"	41.81
Top-K + single generation	41.26
Top-P (P = 0.98) + "multiple generations"	42.06
SMART + LoRA	41.586
SMART + SOAP	41.41
LoRA + SOAP	41.29
Top-K + SOAP	41.83
SMART + LoRA + SOAP (Top - P)	41.58
SMART + LoRA + SOAP + Top-K	40.52
Top-P + SOAP	**42.48**
Top-K + SOAP	41.83

Table 3: These are the best-performing runs for sonnet generation. The batch size was 8 and there were 10 epochs. Bold shows the best.

	Paraphrase Detection (Accuracy)
“Vanilla” fine-tuning (baseline)	**0.896**
SMART Library	0.894 **(on epoch 2)**
LoRA	0.848
SMART + LoRA + SOAP (Top - P)	0.81

Table 4: These are the best-performing runs for paraphrase detection. Bold shows the best.

We were pleasantly surprised to see the results carry over when adding LoRA, with the benefit of training only a small fraction of the parameters (in our experiements, from 0.1% to 2%). It was interesting to see that changing the ranks from 2 all the way to 64 had a similar effect, when all other parameters were kept constant. This is likely because any large parameter reduction will still hone in on the few weights that need to be finetuned for this downstream task. In other words, the weights needed for this task fall into the total the trainable parameters under all tested ranks.

We also added SMART given that we were concerned only LoRA would introduce the problem of overfitting. If only finetuning a few weights, we might not be able to generalize from the small testing sonnets to the test ones. To combat this, SMART adds random pertubations to the embeddings to ensure the model can better generalize. We found SMART works best when using a small λ_s value (specificaly 4e-3). This was just an order of magnitude lower of the library we consulted (λ_s=0.02 on Huggingface RoBERTa classifier) [8], but significantly smaller than the values explored in the original SMART paper [2] (λ_s=1, 3, 5, SMART added to RoBERTa_LARGE). This is likely because we were training a much smaller model (custom GPT-2) on a very limited dataset, producing smaller losses which required the regularization loss R_s that we added to scale proportionally, resulting in a smaller lamdba.

The next addition we added was SOAP optimization. This extension is a preconditioning method that has been proven to be effective [**?**]. We found that SOAP was most effective with a learning rate of $3e-3$, which is the default. While the authors of SOAP noted that the method is more effective for larger batch sizes, we found that batch sizes of 4 and 8 were most effective for sonnet generation. We hypothesize that this is likely due to the limited size of the sonnet training set. We attempted to finetune the model on the paraphrase task using batch sizes of 32 and 64 to achieve the expected performance improvements with the larger paraphrase dataset, but the results were subpar. We found that adding SOAP improved the chrF scores by a full point for a subset of hyperparameters. Our best result obtained for the Sonnet generation was achieved using SOAP with a batch size of 4, a learning rate of 3e-3, and top sampling with 0.98.

We also added variations to the sonnet generation function. We added top-k generation, for which instead of having a set threshold cutoff for the distribution of next token, we opted to create a distribution that samples from the top k in probability. We did test beam search briefly and saw the sonnets had multiple repeated phrases. After researching the problem, we saw that this was because repeated patterns are favored in probabilities and choosing the k utmost highest probability tokens would lead to a version of overfitting. As such, we opted for a version closer to top k as we found the randomness that accompanies a distribution over the most probable tokens to be better. We also tried generating multiple sonnets and picking the best one by the average probability of all tokens, but the result improvements were marginal at the expense of compute time increases, so we reverted back to only generating a single sonnet. The best improvements were achieved by tuning the top_p hyperparameter, with the best $top_p = 0.98$.

6 Conclusion

To start, when thinking about which extensions to build, we realized that teams often think about computational needs in addition to performance, two metrics often come at the expense of the other. As such, we choose LoRA as an extention to see how much we could shift the parameter requirements with sustaining final results. To our surprise, we were able to achieve a similar performance as the vanilla fine-tuned method with less than 2% of the parameters as the original, large, pre-trained model. This was the case for both sonnet generation and paraphrase detection, emphasizing that when choosing a specific downstream task, the actual number of weights needed to be finetuned is much smaller than the total number of parameters. Additionally, time can be considered a computational need that a team might want to minimize. By replacing Adam with SOAP, we noticed that the model was able to converge much faster and achieve similar results.

The second big takeaway was that different fine-tuning methods can be better suited to different tasks. Sonnet generation depends on selecting the best next token whereas paraphrase detection involves a yes or no answer. As such, we noticed a domain where SMART, for example, was exceedingly successful. SMART is a regularization technique. When we first set out to test fine-tuning with SMART, we expected it's addition to improve our chrF and accuracy scores for both tasks, since it would help the model generalize better to different inputs. Though SMART did help us achieve marginally better results on sonnet generation after tuning hyperparameters, we found that it was an even more helpful addition to Paraphrase detection, where it was able to achieve a dev accuracy of 0.894 (-0.02 from our vanilla baseline) on only two epochs. This is likely because the paraphrase detection task requires the model to look at two sentences with very different syntax and decide whether they are semantically identical. For this task, we would want to have a model that is insensitive to small variations in wording and word order that do not affect meaning; incorporating noise into the sentence embeddings therefore proves an effective way to help the model not only generalize and therefore achieve better accuracies, but do so quickly. SMART was not as effective in speeding up sonnet generation training. This is likely because the sonnet generation task is preconditioned on existing sonnet lines and the chrF scores results based on the n-gram similarity to the test. This means that, in this task, generalizing well to variations in input is less critical.

Given we tried so many extensions, it begs a limitation where we now want to understand which precise combinations of these methods succeed for a given task and team goal. With more resources (given the extensive time needed to train some of these models), our team would love to explore building a robust training harness to evaluate the performance of different versions of our product under various tasks. With this, we could more effectively route tasks to the right model for a situation.

While previous studies have evaluated these techniques independently, our work systematically compares their effectiveness when used alone versus in combination. Our findings contribute to the ongoing discussion on how to best adapt LLMs for specialized NLP tasks while balancing efficiency, robustness, and performance.

Team contributions

All members worked on the base implementation and writing of papers. On top of that, Mia did SMART, Olivia did top-k and SOAP, and Sidharth did LoRA.

References

[1] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models, 2021.

[2] Haoming Jiang, Pengcheng He, Weizhu Chen, Xiaodong Liu, Jianfeng Gao, and Tuo Zhao. Smart: Robust and efficient fine-tuning for pre-trained natural language models through principled regularized optimization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, 2020.

[3] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer, 2023.

[4] Nikhil Vyas, Depen Morwani, Rosie Zhao, Mujin Kwun, Itai Shapira, David Brandfonbrener, Lucas Janson, and Sham Kakade. Soap: Improving and stabilizing shampoo using adam. 2025.

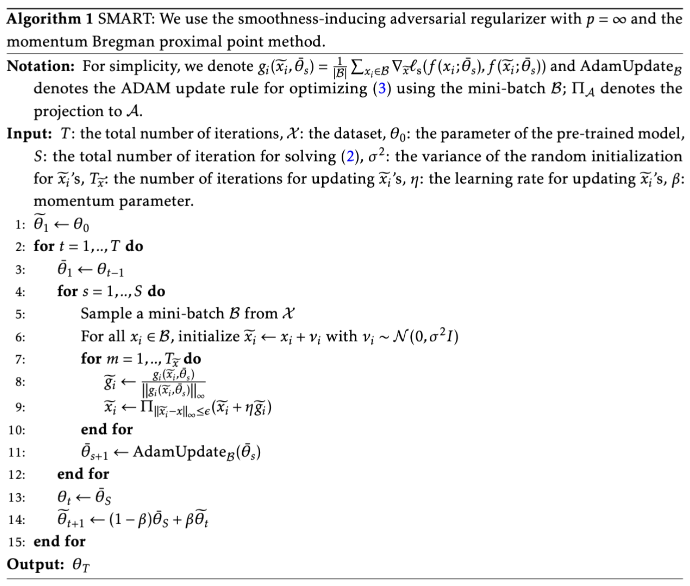
[5] Angela Fan, Mike Lewis, and Yann Dauphin. Hierarchical neural story generation, 2018.

[6] CS 224N Teaching Team. Build gpt-2 - cs 224n default final project handout. In *CS 224N Final Project*, 2025.

[7] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. In *OpenAI Research*, 2019.

[8] ArchinetAI. Smart: Smoothness-aware minimization for efficient adversarial training. `https://github.com/archinetai/smart-pytorch/tree/main`, 2025. This is the library we used to implement SMART.

A Appendix (optional)



Algorithm 1 SMART: We use the smoothness-inducing adversarial regularizer with $p = \infty$ and the momentum Bregman proximal point method.

Notation: For simplicity, we denote $g_i(\widetilde{x}_i, \bar{\theta}_s) = \frac{1}{|\mathcal{B}|}\sum_{x_i \in \mathcal{B}} \nabla_{\widetilde{x}} \ell_s(f(x_i; \bar{\theta}_s), f(\widetilde{x}_i; \bar{\theta}_s))$ and AdamUpdate$_{\mathcal{B}}$ denotes the ADAM update rule for optimizing (3) using the mini-batch $\mathcal{B}$; $\Pi_{\mathcal{A}}$ denotes the projection to $\mathcal{A}$.

Input: T: the total number of iterations, $\mathcal{X}$: the dataset, θ_0: the parameter of the pre-trained model, S: the total number of iteration for solving (2), σ^2: the variance of the random initialization for $\widetilde{x}_i$'s, $T_{\widetilde{x}}$: the number of iterations for updating $\widetilde{x}_i$'s, η: the learning rate for updating $\widetilde{x}_i$'s, β: momentum parameter.

```
 1: $\widetilde{\theta}_1 \leftarrow \theta_0$
 2: for $t = 1, .., T$ do
 3:     $\bar{\theta}_1 \leftarrow \theta_{t-1}$
 4:     for $s = 1, .., S$ do
 5:         Sample a mini-batch $\mathcal{B}$ from $\mathcal{X}$
 6:         For all $x_i \in \mathcal{B}$, initialize $\widetilde{x}_i \leftarrow x_i + \nu_i$ with $\nu_i \sim \mathcal{N}(0, \sigma^2 I)$
 7:         for $m = 1, .., T_{\widetilde{x}}$ do
 8:             $\widetilde{g}_i \leftarrow \frac{g_i(\widetilde{x}_i, \bar{\theta}_s)}{\|g_i(\widetilde{x}_i, \bar{\theta}_s)\|_\infty}$
 9:             $\widetilde{x}_i \leftarrow \Pi_{\|\widetilde{x}_i - x\|_\infty \le \epsilon}(\widetilde{x}_i + \eta \widetilde{g}_i)$
10:         end for
11:         $\bar{\theta}_{s+1} \leftarrow \text{AdamUpdate}_{\mathcal{B}}(\bar{\theta}_s)$
12:     end for
13:     $\theta_t \leftarrow \bar{\theta}_S$
14:     $\widetilde{\theta}_{t+1} \leftarrow (1 - \beta)\bar{\theta}_S + \beta \widetilde{\theta}_t$
15: end for
```

Output: θ_T

Figure 1: SMART implementation pseudo-code from SMART Paper [2]

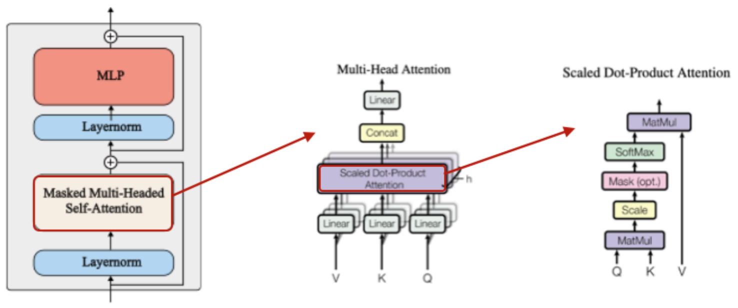


Figure 2: GPT-2 Layer Architecture, with additional illustrations of Multi-Head Attention module and Scaled-Dot Product Attention module