Introducing Flax NNX:

A Pythonic Neural Network Library for JAX

Helping PyTorch users learn NNX and JAX

What is Flax NNX?

- Flax NNX
  - A neural network library for JAX, designed for flexibility and high performance
  - New users are encouraged to use Flax NNX
- Flax Linen (Released in 2020)
  - Original neural network library for JAX
  - Focuses on functional programming

Why Flax NNX?

- Aims to simplify neural network development in JAX.
- **NNX is Pythonic**: Regular Python semantics for Modules, including support for mutability and shared references.
- Emphasizes explicit state management.
- Enables reference sharing and mutability.

Key Design Principles

- Pythonic Interface
  - Use regular Python objects for defining networks.
- Explicit State Management
  - Deliberate control over parameters and mutable variables.
- Python Graph Data Structure
  - Enables reference sharing and mutability.

Benefits for PyTorch Users

- Performance
- Smoother transition into the JAX ecosystem.
- API design shares conceptual similarities with PyTorch.
- Familiar patterns for defining neural network architectures.

Core Concepts

Core Concept: Modules

- `nnx.Module` is the fundamental building block.
- Modules directly hold their own state (parameters).

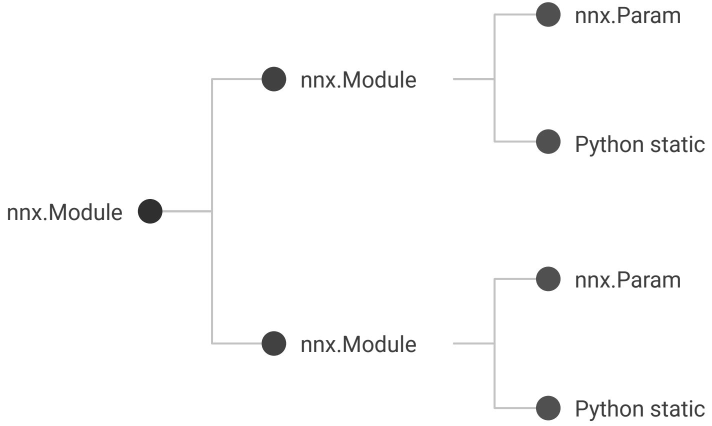
Code Example: Random Layer

```python
from flax import nnx
import jax
import jax.numpy as jnp

class RandomLayer(nnx.Module):
  def __init__(self, size: int, *, rngs: nnx.Rngs):
    self.random_vector = nnx.Param(jax.random.normal(rngs.params(),
                                                       (size,)))

  def __call__(self):
    return self.random_vector.value
```

Flax NNX Python Graphs

Bridging JAX Power with Python Familiarity

- NNX modules (layers, models, etc.) are regular Python objects that are also registered as JAX Pytrees
- Intuitive, object-oriented feel of Python while enabling seamless integration with JAX's functional transformations like **`jax.jit`** and **`jax.vmap`**
- Modules can be composed into nested (graph) structures to express complex models



Flax NNX Python Graphs

Bridging JAX Power with Python Familiarity

```
nnx.display(Linear(2, 5, rngs=nnx.Rngs(params=0)))
```

```
▼ Linear( # Param: 15 (60 B)
    w=▸Param(value=<jax.Array float32(2, 5) ≈0.42 ±0.34 [≥0.029, ≤0.95] nonzero:10>),
    b=▸Param(value=<jax.Array float32(5,) ≈0.0 ±0.0 [≥0.0, ≤0.0] zero:5>),
    din=2,
    dout=5,
)
```

```
▼ MLP( # Param: 165 (660 B), BatchStat: 32 (128 B), RngState: 2 (12 B), Total: 199 (800 B)
  linear1=▼ Linear( # Param: 48 (192 B)
    w=▶ Param(value=<jax.Array float32(2, 16) ≈0.51 ±0.33 [≥0.0059, ≤0.97] nonzero:32>),
    b=▶ Param(value=<jax.Array float32(16,) ≈0.0 ±0.0 [≥0.0, ≤0.0] zero:16>),
    din=2,
    dout=16,
  ),
  dropout=▼ Dropout( # RngState: 2 (12 B)
    rate=0.1,
    broadcast_dims=(),
    deterministic=False,
    rng_collection='dropout',
    rngs=▼ Rngs( # RngState: 2 (12 B)
      default=▶ RngStream(key=RngKey(value=<jax.Array key<fry>()>, tag='default'), count=R
    ),
  ),
  bn=▶ BatchNorm(mean=BatchStat(value=<jax.Array float32(16,) ≈0.01 ±0.0051 [≥0.0013, ≤0.0
  linear2=▶ Linear(w=Param(value=<jax.Array float32(16, 5) ≈0.51 ±0.3 [≥0.0061, ≤1.0] nonz
)
```

Data Structure: Python Graphs

- **Direct Attribute Access**: Access layers and parameters like `model.layer1`, `model.layer1.weight`
- **Standard Python References**: Assigning a layer to multiple attributes means they share the exact same object, just like in Python. Useful for weight sharing!
- **Mutability**: Module state can be modified directly

Python Graphs in Practice: Like `torch.nn.Module` but for JAX

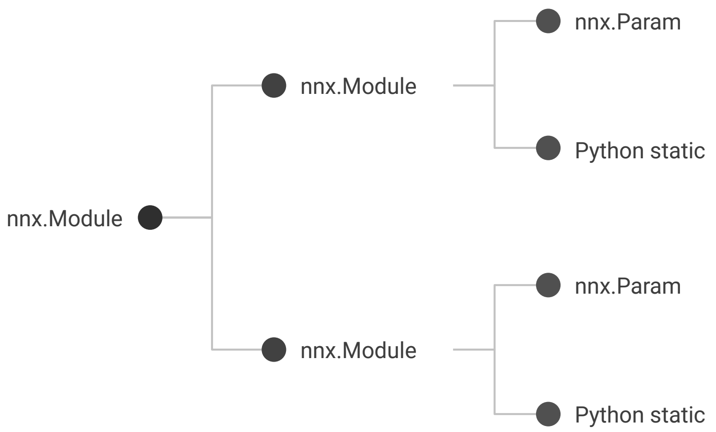
Handling State:

- **Static Config**: Regular Python attributes
  - e.g., `self.dropout_rate = 0.5`
- **Dynamic State (Parameters/Buffers)**: Stored in special `nnx.Variable` container objects
  - e.g., `nnx.Param` for trainable weights
- Access the actual JAX array data via `.value`
  - e.g., `self.weight.value`

Flax NNX Python Graphs

Bridging JAX Power with Python Familiarity

- Why Python Graphs?
  - **Mutability**: Allowing objects (like model layers) to change their internal state directly.
  - **Reference Sharing**: Ensuring that if you assign the same layer object to multiple places in your model, it's truly the same object instance.



Python Graphs in Practice: Like `torch.nn.Module` but for JAX

Reference Sharing:

- Assigning the same module or **`nnx.Variable`** instance to multiple attributes creates shared references, identical to Python's standard behavior

```
shared_layer = nnx.Linear(...)
model.encoder_layer = shared_layer
model.decoder_layer = shared_layer # Both point to the same layer.
```

Variable Types

`Variables` are wrappers over `jax.Arrays` that enable state updates and allow associated metadata.

- `nnx.Param`: Learnable parameters of the model (dynamic state).
- `nnx.BatchStat`, `nnx.Cache`, `nnx.Intermediate`: For specialized state.
- Also `nnx.State` (not a `Variable` type): A pytree that contains a pure version of the object's state.

Code Example: Stateful Parameter Layer

```python
from flax import nnx
import jax.numpy as jnp

class StatefulParameterLayer(nnx.Module):
  def __init__(self, initial_value: float, *, rngs: nnx.Rngs):
    self.weight = nnx.Param(jax.random.uniform(rngs.params()))
    self.bias = {'bias': jnp.array(initial_value)}

  def update_bias(self, new_value: float):
    self.bias['bias'] = jnp.array(new_value)

  def __call__(self, x: jax.Array):
    return x * self.weight.value + self.bias['bias']
```

Explicit Random Number Generation

- Uses **`nnx.Rngs`** object.
- Requires explicit passing of a PRNG key when instantiating modules with randomness.
- Promotes reproducibility and easier parallelization.
- Layers store a **forked copy** of RNGs, ensuring state isolation.

Eager Parameter Initialization

- Parameters are initialized immediately when an `nnx.Module` is instantiated.
- All shape information must be provided during initialization.
- No implicit shape inference.

Functional API: split, merge, update

- **`nnx.split`**: Decomposes an **`nnx.Module`** into its static structure (**`GraphDef`**) and dynamic state (**`State`**).
- **`nnx.merge`**: Reconstructs an **`nnx.Module`** from its **`GraphDef`** and **`State`**.
- **`nnx.update`**: Updates an existing object in-place with the content of a given **`State`**.

Code Example: Counter with Functional API

```python
class Counter(nnx.Module):
  def __init__(self):
    self.count = 0

  def __call__(self):
    self.count += 1
    return self.count

counter = Counter()
print(f'{counter() = }') # 1
graphdef, state = nnx.split(counter)

@jax.jit
def count(state):
  counter = nnx.merge(graphdef, state)
  counter() # 2
  return nnx.state(counter)

state = count(state)
nnx.update(counter, state)
print(f'{counter() = }') # 3
```

nnx.jit: When to Use NNX's Supercharger

- **What it does**: `nnx.jit` ("Just-In-Time" compilation) takes your Python function and uses XLA to compile it into highly optimized code for CPU/GPU/TPU.
- **Think**: Similar goal to `torch.compile()` – speed up execution.
- **The Catch**: You shouldn't just `jit()` every function! Performance gains come from JITting *the right* functions.
- **Key Requirement #1**: `nnx.jit` works best on Pure Functions. We'll define this next.
- **Goal**: Identify the computationally heavy, pure parts of your code and `jit()` those.

nnx.jit: NNX's Supercharger

```python
import jax
import jax.numpy as jnp
from flax import nnx
import torch

jax_w = jnp.ones((3, 4))
jax_b = jnp.ones(3)
jax_x = jnp.ones(4)

pt_w = torch.ones((3, 4))
pt_b = torch.ones(3)
pt_x = torch.ones(4)

def jax_predict(W, b, x):
  return jnp.matmul(W, x) + b

def pt_predict(W, b, x):
  return torch.matmul(W, x) + b
```

nnx.jit: NNX's Supercharger

```
jit_predict = nnx.jit(jax_predict)

def jax_batch(W, b, x):
  for i in range(10000):
    result = jax_predict(W, b, x)

def pt_batch(W, b, x):
  for i in range(10000):
    result = pt_predict(W, b, x)

%timeit pt_batch(pt_w, pt_b, pt_x) # PyTorch
%timeit jit_predict(jax_w, jax_b, jax_x).block_until_ready() # JAX with XLA compile
%timeit jit_predict(jax_w, jax_b, jax_x).block_until_ready() # NNX with jit()
```

```
79.9 ms ± 12.5 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
The slowest run took 6.12 times longer than the fastest. This could mean
that an intermediate result is being cached.
79.4 µs ± 72.3 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)
39.9 µs ± 756 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

nnx.jit: What is a "Pure Function"? (The Key to JIT)

Definition: A function is pure iff:

- It always returns the same output for the same inputs
- It has no side effects (we'll talk about this next)
- **The Rule: Only apply nnx.jit to functions that are pure**
- (Good Practice: Writing pure functions is often good design anyway – easier to test and debug!)*

nnx.jit: What is a "Pure Function"? (The Key to JIT)

What are Side Effects?

- Actions that affect things outside the function's direct input/output:
  - Modifying global variables
  - Changing mutable inputs (e.g., appending to a list passed as an argument)
  - Printing to the console, writing to files (I/O)
  - Changing system settings, interacting with databases

Another Reason Not to jit: Python Control Flow

The Scenario: Your function uses standard Python **`if`**, **`while`**, **`for`** loops where the condition or iteration *depends directly on the value* of an input argument.

```
# Example NOT ideal for direct jit()

def process_value(x, threshold):
  if x > threshold: # Control flow depends on input values
    return x * 2
  else:
    return x / 2
```

Another Reason Not to jit: Python Control Flow

Why it's Problematic for `jit()`:

- JAX traces the function based on the shapes and types of inputs, not specific values (usually)
- A standard Python `if` depending on a value creates a specific path during tracing
- If you call the JITted function later with a value that takes a different path, JAX might error or need to recompile, which can be slow, losing the speed benefit

Another Reason Not to jit: Python Control Flow

The Rule: Avoid JITting functions where standard Python control flow (if/while) depends directly on input values.

Strategy:

- Isolate the pure, numerically intensive parts within the branches/loops into separate functions
- `jit()` those smaller, pure functions
- Keep the Python control flow logic outside the JITted functions

JAX Transformations & The Stateful Model Challenge

Standard JAX Transformations (`jax.jit`, `jax.grad`, `jax.vmap`)

- Designed for **pure functions**:
  - No side effects (don't modify external state).
  - Deterministic output for the same input.
- Operate on **PyTrees** (nested lists, dicts, tuples containing arrays).
- Require **explicit state management**: You must manually pass state (like model parameters, optimizer state, RNG keys) into the function and return the updated state.

JAX Transformations & The Stateful Model Challenge

NNX Modules (and PyTorch Modules) are stateful:

- NNX Modules (like `torch.nn.Module`) are inherently stateful. They contain their parameters and potentially other mutable state (e.g., BatchNorm stats).
- Directly applying `jax.jit()` or `jax.grad()` to a method of a stateful object requires manually extracting the state, passing it through the pure function, and then updating the object with the returned state. This can be cumbersome.

NNX Transformations

- `nnx.jit, nnx.grad, nnx.vmap`
- Wrappers around standard JAX transformations
  - jax.jit, jax.grad, jax.vmap, etc.
- Specifically designed to work directly with NNX graph objects
  - `nnx.Module, nnx.Optimizer, nnx.Rngs`, etc.

The Key Difference: Automatic State Management

JAX Transforms: Require YOU to handle state explicitly (pass in, get back).

```
# Simplified JAX pattern
params, opt_state = ...
grads, new_state = jax.grad(loss_fn, has_aux=True)(params, ...)
updates, opt_state = optimizer.update(grads, opt_state, params) # Pass opt_state in, get it back
params = apply_updates(params, updates)
```

NNX Transforms: Handle state lifting and updating automatically behind the scenes when applied to methods of NNX objects.

```
# Simplified NNX pattern
model, optimizer, rngs = ... # NNX objects holding state
@nnx.grad # Applied to a method or function working with NNX objects
def loss_fn_nnx(...): ...
grads = loss_fn_nnx(...)
optimizer.update(grads) # Updates parameters within model object directly
```

When to Use NNX vs. JAX Transformations

Use NNX Transformations (**`nnx.jit`**, **`nnx.grad`**, etc.) when:

- You are working with NNX graph objects (**`nnx.Module`**, **`nnx.Optimizer`**, **`nnx.Variable`**, **`nnx.Rngs`**). This is the primary use case.
- You want simplified state management and less boilerplate code.
- You prefer a more object-oriented style where transformations apply directly to methods interacting with stateful objects.

When to Use NNX vs. JAX Transformations

Use Standard JAX Transformations (`jax.jit`, `jax.grad`, etc.) when:

- You are working with pure functions that don't involve NNX objects or mutable state directly.
- You are operating on plain **PyTrees** (e.g., data preprocessing functions).
- You need fine-grained, low-level control over the transformation process and state handling (you want to manage state explicitly).
- You need a specific JAX transformation feature that might not yet have an NNX counterpart (less common for core transforms).
- Standard JAX transformations are faster

When to Use NNX vs. JAX Transformations

General Recommendation:

- **If you are using Flax NNX objects, use the corresponding NNX transformations.** They are designed for this purpose and provide a much smoother experience.

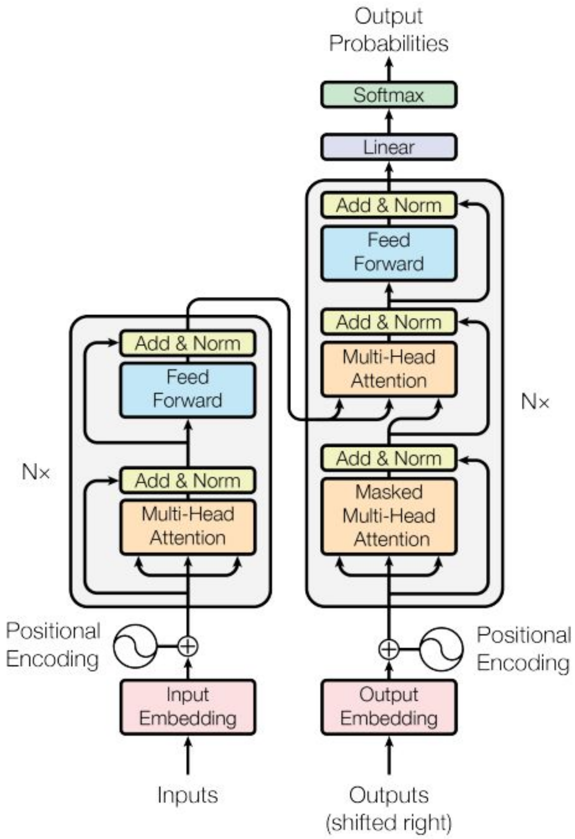
Models

Fundamental Neural Network Layers

- `nnx.Linear`
- `nnx.Conv`
- `nnx.BatchNorm`
- `nnx.LayerNorm`
- `nnx.GroupNorm`
- `nnx.MultiHeadAttention`
- `nnx.LSTMCell`
- `nnx.GRUCell`
- `nnx.Dropout`

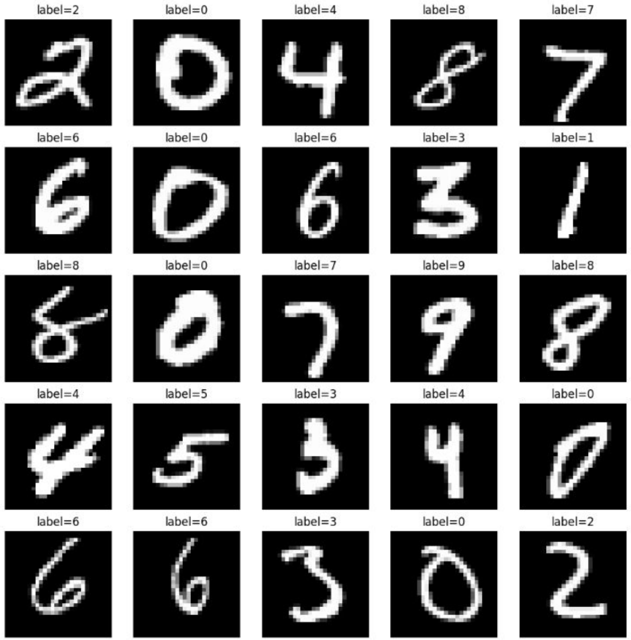
Building Complex Models

- MLPs, CNNs can be easily constructed.
- Requires explicit specification of input and output shapes during initialization.
- Provision of an appropriate random number generator key for parameters.



MNIST Tutorial

- Example of defining a CNN for digit classification using Flax NNX.
- Covers loading the MNIST dataset, defining the CNN model, creating an optimizer using Optax, defining the training loop, and evaluating the model.

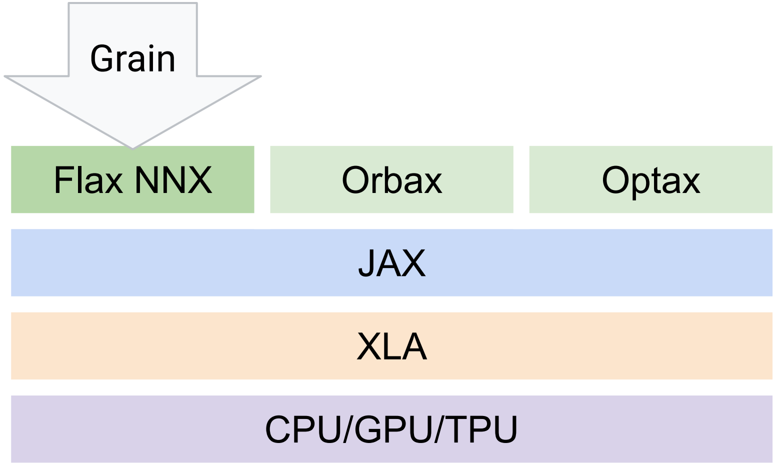


https://flax.readthedocs.io/en/latest/mnist_tutorial.html

Flax NNX and the JAX AI Stack

Flax NNX and the JAX AI Stack

- **JAX**: Array operations and program transformations.
- **Flax NNX**: Building neural networks.
- **Optax**: Gradient processing and optimization.
- **Orbax**: Checkpointing and persistence.
- **ml_dtypes**: NumPy dtype extensions for machine learning.



https://docs.jaxstack.ai

Comparing Flax NNX with PyTorch

Comparison with PyTorch: High-Level

- Both use class-based structures for defining models.
- Similarities in `__init__` and forward pass methods (`__call__` in NNX).
- Key difference: explicit vs. implicit random number generation.

Comparison: State Management

- **PyTorch**: Imperative updates using `zero_grad()` and `step()`.
- **Flax NNX**: Direct attribute assignment and mutation, also enables a functional style with Functional API (split, merge).

Code Example: Shifted ReLU in PyTorch and NNX

```python
# PyTorch
import torch
import torch.nn as nn


class ShiftedReLU_Torch(nn.Module):
    def __init__(self, shift: float):
        super().__init__()
        self.shift = shift


    def forward(self, x):
        return torch.relu(x + self.shift)
```

```python
# Flax NNX
from flax import nnx
import jax.numpy as jnp


class ShiftedReLU_NNX(nnx.Module):
    def __init__(self, shift: float):
        self.shift = shift


    def __call__(self, x: jnp.ndarray):
        return nnx.relu(x + self.shift)
```

Code Example: Simple Classifier in PyTorch and NNX

```python
# PyTorch
import torch
import torch.nn as nn


class SimpleClassifier_Torch(nn.Module):
    def __init__(self, input_size: int,
                 num_classes: int):
        super().__init__()
        self.linear1 = nn.Linear(
                        input_size, 10)
        self.relu = nn.ReLU()
        self.linear2 = nn.Linear(
                        10, num_classes)


    def forward(self, x):
        x = self.linear1(x)
        x = self.relu(x)
        return self.linear2(x)
```

```python
# Flax NNX
from flax import nnx
import jax
import jax.numpy as jnp


class SimpleClassifier_NNX(nnx.Module):
    def __init__(self, input_size: int,
                 num_classes: int, *,
                 rngs: nnx.Rngs):
        self.linear1 = nnx.Linear(
            input_size, 10, rngs=rngs)
        self.relu = nnx.relu
        self.linear2 = nnx.Linear(
            10, num_classes, rngs=rngs)


    def __call__(self, x: jnp.ndarray):
        x = self.linear1(x)
        x = self.relu(x)
        return self.linear2(x)
```

Google

Comparison: Training Loop & Backpropagation

- **PyTorch**: `loss.backward()` for automatic gradient computation, optimizer updates parameters directly.
- **Flax NNX**: `nnx.value_and_grad` to compute gradients and `optimizer.update` to update the model's state with computed gradients.

Comparison: Training Loop & Backpropagation

```python
# PyTorch
import torch
import torch.nn as nn
import torch.optim as optim


# Define a simple model
class SimpleModel_Torch(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(1, 1)

    def forward(self, x):
        return self.linear(x)
```

```python
# Flax NNX
from flax import nnx
import jax
import jax.numpy as jnp
from optax import sgd
from typing import Any


# Define a simple model
class SimpleModel_NNX(nnx.Module):
    def __init__(self, *, rngs: nnx.Rngs):
        self.linear = nnx.Linear(1, 1, rngs=rngs)

    def __call__(self, x: jnp.ndarray):
        return self.linear(x)
```

Comparison: Training Loop & Backpropagation

```python
# PyTorch
model_torch = SimpleModel_Torch()

optimizer_torch = \
  optim.SGD(model_torch.parameters(), lr=0.01)
loss_fn = nn.MSELoss()


# Dummy data
x_torch = torch.tensor(
          [[2.0]], requires_grad=True)
y_torch = torch.tensor([[4.0]])
```

```python
# Flax NNX
model_nnx = SimpleModel_NNX(rngs=nnx.Rngs(0))

# Optimizer
optimizer = nnx.Optimizer(
              model_nnx,
              tx=sgd(learning_rate=0.01),
              wrt=nnx.Param)

# Dummy data
x_nnx = jnp.array([[2.0]])
y_nnx = jnp.array([[4.0]])
```

Comparison: Training Loop & Backpropagation

```python
# PyTorch Training step

# Zero the gradients
optimizer_torch.zero_grad()
output_torch = model_torch(x_torch)
loss_torch = loss_fn(
               output_torch, y_torch)
loss_torch.backward()  # Compute gradients

# Update parameters
optimizer_torch.step()

print("PyTorch Loss:", loss_torch.item())
```

```python
# Flax NNX Training step
@nnx.jit
def train_step(model, optimizer, x, y):
  def loss_fn(model):
    return jnp.mean((model(x) - y) ** 2)

  loss, grads = \
        nnx.value_and_grad(loss_fn)(model)
  # in-place updates
  optimizer.update(model, grads)

  return loss

# Pass the optimizer
loss_nnx = train_step(model_nnx,
                  optimizer, x_nnx, y_nnx)

print("Flax NNX Loss:", loss_nnx)
```

Google

Conclusion

- Flax NNX provides a powerful and intuitive way to build neural networks with JAX.
- **NNX is Pythonic**: Regular Python semantics for Modules, including support for mutability and shared references.

Learning Resources

Code Exercises, Quick References, and Slides

- https://goo.gle/learning-jax

Learn JAX

Community and Docs

Community:

- https://goo.gle/jax-community

Docs

- JAX AI Stack: https://jaxstack.ai
- JAX: https://jax.dev
- Flax NNX: https://flax.readthedocs.io