

AdapterDrop: On the Efficiency of Adapters in Transformers

Andreas Rücklé, Gregor Geigle, Max Glockner,
 Tilman Beck, Jonas Pfeiffer, Nils Reimers, Iryna Gurevych
 Ubiquitous Knowledge Processing Lab (UKP)
 Department of Computer Science, Technische Universität Darmstadt
www.ukp.tu-darmstadt.de

Abstract

Massively pre-trained transformer models are computationally expensive to fine-tune, slow for inference, and have large storage requirements. Recent approaches tackle these shortcomings by training smaller models, dynamically reducing the model size, and by training light-weight adapters. In this paper, we propose **AdapterDrop**, removing adapters from lower transformer layers during training and inference, which incorporates concepts from all three directions. We show that AdapterDrop can dynamically reduce the computational overhead when performing inference over multiple tasks simultaneously, with minimal decrease in task performances. We further prune adapters from AdapterFusion, which improves the inference efficiency while maintaining the task performances entirely.

1 Introduction

While transfer learning has become the go-to method for solving NLP tasks (Pan and Yang, 2010; Torrey and Shavlik, 2010; Ruder, 2019; Howard and Ruder, 2018; Peters et al., 2018), transformer-based models are notoriously deep requiring millions or even billions of parameters (Radford et al., 2018; Devlin et al., 2019; Radford et al., 2019; Liu et al., 2019; Brown et al., 2020). This results in slow inference times and requires large storage space for each fine-tuned model.

Recently, three independent lines of research have evolved to tackle these shortcomings. (1) Smaller and faster models that are either distilled or trained from scratch (Sanh et al., 2019; Sun et al., 2020). (2) Robustly trained transformers in which layers can be dropped at run-time, thereby decreasing inference time dynamically (Fan et al., 2020). (3) Adapters, which, instead of fully fine-tuning the model, only train a newly introduced set of weights at every layer, thereby sharing the

majority of parameters between tasks (Houlsby et al., 2019b; Bapna and Firat, 2019; Pfeiffer et al., 2020b). Adapters have been shown to work well for machine translation (Bapna and Firat, 2019), cross-lingual transfer (Pfeiffer et al., 2020c; Üstün et al., 2020), QA text matching (Rücklé et al., 2020), and task composition for transfer learning (Stickland and Murray, 2019; Pfeiffer et al., 2020a; Lauscher et al., 2020; Wang et al., 2020). Despite their recent popularity, the computational efficiency of adapters has not been explored beyond parameter efficiency. This leaves the question of whether adapters are also efficient at training and inference time.

We close this gap and improve the training and inference efficiency of adapters by incorporating ideas from all three directions mentioned above. We propose different techniques for dropping out adapters from transformers at training and inference time, resulting in more efficient adapter-based models that are dynamically adjustable regarding the available computational resources. Our approaches are agnostic to the pre-trained transformer model (e.g., base, large), which makes them broadly applicable.

Our contributions:

1. We are the first to establish the computational efficiency of adapters compared to full fine-tuning. We show that the training steps of adapters are up to 60% faster than full model fine-tuning with common hyperparameter choices, while only being 4–6% slower during inference.
2. We propose AdapterDrop, the efficient and dynamic removal of adapters with minimal impact on the task performances. We show that dropping adapters from lower transformer layers considerably improves the inference speed in multi-task settings. For example, with adapters dropped from the first five layers, AdapterDrop is 39% faster when performing inference on 8

Setting	Adapter	Relative speed (for Seq.Len./Batch)			
		128/16	128/32	512/16	512/32
Training	Houlsby	1.48	1.53	1.36	1.33
	Pfeiffer	1.57	1.60	1.41	1.37
Inference	Houlsby	0.94	0.94	0.96	0.96
	Pfeiffer	0.95	0.95	0.96	0.96

Table 1: Relative speed of adapters compared to fully fine-tuned models. For example, 1.6 for training with the Pfeiffer adapter means that we can perform 1.6 training steps with this adapter in the time of one update step with full model fine-tuning.

tasks simultaneously.

3. We prune adapters from adapter compositions in AdapterFusion (Pfeiffer et al., 2020a) and retain only the most important adapters after transfer learning, resulting in faster inference while maintaining the task performances entirely.

2 Efficiency of Adapters

We first establish the computational efficiency of adapters *without* AdapterDrop. As illustrated in Figure 1, significant differences exist in the forward and backward pass when fine-tuning adapters compared to fully fine-tuning the model. In the forward pass, adapters add complexity with the additional components; however, it is not necessary to backpropagate through the entire model during the backward pass. We compare the training and inference speed of full model fine-tuning against the recently proposed adapter architectures of Houlsby et al. (2019b) and Pfeiffer et al. (2020a) (depicted in Figure 1) using the AdapterHub.ml (Pfeiffer et al., 2020b) framework. We conduct our measurements with the transformer configuration of BERT base and verify them with different GPUs.¹

We provide measurements corresponding to common experiment configurations in Table 1.

Training. Adapters are always considerably faster compared to full model fine-tuning—60% faster in some configurations. The two adapter architectures differ only marginally in terms of training efficiency: due to its simpler architecture, training steps of the Pfeiffer adapters are slightly faster. The magnitude of the differences depends on the input size; the available CUDA cores are the primary bottleneck.² We do not observe any particular

¹We experiment with newer and older GPUs, Nvidia V100 and Titan X, respectively. See Appendix A.1 for details.

²We include detailed plots in Appendix G.1.

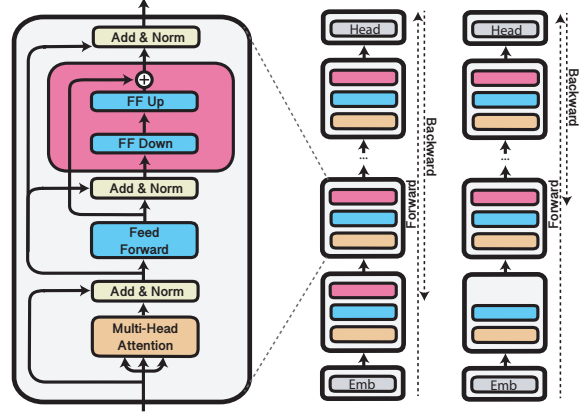


Figure 1: Standard adapter fine-tuning vs. AdapterDrop fine-tuning. The left model includes adapters at every layer whereas the right model has adapters dropped at the first layer. The arrows to the right of each model indicate the information flow for the *Forward* and *Backward* pass through the model.

differences between adapters and full fine-tuning regarding the training convergence.³

The training speedup can be explained through decreased overhead of gradient computation. Most of the parameters are frozen when using adapters and it is not necessary to backpropagate through the first components (see Figure 1).

Inference. Adapters are around 94–96% as fast at inference compared to fully fine-tuned models, which varies depending on the input (e.g., batch size and sequence length). This can have a considerable impact when deployed at scale. The central focus of this work is thus to improve the inference efficiency of Adapters.

3 AdapterDrop

Even though we have established that adapters are more efficient in terms of training time compared to full fine-tuning, there is a perpetuate need for sustainable and efficient models (Strubell et al., 2019). For example, backpropagating through as few layers as possible would further improve the efficiency of training adapters. The efficiency for inference can be improved by sharing representations at lower transformer layers when simultaneously performing inference for multiple tasks. We establish this in Table 2, finding that models are up to 8.4% faster with **every** shared layer.

Motivated by these findings, we propose AdapterDrop: dynamically removing adapters from

³We also pre-train adapters with masked language modeling, finding that this does not yield better results (Appendix B).

Simultaneous Tasks	2	4	8	16
Speedup (each layer)	4.3%	6.6%	7.8%	8.4%

Table 2: Speedup for each shared transformer layer when performing inference for multiple tasks simultaneously (details are given in Appendix G.2)

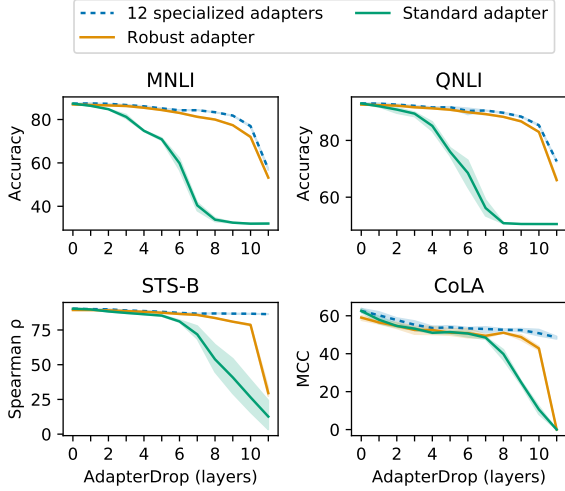


Figure 2: Task performances in relation to the dropped lower layers during evaluation (all eight tasks are included in Figure 7). ‘Standard adapter’ is trained with no dropped layers.

lower transformer layers (depicted in Figure 1). AdapterDrop is similar to dropping out entire transformer layers (Fan et al., 2020), however, specialized to adapter settings—where lower layers often have a small impact on the task performances (Houlsby et al., 2019a).

We study two training methods for AdapterDrop: (1) **Specialized** AdapterDrop: Removing adapters from the first n transformer layers, where n is fixed during training. This yields separate models for each possible n . (2) **Robust** AdapterDrop: Drawing the integer n randomly from $[0, 11]$ for each training batch.⁴ This yields *one robust* model that is applicable to a varying number of dropped layers. We study the effectiveness of AdapterDrop on the devsets of the GLUE benchmark (Wang et al., 2018) using RoBERTa base (Liu et al., 2019).⁵

Figure 2 shows that specialized AdapterDrop maintains good results even with several dropped layers. With the first five layers dropped, specialized AdapterDrop maintains 97.1% of the origi-

⁴We also explored dropping adapters from randomly chosen layers (instead of early layers). This generally performs worse and it requires selecting a suitable dropout rate.

⁵The detailed setup is listed in Appendix A.2.

Adapters	AF vs. Full FT		AF vs. Adapter	
	Training	Inference	Training	Inference
2	0.92	0.64	0.57	0.68
8	0.53	0.38	0.33	0.40
16	0.33	0.24	0.21	0.26

Table 3: Relative speed of AdapterFusion (with 2/8/16 adapters) compared to a fully fine-tuned model and compared to a single-task adapter (right). Measured with a batch size of 32, and a sequence length of 128.

nal performance (averaged over all eight GLUE tasks; see Table 4). Moreover, robust AdapterDrop achieves comparable results, and with five layers dropped it maintains 95.4% of the original performance (on avg). The advantage of *robust* over *specialized* AdapterDrop is that the *robust* variant can be dynamically scaled. Based on current available computational resources, *robust* AdapterDrop can (de)activate layers with the same set of parameters, whereas *specialized* AdapterDrop needs to be trained for every setting explicitly.

The efficiency gains can be large. When performing inference for multiple tasks simultaneously, we measure inference speedups of 21–42% with five dropped layers (Table 2).⁶ Training of our robust adapters is also more efficient, which increases the speed of training steps by 26%.⁷

4 Efficiency of AdapterFusion

AdapterFusion (Pfeiffer et al., 2020a) leverages the knowledge of *several* adapters from different tasks and learns an optimal combination of the adapters’ output representations for a single target task (see Figure 3). AdapterFusion (AF) is particularly useful for small training sets where learning adequate models is difficult. Despite its effectiveness, AF is computationally expensive because all included adapters are passed through sequentially.⁸

Table 3 shows that the differences can be substantial for both training and inference. For instance, compared to a fully fine-tuned model, AF with eight adapters is around 47% slower at training time and 62% slower at inference.⁹

⁶For more details see Appendix G.2

⁷Every dropped adapter improves the speed of training steps by 4.7% and we drop on average 5.5 adapters when training robust adapter models (more hyperparameter settings and details are given in Appendix G.2).

⁸We also test AF with parallel operations and found no efficiency gains (see Appendix H).

⁹All with Pfeiffer adapter and depending on the input size. We provide more measurements in Appendix G.3.

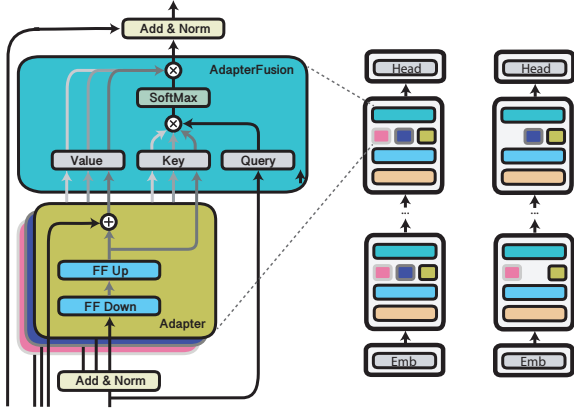


Figure 3: Standard AdapterFusion vs. AdapterFusion pruning, each with 3 adapters initially. The left model includes all adapters at every layer whereas the right model has one adapter pruned at every layer.

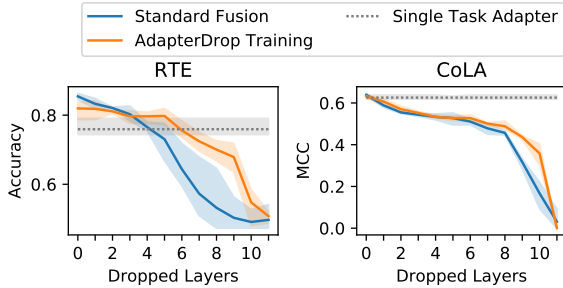


Figure 4: Comparison of AdapterFusion with (orange) and without (blue) AdapterDrop training during inference when omitting early AF layers.

5 AdapterDrop for AdapterFusion

There exists considerable potential for improving the efficiency of AF, especially at *inference*. We address this with two variants of AdapterDrop for AF by (1) removing entire AF layers; (2) pruning the least important adapters from AF models.

5.1 Removing AdapterFusion Layers

We fuse the adapters from all eight GLUE tasks and observe the largest gains of AF on RTE and CoLA. We additionally train robust AF models with the same procedure as in §3. We investigate from how many lower layers we can remove AF at test time while still outperforming the corresponding single-task adapter (without AdapterDrop).

Figure 4 shows that AF performs better than the single-task adapter on RTE until removing AF from the first five layers. This improves the inference efficiency by 26%.¹⁰ On CoLA, we observe a different trend. Removing AF from the first layer results

¹⁰We include detailed measurements in Appendix G.4.

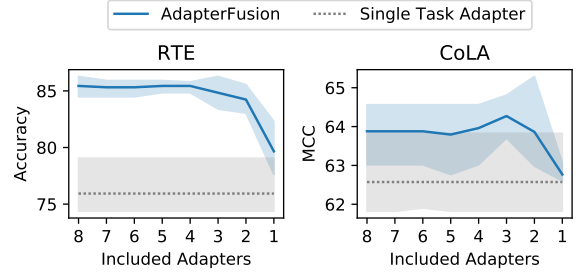


Figure 5: Task performance of AdapterFusion Pruning. AF is trained with eight adapters, and we gradually remove the least important from the model.

in more noticeable performance decreases, achieving lower task performances than the single-task adapter. This is in line with recent work showing that linguistic tasks such as CoLA heavily rely on information from the first layers (Vulić et al., 2020). We deliberately highlight that AdapterDrop might not be suitable for all tasks. However, Figure 7 shows that CoLA represents an extreme case.

5.2 AdapterFusion Pruning

The inference efficiency of AF largely depends on the number of fused adapters, see Table 3. Thus, we can achieve improvements by pruning adapters from the trained AF models (depicted in Figure 3). In each fusion layer, we record the average adapter activations—their relative importance—using all instances of the respective AF training set. We then remove the adapters with lowest activations.

Figure 5 demonstrates that we can safely remove most adapters in AF without affecting the task performance. With just two remaining adapters, we achieve comparable results to the full AF models with eight adapters and improve the inference speed by 68%.

6 Conclusion

Adapters have emerged as a suitable alternative to full model fine-tuning, and their most widely claimed computational advantage is the small model size. In this work, we have demonstrated that the advantages of adapters go far beyond mere parameter efficiency. Even without extensions, training steps with adapters are up to 60% faster.

AdapterDrop considerably expands these advantages by dropping a variable number of adapters from lower transformer layers. We *dynamically* reduce the computational overhead at run-time when performing inference over multiple tasks and maintain task performances to a large extent. We also

considerably improve the efficiency of AdapterFusion by pruning most adapters from the model while maintaining the performances entirely.

We believe that our work can be widely extended and that there exist many more directions to obtain efficient adapter-based models. For instance, we could explore more efficient pre-trained adapters,¹¹ sharing the adapter weights across layers,¹² or pruning adapters from AdapterFusion at training time.¹³ In the Appendix to this paper, we present preliminary results for several related ideas, which may serve as a starting point for future work.

Acknowledgments

Andreas Rücklé, Max Glockner, and Nils Reimers are supported by the German Federal Ministry of Education and Research (BMBF) and the Hesse State Ministry for Higher Education, Research and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE. Tilman Beck is supported by the German Research Foundation (DFG) as part of the Research Training Group KRITIS No. GRK 2222 and by the German Federal Ministry of Education and Research (BMBF) as part of the Software Campus program under the promotional reference 01—S17050. Jonas Pfeiffer is supported by the LOEWE initiative (Hesse, Germany) within the emergenCITY center. We gratefully acknowledge the support of NVIDIA Corporation with the donation of the Titan X Pascal GPU used for this research.

References

- Ankur Bapna and Orhan Firat. 2019. [Simple, Scalable Adaptation for Neural Machine Translation](#). In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP 2019)*, pages 1538–1548.
- Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. [Language Models are Few-Shot Learners](#). *arXiv preprint arXiv:2005.14165*.
- Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. [BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding](#). In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL 2019)*, pages 4171–4186.
- Angela Fan, Edouard Grave, and Armand Joulin. 2020. [Reducing Transformer Depth on Demand with Structured Dropout](#). In *8th International Conference on Learning Representations, (ICLR 2020)*, pages 26–30.
- Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. 2019a. [Parameter-Efficient Transfer Learning for NLP](#). In *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 2790–2799, Long Beach, California, USA. PMLR.
- Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin de Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. 2019b. [Parameter-Efficient Transfer Learning for NLP](#). In *Proceedings of the 36th International Conference on Machine Learning (ICML 2019)*.
- Jeremy Howard and Sebastian Ruder. 2018. [Universal Language Model Fine-tuning for Text Classification](#). In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, (ACL 2018)*, pages 328–339.
- Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. 2019. [ALBERT: A Lite BERT for Self-supervised Learning of Language Representations](#). *arXiv preprint arXiv:1909.11942*.
- Anne Lauscher, Olga Majewska, Leonardo F. R. Ribeiro, Iryna Gurevych, Nikolai Rozanov, and Goran Glavas. 2020. [Common Sense or World Knowledge? Investigating Adapter-Based Knowledge Injection into Pretrained Transformers](#). *arXiv preprint arXiv:2005.11787*.
- Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. [RoBERTa: A Robustly Optimized BERT Pretraining Approach](#). *arXiv preprint arXiv:1907.11692*.

¹¹In Appendix B, we compare randomly initialized adapters and pre-trained adapters (with MLM). Our results suggest that different strategies are necessary for adapters as compared to fully fine-tuned transformers, which can serve as a starting point for further experiments.

¹²We show in Appendix D that an adapter with cross-layer parameter sharing achieves comparable results to a standard adapter while drastically reducing the number of fine-tuned parameters.

¹³Appendix E shows that we can randomly dropout 75% of the adapters during AdapterFusion training with a minimal impact on the task performance.

- Sinno Jialin Pan and Qiang Yang. 2010. [A Survey on Transfer Learning](#). *IEEE Transactions on Knowledge and Data Engineering*, 22(10):1345–1359.
- Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. 2018. [Deep Contextualized Word Representations](#). In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, (NAACL 2018), pages 2227–2237.
- Jonas Pfeiffer, Aishwarya Kamath, Andreas Rücklé, Kyunghyun Cho, and Iryna Gurevych. 2020a. [AdapterFusion: Non-Destructive Task Composition for Transfer Learning](#). *arXiv preprint arXiv:2005.00247*.
- Jonas Pfeiffer, Andreas Rücklé, Clifton Poth, Aishwarya Kamath, Ivan Vulić, Sebastian Ruder, Kyunghyun Cho, and Iryna Gurevych. 2020b. [AdapterHub: A Framework for Adapting Transformers](#). In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP 2020): Systems Demonstrations*.
- Jonas Pfeiffer, Ivan Vulić, Iryna Gurevych, and Sebastian Ruder. 2020c. [MAD-X: An Adapter-based Framework for Multi-task Cross-lingual Transfer](#). In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP 2020)*.
- Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. [Improving Language Understanding by Generative Pre-Training](#). *Technical report, OpenAI*.
- Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. [Language Models are Unsupervised Multitask Learners](#). *Technical report, OpenAI*.
- Andreas Rücklé, Jonas Pfeiffer, and Iryna Gurevych. 2020. [MultiCQA: Exploring the Zero-Shot Transfer of Text Matching Models on a Massive Scale](#). In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP 2020)*.
- Sebastian Ruder. 2019. [Neural Transfer Learning for Natural Language Processing](#). Ph.D. thesis, National University of Ireland, Galway.
- Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2019. [DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter](#). *arXiv preprint arXiv:1910.01108*.
- Asa Cooper Stickland and Iain Murray. 2019. [BERT and PALs: Projected Attention Layers for Efficient Adaptation in Multi-Task Learning](#). In *Proceedings of the 36th International Conference on Machine Learning (ICML 2019)*, pages 5986–5995.
- Emma Strubell, Ananya Ganesh, and Andrew McCallum. 2019. [Energy and Policy Considerations for Deep Learning in NLP](#). In *Proceedings of the 57th Conference of the Association for Computational Linguistics (ACL 2019)*, pages 3645–3650.
- Zhiqing Sun, Hongkun Yu, Xiaodan Song, Renjie Liu, Yiming Yang, and Denny Zhou. 2020. [MobileBERT: a Compact Task-Agnostic BERT for Resource-Limited Devices](#). In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL 2020)*, pages 2158–2170.
- Lisa Torrey and Jude Shavlik. 2010. Transfer learning. In *Handbook of research on machine learning applications and trends: algorithms, methods, and techniques*, pages 242–264. IGI Global.
- Ahmet Üstün, Arianna Bisazza, Gosse Bouma, and Gertjan van Noord. 2020. [UDapter: Language Adaptation for Truly Universal Dependency Parsing](#). In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP 2020)*.
- Ivan Vulić, Edoardo Maria Ponti, Robert Litschko, Goran Glavaš, and Anna Korhonen. 2020. [Probing Pretrained Language Models for Lexical Semantics](#). In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP 2020)*.
- Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. 2018. [GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding](#). In *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, pages 353–355.
- Ruize Wang, Duyu Tang, Nan Duan, Zhongyu Wei, Xuanjing Huang, Jianshu Ji, Guihong Cao, Daxin Jiang, and Ming Zhou. 2020. [K-Adapter: Infusing Knowledge into Pre-Trained Models with Adapters](#). *arXiv preprint arXiv:2002.01808*.

A Measuring Computational and Task Performance

A.1 Computational Efficiency

We use Python 3.6, PyTorch 1.5.1, CUDA 10.1 for all measurements. We repeat them with two different GPUs: NVIDIA Tesla V100 PCIe (32GB) and a NVIDIA Titan X Pascal (12GB). We make use of the `torch.cuda.Event` class and `torch.cuda.synchronize` to measure *only* the exact period of time of a training (or inference) step.¹⁴ For both inference and training, we repeat the respective step 300 times. We report the median

¹⁴This is necessary due to the asynchronous nature of the command execution on CPU and GPU.

to mitigate the impact of outliers caused by GPU warmup.

Relativ speed. We define the relative speed of an adapter compared full model fine-tuning as: $\frac{S_a}{S_f}$ where S_a and S_f are the time of one step with the adapter model and the fully fine-tuned model, respectively. For example, a relative speed of 1.5 means that the adapter model can perform 1.5 steps in the time the fully fine-tuned model performs one step.

Speedup. Speedup describes the positive change in relative speed of an adapter model when using AdapterDrop (or another method). A speedup of $p\%$ means that the adapter model with AdapterDrop requires only $(1 - p/100) \times$ of the runtime than the adapter model without AdapterDrop.

The speedup of AdapterDrop (and AdapterFusion) are additive. If dropping one layer results in $p\%$ speedup, dropping two layers results in $2p\%$ speedup, etc.

A.2 Task Performances

We study the task performances of adapter models on the popular GLUE benchmark (Wang et al., 2018). Following Devlin et al. (2019), we exclude the WNLI because of the problematic data construction.¹⁵ We perform our analyses using RoBERTa base (Liu et al., 2019) as our pre-trained model and report the mean and standard deviation over three runs of the best development performance evaluated after every epoch. We train larger data sets (SST-2, MNLI, QNLI, and QQP) for 10 epochs and the rest of the data sets for 20 epochs. We use a batch size of 32 and, if not otherwise noted, the default hyperparameters for adapter fine-tuning as in (Pfeiffer et al., 2020a).

B Adapter Initialization and Convergence

Besides measuring training and inference time, we are interested in (1) how using adapters compare to standard RoBERTa-base with regards to downstream task convergence, and (2) if initializing adapters with pre-trained weights using *masked language modeling* can lead to faster convergence.

First, we compare RoBERTa-base with adapter models using the architecture proposed by Pfeiffer et al. (2020a). Second, we pretrain an adapter

with masked language modeling (MLM) using documents from the English Wikipedia.¹⁶ The results for both experiments are visualized in Figure 6. When comparing RoBERTa-base with randomly initialized adapters, We find that adapters do not come at the cost of requiring more training steps for convergence (1). For several of the eight GLUE tasks, we observe similar convergence behavior with the standard RoBERTa-base model and its counterpart using adapters.

Further, we observe across all tasks that initializing the adapter weights with MLM pre-training does not have a substantial impact on the downstream task convergence (compared to a randomly initialized adapter). Thus, we find no evidence that pre-training of adapters with our masked language modeling objective leads to better convergence performance in our experiments (2).

C Detailed Results: AdapterDrop Task Performances

We plot the detailed task performances of AdapterDrop with the different training strategies in Figure 7. The relative differences of AdapterDrop to a standard adapter with no AdapterDrop are given in Table 4.

D Adapter with Cross-Layer Parameter Sharing

We can further reduce the number of parameters required for each task by sharing the weights of the adapters across all transformer layers. This is similar to weight sharing in ALBERT (Lan et al., 2019), but specialized on adapters and can therefore be applied to a wide range of pre-trained models.

We use the Pfeiffer adapter architecture in our experiments with the same hyperparameters as in Appendix A.2. Because cross-layer parameter sharing reduces the capacity of adapter models, we study the impact of the adapter compression rate. The compression rate refers to the down-projection factor in the adapter’s bottleneck layer and thus impacts the its capacity (the compression rate specifies by how much ‘FF Down’ in Figure 1 compresses the representations). The standard compression rate is 16, and smaller values result in a larger model capacity.

¹⁵See <https://gluebenchmark.com/faq>

¹⁶We used a recent dump of English Wikipedia. We train with a batch size of 64 and for 250k steps such that no sentence was used twice.

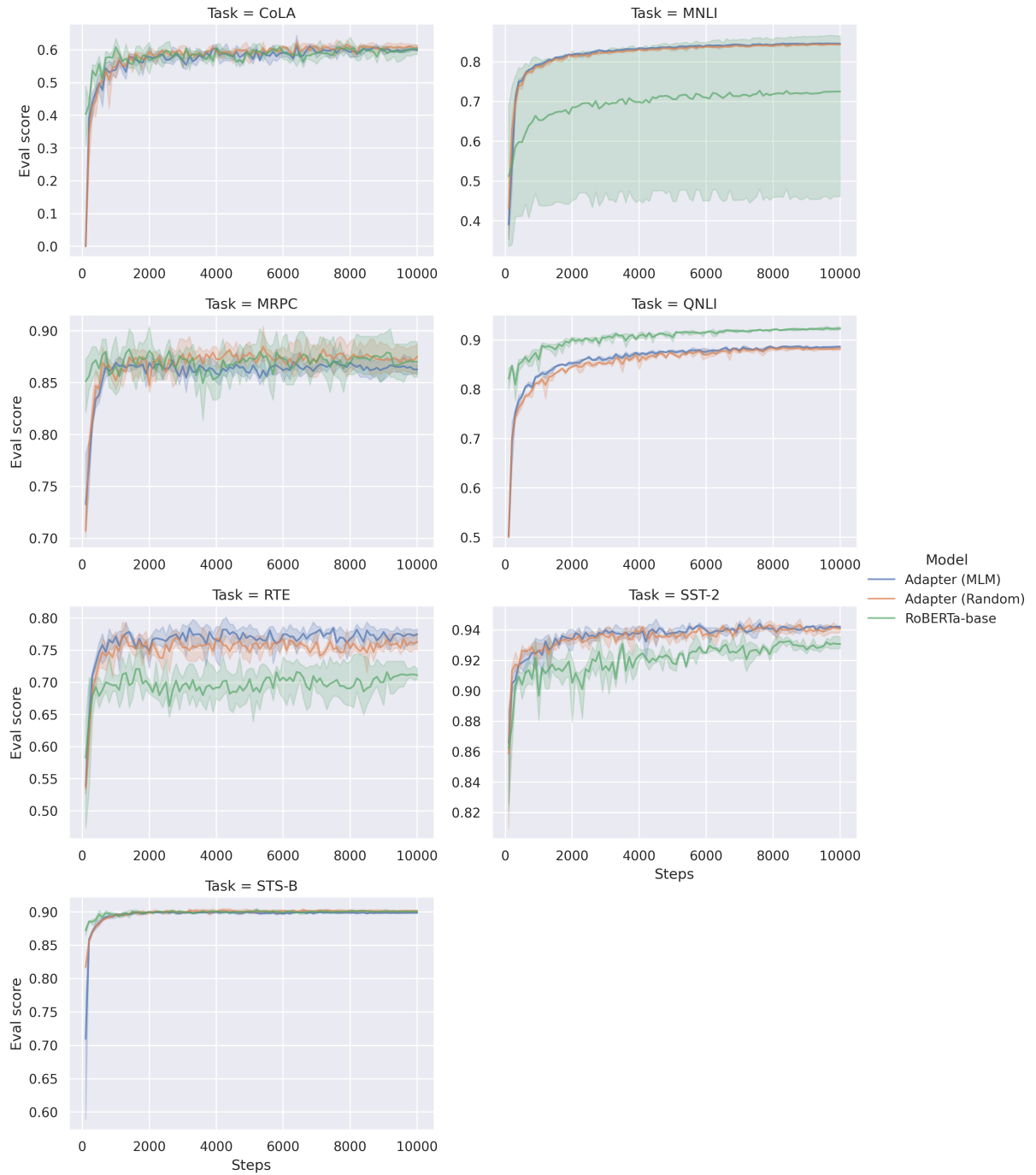


Figure 6: Evaluation performance of fine-tuning RoBERTa-base in comparison with different initialization strategies for adapters (randomly initialized vs. pre-trained on masked language modeling task). Training was conducted for 10k steps with a learning rate of $5e-05$ for RoBERTa-base and 0.0001 for adapters, respectively.

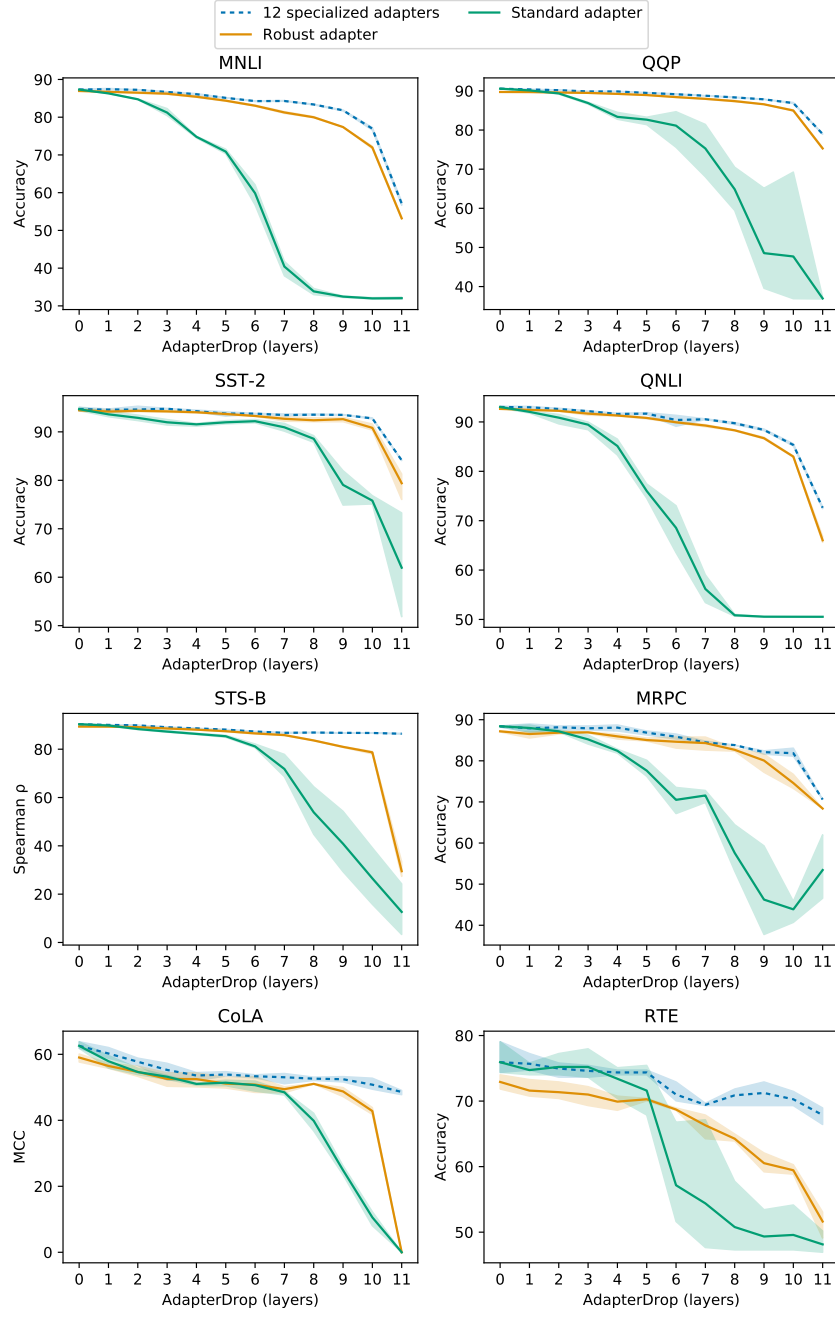


Figure 7: The **AdapterDrop** task performances for all eight GLUE tasks in relation to the dropped layers. ‘12 specialized adapters’ refers to the performance of individual models trained for each AdapterDrop setting separately (i.e., 12 models); ‘Standard adapter’ refers to the adapter that is trained with no dropped layers; AdapterDrop training refers to the adapter that is trained with our proposed training procedure.

	Dropped Layers											
	0	1	2	3	4	5	6	7	8	9	10	11
Standard adapter	100.0	98.5	97.1	95.3	92.0	89.0	82.2	74.6	64.5	54.5	49.3	43.3
Specialized AdapterDrop (12 models)	100.0	99.5	98.9	98.2	97.6	97.1	95.9	95.3	95.1	94.3	92.5	82.9
Robust AdapterDrop	98.5	97.7	97.3	96.8	96.1	95.4	94.5	93.3	92.2	89.9	85.9	62.0

Table 4: Model performances with **AdapterDrop** in relation to a standard adapter with no dropped layers. We report the percentage of retained task performance compared to the standard adapter with no dropped layers during evaluation. The results are averaged over all eight GLUE task. A value of 97.1 for specialized AdapterDrop with five dropped layers means that the model achieves 97.1% of the performance compared to the standard adapter with no dropped layers. Performance scores for each task can be found in Figure 7.

	Standard	Cross-Layer Parameter Sharing		
	Compression rate = 16	1.33	4	16
SST-2	94.7 \pm 0.3	94.2 \pm 0.3	94.2 \pm 0.1	94.1 \pm 0.4
QNLI	93.0 \pm 0.2	92.4 \pm 0.1	93.1 \pm 0.1	90.6 \pm 1.4
MNLI	87.3 \pm 0.1	87.0 \pm 0.1	87.1 \pm 0.0	86.2 \pm 0.2
QQP	90.6 \pm 0.0	90.8 \pm 0.1	90.2 \pm 0.0	88.6 \pm 0.5
CoLA	62.6 \pm 0.9	60.3 \pm 1.6	60.8 \pm 0.4	57.2 \pm 1.0
MRPC	88.4 \pm 0.1	88.2 \pm 0.7	88.5 \pm 1.1	86.8 \pm 0.5
RTE	75.9 \pm 2.2	69.4 \pm 0.5	71.5 \pm 2.7	71.5 \pm 1.0
STS-B	90.3 \pm 0.1	89.5 \pm 0.1	89.7 \pm 0.3	89.0 \pm 0.7
Average	85.35	83.98	84.39	83.0
<i>Params</i>	884k	884k	295k	74k

Table 5: Task performance scores of the standard approach with separate adapter weights vs. **cross-layer parameter sharing**. The compression rate denotes the factor by which ‘FF Down’ in Figure 1 compresses the representations. The number of parameters is given without classification heads.

Table 5 shows that cross-layer parameter sharing with the same compression rate of 16 largely maintains the performance compared to separate weights with an average difference of 2.35%. With a smaller compression rate of 4, we close this gap by more than 50% while still requiring 66% fewer parameters.¹⁷ The resulting models are lightweight: our shared adapter with a compression rate of 16 requires only 307KB storage space.

E Training AdapterFusion with Dropout

We investigate the random dropout of adapters from AdapterFusion during training (using our eight task adapters as in §4) to improve the speed of training steps. Each layer randomly selects different adapters to drop out. This means that the model itself may still use the knowledge from all tasks, although not in the layers individually.

Table 6 shows the results for the four small-

¹⁷Even smaller compression rates do not yield similar gains.

	Fusion Dropout			
	0%	25%	50%	75%
CoLA	63.9 \pm 0.6	62.9 \pm 0.8	62.4 \pm 0.7	60.4 \pm 0.2
MRPC	88.4 \pm 0.1	89.2 \pm 0.5	89.2 \pm 0.4	89.3 \pm 0.1
RTE	85.4 \pm 0.7	82.8 \pm 1.9	82.1 \pm 0.3	80.9 \pm 1.1
STS-B	90.2 \pm 0.1	90.2 \pm 0.1	90.1 \pm 0.1	89.9 \pm 0.1
Speedup (8)	-	15.9%	39.4%	73.7%
Speedup (16)	-	22.5%	58.2%	120.6%

Table 6: Development scores of **AdapterFusion** (compression rate 16x) with or without fusion dropout during *training*. Fusion dropout of 50% means that each adapter has a 50% chance of not being used as input to the fusion layer. The **speedup** depends on the total number of adapters used in AdapterFusion (8 adapters in our setting here, 16 used by Pfeiffer et al. (2020a))

est GLUE tasks in terms of training data size. The speedup that we achieve with AdapterFusion dropout can be substantial: with a dropout rate of 75% (i.e., dropping out 6 out of our 8 adapters) each training step is 74% faster on average (with a sequence length of 128, a batch size of 32). We observe no clear trend in terms of task performances. Fusion dropout leads to consistent decreases on RTE and CoLA, only a small impact on STS-B (no difference when dropping out 25% of adapters), and yields improvements on MRPC.

The effectiveness of Fusion dropout, thus, depends on the individual downstream task. Nevertheless, we believe that this methods could be suitable, e.g., for resource-constrained settings.

F Detailed Results: Removing AdapterFusion Layers

The computational overhead of AF can be reduced during inference by decreasing the number of adapters. We investigate how dropping AF layers impacts the performance on the four smallest GLUE tasks (MRPC, STS-B, CoLA, RTE) and

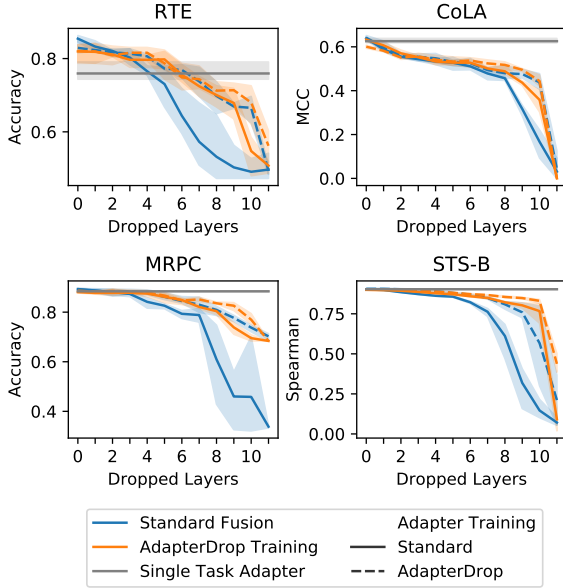


Figure 8: Performance of AF by the number of dropped AF layers. We show the results for AF and the used adapters (both with and without AdapterDrop), and compare the performance with a standard single task adapter.

visualize the results in Figure 8. In this experiment we compare the performance of AF with and without AdapterDrop during training. For both, we use standard adapters as well as adapters created via AdapterDrop as basis for AF. Unsurprisingly, the performance of AF without AdapterDrop within the adapters or fusion drops fastest on all four datasets. Using AdapterDrop when creating the adapters, applying AdapterDrop on AF, or the combination of both significantly reduces the performance drop when omitting fusion layers during inference. On RTE and MRPC, multiple AF layers can be omitted while still performing en par with or better compared to a single task adapter. We further find this robustness to be task dependent. Even AF with AdapterDrop shows a steep fall in performance on RTE and CoLA, while being relatively stable on MRPC and STS-B, even with most layers omitted.

G Detailed Efficiency Measurements

In this section, we present detailed results of our efficiency measurements for V100 and TitanX GPUs.

G.1 Adapters

We present the efficiency results for adapters and fully fine-tuned models in Figure 9, where we plot the required time (absolute numbers) during train-

ing and inference. The relative speed of adapters compared to fully fine-tuned models is given in Table 7.

G.2 AdapterDrop

Multi-task inference. In Figure 10, we plot the speed of adapters in a multi-task setting compared to fully fine-tuned models with sequential processing of inputs. In Table 8, we present the relative speed of adapters in this setting and show the speedup gained with AdapterDrop for each dropped layer. The average speedup in Table 2 is calculated as the average speedup over the batch sizes 16, 32 and 64 in Table 8.

Training adapters with dropped layers. Table 9 shows the speedup of AdapterDrop when training a *single* adapter. The average speedup for training with AdapterDrop is 4.7% per layer for the V100 and 4.5% for the TitanX. This is the average result over batch sizes 16, 32, 64 and sequence length 64, 128, 256, and 256 (see Table 9).

G.3 AdapterFusion

We plot the speed of AdapterFusion with different numbers of included adapters in Figure 11. In Table 10, we present the relative speed of AdapterFusion compared to a fully-finetuned model and a model with one adapter. This also shows the computational overhead (slowdown) that results from adding more adapters to AdapterFusion.

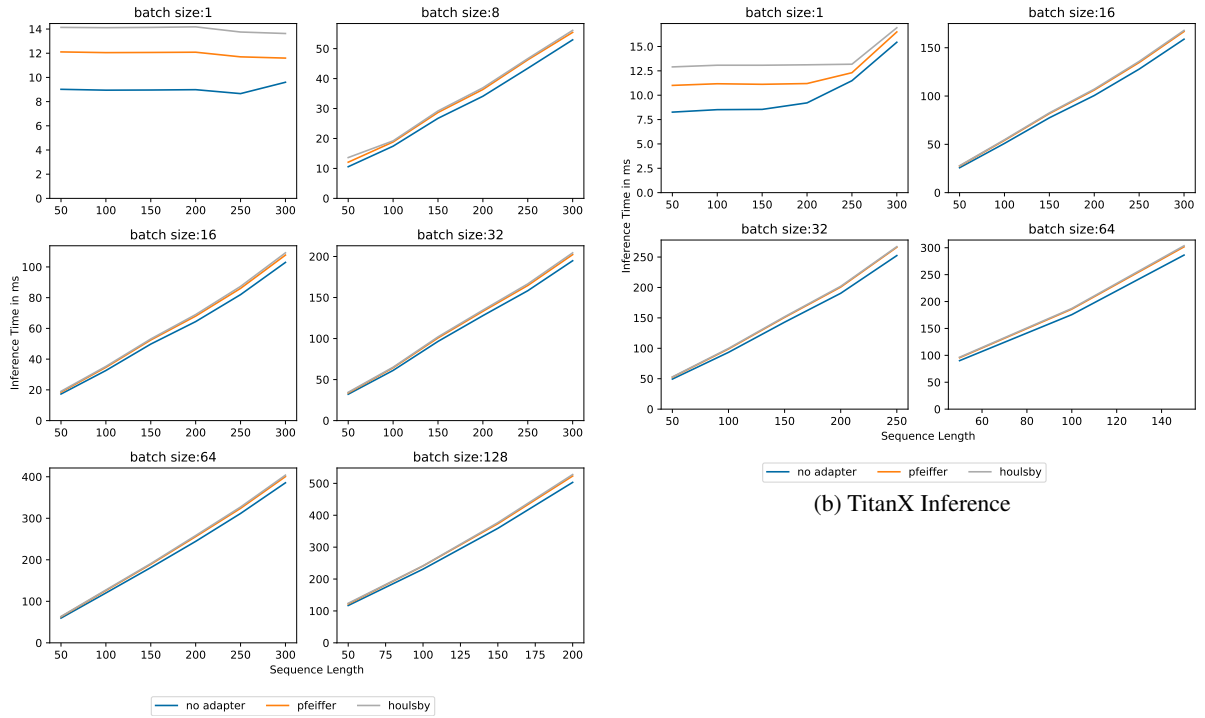
G.4 AdapterDrop for AdapterFusion

Table 11 shows the speedup gained with AdapterDrop for AdapterFusion during training and inference. Figure 12 shows the required time as a function of the dropped layers.

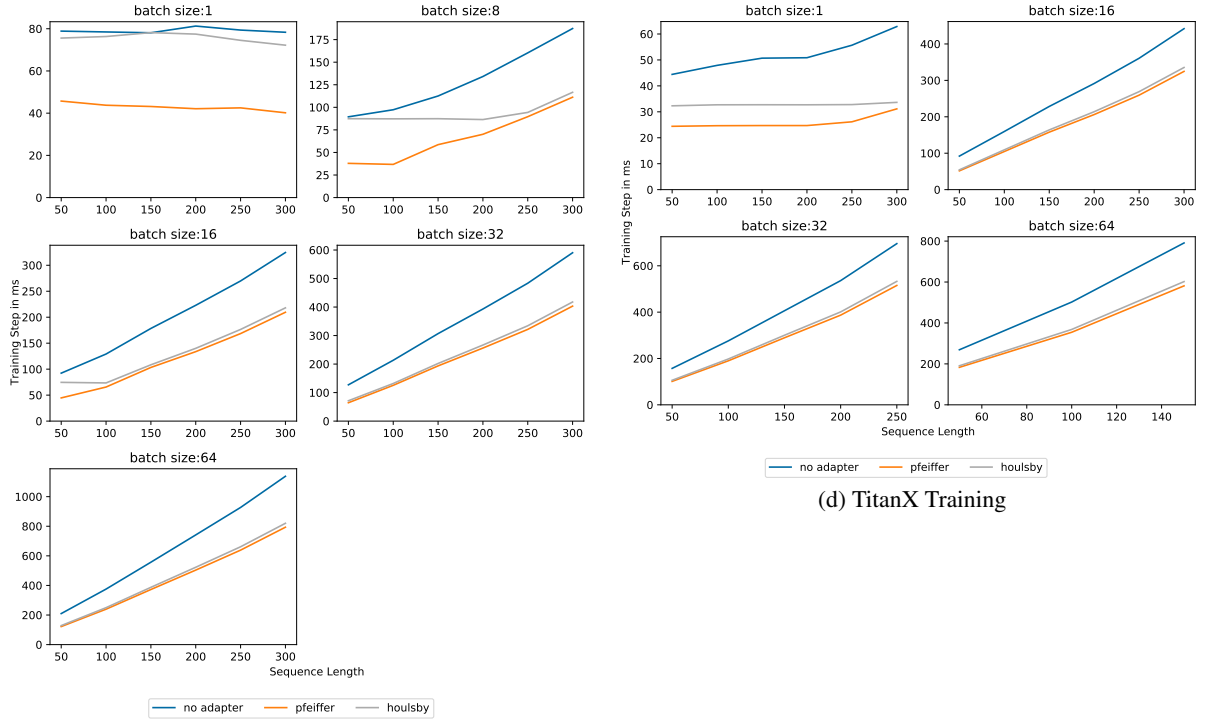
H Parallel Implementation of AdapterFusion

AdapterHub’s implementation of AdapterFusion passes through each task adapter sequentially. We hypothesized that a better efficiency can be achieved with parallel processing of adapters. We implement the parallel computation of the different adapters by reformulation the linear layers as two convolutions.

The first convolution is a convolution with a kernel size equal to the hidden dimension of the transformer and output channels equal to the number of adapters times the downprojection dimension of the



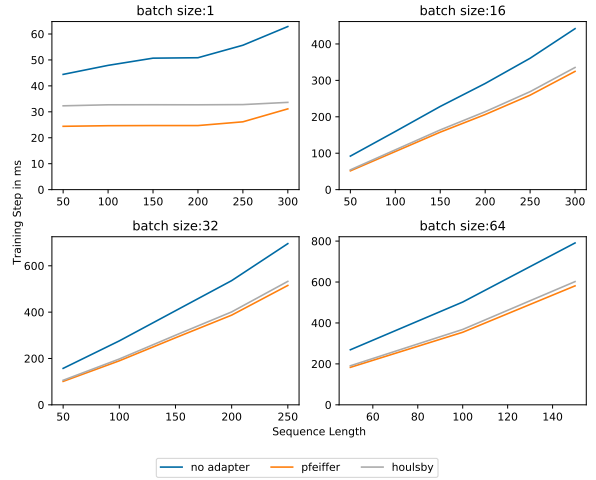
(a) V100 Inference



(b) TitanX Inference



(c) V100 Training



(d) TitanX Training

Figure 9: The absolute time for each **inference** or **training step**. We compare a **transformer model without adapters** and an **adapter model with Pfeiffer or Houslsby architectures**. We note that for small inputs, i.e., batch size 1 or 8, the time does not increase with the sequence length because the GPU is not working at capacity. Figure (b) with batch size 1 shows the transition from working under and working at capacity.

Sequence Len.	Batch Size	V100				TitanX			
		Training		Inference		Training		Inference	
		Houlsby	Pfeiffer	Houlsby	Pfeiffer	Houlsby	Pfeiffer	Houlsby	Pfeiffer
64	16	0.98	1.70	0.92	0.94	1.61	1.69	0.93	0.94
64	32	1.70	1.81	0.94	0.95	1.48	1.55	0.93	0.94
64	64	1.46	1.54	0.94	0.95	1.40	1.46	0.94	0.94
64	128	1.48	1.55	0.95	0.96	1.37	1.42	0.94	0.94
128	16	1.48	1.57	0.94	0.95	1.45	1.52	0.93	0.94
128	32	1.53	1.60	0.94	0.95	1.38	1.44	0.94	0.95
128	64	1.47	1.53	0.95	0.96	1.35	1.40	0.94	0.95
128	128	1.42	1.48	0.95	0.96	-	-	-	-
256	16	1.42	1.49	0.94	0.95	1.34	1.38	0.94	0.95
256	32	1.40	1.46	0.95	0.96	1.31	1.36	0.94	0.96
256	64	1.40	1.45	0.95	0.96	-	-	-	-
256	128	-	-	-	-	-	-	-	-
512	16	1.36	1.41	0.96	0.96	-	-	-	-
512	32	1.33	1.37	0.96	0.96	-	-	-	-
512	64	-	-	-	-	-	-	-	-
512	128	-	-	-	-	-	-	-	-

Table 7: Relative speed of **adapters** compared to fully fine-tuned models. Missing values are due to insufficient GPU memory.

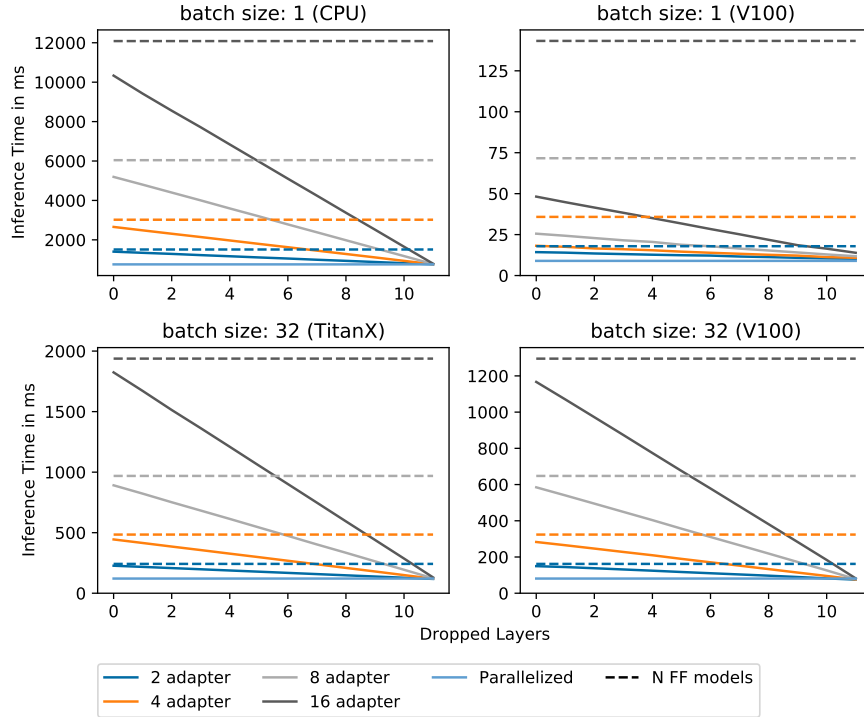
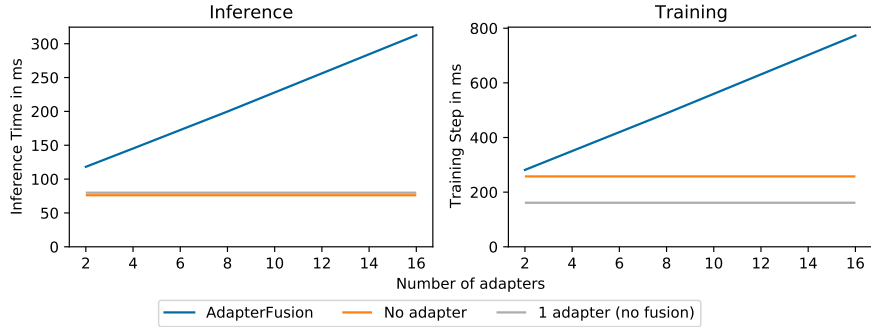
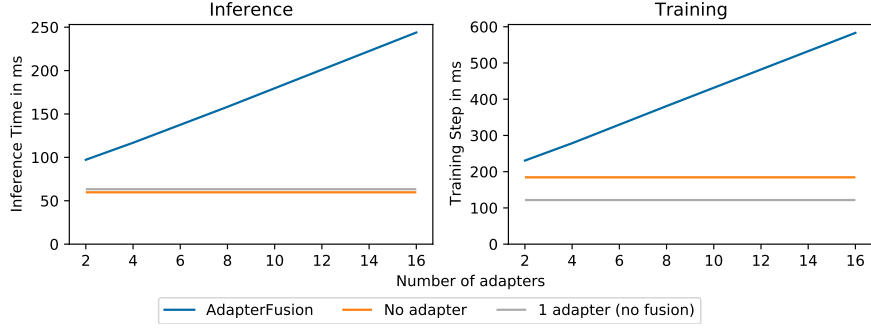


Figure 10: The absolute time required for performing inference for **multiple tasks** on the same input. The measurements are conducted with a sequence length of 128. **N FF models** denotes N fully fine-tuned models, executed sequentially. **Parallelized** denotes the time required by N fully fine-tuned models running fully parallelized. Batch size 1 on the V100 is an outlier compared to the other results with a smaller speedup for each dropped layer but a higher relative speed compared to the fine-tuned models due to the small input size.

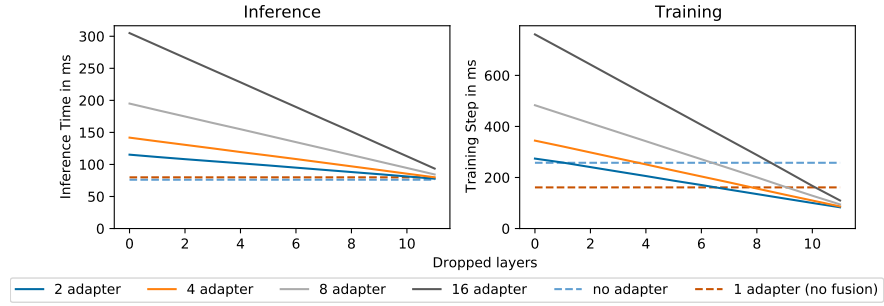


(a) V100

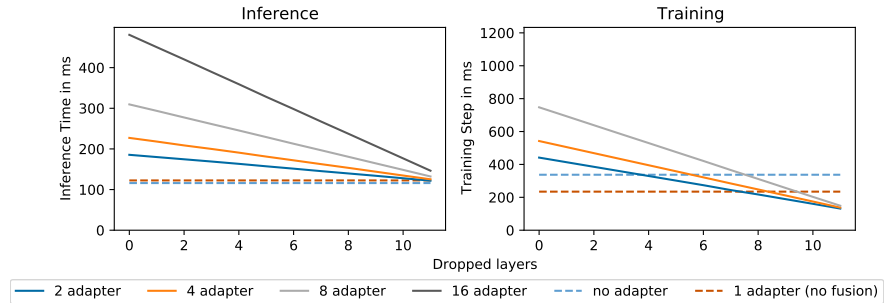


(b) TitanX

Figure 11: Absolute time measurements for **AdapterFusion** at **inference** (left) and **training** (right) as a function of the number of adapters. The measurements were conducted with a batch size of 32 (V100) and 16 (TitanX), and a sequence length of 128.



(a) V100



(b) TitanX

Figure 12: Absolute time measurements for **AdapterFusion with AdapterDrop** at **inference** (left) and **training** (right) as a function of the number of dropped layers. The measurements were conducted with a batch size of 32 and a sequence length of 128. We additionally plot the time of an adapter (without AdapterDrop) and a model without adapters to provide a more thorough comparison.

Device	Batch Size	Adapters	Inference	Speedup
V100	1	2	1.25	2.6%
	1	4	1.97	3.7%
	1	8	2.80	4.9%
	1	16	2.97	6.5%
	16	2	1.13	4.1%
	16	4	1.14	6.5%
	16	8	1.20	7.7%
	16	16	1.16	8.4%
	32	2	1.08	4.5%
	32	4	1.14	6.6%
	32	8	1.11	7.9%
	32	16	1.11	8.5%
	64	2	1.08	4.3%
	64	4	1.05	6.7%
	64	8	1.06	7.9%
	64	16	1.06	8.4%
TitanX	32	2	1.07	4.4%
	32	4	1.09	6.6%
	32	8	1.09	7.8%
	32	16	1.06	8.4%
CPU	1	2	0.98	4.2%
	1	4	1.03	6.5%
	1	8	1.05	7.7%
	1	16	1.06	8.4%

Table 8: The relative inference speed of simultaneous processing of multiple tasks with adapters compared to sequential processing of tasks with fully fine-tuned models. Gray columns show the speedup of **Adapter-Drop** for *every* additional dropped layer. All measurements use a sequence length of 128. Batch size 1 for the V100 is an outlier in both speedup and relative speed compared to the other results due to the small input size (compare with Figure 10).

Batch Size	Seq. Len	Speedup	
		V100	TitanX
16	64	4.6%	4.4%
16	128	4.6%	4.6%
16	256	4.8%	4.6%
16	512	4.7%	-
32	64	4.6%	4.5%
32	128	4.7%	4.5%
32	256	4.6%	4.7%
32	512	4.8%	-
64	64	4.7%	4.5%
64	128	4.6%	4.5%
64	256	4.7%	-
64	512	-	-

Table 9: Speedup for each dropped layer during **training with AdapterDrop** on the V100 and TitanX.

adapters. The second convolution is a grouped convolution¹⁸ which processes the channels in blocks the size of the downprojection dimension. It outputs channels equal to the number of adapters times the hidden dimension.

We show in Figure 13 and in Table 12 that the iterative implementation is *faster* than the parallel implementation for larger input sizes (e.g., batch sizes greater than). This indicates that once the input can no longer be processed entirely in parallel on the GPU (due to limited CUDA cores) the iterative implementation seems to be more efficient.

¹⁸Using the 'groups' parameter in Pytorch (<https://pytorch.org/docs/stable/generated/torch.nn.Conv1d.html#torch.nn.Conv1d>)

Seq. Len	Batch Size	V100						TitanX					
		vs. FF		vs. Adap.		Slowdown		vs. FF		vs. Adap.		Slowdown	
		Tr.	Inf.	Tr.	Inf.	Tr.	Inf.	Tr.	Inf.	Tr.	Inf.	Tr.	Inf.
64	16	0.77	0.62	0.45	0.66	8.2%	10.6%	0.88	0.62	0.52	0.66	10.3%	10.2%
64	32	1.03	0.64	0.57	0.68	12.0%	11.1%	0.80	0.61	0.52	0.64	11.2%	11.0%
64	64	0.87	0.64	0.57	0.67	12.6%	12.0%	0.76	0.61	0.52	0.65	11.6%	11.4%
128	16	0.91	0.65	0.58	0.69	12.0%	11.0%	0.80	0.61	0.53	0.65	10.9%	10.8%
128	32	0.92	0.64	0.57	0.68	12.5%	11.8%	0.76	0.62	0.53	0.66	11.4%	11.1%
128	64	0.87	0.65	0.57	0.68	12.5%	11.6%	-	-	-	-	-	-
256	16	0.88	0.66	0.59	0.69	12.1%	11.3%	0.77	0.65	0.56	0.68	10.8%	10.4%
256	32	0.86	0.68	0.59	0.70	11.9%	11.3%	-	-	-	-	-	-
256	64	-	-	-	-	-	-	-	-	-	-	-	-
512	16	0.87	0.69	0.62	0.72	11.2%	10.1%	-	-	-	-	-	-
512	32	-	-	-	-	-	-	-	-	-	-	-	-
512	64	-	-	-	-	-	-	-	-	-	-	-	-

Table 10: Relative speed of **AdapterFusion** for different sequence lengths and batch sizes. We compute the training (**Tr.**) speed and inference (**Inf.**) speed with two adapters in AdapterFusion. We compare this to: **FF**, a fully fine-tuned model; **Adap**, an adapter model (Pfeiffer architecture). The **slowdown** denotes the computational overhead of *each additional adapter* composed in AdapterFusion (calculated as the average slowdown for adding one adapter to AF consisting of 2–16 adapters). Missing values are due to insufficient GPU memory.

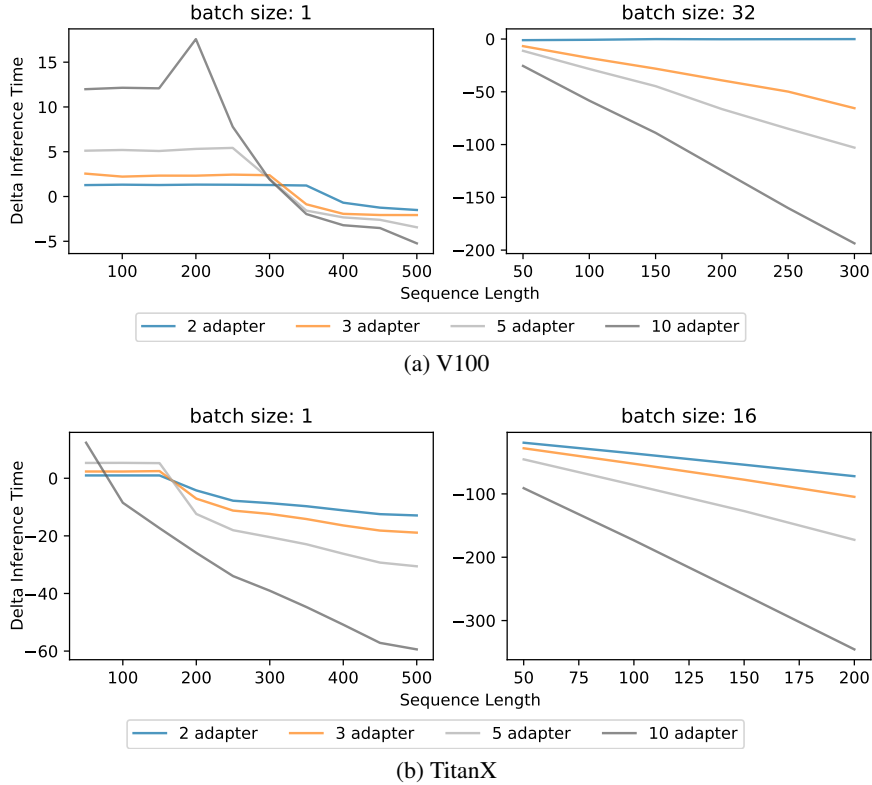


Figure 13: The difference in inference time between **iterative** and **parallel** implementations of **AdapterFusion**. Negative values indicate that the iterative implementation is faster. We calculate the difference as $t_i - t_p$, where t_i, t_p are the times for iterative and parallel implementation, respectively. In Figure (a), the parallel implementation is faster if the input is sufficiently small as the GPU is not working at capacity and is able to use the parallel implementation.

Speedup (per dropped layer)				
Adapters	Inference		Training	
	V100	TitanX	V100	TitanX
2	3.0%	3.1%	6.3%	6.4%
4	4.0%	4.1%	6.8%	6.8%
8	5.2%	5.2%	7.3%	7.3%
16	6.3%	6.3%	7.8%	-

Table 11: The speedup for each dropped layer for **AdapterFusion** during training and inference. Measurements were conducted with a batch size of 32 and sequence length of 128. Missing values are due to insufficient GPU memory.

Adapters	Seq. Len	Batch Size	Rel. Speed	
			V100	TitanX
2	100	1	0.93	0.94
3	100	1	0.89	0.88
5	100	1	0.77	0.76
10	100	1	0.60	1.29
2	100	16	1.02	1.44
3	100	16	1.12	1.58
5	100	16	1.17	1.80
10	100	16	1.27	2.14
2	100	32	1.01	1.48
3	100	32	1.17	1.62
5	100	32	1.23	1.85
10	100	32	1.32	2.24
2	200	1	0.93	1.24
3	200	1	0.88	1.37
5	200	1	0.77	1.55
10	200	1	0.52	1.87
2	200	16	1.01	1.46
3	200	16	1.17	1.59
5	200	16	1.23	1.82
10	200	16	1.32	2.21
2	200	32	1.00	1.11
3	200	32	1.18	1.17
5	200	32	1.26	-
10	200	32	1.34	-
2	300	1	0.93	1.37
3	300	1	0.88	1.50
5	300	1	0.91	1.70
10	300	1	0.94	2.03
2	300	16	1.00	1.48
3	300	16	1.16	1.63
5	300	16	1.22	1.88
10	300	16	1.32	-
2	300	32	1.00	-
3	300	32	1.20	-
5	300	32	1.27	-
10	300	32	1.36	-
2	400	1	1.04	1.39
3	400	1	1.09	1.51
5	400	1	1.10	1.74
10	400	1	1.10	2.08
2	400	16	1.00	-
3	400	16	1.18	-
5	400	16	1.25	-
10	400	16	1.34	-
2	400	32	1.00	-
3	400	32	1.20	-
5	400	32	1.27	-
10	400	32	-	-

Table 12: Relative speed of **AdapterFusion** with the **iterative** implementation versus the **parallel** implementation with different batch sizes, sequence lengths and numbers of adapters for the V100 and TitanX. The parallel implementation is faster if the input is sufficiently small (batch size 1 or 2 adapters) as the GPU is not working at capacity and is able to use the parallel implementation.