Hands on NeuLat:
A Toolbox for **Neu**ral Samplers in **Lat**tice Field Theory

N E U **L** A T

Kim A. Nicoli, **Christopher J. Anders**, Lena Funcke, Karl Jansen,
Shinichi Nakajima, Pan Kessel

October 24, 2024 @ ML4PhysChem

https://github.com/neulat/neulat

About me

Christopher J. Anders

PostDoc at RIKEN AIP, Tokyo

Research Interests

- Deep Learning
- Model Understanding
- Software for ML
- ML in the Sciences

Background

- B.Sc. *Computer Science* @ TU Berlin (2016)
- M.Sc. *Computer Science* @ TU Berlin (2018)
- PhD *Computer Science* @ TU Berlin (2024)

Disclaimer: I am not a physicist!

What is NeuLat?

- software framework for machine-learning-based lattice field theory

What is NeuLat?

- software framework for machine-learning-based lattice field theory
  - e.g., ϕ^4-theory, $U(1)$ gauge theory, up to $3+1D$

What is NeuLat?

- software framework for machine-learning-based lattice field theory
  - e.g., ϕ^4-theory, $U(1)$ gauge theory, up to $3+1D$
- unifies existing tools into one framework

What is NeuLat?

- software framework for machine-learning-based lattice field theory
    - e.g., ϕ^4-theory, $U(1)$ gauge theory, up to $3 + 1D$
- unifies existing tools into one framework
- core team with expertise in LFT, machine learning, software development

What is NeuLat?

- software framework for machine-learning-based lattice field theory
  - e.g., ϕ^4-theory, $U(1)$ gauge theory, up to $3+1D$
- unifies existing tools into one framework
- core team with expertise in LFT, machine learning, software development
- previous contributions in LFT

What is NeuLat?

- software framework for machine-learning-based lattice field theory
  - e.g., ϕ^4-theory, $U(1)$ gauge theory, up to $3+1D$
- unifies existing tools into one framework
- core team with expertise in LFT, machine learning, software development
- previous contributions in LFT
  - asymptotically unbiased estimators (Nicoli et al. (2020))

What is NeuLat?

- software framework for machine-learning-based lattice field theory
  - e.g., ϕ^4-theory, $U(1)$ gauge theory, up to $3+1D$
- unifies existing tools into one framework
- core team with expertise in LFT, machine learning, software development
- previous contributions in LFT
  - asymptotically unbiased estimators (Nicoli et al. (2020))
  - thermodynamic observables (Nicoli et al. (2021))

What is NeuLat?

- software framework for machine-learning-based lattice field theory
  - e.g., ϕ^4-theory, $U(1)$ gauge theory, up to $3+1D$
- unifies existing tools into one framework
- core team with expertise in LFT, machine learning, software development
- previous contributions in LFT
  - asymptotically unbiased estimators (Nicoli et al. (2020))
  - thermodynamic observables (Nicoli et al. (2021))
  - mode-dropping estimators (Nicoli et al. (2023))

What is NeuLat?

- software framework for machine-learning-based lattice field theory
  - e.g., ϕ^4-theory, $U(1)$ gauge theory, up to $3 + 1D$
- unifies existing tools into one framework
- core team with expertise in LFT, machine learning, software development
- previous contributions in LFT
  - asymptotically unbiased estimators (Nicoli et al. (2020))
  - thermodynamic observables (Nicoli et al. (2021))
  - mode-dropping estimators (Nicoli et al. (2023))
  - path gradients (Vaitl et al. (2022))

What is NeuLat?

- software framework for machine-learning-based lattice field theory
  - e.g., ϕ^4-theory, $U(1)$ gauge theory, up to $3+1D$
- unifies existing tools into one framework
- core team with expertise in LFT, machine learning, software development
- previous contributions in LFT
  - asymptotically unbiased estimators (Nicoli et al. (2020))
  - thermodynamic observables (Nicoli et al. (2021))
  - mode-dropping estimators (Nicoli et al. (2023))
  - path gradients (Vaitl et al. (2022))
  - trivializing maps with flows (Bacchio et al. (2023))

Benefits of Research Software Packages

- faster development of new ideas

Benefits of Research Software Packages

- faster development of new ideas
- higher reproducibility

Benefits of Research Software Packages

- faster development of new ideas
- higher reproducibility
- easier access into the field

Benefits of Research Software Packages

- faster development of new ideas
- higher reproducibility
- easier access into the field
- no need to re-invent the wheel every time

Benefits of Research Software Packages

- faster development of new ideas
- higher reproducibility
- easier access into the field
- no need to re-invent the wheel every time
- software hub to share research

Benefits of Research Software Packages

- faster development of new ideas
- higher reproducibility
- easier access into the field
- no need to re-invent the wheel every time
- software hub to share research

Benefits of Research Software Packages

- faster development of new ideas
- higher reproducibility
- easier access into the field
- no need to re-invent the wheel every time
- software hub to share research

Existing examples:

- SchNetPack - Deep Neural Networks for Atomistic Systems
- BGFlow - Boltzmann Generators (BG) and other sampling methods

Why NeuLat?

There are already great tools available!

- Introduction to Normalizing Flows for Lattice Field Theory (Albergo et al., 2021)

Why NeuLat?

There are already great tools available!

- Introduction to Normalizing Flows for Lattice Field Theory (Albergo et al., 2021)
- Hamiltonian Monte Carlo

Why NeuLat?

There are already great tools available!

- Introduction to Normalizing Flows for Lattice Field Theory (Albergo et al., 2021)
- Hamiltonian Monte Carlo
  - fthmc: Field Transformation HMC (Sam Foreman et al.)
  - l2hmc-qcd (Sam Foreman et al.)

Why NeuLat?

There are already great tools available!

- Introduction to Normalizing Flows for Lattice Field Theory (Albergo et al., 2021)
- Hamiltonian Monte Carlo
  - fthmc: Field Transformation HMC (Sam Foreman et al.)
  - l2hmc-qcd (Sam Foreman et al.)
- Normalizing Flows
  - nflows
  - GomalizingFlow (Akio Tomiya et al.)

Why NeuLat?

There are already great tools available!

- Introduction to Normalizing Flows for Lattice Field Theory (Albergo et al., 2021)
- Hamiltonian Monte Carlo
  - fthmc: Field Transformation HMC (Sam Foreman et al.)
  - l2hmc-qcd (Sam Foreman et al.)
- Normalizing Flows
  - nflows
  - GomalizingFlow (Akio Tomiya et al.)
- Flows/HMC
  - NeuMC (Piotr Bialas et al.)

Why NeuLat?

There are already great tools available!

- Introduction to Normalizing Flows for Lattice Field Theory (Albergo et al., 2021)
- Hamiltonian Monte Carlo
  - fthmc: Field Transformation HMC (Sam Foreman et al.)
  - l2hmc-qcd (Sam Foreman et al.)
- Normalizing Flows
  - nflows
  - GomalizingFlow (Akio Tomiya et al.)
- Flows/HMC
  - NeuMC (Piotr Bialas et al.)

But: We want to create a **highly customizable reference implementation**.

Core Features of NeuLat: Based on PyTorch

- **Density Estimator**: Learn approximations of targeted Boltzmann distributions

Core Features of NeuLat: Based on PyTorch

- **Density Estimator**: Learn approximations of targeted Boltzmann distributions
- **Sampling**:
  - various MCMC implementations (HMC, Cluster algorithms, etc.)
  - Normalizing Flow framework
    - Neural Importance Sampling
    - Neural HMC

Core Features of NeuLat: Based on PyTorch

- **Density Estimator**: Learn approximations of targeted Boltzmann distributions
- **Sampling**:
  - various MCMC implementations (HMC, Cluster algorithms, etc.)
  - Normalizing Flow framework
    - Neural Importance Sampling
    - Neural HMC
- **Estimation**:
  - Asymptotically unbiased estimators for physical observables (Nicoli et al. (2020)).
  - Direct estimation of thermodynamic observables with flows and HMC (Nicoli et al. (2021)).
  - Sampling in the presence of mode-collapse (Nicoli et al. (2023)).

Core Features of NeuLat: Based on PyTorch

- **Density Estimator**: Learn approximations of targeted Boltzmann distributions
- **Sampling**:
  - various MCMC implementations (HMC, Cluster algorithms, etc.)
  - Normalizing Flow framework
    - Neural Importance Sampling
    - Neural HMC
- **Estimation**:
  - Asymptotically unbiased estimators for physical observables (Nicoli et al. (2020)).
  - Direct estimation of thermodynamic observables with flows and HMC (Nicoli et al. (2021)).
  - Sampling in the presence of mode-collapse (Nicoli et al. (2023)).
- **Tutorials and Documentation**:
  - Step-by-step tutorials
  - Extensive reference

Core Features of NeuLat: Based on PyTorch

- **Density Estimator**: Learn approximations of targeted Boltzmann distributions
- **Sampling**:
  - various MCMC implementations (HMC, Cluster algorithms, etc.)
  - Normalizing Flow framework
    - Neural Importance Sampling
    - Neural HMC
- **Estimation**:
  - Asymptotically unbiased estimators for physical observables (Nicoli et al. (2020)).
  - Direct estimation of thermodynamic observables with flows and HMC (Nicoli et al. (2021)).
  - Sampling in the presence of mode-collapse (Nicoli et al. (2023)).
- **Tutorials and Documentation**:
  - Step-by-step tutorials
  - Extensive reference
- **Modularity and Customizability**: Swiftly incorporate new actions/theories/models/techniques

NeuLat Overview

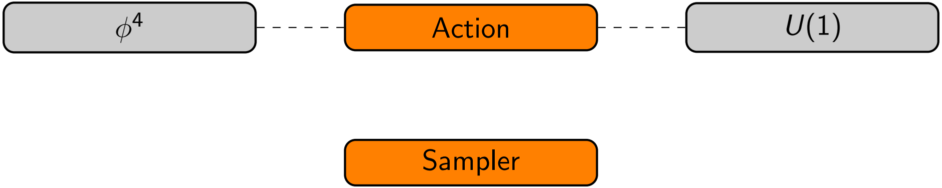
Action

Actions $S[U]$ define physical theories $p(U) \propto e^{-S[U]}$

NeuLat Overview

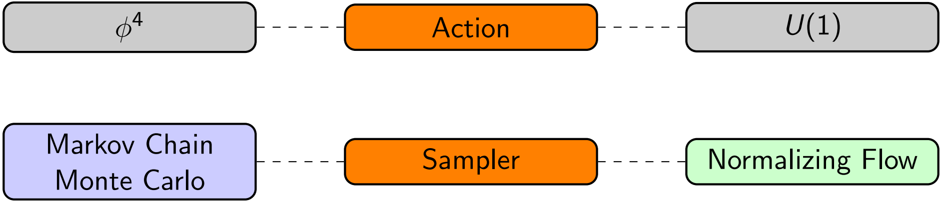
Actions are, e.g., ϕ^4 and $U(1)$

NeuLat Overview



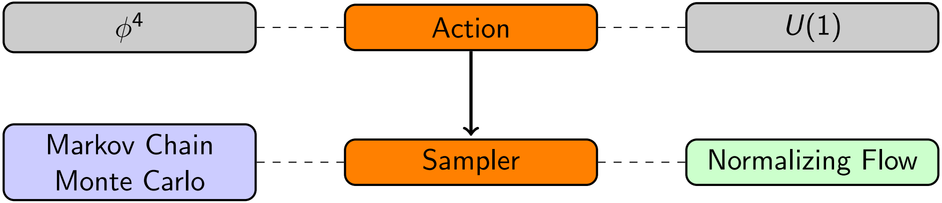
Samplers are anything that can be sampled from

NeuLat Overview



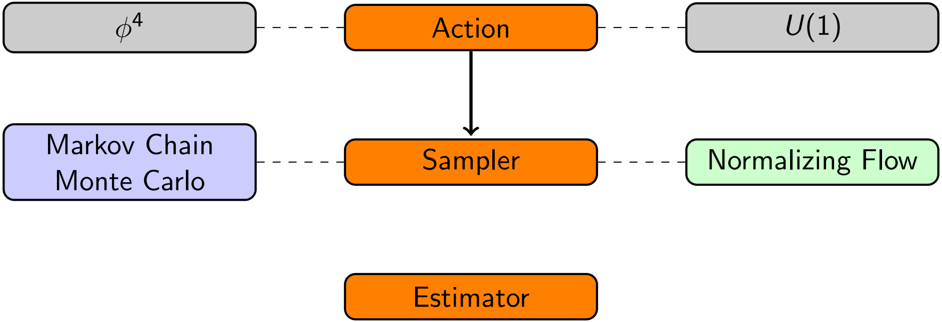
Samplers are, e.g., MCMCs, Flows, $\mathcal{N}(0, 1)$

NeuLat Overview



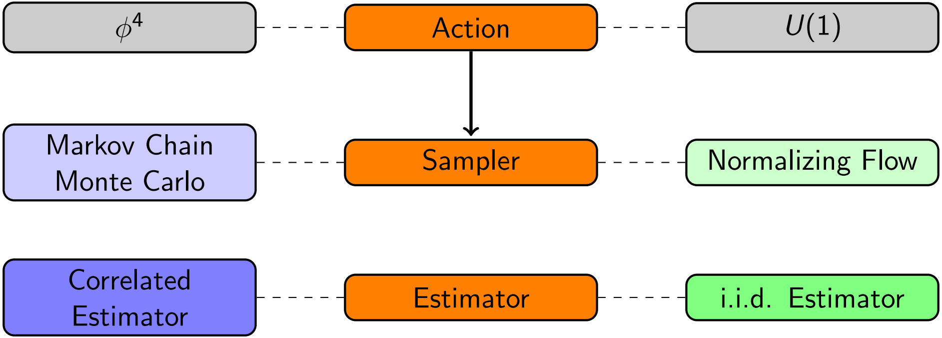
Samplers require Actions

NeuLat Overview



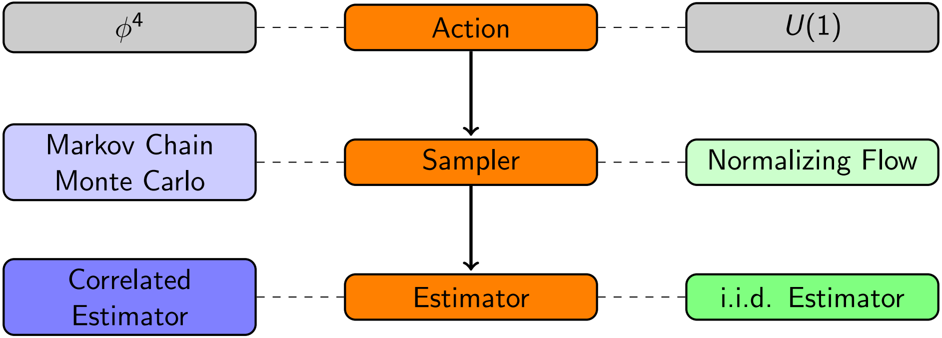
Estimators are used to estimate observables

NeuLat Overview



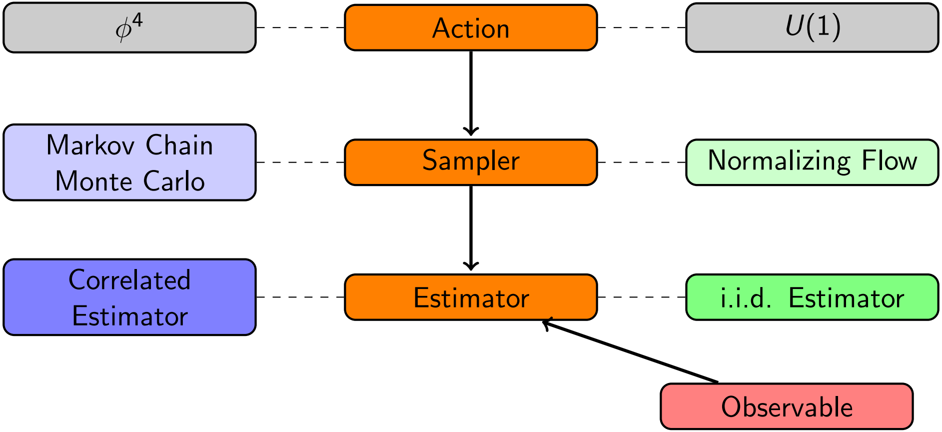
Estimators are, e.g., i.i.d or correlated, based on the samples

NeuLat Overview



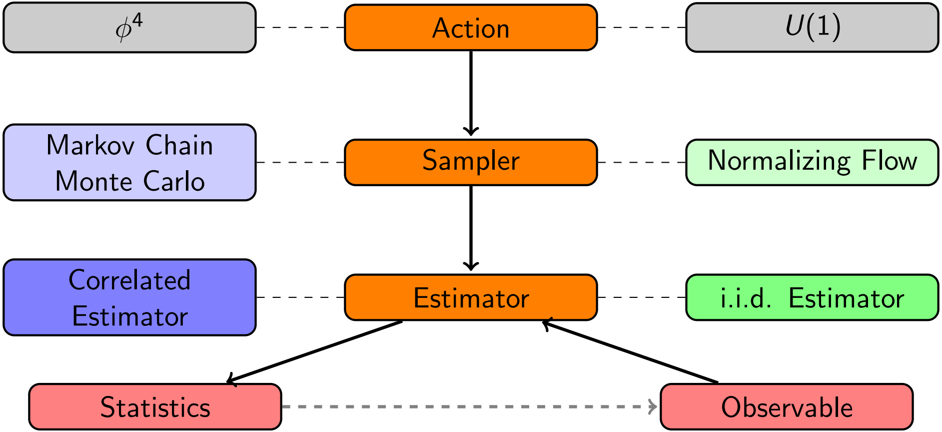
Estimators require samples from Samplers

NeuLat Overview



Observables, such as Magnetization in ϕ^4, are used by the Estimator

NeuLat Overview



Resulting Statistics are estimations for the Observables

Actions

Actions are objects and need to be instantiated.

```
import torch
from neulat.action.phi4 import Phi4Action

# ndim_features is the number of dimensions in the lattice
action = Phi4Action(kappa=0.3, lamb=0.022, ndim_features=2)
```

Actions

Actions are objects and need to be instantiated.

```python
import torch
from neulat.action.phi4 import Phi4Action

# ndim_features is the number of dimensions in the lattice
action = Phi4Action(kappa=0.3, lamb=0.022, ndim_features=2)
```

Action objects can be called to compute action values for configurations.

```python
config = torch.randn(8, 8)
unnormalized_prob = torch.exp(-action(config))
```

Defining Actions

Actions are very simple to implement, for instance ϕ^4:

```
from neulat.action.base import Action

class Phi4Action(Action):
    name = 'phi4_action'
    def __init__(self, kappa, lambd, ndim_feature=2):
        ...
    def forward(self, config):
        dims = tuple(range(-1, -self.ndim_feature, -1))
        kinetic = (-2 * self.kappa) * config * sum(
            torch.roll(config, 1, dim) for dim in dims)
        mass_inter = (1 - 2 * self.lambd) * config ** 2
        inter = self.lambd * config ** 4
        return (kinetic + mass + inter).sum(dim=dims)
```

Samplers

At the core of NeuLat are Samplers, which is anything from which can be sampled.

Samplers

At the core of NeuLat are Samplers, which is anything from which can be sampled.

For instance, the normal distribution is a Sampler in Neulat:

```
from neulat.sampler.distribution import Normal

normal = Normal(loc=0., scale=1., feature_shape=(8, 8))
```

Sampling

Samplers can be sampled from, and may or may not support probability values.

```
samples = normal.sample(sample_shape=8)
logprobs = normal.logprob(samples)

samples2, logprobs2 = normal.sample_with_logprob((2, 2))
```

Sampling

Samplers can be sampled from, and may or may not support probability values.

```
samples = normal.sample(sample_shape=8)
logprobs = normal.logprob(samples)

samples2, logprobs2 = normal.sample_with_logprob((2, 2))
```

In NeuLat, we assume configurations of shape `(*sample_shape, *feature_shape)`.

- `feature_shape` is the shape of the lattice
- `sample_shape` is the number of samples, supporting arbitrary shapes

Hamiltonian Monte Carlo

A more involved Sampler is the HMC:

```
from neulat.sampler.mc.hmc import HMCMarkovChain

hmc = HMCMarkovChain(
    action, # action
    feature_shape=(8, 8),  # lattice shape
    burn_in=5000,  # equilibration steps
    skip_interval=1,  # skipped samples in chain
    overrelax_interval=50,  # steps between sign flips
    eps=0.05,  # step size along trajectory
    traj_steps=20,  # number of steps in trajectory
    bias=0.0,  # bias in initialization
)
```

HMC Sampling

The HMC can be sampled from, as any sampler

```
configs = hmc.sample(sample_shape=13)
```

HMC Sampling

The HMC can be sampled from, as any sampler

```
configs = hmc.sample(sample_shape=13)
```

However, HMC does not implement `logprob` and by extension `sample_with_logprog`, as no normalized probabilities are available

```
# both cause exceptions:
# logprobs = hmc.logprob(sample_shape=13)
# configs2, logprobs2 = hmc.sample_with_logprob(13)
```

HMC Sampling 2

One can also iterate over HMCs to sample

```
configs = []
for n, config in zip(range(25), hmc):
    configs.append(config)
    print(f'Sampled config number {n}.')

# this gives a list of configs, combine them:
configs = torch.cat(configs)
```

HMC Sampling 2

One can also iterate over HMCs to sample

```
configs = []
for n, config in zip(range(25), hmc):
    configs.append(config)
    print(f'Sampled config number {n}.')

# this gives a list of configs, combine them:
configs = torch.cat(configs)
```

But be careful, HMCs are infinite iterators.

Normalizing Flows

Normalizing flows require a base distribution, and a transform.

```python
from neulat.sampler.flow import Flow, SequentialTransform

flow = Flow(
  base_distribution=Normal(feature_shape=(8, 8)),
  transform=SequentialTransform([])  # identity for demo
)
```

Normalizing Flows

Normalizing flows require a base distribution, and a transform.

```
from neulat.sampler.flow import Flow, SequentialTransform

flow = Flow(
  base_distribution=Normal(feature_shape=(8, 8)),
  transform=SequentialTransform([])  # identity for demo
)
```

Normalizing flows are (i.i.d.) Samplers supporting logprobs.

```
configs, logprobs = flow.sample_with_logprob(8)
```

Normalizing Flows: Base Distributions

The base distribution can be any sampler that supports logprobs.

Normalizing Flows: Base Distributions

The base distribution can be any sampler that supports logprobs.

Commonly, simple distributions such as $\mathcal{N}(0, 1)$ are used.

```
flow = Flow(
  base_distribution=Normal(feature_shape=(8, 8)),
  transform=SequentialTransform([])  # identity for demo
)
```

Normalizing Flows: Base Distributions

The base distribution can be any sampler that supports logprobs.

Commonly, simple distributions such as $\mathcal{N}(0, 1)$ are used.

```
flow = Flow(
  base_distribution=Normal(feature_shape=(8, 8)),
  transform=SequentialTransform([])  # identity for demo
)
```

Flows themselves support logprobs, and can thus be base distributions.

```
flow2 = Flow(
  base_distribution=flow,
  transform=SequentialTransform([])  # identity for demo
)
```

Normalizing Flows: Transforms

Transforms are invertible PyTorch modules, and require a `forward` and a `inverse`.

Normalizing Flows: Transforms

Transforms are invertible PyTorch modules, and require a `forward` and a `inverse`.

E.g., implementation for transform $f(\mathbf{x}) = -\mathbf{x}$, $f^{-1}(\mathbf{x}) = -\mathbf{x}$

```python
from sampler.flow.base import Transform, withlogdet

class FlipSign(Transform):
    @withlogdet
    def forward(self, input):
        return -input, 1.
    @withlogdet
    def inverse(self, input):
        return -input, 1.
```

Normalizing Flows: Transforms

Transforms are invertible PyTorch modules, and require a `forward` and a `inverse`.

E.g., implementation for transform $f(\mathbf{x}) = -\mathbf{x}$, $f^{-1}(\mathbf{x}) = -\mathbf{x}$

```
from sampler.flow.base import Transform, withlogdet

class FlipSign(Transform):
    @withlogdet
    def forward(self, input):
        return -input, 1.
    @withlogdet
    def inverse(self, input):
        return -input, 1.
```

The second return value is the *log absolute jacobian determinant* of the transform.

Normalizing Flows: Transforms

Transforms are invertible PyTorch modules, and require a `forward` and a `inverse`.

E.g., implementation for transform $f(\mathbf{x}) = -\mathbf{x}$, $f^{-1}(\mathbf{x}) = -\mathbf{x}$

```python
from sampler.flow.base import Transform, withlogdet

class FlipSign(Transform):
    @withlogdet
    def forward(self, input):
        return -input, 1.
    @withlogdet
    def inverse(self, input):
        return -input, 1.
```

The decorator `@withlogdet` makes sure the logdet is accumulated between transforms.

Normalizing Flows: Transforms 2

A useful transform is the `SequentialTransform`, which is used to apply transforms sequentially:

```
from sampler.flow.base import SequentialTransform

flip_a_bunch = SequentialTransform([
  FlipSign(),
  FlipSign(),
  FlipSign(),
])
```

Normalizing Flows: Transforms 2

A useful transform is the `SequentialTransform`, which is used to apply transforms sequentially:

```
from sampler.flow.base import SequentialTransform

flip_a_bunch = SequentialTransform([
  FlipSign(),
  FlipSign(),
  FlipSign(),
])
```

For common *Coupling* Flows, there is however a more convenient way.

Coupling Flows

Coupling flows like NICE consist of two parts, a `partitioner`, and a `net_factory`

```python
from neulat.sampler.flow.coupling import NICE

coupling = NICE(
    partitioner=partitioner,
    net_factory=net_factory
)
```

Coupling Flows

Coupling flows like NICE consist of two parts, a `partitioner`, and a `net_factory`

```python
from neulat.sampler.flow.coupling import NICE

coupling = NICE(
    partitioner=partitioner,
    net_factory=net_factory
)
```

The `partitioner` *partitions* (or masks) the input into *active* and *passive* components.

Coupling Flows

Coupling flows like NICE consist of two parts, a `partitioner`, and a `net_factory`

```python
from neulat.sampler.flow.coupling import NICE

coupling = NICE(
    partitioner=partitioner,
    net_factory=net_factory
)
```

The `partitioner` *partitions* (or masks) the input into *active* and *passive* components.

The `net_factory` is a function that constructs the *conditioner*, e.g., a neural network that acts on the partitioned input.

Coupling Flows: Partitioners

A very simple partitioner is the `AltFlatPartitioner`, which stands for *alternating flattened partitioner*

```
partitioner = AltFlatPartitioner(feature_shape=(2, 2)),
input = torch.tensor([[1., 2.],[3., 4.]])
active, passive = partitioner(input)
active += 10
output = partitioner(active, passive)
```

Coupling Flows: Partitioners

A very simple partitioner is the `AltFlatPartitioner`, which stands for *alternating flattened partitioner*

```
partitioner = AltFlatPartitioner(feature_shape=(2, 2)),
input = torch.tensor([[1., 2.],[3., 4.]])
active, passive = partitioner(input)
active += 10
output = partitioner(active, passive)
```

This will generate an output of $\begin{pmatrix} 11 & 2 \\ 13 & 4 \end{pmatrix}$

Coupling Flows: Flipping Partitioners

Partitioners usually flip the active and pasive elements.

Such a partitioner can be created by calling `.flip()`:

```
flipped = partitioner.flip()
input = torch.tensor([[1., 2.],[3., 4.]])
active, passive = flipped(input)
active += 1
output = flipped(active, passive)
```

Coupling Flows: Flipping Partitioners

Partitioners usually flip the active and pasive elements.

Such a partitioner can be created by calling `.flip()`:

```
flipped = partitioner.flip()
input = torch.tensor([[1., 2.],[3., 4.]])
active, passive = flipped(input)
active += 1
output = flipped(active, passive)
```

This will generate an output of $\begin{pmatrix} 1 & 12 \\ 3 & 14 \end{pmatrix}$

Coupling Flows: Net Factory (Conditioner)

The `net_factory` define the *conditioner* Θ that transforms the passive input:

$$\mathbf{x}^{l+1}_{\text{active}} = h(\mathbf{x}^{l}_{\text{active}}, \Theta(\mathbf{x}^{l}_{\text{passive}})) \tag{1}$$

Coupling Flows: Net Factory (Conditioner)

The `net_factory` define the *conditioner* Θ that transforms the passive input:

$$\mathbf{x}^{l+1}_{\text{active}} = h(\mathbf{x}^{l}_{\text{active}}, \Theta(\mathbf{x}^{l}_{\text{passive}})) \tag{1}$$

```python
from functools import partial
from neulat.sampler.flow.coupling.affine import NICE, MLP

net_factory = partial(
    MLP,
    n_blocks=3,
    latent_size=1024,
    activation=torch.nn.Tanh,
    bias=False,
)
```

Coupling Flows: Defining Couplings

The coupling `Transform` itself is mostly only concerned with implementing the coupling function h. E.g. in NICE: $h(a, b) = a + b$

```
class NICE(Coupling):
    @withlogdet
    @partitioned
    def forward(self, active, passive):
        return active + self.net(passive), 1.

    @withlogdet
    @partitioned
    def inverse(self, active, passive):
        return active - self.net(passive), 1.
```

Coupling Flows: Defining Couplings

The coupling `Transform` itself is mostly only concerned with implementing the coupling function h. E.g. in NICE: $h(a, b) = a + b$

```
class NICE(Coupling):
    @withlogdet
    @partitioned
    def forward(self, active, passive):
        return active + self.net(passive), 1.

    @withlogdet
    @partitioned
    def inverse(self, active, passive):
        return active - self.net(passive), 1.
```

Recall: `@withlogdet` makes sure the log abs jacobian det is propagated.

Coupling Flows: Defining Couplings

The coupling `Transform` itself is mostly only concerned with implementing the coupling function h. E.g. in NICE: $h(a, b) = a + b$

```
class NICE(Coupling):
    @withlogdet
    @partitioned
    def forward(self, active, passive):
        return active + self.net(passive), 1.

    @withlogdet
    @partitioned
    def inverse(self, active, passive):
        return active - self.net(passive), 1.
```

New: `@partitioned` automates the partitioning/masking in subsequent couplings!

Coupling Flows: Completing the Flow

Putting all the previous parts together, we can create a flow in the following way:

```
flow = Flow(
  base_distribution=Normal(0.0, 1.0, feature_shape=(8, 8)),
  transform=6 * NICE(
      partitioner=AltFlatPartitioner(feature_shape=(8, 8)),
      net_factory=partial(MLP, n_blocks=3, latent_size=1024,
        activation=torch.nn.Tanh, bias=False)
)
```

Coupling Flows: Completing the Flow

Putting all the previous parts together, we can create a flow in the following way:

```
flow = Flow(
  base_distribution=Normal(0.0, 1.0, feature_shape=(8, 8)),
  transform=6 * NICE(
      partitioner=AltFlatPartitioner(feature_shape=(8, 8)),
      net_factory=partial(MLP, n_blocks=3, latent_size=1024,
        activation=torch.nn.Tanh, bias=False)
)
```

Notice the `transform=6 * NICE`. This creates a sequential transform of 6 Couplings, with alternating masking/partitioning!

Coupling Flows: Traning

Training of the flow with ReverseKL is straight forward:

```
from neulat.loss import ReverseKLLoss

optim = torch.optim.Adam(flow.transform.parameters(), lr=5e-4)
loss_fn = ReverseKLLoss()
for _ in range(1000):
    configs, log_probs = flow.sample_with_logprob(10)
    loss = loss_fn(action(configs), log_probs)  # loss contains `mean` and `std`
    optim.zero_grad()
    loss.mean.backward()  # we train only using the loss `mean`
    optim.step()
```

Estimating Observables from i.i.d. Samples

Observables themselves are classes in NeuLat. In order to estimate them, we additionally need an `Estimator`, and configurations. For instance:

```
from neulat.observable.base import AbsMagnetization, Magnetization
from neulat.estimator.base import IidEstimator

observables = [AbsMagnetization(), Magnetization(), action]
iid_estimator = IidEstimator(observables)
configs = flow.sample(1000)
flow_statistics = iid_estimator.named_evaluate(configs)
```

The dict `flow_statistics` will contain one entry per observable, e.g.:

`{'absmag': Statistics(mean=0.6408, std=0.0473), 'mag': ...}`

Estimating Observables from Correlated Samples

Estimation of Observables from correlated samples (e.g., from HMC) requires the use of the appropriate estimator:

```python
from neulat.estimator.base import CorrelatedEstimator

correlated_estimator = CorrelatedEstimator(observables)
configs = hmc.sample(1000)
hmc_statistics = correlated_estimator.named_evaluate(configs)
```

The dict `hmc_statistics` will instead contain correlated statistics objects,

`{'absmag': CorrelatedStatistics(mean=32.82628, std=1.4674, tau_int=0.5909, tau_int_err=0.3162), ...}`

Neural Importance Sampling

To obtain an unbiased estimator, Nicoli et. al proposed to use Importance Sampling. This additionally requires the logprobs of the flow, as well as the specific action:

```
from neulat.estimator.base import ImportanceSamplingEstimator

flow_configs, flow_logprobs = flow.sample_with_logprob(1000)
iw_estimator = ImportanceSamplingEstimator(observables, action)
flow_iw_stats = iw_estimator.evaluate(flow_configs, flow_logprobs)
```

The dict `flow_iw_statistics` will contain the same Statistics object the IidEstimator returned:

`{'absmag': Statistics(mean=2.6021, std=0.4674, ...}`

Planned Features

With the help of the community, we plan to extend NeuLat into many directions, including following features

- Stochastic normalizing flows (Caselle et al., 2022)

Planned Features

With the help of the community, we plan to extend NeuLat into many directions, including following features

- Stochastic normalizing flows (Caselle et al., 2022)
- Conditional normalizing flows (Gerdes et al., 2022)

Planned Features

With the help of the community, we plan to extend NeuLat into many directions, including following features

- Stochastic normalizing flows (Caselle et al., 2022)
- Conditional normalizing flows (Gerdes et al., 2022)
- Path gradients (Vaitl et al., 2022)

Planned Features

With the help of the community, we plan to extend NeuLat into many directions, including following features

- Stochastic normalizing flows (Caselle et al., 2022)
- Conditional normalizing flows (Gerdes et al., 2022)
- Path gradients (Vaitl et al., 2022)
- Continuous normalizing flows

Planned Features

With the help of the community, we plan to extend NeuLat into many directions, including following features

- Stochastic normalizing flows (Caselle et al., 2022)
- Conditional normalizing flows (Gerdes et al., 2022)
- Path gradients (Vaitl et al., 2022)
- Continuous normalizing flows
- **YOUR FEATURE HERE!**

We want NeuLat to be a community effort! Please reach out to us!

Conclusion I leave

- NeuLat is a software framework for flow-based sampling of LFT.

Conclusion I leave

- NeuLat is a software framework for flow-based sampling of LFT.
- Its primary goal is to be highly customizable and easily accessible.

Conclusion I leave

- NeuLat is a software framework for flow-based sampling of LFT.
- Its primary goal is to be highly customizable and easily accessible.
- It serves as a reference implementation, accelerating reserach.

Conclusion I leave

- NeuLat is a software framework for flow-based sampling of LFT.
- Its primary goal is to be highly customizable and easily accessible.
- It serves as a reference implementation, accelerating reserach.
- NeuLat is meant to be a community-effort.

Before I leave

NeuLat will be available soon at

`https://github.com/neulat/neulat`

Before I leave

NeuLat will be available soon at

`https://github.com/neulat/neulat`

Thank you for your attention!