

Felix Dangel

January 28, 2025



```
import jax
import jax.numpy as jnp
def selu(x, alpha=1.67, lam=1.05):
  return lam * jnp.where(x > 0, x, alpha * jnp.exp(x) - alpha)
x = jnp.arange(1_000_000)
%timeit selu(x).block_until_ready()
6.39 ms ± 152 µs
```

```
import jax
import jax.numpy as jnp
def selu(x, alpha = 1.67, lam = 1.05):
  return lam * jnp.where(x > 0, x, alpha * jnp.exp(x) - alpha)
x = inp.arange(1_000_000)
%timeit selu(x).block_until_ready()
6.39 \text{ ms} \pm 152 \text{ µs}
selu_jit = jax.jit(selu)
# Pre-compile the function before timing ...
selu_jit(x).block_until_ready()
%timeit selu_jit(x).block_until_ready()
```

```
import jax
import jax.numpy as jnp
def selu(x, alpha = 1.67, lam = 1.05):
  return lam * jnp.where(x > 0, x, alpha * jnp.exp(x) - alpha)
x = inp.arange(1_000_000)
%timeit selu(x).block_until_ready()
6.39 \text{ ms} \pm 152 \text{ µs}
selu_jit = jax.jit(selu)
# Pre-compile the function before timing ...
selu_jit(x).block_until_ready()
%timeit selu_jit(x).block_until_ready()
```

981 μ s \pm 1.7 μ s

Cool, faster run times for free.

Does it know basic math, too?

Cool, faster run times for free.

Does it know basic math, too?

```
def transpose (A, times = 1):
    result = A
    for _ in range(times):
        result = result.T
    return result
transpose_2 = lambda A: transpose(A, times=2)
transpose_10 = lambda A: transpose(A, times=10)
A = inp. arange(1_000_000). reshape(1_000, 1_000)
%timeit transpose(A).block_until_ready()
```

```
%timeit transpose_2(A).block_until_ready()
%timeit transpose_10(A).block_until_ready()
```

Cool, faster run times for free.

Does it know basic math, too?

```
def transpose (A, times = 1):
    result = A
    for _ in range(times):
        result = result.T
    return result
transpose_2 = lambda A: transpose(A, times=2)
transpose_10 = lambda A: transpose(A, times=10)
A = jnp.arange(1_000_000).reshape(1_000, 1_000)
%timeit_transpose(A) block_until_ready()
```

```
%timeit transpose(A).block_until_ready()
%timeit transpose_2(A).block_until_ready()
%timeit transpose_10(A).block_until_ready()
```

Cool, faster run times for free.

 $15.9 \text{ ms} \pm 1.6 \text{ ms}$

```
Does it know basic math, too? Yes!
```

 $383 \text{ us} \pm 13.9 \text{ us}$

```
def transpose (A, times = 1):
     result = A
     for _ in range(times):
          result = result.T
     return result
transpose_2 = lambda A: transpose(A, times=2)
transpose_10 = lambda A: transpose(A, times=10)
A = inp. arange(1_000_000). reshape(1_000, 1_000)
%timeit transpose(A).block_until_ready()
%timeit transpose_2(A).block_until_ready()
%timeit transpose_10(A).block_until_ready()
1.84 \text{ ms} \pm 593 \text{ } \mu \text{s}
                                                        1.82 \text{ ms} \pm 382 \text{ } \mu \text{s}
                                After jit:
3.58 \text{ ms} \pm 187 \text{ us}
                                                        351 \ \mu s \ \pm \ 14 \ \mu s
```

2





1. Compute a sequence of matrix-vector products, Wx_1, \ldots, Wx_S



- 1. Compute a sequence of matrix-vector products, Wx_1, \ldots, Wx_S
- 2. Sum the results, $\sum_{s=1}^{S} W \mathbf{x}_s$



- 1. Compute a sequence of matrix-vector products, Wx_1, \ldots, Wx_S
- 2. Sum the results, $\sum_{s=1}^{S} W \mathbf{x}_s$
- + Quick-and-dirty way: (W @ X).sum(1)

44.8 ms ± 13.6 ms



- 1. Compute a sequence of matrix-vector products, Wx_1, \ldots, Wx_S
- 2. Sum the results, $\sum_{s=1}^{S} W \mathbf{x}_s$
- + Quick-and-dirty way: (W @ X).sum(1)
- + Quick-and-dirty but jit ed

44.8 ms ± 13.6 ms 57.5 ms ± 15.3 ms



Assume we have a sequence of vectors $\boldsymbol{X} = \begin{pmatrix} \boldsymbol{x}_1 & \dots & \boldsymbol{x}_S \end{pmatrix}$ and we want to

- 1. Compute a sequence of matrix-vector products, $\textbf{Wx}_1, \ldots, \textbf{Wx}_S$
- 2. Sum the results, $\sum_{s=1}^{S} W \mathbf{x}_s$
- + Quick-and-dirty way: (W @ X).sum(1)
- + Quick-and-dirty but jit ed
- + A smart way: W @ X.sum(1)

But is this really an important limitation?

44.8	ms	±	13.6	ms
57.5	ms	±	15.3	ms
697	7 με	s <u>t</u>	: 111	μs

How to jit your jet:

Accelerating Differential Operators by Teaching Compilers About Linearity



$$\Delta f(\boldsymbol{x}) := \sum_{d=1}^{D} \frac{\partial^2 f(\boldsymbol{x})}{\partial x_d^2} = \operatorname{Tr}(\nabla^2 f(\boldsymbol{x}))$$

$$\mathbf{V}$$

$$\Delta f(\boldsymbol{x}) := \sum_{d=1}^{D} \frac{\partial^2 f(\boldsymbol{x})}{\partial x_d^2} = \operatorname{Tr}(\nabla^2 f(\boldsymbol{x}))$$

- 1. PINNs: Solving PDEs with NNs
 - $\mathcal{L} f(\mathbf{x}) = \mathbf{a}(\mathbf{x}) \text{ on } \Omega$ $\mathcal{B} f(\mathbf{x}) = \mathbf{b}(\mathbf{x}) \text{ on } \partial \Omega$

$$\mathbf{V}$$

$$\Delta f(\boldsymbol{x}) \coloneqq \sum_{d=1}^{D} \frac{\partial^2 f(\boldsymbol{x})}{\partial x_d^2} = \mathsf{Tr}(\nabla^2 f(\boldsymbol{x}))$$

1. PINNs: Solving PDEs with NNs

$$\mathcal{L} f(\mathbf{x}) = \mathbf{a}(\mathbf{x}) \text{ on } \Omega$$

 $\mathcal{B} f(\mathbf{x}) = \mathbf{b}(\mathbf{x}) \text{ on } \partial \Omega$

$$egin{aligned} \min_{m{ heta}} & \int_{\Omega} (\mathcal{L} f_{m{ heta}}(m{x}) - m{a}(m{x}))^2 dm{x} \ & + \int_{\partial\Omega} (\mathcal{B} f_{m{ heta}}(m{x}) - m{b}(m{x}))^2 dm{s} \end{aligned}$$

$$\mathbf{V}$$

$$\Delta f(\boldsymbol{x}) \coloneqq \sum_{d=1}^{D} \frac{\partial^2 f(\boldsymbol{x})}{\partial x_d^2} = \mathsf{Tr}(\nabla^2 f(\boldsymbol{x}))$$

1. PINNs: Solving PDEs with NNs

$$\mathcal{L} f(\mathbf{x}) = \mathbf{a}(\mathbf{x}) \quad \text{on} \quad \Omega$$

 $\mathcal{B} f(\mathbf{x}) = \mathbf{b}(\mathbf{x}) \quad \text{on} \quad \partial \Omega$

2. VMC: Finding quantum ground states

$$\min_{\theta} \langle f_{\theta}(\mathbf{x}) | \mathcal{H} | f_{\theta}(\mathbf{x}) \rangle / \langle f_{\theta}(\mathbf{x}) | f_{\theta}(\mathbf{x}) \rangle$$

$$egin{aligned} \min_{m{ heta}} & \int_{\Omega} (\mathcal{L} f_{m{ heta}}(m{x}) - m{a}(m{x}))^2 dm{x} \ & + \int_{\partial\Omega} (\mathcal{B} f_{m{ heta}}(m{x}) - m{b}(m{x}))^2 dm{s} \end{aligned}$$



Naive: Use nested first-order AD

$$abla_{\mathbf{x}}\left(\left(
abla_{\mathbf{x}}f\right)^{ op}\mathbf{v}
ight)=\left(
abla_{\mathbf{x}}^{2}f
ight)\mathbf{v}$$

 \mathbf{V}

Naive: Use nested first-order AD

$$abla_{\mathbf{x}}\left(\left(
abla_{\mathbf{x}}f\right)^{ op}\mathbf{v}
ight)=\left(
abla_{\mathbf{x}}^{2}f
ight)\mathbf{v}$$

One element per HVP

$$\Delta f = \sum_{d=1}^{D} oldsymbol{e}_{d}^{ op} ig(
abla_{\mathbf{x}}^{2} f ig) oldsymbol{e}_{d}$$

Is there a better way?

High-level idea: Consider a function **f**(**x**)

Path in input space $\mathbf{x}(t), t \in [-\epsilon, \epsilon]$

High-level idea: Consider a function **f**(**x**)

Path in input space f Path in output space $f(\mathbf{x}(t))$

High-level idea: Consider a function $f(\mathbf{x})$

Path in input space f $\mathbf{x}(t), t \in [-\epsilon, \epsilon]$ Path in output space $f(\mathbf{x}(t))$ Taylor Truncated Taylor series $f(\mathbf{x}(t)) \approx \mathbf{f}_0 + t\mathbf{f}_1 + \frac{1}{2}t^2\mathbf{f}_2$

High-level idea: Consider a function $f(\mathbf{x})$



High-level idea: Consider a function $f(\mathbf{x})$



High-level idea: Consider a function **f**(**x**)



From wikipedia:

In mathematics, the jet is an operation that takes a differentiable function f and produces a polynomial, the Taylor polynomial (truncated Taylor series) of f.

 \mathbf{V}

Consider a path in input space:
$$\mathbf{x}(t) = \mathbf{x} + t\mathbf{v}_1 + \frac{1}{2}t^2\mathbf{v}_2$$

$$k \qquad \frac{\partial^k \mathbf{x}(t)}{\partial t^k} \bigg|_{t=0} \qquad \frac{\partial^k f(\mathbf{x}(t))}{\partial t^k} \bigg|_{t=0}$$

X

0

 \mathbf{V}

Consider a path in input space:
$$\mathbf{x}(t) = \mathbf{x} + t\mathbf{v}_1 + \frac{1}{2}t^2\mathbf{v}_2$$

$$k \quad \frac{\partial^{k} \mathbf{x}(t)}{\partial t^{k}}\Big|_{t=0} \quad \frac{\partial^{k} f(\mathbf{x}(t))}{\partial t^{k}}\Big|_{t=0}$$

$$0 \quad \mathbf{x} \qquad f(\mathbf{x})$$

$$1 \quad \mathbf{v}_{1}$$

$$2 \quad \mathbf{v}_{2}$$

 \mathbf{V}

Consider a path in input space: **x**(

V₂

2

$$\boldsymbol{x}(t) = \boldsymbol{x} + t\boldsymbol{v}_1 + \frac{1}{2}t^2\boldsymbol{v}_2$$

$$k \qquad \frac{\partial^{k} \mathbf{x}(t)}{\partial t^{k}}\Big|_{t=0} \qquad \frac{\partial^{k} f(\mathbf{x}(t))}{\partial t^{k}}\Big|_{t=0}$$
$$0 \qquad \mathbf{x} \qquad f(\mathbf{x})$$
$$1 \qquad \mathbf{v}_{1} \qquad \frac{\partial f}{\partial \mathbf{x}} \frac{\partial \mathbf{x}}{\partial t}$$

Consider a path in input space:

V₂

2

$$oldsymbol{x}(t) = oldsymbol{x} + toldsymbol{v}_1 + rac{1}{2}t^2oldsymbol{v}_2$$

$$k \quad \frac{\partial^{k} \mathbf{x}(t)}{\partial t^{k}}\Big|_{t=0} \quad \frac{\partial^{k} f(\mathbf{x}(t))}{\partial t^{k}}\Big|_{t=0}$$

$$0 \quad \mathbf{x} \qquad f(\mathbf{x})$$

$$1 \quad \mathbf{v}_{1} \qquad \frac{\partial f}{\partial \mathbf{x}} \frac{\partial \mathbf{x}}{\partial t}$$

 \mathbf{V}

Consider a path in input space:

$$\boldsymbol{x}(t) = \boldsymbol{x} + t\boldsymbol{v}_1 + \frac{1}{2}t^2\boldsymbol{v}_2$$

$$k \qquad \frac{\partial^k \mathbf{x}(t)}{\partial t^k}\Big|_{t=0} \qquad \frac{\partial^k f(\mathbf{x}(t))}{\partial t^k}\Big|_{t=0}$$





Consider a path in input space:

$$\mathbf{x}(t) = \mathbf{x} + t\mathbf{v}_1 + \frac{1}{2}t^2\mathbf{v}_2$$

$$k \qquad \frac{\partial^k \mathbf{x}(t)}{\partial t^k}\Big|_{t=0} \qquad \frac{\partial^k f(\mathbf{x}(t))}{\partial t^k}\Big|_{t=0}$$





Consider a path in input space:

$$oldsymbol{x}(t) = oldsymbol{x} + toldsymbol{v}_1 + rac{1}{2}t^2oldsymbol{v}_2$$

$$k \qquad \frac{\partial^k \mathbf{x}(t)}{\partial t^k}\Big|_{t=0} \qquad \frac{\partial^k f(\mathbf{x}(t))}{\partial t^k}\Big|_{t=0}$$



f0, f1, f2 = jet(f)(x, v1, v2)

V



$$\frac{\partial^k \mathbf{x}(t)}{\partial t^k}\Big|_{t=0} \qquad \frac{\partial^k h(\mathbf{x}(t))}{\partial t^k}\Big|_{t=0} \qquad \frac{\partial^k g(h(\mathbf{x}(t)))}{\partial t^k}\Big|_{t=0}$$







 \mathbf{V}

$$\frac{\partial^{k} \mathbf{x}(t)}{\partial t^{k}}\Big|_{t=0} \qquad \frac{\partial^{k} h(\mathbf{x}(t))}{\partial t^{k}}\Big|_{t=0} \qquad \frac{\partial^{k} g(h(\mathbf{x}(t)))}{\partial t^{k}}\Big|_{t=0} \qquad = \frac{\partial^{k} f(\mathbf{x}(t))}{\partial t^{k}}\Big|_{t=0}$$

$$\mathbf{x}^{(0)} \qquad \mathbf{x}^{(1)} \qquad \mathbf{x}^{(2)} \qquad = f(\mathbf{x}^{(0)})$$

$$\mathbf{y}^{(0)}_{1} \qquad \mathbf{y}^{(1)}_{1} \qquad \mathbf{y}^{(2)}_{1} \qquad = \partial f[\mathbf{v}^{(0)}_{1}]$$

$$\mathbf{y}^{(0)}_{2} \qquad \mathbf{y}^{(1)}_{2} \qquad \mathbf{y}^{(2)}_{2} \qquad = \partial^{2} f[\mathbf{v}^{(0)}_{1}, \mathbf{v}^{(0)}_{1}] + \partial f[\mathbf{v}^{(0)}_{2}]$$



Remember the Laplacian

$$\sum_{d=1}^{D} \frac{\partial^2 f(\mathbf{x})}{\partial x_d^2} = \sum_{d=1}^{D} \partial^2 f[\mathbf{e}_d, \mathbf{e}_d]$$

jet(f) (x, v1, v2)





$$= \partial^2 f[\bm{v}_1^{(0)}, \bm{v}_1^{(0)}] + \partial f[\bm{v}_2^{(0)}]$$

 $\mathbf{V}_{\mathbf{v}}$

Remember the Laplacian

$$\sum_{d=1}^{D} \frac{\partial^2 f(\boldsymbol{x})}{\partial x_d^2} = \sum_{d=1}^{D} \partial^2 f[\boldsymbol{e}_d, \boldsymbol{e}_d]$$

jet(f) (x, v1=ed, v2=0)



$$\mathbf{V}^{\mathbf{A}}$$









 \mathbf{V}

jit(vmap(jet(f)))



jit(vmap(jet(f)))



jit(vmap(jet(f)))



jit(vmap(jet(f)))



jit(vmap(jet(f)))



 \mathbf{V}

jit(vmap(jet(f)))



 \mathbf{V}

jit(vmap(jet(f)))



 \mathbf{V}

jit(vmap(jet(f)))



jit(vmap(jet(f)))

Before: 1 + D + D coefficients

After: 1 + *D* + 1 coefficients



Collapsing the aggregation into Taylor mode ...

+ Is already done (forward Laplacian [Li et al., 2023]), but manually

microsoft/folx

Implementation of Forward Laplacian algorithm in JAX



Collapsing the aggregation into Taylor mode ...

- + Is already done (forward Laplacian [Li et al., 2023]), but manually
- + Generalizes to higher orders, e.g.



microsoft/folx



Implementation of Forward Laplacian algorithm in JAX



Collapsing the aggregation into Taylor mode ...

- + Is already done (forward Laplacian [Li et al., 2023]), but manually
- + Generalizes to higher orders, e.g.

 $\sum_{d} \frac{\partial^{K} f}{\partial \mathbf{x}_{d}^{K}}$

+ Also applies to randomized Taylor mode [Shi et al., 2024]

microsoft/folx



Implementation of Forward Laplacian algorithm in JAX



Collapsing the aggregation into Taylor mode ...

- + Is already done (forward Laplacian [Li et al., 2023]), but manually
- + Generalizes to higher orders, e.g.

 $\sum_{d} \frac{\partial^{\mathsf{K}} f}{\partial \mathbf{x}_{d}^{\mathsf{K}}}$

- + Also applies to randomized Taylor mode [Shi et al., 2024]
- + Is currently not done by jit (I think)

microsoft/folx



Implementation of Forward Laplacian algorithm in JAX





Probing JAX (on a 5 \rightarrow 1024 \rightarrow 768 \rightarrow 512 \rightarrow 256 \rightarrow 128 \rightarrow 1 tanh-MLP):

- Laplacian via jax.hessian :
- + Laplacian via folx.forward_laplacian
- Laplacian via jax.experimental.jet

168 ms ± 10.7 ms (1.0 x) 109 ms ± 4.19 ms (0.6 x) 263 ms ± 20.5 ms (1.6 x)

 \mathbf{V}

Probing JAX (on a 5 \rightarrow 1024 \rightarrow 768 \rightarrow 512 \rightarrow 256 \rightarrow 128 \rightarrow 1 tanh-MLP):

- Laplacian via jax.hessian :
- + Laplacian via folx.forward_laplacian
- Laplacian via jax.experimental.jet

168 ms ± 10.7 ms (1.0 x) 109 ms ± 4.19 ms (0.6 x) 263 ms ± 20.5 ms (1.6 x)

PyTorch prototype:

+ Can trace a function, replace ops with Taylor mode (jet)

V

Probing JAX (on a 5 \rightarrow 1024 \rightarrow 768 \rightarrow 512 \rightarrow 256 \rightarrow 128 \rightarrow 1 tanh-MLP):

- Laplacian via jax.hessian :
- + Laplacian via folx.forward_laplacian
- Laplacian via jax.experimental.jet

168 ms \pm 10.7 ms (1.0 x) 109 ms \pm 4.19 ms (0.6 x) 263 ms \pm 20.5 ms (1.6 x)

PyTorch prototype:

- + Can trace a function, replace ops with Taylor mode (jet)
- + Can vmap and capture the corresponding graph (vmap(jet))

 \mathbf{V}

Probing JAX (on a 5 \rightarrow 1024 \rightarrow 768 \rightarrow 512 \rightarrow 256 \rightarrow 128 \rightarrow 1 tanh-MLP):

- Laplacian via jax.hessian :
- + Laplacian via folx.forward_laplacian
- + Laplacian via jax.experimental.jet

168 ms ± 10.7 ms (1.0 x) 109 ms ± 4.19 ms (0.6 x) 263 ms ± 20.5 ms (1.6 x)

PyTorch prototype:

- + Can trace a function, replace ops with Taylor mode (jet)
- + Can vmap and capture the corresponding graph (vmap(jet))
- + Missing: Simplifications based on linearity (jit(vmap(jet)))

The Obligatory LLM Slide Before We Are Done

For fun, let's ask an LLM to simplify the compute graph for us.

ChatGPT ~

B

You are an extremely good Python programmer. Simplify the following code, which computes the Laplacian of a function, by applying two steps:

 Prune unnecessary computations on X. The tensor is repeated and then processed by operations that treat this new dimension as batch axis.

2. Instead of propagating the second-order derivatives per coordinate, then sum to get the Laplacian, pull the sum into the forward propagation. This is known as the forward Laplacian.

Here is the code:

```
def forward(self, X):
    x = X
    unsqueeze = x.unsqueeze(0)
    repeat = unsqueeze.repeat(5, 1, 1); unsqueeze =
    zeros_like = torch.zeros_like(x); x = None
    unsqueeze_1 = zeros_like.unsqueeze(0); zeros_li
    expand = unsqueeze_1.expand(5, -1, -1); unsquee
    _tensor_constant0 = self._tensor_constant0
    stack = \<sup>4</sup> ch.stack((repeat, _tensor_constant0)
```

The Obligatory LLM Slide Before We Are Done

For fun, let's ask an LLM to simplify the compute graph for us.

Does not work.

ChatGPT ~

B



You are an extremely good Python programmer. Simplify the following code, which computes the Laplacian of a function, by applying two steps:

 Prune unnecessary computations on X. The tensor is repeated and then processed by operations that treat this new dimension as batch axis.

2. Instead of propagating the second-order derivatives per coordinate, then sum to get the Laplacian, pull the sum into the forward propagation. This is known as the forward Laplacian.

Here is the code:

```
def forward(self, X):
    x = X
    unsqueeze = x.unsqueeze(0)
    repeat = unsqueeze.repeat(5, 1, 1); unsqueeze =
    zeros_like = torch.zeros_like(x); x = None
    unsqueeze_1 = zeros_like.unsqueeze(0); zeros_li
    expand = unsqueeze_1.expand(5, -1, -1); unsquee
    _tensor_constant0 = self._tensor_constant0
    stack = \ifty ch.stack((repeat, _tensor_constant0,
    })
```

The Obligatory LLM Slide Before We Are Done

For fun, let's ask an LLM to simplify the compute graph for us.

Does not work.

In PyTorch, even the un jit ted jet is a good alternative for computing Laplacians:

Hessian	trace:	83.4	±	9.2	ms
PyTorch	jet:	84.3	±	8.8	ms

ChatGPT ~



You are an extremely good Python programmer. Simplify the following code, which computes the Laplacian of a function, by applying two steps:

 Prune unnecessary computations on X. The tensor is repeated and then processed by operations that treat this new dimension as batch axis.

2. Instead of propagating the second-order derivatives per coordinate, then sum to get the Laplacian, pull the sum into the forward propagation. This is known as the forward Laplacian.

Here is the code:

```
def forward(self, X):
    x = X
    unsqueeze = x.unsqueeze(0)
    repeat = unsqueeze.repeat(5, 1, 1); unsqueeze =
    zeros_like = torch.zeros_like(x); x = None
    unsqueeze_1 = zeros_like.unsqueeze(0); zeros_li
    expand = unsqueeze_1.expand(5, -1, -1); unsquee
    _tensor_constant0 = self._tensor_constant0
    stack = \<sup>4</sup> ch.stack((repeat, _tensor_constant0)
```



act then sum \xrightarrow{jit} sum then act

1. We can simplify important diff ops: "pull the sum into Taylor mode" (jet).



act then sum \xrightarrow{jit} sum then act

- 1. We can simplify important diff ops: "pull the sum into Taylor mode" (jet).
- 2. This could be done by a compiler (jit) and requires the concept of linearity.





- 1. We can simplify important diff ops: "pull the sum into Taylor mode" (jet).
- 2. This could be done by a compiler (jit) and requires the concept of linearity.
- 3. Doing so would contain currently specialized implementations without extra work

forward_laplacian(f) = jit(sum(vmap(jet(f))))





- 1. We can simplify important diff ops: "pull the sum into Taylor mode" (jet).
- 2. This could be done by a compiler (jit) and requires the concept of linearity.
- 3. Doing so would contain currently specialized implementations without extra work

forward_laplacian(f) = jit(sum(vmap(jet(f))))

Points for discussion:

- + How linear algebra-aware should DL compilers be?
- + Can we use diff ops to regularize NNs and enforce (un)desirable properties?
- + What is the mathematical concept for "collapsing" multiple jets?



- Ruichen Li, Haotian Ye, Du Jiang, Xuelan Wen, Chuwei Wang, Zhe Li, Xiang Li, Di He, Ji Chen, Weiluo Ren, et al. Forward laplacian: A new computational framework for neural network-based variational monte carlo, 2023.
- Zekun Shi, Zheyuan Hu, Min Lin, and Kenji Kawaguchi. Stochastic taylor derivative estimator: Efficient amortization for arbitrary differential operators. In Advances in Neural Information Processing Systems (NeurIPS), 2024.