Betty: An Automatic Differentiation Library for Generalized Meta Learning

A simple, scalable, and modular meta learning library built upon PyTorch

Introduction

Meta Learning / Multilevel Optimization

What is (Generalized) Meta Learning?
Find the optimal decision based on the prediction of how your decision affects other agents, whose decisions will again affect your state.

\[ \lambda = \min \; f(\lambda; \theta) \]
\[ \theta = \min \; g(\theta; \lambda) \]

Implementation Difficulties:
Gradient computation is achieved by composing best-response Jacobsian via the chain rule, requiring both programming and mathematical proficiency.

Memory/Compute Bottleneck:
Best-response Jacobian algorithms are memory and compute intensive, as they require multiple forward/backward computations and oftentimes second-order gradient (i.e. Hessian) information.

Betty makes meta learning ...
- Easy by abstracting away gradient computation with autodiff.
- Scalable with various systems support, such as distributed training.
- Modular as users can try out different algorithmic and systems design choices with one-liner changes in the config.
- Systematic by allowing the same programming interface for diverse meta learning applications.

How does Betty work?
Gradient Computation:
\[ \frac{\partial \mathcal{L}}{\partial \theta} = \sum_{i=1}^{N} \frac{\partial \mathcal{L}}{\partial \theta_i} = \sum_{i=1}^{N} \nabla_{\theta_i} \mathcal{L} \]

Dataflow Graph Interpretation:

How to use Betty?
With Betty, users simply need to do two things to implement any meta learning applications:

1. Define each optimization problem using the Problem class:

```python
from betty.problems import ImplicitProblem
from betty.configs import ImplicitConfig

# set up module, optimizer, data loader (i.e. (1)-(3))
class Module, class_optimizer, class_data_loader = setup_classification()

class Classifier(ImplicitProblem):
    # set up loss function
    def training_step(self, batch):
        inputs, labels = batch
        outputs = self.module(inputs)
        loss = F.cross_entropy(outputs, labels)
        return loss

# set up problem configuration
clf_config = Config(type='darts', unroll_steps=1, log_step=100)

clf_prob = Classifier(name='clf', module=clf_module, optimizer=clf_optimizer, train_data_loader=clf_data_loader, config=clf_config)
```

name will be used to access other problems. For example, other problems can access the Classifier problem via self-classifier

2. Define the hierarchical problem structure using the Engine class:

```python
from betty.ends import ImplicitEnd

# set up all involved problems
problems = [clf_prob, inp_prob]

# set up upper-to-lower and lower-to-upper dependencies
u2l = [inp_prob : clf_prob]
l2u = [clf_prob : inp_prob]
dependencies = [u2l, 'u2l', 'l2u', 12u]

# set up engine configuration
engine_config = EngineConfig(train_iters=10000, valid_step=100)

# instantiate Engine class
engine = Engine(problems, dependencies, dependencies, engine_config)

# execute multilevel optimization
engine.run()
```

Applications

1. Data Reweighting for Class Imbalance
We re-implemented Meta-Weight-Net (MWN), where MWN learns to assign larger/smaller weights to samples from rare/common classes.

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>IF 20</th>
<th>IF 100</th>
<th>IF 50</th>
<th>Memory</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>MWN (original)</td>
<td>67.91</td>
<td>57.21</td>
<td>60.06</td>
<td>2381MB</td>
<td>35.8m</td>
</tr>
<tr>
<td>MWN (step=1)</td>
<td>71.06</td>
<td>64.83</td>
<td>64.24</td>
<td>2425MB</td>
<td>67.4m</td>
</tr>
<tr>
<td>MWN (step=2)</td>
<td>66.45</td>
<td>70.52</td>
<td>75.90</td>
<td>2415MB</td>
<td>67.1m</td>
</tr>
<tr>
<td>MWN (step=3)</td>
<td>70.56</td>
<td>80.45</td>
<td>83.11</td>
<td>2501MB</td>
<td>28.5m</td>
</tr>
<tr>
<td>MWN (step=4)</td>
<td>75.56</td>
<td>89.94</td>
<td>90.98</td>
<td>6861MB</td>
<td>52.0s</td>
</tr>
</tbody>
</table>

Table 1: Small-scale MWN experiments w/ CIFAR10 & ResNet30. IF denotes an imbalance factor.

2. Reweighting & Correcting Corrupted Labels
While weak supervision significantly reduces the data labeling cost, generated labels are generally noisy. We meta-learn to reweight and correct potentially wrong labels.

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>IF 20</th>
<th>IF 50</th>
<th>IF 100</th>
<th>Memory</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>69.39</td>
<td>87.49</td>
<td>105.1MB</td>
<td>1.0x</td>
<td></td>
</tr>
<tr>
<td>MWN</td>
<td>85.79</td>
<td>91.04</td>
<td>105.1MB</td>
<td>1.01x</td>
<td></td>
</tr>
<tr>
<td>+ distributed</td>
<td>89.49</td>
<td>90.98</td>
<td>6861MB</td>
<td>1.02x</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Large-scale MWN experiments w/ SST & BIT-base.

3. Domain Adaptation for Pretraining & Finetuning
While pretraining & finetuning is a popular ML technique, certain pretraining data may be detrimental for the downstream task. We meta-learn to reweight pretraining data to maximize positive transfer.

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>CR</th>
<th>AR</th>
<th>AV</th>
<th>PV</th>
<th>PR</th>
<th>Rv</th>
<th>Rm</th>
<th>CI</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>67.45</td>
<td>79.74</td>
<td>71.78</td>
<td>77.25</td>
<td>65.33</td>
<td>1.00x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+ BWT</td>
<td>70.56</td>
<td>82.79</td>
<td>77.18</td>
<td>77.25</td>
<td>65.33</td>
<td>1.00x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+ BWT &amp; CFT</td>
<td>66.78</td>
<td>83.16</td>
<td>77.50</td>
<td>67.43</td>
<td>61.52</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Wrench benchmark results. BWT stands for reweighting and CFT for correction.

4. Differentiable Neural Architecture Search
Finally, we re-implement DARTS to further demonstrate the general applicability of Betty to diverse meta learning applications.

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Accuracy</th>
<th>Params</th>
<th>Memory</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random Search</td>
<td>96.71</td>
<td>3.2M</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DARTS (original)</td>
<td>97.24</td>
<td>3.3M</td>
<td>10463MB</td>
<td>25.4h</td>
</tr>
<tr>
<td>DARTS(step=5)</td>
<td>97.30</td>
<td>3.5M</td>
<td>10463MB</td>
<td>23.6h</td>
</tr>
<tr>
<td>DARTS(step=10)</td>
<td>97.22</td>
<td>3.2M</td>
<td>10463MB</td>
<td>28.5h</td>
</tr>
</tbody>
</table>

Table 4: Domain Adaptation for Pretraining & Finetuning results.