NAIRR Pilot National Artificial Intelligence Research Resource Pilot

Introduction to PyTorch

Lonnie D. Crosby, Ph.D.
Group Leader, Scientific Computing - NICS
University of Tennessee-Knoxville

April 2, 2025 / Track 2 – Intermediate to Advanced
AI Workshop Denver, CO April 2-3, 2025

Introduction to Pytorch

What is PyTorch?

PyTorch is a **Python tensor** and **deep learning** library using GPUs and CPUs for computation. [1]

From the pytorch/pytorch GitHub repository's README: [2]
- Tensor computation library, analogous to Numpy, that can leverage GPU acceleration.
- Supports Dynamic Neural Networks via tape-based autograd functionality.

Objectives for this session:
- PyTorch Tensors and auto differentiation
- Building PyTorch models
- PyTorch Datasets and DataLoaders
- Training PyTorch models (Optimization)
- Saving and Loading PyTorch models
- Hands-on-Exercises

[1] The Linux Foundation, "PyTorch Documentation," pytorch.org. https://pytorch.org/docs/stable/index.html (accessed Mar. 6, 2025).

[2] "GitHub – pytorch/pytorch," github.com. https://github.com/pytorch/pytorch?tab=readme-ov-file (accessed Mar. 6, 2025).

Note about source materials

PyTorch Tutorials (https://pytorch.org/tutorials/) [1]

Much of the content of this session comes from the various tutorials published on the PyTorch website (https://pytorch.org/tutorials/). These tutorials cover topics such as:

- PyTorch Recipes
- Introduction to PyTorch
- Learning PyTorch
- Image and Video
- Audio
- Deploying PyTorch Models in Production

Tutorials may include content such as Microsoft Learn, Google Colab, Jupyter Notebooks, or content on GitHub.

[1] The Linux Foundation, "Welcome to PyTorch Tutorials – PyTorch Tutorials 2.6.0 +cu124 documentation," pytorch.org. https://pytorch.org/tutorials (accessed Mar. 12, 2025).

Overview – Big Picture

Overview:
The objective in this session is to introduce you to PyTorch as a tool to implement and optimize deep learning models using Python. The building blocks for this will be the construction of a model that maps some input to some output. **X -> Y**

Input: X May be a vector of numbers (integers or floats), an image (vector of pixel values), etc..
Map: -> PyTorch model, which may be a deep neural network, convolutional neural network, etc..
Output: Y May be a result (integer, float), a classification (prob. of membership in a class), etc..

Map: PyTorch Model that contains parameters that can be optimized to improve the performance of the model.
Optimization: Comparison of result, **Y**, with expected result, $\mathbf{Y^0}$.

Project Workflow
Projects begin with Data ($\mathbf{X^0}$,$\mathbf{Y^0}$) - Split into Train, Validate, and Test sets
Training Models with the Training set.
Evaluating different Models with the Validation set.
Determining final model performance with the Test set.

PyTorch Tensors

What is a PyTorch Tensor?

Tensors are generalized mathematical objects with zero or more indices each consisting of an appropriate number of dimensions. [1,2]

Some Special Cases are:

- Scalars (magnitude) –> rank-0 tensors: $a \in \mathbb{R}$
- Vectors (magnitude, direction) –> rank-1 tensor: $\mathbf{a} \in \mathbb{R}^n$, where elements are $a_i \mid i \in \{1, 2, \ldots, n\}$
- Matrix (mapping between two vector spaces) –> rank-2 tensor: $\mathbf{A} \in \mathbb{R}^{n \times m}$, where elements are $A_{ij} \mid i \in \{1, 2, \ldots, n\}, j \in \{1, 2, \ldots, m\}$

More Generally, tensors can have any number of dimensions (ranks) with arbitrary numbers of elements each.

$\mathbf{A} \in \mathbb{R}^{n \times m \times p \times q}$,where elements are $A_{ijkl} \mid i \in \{1, 2, \ldots, n\}, j \in \{1, 2, \ldots, m\}, k \in \{1, 2, \ldots, p\}, l \in \{1, 2, \ldots, q\}$

[1] Merriam-Webster, Inc., "TENSOR Definition & Meaning," merriam-webster.com. https://www.merriam-webster.com/dictionary/tensor (accessed Mar. 12, 2025).

[2] Wolfram, "Tensor – from Wolfram MathWorld", wolfram.com. https://mathworld.wolfram.com/Tensor.html (accessed Mar. 14, 2025)

[3] The Linux Foundation, "Tensors– PyTorch Tutorials 2.6.0 +cu124 documentation," pytorch.org. https://pytorch.org/tutorials/beginner/basics/tensorqs_tutorial.html (accessed Mar. 12, 2025).

Declaring and Initializing PyTorch Tensors

```
import torch
import numpy as np

data = [[2, 4, 6],[5, 10, 15]]
np_data = np.array(data)

pt_data = torch.tensor(data)
pt_data_from_np = torch.from_numpy(np_data)

print(np_data.shape)
print(pt_data.shape)
print(pt_data_from_np.shape)

print(data)
print(pt_data)
print(pt_data_from_np)
```

Out:

```
(2,3)
torch.Size([2,3])
torch.Size([2,3])

[[2, 4, 6], [5, 10, 15]]
tensor([[ 2,  4,  6],
        [ 5, 10, 15]])
```

Declaring and Initializing PyTorch Tensors

```
print(pt_data)

pt_data_float =
torch.tensor(pt_data.numpy(),
     dtype=torch.float)

pt_data_float = pt_data.to
(dtype=torch.float)
```

Out:

```
tensor([[ 2,  4,  6],
        [ 5, 10, 15]])


tensor([[ 2.,  4.,  6.],
        [ 5., 10., 15.]])
```

```
print(torch.ones((2,3)))
print(torch.ones(2,3))

print(torch.rand((2,3)))
print(torch.rand(2,3))

print(torch.zeros((2,3)))
print(torch.zeros(2,3))
```

Out:

```
tensor([[ 1, 1, 1],
        [ 1, 1, 1]])

tensor([[0.7305, 0.2068, 0.3800],
        [0.0991, 0.0703, 0.3698]])

tensor([[0., 0., 0.],
        [0., 0., 0.]])
```

Tensor Operations and Attributes

```
print(pt_data)

print(pt_data.T)



print(pt_data.reshape(3,2)



print(pt_data.flatten())

print(f"size: {pt_data.shape}\n\
dtype: {pt_data.dtype}\n\
device: {pt_data.device}")

print(pt_data[:,1])
print(pt_data[1])
print(pt_data[-1,1:])
```

Out:

```
tensor([[ 2,  4,  6],
        [ 5, 10, 15]])
tensor([[ 2,  5],
        [ 4, 10],
        [ 6, 15]])
tensor([[ 2,  4],
        [ 6,  5],
        [10, 15]])
tensor([ 2,  4,  6,  5, 10, 15])

Size: torch.Size([2, 3])
dtype: torch.int64
device: cpu

tensor([ 4, 10])
tensor([ 5, 10, 15])
tensor([10, 15])
```

Tensor Arithmetic Operations

```
print(pt_data)

pt_sum = pt_data + pt_data
pt_sum2 = pt_data.add(pt_data)
print(pt_sum)

pt_diff = pt_data - pt_data
pt_diff = pt_data.sub(pt_data)
print(pt_diff)

pt_matmul = pt_data @ pt_data.T
pt_matmul2 = pt_data.matmul(pt_data.T)
print(pt_matmul)

pt_mul = pt_data * pt_data
pt_mul2 = pt_data.mul(pt_data)
print(pt_mul)
```

Out:

```
tensor([[ 2,  4,  6],
        [ 5, 10, 15]])

tensor([[ 4,  8, 12],
        [10, 20, 30]])

tensor([[0, 0, 0],
        [0, 0, 0]])

tensor([[ 56, 140],
        [140, 350]])

tensor([[  4,  16,  36],
        [ 25, 100, 225]])
```

Tensor Arithmetic Operations (Broadcasting)

Tensor Shape and Dimensions

- Operations like addition, subtraction, and element-wise operations must be between tensors of the same shape and dimensions.

Matrix Multiply (torch.matmul, @)

- Supports vector-matrix (1D, 2D)
- Supports matrix-vector (2D, 1D)
- Supoorts matrix-matrix (2D, 2D)
- Supports batched matrix-multiplies between vectors of ND, ND] or [(N-1)D, D] or ND, (N-1)D]
  - Prepends or Appends a dimension of 1 to the shape of tensor with (N-1) dimensions.
  - Matrix dimensions (last two dimensions) treated as matrices and leading dimensions (treated at batch dimensions and broadcasted over).

Broadcasting

- Method to make tensor dimensions match by copying along certain dimensions
- Dimensions must match between tensors, or one tensor must have a “1” in the dimension. This tensor will be copied along this dimension to make these match.

NAIRR Pilot National Artificial Intelligence Research Resource Pilot

Tensor Arithmetic Operations (Broadcasting)

Example #1
- Tensor1.shape = (k, m, n, p)
- Tensor2.shape = (k, 1, n, p) → broadcast to (k, m, n, p) by copying along the 2nd leading dimension.
- Result.shape = (k, m, n, p)

Example #2
- Tensor1.shape = (k, m, 1, p) → broadcast to (k, m, n, p) by copying along the 3rd leading dimension.
- Tensor2.shape = (k, m, n, p)
- Result.shape = (k, m, n, p)

Matmul Example #1 (Tensor1 @ Tensor2)
- Tensor1.shape = (k, 1, r, n) → broadcast to (k, m, r, n) by copying along the 2nd leading dimension.
- Tensor2.shape = (k, m, n, p)
- Result.shape = (k, m, r, p) → due to matrix multiple between (r,n) and (n,p).

Matmul Example #2 (Tensor1 @ Tensor2)
- Tensor1.shape = (k, 1, r, n) → broadcast to (k, m, r, n) by copying along the 2nd leading dimension.
- Tensor2.shape = (1, m, n, p) → broadcast to (k, m, n, p) by copying along the 1st leading dimension.
- Result.shape = (k, m, r, p) → due to matrix multiple between (r,n) and (n,p).

PyTorch Tensor Summary [1]

Declaring and Initializing Tensors

- Tensors can be created from python data (lists), numpy arrays, or other tensors.
- Tensors can be initialized with data (as above) or via random (torch.rand()) or constant values (torch.ones() or torch.zeros()).

Tensor attributes and operations

- Tensors have attributes such as .dtype, .shape, or .device
- Tensors created on 'CPU' by default but can be moved via .to(device) operation. (More on this latter)
- Tensors support standard numpy-like indexing and slicing.
- Tensors support many numpy-like operations (.sum, .flatten(), .T, .reshape())

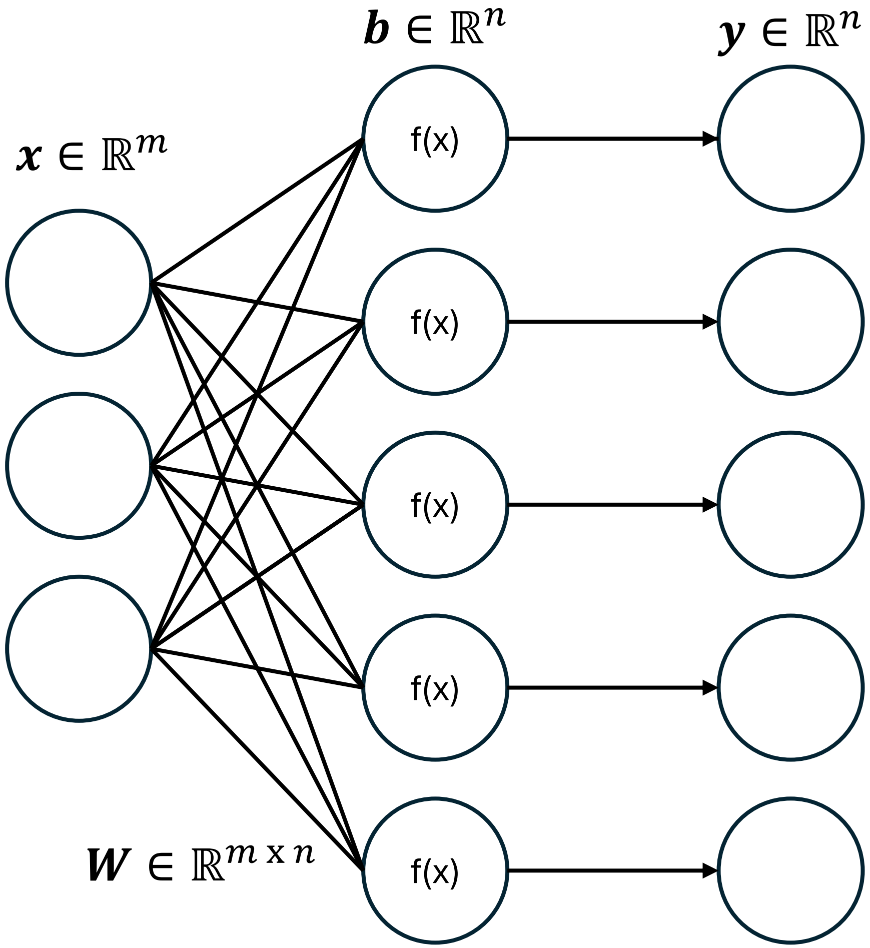
Tensor arithmetic operations

- Tensors support the standard operators (+, -) and elementwise (*, /). The '@' is a matrix multiplication.
- Tensors support methods for operators (.add, .sub), elementwise (.mul, .div), and matrix mult. (.matmul).
- Tensors support numpy like broadcasting rules.

A comprehensive list of tensor operations (methods) is available in the torch documentation:
(https://pytorch.org/docs/stable/torch.html)

[1] The Linux Foundation, "Tensors– PyTorch Tutorials 2.6.0 +cu124 documentation," pytorch.org. https://pytorch.org/tutorials/beginner/basics/tensorqs_tutorial.html (accessed Mar. 12, 2025).

Computational Graphs and Automatic Differentiation

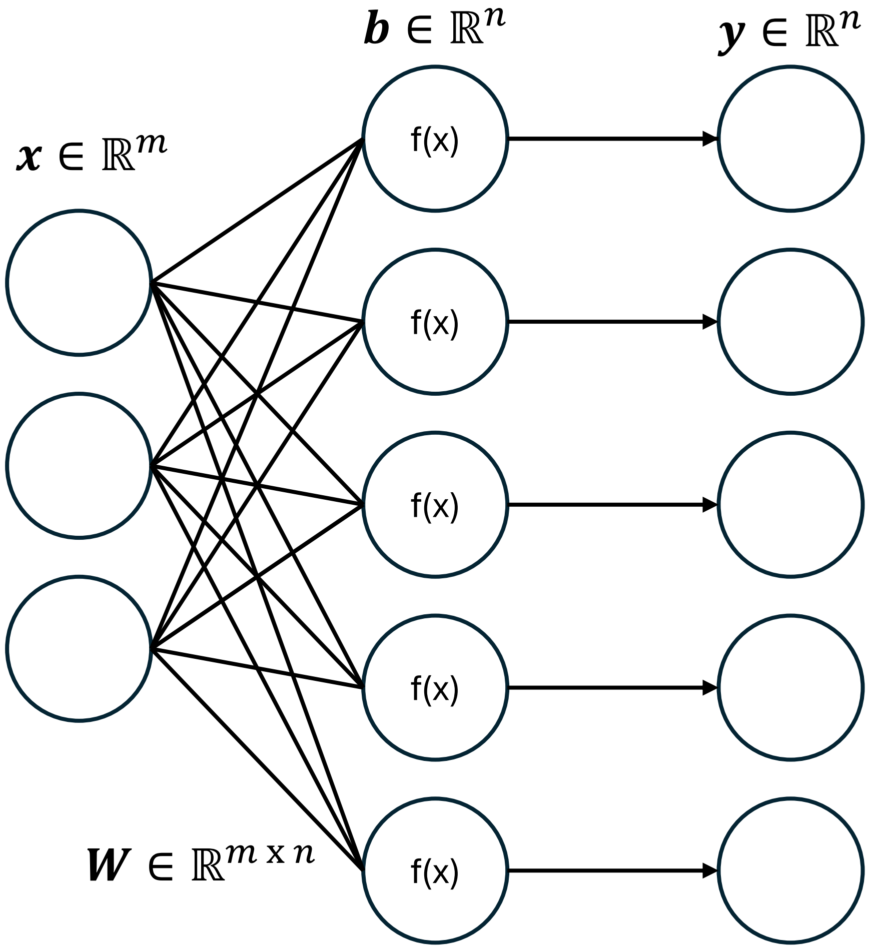


$\text{ReLU}(\boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{b}) = \boldsymbol{y}$ $\quad$ $\text{ReLU}(\sum_i w_{ji}^T x_i + b_j) = y_j$

$$\text{ReLU}(x) = \max(0, x)$$

```
m=3
n=5
w_size=(m,n)
x = torch.ones(m)
W = torch.rand(w_size, requires_grad=True)
b = torch.rand(n, requires_grad=True)

fx = W.T.matmul(x) + b
y = torch.relu(fx)

y_0 = torch.rand(n)
sq_err = (y_0 - y)**2
loss = sq_err.sum()
```

Computational Graphs and Automatic Differentiation

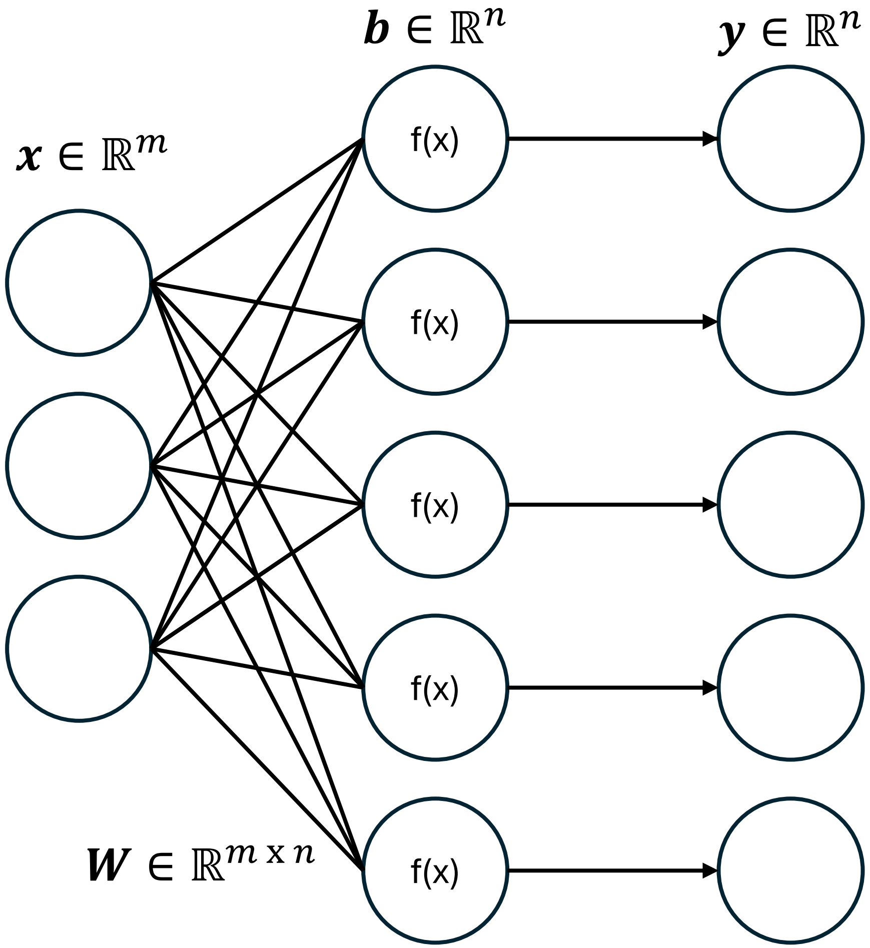


$\text{ReLU}\left(\boldsymbol{W}^{\boldsymbol{T}}\boldsymbol{x} + \boldsymbol{b}\right) = \boldsymbol{y}$ $\quad$ $\text{ReLU}\left(\sum_i w_{ji}^T x_i + b_j\right) = y_j$

$$\text{ReLU}(x) = \max(0, x)$$

```
m=3
n=5

linear = torch.nn.Linear(m,n)
relu = torch.nn.ReLU()

y = relu(linear(x))
y_0 = torch.rand(n)

mse_loss = torch.nn.MSELoss(reduction=‘sum’)

loss = mse_loss(y, y_0)
```

Computational Graphs and Automatic Differentiation



$\text{ReLU}(\boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{b}) = \boldsymbol{y}$ loss= $\sum_i (y_j^0 - y_j)^2$

```
loss = mse_loss(y, y_0)

loss.backward()

W.grad

b.grad

print(loss.item())
```

$\frac{\partial loss}{\partial \boldsymbol{W}}$: W.grad

$\frac{\partial loss}{\partial \boldsymbol{b}}$: b.grad

Computational Graphs and Automatic Differentiation Summary [1]

Computational Graph

- Tensors can be used to define inputs, parameters (requires_grad=True), and operations.
- Tensor operations are stored in a computational graph that allows for automatic differentiation of parameters.

Automatic Differentiation

- The computational graph is created and the result calculated on the forward pass.
- A loss function is defined (must result in a scalar) the represents the function to be minimized.
- Calling the .backward() method on this loss tensor calls the backward pass that calculates the gradients of parameters.
- These gradients can be accessed via the .grad attribute on the parameter tensors.
- The single value tensor (loss) can be extracted by the .item() method on the tensor.

Model Layers

- Pytorch provides a library of model layers via the torch.nn namespace, a full list of available layers are listed in the documentation: (https://pytorch.org/docs/stable/nn.html)

[1] The Linux Foundation, "Automatic Differentiation with torch.autograd – PyTorch Tutorials 2.6.0 +cu124 documentation," pytorch.org. https://pytorch.org/tutorials/beginner/basics/autogradqs_tutorial.html (accessed Mar. 13, 2025).

Building PyTorch Models

```
import torch
from torch import nn

class NNModel(nn.Module):
    def __init__(self,n,m):
        super().__init__()
        self.linear = nn.Linear(n,m)
        self.relu = nn.ReLU()

    def forward(self, input_tensor):
        fx = self.linear(input_tensor)
        y = self.relu(fx)
        return y
```

```
input_x = torch.rand(3)
Print(input_x)

my_model = NNModel(3,5)
output_y = my_model(input_x)

print(output_y)
```

Out:

```
tensor([0.9986, 0.7325, 0.7332])

tensor([0.8556, 0.0000, 0.0000,
0.4691, 0.0000],
grad_fn=<ReluBackward0>)
```

Building PyTorch Models: Vectorization

```
input_x = torch.rand(25,3)
Print(input_x)

my_model = NNModel(3,5)
output_y = my_model(input_x)

print(output_y)
```

Out:

```
tensor([0.9237, 0.9082, 0.9080],
       [ …     , …      , …      ],
       [0.7065, 0.8609, 0.4841]])

tensor([[0.8348, 0.0000, 0.0000, 0.5083, 0.0000],
        […     , …      , …      , …      , …      ],
        [0.6665, 0.0000, 0.0000, 0.3808, 0.0399]],
       grad_fn=<ReluBackward0>)
```

Building PyTorch Models: Structure and Parameters

```
print(my_model)

for name, param in \
my_model.named_parameters():
    print(f"{name}: {param}")
```

Out:

```
NNModel(
  (linear):
Linear(in_features=3,out_features=5, bias=True)
  (relu): ReLU()
)
```

Out:

```
linear.weight: Parameter containing:
tensor([[ 0.3422, -0.2172,  0.2459],
        [-0.5573, -0.4091,  0.5427],
        [-0.5000,  0.4573,  0.3234],
        [ 0.5427,  0.4859, -0.0315],
        [-0.2629,  0.5335, -0.3348]], requires_grad=True)
linear.bias: Parameter containing:
tensor([ 0.4927,  0.0171, -0.4328, -0.4057, -0.0716], requires_grad=True)
```

Building PyTorch Models Summary [1]

Model Structure

- Models are python classes that inherit from torch.nn.module.
- These classes instantiate the various layers of the network in the "__init__" method.
- These classes implement the forward pass of the network in the "forward" method and return the result tensor.

Model Use

- Models are instantiated via the "__init__" method to set the structure of the model.
- Parameters can be used in the "__init__" method to set various hyperparameters for the model.
- The instantiated model object can be called directly to perform the forward pass on the network by giving the input tensor as the call's argument.
- A call on the model object returns the result tensor.
- The forward pass can be vectorized (multiple simultaneous inputs) by stacking inputs in additional leading dimensions of the input tensor.

Model Information

- Printing the model object returns the structure of the model.
- The parameters of the model can be accessed via the .named_parameters() function of the model object.

[1] The Linux Foundation, "Build the Neural Network – PyTorch Tutorials 2.6.0 +cu124 documentation," pytorch.org. https://pytorch.org/tutorials/beginner/basics/buildmodel_tutorial.html (accessed Mar. 13, 2025).

PyTorch Datasets and DataLoaders

```
import torch
from torch.utils.data import Dataset

class RandDataset(Dataset):
    def __init__(self, input_dims, output_dims, length, transform=None, target_transform=None):
        self.input_dims = input_dims
        self.output_dims = output_dims
        self.transform = transform
        self.target_transform = target_transform
        self.mapping = torch.rand(output_dims, input_dims)
        self.len = length

    def __len__(self):
        return self.len

    def __getitem__(self, idx):
        input_tensor = torch.rand(self.input_dims)
        if self.transform:
            input_tensor = self.transform(input_tensor)

        output_tensor = self.mapping.matmul(input_tensor)
        if self.target_transform:
            output_tensor = self.target_transform(output_tensor)

        return input_tensor, output_tensor
```

PyTorch Datasets and DataLoaders

```
from torch.utils.data import DataLoader

rd = RandDataset(input_dims=3, output_dims=5,
length=64)

for idx, rd_output in enumerate(rd):
    if idx < 3:
        input_tensor, output_tensor = rd_output
        print(f"{input_tensor} -> {output_tensor}")
    else:
        break
```

Out:

```
tensor([0.1164, 0.4654, 0.5546]) ->
tensor([0.0940, 0.8165, 0.5114,
0.6640, 0.3020])

tensor([0.8446, 0.3525, 0.5987]) ->
tensor([0.1566, 1.0676, 0.9402,
1.0815, 0.3526])

tensor([0.6241, 0.1571, 0.3631]) ->
tensor([0.1001, 0.6570, 0.6103,
0.7018, 0.2117])
```

PyTorch Datasets and DataLoaders

```
from torch.utils.data import DataLoader

rd = RandDataset(input_dims=3, output_dims=5, length=64)

rd_dataloader = DataLoader(rd, batch_size=32, shuffle=True)
for input_tensor, output_tensor in rd_dataloader:
    print(f"{input_tensor} -> {output_tensor}")
```

Out:

```
tensor([[0.1164, 0.4654, 0.5546],
        [ …     , …      , …      ],
        [0.6529, 0.7272, 0.7023]]) -> tensor([[0.6841, 1.0017, 1.7319, 1.2476, 1.2713],
                                              [ …     , …      , …      , …      , …      ],
                                              [0.5391, 0.3703, 0.6725, 0.5033, 0.3757]])

tensor([[0.8643, 0.1568, 0.8459],
        [ …     , …      , …      ],
        [0.2081, 0.1563, 0.9666]]) -> tensor([[0.4864, 0.8983, 1.5787, 1.1396, 1.2151],
                                              [ …     , …      , …      , …      , …      ],
                                              [0.5943, 0.6316, 0.8847, 0.5879, 0.5603]])
```

PyTorch Datasets and DataLoaders Summary [1]

Datasets

- Datasets are python classes that inherit from torch.utils.data.Dataset.
- Need to implement the __getitem__(self, idx) method which takes an integer index to return the input and output tensors of the dataset.
- The __len__ method is implemented to return the number of data points in the dataset. Used by the DataLoader to stop iterations at an epoch (once through all the data).
- Can be constructed with __init__ method options to pass a transform or target_transform key word option that will transform the input and output tensors. The options take a callable function that will be applied to the tensors.
- Datsets only return one input, output tuple at a time.

DataLoaders

- DataLoaders take a dataset and allow you to select options such as batch_size and shuffle.
- DataLoaders will return a batch of inputs and outputs, size defined by batch_size option.
- DataLoaders if iterated on will proceed through one epoch.

[1] The Linux Foundation, “Datasets & DataLoaders– PyTorch Tutorials 2.6.0 +cu124 documentation,” pytorch.org. https://pytorch.org/tutorials/beginner/basics/data_tutorial.html (accessed Mar. 20, 2025).

Overview – Big Picture Up To Now

DataLoader:

We have implemented a RandDataset that returns input vectors (x^0) of some length (m) to output vectors (y^0) of some length (n). Internally, this is done by setting a random (mxn) matrix (A) that performs the mapping.

$$\boldsymbol{A^T x^0 = y^0}$$

Model:

We have implemented a PyTorch deep learning model with at least one linear layer and one ReLU function that takes input vectors (x) of some length (m) and returns output vectors (y) of some length (n). Parameters in the linear layers determine how the output vector (y).

$$\boldsymbol{x \xrightarrow{Model} y}$$

Training:

We want to train the model (optimize the parameters) to ensure that given some input vector $x = x^0$ that the model returns $y = y^0$

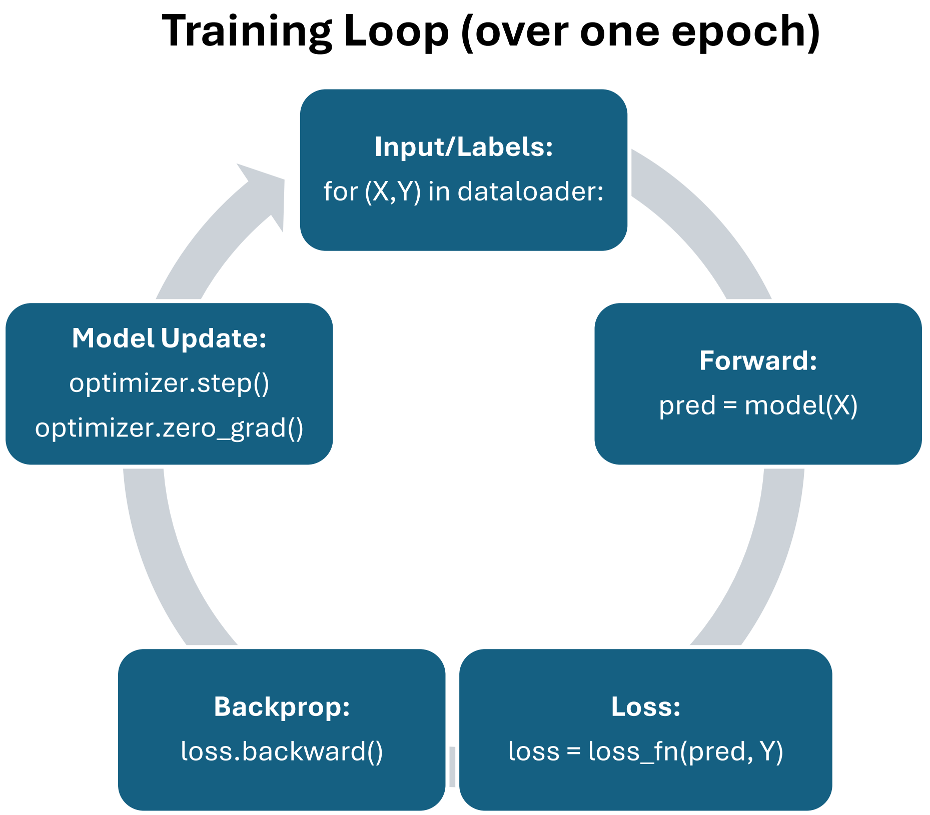
Training PyTorch Models: DataLoaders, Train and Test sets

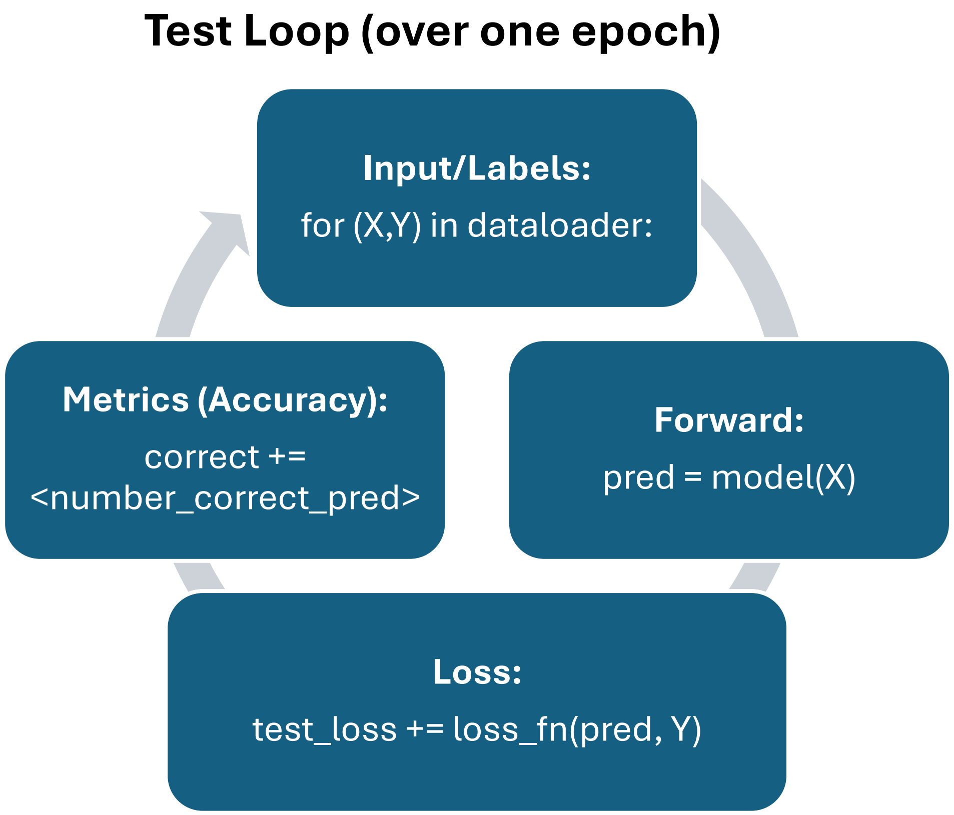
```
import copy
from torch.utils.data import DataLoader

batch_size = 32

train_dataset = RandDataset(3,5,32000)
test_dataset = copy.deepcopy(train_dataset)
test_dataset.length = 1000

train_dataloader = DataLoader(train_dataset, batch_size=batch_size)
test_dataloader = DataLoader(test_dataset, batch_size=batch_size)
```

Training PyTorch Models: Models, Loss functions, and Optimizers

```
from torch import nn, optim

learning_rate = 1e-2

my_model = NNModel(3,5,20)

loss_fn = nn.MSELoss(reduction='sum’)
optimizer = optim.SGD(my_model.parameters(), lr=learning_rate)
```

Training PyTorch Models: Graphical Overview





Training PyTorch Models: Training Loop

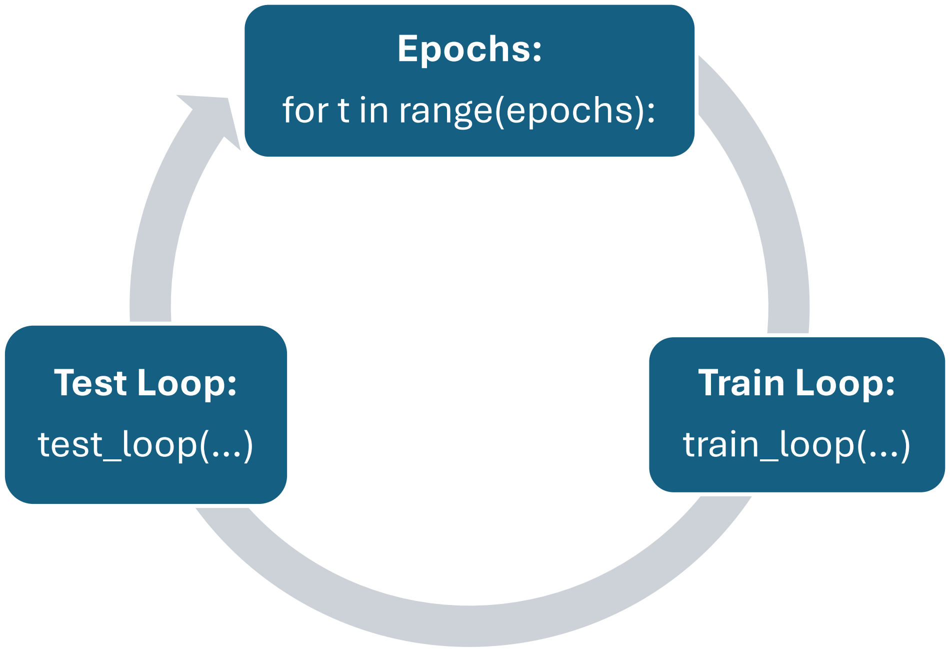
```
def train_loop(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    model.train()
    for batch, (X,Y) in enumerate(dataloader):
        pred = model(X)
        loss = loss_fn(pred, Y)
        avg_loss = loss / len(pred)

        avg_loss.backward()
        optimizer.step()
        optimizer.zero_grad()

        if (batch+1) %100 == 0:
            avg_loss, current = avg_loss.item(), batch * batch_size + len(pred)
            print(f"Avg. loss: {avg_loss:>7f}, [current:{current:>5d}/{size:>5d}]")
```

Training PyTorch Models: Test Loop

```
def test_loop(dataloader, model, loss_fn, tolerance):
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    model.eval()
    test_loss, correct = 0, 0

    with torch.no_grad():
        for (X,Y) in dataloader:
            pred = model(X)
            test_loss += (loss_fn(pred, Y) / len(pred)).item()
            correct += ((pred - Y).abs() <
tolerance).all(dim=1).type(torch.float).sum().item()

    test_loss /= num_batches
    correct /= size

    print(f"Test Error: \n Accuracy: {(100*correct):>0.1f}%, Avg. loss: {test_loss:>8f}\n")
```

Training PyTorch Models: Graphical Overview – Epoch Loop

Epoch Loop




```
def epoch_loop(epochs, train_dataloader,
test_dataloader, model, loss_fn,
optimizer, tolerance):

    for t in range(epochs):
        print(f"Epoch {t+1}\n-----------
-------------------")

        train_loop(train_dataloader,
model, loss_fn, optimizer)

        test_loop(test_dataloader,
model, loss_fn, tolerance)

    print("Done")
```

Training PyTorch Models: Executing Training

```
batch_size = 32
learning_rate = 1e-2
epochs = 50
tolerance = 1e-2

train_dataset = RandDataset(3,5,32000)
test_dataset = copy.deepcopy(train_dataset)
test_dataset.length = 1000

train_dataloader = DataLoader(train_dataset,batch_size=batch_size)
test_dataloader = DataLoader(test_dataset,batch_size=batch_size)

my_model = NNModel(3,5,20)

loss_fn = nn.MSELoss(reduction='sum')
optimizer = optim.SGD(my_model.parameters(), lr=learning_rate)

epoch_loop(epochs, train_dataloader, test_dataloader, my_model, loss_fn, optimizer, tolerance)
```

Training PyTorch Models: Executing Training

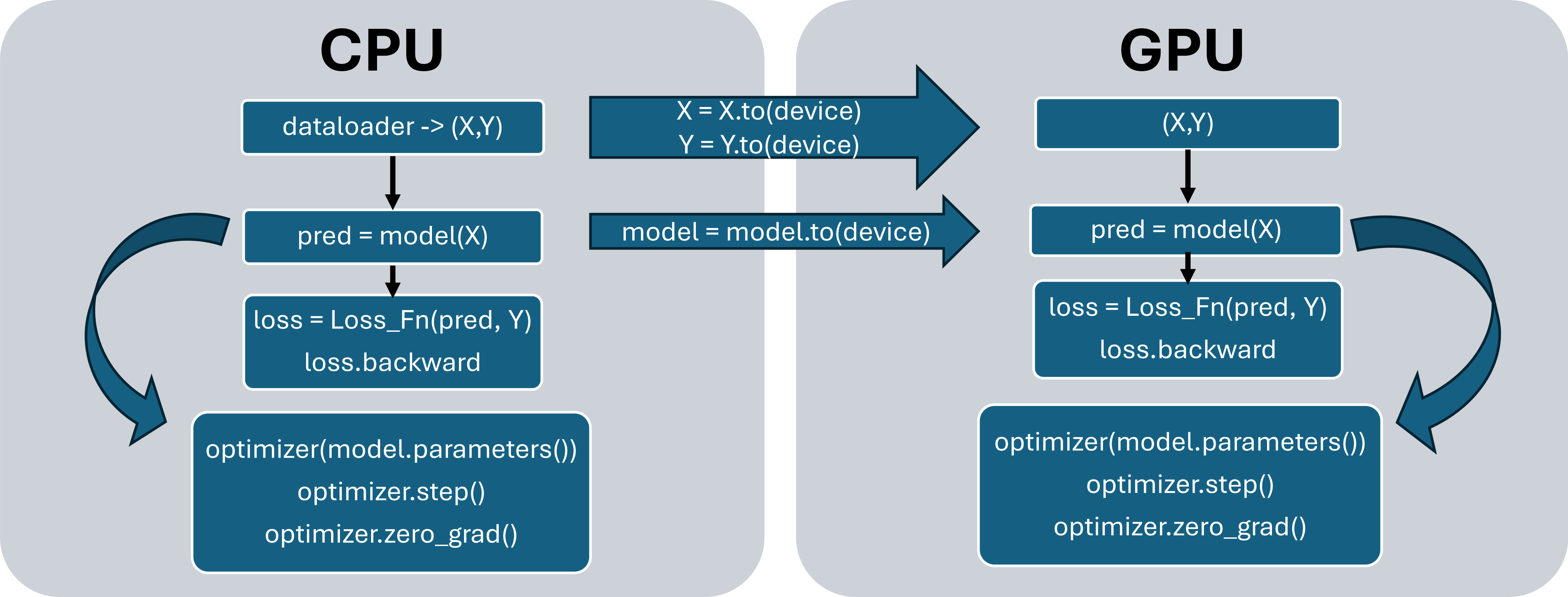
Out:

```
Epoch 1
-------------------------------
Avg. loss: 0.173822, [current: 3200/32000]
Avg. loss: 0.109654, [current: 6400/32000]
Avg. loss: 0.072340, [current: 9600/32000]
Avg. loss: 0.043262, [current:12800/32000]
Avg. loss: 0.032554, [current:16000/32000]
Avg. loss: 0.026890, [current:19200/32000]
Avg. loss: 0.022141, [current:22400/32000]
Avg. loss: 0.020239, [current:25600/32000]
Avg. loss: 0.018051, [current:28800/32000]
Avg. loss: 0.014956, [current:32000/32000]
Test Error:
 Accuracy: 0.1%, Avg. loss: 0.018523
```

Out:

```
Epoch 50
-------------------------------
Avg. loss: 0.000005, [current: 3200/32000]
Avg. loss: 0.000010, [current: 6400/32000]
Avg. loss: 0.000057, [current: 9600/32000]
Avg. loss: 0.000004, [current:12800/32000]
Avg. loss: 0.000012, [current:16000/32000]
Avg. loss: 0.000023, [current:19200/32000]
Avg. loss: 0.000017, [current:22400/32000]
Avg. loss: 0.000007, [current:25600/32000]
Avg. loss: 0.000015, [current:28800/32000]
Avg. loss: 0.000022, [current:32000/32000]
Test Error:
 Accuracy: 99.2%, Avg. loss: 0.000017
```

Training PyTorch Models: Using the GPU



Training PyTorch Models: Using the GPU

```
def train_loop(dataloader, model, loss_fn, optimizer, device=None):
    …
    for batch, (X,Y) in enumerate(dataloader):
        if device:
            X = X.to(device)
            Y = Y.to(device)
        …

def test_loop(dataloader, model, loss_fn, tolerance, device=None):
    …
    with torch.no_grad():
        for (X,Y) in dataloader:
            if device:
                X = X.to(device)
                Y = Y.to(device)
            …
```

Training PyTorch Models: Using the GPU

```
device = torch.device(‘cpu’)

if torch.cuda.is_available():
    device = torch.device(torch.cuda.current_device())

print(f”Using device – {device}”)


my_model = NNModel(3,5,20)
my_model = my_model.to(device)

optimizer = optim.SGD(my_model.parameters(), lr=learning_rate)

epoch_loop(epochs, train_dataloader, test_dataloader, my_model, loss_fn,
optimizer, tolerance, device)
```

Training PyTorch Models Summary [1]

Components

- DataLoaders provide the training and test data (input and expected result) for training and validation.
- Models provide predictions.
- Loss is calculated from a loss function which take the predictions and expected results as input.
    - Additional loss functions are documented
- An optimizer is connected to a model via its model.parameters(). This allows it to get the gradients from parameters and update them.

Optimizers

- PyTorch provides a library of optimizers in the torch.optim namespace, a full list of available loss functions are listed in the documentation: (https://pytorch.org/docs/stable/optim.html).

Loss Functions

- PyTorch provides a library of loss functions in the torch.nn namespace, a full list of available loss functions are listed in the documentation: (https://pytorch.org/docs/stable/nn.html#loss-functions).

[1] The Linux Foundation, “Optimizing Model Parameters – PyTorch Tutorials 2.6.0 +cu124 documentation,” pytorch.org. https://pytorch.org/tutorials/beginner/basics/optimization_tutorial.html (accessed Mar. 24, 2025).

Training PyTorch Models Summary [1]

Training and Test Loops

- Performs loops over each input batch via DataLoader, getting the input and expected output.
- Use model to perform prediction.
- Use loss function to calculate the loss based on prediction and expected output.
- Calculate parameter gradients from loss via loss.backwards()
- Update model parameters and reset model gradients via the optimizer.

Using the GPU

- The batch input and expected output tensors can be moved to the GPU.
- The model can also be moved to the GPU.
- Resulting predictions and loss will be on the GPU as a result of the moves above.
- Optimizer operates on the model's parameters and gradients, which are already on the GPU.

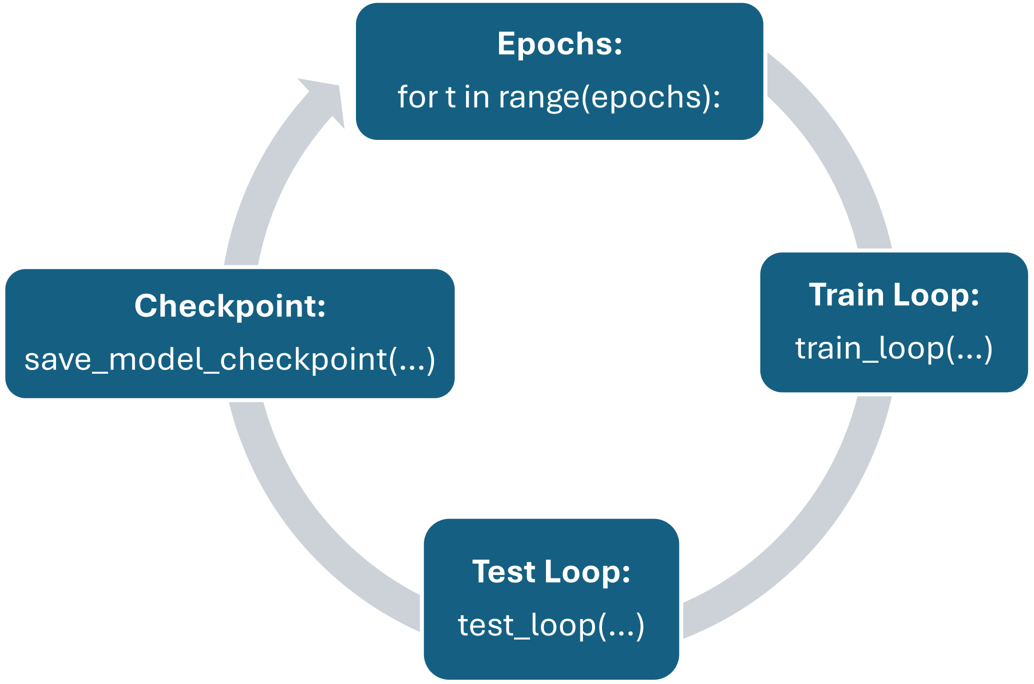
[1] The Linux Foundation, "Optimizing Model Parameters – PyTorch Tutorials 2.6.0 +cu124 documentation," pytorch.org. https://pytorch.org/tutorials/beginner/basics/optimization_tutorial.html (accessed Mar. 24, 2025).

Loading and Saving PyTorch Models: Graphical Overview

Restart

Restart:
restore_model_checkpoint(...)

Epoch Loop



Saving and Loading PyTorch Models: Epoch Loop

```
def epoch_loop(epochs, train_dataloader, test_dataloader, model, loss_fn,
optimizer, tolerance, device=None, file_path=None):

    if file_path:
        epoch_last = restore_model_checkpoint(model, optimizer, train_dataloader,
test_dataloader, file_path)

    for t in range(epoch_last+1,epochs):
        print(f"Epoch {t+1}\n-------------------------------")
        train_loop(train_dataloader, model, loss_fn, optimizer, device)
        test_loop(test_dataloader, model, loss_fn, tolerance, device)

        if file_path:
            save_model_checkpoint(model, optimizer, train_dataloader, t,
file_path)
    print("Done")
```

Saving and Loading PyTorch Models: Save Checkpoint

```
def save_model_checkpoint(model, optimizer, dataloader, epoch, file_path):

    save_dict = dict(
        model_state_dict = model.state_dict(),
        optimizer_state_dict = optimizer.state_dict(),
        epoch = epoch,
        dataloader_mapping = dataloader.dataset.mapping,
    )

    torch.save(save_dict,file_path)
```

Saving and Loading PyTorch Models: Restore from Checkpoint

```
def restore_model_checkpoint(model, optimizer, train_dataloader, test_dataloader,
file_path):
    epoch = -1

    if file_path.exists():
        print(f"Restarting from checkpoint: {str(file_path)}")

        checkpoint = torch.load(file_path, weights_only=True)

        model.load_state_dict(checkpoint['model_state_dict'])
        optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
        train_dataloader.dataset.mapping = checkpoint['dataloader_mapping']
        test_dataloader.dataset.mapping = checkpoint['dataloader_mapping’]
        epoch = checkpoint['epoch’]

    return epoch
```

Training PyTorch Models: Executing Training with Checkpointing

```
from pathlib import Path

batch_size = 32
learning_rate = 1e-2
epochs = 50
tolerance = 1e-2
checkpoint_file = Path() / 'model_checkpoint.pth'

train_dataset = RandDataset(3,5,32000)
test_dataset = copy.deepcopy(train_dataset)
test_dataset.length = 1000

train_dataloader = DataLoader(train_dataset,batch_size=batch_size)
test_dataloader = DataLoader(test_dataset,batch_size=batch_size)

my_model = NNModel(3,5,20)

loss_fn = nn.MSELoss(reduction='sum')
optimizer = optim.SGD(my_model.parameters(), lr=learning_rate)

epoch_loop(epochs, train_dataloader, test_dataloader, my_model, loss_fn, optimizer, tolerance, checkpoint_file)
```

Training PyTorch Models: Executing Training with Checkpointing

Out:

```
Epoch 1
-------------------------------
Avg. loss: 0.173822, [current: 3200/32000]
Avg. loss: 0.109654, [current: 6400/32000]
Avg. loss: 0.072340, [current: 9600/32000]
Avg. loss: 0.043262, [current:12800/32000]
Avg. loss: 0.032554, [current:16000/32000]
Avg. loss: 0.026890, [current:19200/32000]
Avg. loss: 0.022141, [current:22400/32000]
Avg. loss: 0.020239, [current:25600/32000]
Avg. loss: 0.018051, [current:28800/32000]
Avg. loss: 0.014956, [current:32000/32000]
Test Error:
 Accuracy: 0.1%, Avg. loss: 0.018523
```

Out:

```
Epoch 6
-------------------------------
Avg. loss: 0.000998, [current: 3200/32000]
Avg. loss: 0.001015, [current: 6400/32000]
Avg. loss: 0.000801, [current: 9600/32000]
Avg. loss: 0.000676, [current:12800/32000]
Avg. loss: 0.000612, [current:16000/32000]
Avg. loss: 0.000665, [current:19200/32000]
Avg. loss: 0.000377, [current:22400/32000]
Avg. loss: 0.000530, [current:25600/32000]
Avg. loss: 0.000601, [current:28800/32000]
Avg. loss: 0.000609, [current:32000/32000]
Test Error:
 Accuracy: 36.9%, Avg. loss: 0.000669
```

Training PyTorch Models: Executing Training with Checkpointing

Out:

```
Restarting from checkpoint: model_checkpoint.pth
Epoch 7
------------------------------
Avg. loss: 0.000723, [current: 3200/32000]
Avg. loss: 0.000534, [current: 6400/32000]
Avg. loss: 0.000824, [current: 9600/32000]
Avg. loss: 0.000949, [current:12800/32000]
Avg. loss: 0.000497, [current:16000/32000]
Avg. loss: 0.000587, [current:19200/32000]
Avg. loss: 0.000697, [current:22400/32000]
Avg. loss: 0.000450, [current:25600/32000]
Avg. loss: 0.000379, [current:28800/32000]
Avg. loss: 0.000571, [current:32000/32000]
Test Error:
 Accuracy: 46.2%, Avg. loss: 0.000577
```

→

Out:

```
Epoch 12
------------------------------
Avg. loss: 0.000704, [current: 3200/32000]
Avg. loss: 0.000259, [current: 6400/32000]
Avg. loss: 0.000199, [current: 9600/32000]
Avg. loss: 0.000274, [current:12800/32000]
Avg. loss: 0.000640, [current:16000/32000]
Avg. loss: 0.000586, [current:19200/32000]
Avg. loss: 0.000350, [current:22400/32000]
Avg. loss: 0.000317, [current:25600/32000]
Avg. loss: 0.000140, [current:28800/32000]
Avg. loss: 0.000320, [current:32000/32000]
Test Error:
 Accuracy: 66.3%, Avg. loss: 0.000353
```

Saving and Loading PyTorch Models [1,2]

Saving Model and Optimizer Parameters

- A model's parameters can be exported via its .state_dict() method.
- An optimizers parameters can be exported via its .state_dict() method.

Saving/Loading a Checkpoint File

- torch.save is used to save checkpoints of models. It takes a dictionary of objects to save and a file path.
- torch.load is used to load checkpoints of models. It takes a file path and parameters such as weights_only.

Saving/Loading other information

- Other information such as last epoch and data needed to restore the state of the training, validation, or model can be saved in the checkpoint file.
- Assign each data value to a different dictionary key in the dictionary used to save the file via torch.save.
- Restore each data value by its key from the dictionary returned from the torch.load call.

[1] The Linux Foundation, "Save and Load the Model – PyTorch Tutorials 2.6.0 +cu124 documentation," pytorch.org. https://pytorch.org/tutorials/beginner/basics/saveloadrun_tutorial.html (accessed Mar. 24, 2025).

[2] The Linux Foundation, "Saving and Loading Models – PyTorch Tutorials 2.6.0 +cu124 documentation," pytorch.org. https://pytorch.org/tutorials/beginner/saving_loading_models.html (accessed Mar. 24, 2025).

Hands On Session

Open OnDemand – JupyterLab Notebooks

- Expanse: https://portal.expanse.sdsc.edu/pun/sys/dashboard/
- Delta: https://openondemand.delta.ncsa.illinois.edu/pun/sys/dashboard/
- DeltaAI: https://gh-ondemand.delta.ncsa.illinois.edu/pun/sys/dashboard/

SSH Command Line Access and File systems

- Expanse: ssh <username>@login.expanse.sdsc.edu
- Delta: ssh <username>@login.delta.ncsa.illinois.edu
- DeltaAI: ssh <username>@dtai-login.delta.ncsa.illinois.edu

Downloading Exercises

- git clone https://github.com/access-ci-org/AI-Unlocked-Workshop-2025.git
    - AI-Unlocked-Workshop-2025/track2-Intermediate-to-Advanced/introduction-to-pytorch/

Hands On Session: Expanse

Open OnDemand – JupyterLab Notebooks

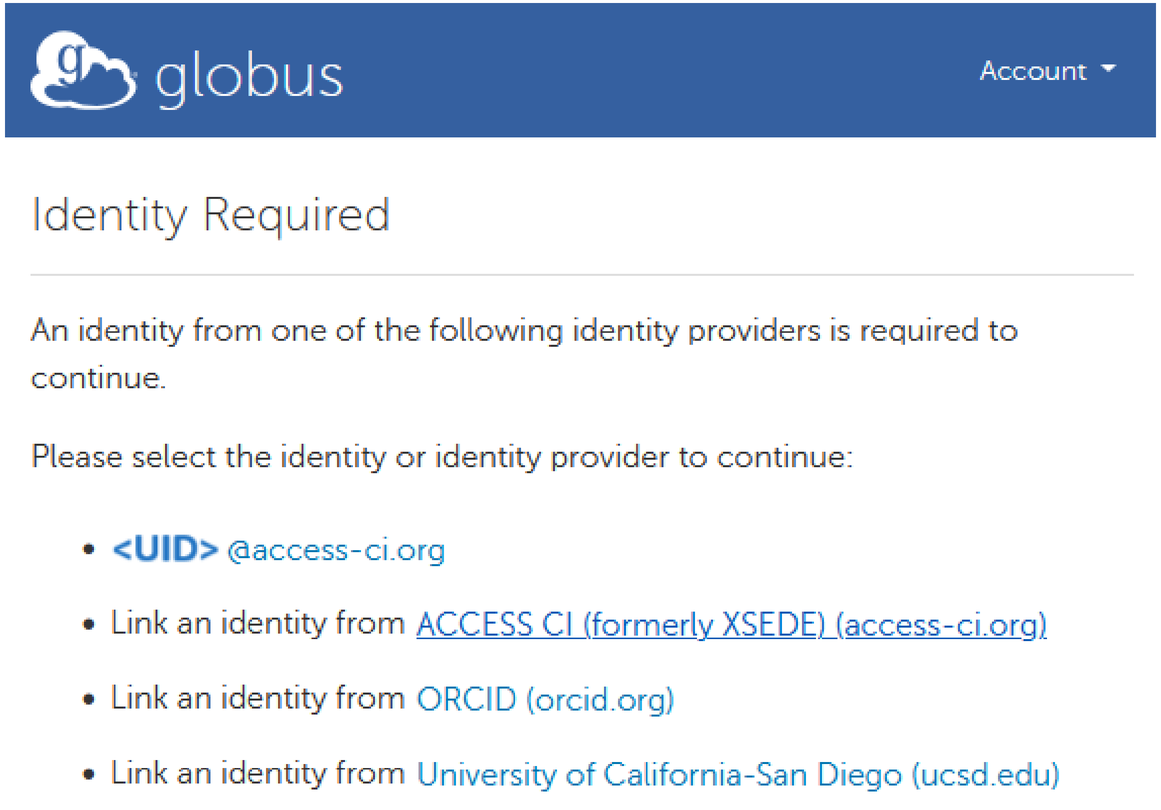
- Expanse: https://portal.expanse.sdsc.edu/pun/sys/dashboard/

SSH Command Line Access and File systems

- Expanse: ssh <username>@login.expanse.sdsc.edu

Downloading Exercises

- git clone https://github.com/access-ci-org/AI-Unlocked-Workshop-2025.git
    - AI-Unlocked-Workshop-2025/track2-Intermediate-to-Advanced/introduction-to-pytorch/

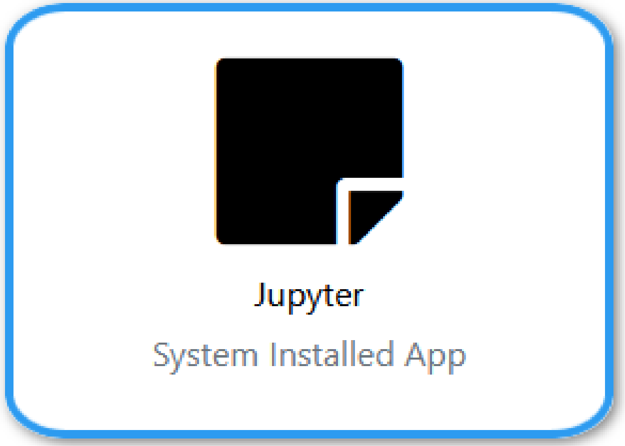


Hands On Session: Expanse

Open OnDemand – JupyterLab Notebooks

- Expanse: https://portal.expanse.sdsc.edu/pun/sys/dashboard/

Look for the Jupyter application (Icon below)



Jupyter Session

Account:

TG-CIS250186

Partition (Please choose the gpu, gpu-shared, or gpu-preempt as the partition if using gpus):

gpu-shared

Time limit (min):

60

Number of cores:

10

Memory required per node (GB):

96

GPUs (optional):

1

Singularity Image File Location: (Use your own or to include from existing container library at /cm/shared/apps/container e.g., /cm/shared/apps/containers/singularity/pytorch/pytorch-latest.sif)

/cm/shared/apps/containers/singularity/pytorch/pytorch-latest.sif

Environment modules to be loaded (E.g., to use latest version of system Anaconda3 include cpu,gcc,anaconda3):

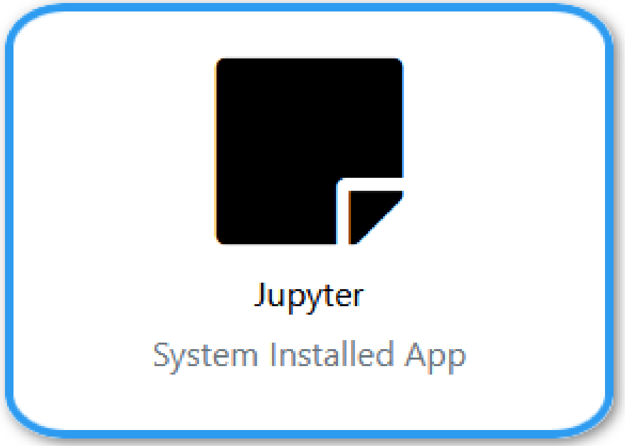
singularitypro

Hands On Session: Expanse

Open OnDemand – JupyterLab Notebooks

- Expanse: https://portal.expanse.sdsc.edu/pun/sys/dashboard/

Look for the Jupyter application (Icon below)



Field	Expanse
Account	TG-CIS250186
Partition	gpu-shared
Time limit (min)	60
Number of cores	10
Memory required per node (GB)	96
GPUs (optional)	1
Singularity Image File Location	/cm/shared/apps/containers/singularity/pytorch/pytorch-latest.sif
Environment modules to be loaded	singularitypro
Working directory	home or lustre
Type	JupyterLab

Hands On Session: Delta and DeltaAI

Open OnDemand – JupyterLab Notebooks

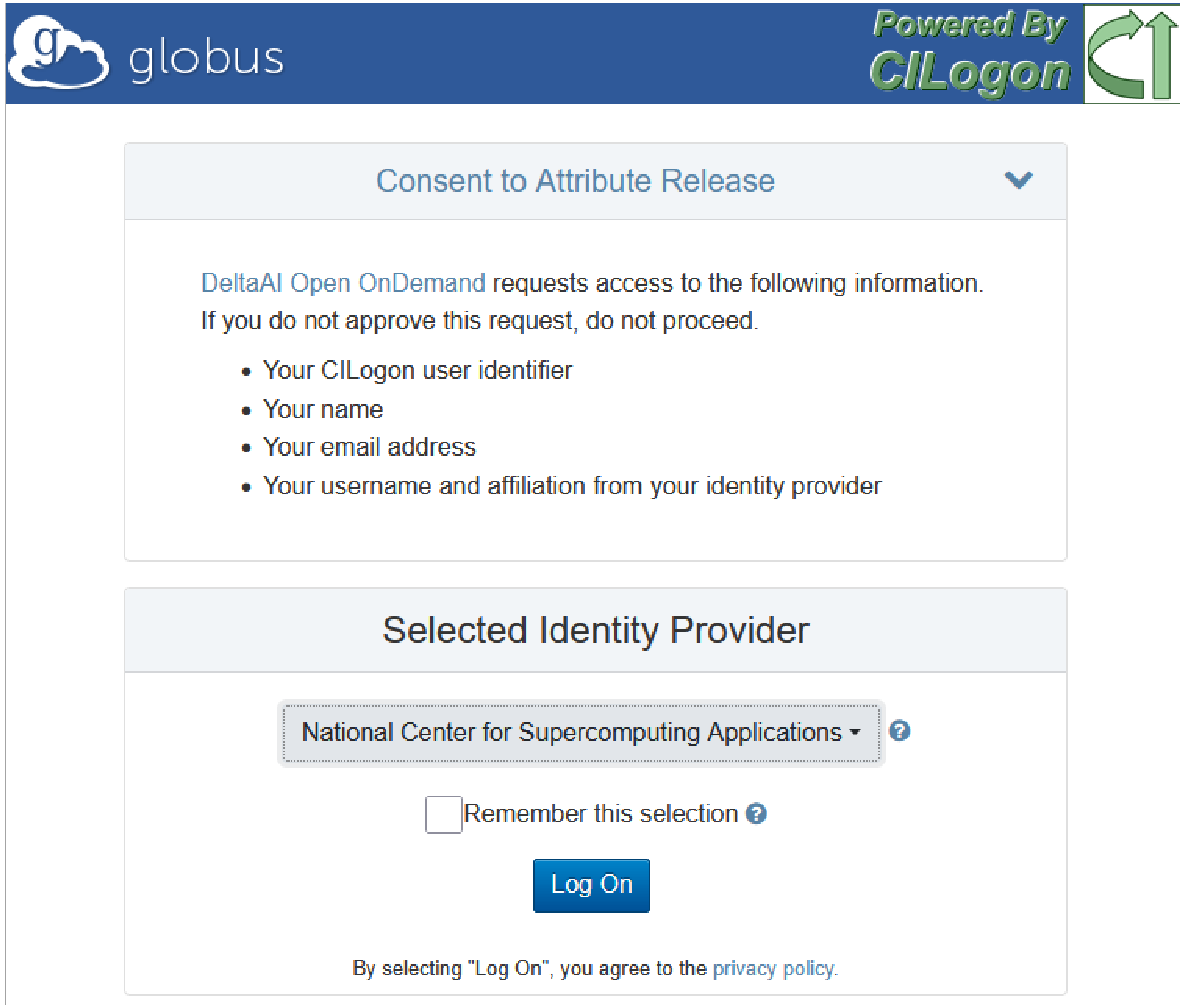
- Delta: https://openondemand.delta.ncsa.illinois.edu/pun/sys/dashboard/
- DeltaAI: https://gh-ondemand.delta.ncsa.illinois.edu/pun/sys/dashboard/

SSH Command Line Access and File systems

- Delta: ssh <username>@login.delta.ncsa.illinois.edu
- DeltaAI: ssh <username>@dtai-login.delta.ncsa.illinois.edu

Downloading Exercises

- git clone https://github.com/access-ci-org/AI-Unlocked-Workshop-2025.git
    - AI-Unlocked-Workshop-2025/track2-Intermediate-to-Advanced/introduction-to-pytorch/

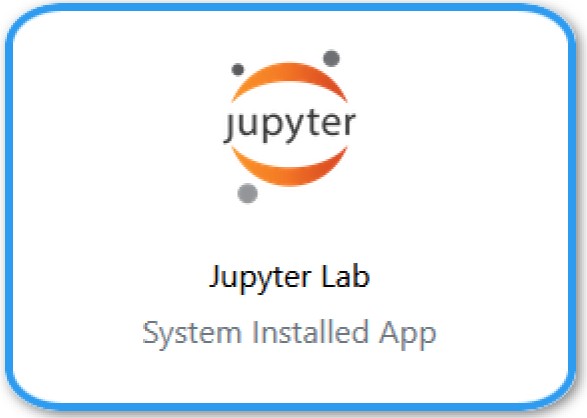


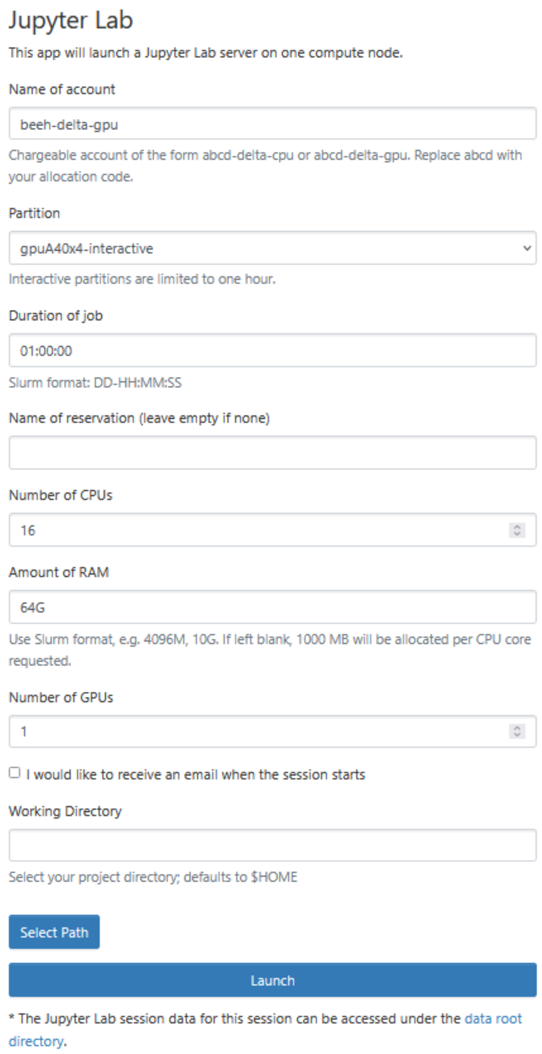
Hands On Session: Delta and DeltaAI

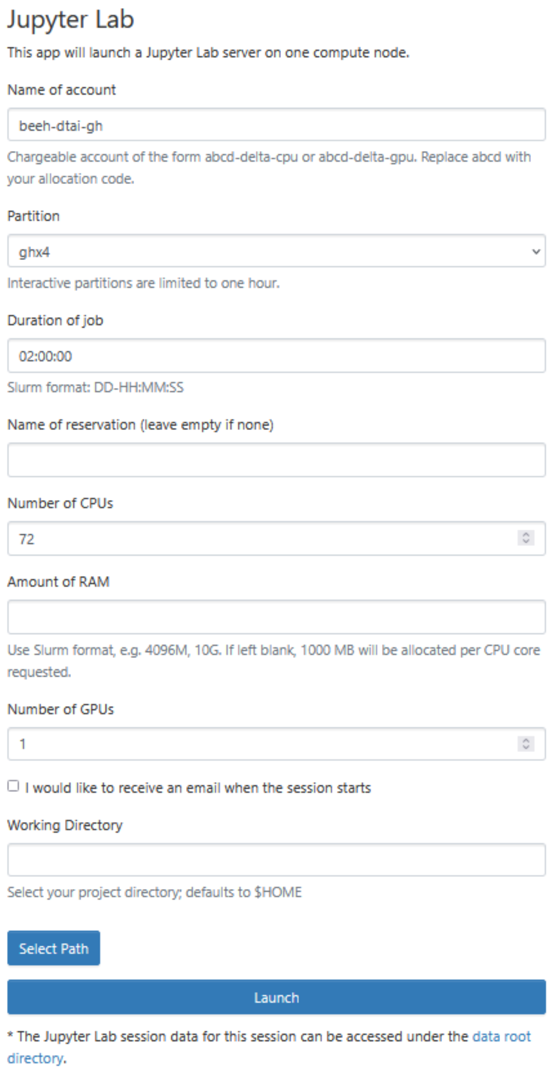
Open OnDemand – JupyterLab Notebooks

- Delta: https://openondemand.delta.ncsa.illinois.edu/pun/sys/dashboard/
- DeltaAI: https://gh-ondemand.delta.ncsa.illinois.edu/pun/sys/dashboard/

Look for the JupyterLab application (Icon below)





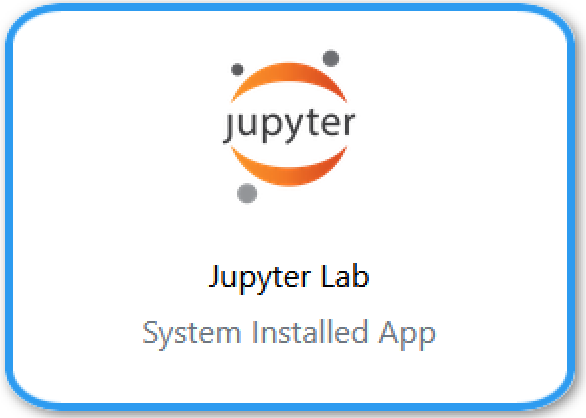


Hands On Session: Delta and DeltaAI

Open OnDemand – JupyterLab Notebooks

- Delta: https://openondemand.delta.ncsa.illinois.edu/pun/sys/dashboard/
- DeltaAI: https://gh-ondemand.delta.ncsa.illinois.edu/pun/sys/dashboard/

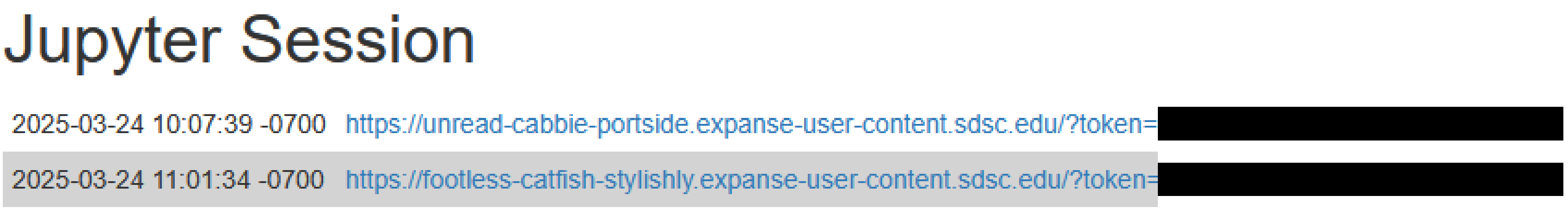
Look for the JupyterLab application (Icon below)



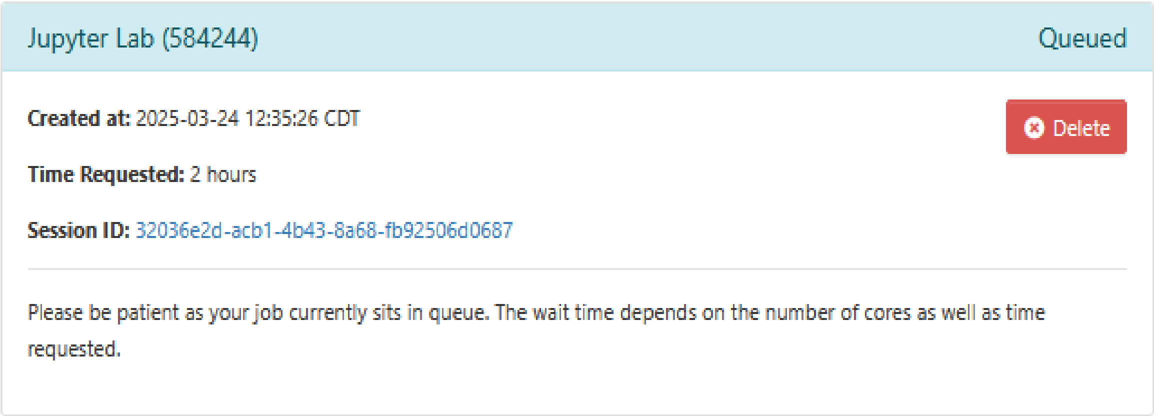
Field	Delta	DeltaAI
Name of account	beeh-delta-gpu	beeh-dtai-gh
Partition	gpuA100x4-interactive or gpuA40x4-interactive	ghx4
Duration of Job	1:00:00	2:00:00
Number of CPUs	16	72
Amount of RAM	64G	< leave blank>
Number of GPUs	1	1

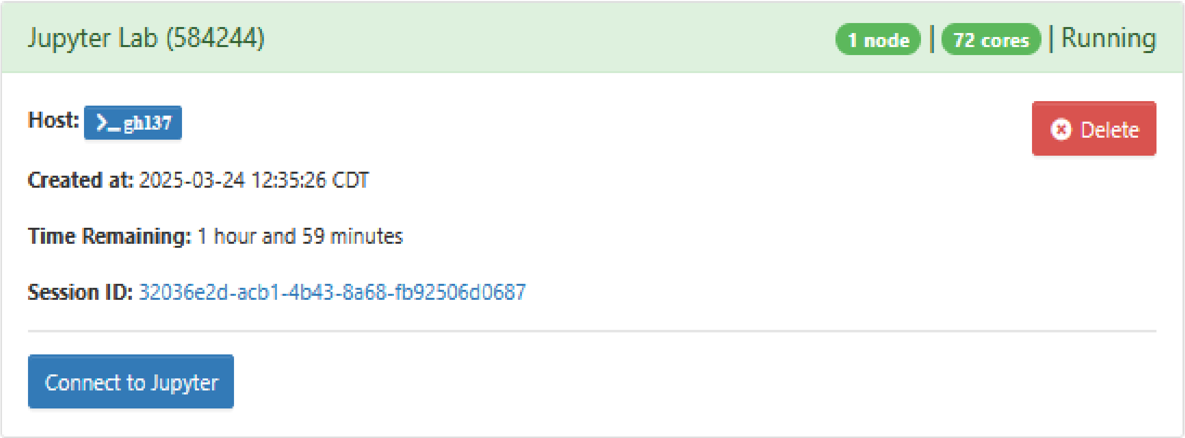
Hands On Session: Launching Jupyter (Expanse, Delta, DeltaAI)

Expanse:

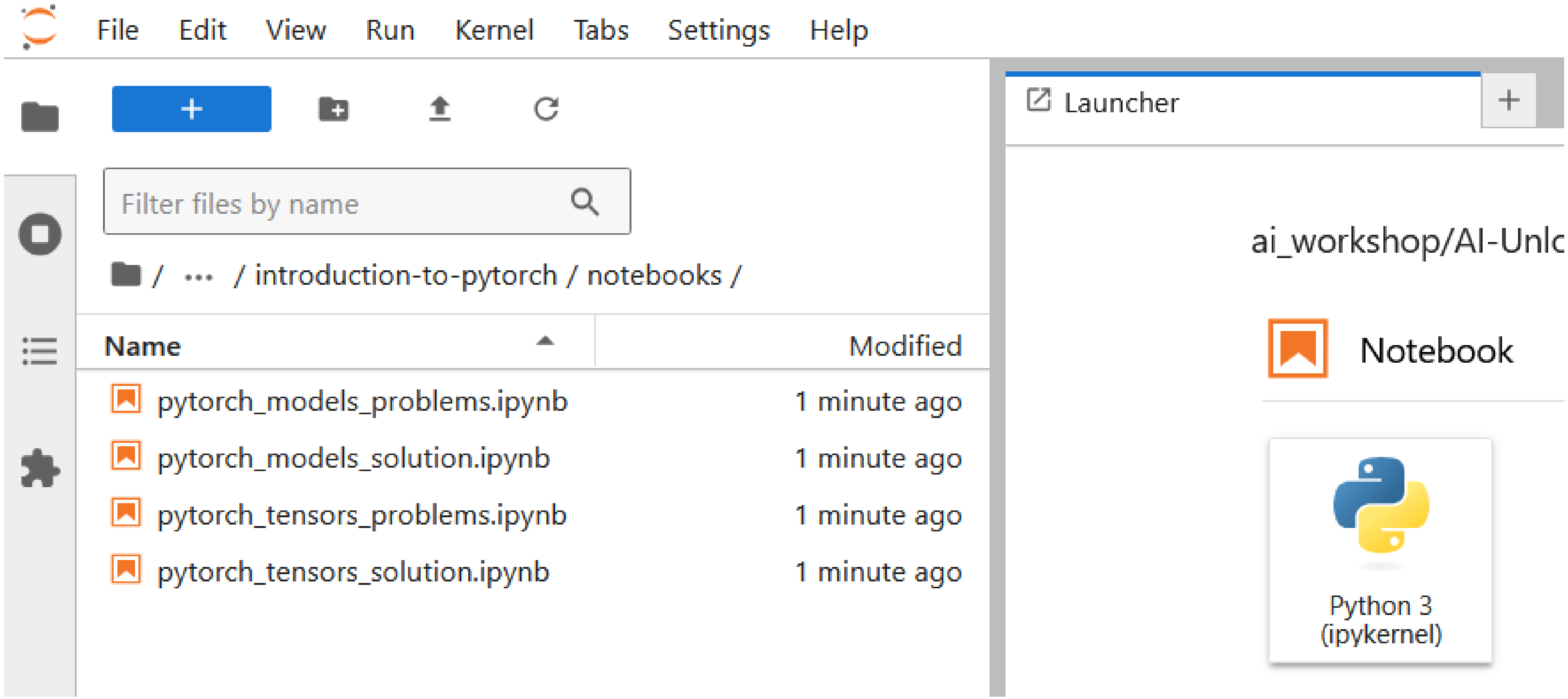


Delta and DeltaAI:





JupyterLab Session



JupyterLab Session

File Edit View Run Kernel Tabs Settings Help

pytorch_tensors_problems.ip × pytorch_models_problems.ip × +

Markdown Notebook Python 3 (ipykernel)

PyTorch Tensor Notebook

This notebook will allow you to get practice in utilizing tensors in PyTorch and explore their proerties and uses. Code excersises denoted by a problem number (i.e. Problem #1) will include a task and a code block that asks for your solution. These blocks will be denoted by comments of the form '# YOUR CODE HERE #'. The code immediately following include assertions that are used to check completeness of the response. They will raise an exception if the previous solution is not complete or not correct.

Declaring, Initializing, and Operating on PyTorch Tensors

Reference: The Linux Foundation, "Tensors-PyTorch Tutorials 2.6.0 +cu124 documentation," pytorch.org https://pytorch.org/tutorials/beginner/basics/tensorqs_tutorial.html (accessed Mar. 12, 2025).

[ ]:
```
import torch
import numpy as np
```

Problem #1: Given the data in python list (data) and numpy array format (numpy_data). Initialize a PyTorch tensor with the data named "pt_tensor", using data, and "pt_tensor_numpy", using numpy_data.

[ ]:
```
data = [[3, 6, 9, 12],[7, 14, 21, 28],[9, 18, 27, 36]]
numpy_data = np.array(data)

# YOUR CODE HERE #

assert pt_tensor.shape == (3,4)
assert pt_tensor_numpy.shape == (3,4)
assert (pt_tensor - pt_tensor_numpy).sum().item() == 0
```