# ASDL: A Unified Interface for Gradient Preconditioning in PyTorch

Kazuki Osawa[1], Satoki Ishikawa[2], Rio Yokota[2], Shigang Li[3], and Torsten Hoefler[1]

[1]: ETH Zurich (Scalable Parallel Computing Lab), [2]: Tokyo Institute of Technology, [3]: Beijing University of Posts and Telecommunications

## Gradient Preconditioning in Deep Learning

Gradient-based optimization

$$\theta_{t+1} \leftarrow \underbrace{\theta_t}_{\text{Parameter}} - \eta \overbrace{\underbrace{P_t}_{\text{Preconditioning matrix}} \underbrace{g_t}_{\text{Gradient}}}^{\text{Preconditioned gradient}}$$

*How to deal with nonconvexity, stochasticity, and high dimensionality in Deep Learning?*

### Background & problem: A diverse set of gradient preconditioning methods

**1. Curvature matrix $C$**

- **Loss sharpness**
  - Hessian $H$
  - Absolute Hessian $H_{|\lambda|}$
  - BFGS Hessian $\hat{H}_{\text{bfgs}}$
  - Gauss-Newton matrix $G$
- **Gradient covariance**
  - Fisher information matrix $F$
  - FIM est. by MC samples $\hat{F}_{n\text{mc}}$
- **Gradient 2nd moment**
  - Empirical Fisher $\hat{F}_{\text{emp}}$
  - Batched empirical Fisher $\hat{F}^{\text{batch}}_{\text{emp}}$

**2. Representation of $C$**

- **Full**
  - Dense/sparse/low-rank
  - Matrix-free/Gram
- **Layer-wise** (block-diagonal)
  - Dense/sparse/low-rank
  - Gram/Kronecker-factored
- **Unit-wise** (block-diagonal)
  - Dense/sparse/low-rank
  - Gram
- **Element-wise** (diagonal)
  - Dense/sparse

**3. Solver for $Pg \approx C^{-1}g$**

- **Local iterative** (matrix-free)
  - Conjugate gradient
  - Krylov subspace
  - Neumann series
- **Global iterative**
  - BFGS
  - Learning by SGD
- **Local/global direct**
  - Cholesky inverse/solve
  - Eigendecomposition
  - SMW formula

"KF-io": input-output Kronecker-factored. "KF-dim": dimension-wise Kronecker-factored. "RR": rank reduction. "SMW": Sherman-Morrison-Woodbury formula. "L": local = one mini-batch at one time step. "G": global = multiple mini-batches at multiple time steps. "iter": iterative.

| Method | 1. Curvature matrix $C$ | | 2. Representation of $C$ | | 3. Solver for $Pg \approx C^{-1}g$ | |
|---|---|---|---|---|---|---|
| | type | matrix | granularity | format | type | key operations |
| LiSSA (Agarwal et al., 2017) | sharpness | $H$ | full | dense | G iter | Neumann series |
| PSGD (Li, 2018) | sharpness | $H_{\lvert\lambda\rvert}$ | full | dense | G iter | triangular solve & SGD |
| Neumann optimizer (Krishnan et al., 2017) | sharpness | $H$ | full | matrix-free | L iter | Neumann series |
| Hessian-free (Martens, 2010) | sharpness | $H, G$ | full | matrix-free | L iter | conjugate gradient |
| KSD (Vinyals & Povey, 2011) | sharpness | $H, G$ | full | matrix-free | L iter | Krylov subspace method |
| L-BFGS (Liu & Nocedal, 1989) | sharpness | $\hat{H}_{\text{bfgs}}$ | full | matrix-free | G iter | approx. BFGS |
| SMW-GN (Ren & Goldfarb, 2019) | sharpness | $G$ | full | Gram, RR | L direct | SMW inverse |
| SMW-NG (Ren & Goldfarb, 2019) | grad $2^{\text{nd}}$m | $\hat{F}_{\text{emp}}$ | full | Gram, RR | L direct | SMW inverse |
| TONGA (Roux et al., 2008) | grad $2^{\text{nd}}$m | $\hat{F}_{\text{emp}}$ | full | Gram, RR | G direct | SMW solve & eigendecomp. |
| M-FAC (Frantar et al., 2021) | grad $2^{\text{nd}}$m | $\hat{F}^{\text{batch}}_{\text{emp}}$ | full | Gram, RR | G direct | SMW solve |
| GGT (Agarwal et al., 2019) | grad $2^{\text{nd}}$m | $(\hat{F}^{\text{batch}}_{\text{emp}})^{1/2}$ | full | Gram, RR | G direct | SMW solve |
| FANG (Grosse & Salakhutdinov, 2015) | grad cov | $\hat{F}_{n\text{mc}}$ | full | sparse | L/G direct | incomplete Cholesky |
| ★ PSGD (KF) (Li, 2018) | sharpness | $H_{\lvert\lambda\rvert}$ | layer | KF-io | G iter | triangular solve & SGD |
| ★ K-BFGS (Goldfarb et al., 2021) | sharpness | $\hat{H}_{\text{bfgs}}$ | layer | KF-io | G iter | BFGS |
| ★ K-FAC (Martens & Grosse, 2015) | grad cov, $2^{\text{nd}}$m | $\hat{F}_{n\text{mc}}, \hat{F}_{\text{emp}}$ | layer | KF-io | L/G direct | Cholesky inverse |
| KFLR (Botev et al., 2017) | grad cov | $F$ | layer | KF-io | L/G direct | Cholesky inverse |
| KFRA (Botev et al., 2017) | grad cov, $2^{\text{nd}}$m | $\hat{F}_{n\text{mc}}, \hat{F}_{\text{emp}}$ | layer | KF-io | L/G direct | Cholesky inverse & recursion |
| EKFAC (George et al., 2018) | grad cov, $2^{\text{nd}}$m | $\hat{F}_{\text{emp}}$ | layer | KF-io | L/G direct | eigendecomp. (or SVD) |
| SKFAC (Tang et al., 2021) | grad cov, $2^{\text{nd}}$m | $\hat{F}_{1\text{mc}}, \hat{F}_{\text{emp}}$ | layer | KF-io, RR | L direct | SMW inverse & reduction |
| ★ SENG (Yang et al., 2021) | grad $2^{\text{nd}}$m | $\hat{F}_{\text{emp}}$ | layer | Gram, RR | L/G direct | SMW inverse & sketching |
| TNT (Ren & Goldfarb, 2021) | grad cov, $2^{\text{nd}}$m | $\hat{F}_{n\text{mc}}, \hat{F}_{\text{emp}}$ | layer | KF-dim | L direct | Cholesky inverse |
| ★ Shampoo (Gupta et al., 2018) | grad $2^{\text{nd}}$m | $(\hat{F}^{\text{batch}}_{\text{emp}})^{1/2}$ | layer | KF-dim | G direct | eigendecomp. |
| unit-wise NG (Ollivier, 2015) | grad cov, $2^{\text{nd}}$m | $\hat{F}_{n\text{mc}}, \hat{F}_{\text{emp}}$ | unit | dense | L/G direct | Cholesky inverse |
| TONGA (unit) (Roux et al., 2008) | grad $2^{\text{nd}}$m | $\hat{F}_{\text{emp}}$ | unit | Gram, RR | G direct | SMW solve & eigendecomp. |
| AdaHessian (Yao et al., 2020b) | sharpness | $H_{\lvert\lambda\rvert}$ | element | dense | G direct | element-wise division |
| SFN (Dauphin et al., 2014) | sharpness | $H_{\lvert\lambda\rvert}$ | element | dense | L/G direct | element-wise division |
| Equilibrated SGD (Dauphin et al., 2015) | sharpness | $H_{\lvert\lambda\rvert}$ | element | dense | L/G direct | element-wise division |
| AdaGrad (Duchi et al., 2011) | grad $2^{\text{nd}}$m | $(\hat{F}^{\text{batch}}_{\text{emp}})^{1/2}$ | element | dense | G direct | element-wise division |
| Adam (Kingma & Ba, 2015) | grad $2^{\text{nd}}$m | $(\hat{F}^{\text{batch}}_{\text{emp}})^{1/2}$ | element | dense | G direct | element-wise division |

★: methods to be analyzed in this study

- ✗ Each requires algorithm-specific and complex implementations.
- ✗ The compute performance, prediction accuracy, and feasibility (time and memory) are highly dependent on neural network architectures and specific training settings.

## Automatic Second-order Differentiation Library (ASDL)

### Our solution: A unified interface by ASDL

- ✓ ASDL offers various implementations and a unified interface for gradient preconditioning in PyTorch (an automatic-differentiation library).
- ✓ ASDL enables an easy integration of gradient preconditioning into a training with procedures that are algorithm-independent and as simple (same logical structure) as the standard training pipeline (see the figure on the right.)
- ✓ ASDL works with arbitrary deep neural networks defined with basic building blocks (e.g., nn.Linear, nn.Conv2d, nn.BatchNormNd, nn.LayerNorm, nn.Embedding) in PyTorch.

ASDL enables flexible switching and structured comparison of gradient preconditioning methods in DL

Standard training pipeline in PyTorch

```
For x, t in data_loader:
  optimizer.zero_grad()

  # Compute mini-batch gradient g
  y = model(x)
  loss = F.cross_entorpy(y, t)
  loss.backward()

  optimizer.step()
```

replaceable

*Unified interface to compute preconditioned mini-batch gradient $Pg$*

```
gm = XXXGradientMaker(model, XXXGradientConfig())

dummy_y = gm.setup_model_call(model, x)
gm.setup_loss_call(F.cross_entropy, dummy_y, t)
y, loss = gm.forward_and_backward()
```

equivalent to

```
# PSGD [PsgdGradientMaker]
y = model(x)
loss = F.cross_entropy(y, t)
if step % interval == 0:
  grads = torch.autograd.grad(
            loss, params,
            create_graph=True)
  vs = [torch.randn_like(p)
          for p in params]
  Hvs = torch.autograd.grad(
          grads, params,
          grad_outputs=vs)
  update_preconditioner(vs, Hvs)
else:
  loss.backward()
precondition()
```

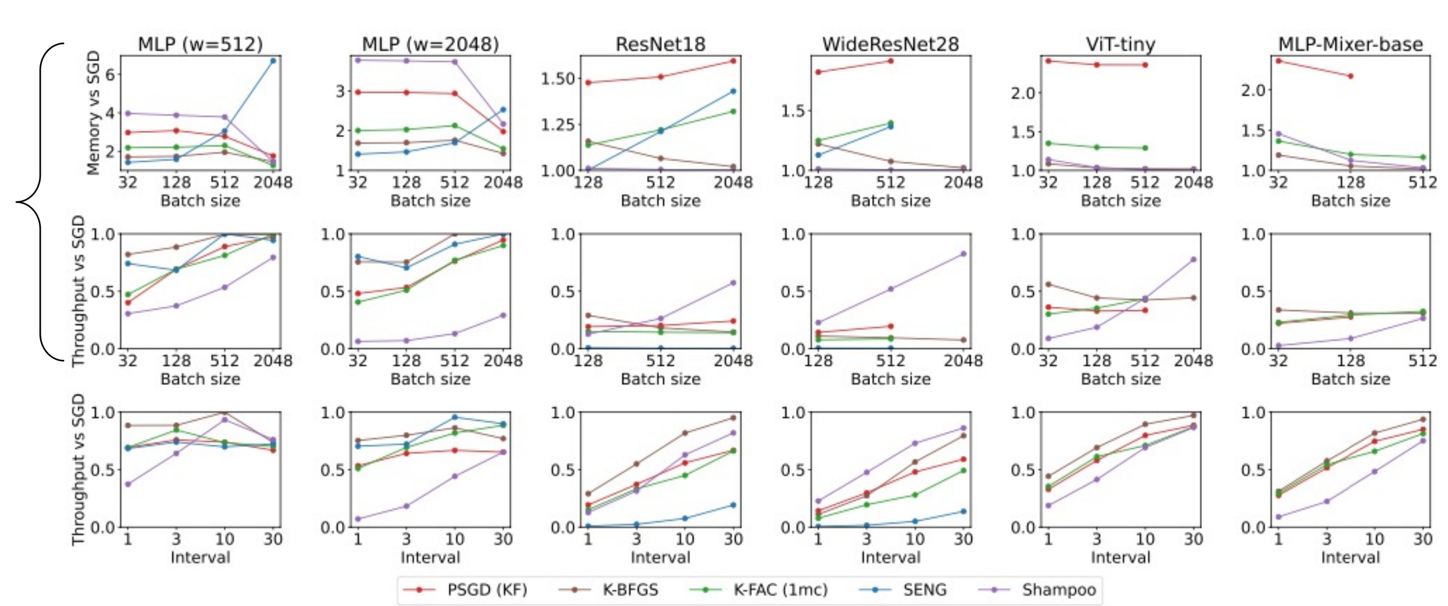
```
# K-FAC (1mc) [KfacGradientMaker]
if step % interval == 0:
  with extend(operations):
    y = model(x)
    loss = F.cross_entropy(y, t)
    p = F.softmax(y)
    log_p = F.log_softmax(y)
    with torch.no_grad():
     t_mc = Categorical(p).sample()
    nll = F.nll_loss(log_p, t_mc)
    nll.backward(retain_graph=True)
  update_preconditioner()
else:
  y = model(x)
  loss = F.cross_entropy(y, t)
loss.backward()
precondition()
```

```
# SENG [SengGradientMaker]
if step % interval == 0:
  with extend(operations):
    y = model(x)
    loss = F.cross_entropy(
        y, t, reduction='sum')
    loss.backward()
    truncate_and_sketch()
    calculate_gram_inv()
    loss /= data_size
else:
  y = model(x)
  loss = F.cross_entropy(
        y, t, reduction='mean')
  loss.backward()
precondition()
```

...

**Unified interface for gradient preconditioning in PyTorch.** XXXGradientMaker ("XXX": algorithm name), offered by ASDL, hides *algorithm-specific* and *complex* operations for Pg in a *unified* way. For training without gradient preconditioning, GradientMaker computes g with the same interface (i.e., no need to switch scripts).

## Case Studies with ASDL




The ratio of **peak memory** (>= 1) (top) and **throughput [image/s]** (<= 1) (middle, bottom) of gradient preconditioning methods compared to SGD with various mini-batch sizes B and matrix (C and P) update intervals T, measured on a NVIDIA A100 GPU. For the middle row, T=1. For the bottom row, B=128. Missing points are due to the GPU memory limitation.


**Sensitivity of the mini-batch size and matrix update interval** to the test accuracy (the best value among different learning rates for each pair is shown). The type of the solver ("Global" or "Local") is indicated at the top of each column. For SENG at ViT-tiny, the plot is not shown because it is not feasible with large mini-batch sizes and only B=32 results are available.

**The test accuracy** for models achieving the best validation accuracy. For each task, the best accuracy is bolded. "w": width. For ResNet18, the results with 20 and 100 epochs are shown (the number of epochs is fixed for the others). SENG consumes lots of memory and is infeasible with MLP-Mixer-base.

④

| Method | MNIST | | | CIFAR-10 | | | |
|---|---|---|---|---|---|---|---|
| | MLP (w=128) | MLP (w=512) | MLP (w=2048) | ResNet18 | WideResNet28 | ViT-tiny | MLP-Mixer-base |
| SGD | **98.9** | 99.1 | **99.2** | 91.2 / 95.7 | 96.7 | 97.8 | 97.2 |
| AdamW | 98.7 | 99.0 | 99.1 | 89.9 / 94.8 | 96.0 | 97.9 | **97.7** |
| PSGD (KF) | **98.9** | 99.1 | **99.2** | 93.3 / **96.2** | 96.6 | **98.0** | 97.5 |
| K-BFGS | 98.7 | 98.9 | 99.0 | 91.4 / 95.7 | 96.5 | 97.7 | 97.5 |
| K-FAC (1mc) | 98.8 | **99.2** | **99.2** | **93.6** / 96.1 | **96.9** | 97.4 | **97.7** |
| SENG | 98.8 | 99.0 | 99.1 | 91.6 / 95.8 | 96.6 | 97.7 | - |
| Shampoo | 98.8 | 99.1 | **99.2** | 92.5 / 96.1 | **96.9** | **98.0** | 97.4 |

### ❖ Key observations

① SENG achieves a high throughput w/ a low memory cost w/ a small mini-batch (and vice versa). For PSGD, K-BFGS, K-FAC, and Shampoo, memory and throughput ratios improve w/ a large mini-batch (Shampoo is particularly slow for most networks otherwise).

② Increasing the matrix update interval significantly improves the throughput, but the degree of speedup depends on methods.

③ "Global" methods (PSGD, K-BFGS, Shampoo) tend to perform better w/ a smaller mini-batch size while a "Local" one (K-FAC) tends to perform better w/ a larger mini-batch size.

④ The best test accuracy for each task is achieved by one of the gradient preconditioning methods, but the best performing method depends on the task.

https://github.com/kazukiosawa/asdl