SpeedLimit: Neural Architecture Search for Quantized Transformer Models

Yuji Chai [*1,2] **Luke Bailey** [*1] **Yunho Jin** [1] **Matthew Karle** [1] **Glenn G. Ko** [1,2]
David Brooks [1] **Gu-Yeon Wei** [1] **H. T. Kung** [1]

Abstract

While research in the field of transformer models has primarily focused on enhancing performance metrics such as accuracy and perplexity, practical applications in industry often necessitate a rigorous consideration of inference latency constraints. Addressing this challenge, we introduce *SpeedLimit*, a novel Neural Architecture Search (NAS) technique that optimizes accuracy whilst adhering to an upper-bound latency constraint. Our method incorporates 8-bit integer quantization in the search process to outperform the current state-of-the-art technique. Our results underline the feasibility and efficacy of seeking an optimal balance between performance and latency, providing new avenues for deploying state-of-the-art transformer models in latency-sensitive environments.

1. Introduction

Transformer models are incredibly capable across diverse domains, most notably natural language processing (Brown et al., 2020; Radford et al., 2019; Devlin et al., 2018; Liu et al., 2019) and computer vision (Dosovitskiy et al., 2020; Parmar et al., 2018; Chen et al., 2021). In recent years, transformer capability has been driven in large parts by expanding model sizes (Kaplan et al., 2020). With state-of-the-art (SOTA) models now exceeding hundreds of gigabytes in size (Chowdhery et al., 2022; Smith et al., 2022), it has become incredibly costly to deploy and maintain services that rely on them. Additionally, the models are so large that low-latency deployments have become impossible without paying for the most advanced hardware. In many application areas, upper bound latency restrictions are required, for example chatbot services or real time object recognition. These applications occur across many diverse settings, from large cloud computing clusters to small edge devices.

[*]Equal contribution [1]Harvard University [2]Stochastic Inc. Correspondence to: Luke Bailey <lukebailey@college.harvard.edu>.

Work presented at the ES-FoMo Workshop at ICML 2023. Copyright 2023 by the author(s).

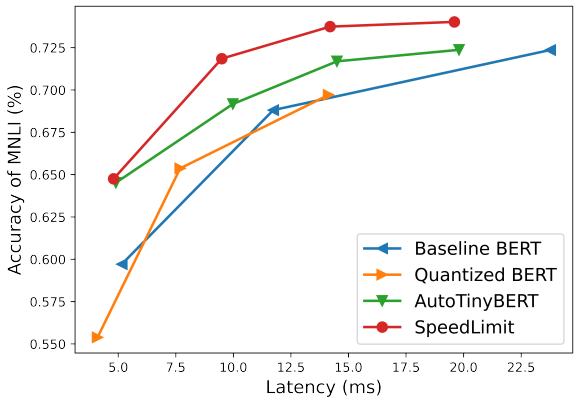

Figure 1. Latency vs. MNLI task accuracy. *SpeedLimit* outperforms AutoTinyBERT (Yin et al., 2021), full precision BERT (baseline), and int8 quantized BERT (Quantized) across all latency requirements tested.

In these scenarios, the latency requirements are fixed and machine learning engineers must find the most performant model that meets said requirement. To solve this problem we propose *SpeedLimit*, a neural architecture search (NAS) technique for finding optimal transformer models with fixed upper bound latency constraints.

In this paper we focus on BERT (Devlin et al., 2018) as our motivating transformer architecture. The current SOTA NAS process to find BERT models that satisfy a latency requirement, called AutoTinyBERT (Yin et al., 2021), focuses solely on models with float32 parameters. Seen as the goal of the algorithm is to find the most accurate models at a certain latency target, it seems naive to not incorporate 8-bit integer (int8) quantization into this process in light of the inference speedup associated with quantized models on increasingly common commodity hardware. Additionally, as we will show, naively quantizing the float32 model that other NAS techniques create does not guarantee the best int8 model. We thus aim to fix this problem by incorporating quantization into the NAS process itself. This means we search directly for the best int8 model at a given latency constraint.

SpeedLimit applies a two-stage NAS technique to find such an optimal int8 quantized architecture (Cai et al., 2020). Our approach involves training a Supermodel via knowledge dis-

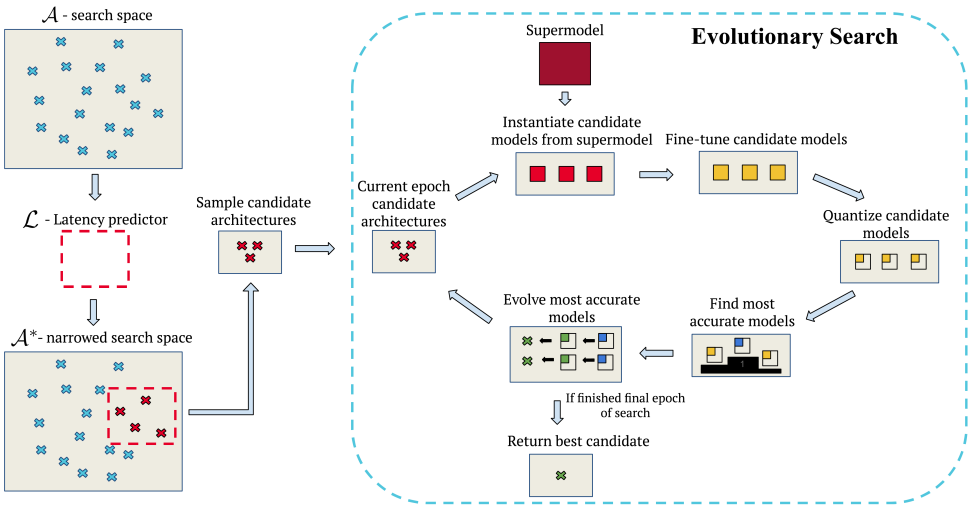

Figure 2. SpeedLimit pipeline.

tillation, and then employing an evolutionary search to find the best quantized subnetworks of this Supermodel, called candidates. Two-stage NAS streamlines the evaluation process by eliminating the need for extensive training from scratch, which is particularly vital for large SOTA transformers. Candidates are assessed based on their latency and accuracy, with the top-performing one chosen as the final model. *SpeedLimit* is able to find models that outperform those found by AutoTinyBERT and a baseline of default BERT models (both full precision and int8 quantized) in terms of accuracy and latency.

2. Background and related work

Neural architecture search (NAS) is a technique for automatically finding optimal deep neural network architectures (Elsken et al., 2019). NAS techniques define some search space of architectures, and a search strategy over this space. Common search algorithms include evolutionary search, reinforcement learning, Bayesian optimization, and gradient based methods (Elsken et al., 2019). A search algorithm works in tandem with some performance estimation strategy that the algorithm uses to query for the runtime performance of each architecture during the search. NAS techniques can be broadly classified into two categories according to their performance estimation strategy: one-stage and two-stage.

In one-stage NAS, the performance of an architecture is calculated by simply instantiating it, training it from scratch, and evaluating the resulting model. In two-stage NAS, before performing the search, a Supermodel (that is larger than any candidate architecture in the search space) is trained. Then during the search process, the performance of architectures is estimated by instantiating them with weights extracted from the Supermodel and either evaluating the resulting models immediately or after a small number of fine-tuning steps (Cai et al., 2020). Because of the reduction in training candidate architectures, two-stage NAS accelerates the search process significantly, especially for large models. AutoTinyBERT (Yin et al., 2021) was able to use two-stage NAS to produce models with reduced inference latency without loss in performance on the GLUE benchmark (Yin et al., 2021) compared to the SOTA search-based method and distillation-based methods (Xu et al., 2021; Sanh et al., 2019; Jiao et al., 2019; Wang et al., 2020; Sun et al., 2020). AutoTinyBERT specifically uses NAS to create models that satisfy certain latency constraints, whilst maximizing model accuracy, as opposed to targeting accuracy directly.

Quantization is another technique aimed at model compression by converting high-precision floating-point parameter values to lower-precision data types, which in turn reduces memory footprint and can speed up model inference if the target hardware supports faster computation on said quantized data types. Specifically, accelerated int8 quantized inference is now supported by server-grade CPUs and GPUs, allowing such models to include more parameters compared to float32 models under the same latency constraint. This opens up a new direction for optimization by trading parameters' precision for larger parameter counts, which inspires this work. Prior works such as I-BERT (Kim et al., 2021) and Q8BERT (Zafrir et al., 2019) have successfully applied quantization to BERT, maintaining high test accuracy while achieving model compression and inference speedup.

Finally, the 2018 JASQ method (Chen et al., 2018) com-

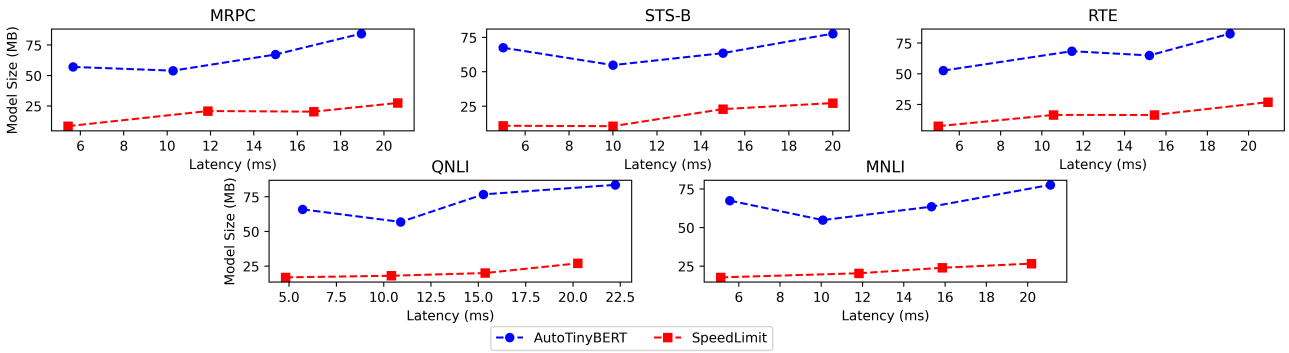

Figure 3. Latency and sizes of models found by *SpeedLimit* and AutoTinyBERT.

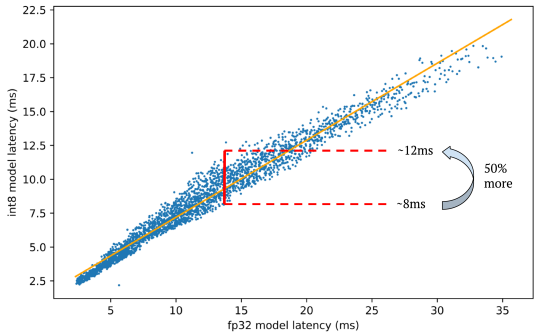

Figure 4. Comparison of various models' int8 version and float32 version latencies. The same float32 and int8 model configuration can result in very different corresponding latency values.

bined quantization with NAS, and helped to motivate our contributions, but is ultimately a very different endeavor. JASQ searches both architectures and quantization policies to produce an optimal CNN with quantization precision potentially mixed across layers. Unlike our method, they target much smaller models and only in the vision and not language domain. Accordingly, they employ one-stage as opposed to two-stage NAS. Additionally, their method targets a certain model size constraint as opposed to latency, hence their use of mixed precision quantization. We solely target int8 quantization as it has more widely supported computational speedup compared to other quantized datatypes.

3. Methodology

We propose *SpeedLimit*, a method to automatically search for an optimal quantized Transformer architecture given a latency constraint. By optimal, we mean maximizing the performance of the model by some metric on a downstream fine-tuning task whilst still hitting an inference latency constraint. We use BERT as an example Transformer, however our techniques can easily be applied to other Transformer based models.

Figure 2 shows the full *SpeedLimit* pipeline. Firstly, given some latency constraint, we use a latency predictor to narrow the search space of all possible architectures to just that of int8 architectures that will meet the inputted latency constraint. We call the resulting search space the narrowed search space. The use of the latency predictor greatly accelerates the search space narrowing process, as it removes the need to profile the performance of all the model architectures present in the original search space. Next, we apply two-stage NAS to the narrowed search space, using iterative rounds of evolutionary search to find the optimal BERT configuration. As we use two-stage NAS, before conducting the search we train a large Supermodel. At each round of the search, candidate models are extracted from this Supermodel, fine-tuned for a small number of epochs, quantized and evaluated. For quantization, we use the PyTorch implementation of dynamic quantization, however this can easily be replaces with other post-training quantization techniques such as GPTQ (Frantar et al., 2022). The best models are evolved and make up the candidates for the following round. After a user inputted number of rounds, the best performing current candidate is returned.

Search space: BERT contains many architecture hyperparameters that can be changed. In this work, we restrict our search space to only contain architectures that have homogeneous encoder blocks (the architecture of each encoder block is identical). Let e be the number of encoder blocks present in the model and d be the number of different individual encoder block architectures. Using homogeneous encoder blocks reduces our search space from polynomial order e, $\mathcal{O}(d^e)$, to simply linear $\mathcal{O}(d)$. This reduction in search space size is essential because even two-stage NAS is very computationally intensive. We consider varying h, the dimensionality of encoder, attention head, and pooler layers, f, the size of the dense feed forward layers, and e. With these hyperparameters, each model architecture we consider can be encoded in a three element tuple (e, h, f). Let $\mathcal{A}$ be the set of all architectures we consider. Naturally

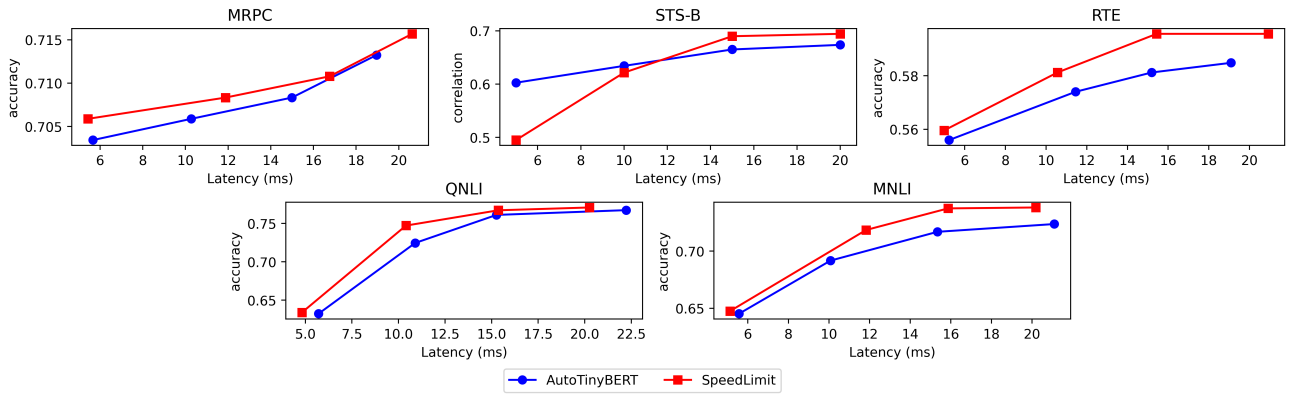

Figure 5. Accuracy and latency values of models found by *SpeedLimit* and AutoTinyBERT.

we must bound the values of e, h and f to ensure that $|\mathcal{A}|$ is small enough to allow searches in reasonable times. Table 1 in the Appendix presents the architectures we used during testing.

Given a user-inputted latency constraint l, we aim to search only the architectures that correspond to int8 quantized models meeting this constraint. We define $\mathcal{A}^*$ to be the set of such models, which we call the narrowed search space. Ideally, we would instantiate every architecture in $\mathcal{A}$, quantize them, find their latencies, and only places architectures in $\mathcal{A}^*$ that meet l. Unfortunately, this is incredibly time-consuming because instantiating and quantizing large transformer models takes a non-trivial amount of time, and $|\mathcal{A}|$ is large. Instead, we take a small subset of $\mathcal{A}$, find the latencies of the quantized models corresponding to these architectures, train a small latency predictor $\mathcal{L}$ on this data, and use this latency predictor to find $\mathcal{A}^*$. Overall, this process takes far less time than checking the latencies of all architectures in $\mathcal{A}$. More formally, after training $\mathcal{L}$, we find the narrowed search space as $\mathcal{A}^* = \{a \in \mathcal{A} : \mathcal{L}(a) < l\}$. For more details on the exact training procedure for $\mathcal{L}$ used in our experiments, see section C of the Appendix.

Supermodel: During evolutionary search, candidate models are selected and instantiated using weights extracted from the Supermodel. For this reason, we need the Supermodel to have an architecture that is larger than any of those contained within $\mathcal{A}$. Let A_s be the architecture of the Supermodel and $\mathcal{E}$, $\mathcal{H}$, $\mathcal{F}$ be the sets of e, h and f values present in $\mathcal{A}$. Thus, we have: $A_s = (\max(\mathcal{E}), \max(\mathcal{H}), \max(\mathcal{F}))$. We instantiate the Supermodel with architecture A_s and use the modified training algorithm proposed in (Yin et al., 2021) (that encourages the model to be more amenable to weight sharing) to train it. See section A of the Appendix for more information on the weight extraction procedure and Supermodel training algorithm.

Conducting evolutionary search: We adapt the evolutionary search process from AutoTinyBERT (our algorithm is outlined in section B of the Appendix). The search takes as input the narrowed search space $\mathcal{A}^*$. For a user inputted number of rounds, the algorithm extracts the current candidate architectures from the Supermodel, fine-tunes them, quantizes them, and evaluates their performance. The best performing architectures are evolved and used as the starting candidates for the next iteration of search (with the candidates in the first round of search being chosen randomly from $\mathcal{A}^*$). Evolving involves either mutating the candidate by randomly perturbing (e, h, f) values (ensuring the result still adheres to the latency constraint) with probability p_m or sampling an entirely new candidate with probability $1 - p_m$. Thus, p_m is a hyperparameter that controls the rate of exploration of the search algorithm.

4. Results and Discussion

Latency results for BERT models: To motivate why searching for int8 quantized models can outperform searching for full precision models, we benchmarked various quantized and non-quantized BERT models whose architectures were drawn from $\mathcal{A}$ as defined in Table 1 of the Appendix. For experiments, we used an Intel Ice Lake Xeon Platinum 8358 CPU @ 2.60GHz. This CPU was chosen intentionally as it comes enabled with the AVX-VNNI extension that allow for accelerated int8 operations. To benchmark the models, we simply drew a set of architectures and instantiated full floating point models of each. We recorded the averaged latency of every model over 4 single sentence inputs, each containing 128 tokens (after one untimed run to allow for model warm-up). We then used PyTorch's dynamic quantization library to quantize the models to int8 and re-ran the average latency test. Figure 4 shows the collected data. Every point represents a certain architecture for a BERT model. We fitted a linear regression to the data to quantify the relationship between int8 and float32 inference

times, finding $latency_{float32} = 1.75 \times latency_{int8} - 2.65$.

The regression analysis illustrates a significant reduction in model latency when using int8 as compared to float32, indicating the potential for employing a more parameter-dense, quantized model within the same latency constraint. Interestingly, Figure 4 shows two distinct models, identical in float32 latency, have a roughly 50% discrepancy in int8 latency. This phenomenon, pervasive across our dataset, underscores the necessity of an independent, quantization-aware search for int8 models. Relying solely on a float32 model search followed by quantization could result in sub-optimal int8 models due to the observed variance in int8 latencies for models with identical float32 latencies.

Optimal Configurations for int8 BERT Models: Using the method described in section 3 we conducted searches for four different latency targets. We used $\mathcal{A}$ and $\mathcal{L}$ as described in section C of the Appendix. For each latency target experiment, we conducted four rounds of evolutionary search. The final search result is shown in Figure 5. We see our method outperforms AutoTinyBERT for all latency targets bar two for the STS-B dataset. In the best case, we see up to a 2.7% accuracy gain on the MNLI dataset. More generally, a better search strategy should give a curve closer to the upper left corner, thus being closer to the Pareto frontier. We see that, across almost all tasks, *SpeedLimit* beats AutoTinyBERT by this qualitative metric. We also note (Yin et al., 2021) reports AutoTinyBERT outperforms standard BERT models with corresponding latencies. We test this specifically using the MNLI dataset in Figure 1. More precisely, we compare the latency and accuracy of models found by *SpeedLimit* against a set of default BERT architectures fine-tuned using full float32 precision (Baseline BERT). We also compare against these baseline models quantized to int8 using PyTorch dynamic quantization (Quantized BERT). As expected (seen as we outperform AutoTinyBERT), we outperform both of these BERT baselines.

With quantization, we allow the model to have more parameters while remaining within the latency range restriction. Although the reduced precision for all the parameters changing from float32 to int8 will reduce the model's performance to some extent, this loss is overpowered by the accuracy gain of additional parameters. We also recorded the size of the models that are presented in Figure 3. We see that in addition to better performance, the memory footprint of models found by *SpeedLimit* are on average $3.4\times$ smaller than those found by AutoTinyBERT. This reduction in size is critical for many applications, especially when deploying models such as BERT on edge devices with low memory resources (Murshed et al., 2021).

5. Conclusion

We present *SpeedLimit*, a novel method that combines two-stage NAS and quantization to find latency constrained BERT models. We are able to outperform the current SOTA method in terms of accuracy, latency, and memory footprint of outputted models. Our method serves as a key step forward in allowing practitioners to deploy performant models in strict latency constrained environments.

References

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. Language models are few-shot learners. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., and Lin, H. (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 1877–1901. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf.

Cai, H., Gan, C., Wang, T., Zhang, Z., and Han, S. Once for all: Train one network and specialize it for efficient deployment. In *International Conference on Learning Representations*, 2020. URL https://arxiv.org/pdf/1908.09791.pdf.

Chen, X., Hsieh, C.-J., and Gong, B. When vision transformers outperform resnets without pretraining or strong data augmentations. *arXiv preprint arXiv:2106.01548*, 2021.

Chen, Y., Meng, G., Zhang, Q., Zhang, X., Song, L., Xiang, S., and Pan, C. Joint neural architecture search and quantization, 2018.

Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., Schuh, P., Shi, K., Tsvyashchenko, S., Maynez, J., Rao, A., Barnes, P., Tay, Y., Shazeer, N. M., Prabhakaran, V., Reif, E., Du, N., Hutchinson, B. C., Pope, R., Bradbury, J., Austin, J., Isard, M., Gur-Ari, G., Yin, P., Duke, T., Levskaya, A., Ghemawat, S., Dev, S., Michalewski, H., García, X., Misra, V., Robinson, K., Fedus, L., Zhou, D., Ippolito, D., Luan, D., Lim, H., Zoph, B., Spiridonov, A., Sepassi, R., Dohan, D., Agrawal, S., Omernick, M., Dai, A. M., Pillai, T. S., Pellat, M., Lewkowycz, A., Moreira, E. O., Child, R., Polozov, O., Lee, K., Zhou, Z., Wang, X., Saeta, B., Diaz,

M., Firat, O., Catasta, M., Wei, J., Meier-Hellstern, K. S., Eck, D., Dean, J., Petrov, S., and Fiedel, N. Palm: Scaling language modeling with pathways. 2022.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding, 2018.

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

Elsken, T., Metzen, J. H., and Hutter, F. Neural architecture search: A survey. *The Journal of Machine Learning Research*, 20(1):1997–2017, 2019.

Frantar, E., Ashkboos, S., Hoefler, T., and Alistarh, D. Gptq: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*, 2022.

Jiao, X., Yin, Y., Shang, L., Jiang, X., Chen, X., Li, L., Wang, F., and Liu, Q. Tinybert: Distilling bert for natural language understanding, 2019.

Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., and Amodei, D. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

Kim, S., Gholami, A., Yao, Z., Mahoney, M. W., and Keutzer, K. I-bert: Integer-only bert quantization. *International Conference on Machine Learning (Accepted)*, 2021.

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., and Stoyanov, V. Roberta: A robustly optimized bert pretraining approach, 2019. URL http://arxiv.org/abs/1907.11692. cite arxiv:1907.11692.

Murshed, M. S., Murphy, C., Hou, D., Khan, N., Ananthanarayanan, G., and Hussain, F. Machine learning at the network edge: A survey. *ACM Computing Surveys (CSUR)*, 54(8):1–37, 2021.

Parmar, N., Vaswani, A., Uszkoreit, J., Kaiser, L., Shazeer, N., Ku, A., and Tran, D. Image transformer. In Dy, J. and Krause, A. (eds.), *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 4055–4064. PMLR, 10–15 Jul 2018. URL https://proceedings.mlr.press/v80/parmar18a.html.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. Language models are unsupervised multitask learners. 2019.

Sanh, V., Debut, L., Chaumond, J., and Wolf, T. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter, 2019.

Smith, S., Patwary, M., Norick, B., LeGresley, P., Rajbhandari, S., Casper, J., Liu, Z., Prabhumoye, S., Zerveas, G., Korthikanti, V., Zhang, E., Child, R., Aminabadi, R. Y., Bernauer, J., Song, X., Shoeybi, M., He, Y., Houston, M., Tiwary, S., and Catanzaro, B. Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model, 2022.

Sun, Z., Yu, H., Song, X., Liu, R., Yang, Y., and Zhou, D. Mobilebert: a compact task-agnostic bert for resource-limited devices, 2020.

Wang, W., Wei, F., Dong, L., Bao, H., Yang, N., and Zhou, M. Minilm: Deep self-attention distillation for task-agnostic compression of pre-trained transformers, 2020.

Xu, J., Tan, X., Luo, R., Song, K., Li, J., Qin, T., and Liu, T.-Y. Nas-bert: Task-agnostic and adaptive-size bert compression with neural architecture search, 2021.

Yin, Y., Chen, C., Shang, L., Jiang, X., Chen, X., and Liu, Q. 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 (Volume 1: Long Papers)*, pp. 5146–5157, Online, August 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.acl-long.400. URL https://aclanthology.org/2021.acl-long.400.

Zafrir, O., Boudoukh, G., Izsak, P., and Wasserblat, M. Q8bert: Quantized 8bit bert, 2019.

Zhu, Y., Kiros, R., Zemel, R., Salakhutdinov, R., Urtasun, R., Torralba, A., and Fidler, S. 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, 2015.

A. Super Model

We use the same Supermodel training and submodel weight extraction technique as (Yin et al., 2021). We summarize the training scheme here. We divide each training input batch into n sub-batches and distribute them onto n threads. Then for m steps, we sample n sub models from the Supermodel and distribute them onto the n threads and calculate the gradient update to be taken. After these m steps have completed, we average all the gradient updates across the n threads and use this average gradient to update the model weights.

For the loss, we use pure knowledge distillation from a BERT-base-uncased teacher model conducting masked language modelling on the BookCorpus (Zhu et al., 2015) dataset. We use the same hyperparameters as (Yin et al., 2021) (peak learning rate of 1e-5, warm-up rate of 0.1, $n = 16$ and $m = 3$) except we use a smaller batch size of 12, a maximum sequence length of 512, and train for 4 epochs.

B. Search Procedure

Algorithm 1 summarizes the evolutionary search algorithm used in *SpeedLimit*.

Algorithm 1 Evolutionary Search

Input: T, the number of generations of evolutionary search, S the number of candidates to consider at generation, p_m the mutation probability, $\mathcal{A}^*$
$\mathcal{G}_1 \leftarrow \mathcal{A}^*$
for $t = 1, 2, \ldots, T$ **do**
  $\mathcal{G}_t \leftarrow \{\}$
  while $|\mathcal{G}_t| < S$ **do**
    $\alpha_{\text{old}} \leftarrow$ a sample (without replacement) from $\mathcal{G}_{t-1}$.
    $\alpha_{\text{quant}} \leftarrow$ an 8-bit quantized version of α_{old}.
    $p \leftarrow$ a uniform random number from 0 to 1.
    if $p < p_m$ **then**
      $\alpha_{\text{new}} \leftarrow$ a mutation of α_{quant}.
    else
      $\alpha_{\text{new}} \leftarrow$ a random sample from $\mathcal{A}$.
    end if
    Append α_{new} to $\mathcal{G}_t$
  end while
end for
$\mathcal{M} \leftarrow$ the set of models with architectures from $\mathcal{G}_T$ and weights from the Supermodel.
$\mathcal{M}_{\text{quant}} \leftarrow \{\}$
for $m \in \mathcal{M}$ **do**
  Append the quantized version of m to $\mathcal{M}_{\text{quant}}$
end for
$\alpha_{\text{opt}} \leftarrow$ the architecture of model with the best accuracy on the target task from $\mathcal{M}_{\text{quant}}$.
return α_{opt}

C. Experimental Details

For our experimental results presented in section 4, we applied constraints on the (e, h, f) values that we included in the search space of architectures $\mathcal{A}$. This was done to ensure that narrowing the search space to $\mathcal{A}^*$, set of architectures that met the inputted latency constraint, could be done in reasonable time. The values we selected to be in $\mathcal{A}$ are shown in table 1.

For the latency predictor $\mathcal{L}$, we used a simple multi layer perceptron with 3 hidden layers and 2000 nodes per layer. We trained using an Adam optimizer with a learning rate of $1e-5$, $\beta_1 = 0.9$, $\beta_2 = 0.99$, $\theta_0 = 1e-08$ and a mean squared error loss. We conducted a grid search to find the optimal learning rate, and used the default values from the PyTorch implementation of the Adam optimizer for β_1, β_2 and θ_0. We trained the latency predictor for 5000 steps, each of which used a batch of 128 random architectures from $\mathcal{A}$ (as defined in table 1) labeled with quantized latencies. To find the latencies for this training data, we instantiated the given architecture, quantized it, and recorded the latency across 5 forward passes

with a single input containing 128 tokens. We excluded the first latency value as they were usually significantly larger than subsequent values due to model warm-up (caching of model parameters), and used the average of the remaining 4 latency values as the latency label. Our implementation of $\mathcal{L}$ was able to achieve a 3.28% mean average percentage error on a held out testing set.

Table 1. Hyperparameter values in search space

Hyperparameter	Value present in $\mathcal{A}$
e	$[1, 2, 3, 4, 5]$
h	$[120, 132, \ldots, 12k, \ldots, 516, 528]$
f	$[128, 140, \ldots, 12k, \ldots, 1004, 1016]$