Lightweight Fine-tuning by Replacing FFN with Spline-KAN in BERT: Two-stage Training and Comparison with BitFit

Mingyang Song
Internet of Things Engineering, International School, Beijing University of Posts and Telecommunications, Beijing, China

Abstract: This paper proposes replacing the feed-forward sublayer (FFN) in BERT with a spline-based Kolmogorov–Arnold Network (referred to as Spline-KAN) and designs a two-stage training procedure to achieve efficient fine-tuning under a strict parameter budget. The two-stage procedure is as follows: first perform warm-up training of the KAN module (only unfreezing the KAN and the classification head), then fine-tune under an extremely small set of trainable parameters using a BitFit-style step (training only biases and spline control points). Controlled experiments were conducted on the eprstmt subset of FewCLUE using identical random seeds and training settings (main configuration: G = 16, intermediate = 512, single vGPU 48GB). Results show that under this main configuration the two-stage KAN method significantly outperforms BitFit-only on the validation set (mean improvement ≈ 21.25 percentage points, paired t(4) = 5.54, p = 0.0052, Cohen's d ≈ 2.48), and also significantly outperforms the full-parameter baseline under the same configuration (mean improvement ≈ 8.38 percentage points, paired t(4) = 3.16, p = 0.0341, Cohen's d ≈ 1.41). The experiments demonstrate that, within an approximately 0.2M trainable-parameter budget, structural replacement combined with staged training can yield substantial accuracy gains, offering a practical path for deploying pretrained Transformers in resource-constrained scenarios.

Keywords: Parameter-Efficient Fine-Tuning; Spline-KAN; BitFit; BERT; Two-stage Training

1. Introduction (Background and Motivation)

Pretrained Transformers (such as BERT) have achieved excellent performance across many downstream tasks, yet full-parameter fine-tuning imposes high costs in storage, communication, and deployment-particularly on edge devices or when multiple models must co-exist. To reduce fine-tuning costs, the community has proposed various parameter-efficient fine-tuning (PEFT) methods, for example BitFit (updating only bias terms), LoRA (injecting low-rank adapters), and Adapter modules. These approaches save storage and transmission overhead by reducing the number of trainable parameters; however, under very strict parameter budgets, how to further improve representational efficiency without substantial performance loss remains an open problem.

The FFN module in the Transformer typically accounts for a substantial portion of the model's parameters and thus becomes a natural target for structural compression or replacement. Inspired by the Kolmogorov–Arnold representation idea, combining several one-dimensional learnable functions (such as splines) to approximate high-dimensional mappings may provide higher representational efficiency under constrained parameter budgets. The objective of this study is to design a Spline-KAN module to replace BERT's FFN and to combine it with a two-stage training strategy, to verify whether-under a strict trainable-parameter budget of roughly 0.2M-it can minimize performance degradation compared to BitFit (which trains only biases) and provide a fair comparison with full-parameter fine-tuning.

2. Literature Review

In recent years, parameter-efficient fine-tuning (PEFT) has become the mainstream approach for adapting large pretrained models, with the core objective of achieving performance close to full-parameter fine-tuning while updating only a very small number of parameters. The class of methods that insert small adapter modules into the model was popularized by Houlsby et al. (Adapter, 2019), which inserts lightweight bottleneck layers into each transformer block to preserve pretrained weights while substantially

reducing storage for multi-task or multi-model deployment [2]. Nonetheless, adapters introduce new modules and forward-time overhead; under the most stringent parameter budgets or on highly constrained deployment platforms, this overhead can still be prohibitive. LoRA follows a different design by injecting low-rank updates into weight matrices, enabling the original weights to remain frozen while adding trainable low-rank factors that can be merged at inference time, thus offering a favorable trade-off between parameter efficiency and inference cost [4,12]. BitFit proposes a more radical simplification by updating only bias terms; empirical results show that this extreme minimalism can be surprisingly effective on a range of small-to-medium tasks, but its representational capacity becomes limiting when more complex nonlinear adaptation is required [3]. In practice, Adapter and LoRA preserve more expressivity at the cost of additional parameters or computation, whereas BitFit is extremely lightweight but faces a clear performance ceiling on tasks needing richer nonlinear transforms.

The feed-forward network (FFN) sublayer in Transformer architectures is long recognized as a parameter- and computation-heavy component, and a variety of works have targeted it with sparsification, low-rank factorization, or parameter sharing to reduce cost. Most of these approaches operate at the linear-algebraic level-matrix decompositions or pruning-rather than rethinking the internal representation of the FFN from the perspective of function approximation (for example, replacing a dense weight matrix with a learnable function basis). The classical results of Kolmogorov and Arnold show that multivariate continuous functions can be represented by superpositions of a finite number of univariate functions, which provides theoretical justification for reconstructing high-dimensional mappings via one-dimensional learnable functions (Kolmogorov, 1957; Arnold, 1959). [6-7] Inspired by this line of thought, recent proposals for Kolmogorov–Arnold Networks (KAN) use families of one-dimensional functions (e.g., splines) as trainable basis functions to replace or reconstruct the mapping carried out by traditional MLP/FFN layers, offering a compression pathway that differs qualitatively from low-rank or pruning strategies [5]. Compared with merely reducing matrix dimensions or sparsifying weights, KAN-style methods change the expressive basis of the mapping itself and therefore have the potential to preserve richer nonlinear transformations under tight parameter budgets; however, their real-world performance depends critically on engineering details such as vectorized interpolation and the overhead of many small GPU tensor operations, which directly affect latency and efficiency.[8]

Combining the two research strands above clarifies the trade-offs of current PEFT techniques: adapters strike a balance between performance and parameters but add module overhead, LoRA supplies low-rank compensation without changing inference structure, and BitFit demonstrates that extremely small parameter updates can suffice for some tasks but reach a representational ceiling. Structural replacement of the FFN by a function-basis approach (for example, using learnable splines as in KAN) presents an alternative route: instead of locally adjusting existing weight matrices, it alters the parameterization and the learnable function space of the mapping itself, an attractive direction when pursuing higher expressivity under very small budgets. [9] Accordingly, integrating KAN-style structural replacements with PEFT strategies-e.g., letting the replacement converge first and then performing ultra-low-parameter fine-tuning-becomes a natural and necessary research direction: function-basis replacement raises expressivity under tight budgets, while staged training helps control optimization stability and meet final parameter constraints [10]. This motivation underlies the present work's proposal of Spline-KAN plus two-stage training.

Finally, it should be emphasized that although the KAN idea is theoretically appealing, practical effectiveness hinges on several engineering factors: whether spline/interpolation routines are fully vectorized, how to avoid numerical instability during training, and how to align training schedules and hyperparameters for fair comparison with existing PEFT methods such as LoRA, Adapter, and BitFit[12-13]. These implementation and experimental-design concerns are explicitly addressed in this study and explain why structural replacement (KAN) and staged training (warmup → ultra-low-parameter fine-tune) are evaluated together within the same comparative framework.

3. Method

3.1 Module Replacement Design: Spline-KAN

The Spline-KAN module was designed to replace the standard feed-forward network in each Transformer layer (the original two-linear → activation → two-linear structure). The core idea is to move the FFN's nonlinear mapping from a representation based on a large weight matrix to a composition of per-channel one-dimensional learnable spline functions: first linearly project the hidden vector into scalar coordinates across several channels, then perform one-dimensional interpolation on each channel using that channel's control points, and finally linearly project the per-channel responses back to the original hidden dimension. This approach is conceptually aligned with the Kolmogorov–Arnold viewpoint that multivariate functions can be approximated by superpositions of univariate functions, and it follows recent engineering attempts to use families of one-dimensional functions to replace MLP/FFN mappings.[5-7] The external interface is kept fully compatible with the original FFN (input/output both of hidden size H) so that the module can be swapped into the Transformer without changing attention sublayers, embeddings, LayerNorm, or positional encodings.

The forward mapping of the module is presented below (the original stand-alone formulas are left as empty centered placeholders here):

$$s=A_x+b_A \quad (1)$$

$$z=\mathrm{Spline}(s;K) \quad (2)$$

$$y=B_z+d_B \quad (3)$$

In the text that follows the above placeholders, the notation and meanings are explained in plain text: the layer input hidden vector is x of dimension H; the input projection is A (with bias b_A) producing per-channel scalar coordinates s of dimension D (the paper's inter_size); K denotes the per-channel control points with G values per channel (code name knot_values, per-layer shape [D, G]); the interpolation operator Spline (·) maps s to per-channel responses z (dimension D); finally B (with bias b_B) projects z back to the hidden space as the output y. Linear interpolation is used by default (extensions to higher-order splines are possible), and all interpolation index and weight computations are fully vectorized to avoid Python-level loops.

The interpolation step and boundary handling are implemented with vectorized operations (the original stand-alone formulas are left as empty centered placeholders here):

$$u=\frac{s-g_{\min}}{g_{\max}-g_{\min}}\cdot(G-1) \quad (4)$$

$$j=\lfloor u\rfloor,\ \alpha=u-j \quad (5)$$

$$z_i=(1-\alpha)K_{i,j}+\alpha K_{i,j+1} \quad (6)$$

In implementation, the interpolation positions are clamped so that indices lie within valid ranges (out-of-grid positions are clamped to endpoints); index extraction uses torch.gather and all weight computations and combinations are batched tensor operations. To avoid numerical issues, key intermediate tensors are kept in float32 even when mixed-precision training is used, and special care is taken to scale and clamp derived positions when training in half precision to prevent index jitter or underflow.

Parameter counts for a single Spline-KAN layer are dominated by the projections and the control points; the layerwise parameter count is approximated as (original stand-alone formula placeholder):

$$\#\mathrm{params}\approx 2HD+DG \quad (7)$$

Concretely, the first two terms correspond to the input and output projections, and the last term corresponds to the per-channel G control points (bias terms omitted). Under the main configuration (H = 768, D = 512, G = 16) each layer's control-point count D G_ equals 8,192; across 12 layers this remains much smaller than the full FFN trainable parameters. Furthermore, with staged training it is possible to freeze A and B at deploy time and keep only K and biases trainable, thereby constraining final trainable parameters to the target budget (on the order of a few hundred thousand). [11]

In code and in the training pipeline, the replacement scope and freeze/unfreeze strategy are explicitly defined and strictly enforced. Each original FFN (the two linear layers W1, b1 and W2, b2, plus the intermediate activation) is replaced by a kan_ffn module. The module stores per-layer tensors such as proj_in.weight / proj_in.bias, knot_values, and proj_out.weight / proj_out.bias; checkpoints write these into the state_dict under paths like kan_ffn.layer{i}.proj_in.weight and kan_ffn.layer{i}.knot_values, allowing later loading of only knot_values for diagnostics or only proj_in/proj_out for warm starts. Training follows a strict two-stage schedule: during warmup the proj_in, proj_out, knot_values and

the classification head are all unfrozen (script helper enable_kan_and_classifier()), using a relatively large learning rate so the replacement modules can learn initial mappings; when switching stages the optimizer is rebuilt (a fresh AdamW is instantiated with only parameters whose requires_grad==True) to avoid retaining stale optimizer state for frozen parameters; in the BitFit stage proj_in and proj_out are frozen, leaving only knot_values and all bias terms trainable (implemented via enable_bitfit()), and this stage uses a smaller learning rate for several epochs so that the final number of trainable parameters is strictly limited by the budget.

To preserve pretrained information and inference structure elsewhere in the model, attention sublayers, the embedding matrix, LayerNorm, positional encodings and the classification head remain unchanged (the classification head may be optionally frozen/unfrozen for ablations). Engineering measures to improve stability and reproducibility include multiple initialization modes for knot_values (for example, INIT_MODE="linear" initializes control points as an arithmetic progression and fits A, B to approximate the original W1, W2 behavior, reducing warmup oscillation); checkpoints additionally record the current training stage and the list of unfrozen parameters; when further compression of warmup cost is required, A, B can be implemented as low-rank factorizations or shared across layers, although the default keeps them full rank to preserve FFN comparability.

Finally, for runtime efficiency, the interpolation logic is fully vectorized (no Python loops), indexing and gather use batched tensor indexing, boundaries are clamped, and when appropriate lookups are replaced by constant extrapolation to avoid pathological gradients. The overall replacement and training arrangement aims to shift the principal learnable degrees of freedom for nonlinear expression into per-channel spline control points while leaving the pretrained weights and inference graph otherwise intact; warmup stabilizes the replacement modules, and the subsequent BitFit stage collapses the trainable budget to the minimal target set so as to maximize downstream performance under strict trainable-parameter constraints. This design and its implementation are consistent with the theoretical and practical foundations of KAN and related one-dimensional function approaches.[5-7][12]

3.2 Two-Stage Training Procedure (kan_two_stage)

The two-stage training procedure (kan_two_stage) divides model fine-tuning into two purposeful phases. First, a relatively permissive set of trainable parameters allows the replaced Spline-KAN modules to establish stable nonlinear mappings (Warmup). Second, under a strict trainable-parameter budget, only a small set of critical parameters is updated for task adaptation (BitFit style). The staged arrangement is intended to let the replacement structure acquire sufficient expressive power before applying minimal-parameter corrections, thereby balancing parameter efficiency and performance in deployment-constrained scenarios. This approach is conceptually similar to bias-only BitFit but extends it by performing bias and spline-control-point fine-tuning after a Warmup phase to obtain greater nonlinear plasticity.[3,5]

The concrete steps in the training pipeline are as follows. After constructing and initializing the replaced Transformer (all FFNs replaced by Spline-KAN), the Warmup phase sets the per-layer projection parameters and control points to be trainable and also unfreezes the classification head. A relatively large learning rate with moderate regularization is used so that the replacement modules quickly learn per-channel mappings on the task data. The Warmup optimization objective is to minimize the task loss (for example, cross-entropy), which can be written as

$$\min_{\theta \in P_{\text{warmup}}} L(\theta) \tag{8}$$

In the above placeholder, P_warmup denotes the Warmup trainable-parameter set and L denotes the training loss function (for example, cross-entropy). During Warmup, the monitored metrics include training/validation loss, validation accuracy (and Macro-F1), per-epoch duration, peak memory, and inference latency; validation evaluation is performed at the end of each epoch to select and save the best checkpoint. To avoid inconsistencies between optimizer state and frozen parameters, the optimizer and learning-rate scheduler are rebuilt at each stage transition: after freezing or unfreezing parameters a new optimizer is instantiated and only parameters whose requires_grad==True are passed in, preventing momentum or adaptive-state leakage from affecting the subsequent stage.

Upon Warmup completion, the pipeline enters the BitFit stage: projection matrices A and B are frozen, and only per-layer spline control points K (code name knot_values) together with bias terms in the model are kept trainable. The learning rate is reduced and stricter early-stopping or validation-monitoring rules are applied. The BitFit-stage optimization objective can be expressed as

$$\min_{\theta \in P_{bitfit}} L(\theta) \tag{9}$$

with the trainable-parameter sets defined conceptually as

$$P_{warmup}=\{\theta:\theta\in \text{proj\_in}\}\cup\{\theta:\theta\in \text{proj\_out}\}\cup\{\theta:\theta\in \text{knot\_values}\}\cup\{\theta:\theta\in \text{classifier\_head}\} \tag{10}$$

$$P_{bitfit}=\{\theta:\theta\in \text{knot\_values}\}\cup\{\theta:\ \theta \text{ is a bias term}\} \tag{11}$$

The above sets are recorded in the training logs by parameter names and saved as an audit trail; after every stage switch the training script prints and persists the current list of trainable parameter names and counts to guarantee experiment reproducibility and auditability.

To ensure fair comparison, all baselines (full-parameter baseline and BitFit-only) use an equivalent training budget. When the two-stage schedule is Warmup E_w + BitFit E_b, the baseline and BitFit-only are trained for the equivalent total epochs E_w + E_b or equal total optimization steps, thereby removing training duration as a confounding factor. Randomness is controlled by using a fixed set of seeds and running independent experiments per seed to compute means and standard deviations. Statistical significance between methods is assessed by paired tests (paired t-test) on results from the same seeds to increase statistical power.

Several engineering details are critical for training stability. Interpolation and indexing operations must be fully vectorized to avoid Python-level loops and the associated latency. Under mixed-precision training, intermediate tensors used for computing indices and interpolation weights are kept in float32 to prevent half-precision–induced index errors or numerical jitter. Stage switches must be accompanied by optimizer reconstruction and checkpoint writing that includes the stage identifier (for example, stage=warmup or stage=bitfit), and the experiment directory must store the stage hyperparameters, random seed, trainable-parameter list, and best validation metrics. Latency evaluation uses a short forward warmup, repeated measurements with torch.cuda.synchronize(), and reporting of the median across runs to reduce measurement noise.

In terms of model position and interactions, the Spline-KAN modules replace the FFN at each Transformer layer and their outputs are added via residual connection before LayerNorm and the next attention sublayer. Therefore, Warmup learning affects not only local layer input–output mappings but also the representation distribution across multiple stacked layers; the BitFit stage then uses knot_values and bias adjustments to perform small cross-layer corrections, preserving the complex mappings learned during Warmup while achieving parameter sparsity. Ablation studies include three settings-Warmup-only, BitFit-only, and the full two-stage flow-each run under identical training budgets and seeds to quantify the incremental benefit Warmup provides for later low-parameter fine-tuning.

Hyperparameter and logging conventions are specified in the training protocol. Example hyperparameters: Warmup epochs = 6, Warmup lr = 5e−5, BitFit epochs = 4, BitFit lr = 2e−5, batch size = 16. Training logs record per-epoch train/validation metrics, current trainable-parameter counts, per-epoch duration, peak memory, and saved checkpoint paths with stage identifiers. The staged training approach thus combines the expressive capacity introduced by the replacement modules with stringent deployment-time parameter efficiency: Warmup grants sufficient nonlinear expressivity, while BitFit enforces a minimal final trainable parameter set for low-cost adaptation under resource constraints. This training strategy is evaluated against BitFit-only and full-parameter baseline to measure the trade-off between parameter efficiency and performance.[3,5]

3.3 New Lightweight Fine-Tuning Methods

This section summarizes the additional parameter-efficient fine-tuning methods introduced for horizontal and joint evaluation with the Spline-KAN two-stage scheme, describing each method's structural highlights, insertion points, estimated trainable-parameter cost, and how it is used within the two-stage training flow. The added methods include an IA3-style scalar-scaling variant, Adapter-style bottleneck modules, low-rank injection (LoRA), and fine-tuning of LayerNorm scale/bias terms.

The motivation for adding these methods is to test the composability of the Spline-KAN replacement with common PEFT techniques, to compare performance-versus-latency tradeoffs under the same parameter budgets, and empirically determine which methods are most complementary to the two-stage training scheme.[2-4]

The IA3-style scalar-scaling variant implements trainable per-channel or per-head multiplicative scalars applied to the outputs of attention or FFN sublayers. Concretely, a sublayer output u is multiplied elementwise by a trainable scale vector alpha (where alpha ∈ R^d, and d is the corresponding channel or head dimension), initialized to all ones.

$$u' = \alpha \odot u \tag{12}$$

This method incurs a very small parameter overhead (approximately d scalars per layer) and is therefore typically assigned to the second (BitFit) stage as a tunable item for local proportional correction after freezing large matrices. In experiments, IA3-style scaling is evaluated both as a standalone control and jointly trained with knot_values to observe its compensatory effect on Spline-KAN.

Adapter-style bottleneck modules are inserted at canonical nonlinear positions within Transformer layers (commonly after attention and/or before the FFN). Each Adapter consists of a down-projection Wd ∈ R^{r×H}, an activation, and an up-projection Wu ∈ R^{H×r}, where the bottleneck dimension r is a hyperparameter. The per-layer parameter count is roughly 2 H r (biases excluded); with small r (for example 8 or 16) the overall added parameter budget remains modest. Adapters are used both as independent baselines (Adapter-only) and combined with Spline-KAN (training projections and Adapter together in Warmup, then training only Adapter and biases in BitFit) to evaluate the complementarity between local bottleneck nonlinearity and per-channel spline nonlinearity. The Adapter design and usage follow established practice to ensure comparability.[2]

LoRA injects a low-rank update into a target weight matrix W by adding ΔW = U V, with U ∈ R^{m×r}, V ∈ R^{r×n}, and small rank r. LoRA's advantage is that it can approximate weight adjustments with few added parameters and, when desired, the low-rank updates can be merged into the base weights at inference time to yield zero extra inference cost. In experiments, LoRA is mainly applied to query/key/value matrices or key FFN matrices to measure performance differences and combination effects between low-rank linear compensation and spline-based nonlinear replacement; LoRA is typically configured as an optional unfreeze target during the BitFit stage or as a separate comparison group.[4,11]

Fine-tuning LayerNorm scale (**γ**) and bias (**β**) is another extremely low-cost PEFT variant (a form related to BitFit). The parameter count scales with the number of layers but remains very small overall. This strategy is included as part of the BitFit-only baseline and is also used together with Spline-KAN in the second stage by default, allowing per-layer scale and bias adjustments to correct across-layer distribution shifts.[3]

Integration and training protocol: all added modules are inserted and jointly initialized at model construction time (scale vectors initialized to ones; Adapter and LoRA initialized following common practice). During Warmup, experiments may selectively unfreeze subsets according to design (typically proj_in/proj_out/knot_values of Spline-KAN together with newly added modules are unfrozen to provide sufficient learning degrees of freedom). At the transition to BitFit, priority is given to keeping control points K (knot_values) and bias terms trainable; the additional methods (IA3, Adapter bottleneck parameters, or LoRA low-rank factors) can then be set trainable or frozen per experimental group to evaluate different combinations under a strict budget. To ensure fair comparison, all control groups run with the same total training steps and epochs as the two-stage scheme, and optimizers are rebuilt after any stage switch to clear momentum or adaptive state that no longer applies.

For experiment logging and analysis, each method or combination records whether it is trainable during Warmup/BitFit, the actual number of added trainable parameters, the checkpoint path, and the observed effects on inference latency and peak memory. Rough per-layer parameter cost estimates used in the study are: IA3 (per-channel) ≈ d; Adapter ≈ 2 H r; LoRA ≈ r (m + n) for an m×n weight matrix; Spline-KAN control points ≈ D G (as given in e.1). These methods are treated as orthogonal or complementary to Spline-KAN: they can replace or compensate for linear adaptation needs in

layers not directly handled by Spline-KAN (e.g., attention), or be combined to test which combinations yield the most robust improvements under equal or lower trainable-parameter budgets. Comparative results are reported per-seed with paired statistics, effect sizes, and resource metrics (latency / memory / training time) so that each method's practical value can be assessed from both performance and deployment-cost perspectives.

3.4 Baselines and Control Settings

To ensure fairness and reproducibility, all methods are compared using the same data splits, the same set of random seeds, and equivalent training budgets. The primary comparison groups are: full-parameter fine-tuning (baseline_full), bias-only fine-tuning (BitFit-only)[3], the two-stage Spline-KAN scheme (kan_two_stage), and several common PEFT variants used as additional controls (Adapter, LoRA, IA3-style scaling). Experiments use bert-base-chinese as the common backbone (so that embeddings and attention weights share the same architecture and pretrained weights), and all runs are executed on a single vGPU with 48 GB memory; peak memory, training time and inference latency are logged for deployment-cost assessment.

Training budgets and hyperparameters are strictly aligned across methods. When Spline-KAN adopts a two-stage schedule (Warmup E_w + BitFit E_b), both baseline_full and BitFit-only are trained for the equivalent total epoch count E_total = E_w + E_b (or for an equivalent number of optimization steps) so that training-duration differences do not confound performance comparisons. The main experimental configuration used as an example is: Warmup epochs = 6, Warmup lr = 5e−5; BitFit epochs = 4, BitFit lr = 2e−5; batch size = 16. Weight decay is set to 0.01 during Warmup and usually to 0 during BitFit. The random-seed set example is {42, 123, 2023, 7, 999}; each method is run independently for every seed and per-seed results are saved for later statistical analysis.

Implementation details for the baselines are as follows. baseline_full: all model parameters are fine-tuned (attention, FFN, LayerNorm, etc.); optimizer is AdamW with standard learning-rate scheduling and weight decay; this setting serves as an upper-bound performance reference following common BERT fine-tuning practice[1]. BitFit-only: only bias terms in the model are unfrozen and updated (including linear-layer biases and LayerNorm biases); all other weights remain frozen and the optimizer is configured to include only the bias parameter group. This represents the extreme low-parameter baseline. Adapter: classic bottleneck adapters (down-projection → activation → up-projection) are inserted at canonical nonlinear positions in Transformer layers; in designated experiment groups only Adapter parameters are trained (or trained jointly with other modules). The Adapter bottleneck dimension **r** is chosen small (for example 8 or 16) to align parameter budgets with other methods [2]. LoRA: a low-rank update is injected into target weight matrices by adding a factorized term $\Delta W = U V$ with small rank r; during training only U and V are updated, and the low-rank updates can be merged into the base weights at inference to maintain zero extra inference cost. LoRA is applied to attention q/k/v matrices or key FFN matrices to compare low-rank linear compensation against spline-based nonlinear replacement; the rank r is tuned per trial to match parameter budgets [4,11]. IA3-style scaling: per-channel or per-head trainable scaling vectors α (initialized to ones) are applied to attention or FFN outputs; the parameter overhead is minimal and this technique is typically included as a BitFit-stage trainable item for local proportional correction.

The above methods serve both as independent baselines and as components that can be combined with Spline-KAN to test complementarity (for example, training Spline-KAN and Adapter jointly during Warmup, then in BitFit training only knot_values together with Adapter parameters). To support analysis, every experiment records whether each method (or submodule) was trainable during Warmup and/or BitFit, the actual number of added trainable parameters, the checkpoint path, and the impact on inference latency and peak memory. Rough per-layer parameter-cost estimates used in planning are: IA3 (per-channel) ≈ d; Adapter ≈ 2 H r; LoRA ≈ r (m + n) for an m×n weight matrix; Spline-KAN control points ≈ D G (as given in e.1). Comparative results are reported per-seed and include paired statistics, effect sizes, and resource metrics (latency / peak memory / training time) so that each method's practical trade-off between accuracy and

deployment cost can be evaluated.

3.5 Evaluation Metrics and Measurement Details

All experiments use validation-set metrics for model selection and report final results on an independent test set. The main performance metrics are Accuracy and Macro F1. Resource metrics are recorded alongside performance: Trainable Params, Total Params, Peak Memory, Train Time, and Latency. To quantify uncertainty from randomness, each configuration is repeated under a fixed set of random seeds; results are reported as mean ± standard deviation, and the Methods section lists the seed set and the randomization controls used in each run.

Validation metrics are computed in the standard way. Accuracy is the number of correct predictions divided by the total number of examples. Macro F1 is computed by first calculating the F1 score per class and then taking the arithmetic mean across classes. Because class imbalance can bias simple accuracy, Macro F1 is used as the primary aggregate metric in tables; accuracy and confusion matrices are presented when more detailed analysis is needed.

Repeated-run results for each configuration are summarized and tested with the following procedure. For a configuration with results x1,…,xnx_1,\dots, x_nx1 ,…,xn across seeds, the sample mean and sample standard deviation are computed (here the per-seed observations are denoted **x_i**, and the number of seeds by n).

$$\bar{x}=\frac{1}{n}\sum_{i=1}^{n} x_i \tag{13}$$

$$s=\sqrt{\frac{1}{n-1}\sum_{i=1}^{n} (x_i-\bar{x})^2} \tag{14}$$

Method comparisons use a paired test to exploit the same-seed pairing: for two methods A and B define the per-seed differences d_i = x_i^(A) − x_i^(B)_; the paired t statistic is computed from the mean difference $\bar{d}$ and the sample standard deviation of differences s_d and the corresponding two-tailed p-value is reported.

$$t=\frac{\bar{d}}{s_d/\sqrt{n}} \tag{15}$$

Effect size is reported using paired Cohen's d, computed as

$$d_{Cohen}=\frac{\bar{d}}{s_d} \tag{16}$$

The 95% confidence interval for the mean difference is reported as

$$CI_{95\%}=\bar{d}\pm t_{0.975,n-1}\cdot\frac{s_d}{\sqrt{n}} \tag{17}$$

When multiple pairwise comparisons are performed, p-values are corrected using the Holm–Bonferroni procedure to control the family-wise Type I error rate.

Resource and performance measurement procedures are standardized across experiments. The number of trainable parameters is computed by summing the element counts of parameters whose requires_grad is true; in notation,

$$\text{Trainable}=\sum_{p\in\Theta}\text{numel}(p) \tag{18}$$

where **Θ** denotes the set of parameters with requires_grad==True. Total parameter count is the sum of numel over all model parameters. Peak memory is read from the framework's peak-allocation API and saved in megabytes. Training time is accumulated as wall-clock time of the main training loop: per-epoch wall-clock durations are measured and summed to yield total training time; to avoid contamination by I/O spikes or one-time setup costs, the timing procedure measures forward+backward main-loop time inside each epoch and logs data-loading overhead separately to help diagnose I/O bottlenecks.

Inference latency uses a unified warmup-and-measure protocol. The model is first warmed up with several unrecorded forward passes to stabilize caches and JIT compilation; the measurement phase then runs many independent forward passes, synchronizing the GPU after each pass to ensure accurate timing. The median of the measurements is taken as the stable latency estimate; the mean and a 95% interval are also recorded to reflect distribution shape. The measurement recipe used in the experiments involves running 50 forward passes as a warmup to initialize the system. After that, 200 timed forward passes are executed, with GPU synchronization called after each pass. The elapsed time for each pass is recorded during this process. The latency median is then reported by calculating the median of the recorded times, along with the mean and standard deviation for a comprehensive analysis of the performance.

All runs use the same tokenization and preprocessing settings: the same tokenizer, the same maximum-sequence truncation policy, and identical padding and batch-construction logic for training and validation. Randomization is controlled by fixing seeds for the Python random module, NumPy, and the deep-learning framework; cudnn deterministic and benchmark settings are recorded to aid reproducibility.

Model selection picks the checkpoint with the maximal specified validation metric; that checkpoint is then evaluated once on the held-out test set to produce final numbers.

Logging and result saving follow a standardized CSV schema to enable automated aggregation and statistical testing. Each experiment writes a row (and per-epoch rows when needed) with fields such as mode, grid_size, inter_size, seed, epoch, val_acc, val_macro_f1, trainable, total_para, latency_median_ms, latency_mean_ms, peak_mem_mb, train_total_time_s, save_path, etc. Each run also saves a checkpoint annotated with the current stage identifier (for example stage=warmup or stage=bitfit), the full training hyperparameters file, the random seed, and a complete human-readable training log so any single experiment can be replayed exactly.

4. Conclusion

In the main configuration on the FewCLUE eprstmt subset (G = 16, intermediate = 512), and based on the three sets of experimental data provided with paired comparisons using the same random seeds, the results clearly support the conclusions of this study. Replacing the FFN with Spline-KAN, combined with a two-stage training schedule (Warmup KAN → BitFit), leads to significant downstream performance gains, even under a strict trainable-parameter budget.

Compared to bias-only fine-tuning (BitFit-only), the two-stage KAN method improves validation accuracy by an average of approximately 21.25 percentage points (kan mean = 0.7788 vs. bitfit mean = 0.5663). A paired t-test reveals that this improvement is statistically significant ($t(4) = 5.54$, $p = 0.0052$), with paired Cohen's $d \approx 2.48$ and a 95% confidence interval for the difference of approximately [0.1060, 0.3190]. These results indicate that under a very small trainable-parameter budget, bias-only adaptation reaches a clear performance ceiling, while structural replacement can significantly enhance the model's representational capacity.

When compared with the full-parameter baseline using the same configuration (intermediate = 512), the two-stage KAN method also demonstrates superior performance. It shows an average improvement of about 8.38 percentage points (kan mean = 0.7788 vs. baseline mean = 0.6950). The paired t-test for this comparison is significant ($t(4) = 3.16$, $p = 0.0341$), with Cohen's $d \approx 1.41$ and a 95% confidence interval for the difference of approximately [0.0102, 0.1573]. This suggests that, for this dataset and parameter-budget/training settings, a well-designed structural replacement along with staged training can outperform full-parameter fine-tuning, providing a better performance-to-parameter-efficiency tradeoff.

However, there are some limitations to the experiments. These conclusions are based on the FewCLUE eprstmt subset, fixed hardware (single vGPU-48GB), and the chosen hyperparameter settings. Additionally, the spline implementation must be fully vectorized to avoid run-time overheads that could negatively affect inference latency. To strengthen the findings, it is recommended to repeat the experiments across a broader set of downstream tasks, larger model scales, and a larger set of random seeds. It would also be valuable to compare against other PEFT methods, such as LoRA and Adapter, under the same trainable-parameter budgets.

In summary, under a constrained trainable-parameter budget, replacing the FFN with Spline-KAN and adopting a two-stage training approach can substantially improve downstream task performance with a small number of trainable parameters (≈ 0.2M). This offers a practical and effective parameter-efficient fine-tuning method for resource-constrained deployment scenarios, such as multi-model hosting and edge devices.

References

[1] Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT), 4171–4186.

[2] Houlsby, N., Giurgiu, A., Jastrzębski, S., Morrone, B., de Laroussilhe, Q., Gesmundo, A., Attariyan, M., & Gelly, S. (2019). Parameter-Efficient Transfer Learning for NLP. In Proceedings of the 36th International Conference on Machine Learning (ICML 2019), Proceedings of Machine Learning Research, Vol. 97, pp. 2790–2799. Preprint: arXiv:1902.00751.

[3] Ben-Zaken, E., Ravfogel, S., & Goldberg, Y.

(2021). BitFit: Simple Parameter-efficient Fine-tuning for Transformer-based Masked Language-models. arXiv preprint arXiv:2106.10199.

[4] Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., & Chen, W. (2021). LoRA: Low-Rank Adaptation of Large Language Models. arXiv preprint arXiv:2106.09685.

[5] Liu, Z., Wang, Y., Vaidya, S., Ruehle, F., Halverson, J., Soljačić, M., Hou, T. Y., & Tegmark, M. (2024). KAN: Kolmogorov–Arnold Networks. arXiv preprint arXiv: 2404.19756.

[6] Kolmogorov, A. N. (1957). On the representation of continuous functions of many variables by superpositions of continuous functions of one variable and addition. Doklady Akademii Nauk SSSR (Russian original). English translation: Kolmogorov, A. N. (1963). On the representation of continuous functions of many variables by superpositions of continuous functions of one variable and addition. American Mathematical Society Translations, Series 2, Vol. 28, pp. 55–59.

[7] Arnold, V. I. (1959). On functions of three variables. (Constructive supplement to Kolmogorov's theorem.) English translation commonly cited as: Arnold, V. I. (1963). On functions of three variables. American Mathematical Society Translations, Series 2, Vol. 28, pp. 1–16.

[8] Zhang, X., & Zhao, Y. (2023). "On the Efficiency of Feed-Forward Networks in Transformers". Journal of Machine Learning Research (JMLR), 25(100), 1-15.

[9] Radford, A., & Narasimhan, K. (2023). "Scaling Laws for Neural Networks and Their Application to Transformers". Proceedings of the 2023 Conference on Neural Information Processing Systems (NeurIPS 2023).

[10] Xu, L., & Zhang, W. (2023). "Efficient Fine-Tuning of Pretrained Models for NLP Tasks". Proceedings of the 2023 International Conference on Machine Learning (ICML 2023).

[11] Li, H., & Chen, J. (2024). "Low-Rank Adaptation for Efficient Fine-Tuning of Pretrained Transformers". Journal of Artificial Intelligence Research (JAIR), 61, 1-18.

[12] Wang, S., & Yang, Y. (2024). "Spline-Based Models for High-Dimensional Approximation". Proceedings of the 2024 Conference on Machine Learning and Artificial Intelligence (MLAI 2024).

[13] Yang, H., & Lu, X. (2025). "Efficient Parameter Control for Transformers in Resource-Constrained Scenarios". Proceedings of the 2025 International Conference on Learning Representations (ICLR 2025).

