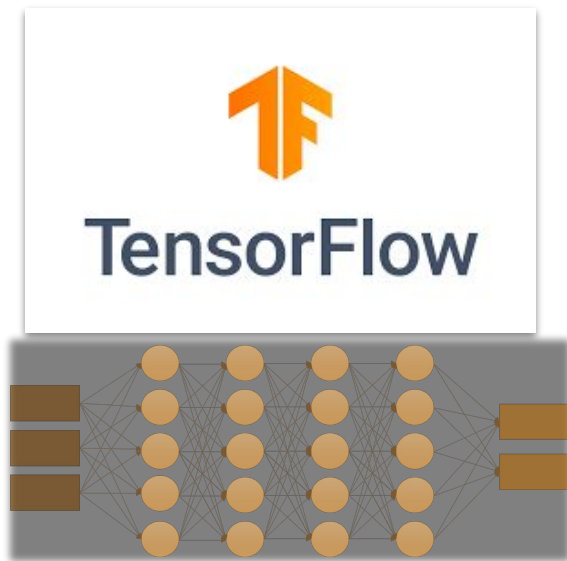
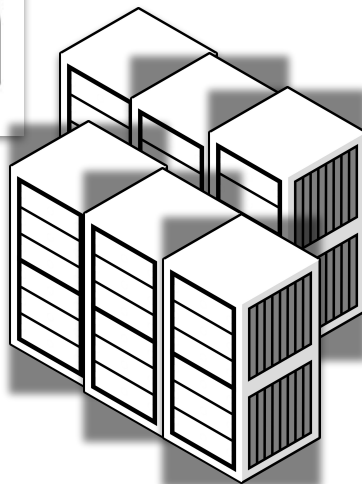


Training ML/DL Models in HPC3

 PyTorch



Ivan Chang

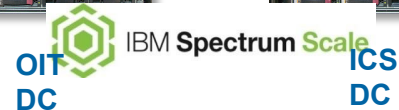
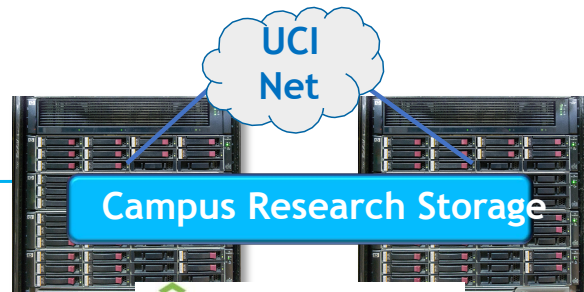
Institute for Precision Health, Genomics Research and Technology Hub, Research Cyberinfrastructure Center
Sep 23rd 2025

Outline

- HPC3 resources guide for ML/DL/AI modeling
- Comparing ML/DL/AI modeling frameworks: TensorFlow and Pytorch
 - Which to use?
- Pytorch DL modeling quickstart:
 - Data structure (tensor)
 - Data loading and Data set
 - Neural network model building
 - Model training and optimization
- Quick tour of HPC3's Biojhub4 (Jupyterhub) for hands on material
- Notes on training Pytorch based genomics DL models:
 - Deepbind (ported from Theano)
 - CellPose-SAM
 - scGPT

HPC3 resources guide for ML/DL/AI modeling

UCI HPC3 Hardware Specifications



HPC3 - Rocky 9.6 Linux Distro

- 11568 Cores/253 Hosts (compute nodes)
- 73,132 GB Aggregated memory
- 14 nodes with 4 Nvidia V100 (16GB) GPUs
- 18 nodes with 4 Nvidia A30 (24GB) GPUs
- 4 nodes with 2 Nvidia A100 (80GB) GPUs
- 2 nodes with 4 Nvidia L40S (48GB) GPUs
- EDR (100Gbps) Infiniband
- 10GbE Ethernet
- Minimum
 - 4GB memory/core
 - AVX2 instruction set (Epyc/Intel CPUs)

Nine Parallel File Systems

DFS3, DFS4, DFS5, ...

- 9PB usable storage
- ~6GB/sec bandwidth/System
- Single Copy/No Snapshots

CRSP - Campus Research Storage Pool

- 1 PB usable storage
- Available anywhere on UCI Network
- Dual Copy of All Data
- Snapshots
- Highly available

<https://rcic.uci.edu/hpc3/specs.html>

High-level View of what things cost

No Cost Allocations

Role	HPC3 Core Hours	GPU Hours	Home Area Storage	DFS Storage	CRSP Storage
Faculty	200K hours/year ¹	By Request ~2K hours/year ¹	50GB	1TB in Pub	1 TB
Student	1000 hours	---	50GB	1TB in Pub	---

Cloud-like Costs

	HPC3 Core Hours	GPU Hours	Home Area Storage	DFS Storage	CRSP Storage
Faculty	\$.01/core hour	\$0.32/GPU hour	Not expandable	\$100/TB/5 years	\$60/TB/year
AWS Equivalent	C5n.large \$.063	P3.2xlarge \$1.95	---	---	S3 ² Standard \$242/TB/year

¹ Exact amounts dependent on # requests/available hardware

² Comparison difficult - S3 has higher durability, CRSP has no networking fee.

HPC3 Policies for CPU and memory scheduling

Partition	Default memory/core	Max memory/core	Default / Max runtime	Cost	Jobs preemption
CPU Partitions					
standard	3 GB	6 GB	2 day / 14 day	1 / core-hr	No
free	3 GB	18 GB	1 day / 3 day	0	Yes
highmem	6 GB	10 GB	2 day / 14 day	1 / core-hr	No
hugemem	18 GB	18 GB	2 day / 14 day	1 / core-hr	No
maxmem	1.5 TB/node	1.5 TB/node	1 day / 7 day	40 / node-hr	No
GPU Partitions					
gpu3	3 GB	9 GB	2 day / 14 day	1 / core-hr, 32 / GPU-hr	No
free-gpu	3 GB	9 GB	1 day / 3 day	0	Yes

HPC3 ML/DL Software Stack

[/opt/rcic/Modules/modulefiles/AI-LEARNING](#)

pytorch/1.11.0 pytorch/2.0.1 pytorch/2.3.0 tensorflow/2.8.0 tensorflow/2.16.1 tensorRT/8.4.2.4
tensorRT/8.6.1.6

[/opt/rcic/Modules/modulefiles/COMPILERS](#)

cuda sdk/22.9

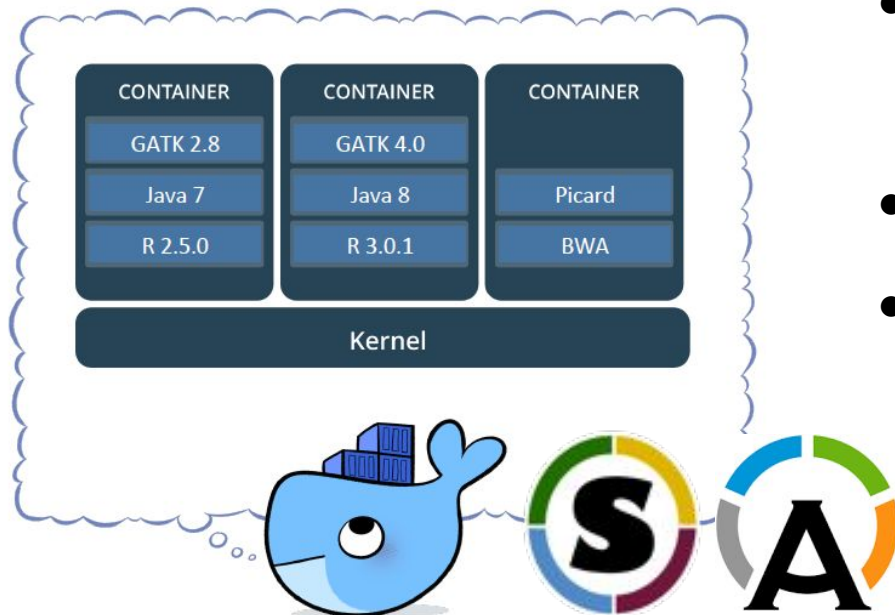
[/opt/rcic/Modules/modulefiles/TOOLS](#)

cuda/11.4.0 cuda/11.7.1 cuda/12.2.0

[/opt/rcic/Modules/modulefiles/LANGUAGES](#)

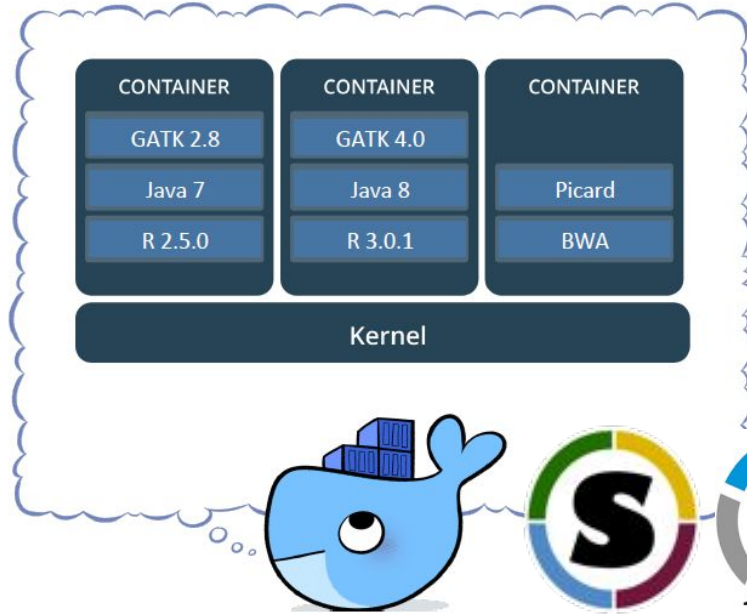
anaconda/2020.07 go/1.20.4 julia/1.8.2 MATLAB/R2023b python/3.8.0 R/4.4.2
anaconda/2021.11 go/1.22.3 julia/1.9.3 miniconda3/23.5.2 python/3.10.2
anaconda/2022.05 java/1.8.0 mamba/24.3.0 miniconda3/24.9.2 R/4.0.4
anaconda/2024.06 java/11 MATLAB/R2020a perl/5.30.0 R/4.1.2
bioconda/4.8.3 java/17 MATLAB/R2020b perl/5.34.1 R/4.2.2
go/1.17.7 julia/1.6.0 MATLAB/R2021b python/2.7.17 R/4.3.3

HPC3 Software via Containers



- HPC3 supports **Singularity** and **Apptainer** containers natively, as well as **Docker** containers through porting
- Containers are isolated, but share OS and bins/libraries
- Provide highly customized software environment apart from the host system

Container Benefits



Modified from <https://www.docker.com/what-container>

- **Portability:** containers can be published and shared via cloud-based container hubs (<https://hub.docker.com/>, <https://dockstore.org/>, <https://www.singularity-hub.org/>) or transferred directly as image files
- **Versioning:** container build files can be stored in git/github repositories
- **Reproducibility:** published containers are immutable, and provides a snapshot of the computing environment used to run analysis

Jupyter Ecosystem - Running Containers Interactively on HPC3



Sign In






Username:

Password:

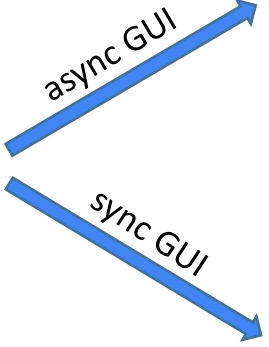
Sign In

HPC3 User Authentication

<https://hpc3.rcic.uci.edu/biojhub4/>

-  Singularity
-  Spatial Transcriptomics
-  Pytorch/TensorFlow GPU Programming
-  OpenCV Computer Vision
-  ...

Containerized software environment



Comparing ML/DL/AI Modeling Frameworks:





TensorFlow

TensorFlow was released on November 15th, 2015 by Google as its open-source framework for machine learning. It supports deep-learning, neural networks, and general numerical computations on CPUs, GPUs, and clusters of GPUs.

- **Mature Ecosystem and Deployment:** TensorFlow, developed by Google, boasts a more established ecosystem, extensive tools for deployment (e.g., TensorFlow Serving, TensorFlow Lite), and a vast collection of pre-trained models on TensorFlow Hub.
- **Scalability and Production:** It is well-suited for large-scale, production-level deployments and complex AI projects, with robust features for distributed training and optimization.
- **Keras Integration:** TensorFlow 2.x integrates Keras, simplifying model building with a high-level API.



HPC3/GRTHub supported DL applications based on TF:



DeepVariant is a deep learning-based variant caller that takes aligned reads (in BAM or CRAM format), produces pileup image tensors from them, classifies each tensor using a **convolutional neural network**, and finally reports the results in a standard VCF or gVCF file.



DeepCell is a deep learning library for single-cell analysis of biological images. It allows users to apply pre-existing models to imaging data as well as to develop new deep learning models for single-cell analysis. This library specializes in models for **cell segmentation** (whole-cell and nuclear) in 2D and 3D images as well as cell tracking in 2D time-lapse datasets. These models are applicable to data ranging from multiplexed images of tissues to dynamic live-cell imaging movies.

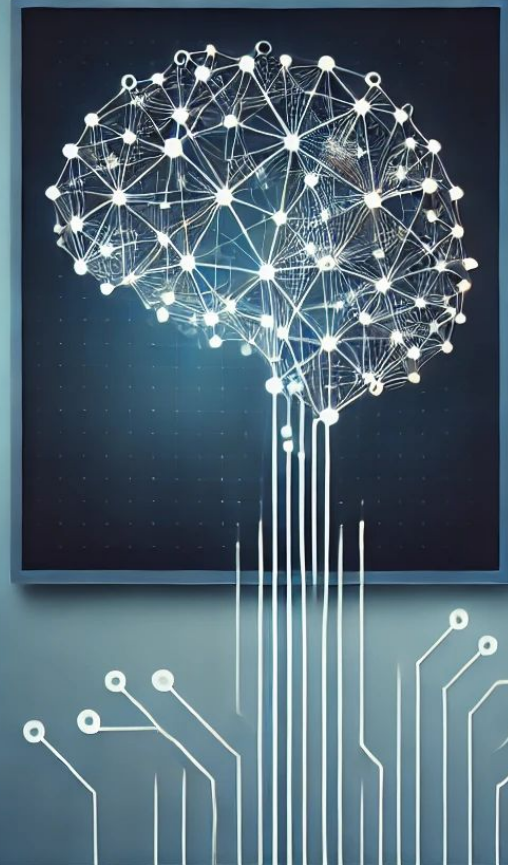
IMC-Denoise

IMC-Denoise is a content aware **denoising** pipeline based on **diffusion model** to enhance Imaging Mass Cytometry

PyTorch

PyTorch's was released in September 2016 by Facebook AI Research (now Meta AI) and was made open source in 2017. Since 2022, it has been under the stewardship of the PyTorch Foundation, which is part of the Linux Foundation.

- **Pythonic and Dynamic:** PyTorch is known for its intuitive, Python-like syntax and dynamic computation graph, making it easier for debugging and experimentation.
- **Research-Oriented:** It is widely favored in academic research due to its flexibility and ease of prototyping novel architectures.
- **Growing Ecosystem:** While newer than TensorFlow, its community and ecosystem are rapidly expanding, with increasing tools and resources available.



HPC3/GRTHub supported DL applications based on Pytorch



CellPose-SAM (CellPose4)

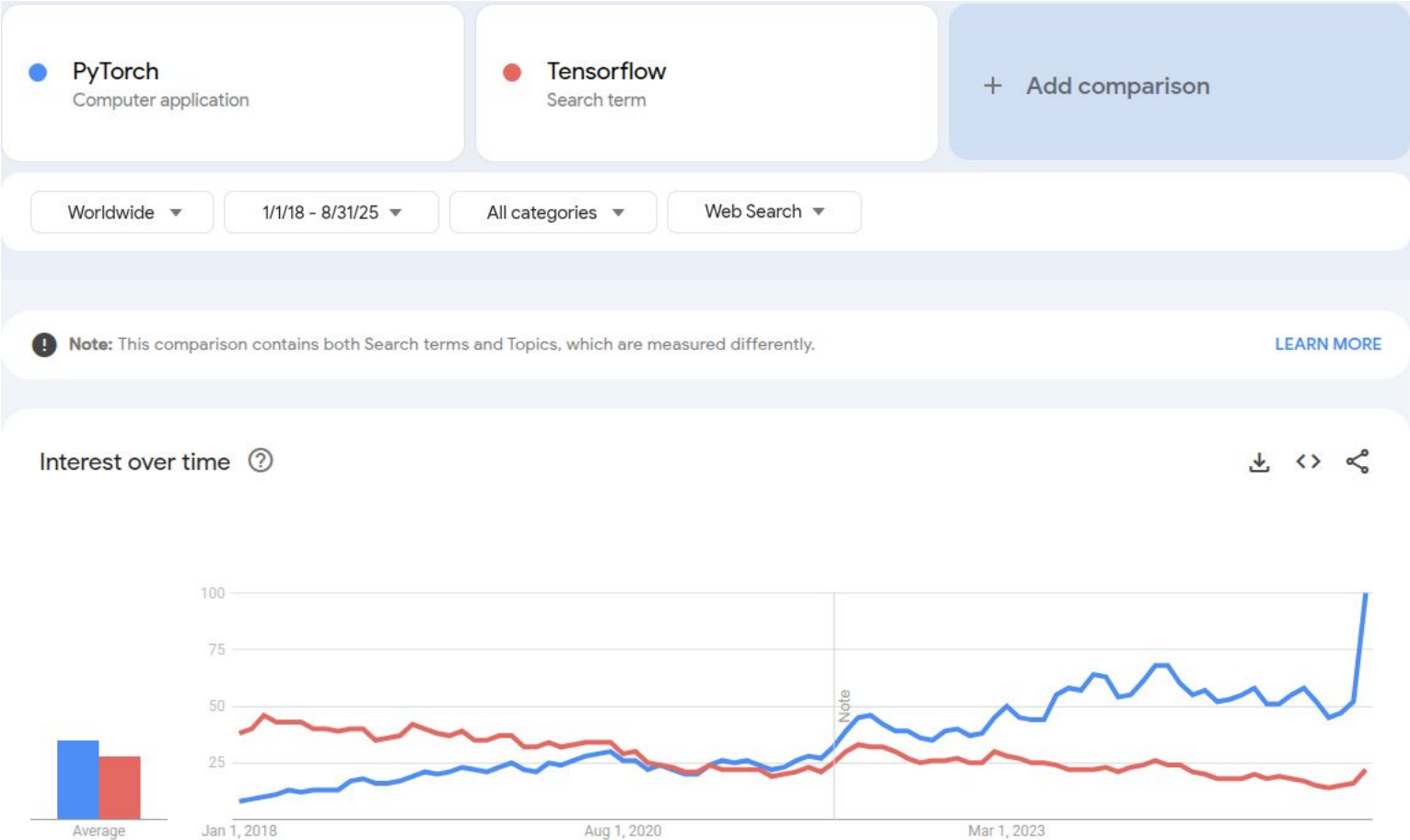
A generalist algorithm for cellular segmentation, designed to work across various cell types and imaging modalities. It uses deep learning techniques, combining U-Net convolutional neural networks to perform segmentation tasks, and pretrained transformer backbone of a foundation model (SAM) for “superhuman” generalization.



scGPT

scGPT is a generative AI foundation model inspired by Large Language Models (LLMs), but trained on gene expression data from 33 million cells rather than human text to generate embeddings for single-cell multi-omics.

Framework interest over time

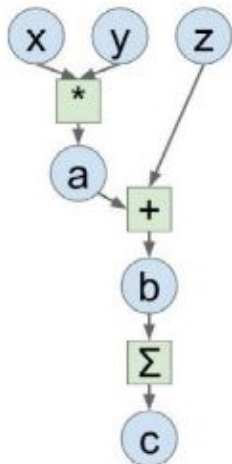


Why Pytorch?

- Strong GPU/TPU support, stable, less dependency issues
- **Autograd**- automatic differentiation
- Many algorithms and components are already implemented, such as *torch.nn.Transformer*
- Pytorch tensor similar to NumPy

Why PyTorch?

Computation Graph



Numpy

```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```

Tensorflow

```
import numpy as np
np.random.seed(0)
import tensorflow as tf

N, D = 3, 4

with tf.device('/gpu:0'):
    x = tf.placeholder(tf.float32)
    y = tf.placeholder(tf.float32)
    z = tf.placeholder(tf.float32)

    a = x * y
    b = a + z
    c = tf.reduce_sum(b)

grad_x, grad_y, grad_z = tf.gradients(c, [x, y, z])

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        y: np.random.randn(N, D),
        z: np.random.randn(N, D),
    }
    out = sess.run([c, grad_x, grad_y, grad_z],
                    feed_dict=values)
    c_val, grad_x_val, grad_y_val, grad_z_val = out
```

PyTorch

```
import torch

N, D = 3, 4

x = torch.rand((N, D), requires_grad=True)
y = torch.rand((N, D), requires_grad=True)
z = torch.rand((N, D), requires_grad=True)

a = x * y
b = a + z
c = torch.sum(b)

c.backward()
```



DL modeling quickstart

Pytorch Data structure (Tensors)

Tensors are similar to NumPy's ndarrays, with the addition being that Tensors can also be used on a GPU to accelerate computing.

Common operations for creation and manipulation of these Tensors are similar to those for ndarrays in NumPy. (rand, ones, zeros, indexing, slicing, reshape, transpose, cross product, matrix product, element wise multiplication)

Tensors

Attributes of a tensor 't':

- `t = torch.randn(1)`

`requires_grad`- making a trainable parameter

- By default False
- Turn on:
 - `t.requires_grad_()` or
 - `t = torch.randn(1, requires_grad=True)`
- Accessing tensor value:
 - `t.data`
- Accessing tensor gradient
 - `t.grad`

`grad_fn`- history of operations for autograd

- `t.grad_fn`

```
1 import torch
2
3 N, D = 3, 4
4
5 x = torch.rand((N, D), requires_grad=True)
6 y = torch.rand((N, D), requires_grad=True)
7 z = torch.rand((N, D), requires_grad=True)
8
9 a = x * y
10 b = a + z
11 c = torch.sum(b)
12
13 c.backward()
14
15 print(c.grad_fn)
16 print(x.data)
17 print(x.grad)
```

```
<SumBackward0 object at 0x7fd0cb970cc0>
tensor([[0.4118, 0.2576, 0.3470, 0.0240],
        [0.7797, 0.1519, 0.7513, 0.7269],
        [0.8572, 0.1165, 0.8596, 0.2636]])
tensor([[0.6855, 0.9696, 0.4295, 0.4961],
        [0.3849, 0.0825, 0.7400, 0.0036],
        [0.8104, 0.8741, 0.9729, 0.3821]])
```

Attributes of a Tensor

Tensor attributes describe their shape, datatype, and the device on which they are stored.

```
tensor = torch.rand(3,4)

print(f"Shape of tensor: {tensor.shape}")
print(f"Datatype of tensor: {tensor.dtype}")
print(f"Device tensor is stored on: {tensor.device}")
```

Out:

```
Shape of tensor: torch.Size([3, 4])
Datatype of tensor: torch.float32
Device tensor is stored on: cpu
```

Operations on Tensors

By default, tensors are created on the CPU. We need to explicitly move tensors to the GPU using `.to` method (after checking for GPU availability). Keep in mind that copying large tensors across devices can be expensive in terms of time and memory!

```
# We move our tensor to the GPU if available  
if torch.cuda.is_available():  
    tensor = tensor.to('cuda')
```

Autograd

- Automatic Differentiation Package
- Don't need to worry about partial differentiation, chain rule etc.
 - `backward()` does that
- Gradients are accumulated for each step by default:
 - Need to zero out gradients after each update
 - `tensor.grad_zero()`

```
# Create tensors.
x = torch.tensor(1., requires_grad=True)
w = torch.tensor(2., requires_grad=True)
b = torch.tensor(3., requires_grad=True)

# Build a computational graph.
y = w * x + b    # y = 2 * x + 3

# Compute gradients.
y.backward()

# Print out the gradients.
print(x.grad)    # x.grad = 2
print(w.grad)    # w.grad = 1
print(b.grad)    # b.grad = 1
```


Loading a Dataset

```
import torch
from torch.utils.data import Dataset
from torchvision import datasets
from torchvision.transforms import ToTensor, Lambda
import matplotlib.pyplot as plt

training_data = datasets.FashionMNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor()
)

test_data = datasets.FashionMNIST(
    root="data",
    train=False,
    download=True,
    transform=ToTensor()
)
```

Creating a Custom Dataset for your files

```
import os
import pandas as pd
from torchvision.io import read_image

class CustomImageDataset(Dataset):
    def __init__(self, annotations_file, img_dir, transform=None, target_transform=None):
        self.img_labels = pd.read_csv(annotations_file)
        self.img_dir = img_dir
        self.transform = transform
        self.target_transform = target_transform

    def __len__(self):
        return len(self.img_labels)

    def __getitem__(self, idx):
        img_path = os.path.join(self.img_dir, self.img_labels.iloc[idx, 0])
        image = read_image(img_path)
        label = self.img_labels.iloc[idx, 1]
        if self.transform:
            image = self.transform(image)
        if self.target_transform:
            label = self.target_transform(label)
        sample = {"image": image, "label": label}
        return sample
```

Preparing your data for training with DataLoaders

The `Dataset` retrieves our dataset's features and labels one sample at a time. While training a model, we typically want to pass samples in “minibatches”, reshuffle the data at every epoch to reduce model overfitting, and use Python's `multiprocessing` to speed up data retrieval.

`DataLoader` is an iterable that abstracts this complexity for us in an easy API.

```
from torch.utils.data import DataLoader

train_dataloader = DataLoader(training_data, batch_size=64, shuffle=True)
test_dataloader = DataLoader(test_data, batch_size=64, shuffle=True)
```

Iterate through the DataLoader

We have loaded that dataset into the `Dataloader` and can iterate through the dataset as needed. Each iteration below returns a batch of `train_features` and `train_labels` (containing `batch_size=64` features and labels respectively). Because we specified `shuffle=True`, after we iterate over all batches the data is shuffled (for finer-grained control over the data loading order, take a look at [Samplers](#)).

```
# Display image and label.
train_features, train_labels = next(iter(train_dataloader))
print(f"Feature batch shape: {train_features.size()}")
print(f"Labels batch shape: {train_labels.size()}")
img = train_features[0].squeeze()
label = train_labels[0]
plt.imshow(img, cmap="gray")
plt.show()
print(f"Label: {label}")
```

Building DL model by designing neural network layers

torch.nn.Module

```
import os
import torch
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
```



Get Device for Training

We want to be able to train our model on a hardware accelerator like the GPU, if it is available. Let's check to see if `torch.cuda` is available, else we continue to use the CPU.

```
device = 'cuda' if torch.cuda.is_available() else 'cpu'  
print('Using {} device'.format(device))
```

Out:

```
Using cuda device
```

Define the Class

We define our neural network by subclassing `nn.Module`, and initialize the neural network layers in `__init__`. Every `nn.Module` subclass implements the operations on input data in the `forward` method.

```
class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10),
            nn.ReLU()
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits
```

Define the Class

We create an instance of `NeuralNetwork`, and move it to the `device`, and print its structure.

```
model = NeuralNetwork().to(device)
print(model)
```

Out:

```
NeuralNetwork(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (linear_relu_stack): Sequential(
    (0): Linear(in_features=784, out_features=512, bias=True)
    (1): ReLU()
    (2): Linear(in_features=512, out_features=512, bias=True)
    (3): ReLU()
    (4): Linear(in_features=512, out_features=10, bias=True)
    (5): ReLU()
  )
)
```


Define the Class

To use the model, we pass it the input data. This executes the model's `forward`, along with some **background operations**. Do not call `model.forward()` directly!

Calling the model on the input returns a 10-dimensional tensor with raw predicted values for each class. We get the prediction probabilities by passing it through an instance of the `nn.Softmax` module.

```
X = torch.rand(1, 28, 28, device=device)
logits = model(X)
pred_probab = nn.Softmax(dim=1)(logits)
y_pred = pred_probab.argmax(1)
print(f"Predicted class: {y_pred}")
```

Out:

```
Predicted class: tensor([2], device='cuda:0')
```

Loss Function

Common loss functions include `nn.MSELoss` (Mean Square Error) for regression tasks, and `nn.NLLLoss` (Negative Log Likelihood) for classification. `nn.CrossEntropyLoss` combines `nn.LogSoftmax` and `nn.NLLLoss`.

We pass our model's output logits to `nn.CrossEntropyLoss`, which will normalize the logits and compute the prediction error.

```
# Initialize the loss function  
loss_fn = nn.CrossEntropyLoss()
```



Optimizer

We initialize the optimizer by registering the model's parameters that need to be trained, and passing in the learning rate hyperparameter.

```
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

Inside the training loop, optimization happens in three steps:

- Call `optimizer.zero_grad()` to reset the gradients of model parameters. Gradients by default add up; to prevent double-counting, we explicitly zero them at each iteration.
- Backpropagate the prediction loss with a call to `loss.backward()`. PyTorch deposits the gradients of the loss w.r.t. each parameter.
- Once we have our gradients, we call `optimizer.step()` to adjust the parameters by the gradients collected in the backward pass.

Full Implementation - Train Loop

```
def train_loop(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    for batch, (X, y) in enumerate(dataloader):
        # Compute prediction and loss
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

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

Full Implementation - Test Loop

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

    with torch.no_grad():
        for X, y in dataloader:
            pred = model(X)
            test_loss += loss_fn(pred, y).item()
            correct += (pred.argmax(1) == y).type(torch.float).sum().item()

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

Full Implementation

```
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

epochs = 10
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)
print("Done!")
```

Full Implementation

Out:

```
Epoch 1
-----
loss: 2.299511 [  0/60000]
loss: 2.301767 [ 6400/60000]
loss: 2.289777 [12800/60000]
loss: 2.291731 [19200/60000]
loss: 2.269755 [25600/60000]
loss: 2.261175 [32000/60000]
loss: 2.258553 [38400/60000]
loss: 2.240743 [44800/60000]
loss: 2.260818 [51200/60000]
loss: 2.243683 [57600/60000]
Test Error:
  Accuracy: 37.3%, Avg loss: 0.035121

Epoch 2
-----
loss: 2.229830 [  0/60000]
loss: 2.241497 [ 6400/60000]
loss: 2.221580 [12800/60000]
```

Saving and Loading Model Weights

PyTorch models store the learned parameters in an internal state dictionary, called `state_dict`. These can be persisted via the `torch.save` method:

```
model = models.vgg16(pretrained=True)
torch.save(model.state_dict(), 'model_weights.pth')
```

To load model weights, you need to create an instance of the same model first, and then load the parameters using `load_state_dict()` method.

```
model = models.vgg16() # we do not specify pretrained=True, i.e. do not load default weights
model.load_state_dict(torch.load('model_weights.pth'))
model.eval()
```


Saving and Loading Models with Shapes

When loading model weights, we needed to instantiate the model class first, because the class defines the structure of a network. We might want to save the structure of this class together with the model, in which case we can pass `model` (and not `model.state_dict()`) to the saving function:

```
torch.save(model, 'model.pth')
```

We can then load the model like this:

```
model = torch.load('model.pth')
```

Quick tour of HPC3's Biohub4 (Jupyterhub)

Using your favorite browser go to: <https://hpc3.rcic.uci.edu/biojhub4/hub/login> You will see the following screen where you will Use your usual HPC3 credentials to sign in:

Sign in

Username:

Password:

Sign In

After authentication you will see a screen with server options as in the figure below:

Server Options

Select Partition/Reservation to Use
free-gpu

Select Account to Charge
iychang

Specify number of CPU cores (max 32)
6

memory per CPU core (max 6Gb per core for standard, 10Gb for maxmem)
4

Specify runtime (HH:MM:SS format, 19hr max, 6hr default)
06:00:00

Select a Containerized Notebook Image
CellPose-Sam (Spatial Genomics) + Base 2025Q3 (Pytorch)

Resume last session if available

Enable full HPC3 software stack (override container stack)

Start

For this workshop, modify the *Select Account to Charge* to be one of your Slurm accounts, change number of CPUs to **6** and the memory per CPU core to **6**, select the “CellPose-Sam (Spatial Genomics) + Base 2025Q3 (Pytorch)” container. Also change Partition to gpu (require gpu account) or free-gpu partition in order to run this notebook with gpu acceleration. If gpu partition is not available, please use standard or free partition for cpu operation instead (10x slower).” Press **start** when done with selecting options.

Main Jupyter Interface

The screenshot displays the Jupyter Notebook interface. On the left is a file browser with a search bar and a table of files. On the right is the Launcher screen, which offers various options for creating a new notebook or console.

File Browser (Left):

Name	Last Modified
cellranger	11 hours ago
parsebio	3 hours ago
seurat4shinycode	9 hours ago
Seurat4_GRTHworkshop_Jan23.ipynb	10 hours ago
Single_Cell_Workshop.ipynb	3 hours ago

Launcher Screen (Right):

biojhub3_dir/dfs3a/workshop

Notebook

- Python 3 (ipykernel)
- Bash
- R
- RStudio [?]
- Shiny [?]

Console

- Python 3 (ipykernel)
- Bash
- R

Other

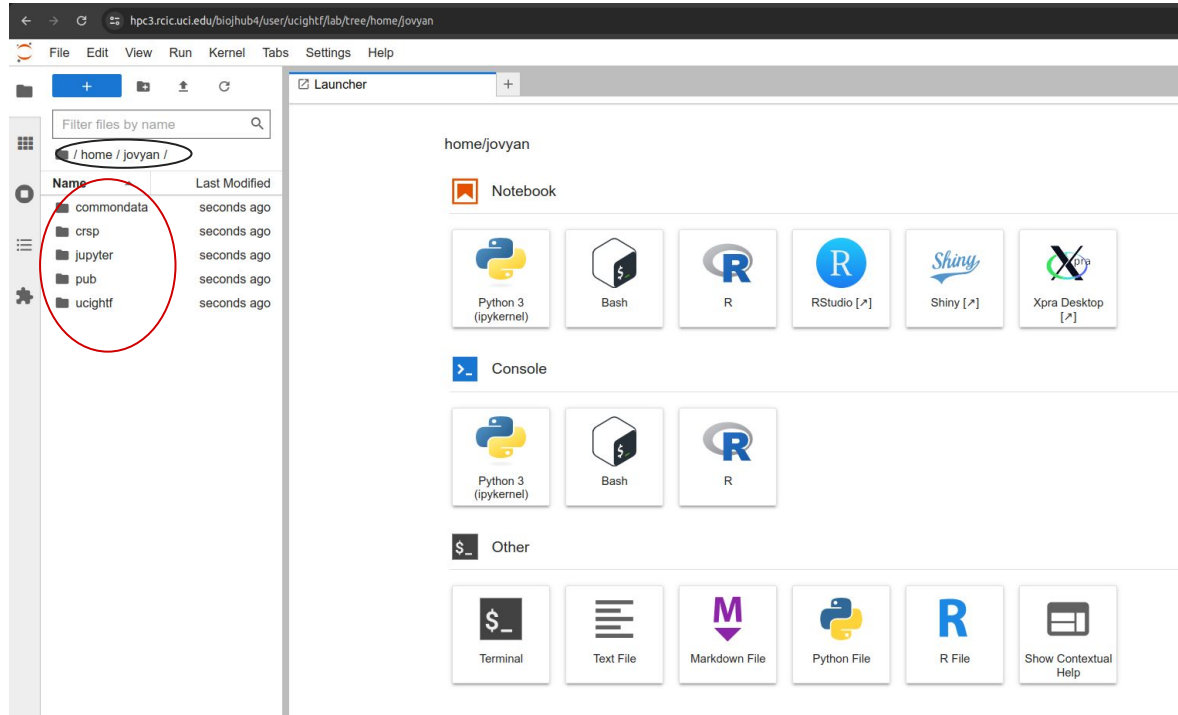
- Terminal
- Text File
- Markdown File
- Python File
- R File
- Show Contextual Help

Once the notebook is done spawning, you will get a file browser on the left, and a Launcher screen with a number GUI apps you can use on the right.

Fictitious home directory and short cuts

Default Jupyter container user

Shortcuts to oft used paths



<https://hpc3.rcic.uci.edu/biojhub4/>

[/dfs8/commondata/workshop/DeepLearning](#)

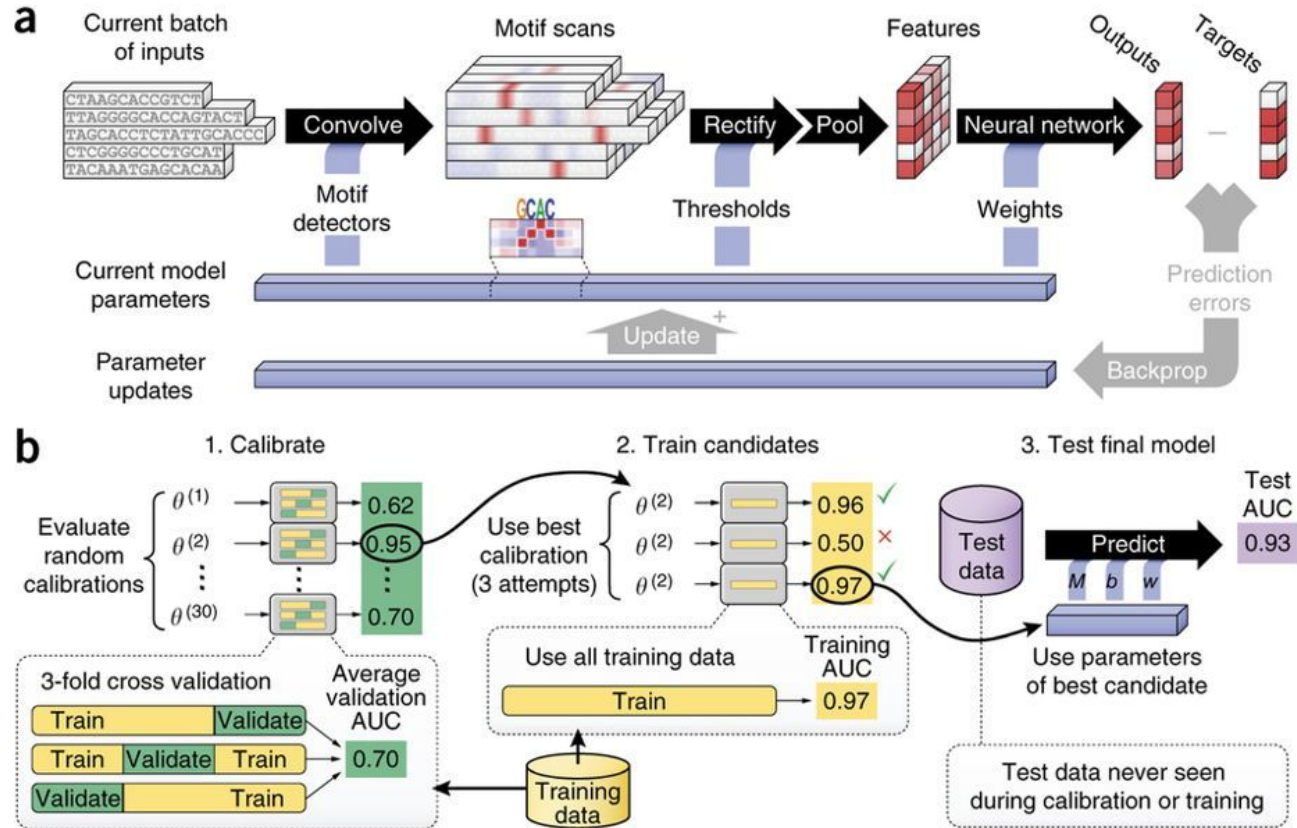
Hands-on session

- Log into biohub4 via your HPC3 credentials
<https://hpc3.rcic.uci.edu/biohub4/>
 - make sure to include the trailing /
- Find the workshop folder under commondata
- Open the DeepLearning folder under workshop
- Save a copy of the notebooks to your home directory
- Follow the instructions in Jupyter notebook

Notes on training Pytorch based genomics DL models

Deepbind

- Implemented in 2015 before Pytorch or TensorFlow
- Ported to Pytorch just combining main Pytorch functions.
- Not optimized for real analysis, but great example for understanding how to customize your own model in Pytorch



CellPose-SAM

On the HPC3, please use the "CellPose-Sam (Spatial Genomics) + Base 2025Q3 (Pytorch)" container with gpu (require gpu account) or free-gpu partition in order to run this notebook with gpu acceleration. If gpu partition is not available, please use standard or free partition for cpu operation instead (10x slower).

- Based on Pytorch, but highly structured wrappers and GUI allow users to apply model predictions, model training, and model optimization without directly interfacing with Pytorch.
- Use the XPRA Desktop to run CellPose GUI
- Open `run_Cellpose_SAM` notebook to see how to run CellPose-SAM in python
- Open `train_Cellpose_SAM` notebook to see how to train CellPose-SAM with your data.

scGPT-zero shot tutorial

On the HPC3, please use the "scGPT[GPU] (Genomics Foundation Model) + Base 2025Q3 (Pytorch)" container with gpu (require gpu account) or free-gpu partition in order to run this notebook. If gpu partition is not available, please use the regular "scGPT (Genomics Foundation Model) + Base 2025Q3 (Pytorch)" container with standard partition for cpu operation instead (10x slower).

- This tutorial covers the zero-shot integration with continual pre-trained scGPT. This particular workflow works for scRNA-seq datasets without fine-tuning (or any extensive training) of scGPT.
- Continual pre-trained scGPT (scGPT_CP) is a model that inherits the pre-trained scGPT whole-human model checkpoint, and is further supervised by extra cell type labels (using the [Tabula Sapiens](<https://tabula-sapiens-portal.ds.czbiohub.org/>) dataset) during the continual pre-training stage. We observed that the scGPT_CP model can achieve comparable or better zero-shot performance on cell embedding related tasks compared to the original checkpoint, especially on datasets with observable technical batch effects.

Be sure to stop your Jupyterhub notebook server after you are done. From the **File** menu choose **Hub Control Panel** and you will be forwarded to a screen similar where you can press on [Stop My Server](#) to shut down the server:

UCI Research Cyberinfrastructure Center Home Token Admin npw [Logout](#)

[Stop My Server](#) [My Server](#)

Named Servers

In addition to your default server, you may have additional 3 server(s) with names. This allows you to have more than one server running at the same time.

Server name	URL	Last activity	Actions
<input type="text" value="Name your server"/>	Add New Server		