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

Parameter-Efficient Transfer Learning for NLP


Reproduced By:
Kallie Whritenour & Stephanie Schoch
Background on Transfer Learning

- “Transfer learning and domain adaptation refer to the situation where what has been learned in one setting ... is exploited to improve generalization in another setting.” [2]

- Common transfer learning techniques in NLP:
  - **Feature-based transfer:**
    - Real-valued embedding vectors (at word, sentence, or paragraph level) are pre-trained and fed to custom downstream models.
  - **Fine-tuning:**
    - Pre-trained network weights are copied and tuned on a downstream task
      - Original parameters are adjusted for each new task
    - Better performance and more parameter efficient than feature-based transfer (Howard & Ruder, 2018)
      - Fine-tuning with lower layers of a network shared between tasks: increases parameter efficiency
• **BERT**: Vaswani et al. (2017)
  - Transformer network
  - Trained on large text corpora with unsupervised loss
  - SOTA: text classification & extractive question answering

*Figure 1: The Transformer - model architecture.*
Motivation for Paper

● **Limitations of Related Work:**
  ○ Other approaches, like Multi-Task Learning (Caruana, 1997) requires access to all tasks during training.
  ○ Fine-tuning large pre-trained models for transfer learning in NLP is effective but parameter inefficient.
    ■ New sets of weights are required for each task (limited parameter efficiency/compactness)
    ■ Feature-based transfer is even more inefficient.

● **Goal:**
  ○ Build a system that performs well on all tasks in an *online setting*, without training all model parameters for each new task.
    ■ *Online setting*: tasks arrive in a stream

● **Potential Applications/Impact:**
  ○ Cloud services: many tasks arrive from customers in a sequence
Claims

• **Argue:** Fine-tuning large pre-trained models (i.e., BERT) for many downstream tasks is parameter inefficient
  – Parameter efficient solution would involve sharing between tasks

• **Proposed:** Transfer with *adapter modules*
  – *Adapter Modules:* New modules added between layers of a pre-trained network
    • New function is defined with parameters copied from pre-training, small number of parameters are added to the model per task
  – More parameter efficient with minimal performance tradeoff
    • Original network parameters are fixed (parameter sharing), few trainable parameters added per task
  – Yields *compact* and *extensible* downstream models (useful for online tasks):
    • *Compact:* solve many tasks using small number of additional trainable parameters per task
    • *Extensible:* can be trained to solve new tasks without forgetting previous ones
Transfer Learning Tradeoff

Task Generalization

Number of Parameters

- Adapter Tuning for NLP
  - Few parameters added for new task
  - Minimal performance drop
Key Properties of Proposed Strategy

1. Attains **good performance**
2. Permits **training on tasks sequentially**
   (does not require simultaneous access to all datasets)
3. Adds only a **small number of additional parameters** per task
Adapter-based Tuning for Transformers

- Instantiate adapter-based tuning for text Transformers (SOTA for many NLP tasks)
- Consider standard Transformer architecture, proposed in Vaswani et al. (2017).

Figure 1: The Transformer - model architecture.
Inserted serial adapter after each of the two sub-layers in Transformer layer (attention layer, feedforward layer). Adapter is always applied directly to output of sub-layer (after projection back to input size, but before adding skip connection back. Adapter output is then passed directly into the following layer normalization.

Each sublayer is followed immediately by a projection mapping features size back to size of layer's input. Skip connection is applied across each sub-layer. Output of each sub-layer fed into layer normalization.

**Bottleneck architecture** to limit number of parameters

- **Important**: New layers are injected into original network, but original network weights are untouched/shared by many tasks!
Data Summary

- **Task Categories**: Classification, Extractive Question Answering

  - **Classification**:  
    - Transfer BERT Transformer model, with adapters, to 26 text classification tasks (including GLUE benchmark)
      - GLUE (General Language Understanding Evaluation) benchmark:
        - Benchmark of nine sentence- or sentence-pair language understanding tasks built on established existing datasets
      - 17 public classification tasks
        - Analyze parameter/performance trade-off

  - **Extractive Question Answering**:  
    - Tested on: SQuAD Extractive Question Answering v1.1
    - Used to show that adapters work on tasks other than classification
GLUE benchmark: Procedure

- **Transfer from pre-trained BERT-LARGE model:**
  - 24 layers, total of 330M parameters
  - Perform small hyperparameter sweep (learning rates & number of epochs) for adapter tuning
  - Trained on 4 Google Cloud TPUs with a batch size of 32
- **Test using fixed adapter size (# of units in bottleneck), and selecting best size per task from \{8, 64, 256\}**
- **Compare to fine-tuning public, pre-trained BERT transformer network**
  - Current standard for transfer of large pre-trained models, and strategy successfully used with BERT
  - For N tasks, full fine-tuning requires N x # parameters of pre-trained model
  - **Goal:** attain equal performance with fewer total parameters
Experimental Results: GLUE Text Classification

• Performance on GLUE (mean GLUE score across 9 tasks):
  – 80%: adapters
  – 80.4%: full-fine tuning of standard BERT
  – Near state-of-the-art performance
    • Adapters add only 3% of # parameters trained by fine-tuning:
      – Fine-tuning requires 9 x total # BERT parameters
      – Adapters require only 1.3 x parameters

• Other observation:
  – Optimal adapter size varies per dataset
    • Always restricting to size 64 results in small decrease in mean GLUE score: 79.6%

**Takeaway:** On GLUE, adapter-tuning achieved scores within 0.4% of full fine-tuning of BERT, but used only 3% of # parameters trained by fine-tuning!
Project Goals and Included Components

• **Goal:**
  – Reproduce the results from the GLUE tasks presented in the paper.

• **Project components:**
  – Transfer from pre-trained BERT-LARGE model (24 layers, 330M parameters)
    • Fine-tune BERT on each task (100% of parameters trained)
    • Train BERT w/ Adapters on each task
Our Selected GLUE Tasks:

- Selected a subset of GLUE tasks:
  - **Similarity and Paraphrase Task:**
    - Microsoft Research Paraphrase Corpus (MRPC)
      - Automatically extracted sentence pairs from online news sources
      - Human annotations: are sentences semantically equivalent
  - **Single-Sentence Classification Task:**
    - The Corpus of Linguistic Acceptability (CoLA)
      - Sentences w/ acceptability judgements from 22 books and journal articles on linguistic theory
      - Each example: single string of English words, annotated with whether it is a grammatically possible sentence
import datetime
import json
import os
import pprint
import random
import string
import sys

!pip install tensorflow==1.13.1
import tensorflow as tf
tf.logging.set_verbosity(tf.logging.ERROR)

print(tf.__version__)

assert 'COLAB_TPU_ADDR' in os.environ, 'ERROR: Not connected to a TPU runtime'
TPU_ADDRESS = 'grpc://' + os.environ['COLAB_TPU_ADDR']
print('TPU address is', TPU_ADDRESS)

from google.colab import auth
auth.authenticate_user()

with tf.Session(TPU_ADDRESS) as session:
    print('TPU devices: ')
    pprint.pprint(session.list_devices())

# Upload credentials to TPU.
with open('/content/adc.json', 'r') as f:
    auth_info = json.load(f)
tf.contrib.cloud.configure_gcs(session, credentials=auth_info)
# Now credentials are set for all future sessions on this TPU.
We can see that we successfully found 7 TPUs and their address, which we will need to reference later in the code because it changes from session to session
Code Walkthrough: Evaluation Setup

Google TPUs need to access data and pretrained models from Google Cloud Services

Set and check that we've successfully found our GCS Bucket

Here we see our Bucket is correct!
Code Walkthrough: Evaluation Code

Call to run classifier code - Train Mode
- can set hyper-params here
- training takes multiple hours
- Pass data, model, output and TPU paths here

Call to run classifier code - Eval Mode
- Loads fully tuned models trained previously
- Eval takes ~3 min
- Doesn’t change model weights, only applies model to data
Code Walkthrough: Evaluation Code

run_classifier.py
- Controls Training and Evaluation Loop
- Calls other functions
  - organizes running the different components of the model provided by the scripts
- Outputs TF flags

modeling.py
- defines BERT model with adapters
- parameter setting passed here

optimization.py
- Defines Adam Weight Decay Optimizer

tokenization.py
- tokenizes input data based on model used
  - Cased vs. Uncased
Total Parameters (Size of Bert and # of params trained during full fine-tuning) VS Trainable Params (adapter-only parameters are the only trained weights during transfer of Adapter-Bert Model)

- Adapters require only around 3% of trainable parameters compared to fine-tuning (this ratio depends on size of the adapter layers which can be specified as a hyper param.)
Hyper Parameter Setting for Training MRPC Model:

- batch size: 32
- learning rate: 2e-5
- number of epochs: 15
- max. sequence length: 128
- adapter size: 64

Bert Model Parameters: as default
- hidden_size=768,
- num_hidden_layers=12,
- num_attention_heads=12,
- intermediate_size=3072,
- hidden_act="gelu",
- hidden_dropout_prob=0.1,
- attention_probs_dropout_prob=0.1

INFO:tensorflow:***** Eval results *****
INFO:tensorflow: eval_accuracy = 0.85784316
INFO:tensorflow: eval_loss = 1.0469517

Performance on MRPC evaluation set
**Results**

- We achieved comparable results on a subset of GLUE tasks:

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

**Evaluation Accuracy by Model and Dataset**

<table>
<thead>
<tr>
<th></th>
<th>CoLA</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>BERT</td>
<td>0.8504902</td>
<td>0.83895555</td>
</tr>
<tr>
<td>BERT w/ Adapters</td>
<td>0.85784316</td>
<td>0.83783818</td>
</tr>
</tbody>
</table>

- **CoLA discrepancy:** Paper reported Matthew’s coefficient in the table, we reported accuracy, with results similar to Figure on the right:
Visualization of Results

Figure 1: Results for Google’s BERT-CoLA and Total are not reported due to difference in reported metrics. Google’s Adapter-CoLA is taken from line graph in paper, and total is recalculated.
Conclusions

• **Major conclusion/contribution from paper:**
  – Addition of adapter modules adds a small percentage of new parameters for each new task, while still achieving state-of-the-art performance

• **Our results support this!**
Discussion

• **Major Challenge of Project:**
  – Complexity of transformer architecture
  – How to train the models:
    • Need to be run on a GPU with at least 12GB of RAM, or a Cloud TPU
      – Cannot train on local machines
    • Tensorflow versioning issues with UVA CS Server
    • Setting up virtual environment & project directory on server.
  – Data-size exceeds allocated Google Colab space.
    • Needed to set up Cloud TPU Storage Bucket & configure model to work with Google Colab & TPU
  – **Time to train the models!**
Division of Work

- Setting-up Training Environments
  - SLURM: Kallie
  - **Google Colab w/ Cloud Storage Bucket: Kallie**
    - Project directory/Virtual environment w/ server GPUs: Stephanie
- Training Final Models: Kallie
- Running Final Experiments: Kallie
- Prepping Jupyter Notebook: Kallie
- Slides:
  - Paper review slides:
    - Related Work, Graphic Visualization, Conclusions & Future Work: Kallie
    - Motivation, Background, Claim/Target Task, Proposed Solution & Key Properties, Adapter & Architecture Explanation Slides, Data Summary, Experiments: Stephanie
  - Additional final project slides:
    - Results w/ Visualization, Discussion, Project Components: Stephanie
    - Code Walkthrough: Kallie, Stephanie
References


EXTRA SLIDES

The following slides are not part of the presentation, but can be referred to during QA.
Features of Adapter Modules

- Two main features of adapter modules:
  - **Small number of parameters**
    - Adapter modules = small compared to the layers of the original network, so total model size grows slowly when more tasks are added
  - **Near identity initialization**
    - Required for stable training of the adapted model
    - Original network is unaffected when training starts since adapters are initialized to a near-identity function
    - During training, adapter modules can be:
      - Ignored if not required
      - Activated to change distribution of activations throughout network
    - If initialization deviates too far from identity function, model may fail to train
Implications of Bottleneck Architecture

- Total # parameters added per layer (including biases): $2md + d + m$
- $m < d$: limit number of parameters added per task
- Bottleneck dimension $m$: provides means to trade-off performance w/ parameter efficiency
  - Few parameters relative to attention & feedforward layers of original model

- Adapter module itself has a skip-connection internally
  - If parameters of projection layers are initialized to near-zero, the module is initialized to an approximate identity function.
- Additional step: trained new layer normalization parameters per task, alongside layers in the adapter module
  - Yields parameter efficient adaptation of network ($2d$ parameters per layer)
- **Important**: New layers are injected into original network, but original network weights are untouched and shared by many tasks!
Classification: Experiment Set-Up

- **Base Model**: public, pre-trained BERT transformer network
- Classification approach & training procedure from Devlin et al. (2018):
  - **Classification approach**:
    - First token in each sequence is special “classification token”
    - Attach linear layer to embedding of this token to predict class label.
  - **Training procedure**:
    - Optimize using Adam (learning rate is increased linearly over the first 10% of the steps, then decayed linearly to zero)
      - All runs trained on 4 Google Cloud TPUs with a batch size of 32
      - Run a hyperparameter sweep and select the best model according to accuracy on the validation set, for each dataset and algorithm
Additional Classification Tasks

- Used for validation of adapter efficacy in yielding compact, high-performing models
- Diverse range of tasks & datasets (vary across # training examples, # classes, avg. text length, etc.)
- Procedure:
  - Batch size 32, swept learning rates, selected # training epochs from \{20, 50, 100\} via manual inspection of validation set learning curves.
  - Test adapter sizes \{2, 4, 8, 16, 32, 64\}
  - Run additional baseline: variable fine-tuning
  - Collected benchmark performances (since no comprehensive set of SOTA for set of tasks)
- **Result:** *Similar to GLUE, performance of adapter-tuning is close to full fine-tuning (0.4% difference)*
Parameter/Performance Trade-Off

- Smaller adapter size = fewer parameters = higher parameter efficiency... but what is the impact on performance?

- **Adapter size: parameter efficiency/performance trade-off**
  - Compared two baselines:
    - Fine-tuning of top k layers of BERT(Base)
    - Tuning only layer normalization parameters
  - Results:
    - Performance decreases dramatically on GLUE when fewer layers are fine-tuned, but adapters had good performance across a range of sizes two orders of magnitude fewer than fine-tuning.
    - Performance decreased dramatically when tuning only layer normalization parameters
SQuAD Extractive Question Answering

• Used as confirmation that adapters work on tasks beyond classification

• Run on SQuAD v1.1:
  
  – **Task:**
    
    • Given question & Wikipedia paragraph, select the answer span to the question from the paragraph.

  – **Results:**
    
    • Performance is comparable to full fine-tuning (while training many fewer parameters):
      
      – **Adapter size 64 (2% of parameters):** best F1 of 90.4%
      – **Full fine-tuning:** 90.7%
      – **Adapter size 2 (0.1% parameters):** best F1 of 89.9%
Experimental Analysis

• Analyses performed:
  – **Ablation**: to determine which adapters are influential
  – **Robustness investigation**: based on
    • *Initialization scale*
    • *Number of neurons*
  – Documentation of unsuccessful architecture extensions
Experimental Analysis: Ablation

- **Procedure:**
  - Remove some trained adapters & re-evaluate the model (without re-training) on the validation set
  - Experiment performed on BERT-BASE with adapter size 64 on MNLI and CoLA datasets

- **Observation 1:** Each adapter has a small influence on the overall network, but the overall effect is large.
  - Removing any single layer’s adapters has only a small impact on performance.
    - Largest performance drop from removing adapters from single layer was 2%
  - When all adapters are removed from network, performance drops substantially (37% MNLI, 69% CoLA) - scores attained by predicting the majority class

- **Observation 2:** Adapters perform well because they prioritize higher layers/automatically focus on higher levels of the network
  - Adapters on the lower layers have a smaller impact than the higher layers
    - Removing adapters from layers 0-4 on MNLI barely affected performance
  - **Intuition:**
    - Lower layers extract lower-level features shared among tasks
    - Higher layers build features unique to different tasks
Robustness Investigation: Initialization Scale

- Initialization scales:
  - Main experiments:
    - Weights in the adapter module drawn from a zero-mean Gaussian with standard deviation $10^{-2}$, truncated to two standard deviation
  - Investigation for analysis of impact of initialization scale on performance:
    - Test standard deviation in interval $[10^{-7}, 1]$

- Observations:
  - On both datasets, performance of adapters is robust for standard deviations below $10^{-2}$.
  - If initialization is too large, performance degrades (more substantially on CoLA).
Robustness Investigation: Number of Neurons

• **Procedure:**
  – Re-examine experimental data from GLUE benchmark:
    • Observe:
      – Stable quality of model across adapter sizes
      – Only small decrease of performance when using fixed adapter size across all tasks
    – Calculate mean validation accuracy across 8 classification tasks by selecting optimal learning rate/# epochs for each adapter size:
      • Mean validation accuracies for adapter sizes 8, 64, 256:
        – 86.2%, 85.8%, 85.7% = **stability**!
Experimental Analysis: Extensions

- Extensions to adapter architecture that didn’t yield significant performance boost:
  - Add a batch/layer normalization to the adapter
  - Increase number of layers per adapter
  - Try different activation functions (such as tanh)
  - Insert adapters only inside attention layer
  - Add adapters in parallel to main layers (possibly with a multiplicative interaction)

- **All cases:** performance similar to bottleneck, which is more simple and yields strong performance.
Project Components Not Included

• Components we did not reproduce w/ justification:
  – Did not perform hyperparameter sweeps:
    • These metrics not reported in paper, only best configuration was reported.
  – Additional classification tasks:
    • To benchmark these, a Neural AutoML algorithm was run for one week on CPUs using 30 machines.
    • Given training time for the model, GLUE tasks seemed more standardized (as demonstrated by lack of baseline for additional tasks) and important to generate results.
  – SQuAD Extractive Question Answering, Ablation and Robustness Investigation
    • Time-prohibitive for training the models.
Conclusion and Future Work

- The addition of adapter modules was found to add only a few parameters for each new task while still achieving state-of-the-art performance
  - adapters were found to automatically place more weight on higher levels, which coincides with learning features that are task specific
  - model performance was stable across adapter module size
  - adapters were robust to single adapter layer removal but model performance dropped significantly when all adapters were removed
- This work can be extended to applications beyond NLP including: Computer Vision, Machine Translation, and other areas
- More work can be undertaken to understand how adapter modules behave under different architectures, tasks, and hyperparameter settings