arXiv:1902.00751v2 [cs.LG] 13 Jun 2019

Parameter-Efficient Transfer Learning for NLP

Neil Houlsby [1] **Andrei Giurgiu** [1 *] **Stanisław Jastrzębski** [2 *] **Bruna Morrone** [1] **Quentin de Laroussilhe** [1]
Andrea Gesmundo [1] **Mona Attariyan** [1] **Sylvain Gelly** [1]

Abstract

Fine-tuning large pre-trained models is an effective transfer mechanism in NLP. However, in the presence of many downstream tasks, fine-tuning is parameter inefficient: an entire new model is required for every task. As an alternative, we propose transfer with adapter modules. Adapter modules yield a compact and extensible model; they add only a few trainable parameters per task, and new tasks can be added without revisiting previous ones. The parameters of the original network remain fixed, yielding a high degree of parameter sharing. To demonstrate adapter's effectiveness, we transfer the recently proposed BERT Transformer model to 26 diverse text classification tasks, including the GLUE benchmark. Adapters attain near state-of-the-art performance, whilst adding only a few parameters per task. On GLUE, we attain within 0.4% of the performance of full fine-tuning, adding only 3.6% parameters per task. By contrast, fine-tuning trains 100% of the parameters per task.[1]

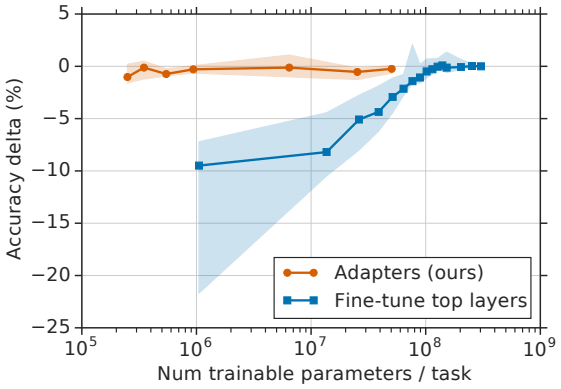

Figure 1. Trade-off between accuracy and number of trained task-specific parameters, for adapter tuning and fine-tuning. The y-axis is normalized by the performance of full fine-tuning, details in Section 3. The curves show the 20th, 50th, and 80th performance percentiles across nine tasks from the GLUE benchmark. Adapter-based tuning attains a similar performance to full fine-tuning with two orders of magnitude fewer trained parameters.

1. Introduction

Transfer from pre-trained models yields strong performance on many NLP tasks (Dai & Le, 2015; Howard & Ruder, 2018; Radford et al., 2018). BERT, a Transformer network trained on large text corpora with an unsupervised loss, attained state-of-the-art performance on text classification and extractive question answering (Devlin et al., 2018).

In this paper we address the online setting, where tasks arrive in a stream. The goal is to build a system that performs well on all of them, but without training an entire new model for every new task. A high degree of sharing between tasks is particularly useful for applications such as cloud services, where models need to be trained to solve many tasks that arrive from customers in sequence. For this, we propose a transfer learning strategy that yields *compact* and *extensible* downstream models. Compact models are those that solve many tasks using a small number of additional parameters per task. Extensible models can be trained incrementally to solve new tasks, without forgetting previous ones. Our method yields a such models without sacrificing performance.

The two most common transfer learning techniques in NLP are feature-based transfer and fine-tuning. Instead, we present an alternative transfer method based on adapter modules (Rebuffi et al., 2017). Features-based transfer involves pre-training real-valued embeddings vectors. These embeddings may be at the word (Mikolov et al., 2013), sentence (Cer et al., 2019), or paragraph level (Le & Mikolov, 2014). The embeddings are then fed to custom downstream models. Fine-tuning involves copying the weights from a pre-trained network and tuning them on the downstream task. Recent work shows that fine-tuning often enjoys better

[*]Equal contribution [1]Google Research [2]Jagiellonian University. Correspondence to: Neil Houlsby <neilhoulsby@google.com>.

Proceedings of the 36th International Conference on Machine Learning, Long Beach, California, PMLR 97, 2019. Copyright 2019 by the author(s).

[1]Code at https://github.com/google-research/adapter-bert

performance than feature-based transfer (Howard & Ruder, 2018).

Both feature-based transfer and fine-tuning require a new set of weights for each task. Fine-tuning is more parameter efficient if the lower layers of a network are shared between tasks. However, our proposed adapter tuning method is even more parameter efficient. Figure 1 demonstrates this trade-off. The x-axis shows the number of parameters trained per task; this corresponds to the marginal increase in the model size required to solve each additional task. Adapter-based tuning requires training two orders of magnitude fewer parameters to fine-tuning, while attaining similar performance.

Adapters are new modules added between layers of a pre-trained network. Adapter-based tuning differs from feature-based transfer and fine-tuning in the following way. Consider a function (neural network) with parameters $\boldsymbol{w}$: $\phi_{\boldsymbol{w}}(\boldsymbol{x})$. Feature-based transfer composes $\phi_{\boldsymbol{w}}$ with a new function, $\chi_{\boldsymbol{v}}$, to yield $\chi_{\boldsymbol{v}}(\phi_{\boldsymbol{w}}(\boldsymbol{x}))$. Only the new, task-specific, parameters, $\boldsymbol{v}$, are then trained. Fine-tuning involves adjusting the original parameters, $\boldsymbol{w}$, for each new task, limiting compactness. For adapter tuning, a new function, $\psi_{\boldsymbol{w},\boldsymbol{v}}(\boldsymbol{x})$, is defined, where parameters $\boldsymbol{w}$ are copied over from pre-training. The initial parameters $\boldsymbol{v}_0$ are set such that the new function resembles the original: $\psi_{\boldsymbol{w},\boldsymbol{v}_0}(\boldsymbol{x}) \approx \phi_{\boldsymbol{w}}(\boldsymbol{x})$. During training, only $\boldsymbol{v}$ are tuned. For deep networks, defining $\psi_{\boldsymbol{w},\boldsymbol{v}}$ typically involves adding new layers to the original network, $\phi_{\boldsymbol{w}}$. If one chooses $|\boldsymbol{v}| \ll |\boldsymbol{w}|$, the resulting model requires $\sim |\boldsymbol{w}|$ parameters for many tasks. Since $\boldsymbol{w}$ is fixed, the model can be extended to new tasks without affecting previous ones.

Adapter-based tuning relates to *multi-task* and *continual* learning. Multi-task learning also results in compact models. However, multi-task learning requires simultaneous access to all tasks, which adapter-based tuning does not require. Continual learning systems aim to learn from an endless stream of tasks. This paradigm is challenging because networks forget previous tasks after re-training (McCloskey & Cohen, 1989; French, 1999). Adapters differ in that the tasks do not interact and the shared parameters are frozen. This means that the model has perfect memory of previous tasks using a small number of task-specific parameters.

We demonstrate on a large and diverse set of text classification tasks that adapters yield parameter-efficient tuning for NLP. The key innovation is to design an effective adapter module and its integration with the base model. We propose a simple yet effective, bottleneck architecture. On the GLUE benchmark, our strategy almost matches the performance of the fully fine-tuned BERT, but uses only 3% task-specific parameters, while fine-tuning uses 100% task-specific parameters. We observe similar results on a further 17 public text datasets, and SQuAD extractive question answering. In summary, adapter-based tuning yields a single, extensible, model that attains near state-of-the-art performance in text classification.

2. Adapter tuning for NLP

We present a strategy for tuning a large text model on several downstream tasks. Our strategy has three key properties: (i) it attains good performance, (ii) it permits training on tasks sequentially, that is, it does not require simultaneous access to all datasets, and (iii) it adds only a small number of additional parameters per task. These properties are especially useful in the context of cloud services, where many models need to be trained on a series of downstream tasks, so a high degree of sharing is desirable.

To achieve these properties, we propose a new bottleneck adapter module. Tuning with adapter modules involves adding a small number of new parameters to a model, which are trained on the downstream task (Rebuffi et al., 2017). When performing vanilla fine-tuning of deep networks, a modification is made to the top layer of the network. This is required because the label spaces and losses for the upstream and downstream tasks differ. Adapter modules perform more general architectural modifications to re-purpose a pre-trained network for a downstream task. In particular, the adapter tuning strategy involves injecting new layers into the original network. The weights of the original network are untouched, whilst the new adapter layers are initialized at random. In standard fine-tuning, the new top-layer and the original weights are co-trained. In contrast, in adapter-tuning, the parameters of the original network are frozen and therefore may be shared by many tasks.

Adapter modules have two main features: a small number of parameters, and a near-identity initialization. The adapter modules need to be small compared to the layers of the original network. This means that the total model size grows relatively slowly when more tasks are added. A near-identity initialization is required for stable training of the adapted model; we investigate this empirically in Section 3.6. By initializing the adapters to a near-identity function, original network is unaffected when training starts. During training, the adapters may then be activated to change the distribution of activations throughout the network. The adapter modules may also be ignored if not required; in Section 3.6 we observe that some adapters have more influence on the network than others. We also observe that if the initialization deviates too far from the identity function, the model may fail to train.

2.1. Instantiation for Transformer Networks

We instantiate adapter-based tuning for text Transformers. These models attain state-of-the-art performance in many NLP tasks, including translation, extractive QA, and text

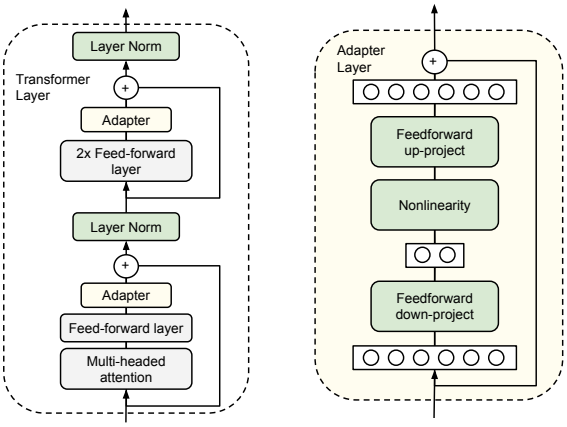

Figure 2. Architecture of the adapter module and its integration with the Transformer. **Left:** We add the adapter module twice to each Transformer layer: after the projection following multi-headed attention and after the two feed-forward layers. **Right:** The adapter consists of a bottleneck which contains few parameters relative to the attention and feedforward layers in the original model. The adapter also contains a skip-connection. During adapter tuning, the green layers are trained on the downstream data, this includes the adapter, the layer normalization parameters, and the final classification layer (not shown in the figure).

classification problems (Vaswani et al., 2017; Radford et al., 2018; Devlin et al., 2018). We consider the standard Transformer architecture, as proposed in Vaswani et al. (2017).

Adapter modules present many architectural choices. We provide a simple design that attains good performance. We experimented with a number of more complex designs, see Section 3.6, but we found the following strategy performed as well as any other that we tested, across many datasets.

Figure 2 shows our adapter architecture, and its application it to the Transformer. Each layer of the Transformer contains two primary sub-layers: an attention layer and a feedforward layer. Both layers are followed immediately by a projection that maps the features size back to the size of layer's input. A skip-connection is applied across each of the sub-layers. The output of each sub-layer is fed into layer normalization. We insert two serial adapters after each of these sub-layers. The adapter is always applied directly to the output of the sub-layer, after the projection back to the input size, but before adding the skip connection back. The output of the adapter is then passed directly into the following layer normalization.

To limit the number of parameters, we propose a bottleneck architecture. The adapters first project the original d-dimensional features into a smaller dimension, m, apply a nonlinearity, then project back to d dimensions. The total number of parameters added per layer, including biases, is $2md + d + m$. By setting $m \ll d$, we limit the number of parameters added per task; in practice, we use around $0.5 - 8\%$ of the parameters of the original model. The bottleneck dimension, m, provides a simple means to trade-off performance with parameter efficiency. The adapter module itself has a skip-connection internally. With the skip-connection, if the parameters of the projection layers are initialized to near-zero, the module is initialized to an approximate identity function.

Alongside the layers in the adapter module, we also train new layer normalization parameters per task. This technique, similar to conditional batch normalization (De Vries et al., 2017), FiLM (Perez et al., 2018), and self-modulation (Chen et al., 2019), also yields parameter-efficient adaptation of a network; with only $2d$ parameters per layer. However, training the layer normalization parameters alone is insufficient for good performance, see Section 3.4.

3. Experiments

We show that adapters achieve parameter efficient transfer for text tasks. On the GLUE benchmark (Wang et al., 2018), adapter tuning is within 0.4% of full fine-tuning of BERT, but it adds only 3% of the number of parameters trained by fine-tuning. We confirm this result on a further 17 public classification tasks and SQuAD question answering. Analysis shows that adapter-based tuning automatically focuses on the higher layers of the network.

3.1. Experimental Settings

We use the public, pre-trained BERT Transformer network as our base model. To perform classification with BERT, we follow the approach in Devlin et al. (2018). The first token in each sequence is a special "classification token". We attach a linear layer to the embedding of this token to predict the class label.

Our training procedure also follows Devlin et al. (2018). We optimize using Adam (Kingma & Ba, 2014), whose learning rate is increased linearly over the first 10% of the steps, and then decayed linearly to zero. All runs are trained on 4 Google Cloud TPUs with a batch size of 32. For each dataset and algorithm, we run a hyperparameter sweep and select the best model according to accuracy on the validation set. For the GLUE tasks, we report the test metrics provided by the submission website[2]. For the other classification tasks we report test-set accuracy.

[2] https://gluebenchmark.com/

We compare to fine-tuning, the current standard for transfer of large pre-trained models, and the strategy successfully used by BERT. For N tasks, full fine-tuning requires $N\times$ the number of parameters of the pre-trained model. Our goal is to attain performance equal to fine-tuning, but with fewer total parameters, ideally near to $1\times$.

3.2. GLUE benchmark

We first evaluate on GLUE.[3] For these datasets, we transfer from the pre-trained $\text{BERT}_{\text{LARGE}}$ model, which contains 24 layers, and a total of 330M parameters, see Devlin et al. (2018) for details. We perform a small hyperparameter sweep for adapter tuning: We sweep learning rates in $\{3 \cdot 10^{-5}, 3 \cdot 10^{-4}, 3 \cdot 10^{-3}\}$, and number of epochs in $\{3, 20\}$. We test both using a fixed adapter size (number of units in the bottleneck), and selecting the best size per task from $\{8, 64, 256\}$. The adapter size is the only adapter-specific hyperparameter that we tune. Finally, due to training instability, we re-run 5 times with different random seeds and select the best model on the validation set.

Table 1 summarizes the results. Adapters achieve a mean GLUE score of 80.0, compared to 80.4 achieved by full fine-tuning. The optimal adapter size varies per dataset. For example, 256 is chosen for MNLI, whereas for the smallest dataset, RTE, 8 is chosen. Restricting always to size 64, leads to a small decrease in average accuracy to 79.6. To solve all of the datasets in Table 1, fine-tuning requires $9\times$ the total number of BERT parameters.[4] In contrast, adapters require only $1.3\times$ parameters.

3.3. Additional Classification Tasks

To further validate that adapters yields compact, performant, models, we test on additional, publicly available, text classification tasks. This suite contains a diverse set of tasks: The number of training examples ranges from 900 to 330k, the number of classes ranges from 2 to 157, and the average text length ranging from 57 to 1.9k characters. Statistics and references for all of the datasets are in the appendix.

For these datasets, we use a batch size of 32. The datasets are diverse, so we sweep a wide range of learning rates: $\{1 \cdot 10^{-5}, 3 \cdot 10^{-5}, 1 \cdot 10^{-4}, 3 \cdot 10^{-3}\}$. Due to the large number of datasets, we select the number of training epochs from the set $\{20, 50, 100\}$ manually, from inspection of the validation set learning curves. We select the optimal values for both fine-tuning and adapters; the exact values are in the appendix.

[3] We omit WNLI as in Devlin et al. (2018) because the no current algorithm beats the baseline of predicting the majority class.

[4] We treat MNLI_{m} and MNLI_{mm} as separate tasks with individually tuned hyperparameters. However, they could be combined into one model, leaving $8\times$ overall.

We test adapters sizes in $\{2, 4, 8, 16, 32, 64\}$. Since some of the datasets are small, fine-tuning the entire network may be sub-optimal. Therefore, we run an additional baseline: variable fine-tuning. For this, we fine-tune only the top n layers, and freeze the remainder. We sweep $n \in \{1, 2, 3, 5, 7, 9, 11, 12\}$. In these experiments, we use the $\text{BERT}_{\text{BASE}}$ model with 12 layers, therefore, variable fine-tuning subsumes full fine-tuning when $n = 12$.

Unlike the GLUE tasks, there is no comprehensive set of state-of-the-art numbers for this suite of tasks. Therefore, to confirm that our BERT-based models are competitive, we collect our own benchmark performances. For this, we run a large-scale hyperparameter search over standard network topologies. Specifically, we run the single-task Neural AutoML algorithm, similar to Zoph & Le (2017); Wong et al. (2018). This algorithm searches over a space of feedforward and convolutional networks, stacked on pre-trained text embeddings modules publicly available via TensorFlow Hub[5]. The embeddings coming from the TensorFlow Hub modules may be frozen or fine-tuned. The full search space is described in the appendix. For each task, we run AutoML for one week on CPUs, using 30 machines. In this time the algorithm explores over 10k models on average per task. We select the best final model for each task according to validation set accuracy.

The results for the AutoML benchmark ("no BERT baseline"), fine-tuning, variable fine-tuning, and adapter-tuning are reported in Table 2. The AutoML baseline demonstrates that the BERT models are competitive. This baseline explores thousands of models, yet the BERT models perform better on average. We see similar pattern of results to GLUE. The performance of adapter-tuning is close to full fine-tuning (0.4% behind). Fine-tuning requires $17\times$ the number of parameters to $\text{BERT}_{\text{BASE}}$ to solve all tasks. Variable fine-tuning performs slightly better than fine-tuning, whilst training fewer layers. The optimal setting of variable fine-tuning results in training 52% of the network on average per task, reducing the total to $9.9\times$ parameters. Adapters, however, offer a much more compact model. They introduce 1.14% new parameters per task, resulting in $1.19\times$ parameters for all 17 tasks.

3.4. Parameter/Performance trade-off

The adapter size controls the parameter efficiency, smaller adapters introduce fewer parameters, at a possible cost to performance. To explore this trade-off, we consider different adapter sizes, and compare to two baselines: (i) Fine-tuning of only the top k layers of $\text{BERT}_{\text{BASE}}$. (ii) Tuning only the layer normalization parameters. The learning rate is tuned using the range presented in Section 3.2.

[5] https://www.tensorflow.org/hub

	Total num params	Trained params / task	CoLA	SST	MRPC	STS-B	QQP	MNLI_m	MNLI_mm	QNLI	RTE	Total
BERT_LARGE	$9.0\times$	100%	60.5	94.9	89.3	87.6	72.1	86.7	85.9	91.1	70.1	80.4
Adapters (8-256)	$1.3\times$	3.6%	59.5	94.0	89.5	86.9	71.8	84.9	85.1	90.7	71.5	80.0
Adapters (64)	$1.2\times$	2.1%	56.9	94.2	89.6	87.3	71.8	85.3	84.6	91.4	68.8	79.6

Table 1. Results on GLUE test sets scored using the GLUE evaluation server. MRPC and QQP are evaluated using F1 score. STS-B is evaluated using Spearman's correlation coefficient. CoLA is evaluated using Matthew's Correlation. The other tasks are evaluated using accuracy. Adapter tuning achieves comparable overall score (80.0) to full fine-tuning (80.4) using $1.3\times$ parameters in total, compared to $9\times$. Fixing the adapter size to 64 leads to a slightly decreased overall score of 79.6 and slightly smaller model.

Dataset	No BERT baseline	BERT_BASE Fine-tune	BERT_BASE Variable FT	BERT_BASE Adapters
20 newsgroups	91.1	92.8 ± 0.1	92.8 ± 0.1	91.7 ± 0.2
Crowdflower airline	84.5	83.6 ± 0.3	84.0 ± 0.1	84.5 ± 0.2
Crowdflower corporate messaging	91.9	92.5 ± 0.5	92.4 ± 0.6	92.9 ± 0.3
Crowdflower disasters	84.9	85.3 ± 0.4	85.3 ± 0.4	84.1 ± 0.2
Crowdflower economic news relevance	81.1	82.1 ± 0.0	78.9 ± 2.8	82.5 ± 0.3
Crowdflower emotion	36.3	38.4 ± 0.1	37.6 ± 0.2	38.7 ± 0.1
Crowdflower global warming	82.7	84.2 ± 0.4	81.9 ± 0.2	82.7 ± 0.3
Crowdflower political audience	81.0	80.9 ± 0.3	80.7 ± 0.8	79.0 ± 0.5
Crowdflower political bias	76.8	75.2 ± 0.9	76.5 ± 0.4	75.9 ± 0.3
Crowdflower political message	43.8	38.9 ± 0.6	44.9 ± 0.6	44.1 ± 0.2
Crowdflower primary emotions	33.5	36.9 ± 1.6	38.2 ± 1.0	33.9 ± 1.4
Crowdflower progressive opinion	70.6	71.6 ± 0.5	75.9 ± 1.3	71.7 ± 1.1
Crowdflower progressive stance	54.3	63.8 ± 1.0	61.5 ± 1.3	60.6 ± 1.4
Crowdflower US economic performance	75.6	75.3 ± 0.1	76.5 ± 0.4	77.3 ± 0.1
Customer complaint database	54.5	55.9 ± 0.1	56.4 ± 0.1	55.4 ± 0.1
News aggregator dataset	95.2	96.3 ± 0.0	96.5 ± 0.0	96.2 ± 0.0
SMS spam collection	98.5	99.3 ± 0.2	99.3 ± 0.2	95.1 ± 2.2
Average	72.7	73.7	74.0	73.3
Total number of params	—	$17\times$	$9.9\times$	$1.19\times$
Trained params/task	—	100%	52.9%	1.14%

Table 2. Test accuracy for additional classification tasks. In these experiments we transfer from the BERT_BASE model. For each task and algorithm, the model with the best validation set accuracy is chosen. We report the mean test accuracy and s.e.m. across runs with different random seeds.

Figure 3 shows the parameter/performance trade-off aggregated over all classification tasks in each suite (GLUE and "additional"). On GLUE, performance decreases dramatically when fewer layers are fine-tuned. Some of the additional tasks benefit from training fewer layers, so performance of fine-tuning decays much less. In both cases, adapters yield good performance across a range of sizes two orders of magnitude fewer than fine-tuning.

Figure 4 shows more details for two GLUE tasks: MNLI_m and CoLA. Tuning the top layers trains more task-specific parameters for all $k > 2$. When fine-tuning using a comparable number of task-specific parameters, the performance decreases substantially compared to adapters. For instance, fine-tuning just the top layer yields approximately 9M trainable parameters and $77.8\% \pm 0.1\%$ validation accuracy on MNLI_m. In contrast, adapter tuning with size 64 yields approximately 2M trainable parameters and $83.7\% \pm 0.1\%$ validation accuracy. For comparison, full fine-tuning attains $84.4\% \pm 0.02\%$ on MNLI_m. We observe a similar trend on CoLA.

As a further comparison, we tune the parameters of layer normalization alone. These layers only contain point-wise additions and multiplications, so introduce very few trainable parameters: 40k for BERT_BASE. However this strategy performs poorly: performance decreases by approximately 3.5% on CoLA and 4% on MNLI.

To summarize, adapter tuning is highly parameter-efficient, and produces a compact model with a strong performance, comparable to full fine-tuning. Training adapters with sizes $0.5 - 5\%$ of the original model, performance is within 1% of the competitive published results on BERT_LARGE.

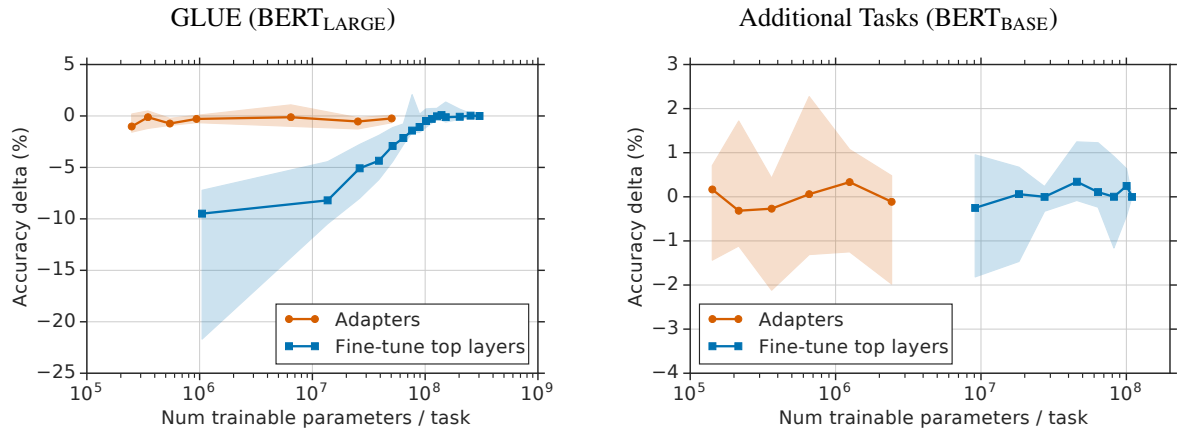

Figure 3. Accuracy versus the number of trained parameters, aggregated across tasks. We compare adapters of different sizes (orange) with fine-tuning the top n layers, for varying n (blue). The lines and shaded areas indicate the 20th, 50th, and 80th percentiles across tasks. For each task and algorithm, the best model is selected for each point along the curve. For GLUE, the validation set accuracy is reported. For the additional tasks, we report the test-set accuracies. To remove the intra-task variance in scores, we normalize the scores for each model and task by subtracting the performance of full fine-tuning on the corresponding task.

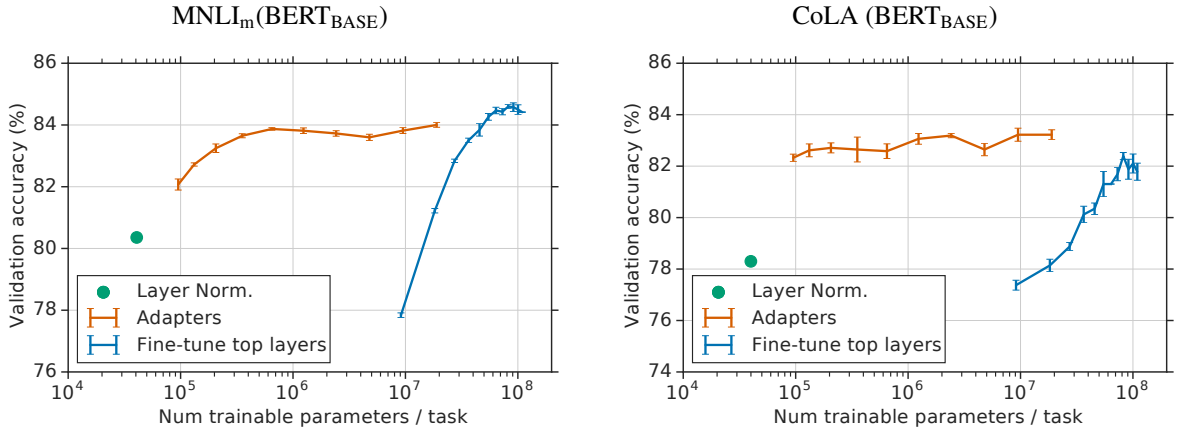

Figure 4. Validation set accuracy versus number of trained parameters for three methods: (i) Adapter tuning with an adapter sizes 2^n for $n = 0 \ldots 9$ (orange). (ii) Fine-tuning the top k layers for $k = 1 \ldots 12$ (blue). (iii) Tuning the layer normalization parameters only (green). Error bars indicate ± 1 s.e.m. across three random seeds.

3.5. SQuAD Extractive Question Answering

Finally, we confirm that adapters work on tasks other than classification by running on SQuAD v1.1 (Rajpurkar et al., 2018). Given a question and Wikipedia paragraph, this task requires selecting the answer span to the question from the paragraph. Figure 5 displays the parameter/performance trade-off of fine-tuning and adapters on the SQuAD validation set. For fine-tuning, we sweep the number of trained layers, learning rate in $\{3{\cdot}10^{-5}, 5{\cdot}10^{-5}, 1{\cdot}10^{-4}\}$, and number of epochs in $\{2, 3, 5\}$. For adapters, we sweep the adapter size, learning rate in $\{3 \cdot 10^{-5}, 1 \cdot 10^{-4}, 3 \cdot 10^{-4}, 1 \cdot 10^{-3}\}$, and number of epochs in $\{3, 10, 20\}$. As for classification, adapters attain performance comparable to full fine-tuning, while training many fewer parameters. Adapters of size 64 (2% parameters) attain a best F1 of 90.4%, while fine-tuning attains 90.7. SQuAD performs well even with very small adapters, those of size 2 (0.1% parameters) attain an F1 of 89.9.

3.6. Analysis and Discussion

We perform an ablation to determine which adapters are influential. For this, we remove some trained adapters and re-evaluate the model (without re-training) on the validation set. Figure 6 shows the change in performance when removing adapters from all continuous layer spans. The experiment is performed on BERT$_{\text{BASE}}$ with adapter size 64 on MNLI and CoLA.

First, we observe that removing any single layer's adapters has only a small impact on performance. The elements on

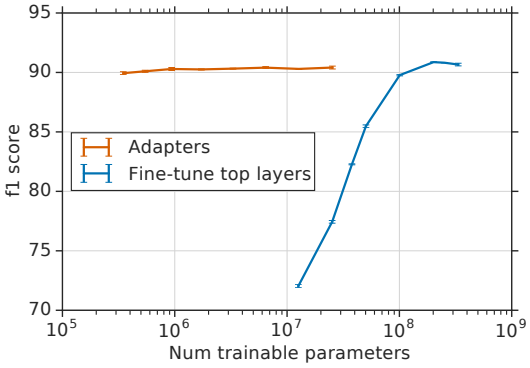

Figure 5. Validation accuracy versus the number of trained parameters for SQuAD v1.1. Error bars indicate the s.e.m. across three seeds, using the best hyperparameters.

the heatmaps' diagonals show the performances of removing adapters from single layers, where largest performance drop is 2%. In contrast, when all of the adapters are removed from the network, the performance drops substantially: to 37% on MNLI and 69% on CoLA – scores attained by predicting the majority class. This indicates that although each adapter has a small influence on the overall network, the overall effect is large.

Second, Figure 6 suggests that adapters on the lower layers have a smaller impact than the higher-layers. Removing the adapters from the layers $0 - 4$ on MNLI barely affects performance. This indicates that adapters perform well because they automatically prioritize higher layers. Indeed, focusing on the upper layers is a popular strategy in fine-tuning (Howard & Ruder, 2018). One intuition is that the lower layers extract lower-level features that are shared among tasks, while the higher layers build features that are unique to different tasks. This relates to our observation that for some tasks, fine-tuning only the top layers outperforms full fine-tuning, see Table 2.

Next, we investigate the robustness of the adapter modules to the number of neurons and initialization scale. In our main experiments the weights in the adapter module were drawn from a zero-mean Gaussian with standard deviation 10^{-2}, truncated to two standard deviations. To analyze the impact of the initialization scale on the performance, we test standard deviations in the interval $[10^{-7}, 1]$. Figure 6 summarizes the results. We observe that on both datasets, the performance of adapters is robust for standard deviations below 10^{-2}. However, when the initialization is too large, performance degrades, more substantially on CoLA.

To investigate robustness of adapters to the number of neurons, we re-examine the experimental data from Section 3.2. We find that the quality of the model across adapter sizes is stable, and a fixed adapter size across all the tasks could be used with small detriment to performance. For each adapter size we calculate the mean validation accuracy across the eight classification tasks by selecting the optimal learning rate and number of epochs[6]. For adapter sizes 8, 64, and 256, the mean validation accuracies are 86.2%, 85.8% and 85.7%, respectively. This message is further corroborated by Figures 4 and 5, which show a stable performance across a few orders of magnitude.

Finally, we tried a number of extensions to the adapter's architecture that did not yield a significant boost in performance. We document them here for completeness. We experimented with (i) adding a batch/layer normalization to the adapter, (ii) increasing the number of layers per adapter, (iii) different activation functions, such as tanh, (iv) inserting adapters only inside the attention layer, (v) adding adapters in parallel to the main layers, and possibly with a multiplicative interaction. In all cases we observed the resulting performance to be similar to the bottleneck proposed in Section 2.1. Therefore, due to its simplicity and strong performance, we recommend the original adapter architecture.

4. Related Work

Pre-trained text representations Pre-trained textual representations are widely used to improve performance on NLP tasks. These representations are trained on large corpora (usually unsupervised), and fed as features to downstream models. In deep networks, these features may also be fine-tuned on the downstream task. Brown clusters, trained on distributional information, are a classic example of pre-trained representations (Brown et al., 1992). Turian et al. (2010) show that pre-trained embeddings of words outperform those trained from scratch. Since deep-learning became popular, word embeddings have been widely used, and many training strategies have arisen (Mikolov et al., 2013; Pennington et al., 2014; Bojanowski et al., 2017). Embeddings of longer texts, sentences and paragraphs, have also been developed (Le & Mikolov, 2014; Kiros et al., 2015; Conneau et al., 2017; Cer et al., 2019).

To encode context in these representations, features are extracted from internal representations of sequence models, such as MT systems (McCann et al., 2017), and BiLSTM language models, as used in ELMo (Peters et al., 2018). As with adapters, ELMo exploits the layers other than the top layer of a pre-trained network. However, this strategy only *reads* from the inner layers. In contrast, adapters *write* to the inner layers, re-configuring the processing of features through the entire network.

Fine-tuning Fine-tuning an entire pre-trained model has become a popular alternative to features (Dai & Le, 2015;

[6] We treat here MNLI_m and MNLI_mm as separate tasks. For consistency, for all datasets we use accuracy metric and exclude the regression STS-B task.

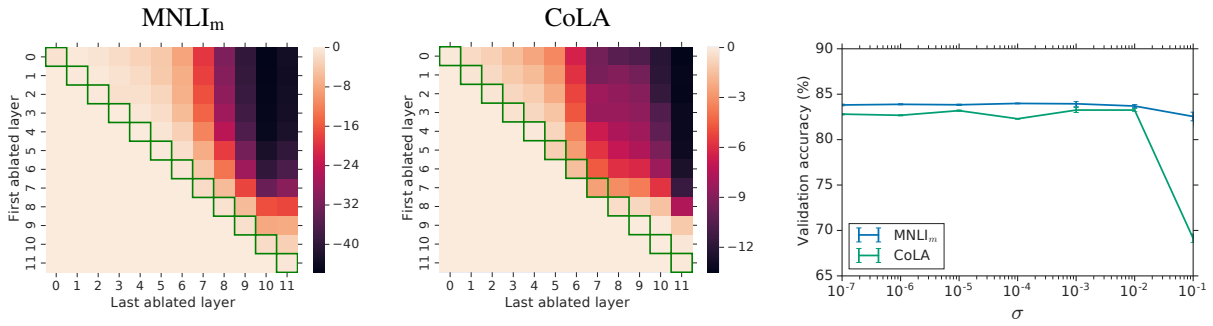

Figure 6. **Left, Center:** Ablation of trained adapters from continuous layer spans. The heatmap shows the relative decrease in validation accuracy to the fully trained adapted model. The y and x axes indicate the first and last layers ablated (inclusive), respectively. The diagonal cells, highlighted in green, indicate ablation of a single layer's adapters. The cell in the top-right indicates ablation of all adapters. Cells in the lower triangle are meaningless, and are set to 0%, the best possible relative performance. **Right:** Performance of BERT$_{\text{BASE}}$ using adapters with different initial weight magnitudes. The x-axis is the standard deviation of the initialization distribution.

Howard & Ruder, 2018; Radford et al., 2018) In NLP, the upstream model is usually a neural language model (Bengio et al., 2003). Recent state-of-the-art results on question answering (Rajpurkar et al., 2016) and text classification (Wang et al., 2018) have been attained by fine-tuning a Transformer network (Vaswani et al., 2017) with a Masked Language Model loss (Devlin et al., 2018). Performance aside, an advantage of fine-tuning is that it does not require task-specific model design, unlike representation-based transfer. However, vanilla fine-tuning does require a new set of network weights for every new task.

Multi-task Learning Multi-task learning (MTL) involves training on tasks simultaneously. Early work shows that sharing network parameters across tasks exploits task regularities, yielding improved performance (Caruana, 1997). The authors share weights in lower layers of a network, and use specialized higher layers. Many NLP systems have exploited MTL. Some examples include: text processing systems (part of speech, chunking, named entity recognition, etc.) (Collobert & Weston, 2008), multilingual models (Huang et al., 2013), semantic parsing (Peng et al., 2017), machine translation (Johnson et al., 2017), and question answering (Choi et al., 2017). MTL yields a single model to solve all problems. However, unlike our adapters, MTL requires simultaneous access to the tasks during training.

Continual Learning As an alternative to simultaneous training, continual, or lifelong, learning aims to learn from a sequence of tasks (Thrun, 1998). However, when re-trained, deep networks tend to forget how to perform previous tasks; a challenge termed catastrophic forgetting (McCloskey & Cohen, 1989; French, 1999). Techniques have been proposed to mitigate forgetting (Kirkpatrick et al., 2017; Zenke et al., 2017), however, unlike for adapters, the memory is imperfect. Progressive Networks avoid forgetting by instantiating a new network "column" for each task (Rusu et al., 2016). However, the number of parameters grows linearly with the number of tasks, since adapters are very small, our models scale much more favorably.

Transfer Learning in Vision Fine-tuning models pre-trained on ImageNet (Deng et al., 2009) is ubiquitous when building image recognition models (Yosinski et al., 2014; Huh et al., 2016). This technique attains state-of-the-art performance on many vision tasks, including classification (Kornblith et al., 2018), fine-grained classifcation (Hermans et al., 2017), segmentation (Long et al., 2015), and detection (Girshick et al., 2014). In vision, convolutional adapter modules have been studied (Rebuffi et al., 2017; 2018; Rosenfeld & Tsotsos, 2018). These works perform incremental learning in multiple domains by adding small convolutional layers to a ResNet (He et al., 2016) or VGG net (Simonyan & Zisserman, 2014). Adapter size is limited using 1×1 convolutions, whilst the original networks typically use 3×3. This yields 11% increase in overall model size per task. Since the kernel size cannot be further reduced other weight compression techniques must be used to attain further savings. Our bottleneck adapters can be much smaller, and still perform well.

Concurrent work explores similar ideas for BERT (Stickland & Murray, 2019). The authors introduce Projected Attention Layers (PALs), small layers with a similar role to our adapters. The main differences are i) Stickland & Murray (2019) use a different architecture, and ii) they perform multitask training, jointly fine-tuning BERT on all GLUE tasks. Sina Semnani (2019) perform an emprical comparison of our bottleneck Adpaters and PALs on SQuAD v2.0 (Rajpurkar et al., 2018).

ACKNOWLEDGMENTS

We would like to thank Andrey Khorlin, Lucas Beyer, Noé Lutz, and Jeremiah Harmsen for useful comments and discussions.

References

Almeida, T. A., Hidalgo, J. M. G., and Yamakami, A. Contributions to the Study of SMS Spam Filtering: New Collection and Results. In *Proceedings of the 11th ACM Symposium on Document Engineering*. ACM, 2011.

Bengio, Y., Ducharme, R., Vincent, P., and Janvin, C. A neural probabilistic language model. *Journal of Machine Learning Research*, 2003.

Bojanowski, P., Grave, E., Joulin, A., and Mikolov, T. Enriching word vectors with subword information. *ACL*, 2017.

Brown, P. F., deSouza, P. V., Mercer, R. L., Pietra, V. J. D., and Lai, J. C. Class-based n-gram models of natural language. *Computational Linguistics*, 1992.

Caruana, R. Multitask learning. *Machine Learning*, 1997.

Cer, D., Yang, Y., Kong, S.-y., Hua, N., Limtiaco, N., John, R. S., Constant, N., Guajardo-Cespedes, M., Yuan, S., Tar, C., et al. Universal sentence encoder. *arXiv preprint arXiv:1803.11175*, 2018.

Cer, D., Yang, Y., Kong, S.-y., Hua, N., Limtiaco, N., St. John, R., Constant, N., Guajardo-Cespedes, M., Yuan, S., Tar, C., Strope, B., and Kurzweil, R. Universal sentence encoder for english. In *EMNLP*, 2019.

Chen, T., Lucic, M., Houlsby, N., and Gelly, S. On self modulation for generative adversarial networks. *ICLR*, 2019.

Choi, E., Hewlett, D., Uszkoreit, J., Polosukhin, I., Lacoste, A., and Berant, J. Coarse-to-fine question answering for long documents. In *ACL*, 2017.

Collobert, R. and Weston, J. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *ICML*, 2008.

Conneau, A., Kiela, D., Schwenk, H., Barrault, L., and Bordes, A. Supervised learning of universal sentence representations from natural language inference data. In *EMNLP*, 2017.

Dai, A. M. and Le, Q. V. Semi-supervised sequence learning. In *NIPS*. 2015.

De Vries, H., Strub, F., Mary, J., Larochelle, H., Pietquin, O., and Courville, A. C. Modulating early visual processing by language. In *NIPS*, 2017.

Deng, J., Dong, W., Socher, R., jia Li, L., Li, K., and Fei-fei, L. Imagenet: A large-scale hierarchical image database. In *In CVPR*, 2009.

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.

French, R. M. Catastrophic forgetting in connectionist networks. *Trends in cognitive sciences*, 1999.

Girshick, R., Donahue, J., Darrell, T., and Malik, J. Rich feature hierarchies for accurate object detection and semantic segmentation. In *CVPR*, 2014.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *CVPR*, 2016.

Hermans, A., Beyer, L., and Leibe, B. In Defense of the Triplet Loss for Person Re-Identification. *arXiv preprint arXiv:1703.07737*, 2017.

Howard, J. and Ruder, S. Universal language model fine-tuning for text classification. In *ACL*, 2018.

Huang, J.-T., Li, J., Yu, D., Deng, L., and Gong, Y. Cross-language knowledge transfer using multilingual deep neural network with shared hidden layers. In *ICASSP*, 2013.

Huh, M., Agrawal, P., and Efros, A. A. What makes imagenet good for transfer learning? *arXiv preprint arXiv:1608.08614*, 2016.

Johnson, M., Schuster, M., Le, Q. V., Krikun, M., Wu, Y., Chen, Z., Thorat, N., Viégas, F., Wattenberg, M., Corrado, G., Hughes, M., and Dean, J. Google's multilingual neural machine translation system: Enabling zero-shot translation. *ACL*, 2017.

Kingma, D. and Ba, J. Adam: A method for stochastic optimization. *ICLR*, 2014.

Kirkpatrick, J., Pascanu, R., Rabinowitz, N., Veness, J., Desjardins, G., Rusu, A. A., Milan, K., Quan, J., Ramalho, T., Grabska-Barwinska, A., et al. Overcoming catastrophic forgetting in neural networks. *PNAS*, 2017.

Kiros, R., Zhu, Y., Salakhutdinov, R. R., Zemel, R., Urtasun, R., Torralba, A., and Fidler, S. Skip-thought vectors. In *NIPS*. 2015.

Kornblith, S., Shlens, J., and Le, Q. V. Do better imagenet models transfer better? *arXiv preprint arXiv:1805.08974*, 2018.

Lang, K. Newsweeder: Learning to filter netnews. In *ICML*, 1995.

Le, Q. and Mikolov, T. Distributed representations of sentences and documents. In *ICML*, 2014.

Lichman, M. UCI machine learning repository, 2013.

Long, J., Shelhamer, E., and Darrell, T. Fully convolutional networks for semantic segmentation. In *CVPR*, 2015.

McCann, B., Bradbury, J., Xiong, C., and Socher, R. Learned in translation: Contextualized word vectors. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R. (eds.), *NIPS*. 2017.

McCloskey, M. and Cohen, N. J. Catastrophic interference in connectionist networks: The sequential learning problem. In *Psychology of learning and motivation*. 1989.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., and Dean, J. Distributed representations of words and phrases and their compositionality. In *NIPS*. 2013.

Peng, H., Thomson, S., and Smith, N. A. Deep multitask learning for semantic dependency parsing. In *ACL*, 2017.

Pennington, J., Socher, R., and Manning, C. Glove: Global vectors for word representation. In *EMNLP*, 2014.

Perez, E., Strub, F., de Vries, H., Dumoulin, V., and Courville, A. C. Film: Visual reasoning with a general conditioning layer. *AAAI*, 2018.

Peters, M., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., and Zettlemoyer, L. Deep contextualized word representations. In *NAACL*, 2018.

Radford, A., Narasimhan, K., Salimans, T., and Sutskever, I. Improving language understanding by generative pre-training. *URL https://s3-us-west-2. amazonaws. com/openai-assets/research-covers/language-unsupervised/language_ understanding_paper. pdf*, 2018.

Rajpurkar, P., Zhang, J., Lopyrev, K., and Liang, P. Squad: 100,000+ questions for machine comprehension of text. In *EMNLP*, 2016.

Rajpurkar, P., Jia, R., and Liang, P. Know what you don't know: Unanswerable questions for squad. In *ACL*, 2018.

Rebuffi, S., Vedaldi, A., and Bilen, H. Efficient parametrization of multi-domain deep neural networks. In *CVPR*, 2018.

Rebuffi, S.-A., Bilen, H., and Vedaldi, A. Learning multiple visual domains with residual adapters. In *NIPS*. 2017.

Rosenfeld, A. and Tsotsos, J. K. Incremental learning through deep adaptation. *IEEE transactions on pattern analysis and machine intelligence*, 2018.

Rusu, A. A., Rabinowitz, N. C., Desjardins, G., Soyer, H., Kirkpatrick, J., Kavukcuoglu, K., Pascanu, R., and Hadsell, R. Progressive neural networks. *arXiv preprint arXiv:1606.04671*, 2016.

Simonyan, K. and Zisserman, A. Very deep convolutional networks for large-scale image recognition. *ICLR*, 2014.

Sina Semnani, Kaushik Sadagopan, F. T. BERT-A: Fine-tuning BERT with Adapters and Data Augmentation. *http://web.stanford.edu/class/cs224n/reports/default/15848417.pdf*, 2019.

Stickland, A. C. and Murray, I. BERT and PALs: Projected Attention Layers for Efficient Adaptation in Multi-Task Learning. *arXiv preprint arXiv:1902.02671*, 2019.

Thrun, S. Learning to learn. chapter Lifelong Learning Algorithms. 1998.

Turian, J., Ratinov, L., and Bengio, Y. Word representations: A simple and general method for semi-supervised learning. In *ACL*, 2010.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. u., and Polosukhin, I. Attention is all you need. In *NIPS*. 2017.

Wang, A., Singh, A., Michael, J., Hill, F., Levy, O., and Bowman, S. R. Glue: A multi-task benchmark and analysis platform for natural language understanding. *ICLR*, 2018.

Wong, C., Houlsby, N., Lu, Y., and Gesmundo, A. Transfer learning with neural automl. In *NeurIPS*. 2018.

Yosinski, J., Clune, J., Bengio, Y., and Lipson, H. How transferable are features in deep neural networks? In Ghahramani, Z., Welling, M., Cortes, C., Lawrence, N. D., and Weinberger, K. Q. (eds.), *NIPS*. 2014.

Zenke, F., Poole, B., and Ganguli, S. Continual learning through synaptic intelligence. *ICML*, 2017.

Zoph, B. and Le, Q. V. Neural architecture search with reinforcement learning. In *ICLR*, 2017.

Supplementary Material for Parameter-Efficient Transfer Learning for NLP

A. Additional Text Classification Tasks

Dataset	Train examples	Validation examples	Test examples	Classes	Avg text length	Reference
20 newsgroups	15076	1885	1885	20	1903	(Lang, 1995)
Crowdflower airline	11712	1464	1464	3	104	crowdflower.com
Crowdflower corporate messaging	2494	312	312	4	121	crowdflower.com
Crowdflower disasters	8688	1086	1086	2	101	crowdflower.com
Crowdflower economic news relevance	6392	799	800	2	1400	crowdflower.com
Crowdflower emotion	32000	4000	4000	13	73	crowdflower.com
Crowdflower global warming	3380	422	423	2	112	crowdflower.com
Crowdflower political audience	4000	500	500	2	205	crowdflower.com
Crowdflower political bias	4000	500	500	2	205	crowdflower.com
Crowdflower political message	4000	500	500	9	205	crowdflower.com
Crowdflower primary emotions	2019	252	253	18	87	crowdflower.com
Crowdflower progressive opinion	927	116	116	3	102	crowdflower.com
Crowdflower progressive stance	927	116	116	4	102	crowdflower.com
Crowdflower US economic performance	3961	495	496	2	305	crowdflower.com
Customer complaint database	146667	18333	18334	157	1046	catalog.data.gov
News aggregator dataset	338349	42294	42294	4	57	(Lichman, 2013)
SMS spam collection	4459	557	558	2	81	(Almeida et al., 2011)

Table 3. Statistics and references for the additional text classification tasks.

Dataset	Epochs (Fine-tune)	Epochs (Adapters)
20 newsgroups	50	50
Crowdflower airline	50	20
Crowdflower corporate messaging	100	50
Crowdflower disasters	50	50
Crowdflower economic news relevance	20	20
Crowdflower emotion	20	20
Crowdflower global warming	100	50
Crowdflower political audience	50	20
Crowdflower political bias	50	50
Crowdflower political message	50	50
Crowdflower primary emotions	100	100
Crowdflower progressive opinion	100	100
Crowdflower progressive stance	100	100
Crowdflower US economic performance	100	20
Customer complaint database	20	20
News aggregator dataset	20	20
SMS spam collection	50	20

Table 4. Number of training epochs selected for the additional classification tasks.

Parameter	Search Space
1) Input embedding modules	Refer to Table 6
2) Fine-tune input embedding module	{True, False}
3) Lowercase text	{True, False}
4) Remove non alphanumeric text	{True, False}
5) Use convolution	{True, False}
6) Convolution activation	{relu, relu6, leaky relu, swish, sigmoid, tanh}
7) Convolution batch norm	{True, False}
8) Convolution max ngram length	{2, 3}
9) Convolution dropout rate	[0.0, 0.4]
10) Convolution number of filters	[50, 200]
11) Convolution embedding dropout rate	[0.0, 0.4]
12) Number of hidden layers	{0, 1, 2, 3, 5}
13) Hidden layers size	{64, 128, 256}
14) Hidden layers activation	{relu, relu6, leaky relu, swish, sigmoid, tanh}
15) Hidden layers normalization	{none, batch norm, layer norm}
16) Hidden layers dropout rate	{0.0, 0.05, 0.1, 0.2, 0.3, 0.4, 0.5}
17) Deep tower learning rate	{0.001, 0.005, 0.01, 0.05, 0.1, 0.5}
18) Deep tower regularization weight	{0.0, 0.0001, 0.001, 0.01}
19) Wide tower learning rate	{0.001, 0.005, 0.01, 0.05, 0.1, 0.5}
20) Wide tower regularization weight	{0.0, 0.0001, 0.001, 0.01}
21) Number of training samples	{1e5, 2e5, 5e5, 1e6, 2e6}

Table 5. The search space of baseline models for the additional text classification tasks.

ID	Dataset size (tokens)	Embed dim.	Vocab. size	Training algorithm	TensorFlow Hub Handles Prefix: `https://tfhub.dev/google/`
English-small	7B	50	982k	Lang. model	`nnlm-en-dim50-with-normalization/1`
English-big	200B	128	999k	Lang. model	`nnlm-en-dim128-with-normalization/1`
English-wiki-small	4B	250	1M	Skipgram	`Wiki-words-250-with-normalization/1`
English-wiki-big	4B	500	1M	Skipgram	`Wiki-words-500-with-normalization/1`
Universal-sentence-encoder	-	512	-	(Cer et al., 2018)	`universal-sentence-encoder/2`

Table 6. Options for text input embedding modules. These are pre-trained text embedding tables. We provide the handle for the modules that are publicly distributed via the TensorFlow Hub service (`https://www.tensorflow.org/hub`).

B. Learning Rate Robustness

We test the robustness of adapters and fine-tuning to the learning rate. We ran experiments with learning rates in the range $[2 \cdot 10^{-5}, 10^{-3}]$, and selected the best hyperparameters for each method at each learning rate. Figure 7 shows the results.

Dataset	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
20 newsgroups	Universal-sentence-encoder	False	True	True	False	relu6	False	2	0.37	94	0.38	1	128	leaky relu	batch norm	0.5	0.5	0	0.05	0.0001	1000000
Crowdflower airline	English-big	False	False	False	True	leaky relu	False	3	0.36	200	0.07	0	128	tanh	layer norm	0.4	0.1	0.001	0.05	0.001	200000
Crowdflower corporate messaging	English-big	False	False	True	True	tanh	True	3	0.40	56	0.40	1	64	tanh	batch norm	0.5	0.5	0.001	0.01	0	200000
Crowdflower disasters	Universal-sentence-encoder	True	True	False	True	swish	True	3	0.27	52	0.22	0	64	relu	none	0.2	0.005	0.0001	0.005	0.01	500000
Crowdflower economic news relevance	Universal-sentence-encoder	True	True	False	False	leaky relu	False	2	0.27	63	0.04	3	128	swish	layer norm	0.2	0.01	0.01	0.001	0	100000
Crowdflower emotion	Universal-sentence-encoder	False	True	False	False	relu6	False	3	0.35	132	0.34	1	64	tanh	none	0.05	0.05	0	0.05	0	200000
Crowdflower global warming	Universal-sentence-encoder	False	True	True	False	swish	False	3	0.39	200	0.36	1	128	leaky relu	batch norm	0.4	0.05	0	0.001	0.001	1000000
Crowdflower political audience	English-small	True	False	True	True	relu	False	3	0.11	98	0.07	0	64	relu	none	0.5	0.05	0.001	0.001	0	100000
Crowdflower political bias	English-big	False	True	True	False	swish	False	3	0.12	81	0.30	0	64	relu6	none	0	0.01	0	0.005	0.01	200000
Crowdflower political message	Universal-sentence-encoder	False	False	True	False	swish	True	2	0.36	57	0.35	0	64	tanh	none	0.5	0.01	0.001	0.005	0	200000
Crowdflower primary emotions	English-big	False	True	True	True	swish	False	3	0.40	191	0.03	0	256	relu6	none	0.5	0.1	0.001	0.05	0	200000
Crowdflower progressive opinion	English-big	True	False	True	True	relu6	False	3	0.40	199	0.28	0	128	relu	batch norm	0.3	0.1	0.01	0.005	0.001	200000
Crowdflower progressive stance	Universal-sentence-encoder	True	False	True	False	relu	True	3	0.01	195	0.00	2	256	tanh	layer norm	0.4	0.005	0	0.005	0.0001	500000
Crowdflower us economic performance	English-big	True	False	True	True	tanh	True	2	0.31	53	0.24	1	256	leaky relu	batch norm	0.3	0.05	0.0001	0.001	0.0001	100000
Customer complaint database	English-big	True	False	False	False	tanh	False	2	0.03	69	0.10	1	256	leaky relu	layer norm	0.1	0.05	0.0001	0.05	0.001	1000000
News aggregator dataset	Universal-sentence-encoder	False	True	True	False	sigmoid	True	2	0.00	156	0.29	3	256	relu	batch norm	0.05	0.05	0	0.5	0.0001	1000000
Sms spam collection	English-wiki-small	True	True	True	True	leaky relu	False	3	0.20	54	0.00	1	128	leaky relu	batch norm	0	0.1	0	0.05	0.01	1000000

Table 7. Search space parameters (see Table 5) for the AutoML baseline models that were selected.

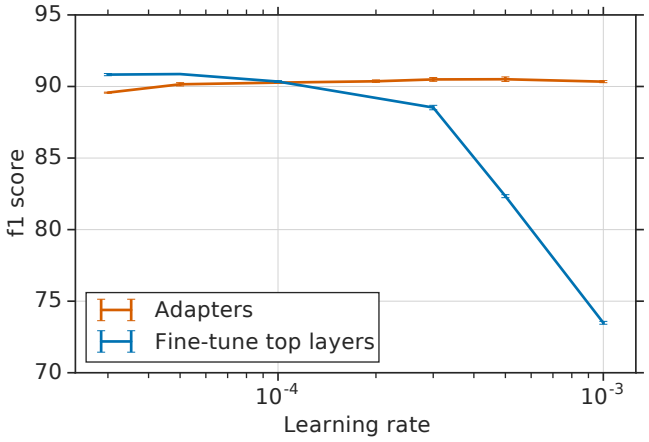

Figure 7. Best performing models at different learning rates. Error vars indicate the s.e.m. across three random seeds.