Efficient Training of Language Models using Few-Shot Learning

Sashank J. Reddi [1] **Sobhan Miryoosefi** [1] **Stefani Karp** [1 2] **Shankar Krishnan** [1] **Satyen Kale** [1] **Seungyeon Kim** [1] **Sanjiv Kumar** [1]

Abstract

Large deep learning models have achieved state-of-the-art performance across various natural language processing (NLP) tasks and demonstrated remarkable few-shot learning performance. However, training them is often challenging and resource-intensive. In this paper, we study an efficient approach to train language models using few-shot learners. We show that, by leveraging the fast learning nature of few-shot learners, one can train language models efficiently in a stagewise manner. Our main insight is that stacking a good few-shot learner on a good small language model provides a good initializer for a larger language model. Using this insight and building upon progressive stacking approaches, we develop novel approaches for training such networks in a stagewise manner. Furthermore, we also provide a theoretical framework and accompanying empirical studies to support our insights, thereby creating a theoretical foundation for progressive stacking. Finally, we provide empirical results to demonstrate the effectiveness of our approach in reducing the training time of few-shot learners.

1. Introduction

Large neural networks have completely revolutionized the fields of computer vision and natural language processing. (e.g. BERT, T5, GPT3) (Devlin et al., 2018; Raffel et al., 2020; Brown et al., 2020). These models have demonstrated remarkable performance across a wide variety of tasks. These models are typically trained with a language modeling objective like masked language modeling (e.g. BERT) or span-corruption (e.g. T5) using adaptive variants of SGD. However, in such large-scale regimes, even training with highly efficient optimizers like SGD (or its adaptive variants) becomes computationally intractable, making these large models challenging and resource-intensive to train. On the other hand, when trained with these objectives, a particularly intriguing property of these models emerges — their ability to adapt to a new (related) task using very few samples from the task (often referred to as few-shot learning in the literature). For instance, the BERT model achieves very competitive performance on challenging question-answering datasets like SQuaD with very few samples (Devlin et al., 2018; Brown et al., 2020). Furthermore, such few-shot learning performance has been demonstrated on a variety of NLP and mathematics reasoning tasks. Thus, improving the training efficiency of these models is of paramount importance to reap these benefits.

Many training speedups achieved thus far came about by introducing new optimizers, including Adam, Adafactor, Shampoo, and LAMB (Kingma & Ba, 2015; Shazeer & Stern, 2018; Anil et al., 2020; You et al., 2020). Recently, an orthogonal approach was proposed to improve the training efficiency of BERT models (Gong et al., 2019). The key idea of this approach is to train the BERT model in stages: begin training with a shallow BERT model and, in each stage, double the model's depth by "stacking" it on top of itself (i.e., copying over the existing model's parameters into newly-created Transformer layers). The final stage involves training the whole BERT model (but for fewer steps than one otherwise would). Surprisingly, this simple algorithm, called *progressive stacking*, improves the training efficiency of BERT.

While training neural networks in parts has been extensively studied in the deep learning literature (e.g. greedy layerwise training of deep belief networks), such stacking approaches have not been studied earlier. Gong et al. (2019) primarily motivates progressive stacking by the similarity of the distributions of the attention scores across layers. One shortcoming of progressive stacking, however, is that the network size grows exponentially. Due to this, often the majority of the training time is concentrated toward the final stage of training, when the network is largest and thus takes the longest to train. Furthermore, we argue that the similarity of the attention scores does not fully explain the efficiency gains due to stacking. Inspired by progressive stacking,

[1]Google Research, NY, USA [2]Carnegie Mellon University, Pittsburgh, PA. Correspondence to: Sashank J. Reddi <sashank@google.com>, Sobhan Miryoosefi <miryoosefi@google.com>.

Proceedings of the 40^{th} International Conference on Machine Learning, Honolulu, Hawaii, USA. PMLR 202, 2023.

some recent works also progressively increase the width of the neural network (Gu et al., 2021; Shen et al., 2022). However, none of these approaches provides any theoretical basis for the approach.

Building upon progressive stacking, this paper studies a generic approach to efficiently train language models. Our key insight is that one can exploit the fast learning nature of few-shot learners to train language models in a stage-wise manner. In particular, we argue that stacking a good few-shot learner on a good small language model can provide a good initializer for the larger model *after just a few iterations of training*. This few-shot learning perspective stands in contrast to recent papers related to stacking that focus on growth operators that *preserve the function value* of the loss after expanding the network. We argue that the ability to train faster while growing the network is a more important aspect than simply preserving the loss function value. Through comprehensive experiments, we show that this property enables fast staged training of large models.

Main Contributions. In light of the above discussion, we highlight the following primary contributions of the paper.

- We develop a principled generic framework for training few-shot learners in a staged manner. The key component of this framework is to stack a good few-shot learner on a good small language model to provide a good initializer for the larger language model.
- We show that standard stacking approaches can be seen as special instantiations of our framework and propose several variants of stacking-based training.
- We provide a theoretical basis for our approach and provide empirical evidence to support our hypothesis. We show that better performance of few-shot learners on distributions used for training language models correlates with better initializers for larger models.
- We conduct comprehensive experiments demonstrating several variations of our approach and show that they outperform the baselines.

Notation. We use $\|\cdot\|_{TV}$ and $D_{\mathrm{KL}}(\cdot\|\cdot)$ to denote the total variation distance and KL divergence, respectively, and $H(\cdot)$ to denote entropy. For a matrix M, we use $[M]_i$ and M_i to denote the i^{th} column and row, respectively. For a stochastic iterative optimizer $\mathcal{A}$, we use $\mathcal{A}(\mathcal{L}, f)$ to denote the resultant prediction function from applying a stochastic update to f with objective function $\mathcal{L}$. With slight abuse of notation, we use $\mathcal{A}(\mathcal{L}, f, t)$ to denote t iterations of the iterative optimizer $\mathcal{A}$. For any sequence $x = (x^1, \cdots, x^m)$, we use x^{-i} to denote the sequence $(x^1, \cdots, x^{i-1}, x^{i+1}, \cdots x^m)$ obtained after removing the i^{th} element of the sequence.

1.1. Related Work

Improving the efficiency of training algorithms has been extensively studied in the statistics, machine learning, and deep learning literature (Robbins & Monro, 1951; McMahan & Streeter, 2010; Duchi et al., 2011; Kingma & Ba, 2015). In particular, SGD and its adaptive variants (e.g. Adagrad, Adam) have become the de-facto algorithms for training large language models like BERT (Devlin et al., 2018; Chowdhery et al., 2022). However, even first-order methods like Adam are inefficient and have high memory requirements in large-scale settings. Adaptive variants like Adafactor and SM3 have been proposed to partially address these issues (Shazeer & Stern, 2018; Anil et al., 2019), but backpropogation can still be quite expensive in these settings, thereby placing limits on the scale of the model. Improving the optimization algorithm itself is orthogonal to our line of study in this paper.

The work most relevant to this paper is that of staged training, which was proposed to improve the training efficiency of language models (Gong et al., 2019; Gu et al., 2021; Shen et al., 2022). These methods start by training a smaller version of the language model and increasing the network size throughout the training process. In particular, Gong et al. (2019) motivate their work by observing the attention matrix patterns across layers of Transformer networks; they exploit this observation by copying network layers to double the size of the network. Gu et al. (2021) and Shen et al. (2022) consider growth operators to increase the size of the network and ensure that the loss function is preserved while growing the network. However, all these methods typically increase network size quickly, thus concentrating training time at larger network sizes and unfortunately limiting the gains. Furthermore, a theoretical framework for such an approach is largely unexplored. Using our framework, we propose several principled variants of stacking which outperform the above methods.

2. Problem Setup

We consider the pretraining phase of a typical language model. In particular, consider the following standard masked language model setup. Consider a vocabulary of words $\mathcal{W}$ and $V = |\mathcal{W}|$ denote the vocabulary size. We assume $\mathcal{W}$ includes a special word denoted by $\varnothing$, which will be useful for defining the masking operation in the training task. For a sequence $x = (x^1, \cdots, x^m)$ where $x^i \in \mathcal{W}$, we use $\mathbb{1}_x \in \{0,1\}^{V \times m}$ to denote the one-hot encoding of x, and with some abuse of notation, $\mathbb{1}_{x^i} \in \mathbb{R}^V$ to denote the one-hot encoding of x^i. We use $\mathbb{P}_{\mathcal{W}}$ to denote the distribution over input sequences of words with sequence length m. Given this distribution, the pretraining task T consists of:

1. **Random masking**: Given a sequence $x = (x^1, \cdots, x^m)$, we randomly choose an index $i \in [m]$

and modify the input to $\tilde{x}$ as follows:

$$\tilde{x}^i = \varnothing \quad \text{and} \quad \tilde{x}^{-i} = x^{-i}.$$

We use $M_i(x)$ to denote the masking operation on input x on index i, i.e. $M_i(x) = \tilde{x}$ defined above. Since $M_i(\cdot)$ only depends on x^{-i} and not on x^i, we overload notation and use $M_i(x^{-i})$ to also denote the same output, i.e. $\tilde{x}$.

2. **Prediction**: Predicting the input sequence given the masked input sequence.

This setup is close to the masked language model setting used in BERT (Devlin et al., 2018). However, our analysis can be generalized for other language modeling objectives (e.g. span corruption in (Raffel et al., 2020)). Let $E \in \mathbb{R}^{V \times d}$ denote fixed embeddings for the vocabulary where i^{th} row of E corresponds to the embedding for word $w_i \in \mathcal{W}$. We assume that $E_i \neq E_j$ for all $i, j \in [V]$. For an injective sequence-to-sequence function $\Phi : \mathcal{W}^m \to \mathbb{R}^{d \times m}$, we define the probability distribution $\mathbb{Q}_\Phi$ over $\mathbb{R}^{d \times m}$ such that $\mathbb{Q}_\Phi(\Phi(x)) = \mathbb{P}_{\mathcal{W}}(x)$ for all $x \in \mathcal{W}^m$. For $v \in \mathbb{R}^{d \times m}$, we define $\Gamma(v)$ to be the $V \times m$ matrix whose columns are obtained from those of Ev by applying the softmax operation to each column of Ev. Interpreting the columns of Ev as giving embeddings for the words in an m-length sequence, $\Gamma(v)$ gives predicted distributions over $\mathcal{W}$ for each position in the m-length sequence represented by v.

Given embeddings E, we define the sequence-to-sequence mapping $\mathbb{1} : \mathcal{W}^m \to \mathbb{R}^{d \times m}$ as $\mathbb{1}(x) = E^\top \mathbb{1}_x$. Since $E_i \neq E_j$ for $i, j \in [V]$ with $i \neq j$, this is an injective function. Hence, this defines a probability distribution $\mathbb{Q}_{\mathbb{1}}$ over $\mathbb{R}^{d \times m}$ as specified above (i.e. $\mathbb{Q}_{\mathbb{1}}(\mathbb{1}(x)) = \mathbb{P}_{\mathcal{W}}(x)$).

Consider a prediction function $f : \mathbb{R}^{d \times m} \to \mathbb{R}^{d \times m}$. Ideally, a good prediction function should be able to predict x^i given x^{-i}. In other words, $[\Gamma(f \circ \mathbb{1}(M_i(x)))]_i$ should be close to $\mathbb{1}_{x^i}$. This is ensured in language models by minimizing the cross-entropy loss as follows, for a given function class $\mathcal{F}$:

$$\min_{f \in \mathcal{F}} L_{\mathbb{Q}_{\mathbb{1}}}(f) := \mathbb{E}_{x \sim \mathbb{P}_{\mathcal{W}}} \mathbb{E}_{i \sim [m]} - \mathbb{1}_{x^i}^\top \log[\Gamma(f \circ \mathbb{1}(M_i(x)))]_i \tag{1}$$

More generally, if the masked input $M_i(x)$ is processed via a sequence-to-sequence function $\Phi : \mathcal{W}^m \to \mathbb{R}^{d \times m}$, the effective input distribution becomes $\mathbb{Q}_\Phi$, and we define the loss as

$$L_{\mathbb{Q}_\Phi}(f) := \mathbb{E}_{x \sim \mathbb{P}_{\mathcal{W}}} \mathbb{E}_{i \sim [m]} - \mathbb{1}_{x^i}^\top \log[\Gamma(f \circ \Phi(M_i(x)))]_i \tag{2}$$

An equivalent formulation of the loss above is given below, which follows immediately from the definition of the loss, and is hence given without proof:

Algorithm 1 Few-Shot Stacking

Input: Target layer size K, per-stage network increment $\{\Delta_p\}_{p=0}^k$ such that $\sum_i \Delta_i = K$, per-stage optimization iterations $\{t_p\}_{p=0}^k$, iterative optimizer $\mathcal{A}$
Initial Phase: Train an initial language model f_0 of size Δ_0 for t_0 iterations with $\mathcal{A}$ on $L_{\mathbb{Q}_{\mathbb{1}}}$
for $p = 1$ to k **do**
    Train a few-shot learner h_p such that $S(h_p) = \Delta_p$.
    $f_p^0 = h_p \circ f_{p-1}$
    for $t = 1$ to t_p **do**
        $f_p^t \leftarrow \mathcal{A}(L_{\mathbb{Q}_{\mathbb{1}}}, f_p^{t-1})$
    end for
    $f_p \leftarrow f_p^{t_p}$
end for
Output: Final prediction function f_k

Lemma 2.1. *The loss $L_{\mathbb{Q}_\Phi}(f)$ equals*

$$\mathop{\mathbb{E}}_{i \sim [m]} \mathop{\mathbb{E}}_{x^{-i} \sim \mathbb{P}_{\mathcal{W}}^{-i}} D_{KL}(\mathbb{P}_{\mathcal{W}}^{|x^{-i}} \,||\, [\Gamma(f \circ \Phi(M_i(x^{-i})))]_i) + L^*,$$

where

$$L^* = \mathbb{E}_{i \sim [m]} \mathbb{E}_{x^{-i} \sim \mathbb{P}_{\mathcal{W}}^{-i}} H(\mathbb{P}_{\mathcal{W}}^{|x^{-i}}). \tag{3}$$

Here, $\mathbb{P}_{\mathcal{W}}^{-i}$ is the marginal distribution on x^{-i}, and $\mathbb{P}_{\mathcal{W}}^{|x^{-i}}$ is the distribution of x^i conditioned on x^{-i}.

For the purpose of this discussion, we mainly focus on a neural network model where $\mathcal{F}_K$ denotes a sequence-to-sequence neural network with K layers (e.g. standard Transformer network). For any neural network f, we use $S(f)$ to denote the size (i.e., number of layers) of the neural network. Note that the aforementioned objective is the population version of the loss that is typically used in practice.

3. Algorithm

We describe our stagewise training approach as follows. Specifically, we consider the setting where the goal is to learn a K-layer sequence-to-sequence neural network in $\mathcal{F}_K$. The training is divided into k stages. We start with training a very small network. At each stage, we increase the size of the network in a gradual manner. The algorithm is presented in Algorithm 1. Our approach has the following key components:

- **Initial few-shot learner.** Train an initial few-shot learner. Usually, this will be a much smaller model compared to the target few-shot learner.
- **Training a smaller few-shot learner.** In the first step, we train a (or reuse existing) small few-shot learner h_p. This few-shot learner can potentially be trained in a recursive manner using Algorithm 1 or could possibly

be obtained from f_{p-1}. We will revisit this in a later section (see Section 3.2).

- **Stacking few-shot learners.** Stack h_p on the language model learned in the previous stage and train the resultant network for several iterations using any standard iterative optimizer.

The main intuition for our approach is that, at the p^{th} stage, if h_p is a reasonably good few-shot learner, then the stacked network $h_p \circ f_{p-1}$ will be a good initializer for the p^{th}-stage network after just a few iterations of training. As we shall see shortly, there are two key ingredients for a good initializer $h_p \circ f_{p-1}$ for the network at stage p:

1. h_p needs to be a reasonably good few-shot learner on distributions close to $\mathbb{P}_{\mathcal{W}}$ (Definition 3.2). This enables fast learning of the stacked component.
2. f_{p-1} must be a good language model with respect to objective (1). This ensures that the input distribution to few-shot learner h_p in the stacked network $h_p \circ f_{p-1}$ (which is $f_{p-1} \circ \mathbb{1}(M_i(x))$ is close to $\mathbb{P}_{\mathcal{W}}$.

By using an effective optimizer (e.g. SGD or Adam) combined with the few-shot learning nature of the network, one can ensure a good initializer for the larger network. We formalize this intuition in the section below.

3.1. Theoretical Analysis

We present our theoretical analysis in this section. We start with formally defining the notions of a "good" few-shot learner, optimizer and better initializer. A few-shot learner can only be expected to perform well on input distributions that are not far from the distribution that was used to train the few-shot learner. This is quantified as follows:

Definition 3.1 (Distribution Distortion). We say an injective function $\Phi : \mathbb{P}_{\mathcal{W}} \to \mathbb{R}^{d \times m}$ induces an ϵ-distribution distortion if

$$\mathop{\mathbb{E}}_{i \sim [m]} \mathop{\mathbb{E}}_{x^{-i} \sim \mathbb{P}_{\mathcal{W}}^{-i}} \left\| [\Gamma(\Phi(M_i(x^{-i})))]_i - \mathbb{P}_{\mathcal{W}}^{|x^{-i}} \right\|_{TV} \leq \epsilon. \quad (4)$$

The definition provides the distance measure between $\mathbb{P}_{\mathcal{W}}$ and the distribution induced by embeddings obtained through Φ. We can now define a good few-shot learner:

Definition 3.2 (Good Few-Shot Learner). A few-shot learner h is (ϵ, δ, c)-good with iterative optimizer $\mathcal{A}$ if for any injective function $\Phi : \mathbb{P}_{\mathcal{W}} \to \mathbb{R}^{d \times m}$ that induces an ϵ-distribution distortion, the following holds: Suppose h is trained for c steps with $\mathcal{A}$, then the resulting function h' is such that $L_{\mathbb{Q}_\Phi}(h') - L_{\mathbb{Q}_\Phi}(h^*) \leq \delta$ where $h^* = \arg\min_{\bar{h} \in \mathcal{F}_{S(h)}} L(\bar{h})$.

Proposition 3.3. *If h is an (ϵ, δ, c)-good few-shot learner, then h is an (ϵ', δ', c)-good few-shot learner for all $\epsilon' < \epsilon$ and $\delta' > \delta$.*

This follows as a simple consequence of Definition 3.2. Thus, larger ϵ and smaller δ imposes a stronger condition on the few-shot learner. We also define the notion of an "effective" optimization algorithm.

Definition 3.4 (Effective Optimizer). Let $f_2' \circ f_1'$ and $f_2'' \circ f_1$ be the predictors obtained after training $f_2 \circ f_1$ for t steps on objective (1) using iterative optimizer $\mathcal{A}$ by training both f_1 & f_2 and just f_2 respectively. We call $\mathcal{A}$ "effective" if for any $t \geq 1$ and function Φ, we have $L_{\mathbb{Q}_\Phi}(f_2' \circ f_1') \leq L_{\mathbb{Q}_\Phi}(f_2'' \circ f_1)$.

This definition highlights that training the whole network rather than part of the network does not hurt the training. Almost all standard deep learning optimizers (e.g. Adam, RMSProp, SGD) exhibit this property in practice. Finally, we need the following definition for optimal initialization.

Definition 3.5 (δ-optimal Initializer). Consider the p^{th} stage of the training process where initializer f_p is of the form $h \circ f_{p-1}$ for some $h \in \mathcal{F}_{\Delta_p}$. Then an initializer f' is called δ-optimal if $L_{\mathbb{Q}_1}(f') \leq L_{\mathbb{Q}_1}(h \circ f_{p-1}) + \delta$ for all $h \in \mathcal{F}_{\Delta_p}$.

In this definition, δ-optimality captures the notion of a good initialization compared to any possible stacked network. We now state the main result of the paper.

Theorem 3.6. *Consider the p^{th} stage (where $p \in [k]$) of Algorithm 1. Suppose the following conditions hold for few-shot learner h_p, iterations t_p, function class $\mathcal{F}_{\Delta_p}$ and optimizer $\mathcal{A}$:*

1. *h_p in Algorithm 1 is a $(\sqrt{2(L_{\mathbb{Q}_1}(f_{p-1}) - L^*)}, \delta, c)$-good few-shot learner (see Definition 3.2 and (3)),*
2. *$\mathcal{A}$ is an effective optimizer (see Definition 3.4),*
3. *The number of iterations in p^{th} stage $t_p > c$ for all $p \in [k]$.*

Then f_p^c, the predictor after training f_p^0 using $\mathcal{A}$ for c steps, is a δ-optimal initializer for the p^{th} stage.

The proof of the theorem appears in Appendix A. The result shows that stacking can provide a good initializer in a few steps of training. It also provides the trade-off between the language model performance of f_{p-1} and the few-shot learning performance of h_p. In particular, if the performance of the language model is weak (i.e., $L_{\mathbb{Q}_1}(f_{p-1}) - L^*$ is large), we may need to use a stronger few-shot learner h_p to obtain a good initialization. On the other hand, suppose the language model performance of f_{p-1} is good, then $L_{\mathbb{Q}_1}(f_{p-1}) - L^*$ is small. In this case, one can use a weak few-shot learner h_p to ensure a good intializer.

3.2. Progressive Stacking as an Instantiation of Few-Shot Stacking

It is often observed that training sequence-to-sequence models on objective (1) also yields strong few-shot learners. GPT, BERT and T5 are prime examples of this phenomenon. One natural instantiation of Algorithm 1 leveraging this few-shot learning capability of the language models is in the p^{th} stage, where f_{p-1} (or some part it) can be used as h_p, thus using the same network as a few-shot learner and language model. This yields a very simple procedure to incrementally learn a language model: in each stage, the non-embedding layers are copied and stacked on top of the existing model. This approach reduces to *progressive stacking*, which works reasonably well in practice. However, a formal justification for this approach is fairly limited. Our approach provides a theoretical grounding for progressive stacking. In particular, we obtain the following corollary.

Corollary 3.7. *Suppose f_{p-1} is a $(\sqrt{2(L_{\mathbb{Q}_1}(f_{p-1}) - L^*)}, \delta, c)$-good few-shot learner and $\mathcal{A}$ is an effective optimizer. Then progressive stacking after c steps provides a δ-optimal initializer for the p^{th} stage.*

Thus, if the language model is also a good few shot learner, then progressive stacking provides an efficient way to learn the network in a stagewise manner. Note that progressive stacking doubles the size of the network at each stage, which is usually not desirable. Below, we discuss alternate stacking approaches to increase the network size in a gradual manner.

3.3. Gradual Stacking

An alternative stacking approach is to use only part of the network (instead of the whole network) as a few-shot learner. As noted above, standard stacking approaches increase the network size exponentially. By stacking only part of the network, one increases the network in a linear or sub-linear manner. We study two different ways of stacking:

- **Pre-stack**: Select the bottom layers (layers closer to the input) and copy them. For instance, if the earlier network is $f_2 \circ f_1$, then the new network is $f_2 \circ f_1 \circ f_1$.
- **Post-stack**: An alternate approach is to post-stack, where the top layers (layers closer to the output) are replicated. In particular, the network $f_2 \circ f_1$ is transformed to the new network $f_2 \circ f_2 \circ f_1$.

If f_2 and f_1 are identity layers, then pre-stack and post-stack respectively reduce to progressive stacking. The primary assumption in pre-stack and post-stack approaches is that the bottom and top layers are reasonably good language models and few-shot learners, respectively.

Algorithm 2 Independent LM & Few-shot learner

Input: Target size K, per-stage increment Δ, per-stage iterations $\{t_p\}_{p=0}^{k}$, optimizer $\mathcal{A}$, few-shot iterations t,
Initial Phase: Train an initial LM f_0 of size Δ_0 for t_0 with $\mathcal{A}$ on $L_{\mathbb{Q}_1}$. Initialize few-shot learners $\{h_p^0\}_{p=1}^{k}$ of size Δ
for $p = 1$ to k **do**
    $h_p \leftarrow \mathcal{A}(L_{\mathbb{Q}_1}, h_p^0, t)$
    $f_p^0 = h_p \circ f_{p-1}$
    $f_p \leftarrow \mathcal{A}(L_{\mathbb{Q}_1}, f_p^0, t_p)$
end for
Output: Final prediction function f_k

3.4. Single-shot Stacking

In the previous section, we examined an approach that increases the network in a gradual manner. An alternate approach is to train a small network and stack it multiple times until we reach the desired model size, which we refer to as "single-shot stacking". The key intuition behind this approach is to generalize the few-shot stacking framework in Algorithm 1 by stacking multiple few-shot learners. In particular, suppose f_0 is the initial few-shot learner trained. Then single shot stacking involves initializing the network to $f_0 \circ \cdots \circ f_0$ (r times) as the initializer for the larger network. This larger network is then trained for a few iterations to obtain the final network. This is analogous to the fine-tuning step usually used in downstream tasks.

3.5. Independent Stacking

Recall that in stacking, we use the smaller network as both the language model and few-shot learner. One advantage of our framework is that we could use arbitrary pretrained sequence-to-sequence language models and few-shot learners for training. This allows us to independently train multiple models and use one as a language model and the other for few-shot learning purposes. The pseudocode for this particular variant is listed in Algorithm 2.

4. Experiments

Experimental Setup For our experiments, we use original BERT models following Devlin et al. (2019) with batch size and sequence length of 256 and 512, respectively. More specifically, BERT-BASE and BERT-LARGE will constitute our baselines. For all the experiments with BERT-BASE and BERT-LARGE (baselines, compared methods, and our method) the total number of training steps is set to 675K and 1M steps, respectively. The goal of our experiments is twofold: (i) compare the performance of our approach with baselines using different step allocation strategies (Section 4.3) and (ii) provide empirical evidence for few-shot

	Speedup	Pretrain Loss
BERT-BASE		
Baseline	1x	1.739
Progressive Stacking	1.29x	1.737
Gradual Stacking	**1.50x**	**1.689**
BERT-LARGE		
Baseline	1x	1.443
Progressive Stacking	1.24x	1.403
Gradual Stacking	**1.64x**	**1.384**

Table 1. Performance and speedup of different stacking methods. The second and third columns denote the speedup over baseline and pretraining loss respectively. The results indicate gradual stacking outperforms the baseline and progressive stacking in both speedup and quality. Training curves are presented in Appendix B.3.

learning in various stacking approaches (Section 4.6).

Our primary baseline is progressive stacking (Gong et al., 2019). As discussed in Section 3.2, we apply progressive stacking to BERT-BASE and BERT-LARGE. Starting with a 3-layer network, we double the size of the network by stacking the network on itself in stages and continue training. In the final stage, we have a full 12-layer model for BERT-BASE or 24-layer model for BERT-LARGE. The number of steps that are allocated to each stage will have an effect on the performance. The proposed strategy by the authors is to allocate $2/3$ of the total steps to the last stage and divide $1/3$ of the total steps equally between remaining stages. For a fair comparison, we include this allocation along with all strategies mentioned in Section 4.3.1 and report the best performance among them (see Appendix B.2).

For gradual stacking (Section 3.3), we only use a small part of the network as the few-shot learner for stacking. In particular, we fix the *stack size* (number of layers stacked at each stage) and in each stage we copy and stack those many layers either from the top on top of the network (post-stack) or from the bottom to the bottom of the network (pre-stack). To provide a comprehensive empirical analysis, we study different choices of the following parameters: steps allocation per stage (Section 4.3.1), stack size (Section 4.3.2) and post-stack vs. pre-stack (Section 4.3.3). For BERT experiments, our best recipe starts with a 3-layer network and stack size of 3 for BERT-BASE and starts with a 4-layer network and stack size of 4 for BERT-LARGE. In addition, our best recipe uses the post-stack approach and divides the total number of steps equally between stages.

Results. We start with showing the best performing results for progressive and gradual stacking. This comparison is shown in Table 1. Gradual stacking achieves speedups of 1.50x and 1.64x in BERT-BASE and BERT-LARGE, respectively[1]. We observe that gradual stacking outperforms progressive stacking both in terms of speedup and quality.

[1]Speedup is defined as $\text{speed(experiment)}/\text{speed(baseline)}$ where speed is defined as total number of steps divided by total wall-clock time of the experiment

# Layers × Steps	Speedup	Pretrain Loss
BERT-BASE		
Baseline		
$12 \times 675K$	1x	1.739
Single-shot Stacking		
$3 \times 200K$	1.24x	1.85
$3 \times 400K$	1.66x	1.849
$4 \times 200K$	1.21x	1.798
$4 \times 400K$	1.52x	1.813
BERT-LARGE		
Baseline		
$24 \times 1M$	1x	1.443
Single-shot Stacking		
4 × 200K	1.19x	7.917
4 × 400K	1.46x	7.363
4 × 600K	1.90x	1.609
6 × 200K	1.16x	1.46
6 × 400K	**1.39x**	**1.402**
6 × 600K	1.74x	1.487
8 × 200K	1.15x	1.483
8 × 400K	**1.34x**	**1.389**
8 × 600K	1.62x	1.486

Table 2. Performance and speedup of different single-shot stacking parameters applied to BERT-BASE and BERT-LARGE. In the first column, *# Layers* refers to the number of layers in the base network and *Steps* refers to the number of steps used to train the base network. Despite outperforming the baseline in some scenarios, it appears that single-shot stacking is sub-optimal compared to gradual stacking method in terms of both speed and quality.

4.1. Single-shot Stacking

We also provide empirical results for single-shot stacking (Section 3.4). Recall that in single-shot stacking, a small part of the network (*base network*) is first trained (first stage) and then stacked using as many copies as needed to create the full network. This network is trained end-to-end for a few steps to generate the model (second stage). The performance of this approach crucially depends on the choice of the base network size and the number of steps used to train it in the first stage. We perform a sweep of these parameters on BERT-BASE and BERT-LARGE. For BERT-BASE, our hyperparameter sweep is $\{3, 4\} \times \{200K, 400K\}$ followed by training the full network for $475K$ and $275K$ steps, respectively. For BERT-LARGE, our hyperparameter sweep is $\{4, 6, 8\} \times \{200K, 400K, 600K\}$ followed by training the full network for $800K$, $600K$ and $400K$ steps, respectively.

Results. The results of the single-shot stacking experiments are summarized in Table 2. Overall, it appears that single-shot stacking produces slightly sub-optimal models compared to gradual stacking. It is also worth remarking that when the size of the base network is small (in the case of BERT-LARGE, 4 or smaller), it causes the training in the 2^{nd} stage (of the full network) to diverge.

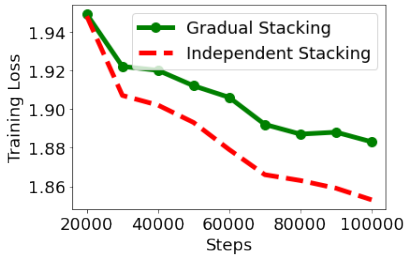

Figure 1. Independent Stacking (Algorithm 2) vs. Gradual Stacking on BERT-BASE in the final stage of two-stage training. We see that independent stacking outperforms gradual stacking.

Strategy	Fraction of total steps in Stage i
Equal	$1/k$
Proportional	$i/(\sum_{j=1}^{k} j)$
Inv. proportional	$(\frac{1}{i})/(\sum_{j=1}^{k} \frac{1}{j})$

Table 3. Allocation strategies: Steps per-stage for k stages.

4.2. Independent Stacking vs. Gradual Stacking

In the previous sections, we observed that gradual stacking is consistently better. Based upon independent stacking in Algorithm 2, we present results when we train BERT-BASE in two stages. For gradual stacking, we train 6 layers in the first stage and stack it to obtain an initializer for 12 layers for just 100K steps. For independent stacking, we train two 6 layer networks independently and stack them (Algorithm 1), and train for 100K steps. The results for this are presented in Figure 1. Remarkably, we see that independent stacking improves over gradual stacking, reminiscent of ensembling but in the stacking framework. Exploring this further constitutes an important future direction.

4.3. Ablation Studies & Analysis

In this section, we comprehensively study the effect of the steps allocation strategy (Section 4.3.1), stack size (Section 4.3.2), and post-stack versus pre-stack (Section 4.3.3) on the performance of gradual stacking.

4.3.1. Steps Allocation

In this section, we study the effect of the steps allocation strategy (i.e., number of steps we allocate to each stage in the stacking approach). For all experiments in this section we use the post-stack approach with a stack size of 3 and 4 for BERT-BASE and BERT-LARGE, respectively. We evaluate the performance of gradual stacking using the allocation strategies in Table 3. The results of this ablation are shown in Table 4. It is obvious that strategies that allocate more steps to the initial stages result in improved speedup. One would expect that strategies that allocate more steps to the later stages would have better quality (e.g. pretraining loss); however, this is not necessarily true (e.g. gradual stacking for Equal vs. Proportional in BERT-LARGE). Our results demonstrate that allocating too many steps to the initial stages or later stages could have a negative impact on the quality and, overall, equal allocation works well. Also note that, if we aim to just match the baseline quality, one could use other allocation strategies with improved speedup. For example, for BERT-LARGE, using the inverse proportional allocation strategy results in 2.11x speedup over the baseline while still outperforming it in terms of quality.

Allocation	Speedup	Pretrain Loss
BERT-BASE		
Baseline		
-	1x	1.739
Gradual Stacking		
Equal	**1.50x**	**1.689**
Proportional	1.29x	1.671
Inv. Proportional	1.77x	1.801
BERT-LARGE		
Baseline		
-	1x	1.443
Gradual Stacking		
Equal	**1.64x**	**1.384**
Proportional	1.35x	1.416
Inv. Proportional	**2.11x**	**1.435**

Table 4. The performance of gradual stacking under different steps allocation strategy. The results suggest that equal allocation works the best amongst all allocations strategies defined in Table 3.

Stack Size	Speedup	Pretrain Loss
BERT-BASE		
Baseline		
-	1x	1.739
Gradual Stacking		
1	1.71x	1.873
2	1.61x	1.803
3	**1.50x**	**1.689**
4	1.43x	1.746
6	1.29x	1.775
BERT-LARGE		
Baseline		
-	1x	1.443
Gradual Stacking		
1	1.81x	1.615
2	1.75x	1.421
4	**1.64x**	**1.384**
6	1.53x	1.385
12	1.30x	1.438

Table 5. The performance of gradual stacking using different choices of stacked network size. We observe that using a stack size which is neither too small nor too large works the best.

4.3.2. Stack Size

In this section, we evaluate the performance of the gradual stacking method based on different choices of stack size. For all these experiments, we use the post-stack approach and equal allocation strategy. The results have been shown in Table 5. Unsurprisingly, smaller stack sizes yield improved speedup. Also, our results indicate that there is a quality-speedup tradeoff with respect to the stack size. Overall, using a stack size which is neither too small nor

{Post,Pre}-Stack	Speedup	Pretrain Loss
BERT-BASE		
Baseline		
-	1x	1.739
Gradual Stacking		
Post-Stack	**1.50x**	**1.689**
Pre-Stack	1.50x	1.731
BERT-LARGE		
Baseline		
-	1x	1.443
Gradual Stacking		
Post-Stack	**1.64x**	**1.384**
Pre-Stack	1.64x	1.415

Table 6. The performance of post-stack vs. pre-stack for gradual stacking. We observe post-stack typically works better.

	Speedup	Pretrain Loss
BERT-LARGE		
Baseline	1x	1.443
Gradual Stacking	1.64x	1.384
Gradual Stacking + Varying sequence length	**2.44x**	**1.429**

Table 7. Performance and speedup of varying sequence length in stagewise pretraining. Results suggest that we could achieve additional speedup; although with slightly worse quality.

large works best. It is worth noting that most of the stack size choices either outperform or are competitive with the baseline, showing the robustness of the method to stack size.

4.3.3. POST-STACK VS PRE-STACK

In this section we compare the performance of *post-stack* versus *pre-stack* for gradual stacking. For all these experiments, stack size 3 and 4 has been chosen for BERT-BASE and BERT-LARGE, respectively. In addition, the steps allocation strategy is to equally divide the total number of steps between stages. The results have been shown in Table 6. Since both approaches achieve the same speedup, the results suggest that post-stack typically works better.

4.4. Varying Sequence Length

An additional axis where we can generate additional performance gains is by varying the sequence length in the masked LM task. The main idea is to train the earlier stages in stacking with smaller sequence length (shorter contexts) which is significantly faster than using the final sequence lengths. Of course, varying the sequence length is not tied to gradual stacking and can be used in conjunction with any other strategy to speed up BERT pre-training. In this study, we compared our best performing gradual stacking setup for BERT-LARGE, which uses a 4-layer network and a stack size of 4. For the 1^{st} two stages, we use a sequence length of 128, followed by two stages with sequence length of 256, and in the final two stages, we use the desired sequence length of 512. Table 7 summarizes the performance and speedup provided by varying sequence length along with our gradual stacking algorithm. We can see that this modification gets a significant speedup over gradual stacking, even though the final loss is slightly higher.

	Pretrain Loss	MNLI	SQuAD2.0
BERT-BASE			
Baseline	1.739	0.8407	**0.8962**
Gradual Stacking	**1.689**	**0.8463**	0.8952
BERT-LARGE			
Baseline	1.443	0.8637	0.921
Gradual Stacking	**1.384**	**0.8721**	**0.9247**

Table 8. Performance of gradual stacking on the downstream tasks MNLI and SQuAD2.0. The results indicates that gradual stacking, while being significantly faster in the pretraining, achieves better or similar downstream performance in BERT-BASE and BERT-LARGE.

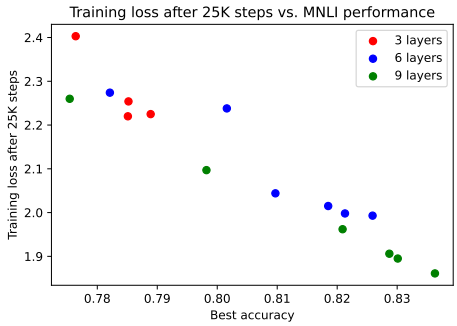

Figure 2. Stacking quality (pretraining loss) at 25K iterations after stacking vs. MNLI performance of the stacked network. The results demonstrate a strong correlation between few-shot learning quality and stacking quality, beyond just the correlation between model size and stacking quality.

4.5. Downstream Performance of Gradual Stacking

In this section, we demonstrate that the improvement in the pretraining loss that we have achieved by gradual stacking translates into an improved downstream performance. We evaluated the performance of gradual stacking on MNLI (Williams et al., 2018) and SQuAD2.0 (Rajpurkar et al., 2018) and the results have been shown in Table 8. The gradual stacking is either competitive or better than the baseline on the downstream tasks while significantly improving the pretraining time (1.50x and 1.64x speedup in BERT-BASE and BERT-LARGE, respectively).

4.6. Effect of Few-Shot Learner Quality

Finally, we empirically explore the correlation between the quality of few-shot learner (measured by the downstream fine-tuning tasks MNLI (Williams et al., 2018) and SQuAD2.0 (Rajpurkar et al., 2018) and *stacking quality* (measured by the pretraining loss after a few iterations of optimization post stacking). Our theory suggests that better

few-shot learning quality enables better stacking quality, which we verify here. To this end, we generate several BERT models of *different* sizes and few-shot learning quality. In particular, we generate models of different sizes (3, 6, and 9 layers) by training them for different numbers of steps (from 150K to 675K), which induces a reasonable range of few-shot learning quality. To generate the final network, the Transformer layers of the above networks are stacked on top of a fixed (partially-trained) 3-layer BERT model. After stacking, the final model is then trained for 100K iterations, and we examine its training loss at 10K, 25K, and 100K iterations. Figure 2 here and Figure 5 in the Appendix display the correlation between (a) the downstream fine-tuning performance on MNLI and SQuAD2.0 of the network that was stacked (*few-shot learning quality*) and (b) the pretraining loss at 25K iterations after stacking (*stacking quality*).

Our experiments show a strong correlation between few-shot learning and stacking quality. In particular, we observe that better few-shot learning quality usually yields better stacking quality, even beyond the correlation between model size and stacking quality. For instance, as seen in Figure 2, there are several instances where *larger* models trained for fewer steps have *worse* few-shot learning quality than *smaller* models trained for more steps, and, in these cases, the larger models also display *worse* stacking quality (see Appendix B for more details). Overall, these experimental results provide strong support for our hypothesis that few-shot learning ability is a key factor in the success of stacking-based approaches.

5. Conclusion & Future Work

In this work, we proposed a general framework to effectively train language models via few-shot learning. Based on our framework, we examined several variants of stacking and developed a theoretical foundation for stacking. Our experiments demonstrate the strong performance of our approach over the baselines. Finally, inspired by the encouraging results with independent stacking, we plan to investigate it further as future work.

References

Anil, R., Gupta, V., Koren, T., and Singer, Y. Memory efficient adaptive optimization. In Wallach, H. M., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E. B., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pp. 9746–9755, 2019.

Anil, R., Gupta, V., Koren, T., Regan, K., and Singer, Y. Second order optimization made practical. *CoRR*, abs/2002.09018, 2020. URL https://arxiv.org/abs/2002.09018.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., 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. M., 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. *CoRR*, abs/2005.14165, 2020. URL https://arxiv.org/abs/2005.14165.

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., Prabhakaran, V., Reif, E., Du, N., Hutchinson, B., Pope, R., Bradbury, J., Austin, J., Isard, M., Gur-Ari, G., Yin, P., Duke, T., Levskaya, A., Ghemawat, S., Dev, S., Michalewski, H., Garcia, 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., Child, R., Polozov, O., Lee, K., Zhou, Z., Wang, X., Saeta, B., Diaz, M., Firat, O., Catasta, M., Wei, J., Meier-Hellstern, K., Eck, D., Dean, J., Petrov, S., and Fiedel, N. Palm: Scaling language modeling with pathways. *CoRR*, abs/2204.02311, 2022. doi: 10.48550/arXiv.2204.02311. URL https://doi.org/10.48550/arXiv.2204.02311.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Devlin, J., Chang, M., Lee, K., and Toutanova, K. BERT: pre-training of deep bidirectional transformers for language understanding. In Burstein, J., Doran, C., and Solorio, T. (eds.), *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pp. 4171–4186. Association for Computational Linguistics, 2019.

Duchi, J., Hazan, E., and Singer, Y. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

Gong, L., He, D., Li, Z., Qin, T., Wang, L., and Liu, T. Efficient training of BERT by progressively stacking. In

Chaudhuri, K. and Salakhutdinov, R. (eds.), *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pp. 2337–2346. PMLR, 09–15 Jun 2019. URL https://proceedings.mlr.press/v97/gong19a.html.

Gu, X., Liu, L., Yu, H., Li, J., Chen, C., and Han, J. On the transformer growth for progressive BERT training. In Toutanova, K., Rumshisky, A., Zettlemoyer, L., Hakkani-Tür, D., Beltagy, I., Bethard, S., Cotterell, R., Chakraborty, T., and Zhou, Y. (eds.), *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2021, Online, June 6-11, 2021*, pp. 5174–5180. Association for Computational Linguistics, 2021.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

McMahan, H. B. and Streeter, M. Adaptive bound optimization for online convex optimization. In *COLT*, 2010.

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

Rajpurkar, P., Jia, R., and Liang, P. Know what you don't know: Unanswerable questions for SQuAD. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pp. 784–789, Melbourne, Australia, July 2018. Association for Computational Linguistics. doi: 10.18653/v1/P18-2124.

Robbins, H. and Monro, S. A stochastic approximation method. *Annals of Mathematical Statistics*, 22:400–407, 1951.

Shazeer, N. and Stern, M. Adafactor: Adaptive learning rates with sublinear memory cost. In Dy, J. G. and Krause, A. (eds.), *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pp. 4603–4611. PMLR, 2018.

Shen, S., Walsh, P., Keutzer, K., Dodge, J., Peters, M., and Beltagy, I. Staged training for transformer language models. In Chaudhuri, K., Jegelka, S., Song, L., Szepesvari, C., Niu, G., and Sabato, S. (eds.), *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pp. 19893–19908. PMLR, 17–23 Jul 2022.

Williams, A., Nangia, N., and Bowman, S. A broad-coverage challenge corpus for sentence understanding through inference. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pp. 1112–1122, New Orleans, Louisiana, June 2018. Association for Computational Linguistics. doi: 10.18653/v1/N18-1101.

You, Y., Li, J., Reddi, S. J., Hseu, J., Kumar, S., Bhojanapalli, S., Song, X., Demmel, J., Keutzer, K., and Hsieh, C. Large batch optimization for deep learning: Training BERT in 76 minutes. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020.

Appendix

A. Proof of Theorem 3.6

Proof. Consider the case where $h_p \circ f_{p-1}$ is trained for c steps with $\mathcal{A}$ and f_{p-1} fixed. Let the resultant predictor function be $h'_p \circ f_{p-1}$. We first observe that

$$L_{\mathbb{Q}_1}(f_p^c) \leq L_{\mathbb{Q}_1}(h'_p \circ f_{p-1}). \tag{5}$$

This follows from effective optimizer property of $\mathcal{A}$. In order to show δ-optimality of f'_p, it is enough to show that

$$L_{\mathbb{Q}_1}(h'_p \circ f_{p-1}) \leq L_{\mathbb{Q}_1}(h \circ f_{p-1}) + \delta, \tag{6}$$

for all $h \in \mathcal{F}_{\Delta_p}$. Combining this with the Equation (5) gives us the required result. To prove the above result, we use the few-shot learning capability of h_p. Consider the function $\Phi(x) = f_{p-1}(\mathbb{1}(x))$. We first observe that

$$L_{\mathbb{Q}_\Phi}(h) = L_{\mathbb{Q}_1}(h \circ f_{p-1}).$$

for any h. This simply follows from the definition of $L_{\mathbb{Q}_\Phi}$ in Equation (2). Thus, showing the inequality in Equation (6) is equivalent to

$$L_{\mathbb{Q}_\Phi}(h'_p) \leq L_{\mathbb{Q}_\Phi}(h) + \delta, \tag{7}$$

for all $h \in \mathcal{F}_{\Delta_p}$. To this end, we first show that Φ satisfies the condition in Equation (4). Let $I : \mathbb{R}^{d\times m} \to \mathbb{R}^{d\times m}$ be the identity mapping. We observe the following:

$$\begin{aligned}
&\mathbb{E}_{i\sim[m]}\mathbb{E}_{x^{-i}\sim\mathbb{P}_{\mathcal{W}}^{-i}} \left\| [\Gamma(\Phi(M_i(x)))]_i - \mathbb{P}_{\mathcal{W}}^{|x^{-i}} \right\|_{TV} \\
&\leq \mathbb{E}_{i\sim[m]}\mathbb{E}_{x^{-i}\sim\mathbb{P}_{\mathcal{W}}^{-i}} \sqrt{2 D_{\mathrm{KL}}(\mathbb{P}_{\mathcal{W}}^{|x^{-i}} \,||\, \Gamma(\Phi(M_i(x)))]_i)} \\
&\leq \sqrt{2\mathbb{E}_{i\sim[m]}\mathbb{E}_{x^{-i}\sim\mathbb{P}_{\mathcal{W}}^{-i}} D_{\mathrm{KL}}(\mathbb{P}_{\mathcal{W}}^{|x^{-i}} \,||\, \Gamma(\Phi(M_i(x)))]_i)} \\
&= \sqrt{2(L_{\mathbb{Q}_\Phi}(I) - L^*)} \\
&= \sqrt{2(L_{\mathbb{Q}_1}(f_{p-1}) - L^*)}
\end{aligned}$$

The first inequality is due to Pinsker's inequality. The second follows from Jensen's inequality. The first equality is due to Lemma 2.1. Since h_p is a $(\sqrt{2(L_{\mathbb{Q}_1}(f_{p-1}) - L^*)}, \delta, c)$-good few-shot learner and since Φ satisfies Equation (4) with $\epsilon = \sqrt{2(L_{\mathbb{Q}_1}(f_{p-1}) - L^*)}$, then we have

$$L_{\mathbb{Q}_\Phi}(h'_p) \leq L_{\mathbb{Q}_\Phi}(h) + \delta.$$

This provides the desired result. □

B. Additional Experimental Results

We use the standard BERT pretraining setup; In particular, BERT is trained on the BooksCorpus (800M words) and Wikipedia (2,500M words). We use the same dataset for the experiments.

Steps Allocation	Speedup	Pretrain Loss
BERT-BASE		
Baseline		
-	1x	1.739
Progressive Stacking		
Original	1.23x	1.81
Equal	1.59x	1.777
Proportional	1.29x	1.737
Inv. Proportional	2.02x	1.853
BERT-LARGE		
Baseline		
-	1x	1.443
Progressive Stacking		
Original	1.24x	1.403
Equal	1.77x	1.486
Proportional	1.28x	1.673
Inv. Proportional	2.72x	1.65

Table 9. The performance of progressive stacking under different steps allocation strategies.

B.1. Shared Hyperparameter Settings

Unless explicitly stated otherwise, all BERT-BASE and BERT-LARGE experiments used the following hyperparameter settings. Each stage began with 10,000 linear warmup steps (from a learning rate of 0 to a learning rate of 0.0001). After warmup, the learning rate was held constant throughout the stage, for all stages other than the final stage. In the final stage, after warmup, the learning rate was linearly decayed to 0. AdamW was used as the optimizer, with $\beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-7}$, and 0 weight decay.

B.2. Steps Allocation Strategy for Progressive Stacking

In Table 9, we can see the performance of progressive stacking under different steps allocation strategies. The *Original* refers to authors' proposed strategy of allocating $2/3$ of the total steps to the last stage and dividing $1/3$ of the total steps equally between the remaining stages.

B.3. Training Curves for Table 1

In Figure 3 and Figure 4, we present the training curves for BERT-BASE and BERT-LARGE, respectively, corresponding to results presented in Table 1. The figures clearly show that our approach outperforms the baseline and the compared method (progressive stacking) in terms of both speed and quality.

B.4. Effect of Few-Shot Learner Quality

Here, we present the experimental settings used in Section 4.6 (including optimizer settings for both pre-training and fine-tuning). We also present Figure 5, which repeats

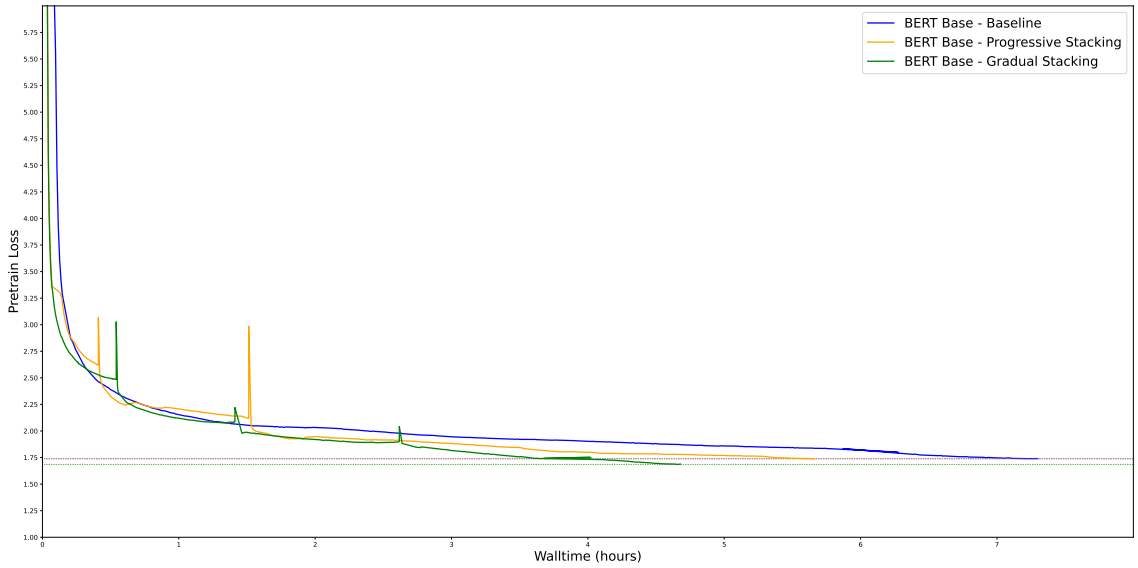

Figure 3. Training curves for gradual stacking (our method), progressive stacking (compared method), and baseline for BERT-BASE. The x-axis corresponds to wall-clock time and y-axis shows the pretrain loss. The dashed lines represent the final pretrain loss value for the different experiments.

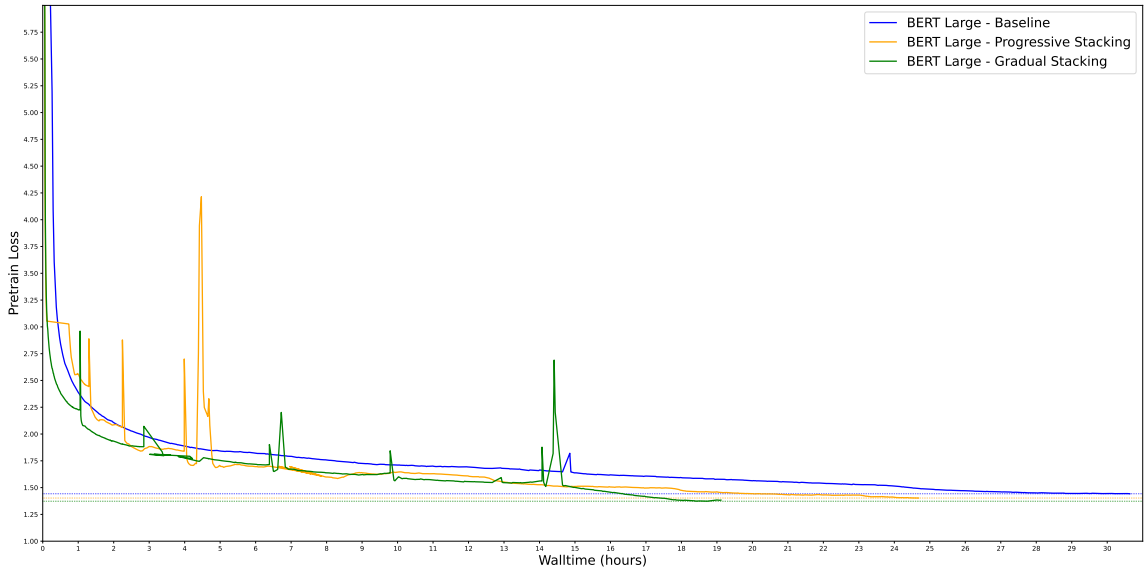

Figure 4. Training curves for gradual stacking (our method), progressive stacking (compared method), and baseline for BERT-LARGE. The x-axis corresponds to wall-clock time and y-axis shows the pretrain loss. The dashed lines represent the final pretrain loss value for the different experiments.

Figure 5. Stacking quality at 25K iterations vs. SQuAD 2.0 performance.

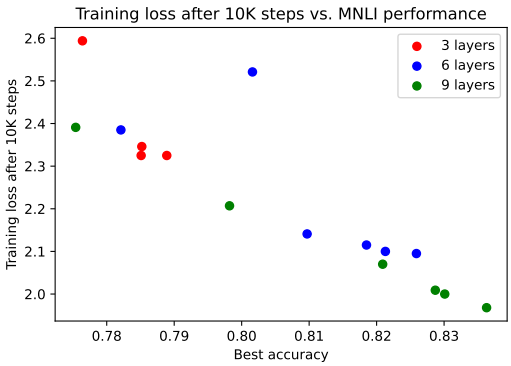

Figure 6. Stacking quality at 10K iterations vs. MNLI performance.

Figure 2 with SQuAD2.0 instead of MultiNLI, as well as Figures 6, 7, 8, and 9, which repeat Figures 2 and 5 with 10K and 100K stacking steps (instead of 25K).

The 3-layer subnetwork was generated according to Section B.1 (including learning rate decay, since it was only one stage). This single stage lasted 675K steps, and models at 150K, 300K, 450K, and 675K steps were saved to be examined for their fine-tuning and stacking quality. The 6-layer model was trained by stacking two 3-layer 150K-step models and training them together for 675K steps, following Section B.1 (including learning rate decay, since it was only one stage); models at 150K, 300K, 450K, and 675K steps were saved to be examined for their fine-tuning and stacking quality. The 9-layer model was trained by stacking three 3-layer 150K-step models and training them together for 675K steps, following Section B.1 (including learning rate decay, since it was only one stage); models at 150K, 300K, 450K, and 675K steps were saved to be examined for their fine-tuning and stacking quality.

Each to-be-stacked subnetwork was stacked upon the 3-layer 150K-step model. The whole model was jointly trained according to Section B.1, but with a warmup period of 50K steps instead of 10K steps and no learning rate decay.

Deviating slightly from Section B.1, fine-tuning on MNLI involved 3681 warmup steps to a learning rate of 3×10^{-5}, followed by learning rate decay to 0 (36813 steps total). Fine-tuning on SQuAD 2.0 involved 544 warmup steps to a learning rate of 8×10^{-5}, followed by learning rate decay to 0 (5440 steps total).

B.5. Analysis of Gradients and Parameter Movement

One of our observations is that there is more than just "similarity to the previous subnetwork in parameter space" driving the success of stacking. Here, we present some empirical results comparing gradual stacking to incremental network

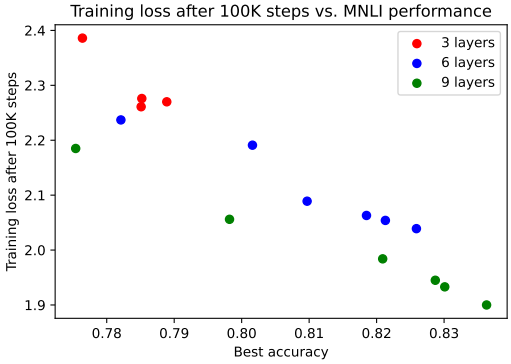

Figure 7. Stacking quality at 100K iterations vs. MNLI performance.

growth with *random* initialization of the new layers (we will use *stacking* and *random* to denote these two procedures, respectively). Figure 10 compares the loss curves of the two approaches when training BERT-BASE using a stack size of 3 and 150K steps per stage, with the exception of the last stage (225K steps). We note that much of the final gap between the two curves already appears in the 3-to-6-layer transition. Therefore, we zoom in and examine what is actually happening between steps 150K and 170K of Figure 10. Figure 11 displays the loss curves, demonstrating how *stacking*'s loss drops below *random*'s loss at around 158K steps.

Examining the gradients, we see that *stacking* yields much larger gradients for the the key and query matrices of the new layers than does *random* (Figure 12). This is in contrast to Figure 13, where the gradient norms of *stacking* and *random* are comparable.

This does not necessarily imply larger overall movement, though. Figure 14 shows the ℓ_2 distance traveled by each layer in *stacking* vs. *random*. We see that, in the later

Figure 8. Stacking quality at 10K iterations vs. SQuAD 2.0 performance.

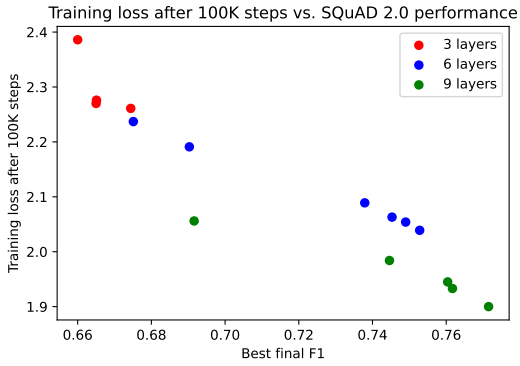

Figure 9. Stacking quality at 100K iterations vs. SQuAD 2.0 performance.

layers, *stacking*'s parameters do move less than *random*'s, though the gap between stacking and random is perhaps smaller than one might suspect if "parameter similarity" were behind the success of stacking. In Figure 15, we see the ℓ_2 distance traveled for the key and query matrices of the newly-added layers. Despite the larger gradients in *stacking*, the overall change in weight matrices (measured in ℓ_2 distance) is not larger.

Figure 10. Loss curves for *stacking* vs. *random* (BERT-BASE).

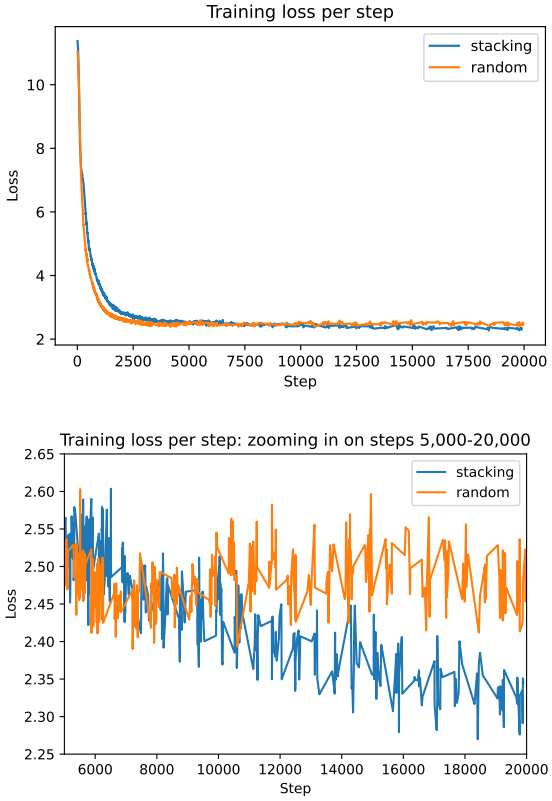

Figure 11. Loss curves for *stacking* vs. *random*, zooming in on steps 150K-170K of Figure 10 (renumbered 0-20K here, for ease of visualization). The top plot shows all 20K steps, and the bottom plot further zooms in on steps 5K-20K, equivalent to steps 155K-170K of Figure 10.

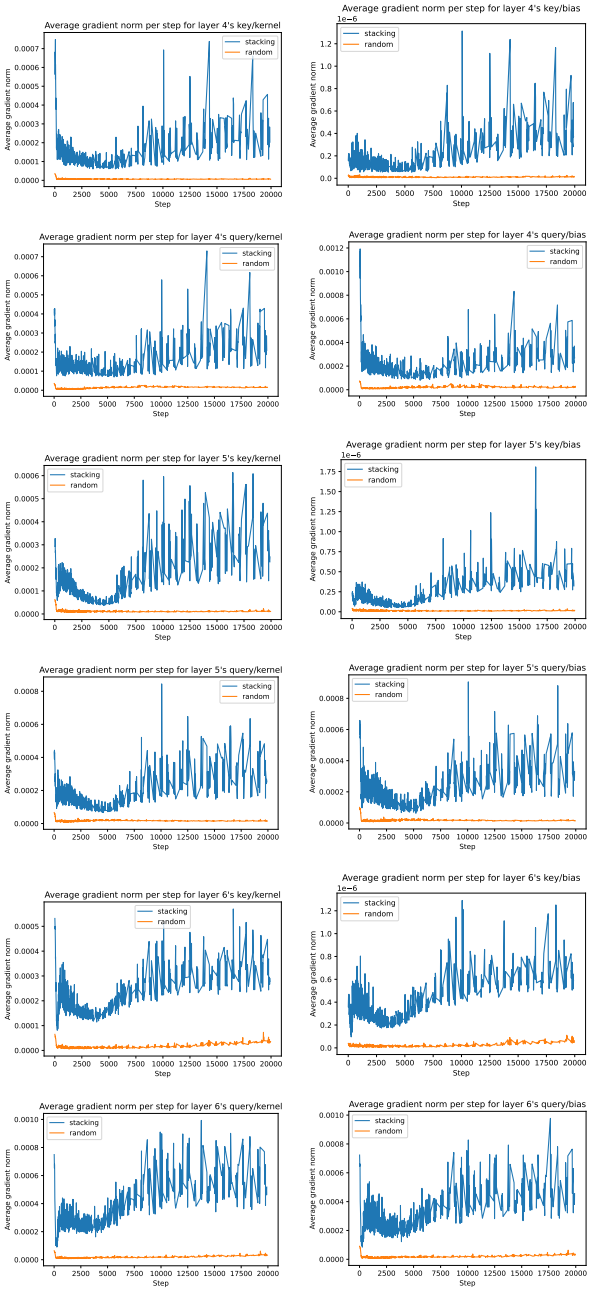

Figure 12. Average gradient norms of the key and query matrices of the new Transformer layers (layer 4-6) in *stacking* and *random*. The x-axis spans steps 0 to 20K, corresponding to steps 150K-170K in Figure 10.

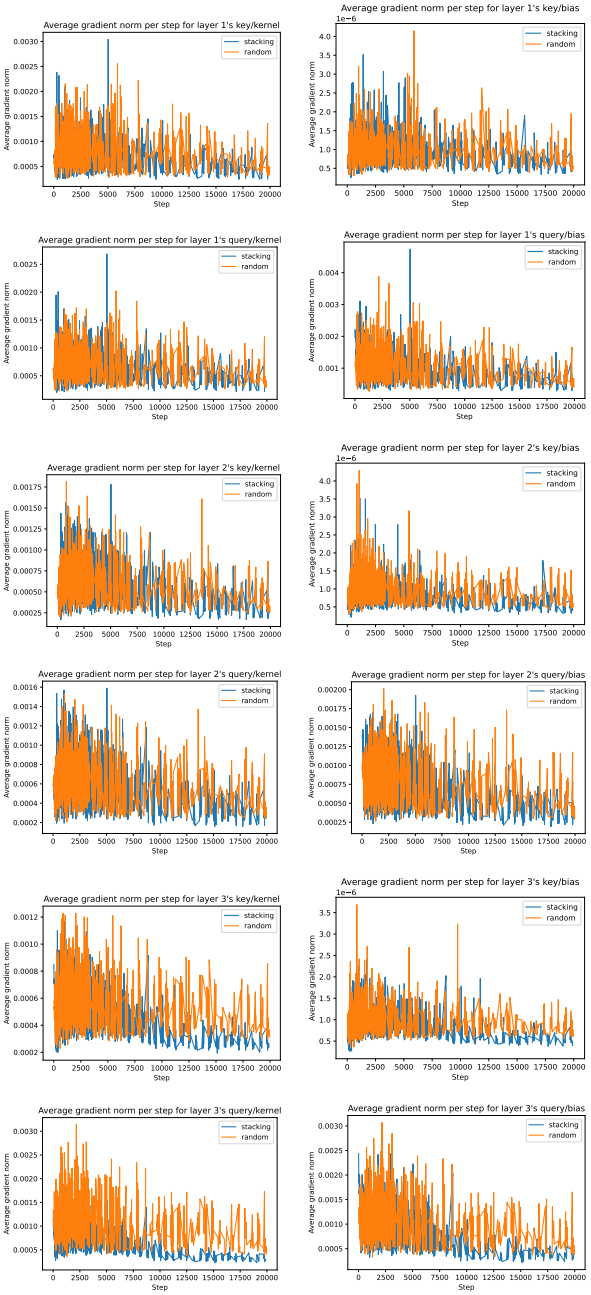

Figure 13. Average gradient norms of the key and query matrices of the existing Transformer layers (layer 1-3) in *stacking* and *random*. The x-axis spans steps 0 to 20K, corresponding to steps 150K-170K in Figure 10.

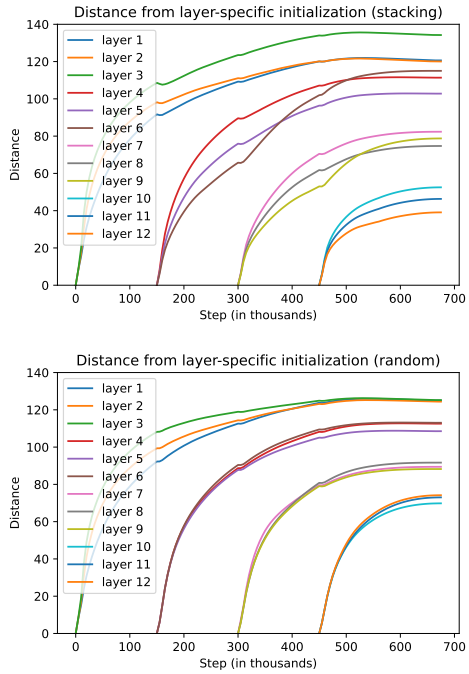

Figure 14. ℓ_2 distance from layer-specific initialization (one curve per layer), for *stacking* vs. *random*.

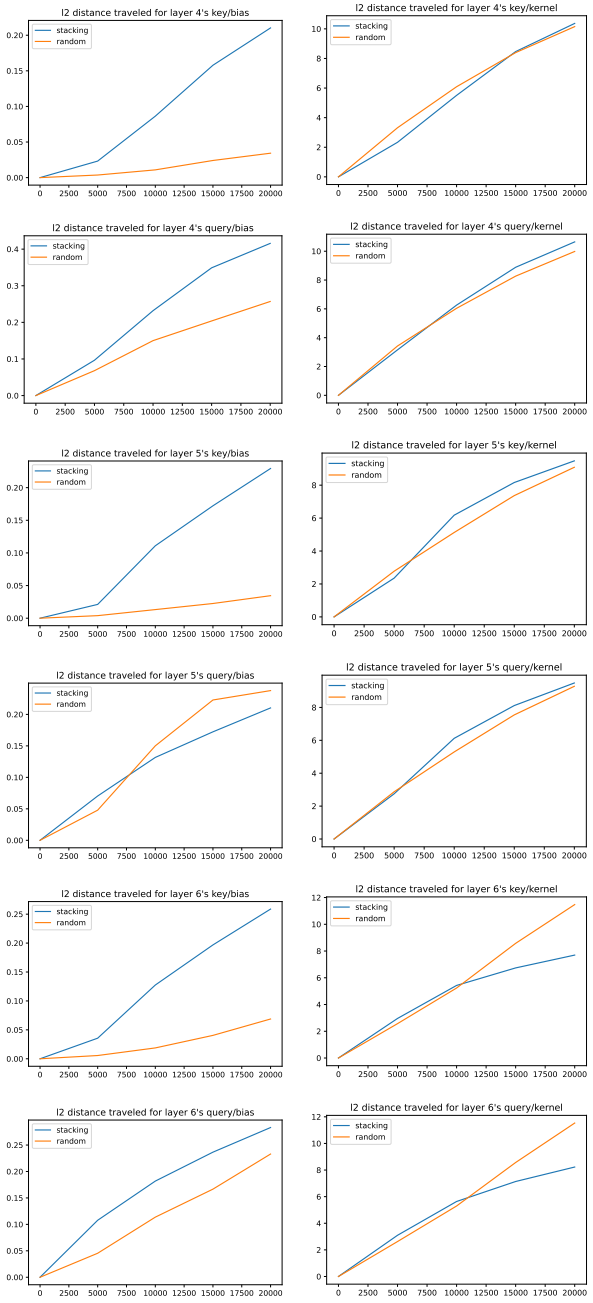

Figure 15. ℓ_2 distance from initialization over time for the key and query matrices of the new Transformer layers. The x-axis spans steps 0 to 20K, corresponding to steps 150K-170K in Figure 10.