COMPACTER: Efficient Low-Rank Hypercomplex Adapter Layers

Rabeeh Karimi Mahabadi\textsuperscript{1, 2} \quad James Henderson\textsuperscript{2} \quad Sebastian Ruder\textsuperscript{3}

\textsuperscript{1}EPFL University, \textsuperscript{2}Idiap Research Institute, \textsuperscript{3}DeepMind
Fine-tuning large-scale pretrained language models with millions and billions of parameters on downstream tasks is:

- Sample-inefficient
- Unstable in low-resource settings
- Requires storing a separate copy of the model for each task
Contributions

- We propose **COMPACTER**
  - A parameter-efficient fine-tuning method
  - With a better trade-off between task performance, memory and training time
- Benchmark recent parameter-efficient methods
  - Provide insights on their performance and efficiency
Background: Adapters

- Freeze the model
- Train adapters and layernorms [1]

![Figure: Adapter integration in a pretrained transformer model.](image-url)
Background: Adapters

- A bottleneck architecture
- Consisting of a down projection, non-linearity, and up projection

Figure: Adapter architecture.
Compact and Efficient Adapter Layers

- Down and Up projections in adapters ($W \in \mathbb{R}^{k \times d}$) are fully connected layers:
  $$Y = Wx + b$$

**Figure:** Adapters’ weights.
Compact and Efficient Adapter Layers

- $\mathbf{W}$ can be learned via parameterized hypercomplex multiplication (PHM) layers [2].
  - Let $\mathbf{W} \in \mathbb{R}^{k \times d}$
  - Assume $k$ and $d$ are divisible by a user-defined hyper-parameter $n \in \mathbb{Z}_{>0}$
  - $\mathbf{W}$ is generated by a summation of Kronecker products between $A_i \in \mathbb{R}^{n \times n}$ and $B_i \in \mathbb{R}^{k/n \times d/n}$

$$\mathbf{W} = \sum_{i=1}^{n} A_i \otimes B_i,$$

- Reduces trainable parameters by $\frac{1}{n}$

**Figure**: Parameterized Hypercomplex Multiplication Layers.
COMPACTER is motivated by the followings:
- There are redundancies in information captured by adapters [1].
- Sharing adapters across layers can cause a small drop in performance [3].
Compacter: Beyond Hypercomplex Adapters

- Each **Compacter** layer’s weight consists of:
  - **Shared Weights** ($A_i$):
    - Common across all adapter layers
    - Capturing useful information for adapting to the target task
  - **Low-rank Weights** ($B_i$):
    - *Adapter-specific* parameters
    - Capturing information relevant for adapting each individual layer

- Low-rank parameterized hypercomplex multiplication layers (LPHM):

  \[
  W = \sum_{i=1}^{n} A_i \otimes B_i = \sum_{i=1}^{n} A_i \otimes (s_i t_i^T).
  \]
We compute sum of Kronecker products of shared matrices $A_i$ and adapter-specific matrices $B_i^j$.
“Fast” weights $B_i$:
- Independent rank-one weights
- Learns adapter-layer specific information

Weights for COMPACTER Layer 1

Weights for COMPACTER Layer 2

Size of $W_1$

COMPACTER's weights

Size of $W_2$
**COMPACTER**

- "Slow" shared weights $A_i$:
  - Shared across all COMPACTER layers
  - Capture general information useful for adapting to the target task

![Diagram of COMPACTER layers and shared weights](image)
Parameter size of COMPACTER weights is much smaller than the size of the weights.
Parameter Efficiency

For a transformer of $L$ layers and adapters of size $k \times d$:

- **Adapter** parameters:
  - $2kd$ parameters for down and up projections (encoder/decoder): $4kd$
  - Total parameters’ complexity: $O(Lkd)$

- **PHM-Adapter**
  - $A_i \in \mathbb{R}^{n \times n}$ and $B_i \in \mathbb{R}^{k \times d}$ define the degree of freedom for $W$
  - Total adapters’ parameters: $4L \times \left( \frac{kd}{n} + n^3 \right)$
  - With a mild assumption $kd > n^4$: $O\left(\frac{1}{n}Lkd\right)$

- **Compacter**
  - $A_i \in \mathbb{R}^{n \times n}$ for all layers: $n^3$
  - Two rank-one weights for each adapter: $4L(k+d)$
  - Total parameters: $4L(k+d) + n^3$
  - With a mild assumption $4L(k+d) > n^3$: $O(L(k+d))$
Benchmarking Parameter-efficient Methods

Our Proposed Methods:

- **Compacter**: We learn adapter weights using LPHM layers.
- **Compacter++**: Removing Compacter layers after the self-attention layer.
- **PHM-Adapter**: We learn adapters’ weight using PHM layers [2].

Baselines:

- **T5_{BASE}**: Fine-tuning all parameters of T5_{BASE} [4]
- **Adapter**: Including adapters after feedforward and self-attention [1]
- **Pfeiffer-Adapter**: Including adapters only after self-attention [5]
- **AdapterDrop**: Dropping adapters from lower transformer layers (first 5 layers) [3]
- **Adapter-LowRank**: Adapter’s weights parameterized as a product of two rank-one weights.
- **BitFit**: Fine-tuning only biases [6, 7].
Intrinsic-SAID: reparameterize in a low-dimensional subspace $\theta^{d'}$ ($d' \ll D$) [8]:

$$\theta_i^D = \theta_{i,0}^D + \lambda_i P \theta_i^{d'-m},$$

- Parameter $\theta_{i,0}^D$ are the pretrained model’s parameters
- $P \in \mathbb{R}^{d'-m} \rightarrow \mathbb{R}^D$ is a random linear projection via the Fastfood transform
- The total trainable parameters are $\theta^{d'-m} \in \mathbb{R}^{d'-m}$ and $\lambda \in \mathbb{R}^m$

Prompt Tuning: Prepends a randomly initialized continuous prompt to the input [9].
- Initializing prompts from pretrained language model’s vocabulary
Trade-off Between Parameter Efficient Fine-tuning Methods

- Trade-off between quantitative performance (score on GLUE (y axis))
- Percentage of trained parameters (x axis, in log scale)
- Memory footprint (size of the circles).
Performance Evaluation: Compacter (++)

- Performs on par with full fine-tuning.
- Outperforms all previous parameter-efficient methods.
- Only trains 0.07% (0.047%) of parameters.
- Reduces memory usage and speeds up the training.

<table>
<thead>
<tr>
<th>Model</th>
<th>Trained params/per task</th>
<th>Avg</th>
<th>Memory (MB)</th>
<th>Δ%</th>
<th>Time/Epoch (min)</th>
<th>Δ%</th>
</tr>
</thead>
<tbody>
<tr>
<td>T5_{BASE}</td>
<td>100%</td>
<td>86.5</td>
<td>167.99</td>
<td>—</td>
<td>42.13</td>
<td>—</td>
</tr>
<tr>
<td>Adapter</td>
<td>0.832%</td>
<td>85.78</td>
<td>124.02</td>
<td>-35.45%</td>
<td>31.81</td>
<td>-24.50%</td>
</tr>
<tr>
<td>Pfeiffer-Adapter</td>
<td>0.427%</td>
<td>86.32</td>
<td>118.4</td>
<td>-41.88%</td>
<td>28.19</td>
<td>-33.09%</td>
</tr>
<tr>
<td>AdapterDrop</td>
<td>0.494%</td>
<td>85.85</td>
<td>119.41</td>
<td>-40.68%</td>
<td>28.08</td>
<td>-33.35%</td>
</tr>
<tr>
<td>Adapter-LowRank</td>
<td>0.073%</td>
<td>85.82</td>
<td>123.8</td>
<td>-35.69%</td>
<td>32.71</td>
<td>-22.36%</td>
</tr>
<tr>
<td>Prompt Tuning</td>
<td>0.034%</td>
<td>75.95</td>
<td>222.27</td>
<td>24.42%</td>
<td>44.54</td>
<td>5.72%</td>
</tr>
<tr>
<td>Intrinsic-SAID</td>
<td>0.009%</td>
<td>85.45</td>
<td>285.40</td>
<td>41.14%</td>
<td>144.01</td>
<td>241.82%</td>
</tr>
<tr>
<td>BitFit</td>
<td>0.126%</td>
<td>84.97</td>
<td>102.31</td>
<td>-64.20%</td>
<td>27.36</td>
<td>-35.06%</td>
</tr>
<tr>
<td>PHM-Adapter</td>
<td>0.179%</td>
<td>86.40</td>
<td>123.93</td>
<td>-35.55%</td>
<td>35.55</td>
<td>-15.62%</td>
</tr>
<tr>
<td>Compacter</td>
<td>0.073%</td>
<td>86.62</td>
<td>123.91</td>
<td>-35.57%</td>
<td>36.48</td>
<td>-13.41%</td>
</tr>
<tr>
<td>Compacter++</td>
<td>0.047%</td>
<td>86.47</td>
<td>118.35</td>
<td>-41.94%</td>
<td>30.96</td>
<td>-26.51%</td>
</tr>
</tbody>
</table>
Performance Evaluation: **Prompt Tuning**

- Low number of parameters but high memory overhead and slow to train
  - Computation of self-attention scales quadratically with the sequence length
- Its performance substantially lags behind full fine-tuning
  - High sensitivity to initialization and learning rate
  - Limited interaction with the model
  - Less suitable to deal with large contexts

<table>
<thead>
<tr>
<th>Model</th>
<th>Trained params/ per task</th>
<th>Avg</th>
<th>Memory (MB)</th>
<th>Δ%</th>
<th>Time/ Epoch (min)</th>
<th>Δ%</th>
</tr>
</thead>
<tbody>
<tr>
<td>T5&lt;sub&gt;BASE&lt;/sub&gt;</td>
<td>100%</td>
<td>86.5</td>
<td>167.99</td>
<td>—</td>
<td>42.13</td>
<td>—</td>
</tr>
<tr>
<td>Adapter</td>
<td>0.832%</td>
<td>85.78</td>
<td>124.02</td>
<td>-35.45%</td>
<td>31.81</td>
<td>-24.50%</td>
</tr>
<tr>
<td>Pfeiffer-Adapter</td>
<td>0.427%</td>
<td>86.32</td>
<td>118.4</td>
<td>-41.88%</td>
<td>28.19</td>
<td>-33.09%</td>
</tr>
<tr>
<td>AdapterDrop</td>
<td>0.494%</td>
<td>85.85</td>
<td>119.41</td>
<td>-40.68%</td>
<td>28.08</td>
<td>-33.35%</td>
</tr>
<tr>
<td>Adapter-LowRank</td>
<td>0.073%</td>
<td>85.82</td>
<td>123.8</td>
<td>-35.69%</td>
<td>32.71</td>
<td>-22.36%</td>
</tr>
<tr>
<td>Prompt Tuning</td>
<td>0.034%</td>
<td>75.95</td>
<td>222.27</td>
<td>24.42%</td>
<td>44.54</td>
<td>5.72%</td>
</tr>
<tr>
<td>Intrinsic-SAID</td>
<td>0.009%</td>
<td>85.45</td>
<td>285.40</td>
<td>41.14%</td>
<td>144.01</td>
<td>241.82%</td>
</tr>
<tr>
<td>BitFit</td>
<td>0.126%</td>
<td>84.97</td>
<td>102.31</td>
<td>-64.20%</td>
<td>27.36</td>
<td>-35.06%</td>
</tr>
<tr>
<td>PHM-Adapter</td>
<td>0.179%</td>
<td>86.40</td>
<td>123.93</td>
<td>-35.55%</td>
<td>35.55</td>
<td>-15.62%</td>
</tr>
<tr>
<td>Compacter</td>
<td>0.073%</td>
<td>86.62</td>
<td>123.91</td>
<td>-35.57%</td>
<td>36.48</td>
<td>-13.41%</td>
</tr>
<tr>
<td>Compacter++</td>
<td>0.047%</td>
<td>86.47</td>
<td>118.35</td>
<td>-41.94%</td>
<td>30.96</td>
<td>-26.51%</td>
</tr>
</tbody>
</table>
**Performance Evaluation: Intrinsic-SAID**

- Tunes only 0.009% of parameters
- Performs worse than fine-tuning
- High memory overhead and slow to train
  - Requires storing large random projection matrices.
  - Computing projections via FastFood transform [10] is slow in practice
  - Not suitable for large-scale pretrained language models

<table>
<thead>
<tr>
<th>Model</th>
<th>Trained params/per task</th>
<th>Avg</th>
<th>Memory (MB)</th>
<th>∆%</th>
<th>Time/Epoch (min)</th>
<th>∆%</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>T5_BASE</strong></td>
<td>100%</td>
<td>86.5</td>
<td>167.99</td>
<td>—</td>
<td>42.13</td>
<td>—</td>
</tr>
<tr>
<td><strong>ADAPTER</strong></td>
<td>0.832%</td>
<td>85.78</td>
<td>124.02</td>
<td>-35.45%</td>
<td>31.81</td>
<td>-24.50%</td>
</tr>
<tr>
<td><strong>Pfeiffer-ADAPTER</strong></td>
<td>0.427%</td>
<td>86.32</td>
<td>118.4</td>
<td>-41.88%</td>
<td>28.19</td>
<td>-33.09%</td>
</tr>
<tr>
<td><strong>ADAPTERDROP</strong></td>
<td>0.494%</td>
<td>85.85</td>
<td>119.41</td>
<td>-40.68%</td>
<td>28.08</td>
<td>-33.35%</td>
</tr>
<tr>
<td><strong>ADAPTER-LOWRANK</strong></td>
<td>0.073%</td>
<td>85.82</td>
<td>123.8</td>
<td>-35.69%</td>
<td>32.71</td>
<td>-22.36%</td>
</tr>
<tr>
<td><strong>PROMPT TUNING</strong></td>
<td>0.034%</td>
<td>75.95</td>
<td>222.27</td>
<td>24.42%</td>
<td>44.54</td>
<td>5.72%</td>
</tr>
<tr>
<td><strong>Intrinsic-SAID</strong></td>
<td>0.009%</td>
<td>85.45</td>
<td>285.40</td>
<td>41.14%</td>
<td>144.01</td>
<td>241.82%</td>
</tr>
<tr>
<td><strong>BitFit</strong></td>
<td>0.126%</td>
<td>84.97</td>
<td>102.31</td>
<td>-64.20%</td>
<td>27.36</td>
<td>-35.06%</td>
</tr>
<tr>
<td><strong>PHM-ADAPTER</strong></td>
<td>0.179%</td>
<td>86.40</td>
<td>123.93</td>
<td>-35.55%</td>
<td>35.55</td>
<td>-15.62%</td>
</tr>
<tr>
<td><strong>COMPACTER</strong></td>
<td>0.073%</td>
<td>86.62</td>
<td>123.91</td>
<td>-35.57%</td>
<td>36.48</td>
<td>-13.41%</td>
</tr>
<tr>
<td><strong>COMPACTER++</strong></td>
<td>0.047%</td>
<td>86.67</td>
<td>118.35</td>
<td>-41.94%</td>
<td>30.96</td>
<td>-26.51%</td>
</tr>
</tbody>
</table>
Performance Evaluation: **BitFit**

- Performs worse than fine-tuning (-1.53 points).
  - Tuning only biases is not sufficient
- Lowest memory overhead and the fastest to train
  - Does not store intermediate activations.

<table>
<thead>
<tr>
<th>Model</th>
<th>Trained params/ per task</th>
<th>Avg</th>
<th>Memory (MB)</th>
<th>Δ%</th>
<th>Time/ Epoch (min)</th>
<th>Δ%</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>T5_BASE</strong></td>
<td>100%</td>
<td>86.5</td>
<td>167.99</td>
<td>—</td>
<td>42.13</td>
<td>—</td>
</tr>
<tr>
<td>Adapter</td>
<td>0.832%</td>
<td>85.78</td>
<td>124.02</td>
<td>-35.45%</td>
<td>31.81</td>
<td>-24.50%</td>
</tr>
<tr>
<td>Pfeiffer-Adapter</td>
<td>0.427%</td>
<td>86.32</td>
<td>118.4</td>
<td>-41.88%</td>
<td>28.19</td>
<td>-33.09%</td>
</tr>
<tr>
<td>AdapterDrop</td>
<td>0.494%</td>
<td>85.85</td>
<td>119.41</td>
<td>-40.68%</td>
<td>28.08</td>
<td>-33.35%</td>
</tr>
<tr>
<td>Adapter-LowRank</td>
<td>0.073%</td>
<td>85.82</td>
<td>123.8</td>
<td>-35.69%</td>
<td>32.71</td>
<td>-22.36%</td>
</tr>
<tr>
<td>Prompt Tuning</td>
<td>0.034%</td>
<td>75.95</td>
<td>222.27</td>
<td>24.42%</td>
<td>44.54</td>
<td>5.72%</td>
</tr>
<tr>
<td>Intrinsic-SAID</td>
<td>0.009%</td>
<td>85.45</td>
<td>285.40</td>
<td>41.14%</td>
<td>144.01</td>
<td>241.82%</td>
</tr>
<tr>
<td><strong>BitFit</strong></td>
<td>0.126%</td>
<td>84.97</td>
<td>102.31</td>
<td>-64.20%</td>
<td>27.36</td>
<td>-35.06%</td>
</tr>
<tr>
<td>PHM-Adapter</td>
<td>0.179%</td>
<td>86.40</td>
<td>123.93</td>
<td>-35.55%</td>
<td>35.55</td>
<td>-15.62%</td>
</tr>
<tr>
<td>Compacter</td>
<td>0.073%</td>
<td>86.62</td>
<td>123.91</td>
<td>-35.57%</td>
<td>36.48</td>
<td>-13.41%</td>
</tr>
<tr>
<td>Compacter++</td>
<td>0.047%</td>
<td>86.47</td>
<td>118.35</td>
<td>-41.94%</td>
<td>30.96</td>
<td>-26.51%</td>
</tr>
</tbody>
</table>
Performance Evaluation: \textsc{Adapter}-based methods

- Low memory-overhead and fast to train
- Generally perform worse than finetuning (exception: \textsc{Pfeiffer-Adapter})
  - \textsc{AdapterDrop}: Adapting lower layer of T5 is important.
  - \textsc{Adapter-LowRank} is not expressive enough.
- Order of magnitude more trainable parameters cf. \textsc{Compacter++}

<table>
<thead>
<tr>
<th>Model</th>
<th>Trained params/ per task</th>
<th>Avg</th>
<th>Memory (MB)</th>
<th>(\Delta%)</th>
<th>Time/Epoch (min)</th>
<th>(\Delta%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textsc{T5\textsubscript{BASE}}</td>
<td>100%</td>
<td>86.5</td>
<td>167.99</td>
<td>—</td>
<td>42.13</td>
<td>—</td>
</tr>
<tr>
<td>\textsc{Adapter}</td>
<td>0.832%</td>
<td>85.78</td>
<td>124.02</td>
<td>-35.45%</td>
<td>31.81</td>
<td>-24.50%</td>
</tr>
<tr>
<td>\textsc{Pfeiffer-Adapter}</td>
<td>0.427%</td>
<td>86.32</td>
<td>118.4</td>
<td>-41.88%</td>
<td>28.19</td>
<td>-33.09%</td>
</tr>
<tr>
<td>\textsc{AdapterDrop}</td>
<td>0.494%</td>
<td>85.85</td>
<td>119.41</td>
<td>-40.68%</td>
<td>28.08</td>
<td>-33.35%</td>
</tr>
<tr>
<td>\textsc{Adapter-LowRank}</td>
<td>0.073%</td>
<td>85.82</td>
<td>123.8</td>
<td>-35.69%</td>
<td>32.71</td>
<td>-22.36%</td>
</tr>
<tr>
<td>\textsc{Prompt Tuning}</td>
<td>0.034%</td>
<td>75.95</td>
<td>222.27</td>
<td>24.42%</td>
<td>44.54</td>
<td>5.72%</td>
</tr>
<tr>
<td>\textsc{Intrinsic-SAID}</td>
<td>0.009%</td>
<td>85.45</td>
<td>285.40</td>
<td>41.14%</td>
<td>144.01</td>
<td>241.82%</td>
</tr>
<tr>
<td>\textsc{BitFit}</td>
<td>0.126%</td>
<td>84.97</td>
<td>102.31</td>
<td>-64.20%</td>
<td>27.36</td>
<td>-35.06%</td>
</tr>
<tr>
<td>\textsc{PHM-Adapter}</td>
<td>0.179%</td>
<td>86.40</td>
<td>123.93</td>
<td>-35.55%</td>
<td>35.55</td>
<td>-15.62%</td>
</tr>
<tr>
<td>\textsc{Compacter}</td>
<td>0.073%</td>
<td>86.62</td>
<td>123.91</td>
<td>-35.57%</td>
<td>36.48</td>
<td>-13.41%</td>
</tr>
<tr>
<td>\textsc{Compacter++}</td>
<td>0.047%</td>
<td>86.47</td>
<td>118.35</td>
<td>-41.94%</td>
<td>30.96</td>
<td>-26.51%</td>
</tr>
</tbody>
</table>
Low-resource Fine-tuning

- Subsampling GLUE for varying sizes (100, 500, 1000, 2000, 4000).
- **Compacter++:**
  - Generalizes substantially better in resource-limited settings.
  - Offers a more effective fine-tuning in this regime.

![Graph showing average scores on GLUE for low-resource setting.](image)

**Figure:** Results on GLUE for low-resource setting.
Takeaways

**COMPACTER (++)**

- Is a light-weight fine-tuning method for large-scale language models.
- Generates adapter’s weights by summing Kronecker products between:
  - shared “slow” weights
  - “fast” rank-one matrices, specific to each adapter layer.
- Reduces the number of parameters substantially from $O(kd)$ to $O(k + d)$.
- Learns only 0.073% (0.047%) parameters, still:
  - Obtains comparable performance in a full-data setting.
  - Outperforms fine-tuning in data-limited scenarios.

Questions?

Please join our poster presentation during NeurIPS, 2021.


