AdapterHub: A Framework for Adapting Transformers

Jonas Pfeiffer[*1], **Andreas Rücklé**[*1], **Clifton Poth**[*1],
Aishwarya Kamath[2], **Ivan Vulić**[4], **Sebastian Ruder**[5],
Kyunghyun Cho[2,3], **Iryna Gurevych**[1]
[1]Technical University Darmstadt
[2]New York University [3]CIFAR Associate Fellow
[4]University of Cambridge
[5]DeepMind
AdapterHub.ml

Abstract

The current modus operandi in NLP involves downloading and fine-tuning pre-trained models consisting of millions or billions of parameters. Storing and sharing such large trained models is expensive, slow, and time-consuming, which impedes progress towards more general and versatile NLP methods that learn from and for many tasks. Adapters—small learnt bottleneck layers inserted within each layer of a pre-trained model— ameliorate this issue by avoiding full fine-tuning of the entire model. However, sharing and integrating adapter layers is not straightforward. We propose AdapterHub, a framework that allows dynamic "stiching-in" of pre-trained adapters for different tasks and languages. The framework, built on top of the popular HuggingFace Transformers library, enables extremely easy and quick adaptations of state-of-the-art pre-trained models (e.g., BERT, RoBERTa, XLM-R) across tasks and languages. Downloading, sharing, and training adapters is as seamless as possible using minimal changes to the training scripts and a specialized infrastructure. Our framework enables scalable and easy access to sharing of task-specific models, particularly in low-resource scenarios. AdapterHub includes all recent adapter architectures and can be found at AdapterHub.ml.

1 Introduction

Recent advances in NLP leverage transformer-based language models (Vaswani et al., 2017), pre-trained on large amounts of text data (Devlin et al., 2019; Liu et al., 2019; Conneau et al., 2020). These models are fine-tuned on a target task and achieve state-of-the-art (SotA) performance for most natural language understanding tasks. Their performance has been shown to scale with their size (Kaplan et al., 2020) and recent models have reached billions of parameters (Raffel et al., 2019; Brown et al., 2020). While fine-tuning large pre-trained models on target task data can be done fairly efficiently (Howard and Ruder, 2018), training them for multiple tasks and sharing trained models is often prohibitive. This precludes research on more modular architectures (Shazeer et al., 2017), task composition (Andreas et al., 2016), and injecting biases and external information (e.g., world or linguistic knowledge) into large models (Lauscher et al., 2019; Wang et al., 2020).

Adapters (Houlsby et al., 2019) have been introduced as an alternative lightweight fine-tuning strategy that achieves on-par performance to full fine-tuning (Peters et al., 2019) on most tasks. They consist of a small set of additional newly initialized weights at every layer of the transformer. These weights are then trained during fine-tuning, while the pre-trained parameters of the large model are kept frozen/fixed. This enables efficient parameter sharing between tasks by training many task-specific and language-specific adapters for the same model, which can be exchanged and combined post-hoc. Adapters have recently achieved strong results in multi-task and cross-lingual transfer learning (Pfeiffer et al., 2020a,b).

However, reusing and sharing adapters is not straightforward. Adapters are rarely released individually; their architectures differ in subtle yet important ways, and they are model, task, and language dependent. To mitigate these issues and facilitate transfer learning with adapters in a range of settings, we propose AdapterHub, a framework that enables seamless training and sharing of adapters.

AdapterHub is built on top of the popular `transformers` framework by HuggingFace[1] (Wolf et al., 2019), which provides access to state-of-the-art pre-trained language models. We en-

*Equal contribution.

[1]https://github.com/huggingface/transformers

hance `transformers` with adapter modules that can be combined with existing SotA models with minimal code edits. We additionally provide a website that enables quick and seamless upload, download, and sharing of pre-trained adapters. AdapterHub is available online at: AdapterHub.ml.

AdapterHub for the first time enables NLP researchers and practitioners to easily and efficiently share and obtain access to models that have been trained for particular tasks, domains, and languages. This opens up the possibility of building on and combining information from many more sources than was previously possible and makes research such as intermediate task training (Pruksachatkun et al., 2020), composing information from many tasks (Pfeiffer et al., 2020a), and training models for very low-resource languages (Pfeiffer et al., 2020b) much more accessible.

Contributions. 1) We propose an easy-to-use and extensible adapter training and sharing framework for transformer-based models such as BERT, RoBERTa, and XLM(-R); 2) we incorporate it into the HuggingFace `transformers` framework, requiring as little as two additional lines of code to train adapters with existing scripts; 3) our framework automatically extracts the adapter weights, storing them separately to the pre-trained transformer model, requiring as little as 1Mb of storage; 4) we provide an open-source framework and website that allows the community to upload their adapter weights, making them easily accessible with only one additional line of code; 5) we incorporate adapter composition as well as adapter stacking out-of-the-box and pave the way for a wide range of other extensions in the future.

2 Adapters

While the predominant methodology for transfer learning is to fine-tune all weights of the pre-trained model, *adapters* have recently been introduced as an alternative approach, with applications in computer vision (Rebuffi et al., 2017) as well as the NLP domain (Houlsby et al., 2019; Bapna and Firat, 2019; Wang et al., 2020; Pfeiffer et al., 2020a,b).

2.1 Adapter Architecture

Adapters are neural modules with a small amount of additional newly introduced parameters Φ within a large pre-trained model with parameters Θ. The parameters Φ are learnt on a target task while keeping Θ fixed; Φ thus learn to encode task-specific representations in intermediate layers of the pre-trained model. Current work predominantly focuses on training adapters for each task separately (Houlsby et al., 2019; Bapna and Firat, 2019; Pfeiffer et al., 2020a,b), which enables parallel training and subsequent combination of the weights.

In NLP, adapters have been mainly used within deep transformer-based architectures (Vaswani et al., 2017). At each transformer layer l, a set of adapter parameters Φ_l is introduced. The placement and architecture of adapter parameters Φ within a pre-trained model is non-trivial and may impact their efficacy: Houlsby et al. (2019) experiment with different adapter architectures, empirically validating that a two-layer feed-forward neural network with a bottleneck works well. While this down- and up-projection has largely been agreed upon, the actual placement of adapters within each transformer block, as well as the introduction of new LayerNorms[2] (Ba et al., 2016) varies in the literature (Houlsby et al., 2019; Bapna and Firat, 2019; Stickland and Murray, 2019; Pfeiffer et al., 2020a). In order to support standard adapter architectures from the literature, as well as to enable easy extensibility, AdapterHub provides a configuration file where the architecture settings can be defined dynamically. We illustrate the different configuration possibilities in Figure 3, and describe them in more detail in §3.

2.2 Why Adapters?

Adapters provide numerous benefits over fully fine-tuning a model such as scalability, modularity, and composition. We now provide a few use-cases for adapters to illustrate their usefulness in practice.

Task-specific Layer-wise Representation Learning. Prior to the introduction of adapters, in order to achieve SotA performance on downstream tasks, the entire pre-trained transformer model needs to be fine-tuned (Peters et al., 2019). Adapters have been shown to work on-par with full fine-tuning, by adapting the representations at every layer. We present the results of fully fine-tuning the model compared to two different adapter architectures on the GLUE benchmark (Wang et al., 2018) in Table 1. The adapters of Houlsby et al. (2019, Figure 3c) and Pfeiffer et al. (2020a, Figure 3b) comprise two and one down- and up-projection

[2]Layer normalization learns to normalize the inputs across the features. This is usually done by introducing a new set of features for mean and variance.

	Full	Pfeif.	Houl.
RTE (Wang et al., 2018)	66.2	70.8	69.8
MRPC (Dolan and Brockett, 2005)	90.5	89.7	91.5
STS-B (Cer et al., 2017)	88.8	89.0	89.2
CoLA (Warstadt et al., 2019)	59.5	58.9	59.1
SST-2 (Socher et al., 2013)	92.6	92.2	92.8
QNLI (Rajpurkar et al., 2016)	91.3	91.3	91.2
MNLI (Williams et al., 2018)	84.1	84.1	84.1
QQP (Iyer et al., 2017)	91.4	90.5	90.8

Table 1: Mean development scores over 3 runs on GLUE (Wang et al., 2018) leveraging the BERT-Base pre-trained weights. We present the results with full fine-tuning (**Full**) and with the adapter architectures of Pfeiffer et al. (2020a, **Pfeif.**, Figure 3b) and Houlsby et al. (2019, **Houl.**, Figure 3c) both with bottleneck size 48. We show F1 for MRPC, Spearman rank correlation for STS-B, and accuracy for the rest. RTE is a combination of datasets (Dagan et al., 2005; Bar-Haim et al., 2006; Giampiccolo et al., 2007).

	Base		**Large**	
CRate	#Params	Size	#Params	Size
64	0.2M	0.9Mb	0.8M	3.2Mb
16	0.9M	3.5Mb	3.1M	13Mb
2	7.1M	28Mb	25.2M	97Mb

Table 2: Number of additional parameters and compressed storage space of the adapter of Pfeiffer et al. (2020a) in (Ro)BERT(a)-Base and Large transformer architectures. The adapter of Houlsby et al. (2019) requires roughly twice as much space. *CRate* refers to the adapter's compression rate: e.g., a. rate of 64 means that the adapter's bottleneck layer is 64 times smaller than the underlying model's hidden layer size.

within each transformer layer, respectively. The former adapter thus has more capacity at the cost of training and inference speed. We find that for all settings, there is no large difference in terms of performance between the model architectures, verifying that training adapters is a suitable and lightweight alternative to full fine-tuning in order to achieve SotA performance on downstream tasks.

Small, Scalable, Shareable. Transformer-based models are very deep neural networks with millions or billions of weights and large storage requirements, e.g., around 2.2Gb of compressed storage space is needed for XLM-R Large (Conneau et al., 2020). Fully fine-tuning these models for each task separately requires storing a copy of the fine-tuned model for each task. This impedes both iterating and parallelizing training, particularly in storage-restricted environments.

Adapters mitigate this problem. Depending on the model size and the adapter bottleneck size, a single task requires as little as 0.9Mb storage space. We present the storage requirements in Table 2. This highlights that $> 99\%$ of the parameters required for each target task are fixed during training and can be shared across all models for inference. For instance, for the popular Bert-Base model with a size of 440Mb, storing 2 fully fine-tuned models amounts to the same storage space required by 125 models with adapters, when using a bottleneck size of 48 and adapters of Pfeiffer et al. (2020a). Moreover, when performing inference on a mobile device, adapters can be leveraged to save a significant amount of storage space, while supporting a large number of target tasks. Additionally, due to the small size of the adapter modules—which in many cases do not exceed the file size of an image—new tasks can be added on-the-fly. Overall, these factors make adapters a much more computationally—and ecologically (Strubell et al., 2019)—viable option compared to updating entire models. Easy access to fine-tuned models may also improve reproducibility as researchers will be able to easily rerun and evaluate trained models of previous work.

Modularity of Representations. Adapters learn to encode information of a task within designated parameters. Due to the encapsulated placement of adapters, wherein the surrounding parameters are fixed, at each layer an adapter is forced to learn an output representation compatible with the subsequent layer of the transformer model. This setting allows for modularity of components such that adapters can be stacked on top of each other, or replaced dynamically. In a recent example, Pfeiffer et al. (2020b) successfully combine adapters that have been independently trained for specific tasks and languages. This demonstrates that adapters are modular and that output representations of different adapters are compatible. As NLP tasks become more complex and require knowledge that is not directly accessible in a single monolithic pre-trained model (Ruder et al., 2019), adapters will provide NLP researchers and practitioners with many more sources of relevant information that can be easily combined in an efficient and modular way.

Non-Interfering Composition of Information. Sharing information across tasks has a long-standing history in machine learning (Ruder, 2017). Multi-task learning (MTL), which shares a set of parameters between tasks, has arguably received the most attention. However, MTL suffers from

problems such as catastrophic forgetting where information learned during earlier stages of training is "overwritten" (Masson et al., 2019), catastrophic interference where the performance of a set of tasks deteriorates when adding new tasks (Hashimoto et al., 2017), and intricate task weighting for tasks with different distributions (Sanh et al., 2019).

The encapsulation of adapters forces them to learn output representations that are compatible across tasks. When training adapters on different downstream tasks, they store the respective information in their designated parameters. Multiple adapters can then be combined, e.g., with attention (Pfeiffer et al., 2020a). Because the respective adapters are trained separately, the necessity of sampling heuristics due to skewed data set sizes no longer arises. By separating knowledge extraction and composition, adapters mitigate the two most common pitfalls of multi-task learning, catastrophic forgetting and catastrophic interference.

Overcoming these problems together with the availability of readily available trained task-specific adapters enables researchers and practitioners to leverage information from specific tasks, domains, or languages that is often more relevant for a specific application—rather than more general pre-trained counterparts. Recent work (Howard and Ruder, 2018; Phang et al., 2018; Pruksachatkun et al., 2020; Gururangan et al., 2020) has shown the benefits of such information, which was previously only available by fully fine-tuning a model on the data of interest prior to task-specific fine-tuning.

3 AdapterHub

AdapterHub consists of two core components: **1)** A library built on top of HuggingFace `transformers`, and **2)** a website that dynamically provides analysis and filtering of pre-trained adapters. AdapterHub provides tools for the entire life-cycle of adapters, illustrated in Figure 1 and discussed in what follows: ① introducing new adapter weights Φ into pre-trained transformer weights Θ; ② training adapter weights Φ on a downstream task (while keeping Θ frozen); ③ automatic extraction of the trained adapter weights Φ' and open-sourcing the adapters; ④ automatic visualization of the adapters with configuration filters; ⑤ on-the-fly downloading/caching the pre-trained adapter weights Φ' and stitching the adapter into the pre-trained transformer model Θ; ⑥ performing inference with the trained adapter transformer model.

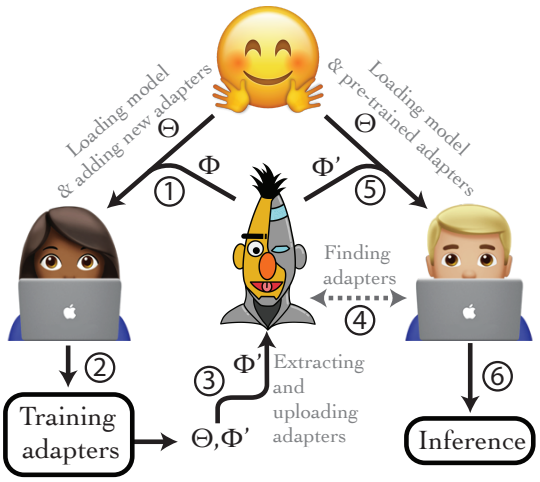


Figure 1: The AdapterHub Process graph. Adapters Φ are introduced into a pre-trained transformer Θ (step ①) and are trained (②). They can then be extracted and open-sourced (③) and visualized (④). Pre-trained adapters are downloaded on-the-fly (⑤) and stitched into a model that is used for inference (⑥).

① Adapters in Transformer Layers

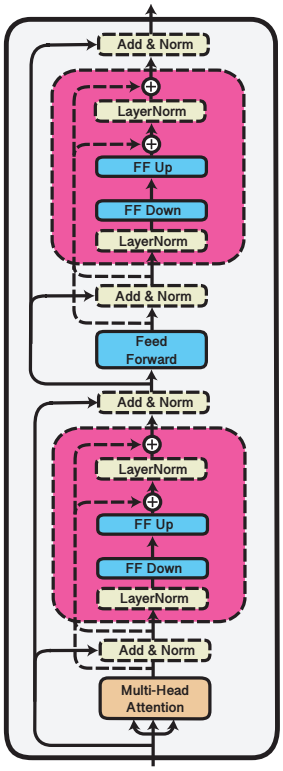
We minimize the required changes to existing HuggingFace training scripts, resulting in only two additional lines of code. In Figure 2 we present the required code to add adapter weights (line 3) and freeze all the transformer weights Θ (line 4). In this example, the model is prepared to train a task adapter on the binary version of the Stanford Sentiment Treebank (SST; Socher et al., 2013) using the adapter architecture of Pfeiffer et al. (2020a). Similarly, language adapters can be added by setting the type parameter to `AdapterType.text_language`, and other adapter architectures can be chosen accordingly.

While we provide ready-made configuration files for well-known architectures in the current literature, adapters are dynamically configurable, which makes it possible to define a multitude of architectures. We illustrate the configurable components as dashed lines and objects in Figure 3. The configurable components are placements of new weights, residual connections as well as placements of LayerNorm layers (Ba et al., 2016).

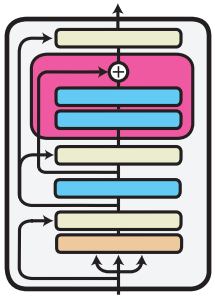
The code changes within the HuggingFace `transformers` framework are realized through `MixIns`, which are inherited by the respective transformer classes. This minimizes the amount of code changes of our proposed extensions and encapsulates adapters as designated classes. It further increases readability as adapters are clearly sepa-

```
1 from adapter_transformers import AutoModelForSequenceClassification, AdapterType
2 model = AutoModelForSequenceClassification.from_pretrained("roberta-base")
3 model.add_adapter("sst-2", AdapterType.text_task, config="pfeiffer")
4 model.train_adapters(["sst-2"])
5 # Train model ...
6 model.save_adapter("adapters/text-task/sst-2/", "sst")
7 # Push link to zip file to AdapterHub ...
```

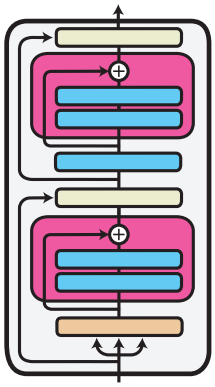
Figure 2: ① Adding new adapter weights Φ to pre-trained RoBERTa-Base weights Θ (line 3), and freezing Θ (line 4). ③ Extracting and storing the trained adapter weights Φ' (line 7).



(a) Configuration Possibilities



(b) Pfeiffer Architecture



(c) Houlsby Architecture

Figure 3: Dynamic customization possibilities where dashed lines in (a) show the current configuration options. These options include the placements of new weights Φ (including down and up projections as well as new LayerNorms), residual connections, bottleneck sizes as well as activation functions. All new weights Φ are illustrated within the pink boxes, everything outside belongs to the pre-trained weights Θ. In addition, we provide pre-set configuration files for architectures in the literature. The resulting configurations for the architecture proposed by Pfeiffer et al. (2020a) and Houlsby et al. (2019) are illustrated in (b) and (c) respectively. We also provide a configuration file for the architecture proposed by Bapna and Firat (2019), not shown here.

rated from the main `transformers` code base, which makes it easy to keep both repositories in sync as well as to extend AdapterHub.

② Training Adapters

Adapters are trained in the same manner as full fine-tuning of the model. The information is passed through the different layers of the transformer where additionally to the pre-trained weights at every layer the representations are additionally passed through the adapter parameters. However, in contrast to full fine-tuning, the pre-trained weights Θ are fixed and only the adapter weights Φ and the prediction head are trained. Because Θ is fixed, the adapter weights Φ are encapsuled within the transformer weights, forcing them to learn compatible representations across tasks.

③ Extracting and Open-Sourcing Adapters

When training adapters instead of full fine-tuning, it is no longer necessary to store checkpoints of the entire model. Instead, only the adapter weights Φ', as well as the prediction head need to be stored, as the base model's weights Θ remain the same. This is integrated automatically as soon as adapters are trained, which significantly reduces the required storage space during training and enables storing a large number of checkpoints simultaneously.

When adapter training has completed, the parameter file together with the corresponding adapter configuration file are zipped and uploaded to a public server. The user then enters the metadata (e.g., URL to weights, user info, description of training procedure, data set used, adapter architecture, GitHub handle, Twitter handle) into a designated YAML file and issues a pull request to the AdapterHub GitHub repository. When all automatic checks pass, the AdapterHub.ml website is automatically regenerated with the newly available adapter, which makes it possible for users to immediately find and use these new weights described by the metadata. We hope that the ease of sharing pre-trained adapters will further facilitate and speed up new developments in transfer learning in NLP.

```
1 from adapter_transformers import AutoModelForSequenceClassification, AdapterType
2 model = AutoModelForSequenceClassification.from_pretrained("roberta-base")
3 model.load_adapter("sst", config="pfeiffer")
```

Figure 4: ⑤ After the correct adapter has been identified by the user on the explore page of AdapterHub.ml, they can load and stitch the pre-trained adapter weights Φ' into the transformer Θ (line 3).

④ Finding Pre-Trained Adapters

The website AdapterHub.ml provides a dynamic overview of the currently available pre-trained adapters. Due to the large number of tasks in many different languages as well as different transformer models, we provide an intuitively understandable hierarchical structure, as well as search options. This makes it easy for users to find adapters that are suitable for their use-case. Namely, AdapterHub's explore page is structured into three hierarchical levels. At the *first* level, adapters can be viewed by task or language. The *second* level allows for a more fine-grained distinction separating adapters into data sets of higher-level NLP tasks following a categorization similar to paperswithcode.com. For languages, the second level distinguishes the adapters by the language they were trained on. The *third* level separates adapters into individual datasets or domains such as SST for sentiment analysis or Wikipedia for Swahili.

When a specific dataset has been selected, the user can see the available pre-trained adapters for this setting. Adapters depend on the transformer model they were trained on and are otherwise *not* compatible.[3] The user selects the model architecture and certain hyper-parameters and is shown the compatible adapters. When selecting one of the adapters, the user is provided with additional information about the adapter, which is available in the metadata (see ③ again for more information).

⑤ Stitching-In Pre-Trained Adapters

Pre-trained adapters can be stitched into the large transformer model as easily as adding randomly initialized weights; this requires a single line of code, see Figure 4, line 3. When selecting an adapter on the website (see ④ again) the user is provided with sample code, which corresponds to the configuration necessary to include the specific weights.[4]

[3] We plan to look into mapping adapters between different models as future work.

[4] When selecting an adapter based on a name, we allow for string matching as long as there is no ambiguity.

⑥ Inference with Adapters

Inference with a pre-trained model that relies on adapters is in line with the standard inference practice based on full fine-tuning. Similar to *training* adapters, during inference the active adapter name is passed into the model together with the text tokens. At every transformer layer the information is passed through the transformer layers and the corresponding adapter parameters.

The adapters can be used for inference in the designated task they were trained on. To this end, we provide an option to upload the prediction heads together with the adapter weights. In addition, they can be used for further research such as transferring the adapter to a new task, stacking multiple adapters, fusing the information from diverse adapters, or enriching AdapterHub with adapters for other modalities, among many other possible modes of usage and future directions.

4 Conclusion and Future Work

We have introduced AdapterHub, a novel easy-to-use framework that enables simple and effective transfer learning via training and community sharing of *adapters*. Adapters are small neural modules that can be stitched into large pre-trained transformer models to facilitate, simplify, and speed up transfer learning across a range of languages and tasks. AdapterHub is built on top of the commonly used HuggingFace `transformers`, and it requires only adding as little as two lines of code to existing training scripts. Using adapters in AdapterHub has numerous benefits such as improved reproducibility, much better efficiency compared to full fine-tuning, easy extensibility to new models and new tasks, and easy access to trained models.

With AdapterHub, we hope to provide a suitable and stable framework for the community to train, search, and use adapters. We plan to continuously improve the framework, extend the composition and modularity possibilities, and support other transformer models, even the ones yet to come.

Acknowledgments

Jonas Pfeiffer is supported by the Hessian research excellence program "Landes-Offensive zur Entwicklung Wissenschaftlich-Ökonomischer Exzellenz" (LOEWE) as part of the research center EmergenCITY. Andreas Rücklé is supported by the German Federal Ministry of Education and Research (BMBF) under the promotional reference 03VP02540 (ArgumenText) and by the European Regional Development Fund (ERDF) and the Hessian State Chancellery – Hessian Minister of Digital Strategy and Development under the promotional reference 20005482 (TexPrax). Aishwarya Kamath is supported in part by a DeepMind PhD Fellowship. The work of Ivan Vulić is supported by the ERC Consolidator Grant LEXICAL: Lexical Acquisition Across Languages (no 648909). Kyunghyun Cho is supported by Samsung Advanced Institute of Technology (Next Generation Deep Learning: from pattern recognition to AI) and Samsung Research (Improving Deep Learning using Latent Structure).

We would like to thank Isabel Pfeiffer for the illustrations.

References

Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. 2016. Learning to Compose Neural Networks for Question Answering. In *Proceedings of NAACL 2016*.

Lei Jimmy Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. 2016. Layer normalization. *arXiv preprint*.

Ankur Bapna and Orhan Firat. 2019. Simple, scalable adaptation for neural machine translation. In *Proceedings of EMNLP-IJCNLP 2019*.

Roy Bar-Haim, Ido Dagan, Bill Dolan, Lisa Ferro, Danilo Giampiccolo, Bernardo Magnini, and Idan Szpektor. 2006. The second pascal recognising textual entailment challenge. In *Proceedings of the PASCAL@ACL 2006*.

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*.

Daniel Cer, Mona Diab, Eneko Agirre, Iñigo Lopez-Gazpio, and Lucia Specia. 2017. SemEval-2017 task 1: Semantic textual similarity multilingual and crosslingual focused evaluation. In *Proceedings of SemEval-2017*.

Alexis Conneau, Kartikay Khandelwal, Naman Goyal, Vishrav Chaudhary, Guillaume Wenzek, Francisco Guzmán, Edouard Grave, Myle Ott, Luke Zettlemoyer, and Veselin Stoyanov. 2020. Unsupervised cross-lingual representation learning at scale. In *Proceedings of ACL 2020*.

Ido Dagan, Oren Glickman, and Bernardo Magnini. 2005. The pascal recognising textual entailment challenge. In *PASCAL MLCW 2005*.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: pre-training of deep bidirectional transformers for language understanding. In *Proceedings of NAACL 2019*.

William B Dolan and Chris Brockett. 2005. Automatically constructing a corpus of sentential paraphrases. In *Proceedings of IWP@IJCNLP 2005*.

Danilo Giampiccolo, Bernardo Magnini, Ido Dagan, and Bill Dolan. 2007. The third pascal recognizing textual entailment challenge. In *Proceedings of the PASCAL@ACL 2007*.

Suchin Gururangan, Ana Marasović, Swabha Swayamdipta, Kyle Lo, Iz Beltagy, Doug Downey, and Noah A Smith. 2020. Don't stop pretraining: Adapt language models to domains and tasks. In *Proceedings of ACL 2020*.

Kazuma Hashimoto, Caiming Xiong, Yoshimasa Tsuruoka, and Richard Socher. 2017. A Joint Many-Task Model: Growing a Neural Network for Multiple NLP Tasks. In *Proceedings of EMNLP 2017*.

Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzkebski, Bruna Morrone, Quentin de Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. 2019. Parameter-efficient transfer learning for NLP. In *Proceedings of ICML 2019*.

Jeremy Howard and Sebastian Ruder. 2018. Universal Language Model Fine-tuning for Text Classification. In *Proceedings of ACL 2018*.

Shankar Iyer, Nikhil Dandekar, and Kornel Csernai. 2017. First quora dataset release: Question pairs.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling Laws for Neural Language Models. *arXiv preprint*.

Anne Lauscher, Ivan Vulić, Edoardo Maria Ponti, Anna Korhonen, and Goran Glavaš. 2019. Specializing unsupervised pretraining models for word-level semantic similarity. *arXiv preprint*.

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*.

Cyprien De Masson, Sebastian Ruder, Lingpeng Kong, and Dani Yogatama. 2019. Episodic Memory in Lifelong Language Learning. In *Advances in NeurIPS 2019*.

Matthew E. Peters, Sebastian Ruder, and Noah A. Smith. 2019. To tune or not to tune? adapting pretrained representations to diverse tasks. In *Proceedings of the RepL4NLP@ACL 2019*.

Jonas Pfeiffer, Aishwarya Kamath, Andreas Rücklé, Kyunghyun Cho, and Iryna Gurevych. 2020a. AdapterFusion: Non-destructive task composition for transfer learning. *arXiv preprint*.

Jonas Pfeiffer, Ivan Vulić, Iryna Gurevych, and Sebastian Ruder. 2020b. MAD-X: An Adapter-based Framework for Multi-task Cross-lingual Transfer. *arXiv preprint*.

Jason Phang, Thibault Févry, and Samuel R Bowman. 2018. Sentence encoders on stilts: Supplementary training on intermediate labeled-data tasks. *arXiv preprint*.

Yada Pruksachatkun, Jason Phang, Haokun Liu, Phu Mon Htut, Xiaoyi Zhang, Richard Yuanzhe Pang, Clara Vania, Katharina Kann, and Samuel R. Bowman. 2020. Intermediate-Task Transfer Learning with Pretrained Models for Natural Language Understanding: When and Why Does It Work? In *Proceedings of ACL 2020*.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2019. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *arXiv preprint*.

Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. SQuAD: 100,000+ Questions for Machine Comprehension of Text. In *Proceedings of EMNLP 2016*.

Sylvestre-Alvise Rebuffi, Hakan Bilen, and Andrea Vedaldi. 2017. Learning multiple visual domains with residual adapters. In *Advances in NeurIPS 2017*.

Sebastian Ruder. 2017. An Overview of Multi-Task Learning in Deep Neural Networks. *arXiv preprint*.

Sebastian Ruder, Matthew E Peters, Swabha Swayamdipta, and Thomas Wolf. 2019. Transfer learning in natural language processing. In *Proceedings of NAACL 2019: Tutorials*.

Victor Sanh, Thomas Wolf, and Sebastian Ruder. 2019. A Hierarchical Multi-task Approach for Learning Embeddings from Semantic Tasks. In *Proceedings of AAAI 2019*.

Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, and Jeff Dean. 2017. Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer. In *Proceedings of ICLR 2017*.

Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, Andrew Ng, and Christopher Potts. 2013. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of EMNLP 2013*.

Asa Cooper Stickland and Iain Murray. 2019. BERT and PALs: Projected attention layers for efficient adaptation in multi-task learning. In *Proceedings of ICML 2019*.

Emma Strubell, Ananya Ganesh, and Andrew McCallum. 2019. Energy and policy considerations for deep learning in NLP. In *Proceedings of ACL 2019*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. In *Advances in NeurIPS 2017*.

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 BlackboxNLP@EMNLP 2018*.

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*.

Alex Warstadt, Amanpreet Singh, and Samuel R Bowman. 2019. Neural network acceptability judgments. *Transactions of ACL 2019*.

Adina Williams, Nikita Nangia, and Samuel R. Bowman. 2018. A broad-coverage challenge corpus for sentence understanding through inference. In *Proceedings NAACL-HLT 2018*.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi andArt Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, and Jamie Brew. 2019. HuggingFace's Transformers: State-of-the-art Natural Language Processing. *arXiv preprint*.