Supplementary for GEMS: Generating Efficient Meta-Subnets

Varad Pimpalkhute  
TCS Research, Mumbai, India  
varad.p@tcs.com

Shruti Kunde  
TCS Research, Mumbai, India  
shruti.kunde@tcs.com

Rekha Singhal  
TCS Research, Mumbai, India  
rekha.singhal@tcs.com

A. Experimental Setup

All experiments are conducted in isolation on a dedicated MIG A100 GPU setup, with 30 GB RAM, 8vCPUS and 10GB GPU memory. For each experiment we record the model accuracy and time required for training. Each experiment has been repeated 3 times with different seeds to ensure sufficient randomness.

A.1. Architecture Details

The model architecture is similar to the original MAML [2] paper, i.e., 4 modules with 3 x 3 convolutions and 64 filters with a stride of 2, followed by batch normalization, a ReLU nonlinearity and 2 x 2 max-pooling. The MLP is a fully connected network, where each layer consists of 2N hidden units, where N is the number of layers of the base learner network. The ReLU activation function is placed between the MLP layers.

A.2. Dataset Details

We have used 4 quasi-benchmark datasets (CuBirds, VGGFlowers, Aircraft, Fungi) from the field of meta-learning for our experimentation purpose. The CU-Birds dataset contains 11,788 images of 200 bird species. The data is split into 200 classes that are divided into 100, 50 and 50 for meta-training, meta-validation and meta-testing respectively. VGGFlower is a dataset consisting of 102 flower categories. Each class consists between 40-258 images. The Fungi dataset contains 1394 species of fungi. It has 85578 training images, 4182 validation images and 9758 testing images. The table below provides a detailed list of hyper-parameters that were used when setting up the experiments.

A.3. Training Details

The default setting for adaption steps is 5 and fast LR is 0.01. We use 0.5 as the random initialization for sparsity in the MMSUP experiments and the value for sparsity is learned during training. These settings are used in our experiment setup for comparing performance of MMSUP and BMP to MAML (base line) and Multi-MAML. The varying sparsity, inner loop learning rate and adaptation steps values depicted in Table 1 are used for setting up experiments conducted as a part of ablation studies outline in the main paper.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iterations</td>
<td>30000</td>
</tr>
<tr>
<td>Meta-batch size</td>
<td>5</td>
</tr>
<tr>
<td>Shots</td>
<td>1</td>
</tr>
<tr>
<td>Ways</td>
<td>5</td>
</tr>
<tr>
<td>Adaptation steps</td>
<td>1,2,3,4,5</td>
</tr>
<tr>
<td>Fast LR</td>
<td>0.1, 0.5, 0.05, 0.001, 0.005</td>
</tr>
<tr>
<td>Meta-LR</td>
<td>0.0001</td>
</tr>
<tr>
<td>Random seeds</td>
<td>21, 42, 56</td>
</tr>
<tr>
<td>Sparsity%</td>
<td>10, 20, ..., 90</td>
</tr>
</tbody>
</table>

B. Additional Results

Table 2 depicts a snapshot of additional combinations of datasets on which we tested performance of BMP and MMSUP.

C. Analysis of GEMS

C.1. Analysis of convergence of MMSUP

In this section, we will study whether there is any positive learning in the network after each training iteration. We want to prove that

$$\mathcal{L}_{n+1} < \mathcal{L}_{n}$$

where, n is the current training iteration.

Let us consider a neuron whose weights are getting masked or unmasked after every iteration. Masking depends on the sparsity value of the layer, i.e., if sparsity is k, then top k weights will be fed to the neuron. Consider two weights – \( W_i \) and \( W_j \). In \( n^{th} \) iteration, \( W_j \) is fed as input to the neuron whereas \( W_i \) is masked. While, in the \( n + 1^{th} \) iteration, \( W_i \) is fed as input to the neuron whereas \( W_j \) is masked. This signifies that,


Thus, during backpropagation, we can say that the value of gradients of \(W_i\) is greater than that of \(W_j\), which brings us to the following equation:

\[
-\frac{\delta L}{\delta I_p} W_i Z_i > -\frac{\delta L}{\delta I_p} W_j Z_j \tag{1}
\]

where, \(I_p\) is the input and \(Z_i\) is the activation function for the \(i^{th}\) weight. Hence, after swapping the weights/edges of the neural network, the equation of the input \(I_p\) changes from

\[
I_p = \sum W_k Z_k + W_i Z_i
\]

to

\[
\bar{I}_p = \sum W_k Z_k + W_j Z_j
\]

Hence, change in the input is given by:

\[
I_p - \bar{I}_p = W_j Z_j - W_i Z_i \tag{2}
\]

We now compute the change in the loss from iteration \(n\) to iteration \(n+1\).

\[
L(\bar{I}_p) = L(I_p) + (\bar{I}_p - I_p)
\]

Expanding the Taylor’s series and approximating it, we get,

\[
L(\bar{I}_p) = L(I_p) + \frac{\delta L}{\delta I_p} (\bar{I}_p - I_p)
\]

Thus, substituting Equation 2 in the above equation, we get,

\[
L(\bar{I}_p) = L(I_p) + \frac{\delta L}{\delta I_p} (W_j Z_j - W_i Z_i) \tag{3}
\]

From Equation 1 and Equation 3, we conclude that, \(L(\bar{I}_p) < L(I_p)\). We can henceforth conclude that as the network learns on the subnetwork, it also converges towards the global minima.

## D. Resources

The results reported in the paper are produced using open source and free software. We build up on learn2learn\(^1\) to make custom modules for BMP and MMSUP keeping in spirits with the PyTorch\(^9\) framework. We also made use of Numpy\(^4\) and Pandas\(^7\) libraries for building custom few-shot learning dataset modules and sparsity in the network using lottery ticket hypothesis. All plots were generated using Matplotlib\(^5\).

All the final experiments were run in a Linux environment. For producing and debugging, we made use of Google Colab Pro that also ran on a Linux environment. The datasets were hand-picked from public domain datasets allowing usage for research purposes. cub200 We made use of CUBirds\(^10\), VGGFlowers\(^8\), FGVC Aircraft\(^6\) and FGVC Fungi\(^3\) datasets.

### E. PyTorch Code Snippets

In this section, we walkthrough some of the important modules required for implementation of BMP and MMSUP approaches. We present pseudo codes on how to reproduce the algorithms in PyTorch.

```python
class BinaryMask(torch.autograd.Function):
    def __init__(self):
        super(BinaryMask, self).__init__()

    @staticmethod
    def forward(ctx, input):
        return grad without computing gradients

    @staticmethod
    def backward(ctx, grad):
        return grad without computing gradients

# Straight Through Estimator (STE)
# Compute Binary mask --
# 1: Values greater than zero
# 0: Values less than or equal to zero

class STE(torch.autograd.Function):
    def __init__(self):
        super(STE, self).__init__()

    @staticmethod
    def forward(ctx, input):
        return grad without computing gradients

    @staticmethod
    def backward(ctx, grad):
        return grad without computing gradients

# Number of adaptation steps
for step in range(adaptation_steps):
    # Compute loss on input batch
    train_error = loss(learner(data_batch),
    # Get backbone parameters
    param_dict = param_dict.values(), retain_graph=True)
    # Stack mean of weights and gradient per layer
    task_embedding = # Stack mean of weights and
```

\(^1\)learn2learn is an open-source PyTorch library for meta-learning research https://github.com/learnables/learn2learn.

---

<table>
<thead>
<tr>
<th>Train Dist</th>
<th>Test Dist</th>
<th>Meta-learning Architectures</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUB200</td>
<td>CUB</td>
<td>MAML</td>
</tr>
<tr>
<td>+ Aircraft</td>
<td>Aircraft</td>
<td>0.468 7.07 0.533 15.51</td>
</tr>
<tr>
<td>VGG102</td>
<td>VGG102</td>
<td>Multi-MAML</td>
</tr>
<tr>
<td>+ Aircraft</td>
<td>Aircraft</td>
<td>0.656 7.05 0.728 14.92</td>
</tr>
<tr>
<td>Fungi</td>
<td>Fungi</td>
<td>BMP</td>
</tr>
<tr>
<td>+ Aircraft</td>
<td>Aircraft</td>
<td>0.372 8.6 0.424 16.92</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MMSUP</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.372 8.6 0.424 16.92</td>
</tr>
</tbody>
</table>

---
layer_pred = regularizer(torch.mul(task_embedding, inp_embedding))
# Use previously learned knowledge
# to compute binary mask
mask = BinaryMask.apply(layer_pred)
# Update weights of the backbone
learner.adapt(train_error, mask)

Code Listing 2. Inner Loop Optimization

class ComputeMask(torch.autograd.Function):
    @staticmethod
    def forward(ctx, params, sparsity):
        # k is the sparsity of the layer
        # 1: For top k parameters
        # 0: Rest of the parameters
        scores = self.weight.detach()
        subnet = ComputeMask.apply(scores, sparsity)
        return F.conv2d(x, subnet, self.bias, self.stride, self.padding, self.dilation, self.groups)

    @staticmethod
    def backward(ctx, g):
        # Return gradient as it is

Code Listing 3. Generate Sparsity Mask

Code Listing 4. Compute Sparse Network

References