Journal of Artificial Intelligence Research 77 (2023) 39-70 Submitted 05/2022; published 05/2023

FlexiBERT: Are Current Transformer Architectures too Homogeneous and Rigid?

Shikhar Tuli STULI@PRINCETON.EDU
Bhishma Dedhia BDEDHIA@PRINCETON.EDU
Dept. of Electrical & Computer Engineering, Princeton University
Princeton, NJ 08544 USA

Shreshth Tuli S.TULI20@IMPERIAL.AC.UK
Department of Computing, Imperial College London
London, SW7 2AZ UK

Niraj K. Jha JHA@PRINCETON.EDU
Dept. of Electrical & Computer Engineering, Princeton University
Princeton, NJ 08544 USA

Abstract

The existence of a plethora of language models makes the problem of selecting the best one for a custom task challenging. Most state-of-the-art methods leverage transformer-based models (e.g., BERT) or their variants. However, training such models and exploring their hyperparameter space is computationally expensive. Prior work proposes several neural architecture search (NAS) methods that employ performance predictors (e.g., surrogate models) to address this issue; however, such works limit analysis to homogeneous models that use fixed dimensionality throughout the network. This leads to sub-optimal architectures. To address this limitation, we propose a suite of *heterogeneous* and *flexible* models, namely FlexiBERT, that have varied encoder layers with a diverse set of possible operations and different hidden dimensions. For better-posed surrogate modeling in this expanded design space, we propose a new graph-similarity-based embedding scheme. We also propose a novel NAS policy, called BOSHNAS, that leverages this new scheme, Bayesian modeling, and second-order optimization, to quickly train and use a neural surrogate model to converge to the optimal architecture. A comprehensive set of experiments shows that the proposed policy, when applied to the FlexiBERT design space, pushes the performance frontier upwards compared to traditional models. FlexiBERT-Mini, one of our proposed models, has 3% fewer parameters than BERT-Mini and achieves 8.9% higher GLUE score. A FlexiBERT model with equivalent performance as the best homogeneous model has 2.6× smaller size. FlexiBERT-Large, another proposed model, attains state-of-the-art results, outperforming the baseline models by at least 5.7% on the GLUE benchmark.

1. Introduction

In recent years, self-attention (SA)-based transformer models (Vaswani et al., 2017; Devlin et al., 2019) have achieved state-of-the-art results on tasks that span the natural language processing (NLP) domain. Large-scale pre-training datasets, increasing computational power, and robust training techniques (Liu et al., 2019) drive this burgeoning success. A challenge that remains is *efficient* optimal model selection for a specific task and a set of user requirements. In this context, one should train only models with the maximum *predicted* performance. This falls in the domain of neural architecture search (NAS) (Zoph & Le, 2017).

1.1 Challenges

The design space of transformer models is vast. Rigorous search has proposed several models in the past. Popular models include BERT, XLM, XLNet, BART, ConvBERT, and FNet (Devlin et al., 2019; Conneau & Lample, 2019; Yang et al., 2019; Lewis et al., 2020; Jiang et al., 2020; Lee-Thorp et al., 2022). Transformer design involves a choice of several hyperparameters, including the number of layers, size of hidden embeddings, number of attention heads, and size of the hidden layer in the feed-forward network (Khetan & Karnin, 2020). This leads to an exponential increase in the design space, making a brute-force approach to explore the design space computationally infeasible (Ying et al., 2019). The aim is to converge to an optimal model as quickly as possible by testing the lowest possible number of datapoints (Pham et al., 2018). Moreover, model performance may not be deterministic, requiring heteroscedastic modeling (Ru et al., 2020).

1.2 Existing Solutions and Motivation

Recent NAS advancements use various techniques to explore and optimize different models in the deep learning domain, from image recognition to speech recognition and machine translation (Zoph & Le, 2017; Mazzawi et al., 2019). In the computer-vision domain, various search approaches, such as genetic algorithms, reinforcement learning, and structure adaptation, realize diverse convolutional neural network (CNN) architectures. Some even introduce new basic operations (Zhang et al., 2018) to enhance performance on different tasks. Many works leverage a performance predictor, often called a surrogate model, to *reliably* predict model accuracy. One can train such a surrogate through active learning by querying a few models from the design space and regressing their performance to the remaining space (under some theoretical assumptions), thus significantly reducing search times (Siems et al., 2020; White et al., 2021b).

Unlike CNN frameworks (Ying et al., 2019; Tan & Le, 2019), meant for vision tasks, there is no universal framework for NLP that differentiates among transformer architectural hyperparameters. Works that do compare different design decisions often do not consider *heterogeneity* and *flexibility* in their search space and explore the space over a limited hyperparameter set (Khetan & Karnin, 2020; Xu et al., 2021; Gao et al., 2022)[1]. For instance, Primer (So et al., 2021) only adds depth-wise convolutions to the attention heads; AutoBERT-Zero (Gao et al., 2022) lacks deep feed-forward stacks; AutoTinyBERT (Yin et al., 2021) does not consider linear transforms (LTs) that outperform traditional SA operations in terms of parameter efficiency; AdaBERT (Chen et al., 2021) only considers a design space of convolution and pooling operations. Most works, in the field of NAS for transformers, target model compression while trying to maintain the same performance (Chen et al., 2021; Yin et al., 2021; Wang et al., 2020), which is orthogonal to our objectives in this work, *i.e.*, searching for novel architectures that push the performance frontier. In addition, all

1. Here, by *heterogeneity*, we mean that different encoder layers can have distinct attention operations, feed-forward stack depths, etc. By *flexibility*, we mean that the hidden dimensions for different encoder layers in a transformer architecture can be mismatched.

Framework	Self-Attention		Conv.	Lin. Transform		Flexible no. of attn. ops.	Flexible feed-fwd. stacks	Flexible hidden dim.		Search technique
	SDP	WMA		DFT	DCT			Ad. width	Full flexibility	
Primer (So et al., 2021)			✓			✓				ES
AdaBERT (Chen et al., 2021)			✓							DS
AutoTinyBERT (Yin et al., 2021)	✓					✓	✓	✓		ST
DynaBERT (Hou et al., 2020)	✓					✓	✓	✓		ST
NAS-BERT (Xu et al., 2021)	✓		✓			✓				ST
AutoBERT-Zero (Gao et al., 2022)	✓		✓			✓				ES
FlexiBERT (ours)	✓	✓	✓	✓	✓	✓	✓	✓	✓	BOSHNAS

Table 1: Comparison of related works with different parameters (✓indicates that the corresponding feature is present). Adaptive width refers to different architectures having possibly different hidden dimensions (albeit each layer within the architecture having the same hidden dimension). Full flexibility corresponds to each encoder layer having, possibly, a different hidden dimension.

previous works only consider *rigid* architectures. For instance, DynaBERT (Hou et al., 2020) only adapts the width of the network by varying the number of attention heads (and *not* the hidden dimension of each head), which is only a simple extension to traditional architectures. Further, their individual models still have the same hidden dimension throughout the network. AutoTinyBERT (Yin et al., 2021) and HAT (Wang et al., 2020), among others, fix the input and output dimensions for each encoder layer (see Appendix A.1 for a background on the SA operation), which leads to *rigid* architectures.

Table 1 gives an overview of various baseline NAS frameworks for transformer architectures. It presents the aforementioned works and the respective features they include. Primer (So et al., 2021) and AutoBERT-Zero (Gao et al., 2022) exploit evolutionary search (ES), which faces various drawbacks that limit *elitist* algorithms (Dang et al., 2021; White et al., 2021a; Siems et al., 2020). AdaBERT (Chen et al., 2021) leverages differentiable architecture search (DS), a popular technique used in many CNN design spaces (Siems et al., 2020). On the other hand, some recent works like AutoTinyBERT (Yin et al., 2021), DynaBERT (Hou et al., 2020), and NAS-BERT (Xu et al., 2021) leverage super-network training, where they train one large transformer and search its sub-networks in a *one-shot* manner. However, this technique is not amenable to diverse design spaces, as the super-network size would drastically increase, limiting the gains from weight transfer to the relatively minuscule sub-network. Moreover, previous works limit their search to either the standard SA operation, *i.e.*, the scaled dot-product (SDP), or the convolution operation. We extend the basic attention operation to also include the weighted multiplicative attention (WMA). Taking motivation from recent advances with LT-based transformer models (Lee-Thorp et al., 2022), we also add discrete Fourier transform (DFT) and discrete cosine transform (DCT) to our design space. AutoTinyBERT and DynaBERT also allow adaptive widths in the transformer architectures in their design space. However, each instance still has the same dimensionality throughout the network (in other words, every encoder layer has the same hidden dimension, as explained above). We mathematically detail why this is inherently a limitation in traditional transformer architectures in Appendix A.1. FlexiBERT, to the best of our knowledge, is the first framework to allow full *flexibility* – not only can different transformer instances in the design space have distinct widths, but each encoder layer within a transformer instance can also have different hidden dimensions. This results in a massive design space with 3.32 billion transformer architectures. Searching this space via a brute-force technique would be computationally infeasible. Hence, we leverage a novel NAS technique, Bayesian Optimization using Second-Order Gradients and Heteroscedastic Models for Neural Architecture Search (BOSHNAS), to search for the best-performing architecture in this enormous design space.

1.3 Our Contributions

To address the limitations of *homogeneous* and *rigid* models, we make the following technical contributions:

- We expand the design space of transformer hyperparameters to incorporate *heterogeneous* architectures that venture beyond simple SA by employing other operations like convolutions and LTs.
- We propose novel projection layers and *relative/trained* positional encodings to make hidden sizes *flexible* across layers – hence the name FlexiBERT.
- We propose `Transformer2vec` that uses similarity measures to compare computational graphs of transformer models to obtain a dense embedding that captures model similarity in a Euclidean space.
- We propose a novel NAS framework, namely, BOSHNAS. It uses a neural network as a heteroscedastic surrogate model and second-order gradient-based optimization using backpropagation to input (GOBI) (Tuli et al., 2021) to speed up search for the next query in the exploration process. It leverages *nearby* trained models to transfer weights in order to reduce the amortized search time for every query.
- Experiments on the GLUE benchmark (Wang et al., 2018) show that BOSHNAS applied to the FlexiBERT design space results in a score improvement of 0.4% compared to the baseline, *i.e.*, NAS-BERT (Xu et al., 2021). The proposed model, FlexiBERT-Mini, has 3% fewer parameters than BERT-Mini and achieves 8.9% higher GLUE score. FlexiBERT also outperforms the best *homogeneous* architecture by 3%, while requiring 2.6× fewer parameters. FlexiBERT-Large, our BERT-Large (Devlin et al., 2019) counterpart, outperforms the state-of-the-art models by at least 5.7% average accuracy on the first eight tasks in the GLUE benchmark (Wang et al., 2018)[2].

We organize the rest of the paper as follows. Section 2 presents related work. Section 3 describes the set of steps and decisions that undergird the FlexiBERT framework. In

2. All the code for the FlexiBERT pipeline is available at `https://github.com/jha-lab/txf_design-space`. The code for running BOSHNAS on any tabular dataset of deep learning architectures is available at `https://github.com/jha-lab/boshnas`.

Section 4, we present the results of design space exploration experiments. Finally, Section 5 concludes the article.

2. Background and Related Work

We briefly describe related work next.

2.1 Transformer Design Space

Traditionally, transformers have primarily relied on the SA operation (Vaswani et al., 2017). Nevertheless, several works have proposed various compute blocks to reduce the number of model parameters and hence computational cost without compromising performance. For instance, ConvBERT uses dynamic span-based convolutional operations that replace SA heads to model local dependencies directly (Jiang et al., 2020). Recently, FNet improved model efficiency using LTs instead (Lee-Thorp et al., 2022). MobileBERT, another recent architecture, uses bottleneck structures and multiple feed-forward stacks to obtain smaller and faster models while achieving competitive results on well-known benchmarks (Sun et al., 2020). For completeness, we present other previously proposed advances to improve the BERT model in Appendix A.2.

2.2 Neural Architecture Search

NAS is an important machine learning technique that algorithmically searches for new neural network architectures within a pre-specified design space under a given objective (He et al., 2021). Prior work implements NAS using various techniques, albeit limited to the CNN design space. A popular approach is to use a reinforcement learning algorithm, REINFORCE, that is superior to other tabular approaches (Williams, 1992). Other approaches include Gaussian-Process-based Bayesian Optimization (GP-BO) (Snoek et al., 2012), ES (Real et al., 2019; Lu et al., 2019), etc. However, these methods come with challenges that limit their ability to reach state-of-the-art results in the CNN design space (White et al., 2021a).

Recently, NAS has also seen the application of surrogate models for performance prediction in CNNs (Siems et al., 2020). This results in the training of much fewer models to predict accuracy for the entire design space under some confidence constraints. However, these predictors are computationally expensive to train. This leads to a bottleneck, especially in large design spaces, in the training of subsequent models since we produce new queries only after we train this predictor for every batch of trained models in the search space. Siems et al. (2020) use a Graph Isomorphism Net (Xu et al., 2019) that regresses performance values directly on the computational graphs formed for each CNN model.

Although previously restricted to CNNs (Zoph et al., 2018), NAS has recently seen applications in the transformer space as well. So et al. (2019) use standard NAS techniques to search for optimal transformer architectures. However, their method trains every new model from scratch. Furthermore, they do not employ knowledge transfer, which transfers weights from previously trained *neighboring* models to speed up subsequent training. This is important in the transformer space since pre-training every model is computationally expensive. Further, the attention heads in their model follow the same dimensionality, *i.e.*, are not fully *flexible*.

One of the state-of-the-art NAS techniques, BANANAS, implements Bayesian Optimization (BO) over a neural network model and predicts performance uncertainty using ensemble networks that are, however, too compute-heavy (White et al., 2021a). BANANAS uses mutation/crossover on the current set of best-performing models and obtains the next best-predicted model in this local space. Instead, we propose using GOBI (Tuli et al., 2021) to efficiently search for the next query in the *global* space. Thanks to random cold restarts, GOBI can search over diverse models in the architecture space. BANANAS also uses path embeddings, which perform sub-optimally for search over a diverse space (Cheng et al., 2021).

2.3 Graph Embeddings that Drive NAS

Many works on NAS for CNNs use graph embeddings to model their performance predictor. Each *computational graph* has a corresponding embedding, representing a specific CNN architecture in the design space. A popular approach to learning with graph-structured data is to make use of graph kernel functions that measure similarity between graphs. A recent work, NASGEM (Cheng et al., 2021), uses the Weisfeiler-Lehman (WL) sub-tree kernel, which compares tree-like substructures of two computational graphs. This helps distinguish between substructures that other kernels, like random walk, may deem identical (Shervashidze et al., 2011). Also, the WL kernel has an attractive computational complexity. This has made it one of the most widely used graph kernels. Graph-distance-driven NAS often leads to enhanced representation capacity that yields optimal search results (Cheng et al., 2021). However, the WL kernel only computes sub-graph similarities based on overlap in graph nodes. It does not consider whether or not two nodes are *inherently* similar. For example, a computational 'block' (or its respective graph node) for an SA head with $h = 128$ and $o = \text{SDP}$ would be closer to another attention block with, say, $h = 256$ and $o = \text{WMA}$, but would be farther from a block representing a feed-forward layer.

Once we have similarities computed between every possible graph pair in the design space, we learn dense embeddings, the Euclidean distance for which should follow the similarity function. These embeddings would be not only helpful in effective visualization of the design space but also for fast computation of *neighboring* graphs in the active-learning loop. Further, a dense embedding helps us practically train a finite-input surrogate function (as opposed to the sparse path encodings used by White et al., 2021a). Many works have achieved this using different techniques. Narayanan et al. (2017) train task-specific graph embeddings using a skip-gram model and negative sampling, taking inspiration from `word2vec` (Mikolov et al., 2013). In this work, we take inspiration from `GloVe` instead (Pennington et al., 2014), by applying manifold learning to all distance pairs (Kruskal, 1964). Hence, using global similarity distances built over domain knowledge and batched gradient-based training, we obtain the proposed `Transformer2vec` embeddings that are superior to traditional generalized graph embeddings.

We take motivation from NASGEM (Cheng et al., 2021), which showed that training a WL kernel-guided encoder has advantages in scalable and flexible search. Thus, we train a performance predictor on the `Transformer2vec` embeddings, which not only aid in the transfer of weights between neighboring models but also support better-posed continuous

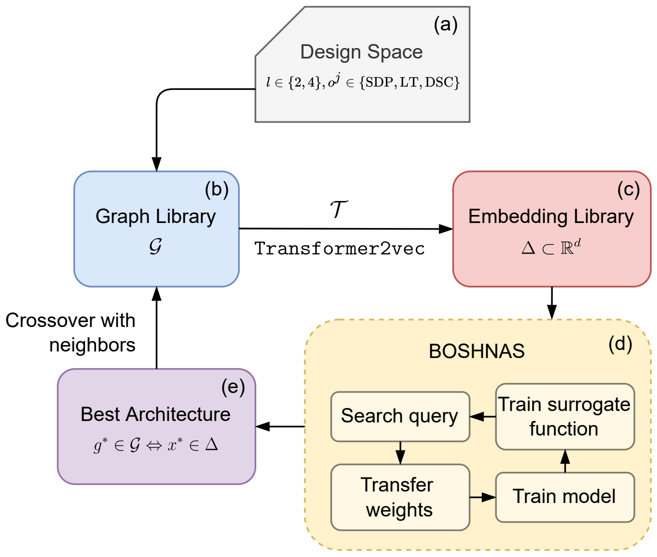

Figure 1: Overview of the FlexiBERT pipeline.

performance approximation. More details on the computation of these embeddings are given in Section 3.3.

3. Methodology

In this work, we train a heteroscedastic surrogate model that predicts the performance of a transformer architecture and uses it to run second-order optimization in the design space. We do this by decoupling the training procedure from pre-processing the embedding of every model in the design space to speed up training. First, we train embeddings to map the space of computational graphs to a Euclidean space (`Transformer2vec`) and then train the surrogate model on the embeddings.

Our work involves exploring a vast and heterogeneous design space and searching for optimal architectures with a given task. To this end, we (a) define a design space via a flexible set of architectural choices (see Section 3.1), (b) generate possible computational graphs ($\mathcal{G}$; see Section 3.2), (c) learn an embedding for each point in the space using a distance metric for graphs (Δ; see Section 3.3), and (d) employ a novel search technique (BOSHNAS) based on surrogate modeling of the performance and its uncertainty over the continuous embedding space (see Section 3.4). In addition, to tackle the enormous design space, we propose a hierarchical search technique that iteratively searches over finer-grained models derived from (e) a crossover of the best models obtained in the current iteration and their neighbors. Figure 1 gives a broad overview of the FlexiBERT pipeline, as explained above. We show an unrolled version of this iterative flow below:

$$\text{Design Space} \rightarrow \mathcal{G}_1 \xrightarrow{\mathcal{T}} \Delta_1 \xrightarrow{\text{BOSHNAS}} g^* \xrightarrow{\text{cross-over}} \mathcal{G}_2 \xrightarrow{\mathcal{T}} \ldots$$

However, for simplicity of notation, we omit the iteration index in further references. We now discuss the key elements of this pipeline in detail.

Design Element	**Allowed Values**
Number of encoder layers (l)	$\{2, 4\}$
Type of attention operation used (o^j)	{SA, LT, DSC}
Number of operation heads (n^j)	$\{2, 4\}$
Hidden size (h^j)	$\{128, 256\}$
Feed-forward dimension (f^j)	$\{512, 1024\}$
Number of feed-forward stacks	$\{1, 3\}$
Operation parameters (p^j):	
if o^j = SA	Self-attention type: {SDP, WMA}
else if o^j = LT	Linear transform type: {DFT , DCT}
else if o^j = DSC	Convolution kernel size: $\{5, 9\}$

Table 2: Design space description. Super-script (j) depicts the value for layer j.

3.1 FlexiBERT Design Space

We now describe the FlexiBERT design space, *i.e.*, box (a) in Figure 1.

3.1.1 Set of Operations in FlexiBERT

The traditional BERT model comprises multiple layers, each containing a bidirectional multi-headed SA module followed by a feed-forward module. Previous works propose several modifications to the original encoder, primarily to the attention module. This gives rise to a richer design space. We consider WMA-based SA in addition to SDP-based operations (Luong et al., 2015).

We also incorporate LT-based attention in FNet (Lee-Thorp et al., 2022) and dynamic-span-based convolution (DSC) in ConvBERT (Jiang et al., 2020), in place of the vanilla SA mechanism. Whereas the original FNet implementation uses DFT, we also consider DCT. The motivation behind using DCT is its widespread application in lossy data compression, which we believe can lead to sparse weights, thus leaving room for optimizations with sparsity-aware machine learning accelerators (Yu & Jha, 2022). Our design space allows variable kernel sizes for convolution-based attention. Consolidating different attention module types that vary in their computational costs into a single design space enables the models to have inter-layer variance in expression capacity. Inspired by MobileBERT (Sun et al., 2020), we also consider architectures with multiple feed-forward stacks. We summarize the entire design space with the range of each operation type in Table 2. The ranges of different hyperparameters are in accordance with the design space spanned by BERT-Tiny to BERT-Mini (Turc et al., 2019), with additional modules included as discussed. We call this the Tiny-to-Mini space. This restricts our curated testbed to models with up to 3.3×10^7 trainable parameters. This curated parameter space allows us to perform extensive experiments, comparing the proposed approach against various baselines.

We can express every model in the design space via a model card, a dictionary containing the chosen value for each design decision. We represent BERT-Tiny (Turc et al., 2019) in this formulation as

$$\Big\{l : 2, o : [\mathrm{SA}, \mathrm{SA}], h : [128, 128], n : [2, 2], f : \left[[512], [512]\right], p : [\mathrm{SDP}, \mathrm{SDP}]\Big\},$$

where the length of the list for every entry in f denotes the size of the feed-forward stack. We employ the model card to derive the computational graph of the model using smaller modules inferred from the design choice (details in Section 3.2).

3.1.2 Flexible Hidden Dimensions

Traditional transformer architectures restrict the flow of information using a constant embedding dimension across the network (a matrix of dimensions $N_T \times h$ from one layer to the next, where N_T denotes the number of tokens and h the hidden dimension; more details in Appendix A.1). Instead, we allow architectures in our design space to have *flexible* dimensions across layers. This enables different layers to capture information of different dimensions, as it learns more abstract features deeper into the network. For this, we make the following modifications:

- *Projection layers*: We add an affine projection network between encoder layers with dissimilar hidden sizes to transform encoding dimensionality.
- *Relative positional encoding*: The vanilla-BERT implementation uses an absolute positional encoding at the input and propagates it ahead through residual connections. Since we relax the restriction of a constant hidden size across layers, this does not apply to many models in our design space (as the learned projections for absolute encodings may not be one-to-one). Instead, we add a *relative* positional encoding at each layer (Shaw et al., 2018; Huang et al., 2018; Yang et al., 2019). Such an encoding can entirely replace absolute positional encodings with relative position representations learned using the SA mechanism. Whereas the SA module implementation remains the same as in previous works, for DSC-based and LT-based attention, we learn the relative encodings separately using SA and add them to the output of the attention module.

  Formally, let $\mathbf{Q}$ and $\mathbf{V}$ denote the query and the value layers, respectively. Let $\mathbf{R}$ denote the relative embedding tensor that the model needs to learn. Let $\mathbf{Z}$ and $\mathbf{X}$ denote the output and the input tensors of the attention module, respectively. In addition, let us define LT-based attention and DSC-based attention as $\mathrm{LT}(\cdot)$ and $\mathrm{DSC}(\cdot)$, respectively. Then,

$$\mathrm{RelativeAttention}(\mathbf{X}) = \mathrm{softmax}\left(\frac{\mathbf{Q}\mathbf{R}^{\top}}{\sqrt{d_{\mathbf{Q}}}}\right)\mathbf{V} \tag{1}$$

$$\mathbf{Z}_{\mathrm{LT}} = \mathrm{LT}(\mathbf{X}) + \mathrm{RelativeAttention}(\mathbf{X}) \tag{2}$$

$$\mathbf{Z}_{\mathrm{DSC}} = \mathrm{DSC}(\mathbf{X}) + \mathrm{RelativeAttention}(\mathbf{X}) \tag{3}$$

One should note that the proposed approach would only be applicable when the positional encodings are *trained* instead of being predetermined (Vaswani et al., 2017). The proposed *relative* and *trained* positional encodings enable us to make the dimensionality of data flow *flexible* across the network layers. This also means that each layer in the feed-forward stack can have a distinct hidden dimension.

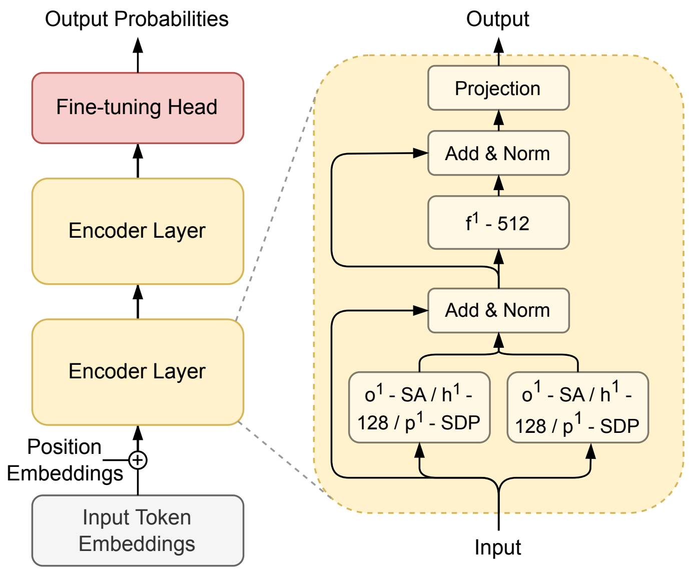

Figure 2: Block-level computation graph for BERT-Tiny in FlexiBERT. The projection layer implements an identity function since the hidden sizes of the input and output encoder layers are equal.

3.2 Graph Library

We now describe the graph library, *i.e.*, box (b) in Figure 1.

3.2.1 Block-level Computational Graphs

To learn a lower-dimensional dense manifold of the given design space, characterized by a large number of FlexiBERT models, we convert each model into a computational graph. We formulate this graph based on the forward flow of connections for each compute block. For our design space, we take all possible combinations of the compute blocks derived from the design decisions presented in Table 2 (see Appendix B.1 for a list of possible compute blocks supported in FlexiBERT). Using this design space and the set of compute blocks, we create all possible computational graphs within the design space for every transformer model. We then use recursive hashing as follows (Ying et al., 2019). For every node in this graph, we concatenate the hash of its input, that node, and its output, and then take the hash of the result. We use SHA256 as our hashing function. Doing this for all nodes and then hashing the concatenated hashes gives us the resultant hash of a given computational graph. This helps us detect isomorphic graphs and remove redundancy.

Figure 2 shows the block-level computational graph for BERT-Tiny. Using the connection patterns for every possible block permutation, we can generate multiple graphs for the given design space.

3.2.2 Levels of Hierarchy

The total number of possible graphs in the design space with *heterogeneous* feed-forward hidden layers is $\sim$3.32 billion. This is substantially larger than any transformer design space used in the past.

To make our approach tractable, we propose a hierarchical search method. We consider each model in the design space to be composed of multiple stacks containing at least one encoder layer. In the first step, we restrict each stack to $s = 2$ layers, where each layer in a stack shares the same design configuration. Naturally, this limits the search space size (we denote the set of all graphs in this space by $\mathcal{G}_1$). Hence, for instance, BERT-Tiny falls under $\mathcal{G}_1$ since the two encoder layers have the same configuration. We learn embeddings in this space and then run NAS to obtain the best-performing models. In the subsequent step, we consider a design space constituted by a finer-grained neighborhood of these models. We derive the neighborhood using pairwise crossover between the best-performing models and their neighbors in a space where the number of layers per stack is $s/2 = 1$, denoted by $\mathcal{G}_2$ (detailed explanation of the crossover operation in Appendix B.4). Finally, we include *heterogeneous* feed-forward stacks ($s = 1^*$) and denote the space by $\mathcal{G}_3$.

3.3 `Transformer2vec`

We now describe the `Transformer2vec` embedding and how we create an embedding library from a graph library $\mathcal{G}$, *i.e.*, box (c) in Figure 1.

3.3.1 Graph Edit Distance

Taking inspiration from Cheng et al. (2021) and Pennington et al. (2014), we train dense embeddings using *global* distance metrics, such as the Graph Edit Distance (GED) (Abu-Aisheh et al., 2015). These embeddings enable fast derivation of *neighboring* graphs in the active learning loop to facilitate the transfer of weights. We call them `Transformer2vec` embeddings. Unlike other approaches like the WL kernel, GED bakes in *domain knowledge* in graph comparisons, as explained in Section 2.3, by using a weighted sum of node insertion, deletion, and substitution costs.

For the GED computation, we first sort all possible compute blocks in the order of their computational complexity. Then, we weight the insertion and deletion cost for every block based on its index in this sorted list and the substitution cost between two blocks based on the difference in the indices in this sorted list. For computing the GED, we use a depth-first algorithm that requires less memory than traditional methods (Abu-Aisheh et al., 2015).

3.3.2 Training Embeddings

Given that there are S graphs in $\mathcal{G}$, we compute the GED for all possible computational graph pairs. This gives us a dataset of $N = \binom{S}{2}$ distances. To train the embedding, we minimize the mean-square error as the loss function between the predicted Euclidean distance and the corresponding GED. For the design space in consideration, we generate d-dimensional embeddings for every level of the hierarchy. Concretely, to train embedding $\mathcal{T}$, we minimize the loss

$$\mathcal{L}_{\mathcal{T}} = \sum_{1 \leq i \leq N, 1 \leq j \leq N, i \neq j} \Big(\mathbf{d}(\mathcal{T}(g_i), \mathcal{T}(g_j)) - \text{GED}(g_i, g_j) \Big)^2, \tag{4}$$

where $\mathbf{d}(\cdot, \cdot)$ is the Euclidean distance and we calculate the GED for the corresponding computational graphs g_i, $g_j \in \mathcal{G}$.

3.3.3 Weight Transfer among Neighboring Models

Pre-training each model in the design space is computationally expensive. Hence, we rely on weight sharing to initialize a query model in order to *directly* fine-tune it and minimize exploration time (details in Appendix B.3). We generate k nearest neighbors of a graph in the design space (we use $k = 100$ for our experiments). Then, naturally, we would like to transfer weights from the corresponding fine-tuned neighbor that is closest to the query, as such models intuitively have similar initial internal representations.

We calculate this similarity using a *biased overlap* measure that counts the number of encoder layers from the input to the output that are common to the current graph (*i.e.*, have exactly the same hyperparameter values). We stop counting the overlap on encountering different encoder layers, regardless of subsequent overlaps. In this ranking, there could be more than one graph with the same biased overlap with the current graph. Since the learned internal representations depend on the subsequent set of operations as well, we break ties based on the embedding distance of these graphs with the current graph. This gives us a set of neighbors, denoted by N_q for a model q, for every graph that are ranked based on both the biased overlap and the embedding distance. It helps increase the probability of finding a trained neighbor with high overlap.

As a hard constraint, we only consider transferring weights if the *biased overlap fraction* ($\mathcal{O}_f(q, n) =$ *biased overlap*$/l_q$, where q is the query model, $n \in N_q$ is the neighbor in consideration, and l_q is the number of layers in q) between the queried model and its neighbor is above a threshold τ. If the query-neighbor pair meets the constraint, we transfer the weights of the shared part from the corresponding neighbor to the query and fine-tune it. Otherwise, we pre-train the query. We denote the weight transfer operation by $W_q \leftarrow W_n$.

3.4 BOSHNAS

We now describe the BOSHNAS search policy, *i.e.*, box (d) in Figure 1.

3.4.1 Uncertainty Types

To overcome the challenges of an unexplored design space, it is important to consider the uncertainty in model predictions to guide the search process. Predicting model performance deterministically is not enough to estimate the next most probably best-performing model. We leverage the upper confidence bound (UCB) exploration on the predicted performance of unexplored models (Russell & Norvig, 2010). This could arise from not only the approximations in the surrogate modeling process but also parameter initializations and variations in model performance due to different training recipes. These are called *epistemic* and *aleatoric* uncertainties, respectively. The former, also called reducible uncertainty, arises from a lack of knowledge or information, and the latter, also called irreducible uncertainty, refers to the inherent variation in the system to be modeled.

3.4.2 Surrogate Model

In BOSHNAS, we use Monte-Carlo (MC) dropout (Gal & Ghahramani, 2016) and a Natural Parameter Network (NPN) (Wang et al., 2016) to model the epistemic and aleatoric uncertainties, respectively. The NPN not only helps with a distinct prediction of aleatoric uncertainty that we use for optimizing the training recipe once we are close to the optimal architecture but also serves as a superior model to Gaussian Processes, Bayesian Neural Networks (BNNs), and other Fully-Connected Neural Networks (FCNNs) (Tuli et al., 2021). Consider the NPN network $f_S(x;\theta)$ with a transformer embedding x as input and parameters θ. The output of such a network is the pair $(\mu, \sigma) \leftarrow f_S(x;\theta)$, where μ is the predicted mean performance and σ is the aleatoric uncertainty. To model the epistemic uncertainty, we use two deep surrogate models: (1) teacher (g_S) and (2) student (h_S) networks. It is a surrogate for the performance of a transformer, using its embedding x as an input. The teacher network is an FCNN with MC Dropout (parameters θ'). To compute the epistemic uncertainty, we generate n samples using $g_S(x, \theta')$. The standard deviation of the sample set is denoted by ξ. To run GOBI (Tuli et al., 2021) and avoid numerical gradients due to their poor performance, we use a student network (FCNN with parameters θ'') that directly predicts the output $\hat{\xi} \leftarrow h_S(x, \theta'')$, a surrogate of ξ (Tuli et al., 2022).

3.4.3 Active Learning and Optimization

For a design space $\mathcal{G}$, we first form an embedding space Δ by transforming all graphs in $\mathcal{G}$ using the `Transformer2vec` embedding. Assuming we have the three networks f_S, g_S, and h_S for our surrogate model, we use the following UCB estimate:

$$\text{UCB} = \mu + k_1 \cdot \sigma + k_2 \cdot \hat{\xi} = \Big(f_S(x,\theta)[0] + k_1 \cdot f_S(x;\theta)[1] \Big) + k_2 \cdot h_S(x, \theta''), \tag{5}$$

where $x \in \Delta$, k_1, and k_2 are hyperparameters.

To generate the next transformer to test, we execute GOBI using neural network inversion and the AdaHessian optimizer (Yao et al., 2021) that uses second-order updates to x ($\nabla_x^2 \text{UCB}$) up till convergence. From this, we get a new query embedding, x'. We find the nearest transformer architecture based on the Euclidean distance of all available transformer architectures in the design space Δ, giving the next closest model x. We fine-tune this model (or pre-train it if there is no nearby trained model with sufficient overlap; see Section 3.3) on the required task to obtain the respective performance. Once we receive the new datapoint, (x, o), we train the models using the loss functions on the updated corpus δ':

$$\begin{aligned}
\mathcal{L}_{\text{NPN}}(f_S, x, o) &= \sum_{(x,o)\in\delta'} \frac{(\mu - o)^2}{2\sigma^2} + \frac{1}{2}\ln \sigma^2, \\
\mathcal{L}_{\text{Teacher}}(g_S, x, o) &= \sum_{(x,o)\in\delta'} (g_S(x, \theta') - o)^2, \\
\mathcal{L}_{\text{Student}}(h_S, x) &= \sum_{x, \forall (x,o)\in\delta'} (h_S(x, \theta'') - \xi)^2,
\end{aligned} \tag{6}$$

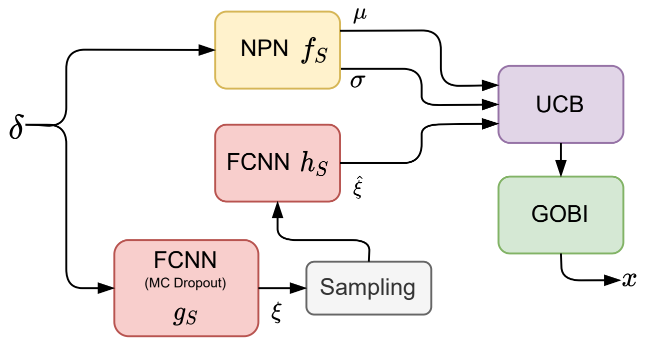

Figure 3: Overview of the BOSHNAS pipeline.

where $\mu, \sigma = f_S(x, \theta)$ and ξ is obtained by sampling $g_S(x, \theta')$. The first is the aleatoric loss to train the NPN model (Wang et al., 2016); the other two are squared-error loss functions. We run multiple random cold restarts of GOBI to get multiple queries for the next step in the search process.

Figure 3 shows different surrogate models in the BOSHNAS pipeline (f_S, g_S, and h_S) in the order of flow. As explained in Section 3.4, the NPN network (f_S) models the performance and the *aleatoric* uncertainty, and the student network (h_S) models the *epistemic* uncertainty from the teacher network (g_S).

Algorithm 1 summarizes the BOSHNAS workflow. Starting from an initial pre-trained set δ in the first level of the hierarchy $\mathcal{G}_1$, we run until convergence the following steps in a multi-worker compute cluster. To trade off between exploration and exploitation, we consider two probabilities: uncertainty-based exploration (α) and diversity-based exploration (β). With probability $1 - \alpha - \beta$, we run second-order GOBI using the surrogate model to minimize UCB in Eq. (5). Adding the converged point (x, o) in δ, we minimize the loss values in Eq. (6) (line 6 in Algorithm 1). We then generate a new query point, transfer weights from a neighboring model, and train it (lines 7-11). With α probability, we sample the search space using the combination of aleatoric and epistemic uncertainties, $k_1 \cdot \sigma + k_2 \cdot \hat{\xi}$, to find a point where the performance estimate is uncertain (line 15). To avoid getting stuck in a localized search subset, we also choose a random point with probability β (line 18). Once we converge in the first level, we continue with the second and third levels, $\mathcal{G}_2$ and $\mathcal{G}_3$, as described in Section 3.2.

4. Experimental Results

In this section, we show how the FlexiBERT model obtained from BOSHNAS outperforms the baselines.

4.1 Setup

For our experiments, we set the number of layers in each stack to $s = 2$ for the first level of the hierarchy, where models have the same configurations in every stack. In the second level, we use $s = 1$. Finally, we also make the feed-forward stacks heterogeneous ($s = 1^*$) in the third level (details given in Section 3.2). For the range of design choices in Table 2 and

```
Algorithm 1: BOSHNAS
Result: best architecture
1  Initialize: overlap threshold ($\tau$), convergence criterion, uncertainty sampling prob.
   ($\alpha$), diversity sampling prob. ($\beta$), surrogate model ($f_S$, $g_S$, and $h_S$) on initial
   corpus $\delta$, design space $g \in \mathcal{G} \Leftrightarrow x \in \Delta$;
2  while convergence criterion not met do
3      wait till a worker is free
4      if $prob \sim U(0,1) < 1 - \alpha - \beta$ then
5          $\delta \leftarrow \delta \cup \{$new performance point $(x, o)\}$;
6          fit(surrogate, $\delta$) using Eqn. (6);
7          $x \leftarrow$ GOBI($f_S$, $h_S$);                    /* Optimization step */
8          for $n$ in $N_x$ do
9              if $n$ is trained & $\mathcal{O}_f(x, n) \geq \tau$ then
10                 $W_x \leftarrow W_n$;
11                 send $x$ to worker;
12                 break;
13     else
14         if $1 - \alpha - \beta \leq prob. < 1 - \beta$ then
15             $x \leftarrow$ argmax($k_1 \cdot \sigma + k_2 \cdot \hat{\xi}$);    /* Uncertainty sampling */
16             send $x$ to worker;
17         else
18             send random $x$ to worker;              /* Diversity sampling */
```

setting $s = 2$, we obtain 9312 unique graphs after removing isomorphic graphs. We set the dimension of the `Transformer2vec` embedding to $d = 16$ after running a grid search. To do this, we minimize the distance prediction error while keeping d small using knee-point detection. We obtain the hyperparameter values in Algorithm 1 through grid search. We use overlap threshold $\tau = 80\%$, $\alpha = \beta = 0.1$, and $k_1 = k_2 = 0.5$ in our experiments. The convergence criterion is met in BOSHNAS when the change in performance is within 10^{-4} for five iterations. We give details of the model training process in Appendix B.2.

4.2 Pre-training and Fine-tuning Models

We adapt our pre-training recipe from the one used in RoBERTa, proposed by Liu et al. (2019), with slight variations in order to reduce the training budget (details in Appendix B.2).

We initialize the architecture space with models adapted from equivalent models presented in literature (Turc et al., 2019; Lee-Thorp et al., 2022; Jiang et al., 2020). The 12 initial models used to initiate the search process are BERT-Tiny, BERT-2/256 (with two encoder layers and a fixed hidden dimension of 256), BERT-4/128, BERT-Mini, FNet-Tiny, FNet-2/256, FNet-4/128, FNet-Mini, ConvBERT-Tiny, ConvBERT-2/256, ConvBERT-4/128, and ConvBERT-Mini (with p^j = DFT for FNets and $p^j = 9$ for ConvBERTs adapted from the original models). These models form the initial set δ in Algorithm 1.

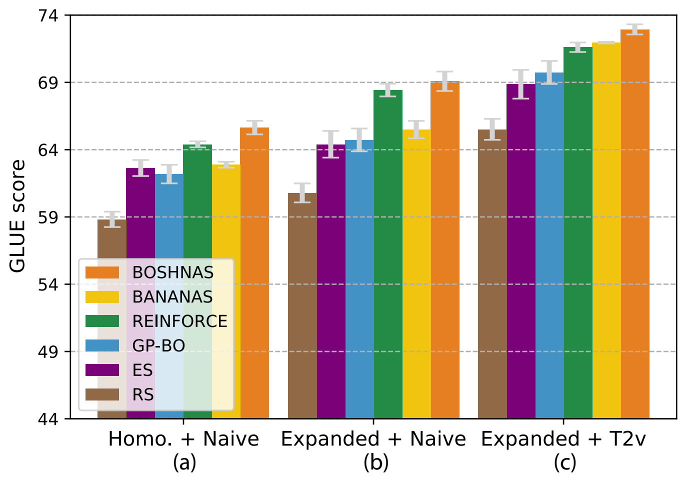

Figure 4: Bar plot comparing all NAS techniques with (a) naive embeddings and a design space of homogeneous models, (b) naive embeddings and an expanded design space of homogeneous and heterogeneous models, and (c) `Transformer2vec` (T2v) embeddings with the expanded design space. Plotted with 90% confidence intervals.

4.3 Ablation Study

We compare BOSHNAS against other popular techniques from the CNN space, namely Random Search (RS), ES, REINFORCE, GP-BO, and a recent state-of-the-art, BANANAS. We present performance on the GLUE benchmark.

Figure 4 shows the best GLUE scores reached by respective baseline NAS techniques along with BOSHNAS used with naive (*i.e.*, feature-based one-hot) or `Transformer2vec` embeddings on a representative design space. We use the space in the first level of the hierarchy (*i.e.*, with 9312 graphs, $s = 2$) and run all these algorithms in an active-learning scenario (all targeted homogeneous models form a subset of this space) over 50 runs for each algorithm. The plot highlights the fact that enhancing the richness of the design space enables the algorithms to search for more accurate models (6% improvement averaged across all models). We also see that `Transformer2vec` embeddings help NAS algorithms reach better-performing architectures (9% average improvement). Overall, BOSHNAS with the `Transformer2vec` embeddings performs the best in this representative design space, outperforming the state-of-the-art (*i.e.*, BANANAS on naive embeddings) by 13%.

Figure 5(a) shows the best GLUE score reached by each baseline NAS algorithm against the number of models it trained. Again, we perform these runs on the representative design space described above, using the `Transformer2vec` encodings. As observed in the figure, BOSHNAS reaches the best GLUE score. Ablation analysis justifies the need for heteroscedastic modeling and second-order optimization (see Figure 5(b)). The heteroscedastic model forces the optimization of the training recipe when the framework approaches optimal architectural design decisions. Second-order gradients, on the other hand, help the search avoid local optima and saddle points and also aid faster convergence.

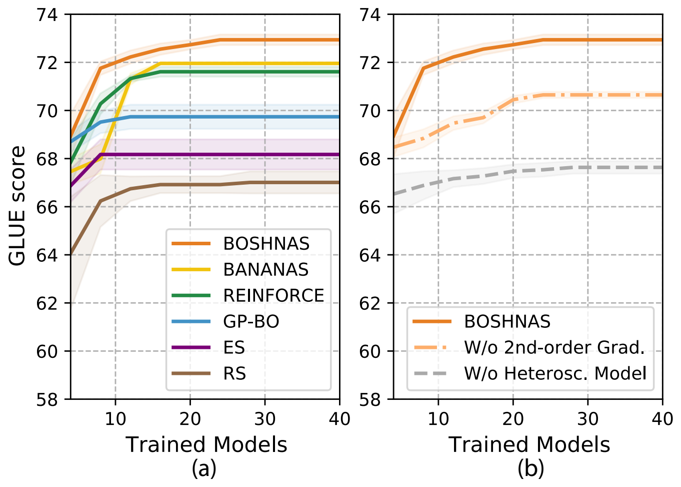

Figure 5: Performance results: (a) best GLUE score with trained models for NAS baselines and (b) ablation of BOSHNAS. Plotted with 90% confidence intervals.

Model	Parameters	CoLA	MNLI	MRPC	QNLI	QQP	RTE	SST-2	STS-B	WNLI	Avg.
BERT-Mini (Turc et al., 2019)	16.6M	0	74.8	71.8	84.1	66.4	57.9	85.9	73.3	62.3	64.0
NAS-BERT$_{10}$ (Xu et al., 2021)	10M	27.8	76.0	81.5	86.3	88.4	66.6	88.6	**84.8**	53.7*	72.6
FlexiBERT-Mini (ours, w/o S.)	**7.2M**	16.7	72.3	72.9	81.7	76.9	64.1	80.9	77.0	65.3	67.5
FlexiBERT-Mini (ours, w/o H.)	20M	12.3	74.4	72.3	76.4	76.3	59.5	81.2	75.4	67.8	66.2
FlexiBERT-Mini† (ours)	13.8M	**28.7**	**77.5**	82.3	86.9	87.8	67.6	**89.7**	83.0	51.8	72.7
FlexiBERT-Mini (ours)	16.1M	23.8	76.1	**82.4**	**87.1**	**88.7**	**69.0**	81.0	78.9	**69.3**	**72.9**

Table 3: Comparison between FlexiBERT and baselines. We evaluate the models on the development set of the GLUE benchmark. We use Matthews correlation for CoLA, Spearman correlation for STS-B, and accuracy for other tasks. We report MNLI on the matched set. We also include ablation models for BOSHNAS without second-order gradients (w/o S.) and without using the heteroscedastic model (w/o H.). Best (second-best) performance values are in boldface (underlined). *Xu et al. (2021) do not report the performance of NAS-BERT$_{10}$ on the WNLI dataset; we obtained it using an equivalent model in our design space. The FlexiBERT-Mini† model only optimizes performance on the first eight tasks for a fair comparison with NAS-BERT.

Table 3 shows the scores of the ablation models on the GLUE benchmarking tasks. We refer to the best model obtained from BOSHNAS in the Tiny-to-Mini space as FlexiBERT-Mini. Once we get the best architecture from the search process (using the same, albeit limited compute budget for feasible search times), we pre-train and fine-tune it on a larger compute budget (details in Appendix B.2). According to the table, FlexiBERT-Mini outperforms the baseline, NAS-BERT (Xu et al., 2021), by 0.4% on the GLUE benchmark. Since NAS-BERT finds the higher-performing architecture while only considering the first eight GLUE tasks (*i.e.*, without the WNLI dataset), for a fair comparison, we find a neighboring model in the

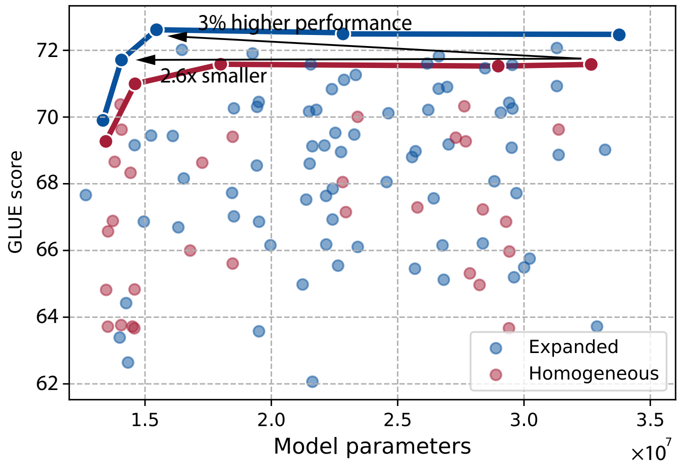

Figure 6: Performance frontiers of FlexiBERT on an expanded design space (under the constraints defined in Table 2) and for traditional homogeneous models.

Model	Parameters	BoolQ	CB	COPA	MultiRC	WiC	WSC	Avg.
BERT-Mini (Turc et al., 2019)	16.6M	62.1	22.2	40.0	59.1	52.2	61.4	49.5
FlexiBERT-Mini (ours)	**16.1M**	**62.2**	**47.4**	**45.0**	**63.2**	**54.3**	**63.5**	**55.9**

Table 4: Comparison between BERT-Mini and FlexiBERT-Mini on the SuperGLUE benchmark. For CB we report macro-average F1. We report accuracy for other tasks.

FlexiBERT design space that only optimizes performance on the first eight tasks. We call this model FlexiBERT-Mini†. We see that although FlexiBERT-Mini† does not have the highest GLUE score, it generally outperforms NAS-BERT$_{10}$ by significant margins on the first eight tasks.

Figure 6 demonstrates that FlexiBERT pushes to improve the performance frontier relative to traditional homogeneous architectures. In other words, the best-performing models in the expanded (Tiny-to-Mini) space outperform traditional models for the same number of parameters. Here, the homogeneous models incorporate the same design decisions for all encoder layers, even with the expanded set of operations (*i.e.*, including convolutional and LT-based attention operations). FlexiBERT-Mini has 3% fewer parameters than BERT-Mini and achieves 8.9% higher GLUE score. FlexiBERT achieves 3% higher performance than the best homogeneous model while the model with equivalent performance has 2.6× smaller size.

Table 4 shows the performance of FlexiBERT-Mini on SuperGLUE (Wang et al., 2019), which contains more challenging tasks relative to those in the GLUE benchmark. FlexiBERT-Mini outperforms BERT-Mini on the tasks in SuperGLUE. We give details of the selected set of training hyperparameters in Appendix B.2.

Model	Parameters	GLUE score
RoBERTa (Liu et al., 2019)	345M	88.5
FNet-Large (Lee-Thorp et al., 2022)	357M	81.9*
AutoTinyBERT (Yin et al., 2021)	85M	81.2*
DynaBERT (Hou et al., 2020)	345M	81.6*
NAS-BERT$_{60}$ (Xu et al., 2021)	60M	83.2*
AutoBERT-Zero Large (Gao et al., 2022)	318M	84.5*
FlexiBERT-Large (ours)	319M	**89.1/90.2***

Table 5: Comparison between FlexiBERT-Large (outside of the constraints defined in Table 2) and baselines on GLUE score. GLUE* scores reported do not consider the WNLI dataset.

4.4 Best Architecture in the Design Space

After running BOSHNAS for each level of the hierarchy, we obtain the respective best-performing models, whose model cards we present in Appendix B.5. From these best-performing models, we can extract the following rules that lead to high-performing transformer architectures:

- Models with DCT in the deeper layers are preferable for higher performance on the GLUE benchmark. Shallower layers prefer the traditional SDP-based attention heads.
- Models with more attention heads, but a smaller hidden dimension, are preferable in the deeper layers. On the other hand, fewer attention heads with higher hidden dimensions are preferable in shallower layers.
- Feed-forward networks with larger widths, but a smaller depth, are preferable in the deeper layers. Shallower layers prefer the opposite, *i.e.*, smaller width and higher depth.

Using these guidelines, we extrapolate the model card for FlexiBERT-Mini to get the design decisions for FlexiBERT-Large, which is an equivalent counterpart of BERT-Large (Devlin et al., 2019). Appendix B.5 presents the approach for extrapolation of hyperparameter choices from FlexiBERT-Mini to obtain FlexiBERT-Large. We train FlexiBERT-Large with the larger compute budget (see Appendix B.2) and show its GLUE score in Table 5. FlexiBERT-Large outperforms the baseline RoBERTa by 0.6% on the entire GLUE benchmarking suite and AutoBERT-Zero Large by 5.7% when only considering the first eight tasks.

Just like FlexiBERT-Large is the BERT-Large counterpart of FlexiBERT-Mini, we similarly form the BERT-Small and BERT-Base equivalents (Turc et al., 2019). Figure 7 presents the performance frontier of these FlexiBERT models with different baseline works. FlexiBERT consistently outperforms the baselines for different constraints on model size, thanks to its search in a vast, *heterogeneous*, and *flexible* design space of architectures.

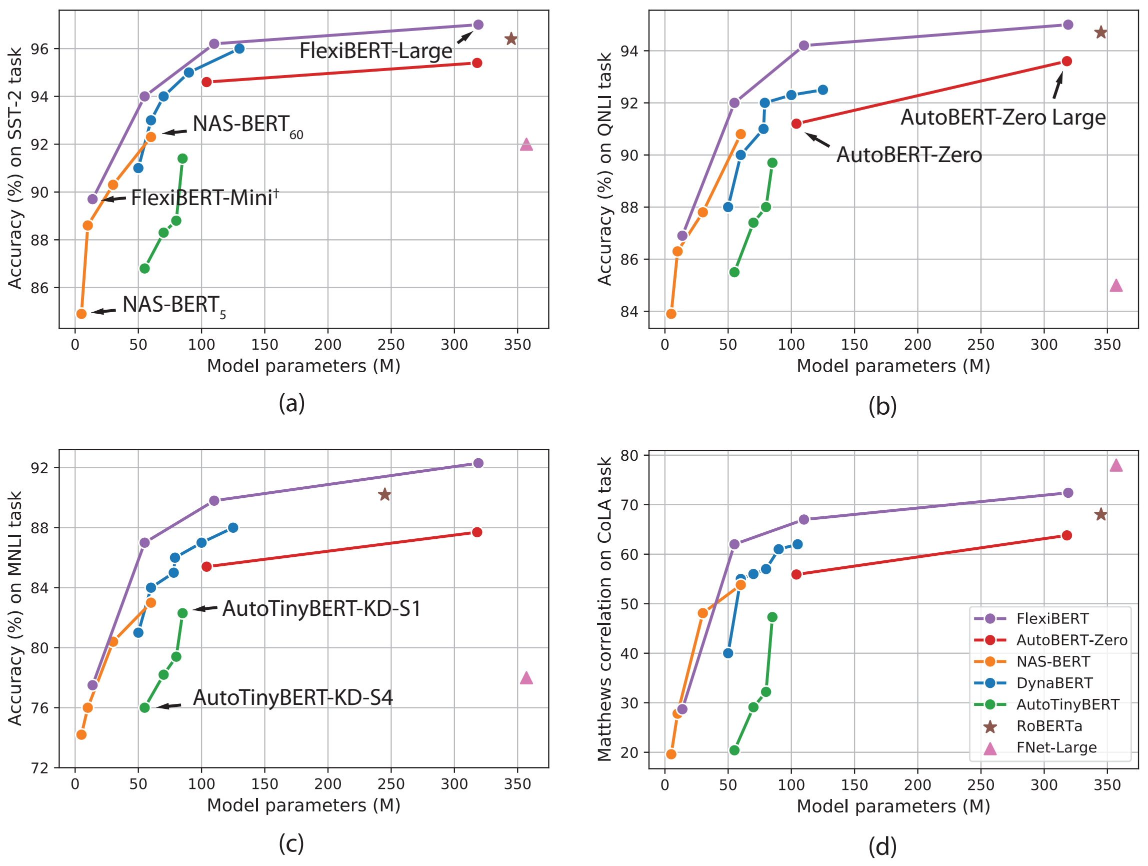

Figure 7: Performance of FlexiBERT and other baseline methods on various GLUE tasks: (a) SST-2, (b) QNLI, (c) MNLI (we plot accuracy of MNLI-m), and (d) CoLA.

5. Conclusion

In this work, we presented FlexiBERT, a suite of *heterogeneous* and *flexible* transformer models. We characterized the effects of this expanded design space and proposed a novel `Transformer2vec` embedding scheme to train a surrogate model that searches the design space for high-performance models. We described a novel NAS algorithm, BOSHNAS, and showed that it outperforms the state-of-the-art by 13%. The FlexiBERT-Mini model searched in this design space has a GLUE score that is 8.9% higher than BERT-Mini, while requiring 3% fewer parameters. It also outperforms the baseline, NAS-BERT$_{10}$ by 0.4%. A FlexiBERT model with equivalent performance as the best homogeneous model achieves 2.6× smaller size. FlexiBERT-Large outperforms the state-of-the-art models by at least 5.7% average accuracy on the first eight tasks in the GLUE benchmark.

Acknowledgments

This work was supported by NSF Grant No. CNS-1907381 and CCF-2203399. The experiments reported in this paper were substantially performed on the computational resources managed and supported by Princeton Research Computing at Princeton University. We also thank Xiaorun Wu for the initial discussions.

Appendix A. Background

Here, we discuss some supplementary background concepts.

A.1 Self-attention

Traditionally, transformers have relied on the SA operation. It is basically a trainable associative memory. We depict the vanilla SA operation as SDP and introduce the WMA operation in our design space as well. For a source vector $\mathbf{s}$ and a hidden-state vector $\mathbf{h}$:

$$\text{SDP} := \frac{\mathbf{s}^\top \mathbf{h}}{\sqrt{d}}, \ \text{WMA} := \mathbf{s}^\top \mathbf{W}_a \mathbf{h} \tag{7}$$

where d is the dimension of the source vector and $\mathbf{W}_a$ is a trainable weight matrix in the attention layer. Naturally, a WMA layer is more expressive than an SDP layer.

The SA mechanism used in the context of transformers also involves the softmax function and matrix multiplication. More concretely, in a multi-headed SA operation (with n heads), there are four matrices: $\mathbf{W}_i^q \in \mathbb{R}^{d_{inp} \times h/n}$, $\mathbf{W}_i^k \in \mathbb{R}^{d_{inp} \times h/n}$, $\mathbf{W}_i^v \in \mathbb{R}^{d_{inp} \times h/n}$, and $\mathbf{W}_i^o \in \mathbb{R}^{h/n \times d_{out}}$. The attention head takes the hidden states of the previous layer as input $\mathbf{H} \in \mathbb{R}^{N_T \times d_{inp}}$, where i refers to an attention head, d_{inp} is the input dimension, d_{out} is the output dimension, and h is the hidden dimension. We then calculate the output of the attention head ($\mathbf{H}_i \in \mathbb{R}^{N_T \times d_{out}}$) as follows:

$$\mathbf{Q}_i, \mathbf{K}_i, \mathbf{V}_i = \mathbf{H}\mathbf{W}_i^q, \mathbf{H}\mathbf{W}_i^k, \mathbf{H}\mathbf{W}_i^v \tag{8}$$

$$\mathbf{H}_i = \text{softmax}\left(\frac{\mathbf{Q}_i \mathbf{K}_i^\intercal}{\sqrt{h}}\right) \mathbf{V}_i \mathbf{W}_i^o \tag{9}$$

For traditional *homogenous* transformer models, d_{out} has to equal d_{inp} (usually, $d_{inp} = d_{out} = h$) due to the residual connections. However, thanks to the *relative* and *trained* positional encodings and the added projection layer at the end of each encoder ($\mathbf{W}^p \in \mathbb{R}^{d_{out} \times d_p}$), we can relax this constraint. This leads to an increased transformer model *flexibility* in the FlexiBERT design space.

A.2 Improving BERT's Performance

BERT is one of the most widely used transformer architectures (Devlin et al., 2019). Researchers have improved BERT's performance by revamping the pre-training technique. RoBERTa proposed a more robust pre-training approach to improve BERT's performance by considering *dynamic* masking in the Masked Language Modeling (MLM) objective (Liu et al., 2019). XLNet introduced Permuted Language Modeling (PLM) (Yang et al., 2019) and MPNet extended it by unifying MLM and PLM techniques (Song et al., 2020).

These methods achieve functional improvement in pre-training. Other approaches include techniques such as denoising autoencoders (Lewis et al., 2020).

On the other hand, Khetan and Karnin (2020) consider optimizing the set of *architectural* design decisions for BERT – number of encoder layers l, size of hidden embeddings h, number of attention heads a, size of the hidden layer in the feed-forward network f, etc. However, it is only concerned with pruning BERT and does not target optimization of accuracy over different tasks. Further, it has a limited search space consisting of only *homogeneous* models.

Appendix B. Experimental Details

We present the details of the experiments performed next.

B.1 Possible Compute Blocks

Based on the design space shown in Table 2, we consider all possible compute blocks, as presented next:

- For layer j, when the operation is SA, we have two or four heads among: h-128/SA-SDP, h-128/SA-WMA, h-256/SA-SDP, and h-256/SA-WMA. If the encoder layer has an LT operation, we have two or four heads among: h-128/LT-DFT, h-128/LT-DCT, h-256/LT-DFT, and h-256/LT-DCT; the latter entry being the type of LT operation. For a *convolutional* (DSC) operation, we have two or four heads among: h-128/DSC-5, h-128/DSC-9, h-256/DSC-5, and h-256/DSC-9; the latter integer referring to the kernel size.
- For layer j, the size of the hidden layer in the feed-forward network is either 512 or 1024. Also, the feed-forward network may either have just one hidden layer or a stack of three layers. At higher levels of the hierarchy in the hierarchical search framework (details in Section 3.2), all the layers in the stack of hidden layers have the same dimension until we relax this constraint in the last leg of the hierarchy.
- Other blocks: Add&Norm, Input, and Output.

B.2 Model Training

We pre-train our models with a combination of publicly available text corpora, viz. `BookCorpus` (BookC) (Zhu et al., 2015), `Wikipedia English` (Wiki), `OpenWebText` (OWT) (Gokaslan & Cohen, 2019), and `CC-News` (CCN) (Mackenzie et al., 2020). We borrow most training hyperparameters from RoBERTa. We set the batch size to 256, learning rate warmed up over the first $10,000$ steps to its peak value at 1×10^{-5} that then decays linearly, weight decay to 0.01, Adam scheduler's parameters $\beta_1 = 0.9$, $\beta_2 = 0.98$ (shown to improve stability; Liu et al., 2019), $\epsilon = 1 \times 10^{-6}$, and run pre-training for $1,000,000$ steps.

Once we find the best models, we pre-train and fine-tune the selected models with a larger compute budget. For pre-training, we add the `C4` dataset (Raffel et al., 2020) and train for $3,000,000$ steps before fine-tuning. We also fine-tune on each GLUE task for 10 epochs instead of 5 (further details given below). We executed this extended training process for the FlexiBERT-Mini and FlexiBERT-Large models. Table 6 shows the improvement in performance of FlexiBERT-Mini trained using knowledge transfer (where we transfer the weights from a nearby trained model) after additional training. When compared to the model

Model	Pre-training data	Pre-training steps	Fine-tuning epochs	GLUE score
FlexiBERT-Mini				
w/ knowledge transfer	BookC, Wiki, OWT, CCN	1,000,000	5	69.7
+ pre-training from scratch	BookC, Wiki, OWT, CCN	1,000,000	5	70.4
+ larger compute budget	BookC, Wiki, OWT, CCN, C4	3,000,000	10	72.9

Table 6: Performance of FlexiBERT-Mini from BOSHNAS after knowledge transfer from a nearby trained model, and after pre-training from scratch along with a larger compute budget.

Task	Learning rate	Batch size	Training size	Deviation
CoLA	2.0×10^{-4}	64	9K	2.1
MNLI	9.4×10^{-5}	64	393K	3.4
MRPC	2.23×10^{-5}	32	4K	3.5
QNLI	5.03×10^{-5}	128	105K	2.6
QQP	3.7×10^{-4}	64	364K	1.3
RTE	1.9×10^{-4}	128	3K	5.1
SST-2	1.2×10^{-4}	128	67K	6.9
STS-B	7.0×10^{-5}	32	7K	4.1
WNLI	4.0×10^{-5}	128	634	4.7

Table 7: Hyperparameters used for fine-tuning FlexiBERT-Mini on the GLUE tasks along with the size of the training set and deviation in performance.

directly fine-tuned after knowledge transfer, we see only a marginal improvement when we pre-train from scratch. This reaffirms the advantage of knowledge transfer that it reduces training time (see Appendix B.3) with a negligible loss in performance. This is a consequence of a high overlap threshold, *i.e.*, 80%, which results in a low performance loss at the cost of maximizing the probability of finding a pre-trained neighbor. Training with a more significant compute budget further improves performance on the GLUE benchmark, validating the importance of data size and diversity in pre-training (Liu et al., 2019). Running a full-fledged BOSHNAS on the larger design space (*i.e.*, with layers from 2 to 24, Tiny-to-Large) can be an easy extension of this work.

While running BOSHNAS, we fine-tune our models on the nine GLUE tasks over five epochs and a batch size of 64, where we implement early stopping. We also run automatic hyperparameter tuning for the fine-tuning process using the Tree-structured Parzen Estimator algorithm (Akiba et al., 2019). The learning rate is randomly selected logarithmically in the $[2 \times 10^{-5},\ 5 \times 10^{-4}]$ range, and the batch size in $\{32, 64, 128\}$ uniformly. Table 7 (Table 8) shows the best hyperparameters for fine-tuning of each GLUE (SuperGLUE) task selected using this auto-tuning technique. This hyperparameter optimization uses random initialization every time, which results in variation in performance each time the model is queried (see *aleatoric* uncertainty explained in Section 3.4).

Since some tasks in the GLUE benchmark are very small, one expects a large deviation in performance as we change the training recipe. Table 7 also shows the deviation in

Task	Learning rate	Batch size
BoolQ	7.5×10^{-5}	64
CB	2.1×10^{-4}	64
COPA	6.3×10^{-5}	32
MultiRC	1.0×10^{-4}	128
WiC	8.6×10^{-5}	128
WSC	4.4×10^{-4}	64

Table 8: Hyperparameters used for fine-tuning FlexiBERT-Mini on the SuperGLUE tasks.

performance on GLUE tasks (as reported in Table 3). We see a large variation in smaller datasets. We observe marginal deviation in performance in large datasets like MNLI, QNLI, and QQP. These deviations correspond to the saliency of aleatoric uncertainty on the training recipe hyperparameters. Our hyperparameter tuning search method chooses the best training recipe resulting in the highest performance.

We have included baselines trained with the *pre-training + fine-tuning* procedure as proposed by Turc et al. (2019) for like-for-like comparisons, and not the *knowledge distillation* counterparts (Xu et al., 2021). Nevertheless, FlexiBERT is orthogonal to (and thus can easily be combined with) knowledge distillation because FlexiBERT focuses on searching the best architecture, while knowledge distillation focuses on better training of a given architecture.

All models were trained on NVIDIA A100 GPUs and 2.6 GHz AMD EPYC Rome processors. The entire process of running BOSHNAS for all levels of the hierarchy took around 300 GPU-days of training.

B.3 Knowledge Transfer

Recent works leverage knowledge transfer. However, these methods are restricted to long short-term memories and simple recurrent neural networks (Mazzawi et al., 2019). Wang et al. (2020) train a super-transformer and share its weights with smaller models. However, this is not feasible for diverse *heterogeneous* and *flexible* architectures. We propose the use of knowledge transfer in transformers for the first time, to the best of our knowledge, by comparing weights with computational graphs of *nearby* models. Furthermore, previous works only consider a static training recipe for all the models in the design space, an assumption we relax in our experiments. We *directly* fine-tune models for which *nearby* models are already pre-trained. We test for this using the *biased overlap* metric defined in Section 3.3. Figure 8 presents the time gains from knowledge transfer when we fine-tune on all GLUE tasks. Since we can directly fine-tune some percentage of models, thanks to their neighboring pre-trained models, we were able to speed up the overall training time by 38%.

B.4 Crossover between Transformer Models

We obtain new transformer models of the subsequent level in the hierarchy by taking a crossover between the best models in the previous level (which had layers per stack = s) and their neighbors. We choose the stack configuration of the children from all unique

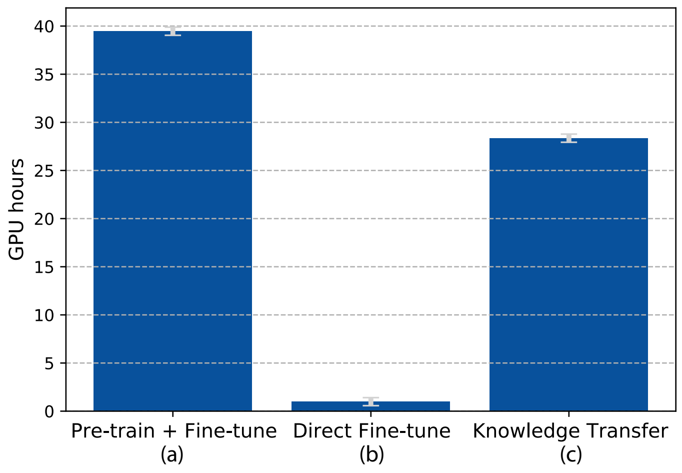

Figure 8: Bar plot showing average time for training a transformer model (in GPU-hours) with and without knowledge transfer. (a) Pre-train + Fine-tune: total training time. (b) Direct fine-tuning: training time for a pre-trained model. (c) Knowledge Transfer: training using weight transfer from a trained *nearby* model gives 38% speedup. Plotted with 90% confidence intervals.

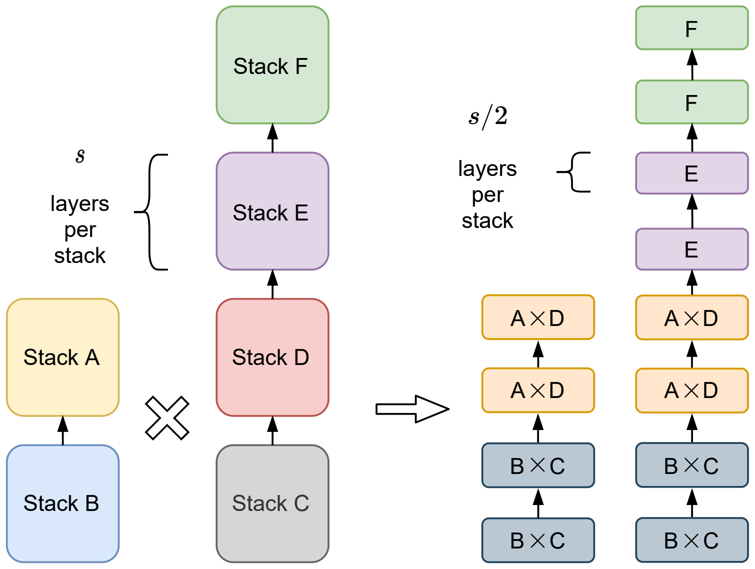

Figure 9: Crossover between two parent models yields a finer-grained design space. Each stack configuration in the children is derived from the product of the parent design choices at the same depth.

hyperparameter values present in the parent models at the same depth. We show a simple example of this scheme in Figure 9. First, we compute the design space of permissible operation blocks for layers in the stack, s, by the product of the individual design choices of the parents for that stack. We then independently form these new layers with the new constraint of $s/2$ layers having the same choice of hyperparameter values. Expanding the design space in such a fashion retains the original hyperparameters that give good performance while also exploring the internal representations learned by combinations of the hyperparameters at the same level.

B.5 Best-performing Models

From different hierarchy levels ($s = 2, 1,$ and 1^*), we get the respective best-performing models after running BOSHNAS as follows:

$$s = 2 : \Big\{ l : 4, o : [\text{LT}, \text{LT}, \text{LT}, \text{LT}], h : [256, 256, 256, 256], n : [4, 4, 2, 2],$$

$$f : [[1024], [1024], [512, 512, 512], [512, 512, 512]], p : [\text{DCT}, \text{DCT}, \text{DCT}, \text{DCT}] \Big\}$$

$$s = 1 : \Big\{ l : 4, o : [\text{SA}, \text{SA}, \text{LT}, \text{LT}], h : [256, 256, 128, 128], n : [2, 2, 4, 4],$$

$$f : [[512, 512, 512], [512, 512, 512], [1024], [1024]], p : [\text{SDP}, \text{SDP}, \text{DCT}, \text{DCT}] \Big\}$$

where, in the last leg of the hierarchy, the stack length is 1, but the feed-forward stacks are also heterogeneous (see Section 3.2). Both $s = 1$ and $s = 1^*$ gave the same solution despite finer granularity in the latter case. Thus, the second model card above is that of FlexiBERT-Mini.

We present the model cards of the FlexiBERT-Mini ablation models, as shown in Table 3, below:

$$(\text{w/o S.}) : \Big\{ l : 2, o : [\text{SA}, \text{SA}], h : [128, 128], n : [4, 4], f : [[1024], [1024]], p : [\text{SDP}, \text{WMA}] \Big\}$$

$$(\text{w/o H.}) : \Big\{ l : 4, o : [\text{LT}, \text{LT}, \text{SA}, \text{SA}], h : [256, 256, 128, 128], n : [4, 4, 4, 4],$$

$$f : [[1024, 1024, 1024], [1024, 1024, 1024], [512, 512, 512], [512, 512, 512]],$$

$$p : [\text{DCT}, \text{DCT}, \text{SDP}, \text{SDP}] \Big\}$$

Figure 10 shows a working schematic of the design choices in the FlexiBERT-Mini and FlexiBERT-Large models. As explained in Section 4.4, we obtain FlexiBERT-Large by extrapolating the design choices in FlexiBERT-Mini to obtain a BERT-Large counterpart (Devlin et al., 2019).

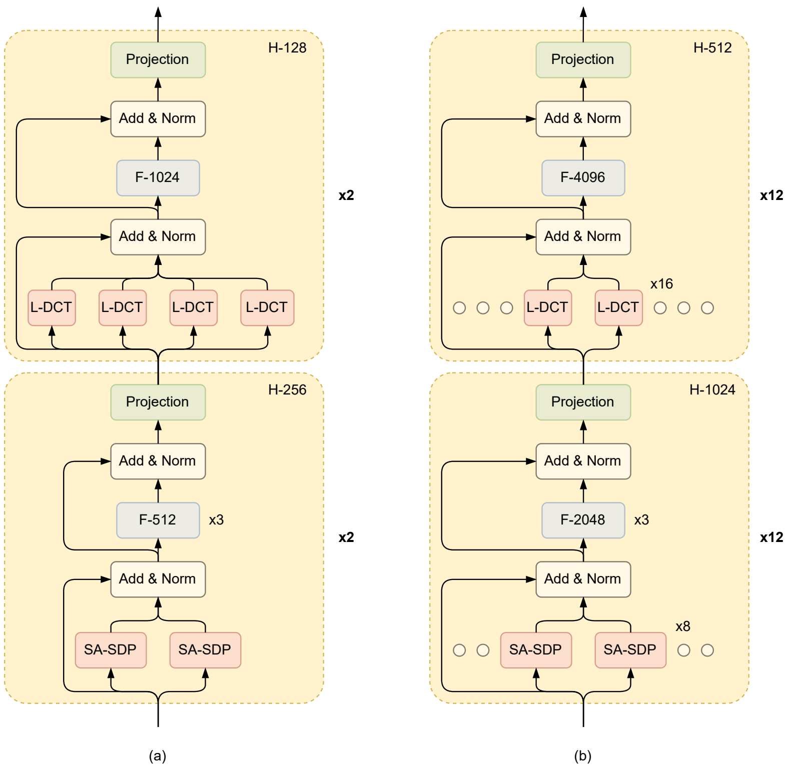

Figure 10: Obtained FlexiBERT models after running the BOSHNAS pipeline: (a) FlexiBERT-Mini and its design choices extrapolated to obtain (b) FlexiBERT-Large.

References

Abu-Aisheh, Z., Raveaux, R., Ramel, J.-Y., & Martineau, P. (2015). An exact graph edit distance algorithm for solving pattern recognition problems. In *Proceedings of the International Conference on Pattern Recognition Applications and Methods*, Vol. 1, pp. 271–278.

Akiba, T., Sano, S., Yanase, T., Ohta, T., & Koyama, M. (2019). Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 2623–2631.

Chen, D., Li, Y., Qiu, M., Wang, Z., Li, B., Ding, B., Deng, H., Huang, J., Lin, W., & Zhou, J. (2021). AdaBERT: Task-adaptive BERT compression with differentiable neural architecture search. In *Proceedings of the 29th International Joint Conference on Artificial Intelligence*, pp. 2463–2469.

Cheng, H.-P., Zhang, T., Zhang, Y., Li, S., Liang, F., Yan, F., Li, M., Chandra, V., Li, H., & Chen, Y. (2021). NASGEM: Neural architecture search via graph embedding method. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 35, pp. 7090–7098.

Conneau, A., & Lample, G. (2019). Cross-lingual language model pretraining. In *Advances in Neural Information Processing Systems*, Vol. 32, pp. 7059–7069.

Dang, D.-C., Eremeev, A., & Lehre, P. K. (2021). Escaping local optima with non-elitist evolutionary algorithms. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 35, pp. 12275–12283.

Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, Vol. 1, pp. 4171–4186.

Gal, Y., & Ghahramani, Z. (2016). Dropout as a Bayesian approximation: Representing model uncertainty in deep learning. In *Proceedings of The 33rd International Conference on Machine Learning*, Vol. 48, pp. 1050–1059.

Gao, J., Xu, H., Shi, H., Ren, X., Yu, P. L. H., Liang, X., Jiang, X., & Li, Z. (2022). AutoBERT-Zero: Evolving bert backbone from scratch. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 36, pp. 10663–10671.

Gokaslan, A., & Cohen, V. (2019). OpenWebText corpus. `http://Skylion007.github.io/OpenWebTextCorpus`.

He, X., Zhao, K., & Chu, X. (2021). AutoML: A survey of the state-of-the-art. *Knowledge-Based Systems*, *212*, 106622.

Hou, L., Huang, Z., Shang, L., Jiang, X., Chen, X., & Liu, Q. (2020). DynaBERT: Dynamic BERT with adaptive width and depth. In *Advances in Neural Information Processing Systems*, Vol. 33, pp. 9782–9793.

Huang, C.-Z. A., Vaswani, A., Uszkoreit, J., Simon, I., Hawthorne, C., Shazeer, N., Dai, A. M., Hoffman, M. D., Dinculescu, M., & Eck, D. (2018). Music transformer: Generating music with long-term structure. In *Proceedings of the International Conference on Learning Representations*.

Jiang, Z.-H., Yu, W., Zhou, D., Chen, Y., Feng, J., & Yan, S. (2020). ConvBERT: Improving BERT with span-based dynamic convolution. In *Advances in Neural Information Processing Systems*, Vol. 33, pp. 12837–12848.

Khetan, A., & Karnin, Z. (2020). schuBERT: Optimizing elements of BERT. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 2807–2818.

Kruskal, J. B. (1964). Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis. *Psychometrika*, *29*(1), 1–27.

Lee-Thorp, J., Ainslie, J., Eckstein, I., & Ontanon, S. (2022). FNet: Mixing tokens with Fourier transforms. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 4296–4313.

Lewis, M., Liu, Y., Goyal, N., Ghazvininejad, M., Mohamed, A., Levy, O., Stoyanov, V., & Zettlemoyer, L. (2020). BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 7871–7880.

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., & Stoyanov, V. (2019). RoBERTa: A robustly optimized BERT pretraining approach. *CoRR*, *abs/1907.11692*.

Lu, Z., Whalen, I., Boddeti, V., Dhebar, Y., Deb, K., Goodman, E., & Banzhaf, W. (2019). NSGA-Net: Neural architecture search using multi-objective genetic algorithm. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 419–427.

Luong, T., Pham, H., & Manning, C. D. (2015). Effective approaches to attention-based neural machine translation. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pp. 1412–1421.

Mackenzie, J., Benham, R., Petri, M., Trippas, J. R., Culpepper, J. S., & Moffat, A. (2020). CC-News-En: A large English news corpus. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*, pp. 3077–3084.

Mazzawi, H., Gonzalvo, X., Kracun, A., Sridhar, P., Subrahmanya, N., Lopez-Moreno, I., Park, H., & Violette, P. (2019). Improving keyword spotting and language identification via neural architecture search at scale. In *Proceedings of Interspeech*, pp. 1278–1282.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., & Dean, J. (2013). Distributed representations of words and phrases and their compositionality. In *Advances in Neural Information Processing Systems*, Vol. 26, pp. 3111–3119.

Narayanan, A., Chandramohan, M., Venkatesan, R., Chen, L., Liu, Y., & Jaiswal, S. (2017). graph2vec: Learning distributed representations of graphs. *CoRR*, *abs/1707.05005*.

Pennington, J., Socher, R., & Manning, C. (2014). GloVe: Global vectors for word representation. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pp. 1532–1543.

Pham, H., Guan, M., Zoph, B., Le, Q., & Dean, J. (2018). Efficient neural architecture search via parameters sharing. In *Proccedings of the 35th International Conference on Machine Learning*, pp. 4095–4104.

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., & Liu, P. J. (2020). Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, *21*(140), 1–67.

Real, E., Aggarwal, A., Huang, Y., & Le, Q. V. (2019). Regularized evolution for image classifier architecture search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33, pp. 4780–4789.

Ru, R., Esperanca, P., & Carlucci, F. M. (2020). Neural architecture generator optimization. In *Advances in Neural Information Processing Systems*, Vol. 33, pp. 12057–12069.

Russell, S., & Norvig, P. (2010). *Artificial Intelligence: A Modern Approach* (3rd edition). Prentice Hall.

Shaw, P., Uszkoreit, J., & Vaswani, A. (2018). Self-attention with relative position representations. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, Vol. 2, pp. 464–468.

Shervashidze, N., Schweitzer, P., van Leeuwen, E. J., Mehlhorn, K., & Borgwardt, K. M. (2011). Weisfeiler-Lehman graph kernels. *Journal of Machine Learning Research*, *12*(77), 2539–2561.

Siems, J., Zimmer, L., Zela, A., Lukasik, J., Keuper, M., & Hutter, F. (2020). NAS-Bench-301 and the case for surrogate benchmarks for neural architecture search. *CoRR*, *abs/2008.09777*.

Snoek, J., Larochelle, H., & Adams, R. P. (2012). Practical Bayesian optimization of machine learning algorithms. In *Advances in Neural Information Processing Systems*, Vol. 25, pp. 2951–2959.

So, D., Mańke, W., Liu, H., Dai, Z., Shazeer, N., & Le, Q. V. (2021). Searching for efficient transformers for language modeling. In *Advances in Neural Information Processing Systems*, Vol. 34, pp. 6010–6022.

So, D. R., Liang, C., & Le, Q. V. (2019). The evolved transformer. *CoRR*, *abs/1901.11117*.

Song, K., Tan, X., Qin, T., Lu, J., & Liu, T.-Y. (2020). MPNet: Masked and permuted pre-training for language understanding. In *Advances in Neural Information Processing Systems*, Vol. 33, pp. 16857–16867.

Sun, Z., Yu, H., Song, X., Liu, R., Yang, Y., & Zhou, D. (2020). MobileBERT: A compact task-agnostic BERT for resource-limited devices. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 2158–2170.

Tan, M., & Le, Q. (2019). EfficientNet: Rethinking model scaling for convolutional neural networks. In *Proceedings of the 36th International Conference on Machine Learning*, pp. 6105–6114.

Tuli, S., Casale, G., & Jennings, N. R. (2022). GOSH: Task scheduling using deep surrogate models in fog computing environments. *IEEE Transactions on Parallel and Distributed Systems*, *33*(11), 2821–2833.

Tuli, S., Poojara, S. R., Srirama, S. N., Casale, G., & Jennings, N. R. (2021). COSCO: Container orchestration using co-simulation and gradient based optimization for fog computing environments. *IEEE Transactions on Parallel and Distributed Systems*, *33*(1), 101–116.

Turc, I., Chang, M., Lee, K., & Toutanova, K. (2019). Well-read students learn better: The impact of student initialization on knowledge distillation. *CoRR*, *abs/1908.08962*.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is all you need. In *Advances in Neural Information Processing Systems*, Vol. 30, pp. 5998–6008.

Wang, A., Pruksachatkun, Y., Nangia, N., Singh, A., Michael, J., Hill, F., Levy, O., & Bowman, S. (2019). SuperGLUE: A stickier benchmark for general-purpose language understanding systems. In *Advances in Neural Information Processing Systems*, Vol. 32.

Wang, A., Singh, A., Michael, J., Hill, F., Levy, O., & Bowman, S. (2018). GLUE: A multi-task benchmark and analysis platform for natural language understanding. In *Proceedings of the EMNLP Workshop on BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, pp. 353–355.

Wang, H., Wu, Z., Liu, Z., Cai, H., Zhu, L., Gan, C., & Han, S. (2020). HAT: Hardware-aware transformers for efficient natural language processing. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 7675–7688.

Wang, H., Shi, X., & Yeung, D.-Y. (2016). Natural-parameter networks: A class of probabilistic neural networks. In *Advances in Neural Information Processing Systems*, Vol. 29, pp. 118–126.

White, C., Neiswanger, W., & Savani, Y. (2021a). BANANAS: Bayesian optimization with neural architectures for neural architecture search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 35, pp. 10293–10301.

White, C., Zela, A., Ru, B., Liu, Y., & Hutter, F. (2021b). How powerful are performance predictors in neural architecture search?. In *Advances in Neural Information Processing Systems*.

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, *8*(3–4), 229–256.

Xu, J., Tan, X., Luo, R., Song, K., Li, J., Qin, T., & Liu, T.-Y. (2021). NAS-BERT: Task-agnostic and adaptive-size BERT compression with neural architecture search. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, pp. 1933–1943.

Xu, K., Hu, W., Leskovec, J., & Jegelka, S. (2019). How powerful are graph neural networks?. In *Proceedings of the International Conference on Learning Representations*.

Yang, Z., Dai, Z., Yang, Y., Carbonell, J., Salakhutdinov, R. R., & Le, Q. V. (2019). XLNet: Generalized autoregressive pretraining for language understanding. In *Advances in Neural Information Processing Systems*, Vol. 32.

Yao, Z., Gholami, A., Shen, S., Mustafa, M., Keutzer, K., & Mahoney, M. (2021). ADAHESSIAN: An adaptive second order optimizer for machine learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 35, pp. 10665–10673.

Yin, Y., Chen, C., Shang, L., Jiang, X., Chen, X., & Liu, Q. (2021). AutoTinyBERT: Automatic hyper-parameter optimization for efficient pre-trained language models. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing*, Vol. 1, pp. 5146–5157.

Ying, C., Klein, A., Christiansen, E., Real, E., Murphy, K., & Hutter, F. (2019). NAS-Bench-101: Towards reproducible neural architecture search. In *Proceedings of the 36th International Conference on Machine Learning*, Vol. 97, pp. 7105–7114.

Yu, Y., & Jha, N. K. (2022). SPRING: A sparsity-aware reduced-precision monolithic 3D CNN accelerator architecture for training and inference. *IEEE Transactions on Emerging Topics in Computing*, *10*(1), 237–249.

Zhang, X., Zhou, X., Lin, M., & Sun, J. (2018). ShuffleNet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 6848–6856.

Zhu, Y., Kiros, R., Zemel, R., Salakhutdinov, R., Urtasun, R., Torralba, A., & Fidler, S. (2015). Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. In *Proceedings of the IEEE International Conference on Computer Vision*, pp. 19–27.

Zoph, B., & Le, Q. (2017). Neural architecture search with reinforcement learning. In *International Conference on Learning Representations*.

Zoph, B., Vasudevan, V., Shlens, J., & Le, Q. V. (2018). Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 8697–8710.