

A Guide to Parameter-Efficient Fine Tuning

<https://arxiv.org/pdf/2303.15647.pdf>

Introduction

- In-context learning (learning from examples in prompt) has proved effective
- We always want more examples
- Parameter Efficient Fine-tuning (PEFT) may help us get better performance on specific tasks

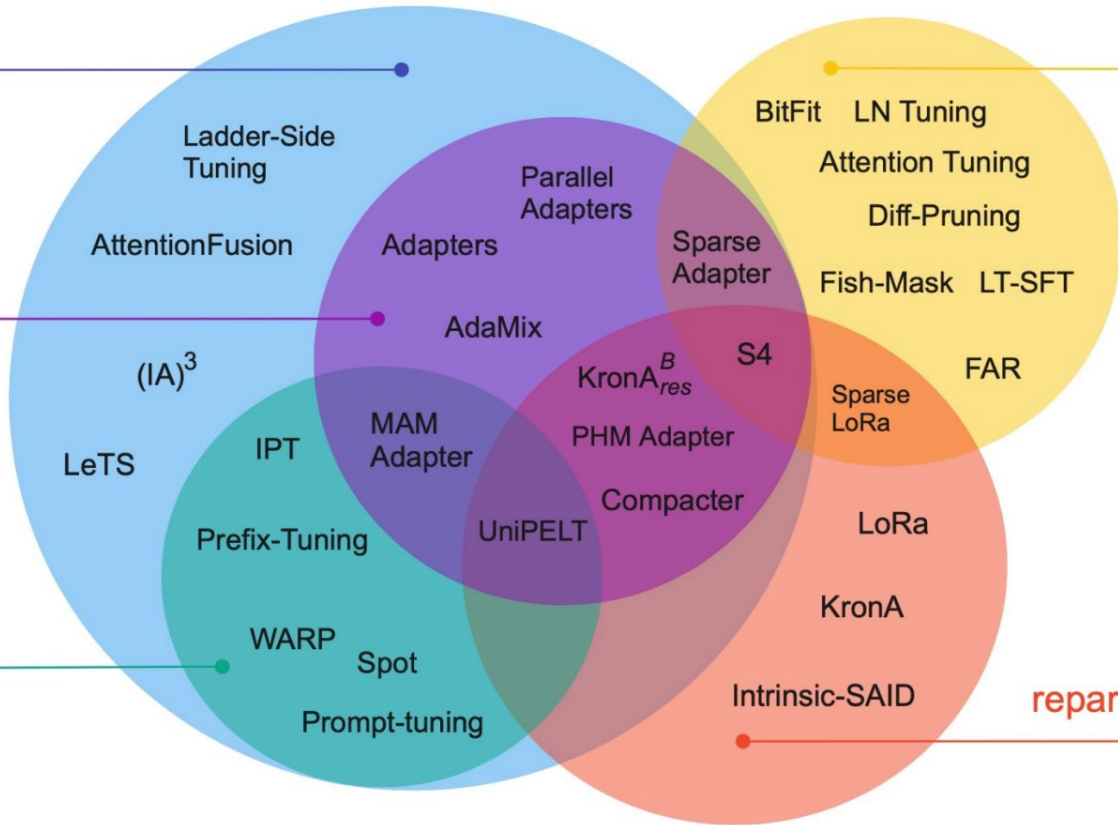
additive

selective

adapters

soft prompts

reparametrization-based



Additive methods

- Add new parameters or layers to the NN
- Major types
 - Adapters
 - Soft prompts
- Adapters
 - Introduce fully connected NN after Transformer sub-layers
- Soft prompts
 - Fine-tune a portion of the model's input embeddings
 - With normal prompting, it's all discrete because you use words
 - By fine-tuning, you can optimize in a continuous with real-valued numbers
 -

Selective methods

- Choose particular parameters to fine-tune
- Example:
 - Fine-tuning only a few top layers of a network

Reparameterization-based methods

- Use low-rank representations to minimize number of parameters that need fine-tuning
- Low-Rank Adaptation (LoRa)

Hybrid Methods

- Use combinations of the different methods
- MAM Adapter
 - Adapters + Prompt tuning
- UniPELT
 - MAM Adapter + LoRa

Axes of improvement

1. Storage efficiency
2. Memory efficiency
3. Computation efficiency
4. Accuracy
5. Inference overhead

Additive Methods

Adapters

- add fully-connected networks after attention and FFN layers

```
def transformer_block_with_adapter(x):  
    residual = x  
    x = SelfAttention(x)  
    x = FFN(x)    # adapter    ←  
    x = LN(x + residual)  
    residual = x  
    x = FFN(x)    # transformer FFN  
    x = FFN(x)    # adapter    ←  
    x = LN(x + residual)  
    return x
```

AdaMix

- Use multiple adapters, but use them as a mixture model
- Less memory efficient than Adapters, but you get better performance
 - Inference cost is the same

```
def transformer_block_with_adamix(x):  
    residual = x  
    x = SelfAttention(x)  
    x = LN(x + residual)  
    residual = x  
    x = FFN(x)  
    # adamix starts here  
    x = random_choice(experts_up)(x)  
    x = nonlinearity(x)  
    x = random_choice(experts_down)(x)  
    x = LN(x + residual)  
    return x
```

```
def consistency_regularization(x):  
    logits1 = transformer_adamix(x)  
    # second pass uses different experts  
    logits2 = transformer_adamix(x)  
    r = symmetrized_KL(logits1, logits2)  
    return r
```

Prompt tuning

- prepend the model input embeddings with a trainable tensor $P \in R^{l \times h}$
- l is the number of tokens
- h is the embedding dimension
- This soft prompt is trained via gradient descent
- More efficient than parameter tuning
- Better for bigger models (10B+)
- Higher inference cost though
- Good summary:
<https://magazine.sebastianraschka.com/p/understanding-parameter-efficient>

```
soft_prompt = torch.nn.Parameter( # Make tensor trainable
    torch.rand(num_tokens, embed_dim)) # Initialize soft prompt tensor

def input_with_softprompt(x, soft_prompt):
    x = concatenate([soft_prompt, x] # Prepend soft prompt to input
                    dim=seq_len)

    return x

model(input_with_softprompt(x))
```

Prefix tuning

- prepend the **every hidden state** with a trainable tensor

$$P \in R^{l \times h}$$

- l is the number of tokens
- h is the embedding dimension

Pseudocode for a single layer:

```
def transformer_block_for_prefix_tuning(x):  
    soft_prompt = FFN(soft_prompt)  
    x = concat([soft_prompt, x], dim=seq)  
    return transformer_block(x)
```

Prompt tuning vs prefix tuning

- Both methods came out at similar times, so hard to know exactly
- Prompt tuning is more parameter efficient, but you might capture more with prefix tuning
 - But prefix tuning might also overfit
- ￣_(\ツ)_/￣

Ladder-Side Tuning

- Trains a small transformer network on the side of the pre-trained network
- Pretrained network acts as a feature extractor, and is exempted from backprop during finetuning
- Only the side network goes through backprop

the output of the
corresponding layer of the
pre-trained network



```
def ladder_side_layer(x, h_pt):  
    h_pt = h_pt @ W_down # to x.shape  
    gate = sigmoid(alpha)  
    x = gate * x + (1 - gate) * h_pt  
    return transformer_block(x)  
  
def ladder_side_network(x):  
    with no_grad():  
        H_pt = pretrained_network(  
            x, return_all_hiddens=True  
        )  
    for i in range(layers):  
        layer = ladder_side_layers[i]  
        x = layer(x, H_pt[i])  
    return x
```

`alpha` is an input-independent trainable scalar gate

Selective Methods

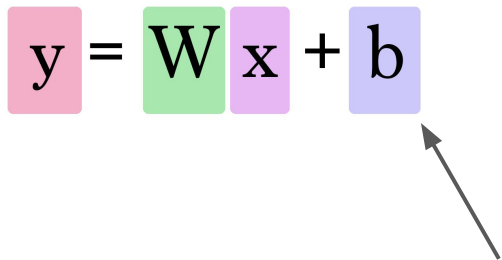
(IA)^3

- Rescale key, value, and hidden FFN activations
- Idea being that in self attention, we can use backprop to find better parameters for key & value
- Promising avenue --

```
def transformer_block_with_ia3(x):  
    residual = x  
    x = ia3_self_attention(x)  
    x = LN(x + residual)  
    residual = x  
    x = x @ W_1          # FFN in  
    x = l_ff * gelu(x)    # (IA)3 scaling  
    x = x @ W_2          # FFN out  
    x = LN(x + residual)  
    return x  
  
def ia3_self_attention(x):  
    k, q, v = x @ W_k, x @ W_q, x @ W_v  
    k = l_k * k  
    v = l_v * v  
    return softmax(q @ k.T) @ v
```

Freeze and Reconfigure (FAR)

- Only fine-tune the biases of the network

$$y = Wx + b$$
The equation $y = Wx + b$ is displayed with each term in a colored box: y is in a pink box, W is in a green box, x is in a purple box, and b is in a blue box. A grey arrow points from the bottom right towards the blue box containing b .

```
params = (p for n, p
           in model.named_parameters()
           if "bias" in n)
optimizer = Optimizer(params)
```

FishMask

- Sparse finetuning method that selects top-p parameters of the model based on their Fisher information
- The Fisher information measures the sensitivity of the distribution's likelihood function to changes in the parameter. If a small change in the parameter results in a large change in the likelihood function, the Fisher information is large, and we say that the data carries a lot of information about that parameter. Conversely, if the likelihood function barely changes with the parameter, the Fisher information is low, and the data doesn't tell us much about that parameter.

FishMask

$$\hat{F}_\theta = \frac{1}{N} \sum_{i=1}^N \mathbb{E}_{y \sim p_\theta(y|x_i)} (\nabla_\theta \log p_\theta(y|x_i))^2$$

- A given entry in \hat{F}_θ relates to the average of the square gradient of the model's output with respect to a given parameter. If a given parameter heavily affects the model's output, then its corresponding entry in \hat{F}_θ will be large, so we can reasonably treat \hat{F}_θ as an approximation of the importance of each parameter. - FishMask
- Longer proof and explanation here
- TLDR: figure out which parameters have highest fisher information, finetune those
- Computationally expensive

Reparameterization Methods

Intrinsic SAID

- The authors of this paper try to figure out why finetuning works, especially when these models have giant datasets, but finetuned models have much smaller datasets
- Authors argue that large pretrained models have low intrinsic dimension i.e. that they can be represented as a lower dimensional form when reparameterized
- They show this empirically by optimizing only 200 trainable parameters randomly projected back into the full space
 - Can tune a RoBERTa model to achieve 90% of the full parameter performance levels on MRPC.
-

LoRA

- Covered by Aneesh

KronA

- Similar to LoRA
 - Uses Kronecker product instead of matrix product
 - $A \otimes B$
- This yields a better rank per parameters tradeoff because the Kronecker product keeps the rank of the original matrices being multiplied. Or, in other words, $\text{rank}(A \otimes B) = \text{rank } A \cdot \text{rank } B$
 - Rank per parameters - can help assess the efficiency of parameter usage in a model. A model that achieves a better rank per parameter can capture more linearly independent features (i.e., has a higher rank) for the same number of parameters, potentially leading to better learning and generalization performance.
 - $\text{rank}(A \otimes B) = \text{rank } A \cdot \text{rank } B$
 - While $\text{rank}(AB) \leq \min\{\text{rank}(A), \text{rank}(B)\}$

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \otimes \begin{bmatrix} 0 & 5 \\ 6 & 7 \end{bmatrix} = \begin{bmatrix} 1 \begin{bmatrix} 0 & 5 \\ 6 & 7 \end{bmatrix} & 2 \begin{bmatrix} 0 & 5 \\ 6 & 7 \end{bmatrix} \\ 3 \begin{bmatrix} 0 & 5 \\ 6 & 7 \end{bmatrix} & 4 \begin{bmatrix} 0 & 5 \\ 6 & 7 \end{bmatrix} \end{bmatrix} = \left[\begin{array}{cc|cc} 1 \times 0 & 1 \times 5 & 2 \times 0 & 2 \times 5 \\ 1 \times 6 & 1 \times 7 & 2 \times 6 & 2 \times 7 \\ \hline 3 \times 0 & 3 \times 5 & 4 \times 0 & 4 \times 5 \\ 3 \times 6 & 3 \times 7 & 4 \times 6 & 4 \times 7 \end{array} \right] = \left[\begin{array}{cc|cc} 0 & 5 & 0 & 10 \\ 6 & 7 & 12 & 14 \\ \hline 0 & 15 & 0 & 20 \\ 18 & 21 & 24 & 28 \end{array} \right].$$

Hybrid Approaches

SparseAdapter

- Use the Adapter technique large hidden dimension for the added module and prune around 40% of the values at initialization

Algorithm 1: Pruning on Adapters

Require: adapter paramters w , sparse ratio s

- 1: $w \leftarrow \text{Initialization}(w)$
 - 2: $z = \text{score}(w)$
 - 3: Compute the s -th percentile of z as z_s
 - 4: $m \leftarrow \mathbb{1} [z - z_s \geq 0]$
 - 5: $\tilde{w} \leftarrow m \odot w$
-

Takeaways

- Generally three methods, and you can also mix and match the methods
- It's hard to know what's actually the best
 - Tradeoffs with each one (see the axes)
 - Little standardization in reporting
- Run a few different types given your model, and see what you end up finding best
- It really comes down to what you want to optimize for