A Supplementary material

A.1 Experiments

Here we provide additional details on the experiments in the main paper. All of the experiments are performed with Pytorch `1.7.0` [1] on Nvidia Titan X GPU with the CUDA version `10.1`. The results and error bars are reported over 8 fixed random seeds `[1-8]`.

A.1.1 Retrieving symmetries from synthetic regression

To fit the regression function we employ the multi-layer perceptron that consists of linear layers followed by Swish [2] activations with the following dimensions: $(10 \times 32) \rightarrow 3 * (32 \times 32) \rightarrow (32 \times 1)$. The network takes the input coordinates and outputs the predicted function value. We use a mean squared error loss to supervise the model. We utilize Adam optimizer [3] with the learning rate of 10^{-3} and train the network for a total of $min(900000/N, 1000)$ epochs, where N is the size of the training dataset. We found such a training schedule is enough for convergence.

To compute the mean symmetry variance per training set size, we average the last 10 singular values of the network polarization matrix.

A.1.2 Retrieving symmetries from rotation-MNIST

We employ two models: shallow (2-layer) and deep (6-layer) perceptrons with 40000 and 160000 parameters respectively. The hidden dimensions for shallow and deep networks are 47 and 116 respectively. Both models are trained for 300 epochs with Adam optimizer with the learning rate set to 10^{-3}.

The symmetries are extracted from the models with the highest validation accuracy over 300 training epochs. Additionally, to account for a possible difference in the magnitudes of the network outputs, we normalize the network output logits to be unit L2 norm.

A.1.3 Symmetries in networks with different configurations

To evaluate networks with various configurations (width, depth, number of parameters), we consider the following family of architectures: $(784 \times hdim) \rightarrow p * (hdim \times hdim) \rightarrow (hdim \times 10)$, where p stands for a number of hidden layers and $hdim$ is a dimension of a hidden layer. Given the required number of parameters and the number of hidden layers, we can calculate the hidden dimension of the network. By varying the number of parameters and number of hidden layers we create wide and deep networks.

We follow the same procedure as in A.1.2 to train the models and extract the symmetries.

A.2 LieGG implementation

We provide the PyTorch implementation of the network polarization matrix computation used in the synthetic regression and rotation-MNIST experiments. The symmetries are retrieved by performing a singular-value decomposition of the network polarization matrix as described in the Method section of the paper.

► synthetic regression:

```
def polarization_matrix(model, data, dim = 5):
    # data: torch.FloatTensor(B, 2*dim)

    B = data.shape[0]

    data.requires_grad = True
    data.retain_grad()

    # compute network grads
    model.eval()
```

```
    y_pred = model(data)
    y_pred.backward(gradient=torch.ones_like(y_pred))

    # get grads and data per input dimension
    dF_1 = data.grad[...,:dim].view(B, dim, 1)
    data_1 = data[...,:dim].view(B, 1, dim)

    dF_2 = data.grad[...,dim:].view(B, dim, 1)
    data_2 = data[...,dim:].view(B, 1, dim)

    # collect into the network polarization matrix
    C = torch.bmm(dF_1, data_1) + torch.bmm(dF_2, data_2)

    return C
```

► **rotation-MNIST:**

```
def polarization_matrix_R2(model, data):
    # LieGG implementation with the groups acting on R^2
    # data: torch.FloatTensor(B, 28, 28)

    B, H, W = data.shape

    # compute image grads
    data_grad_x = data[:, 1:, :-1] - data[:, :-1, :-1]
    data_grad_y = data[:, :-1, 1:] - data[:, :-1, :-1]
    dI = torch.stack([data_grad_x, data_grad_y], -1)

    # compute network grads
    data = data.reshape(B, -1)
    data.requires_grad = True
    data.retain_grad()

    output = model(data)
    output.backward(torch.ones_like(output))

    dF = data.grad.reshape(B, H, W)
    dF = dF[:, :-1, :-1]

    # coordinate mask
    xy = torch.meshgrid(torch.arange(0, H-1), torch.arange(0, H-1))
    xy = torch.stack(xy, -1)
    xy = xy / (H // 2) - 1

    # collect into the network polarization matrix
    C = dF[..., None, None] * dI[..., None] * xy[None, :, :, None, :]
    C = C.view(B, -1, 2, 2).sum(1)

    return C
```

References

[1] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[2] Prajit Ramachandran, Barret Zoph, and Quoc V. Le. Searching for activation functions, 2017.

[3] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.