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 et al.

December 06, 2024 - Taipei
Lattice Field Theory and Machine Learning Workshop

code: `https://github.com/neulat/neulat`
preprint: LATTICE2023 PoS 286

Sampling from Boltzmann Distributions

$$p(\phi) = \frac{\exp\{-S[\phi]\}}{Z}$$

Sampling from Boltzmann Distributions.... with Generative Models

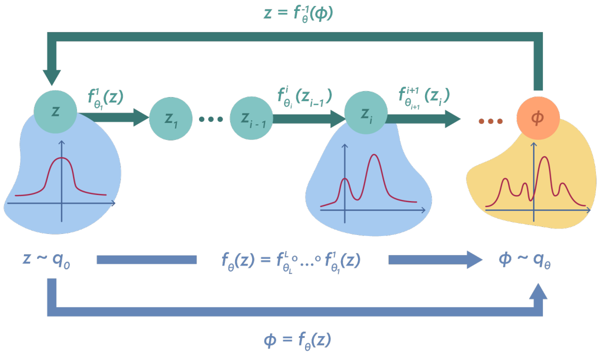
$$p(\phi) = \frac{\exp\{-S(\phi)\}}{Z} \xrightarrow{q_\theta} p(\phi) \approx q_\theta \sim \phi_i$$

Sampling from Boltzmann Distributions.... with Generative Models

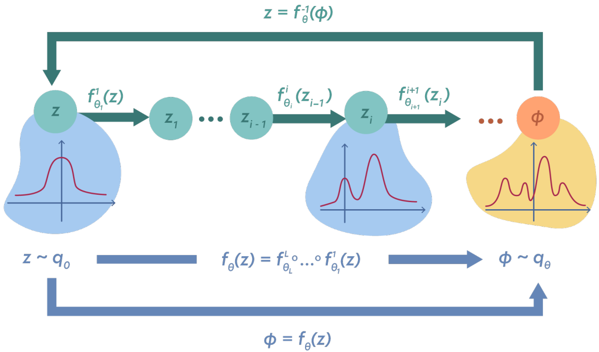
$$p(\phi) = \frac{\exp\{-S(\phi)\}}{Z} \xrightarrow{q_\theta} p(\phi) \approx q_\theta \sim \phi_i$$

Bare with me for now, and wait for Tej's Mathis' and Fernando's talks tomorrow!

Sampling with Normalizing Flows



Sampling with Normalizing Flows



$$q_\theta(\phi) = q_0(f_\theta^{-1}(\phi)) \left| \det \left(\frac{\partial f_\theta}{\partial z} \right) \right|^{-1}$$

What is NeuLat?

Christopher J. Anders

PostDoc at RIKEN AIP, Tokyo

What is NeuLat?

Christopher J. Anders

PostDoc at RIKEN AIP, Tokyo

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

What is NeuLat?

Christopher J. Anders

PostDoc at RIKEN AIP, Tokyo

- 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?

Christopher J. Anders

PostDoc at RIKEN AIP, Tokyo

- 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?

Christopher J. Anders

PostDoc at RIKEN AIP, Tokyo

- 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?

Christopher J. Anders

PostDoc at RIKEN AIP, Tokyo

- 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?

Christopher J. Anders

PostDoc at RIKEN AIP, Tokyo

- 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., Phys. Rev. E (2020)]

What is NeuLat?

Christopher J. Anders

PostDoc at RIKEN AIP, Tokyo

- 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., Phys. Rev. E (2020)]
  - thermodynamic observables [Nicoli et al., Phys. Rev. Lett. (2021)]

What is NeuLat?

Christopher J. Anders

PostDoc at RIKEN AIP, Tokyo

- 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., Phys. Rev. E (2020)]
  - thermodynamic observables [Nicoli et al., Phys. Rev. Lett. (2021)]
  - mode-dropping estimators [Nicoli et al., Phys. Rev. D (2023)]

What is NeuLat?

Christopher J. Anders

PostDoc at RIKEN AIP, Tokyo

- 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., Phys. Rev. E (2020)]
  - thermodynamic observables [Nicoli et al., Phys. Rev. Lett. (2021)]
  - mode-dropping estimators [Nicoli et al., Phys. Rev. D (2023)]
  - path gradients [Vaitl et al., Mach. Learn. Sci. Tech. (2022)]

What is NeuLat?

Christopher J. Anders

PostDoc at RIKEN AIP, Tokyo

- 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., Phys. Rev. E (2020)]
  - thermodynamic observables [Nicoli et al., Phys. Rev. Lett. (2021)]
  - mode-dropping estimators [Nicoli et al., Phys. Rev. D (2023)]
  - path gradients [Vaitl et al., Mach. Learn. Sci. Tech. (2022)]
  - trivializing maps with flows [Bacchio et al., Phys. Rev. D (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 LFT [Albergo et al., 2101.08176 (2021)]

Why NeuLat?

There are already great tools available!

- Introduction to Normalizing Flows for LFT [Albergo et al., 2101.08176 (2021)]
- Flows enhanced HMC

Why NeuLat?

There are already great tools available!

- Introduction to Normalizing Flows for LFT [Albergo et al., 2101.08176 (2021)]
- Flows enhanced HMC
  - **fthmc** (Sam Foreman et al.) [github.com/nftqcd/fthmc] (S. Foreman)
  - **l2hmc-qcd** (Sam Foreman et al.) [github.com/saforem2/l2hmc-qcd] (S. Foreman)

Why NeuLat?

There are already great tools available!

- Introduction to Normalizing Flows for LFT [Albergo et al., 2101.08176 (2021)]
- Flows enhanced HMC
  - **fthmc** (Sam Foreman et al.) [github.com/nftqcd/fthmc] (S. Foreman)
  - **l2hmc-qcd** (Sam Foreman et al.) [github.com/saforem2/l2hmc-qcd] (S. Foreman)
- Normalizing Flows
  - **nflows** [github.com/VincentStimper/normalizing-flows] (V. Stimper at al.)

Why NeuLat?

There are already great tools available!

- Introduction to Normalizing Flows for LFT [Albergo et al., 2101.08176 (2021)]
- Flows enhanced HMC
  - **fthmc** (Sam Foreman et al.) [github.com/nftqcd/fthmc] (S. Foreman)
  - **l2hmc-qcd** (Sam Foreman et al.) [github.com/saforem2/l2hmc-qcd] (S. Foreman)
- Normalizing Flows
  - **nflows** [github.com/VincentStimper/normalizing-flows] (V. Stimper at al.)
- Flows/HMC for LFT
  - **GomalizingFlow.jl** [github.com/AtelierArith/GomalizingFlow.jl] (A. Tomiya)
  - **NeuMC** [github.com/nmcmc/nmcmc-code] (P. Bialas)

Why NeuLat?

There are already great tools available!

- Introduction to Normalizing Flows for LFT [Albergo et al., 2101.08176 (2021)]
- Flows enhanced HMC
  - **fthmc** (Sam Foreman et al.) [github.com/nftqcd/fthmc] (S. Foreman)
  - **l2hmc-qcd** (Sam Foreman et al.) [github.com/saforem2/l2hmc-qcd] (S. Foreman)
- Normalizing Flows
  - **nflows** [github.com/VincentStimper/normalizing-flows] (V. Stimper at al.)
- Flows/HMC for LFT
  - **GomalizingFlow.jl** [github.com/AtelierArith/GomalizingFlow.jl] (A. Tomiya)
  - **NeuMC** [github.com/nmcmc/nmcmc-code] (P. Bialas)

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, NE-HMC, etc.)
  - Normalizing Flow framework
    - Neural Importance Sampling (NIS)
    - Neural HMC

Core Features of NeuLat: Based on PyTorch

- **Density Estimator**: Learn approximations of targeted Boltzmann distributions
- **Sampling**:
  - various MCMC implementations (HMC, Cluster algorithms, NE-HMC, etc.)
  - Normalizing Flow framework
    - Neural Importance Sampling (NIS)
    - Neural HMC
- **Estimation**:
  - Asymptotically unbiased estimators for physical observables [Nicoli et al. (2020)]
  - Direct estimation of thermodynamic observables [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, NE-HMC, etc.)
  - Normalizing Flow framework
    - Neural Importance Sampling (NIS)
    - Neural HMC
- **Estimation**:
  - Asymptotically unbiased estimators for physical observables [Nicoli et al. (2020)]
  - Direct estimation of thermodynamic observables [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, NE-HMC, etc.)
  - Normalizing Flow framework
    - Neural Importance Sampling (NIS)
    - Neural HMC
- **Estimation**:
  - Asymptotically unbiased estimators for physical observables [Nicoli et al. (2020)]
  - Direct estimation of thermodynamic observables [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

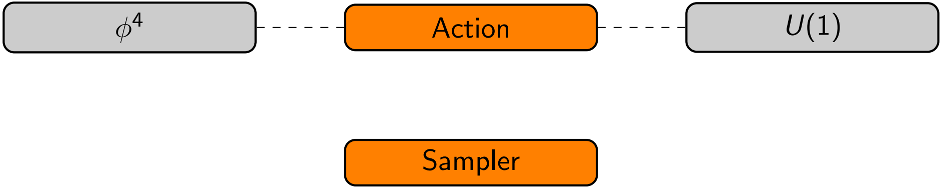
Action

Actions $S[U]$ define target Boltzmann distributions $p(U) \propto e^{-S[U]}$

NeuLat Overview

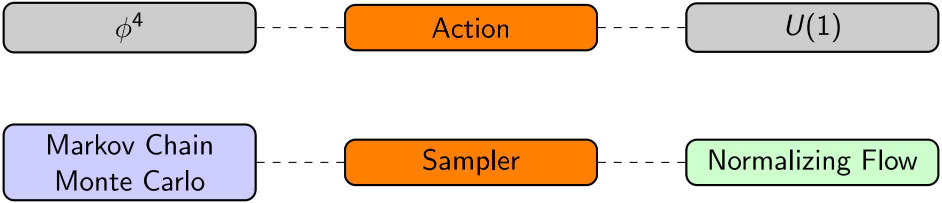
Actions are, e.g., ϕ^4 and $U(1)$

NeuLat Overview



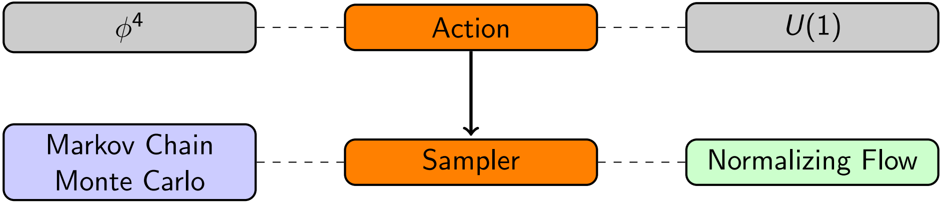
Samplers are **anything** that can be sampled from

NeuLat Overview



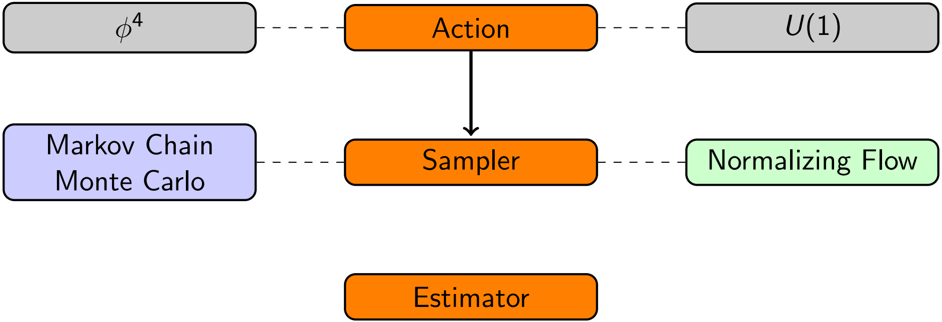
Samplers are, e.g., MCMCs, Flows, $\mathcal{N}(0, 1)$

NeuLat Overview



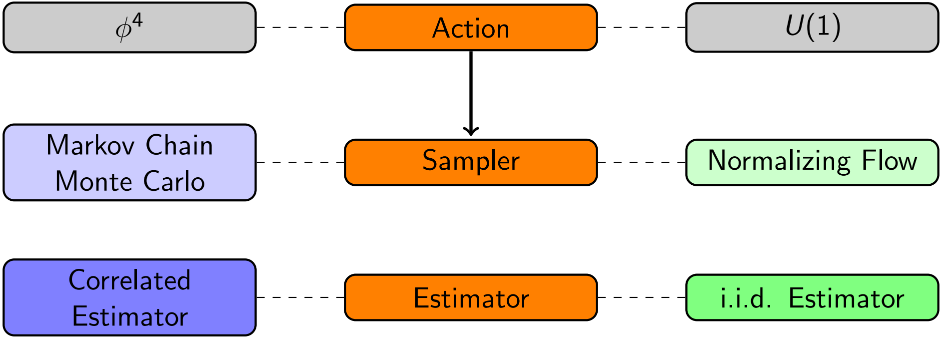
Samplers **require Actions**

NeuLat Overview



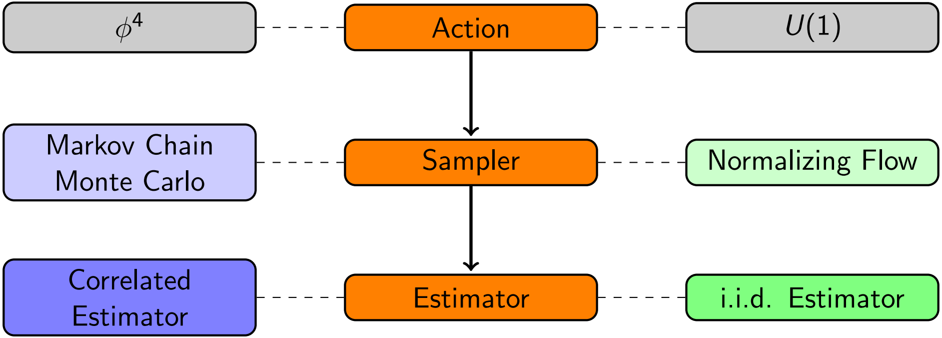
Estimators are used to estimate observables

NeuLat Overview



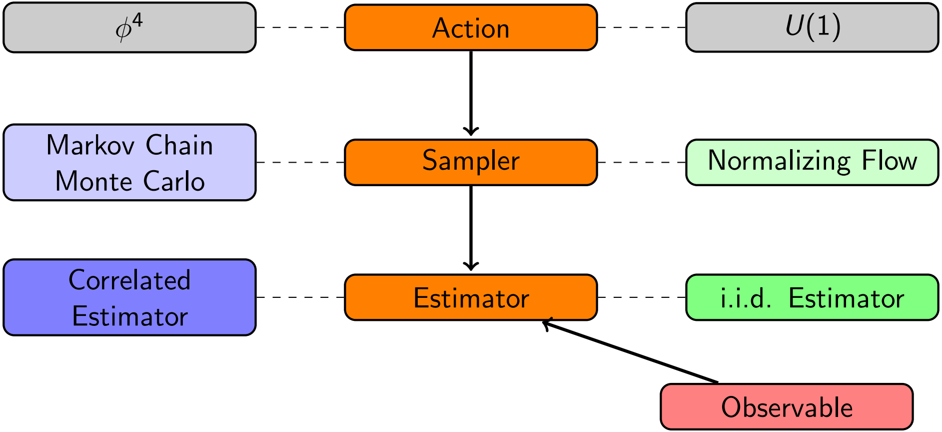
Estimators are, e.g., **i.i.d** or **correlated**, based on the samples

NeuLat Overview



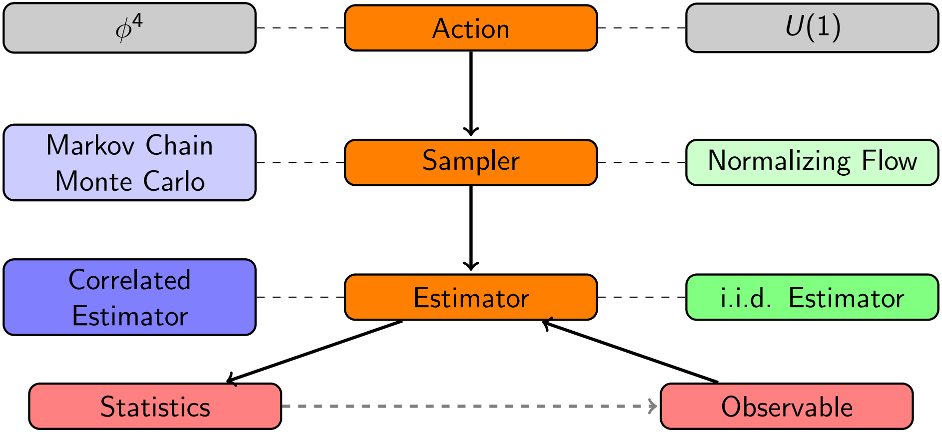
Estimators require **samples** from Samplers

NeuLat Overview



Observables, e.g., Magnetization, Topological Charge, etc., are used by the Estimator

NeuLat Overview



The resulting Statistics are **estimations** for the Observables

Actions

Actions are classes (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)
```

Actions

Actions are classes (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 the ϕ^4 action reads:

```python
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 = (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 are anything from which we can **sample**.

Samplers

At the core of NeuLat are `Samplers`, which are anything from which we can **sample**.

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

```
from neulat.sampler.distribution import Normal

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

Sampling

Samplers can sample, 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 sample, 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)`.

- `sample_shape` is the number of samples, supporting arbitrary shapes
- `feature_shape` is the shape of the lattice (i.e., # of dimensions)

Hamiltonian Monte Carlo

A more involved `Sampler` is the HMC:

```python
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 a trajectory
    traj_steps=20,  # number of steps in the trajectory
    bias=0.0,  # bias in initialization
)
```

HMC Sampling 1/2

As for any `Sampler`, we can sample from HMC

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

HMC Sampling 1/2

As for any `Sampler`, we can sample from HMC

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

However, HMC does not implement `logprob` and, by extension, `sample_with_logprog`, as no normalized probabilities are available. (unnormalized logprobs possible)

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

HMC Sampling 2/2

One can also iterate over HMC chains to sample

```python
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 2/2

One can also iterate over HMC chains to sample

```python
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, HMC chains are infinite iterators. !!

Normalizing Flows

Normalizing flows [Papamakarios et al., JMLR (2021)] 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 [Papamakarios et al., JMLR (2021)] 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 1/2

Transforms are (optionally) invertible PyTorch modules, and require (either or both) a `forward` and a `inverse`.

Normalizing Flows: Transforms 1/2

Transforms are (optionally) invertible PyTorch modules, and require (either or both) 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, 0.
    @withlogdet
    def inverse(self, input):
        return -input, 0.
```

Normalizing Flows: Transforms 1/2

Transforms are (optionally) invertible PyTorch modules, and require (either or both) 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, 0.
    @withlogdet
    def inverse(self, input):
        return -input, 0.
```

Decorator `@withlogdet` ensures the logdet is accumulated between transforms.

Normalizing Flows: Transforms 2/2

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 2/2

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

```python
from sampler.flow.base import SequentialTransform

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

For common *Coupling* Flows, e.g., NICE, RealNVP, 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
)
```

- 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 **alt***ernating* **flat***tened 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: Partitioners

A very simple partitioner is the `AltFlatPartitioner`, which stands for **alt***ernating* **flat***tened 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 $\begin{pmatrix} 11 & 2 \\ 13 & 4 \end{pmatrix}$
- **N.B.** Inputs can be partitioned into more than two parts, e.g., active, passive, frozen.

Coupling Flows: Flipping Partitioners

Partitioners usually flip the active and passive 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 passive 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` defines the *conditioner* Θ that transforms the passive input:

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

```
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: $h(\mathbf{x}_{\text{active}}, \mathbf{x}_{\text{passive}}) = \mathbf{x}_{\text{active}} + m_\theta(\mathbf{x}_{\text{passive}})$

```python
class NICE(Coupling):
    @withlogdet
    @partitioned
    def forward(self, active, passive):
        return active + self.net(passive), 0.

    @withlogdet
    @partitioned
    def inverse(self, active, passive):
        return active - self.net(passive), 0.
```

Coupling Flows: Defining Couplings

The coupling `Transform` itself is mostly only concerned with implementing the coupling function h. E.g. in NICE: $h(\mathbf{x}_{\text{active}}, \mathbf{x}_{\text{passive}}) = \mathbf{x}_{\text{active}} + m_\theta(\mathbf{x}_{\text{passive}})$

```python
class NICE(Coupling):
    @withlogdet
    @partitioned
    def forward(self, active, passive):
        return active + self.net(passive), 0.

    @withlogdet
    @partitioned
    def inverse(self, active, passive):
        return active - self.net(passive), 0.
```

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: $h(\mathbf{x}_{\text{active}}, \mathbf{x}_{\text{passive}}) = \mathbf{x}_{\text{active}} + m_\theta(\mathbf{x}_{\text{passive}})$

```python
class NICE(Coupling):
    @withlogdet
    @partitioned
    def forward(self, active, passive):
        return active + self.net(passive), 0.

    @withlogdet
    @partitioned
    def inverse(self, active, passive):
        return active - self.net(passive), 0.
```

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

Coupling Flows: Assembling 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: Assembling 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)
  )
)
```

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

Coupling Flows: Global Scaling

Global scaling is a `Transform` that scales all elements of the input tensor by the **same** scalar value.
This is necessary to enhance the convergence of the model:

```
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)
    ) + GlobalScaling(feature_ndim=feature_ndim)
)
```

Coupling Flows: Global Scaling

Global scaling is a `Transform` that scales all elements of the input tensor by the **same** scalar value.
This is necessary to enhance the convergence of the model:

```
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)
    ) + GlobalScaling(feature_ndim=feature_ndim)
)
```

Note: `Transform` has an add method that allows simple concatenatenation of different types of transforms.

Coupling Flows: Traning

Training of the flow with, e.g., ReverseKL, is straightforward:

```python
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`
    torch.nn.utils.clip_grad_norm_(model.transform.parameters(), 1.0) # clipping
    optim.step()
```

Estimating Observables: IidEstimator

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

```python
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:

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

Estimating Observables: `CorrelatedEstimator`

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

```
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), ...}`

Estimating Observables: `ImportanceSamplingEstimator`

To obtain an unbiased estimator, Nicoli et al., (2020) proposed to use **Neural Importance Sampling (NIS)**.
This additionally requires the logprobs from the flow, as well as the specific action:

```python
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.named_evaluate(flow_configs, flow_logprobs)
```

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

`{'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

- Non-Equilibirum MCMC [Caselle et al., Phys. Rev. D (2016)]

Planned Features

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

- Non-Equilibirum MCMC [Caselle et al., Phys. Rev. D (2016)]
- Continuous normalizing flows [Chen et al., NeurIPS (2018)]

Planned Features

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

- Non-Equilibirum MCMC [Caselle et al., Phys. Rev. D (2016)]
- Continuous normalizing flows [Chen et al., NeurIPS (2018)]
- Stochastic normalizing flows [Caselle et al., JHEP (2022)]

Planned Features

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

- Non-Equilibirum MCMC [Caselle et al., Phys. Rev. D (2016)]
- Continuous normalizing flows [Chen et al., NeurIPS (2018)]
- Stochastic normalizing flows [Caselle et al., JHEP (2022)]
- Path gradients (WIP) [Vaitl et al., Mach. Learn. Sci. Tech. (2022)]

Planned Features

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

- Non-Equilibirum MCMC [Caselle et al., Phys. Rev. D (2016)]
- Continuous normalizing flows [Chen et al., NeurIPS (2018)]
- Stochastic normalizing flows [Caselle et al., JHEP (2022)]
- Path gradients (WIP) [Vaitl et al., Mach. Learn. Sci. Tech. (2022)]
- Conditional normalizing flows [Gerdes et al., SciPost Phys. (2023)]

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

Planned Features

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

- Non-Equilibirum MCMC [Caselle et al., Phys. Rev. D (2016)]
- Continuous normalizing flows [Chen et al., NeurIPS (2018)]
- Stochastic normalizing flows [Caselle et al., JHEP (2022)]
- Path gradients (WIP) [Vaitl et al., Mach. Learn. Sci. Tech. (2022)]
- Conditional normalizing flows [Gerdes et al., SciPost Phys. (2023)]
- Mode-Dropping Estimator [Nicoli et al., Phys. Rev. D (2023)]

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

Planned Features

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

- Non-Equilibirum MCMC [Caselle et al., Phys. Rev. D (2016)]
- Continuous normalizing flows [Chen et al., NeurIPS (2018)]
- Stochastic normalizing flows [Caselle et al., JHEP (2022)]
- Path gradients (WIP) [Vaitl et al., Mach. Learn. Sci. Tech. (2022)]
- Conditional normalizing flows [Gerdes et al., SciPost Phys. (2023)]
- Mode-Dropping Estimator [Nicoli et al., Phys. Rev. D (2023)]
- **YOUR FEATURE HERE!**

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

Conclusion

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

Conclusion

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

Conclusion

- 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

- 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.

Conclusion

- 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.

And now it's time for a real demo...

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!