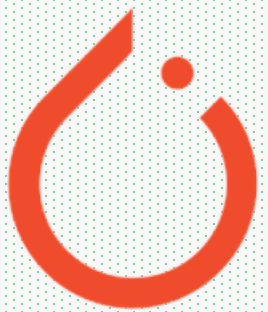
2nd National Symposium on Innovations in Intelligent Systems (2025)

School of Electrical and Electronics Engineering (SEEE)

Building PyTorch Models for Image Classification and Reconstruction



PyTorch

Dr. Susant Kumar Panigrahi

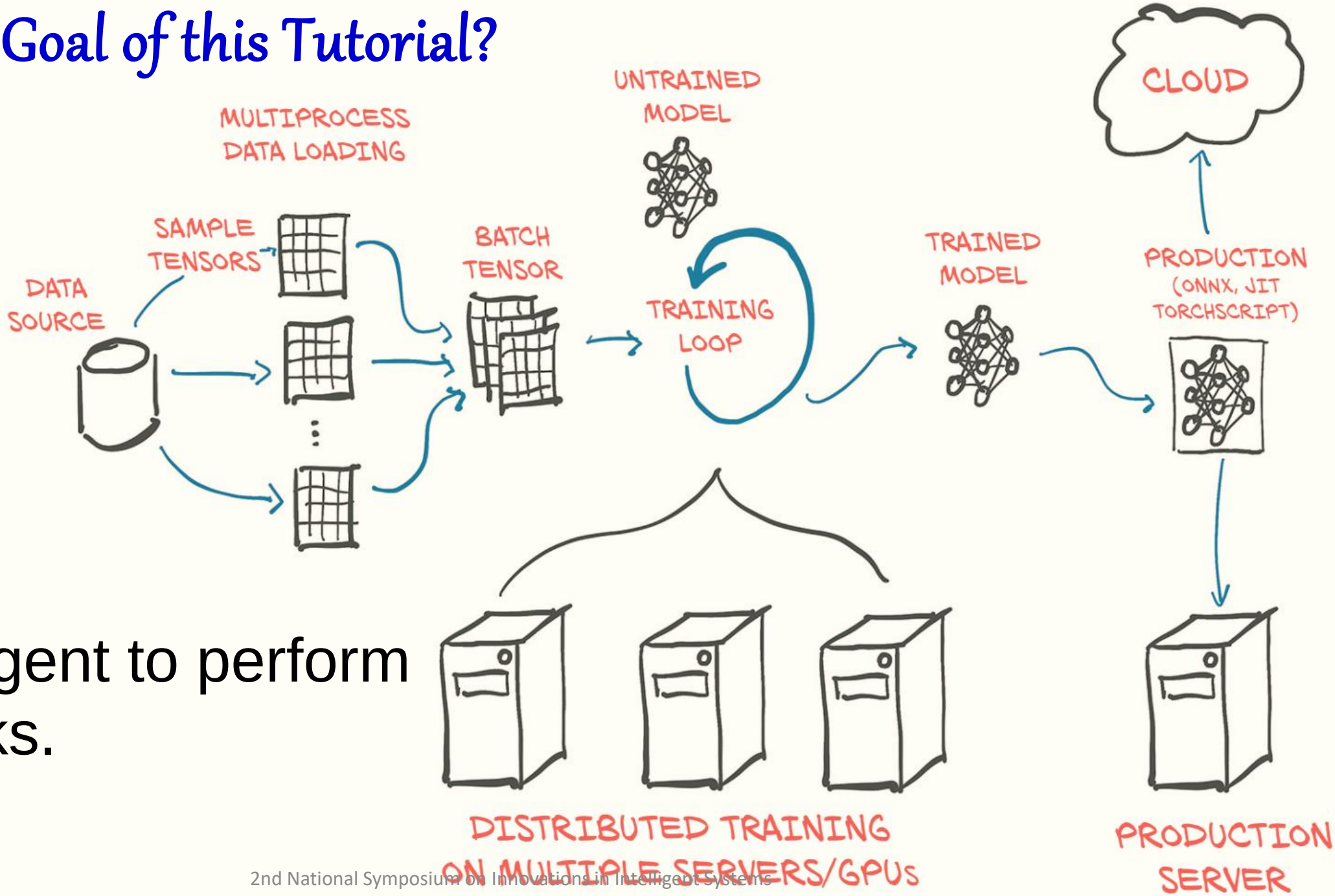
SERB Sponsored N-PDF

Department Of Electrical Engineering

Indian Institute of Technology Kharagpur

Email: spanigrahi@kgipian.iitkgp.ac.in

What is the Goal of this Tutorial?



Train an agent to perform useful tasks.

What is the Goal of this Tutorial?

How do train a model?

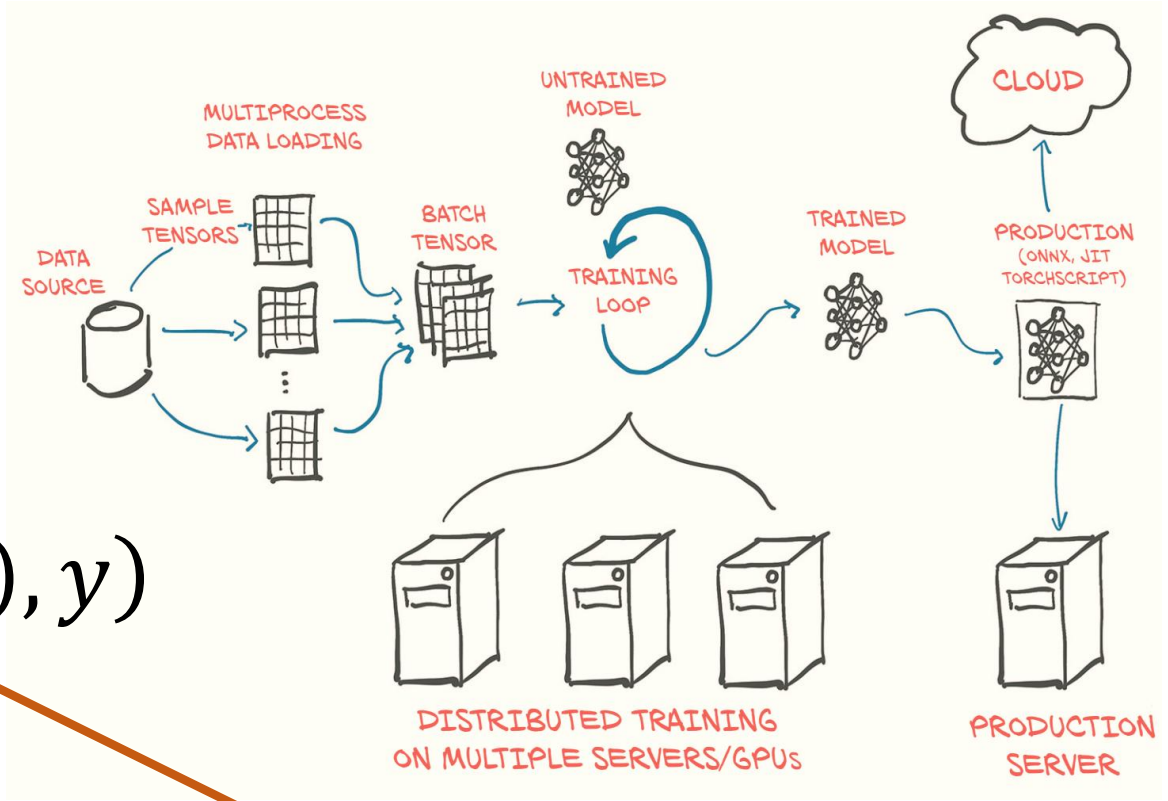
$$w^* = \underset{w}{\operatorname{argmin}} \sum_{(x,y)\in D} \mathcal{L}(f_w(x), y)$$

Gradient Descent

Dataset

Loss Function

Model
Neural Network *or*
Convolution Neural Network

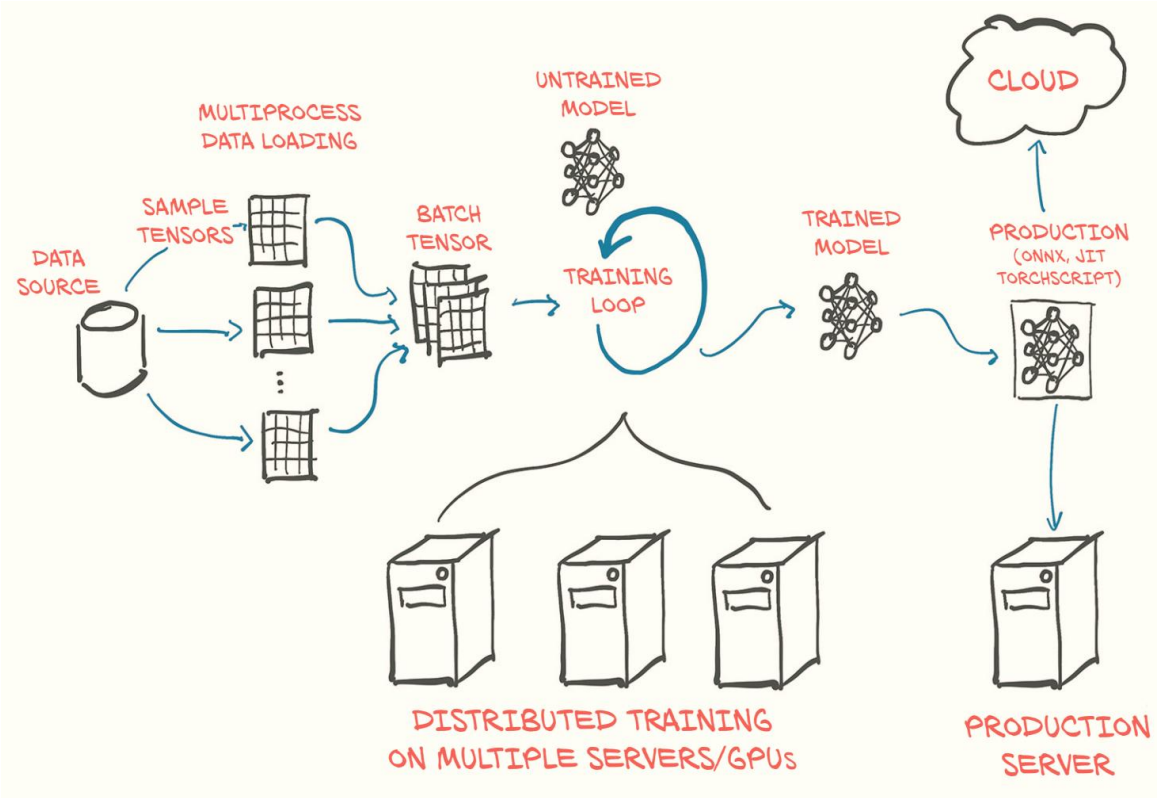


PyTorch does all of these!

What is PyTorch?

Python Library for:

- **Defining Neural Networks**
- **Automatic computation of gradients**
- **Network training and model inference**
- **And more ! (dataloader, optimizer, scheduler, Cuda and Many more.)**



$$w^* = \underset{w}{\operatorname{argmin}} \sum_{(x,y)\in D} \mathcal{L}(f_w(x), y)$$

Gradient Descent

Dataset

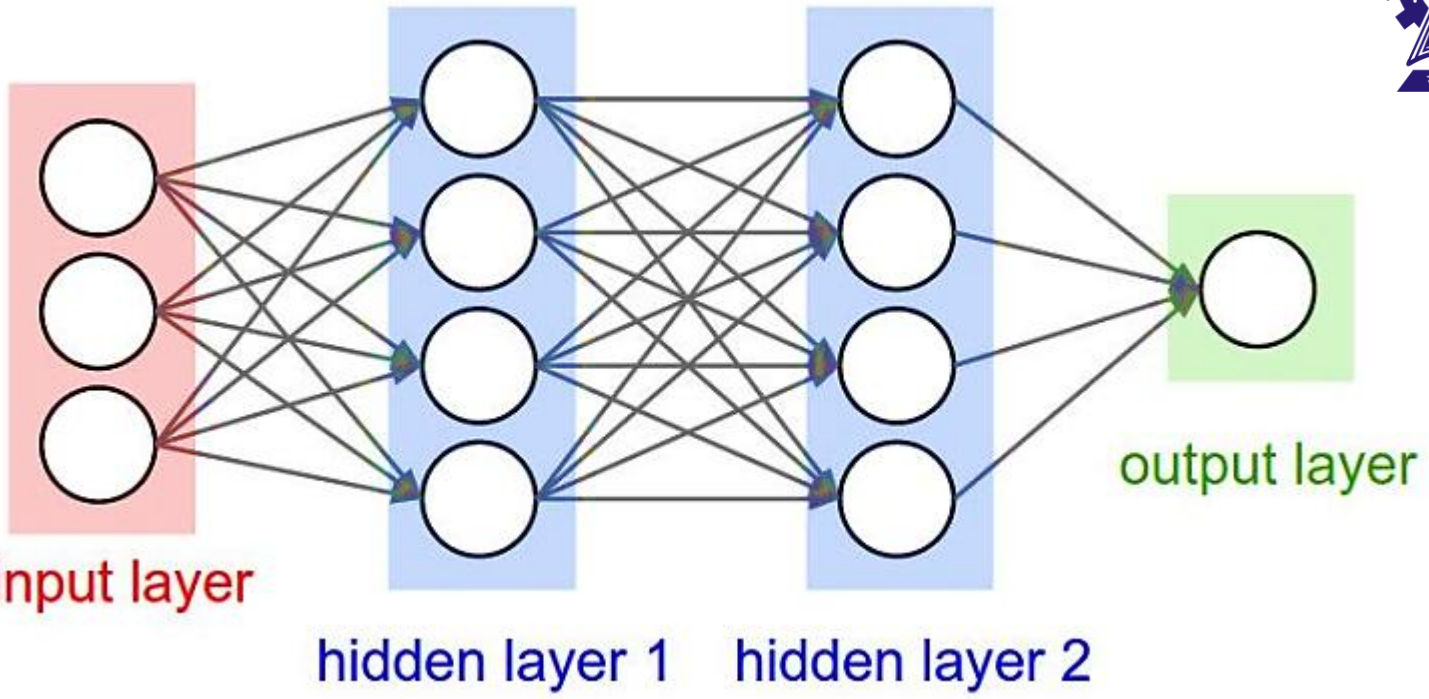
Loss Function

Network

PyTorch

How does PyTorch Work?

PyTorch and Computational Graph



Definition:

$$h_1 = \sigma(W_1 x) \quad h_2 = \sigma(W_2 h_1) \quad y = \sigma(W_3 h_2)$$

PyTorch Computation:

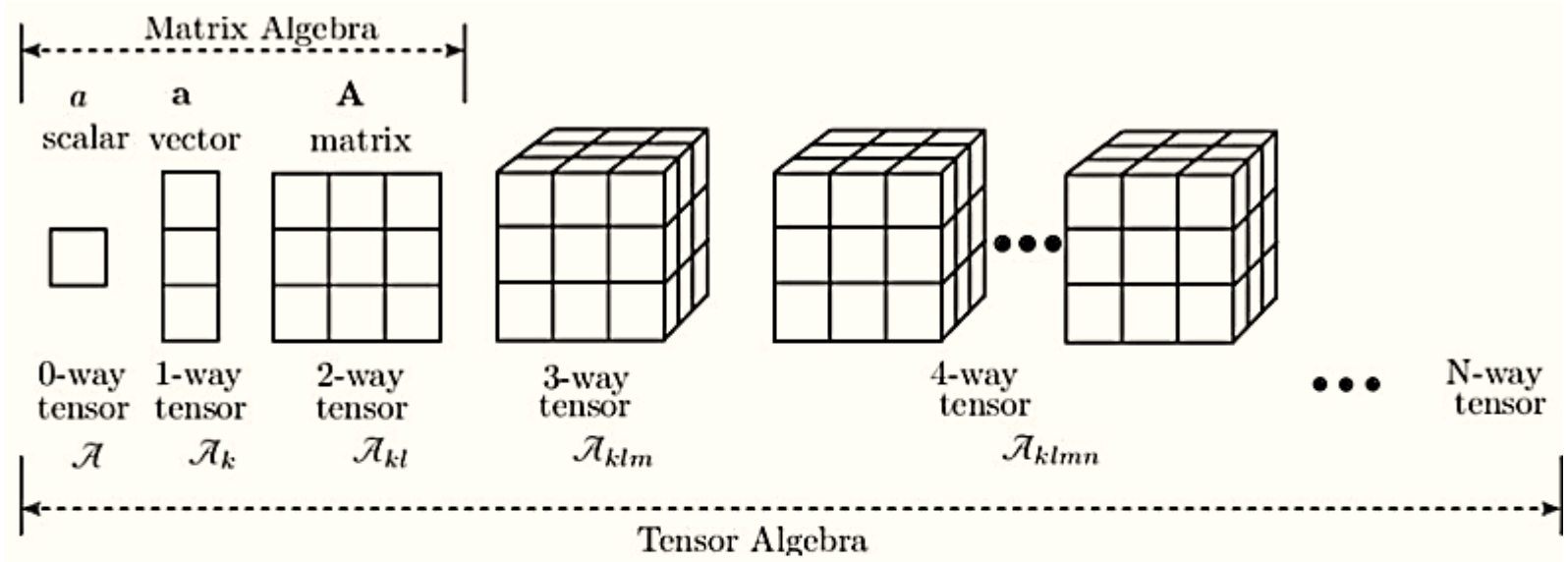
$$\frac{\partial y}{\partial W_1} = \frac{\partial y}{\partial h_2}\frac{\partial h_2}{\partial h_1}\frac{\partial h_1}{\partial W_1} \qquad \frac{\partial y}{\partial W_2} = \frac{\partial y}{\partial h_2}\frac{\partial h_2}{\partial W_2} \qquad \frac{\partial y}{\partial W_3}$$

[picture from Stanford's CS231n]

Multidimensional Arrays Computation

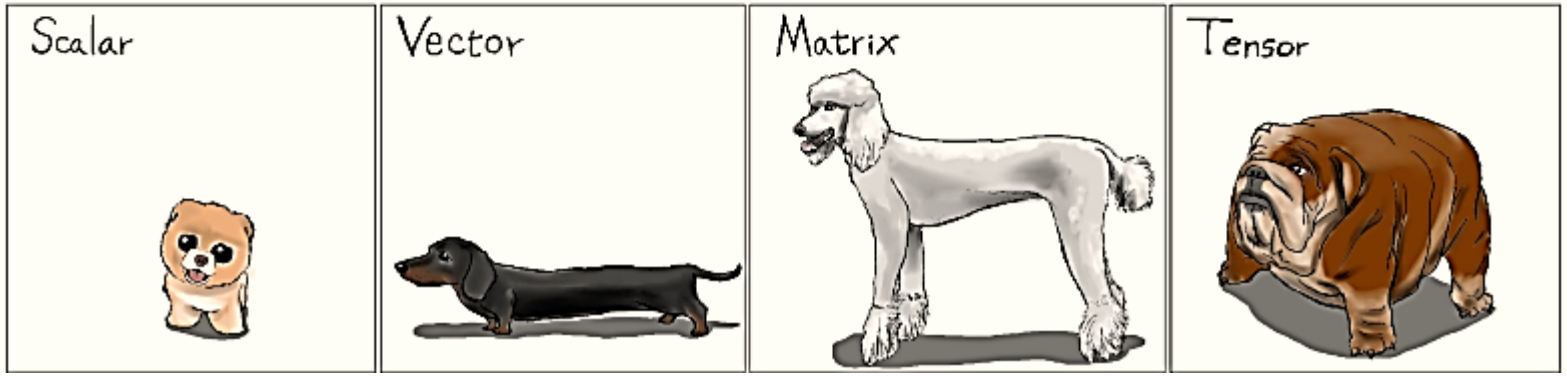
NumPy

- ✓ Fast CPU implementations
- ✓ **CPU-only**
- ✓ **No autodiff**
- ✓ **Imperative**



PyTorch

- ✓ Fast CPU implementations
- ✓ **Allows GPU**
- ✓ **Supports autodiff**
- ✓ **Imperative**



Other features include:

- Datasets and dataloading
- Common neural network operations
- Built-in optimizers (Adam, SGD, …)

PyTorch Basics

2. Basic Operations with Torch tensor

```
import torch
import numpy as np

# Creating an empty tensor.
x = torch.empty(3,3)
x1 = torch.ones(3,3)
x2 = torch.rand(3,3)
x3 = torch.zeros(3,3)
print(x)
print('Torch ones matrix:', x1)
print('Torch rand matrix:', x2)
print('Torch zeros matrix:', x3)
```

```
x = torch.rand(2,2)
y = torch.rand(2,2)
z = x + y print('Addition:', z)
z1 = torch.add(x,y) print('Addition:', z1)
# Inplace addition
y.add_(x)
print('Inplace Addition:', y)

# Other Operations
z = y - x
print('Subtraction:', z)
z1 = torch.subtract(y,x)
print(f'Subtraction: {z1}')
z1.subtract_(y)
print(f'Inplace subtraction: {z1}')
```

https://github.com/susant146/PyTorch_Basics_CNNmodels/blob/main/SI_01_Pytorch_Basics.ipynb

PyTorch Basics

3. Tensor Multiplication and Division

```
x = torch.rand(2,2)
y = torch.rand(2,2)
z = x*y
print('Multiplication: ', z)
z1 = torch.multiply(x,y) # torch.mul and torch.multiply
print('Element-Wise Multiplication:', z1)

# Inplace multiplication: y.multiply_(x)
z = x / y # torch.div(x,y)
print('Element-Wise Division: \n', z)
y.div_(x)
print('Inplace Division: \n', y) # Matrix multiplications

# Define two matrices
A = torch.tensor([[1, 2], [3, 4]]) # 2x2 matrix
B = torch.tensor([[5, 6, 1], [7, 8, 0]]) # 2x2 matrix


        # Alternative method: Using torch.matmul (supports broadcasting for higher dimensions)
        C_broadcast = torch.matmul(A, B)
        print("\nMatrix Multiplication Result (torch.matmul):") print(C_broadcast)
```

https://github.com/susant146/PyTorch_Basics_CNNmodels/blob/main/SI_01_Pytorch_Basics.ipynb

PyTorch Basics

4. Tensor Reshape & View

```
# Create a tensor
tensor = torch.arange(12)
print("Original Tensor:")
print(tensor)

# Reshape to 3x4 matrix
reshaped_tensor = tensor.reshape(3, 4)
print("\nReshaped to 3x4:")
print(reshaped_tensor)

flattened_tensor = reshaped_tensor.reshape(-1)
print("\nFlattened Tensor:")
print(flattened_tensor)
```

```
# Reshape to 2x6 using view # reshape:
Works regardless of tensor's memory layout.
# view: Faster but requires the tensor to
have contiguous memory. Use .contiguous()
before view if needed.
reshaped_view = tensor.view(2, 6)
print("\nReshaped to 2x6 using view:")
print(reshaped_view)

# Ensuring contiguity before view
non_contiguous = tensor.t()
contiguous_tensor =
non_contiguous.contiguous().view(-1)
```

```
# Automatically infer one dimension # Use -1 to let PyTorch infer the size of one dimension.

inferred_shape = tensor.reshape(3, -1)
print("\nReshaped with Inferred Dimension (2x-1):") print(inferred_shape)
```

PyTorch Basics

5. Numpy to PyTorch Tensor & Vice-Versa

```
a = torch.ones(5)
print(a)
b = a.numpy()
print('Numpy array: \n', b)
print('Type: \t', type(b))

# need to be very careful with place
operations
a.add_(2)
print('Modified a: \n', a)
print('See What happend! \n', b)
```

```
a = np.ones(5) print(a) b =
torch.from_numpy(a)

# default dtype is float64, change
# b = b.to(dtype=torch.int)
# or b = b.type(torch.IntTensor)
print(b)
# Inplace operation
a += 1
print('a ndarray: \t',a)
print('b Tensor: \t',b)
# Check this one
a = a+1
print('a ndarray: \t',a)
print('b Tensor: \t',b)
```

PyTorch Basics

6. Check for Available Device: Cuda or CPU

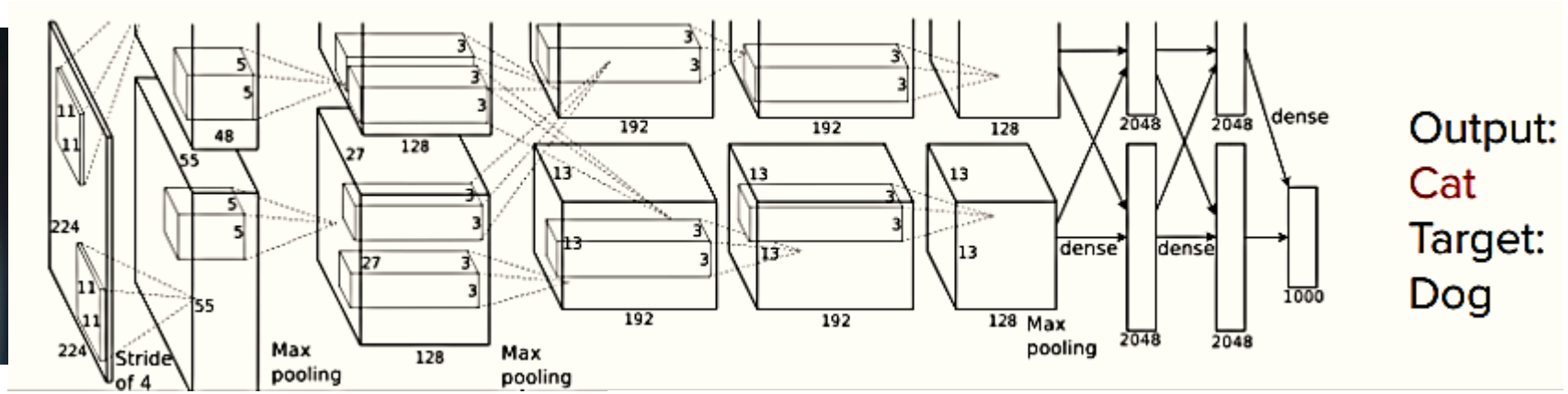
```
if torch.cuda.is_available():
    device = torch.device("cuda") # Use GPU
    print("CUDA is available. Using GPU.")
else:
    device = torch.device("cpu") # Use CPU
    print("CUDA is not available. Using CPU.")
```

Move a Tensors to CUDA

```
x = torch.ones(5, device=device)
y = torch.ones(5)
y = y.to(device)
z = x + y

print(f'The operation performed in {device}, output: {z}')
# As x, y and z are in CUDA, we can't convert them to numpy
# z1 = z.numpy() # Error Here
# bringing z to cpu
z1 = z.cpu() z1 = z1.numpy()

print(f"z1 is in cpu, output ndarray: {z1}")
```

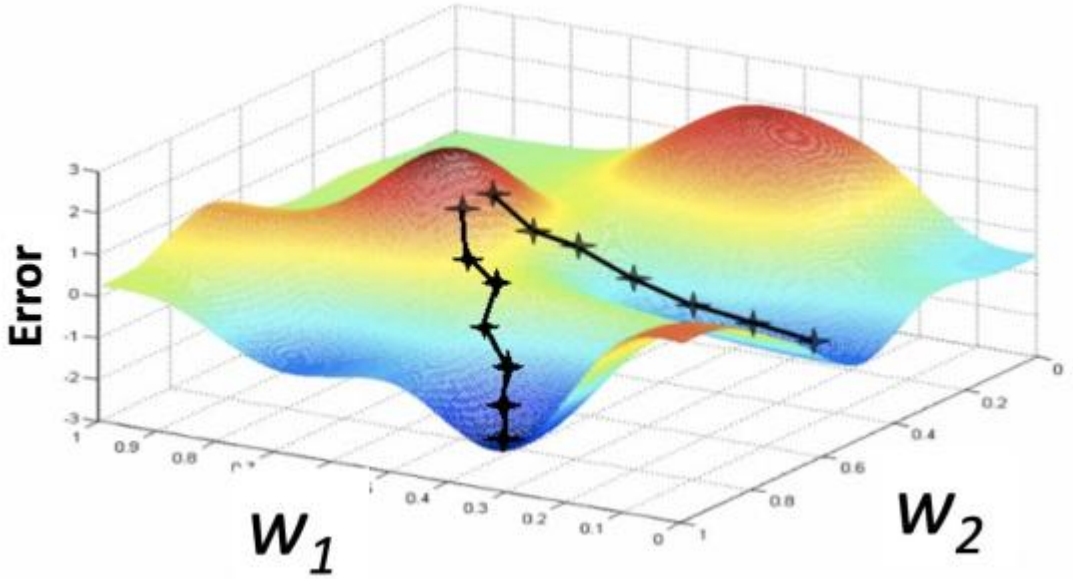
Gradient Descent & Backward Pass

Computational graph and Autograd with Pytorch



Learning as optimization:

1. An input-output function (an ANN): $y = f_w(x)$
2. A Loss function: $L = \mathcal{L}(y, y_{ref})$
3. Optimization problem: $w^* = \underset{w}{\operatorname{argmin}} \sum_{(x,y_{ref})\in D} \mathcal{L}(f_w(x$

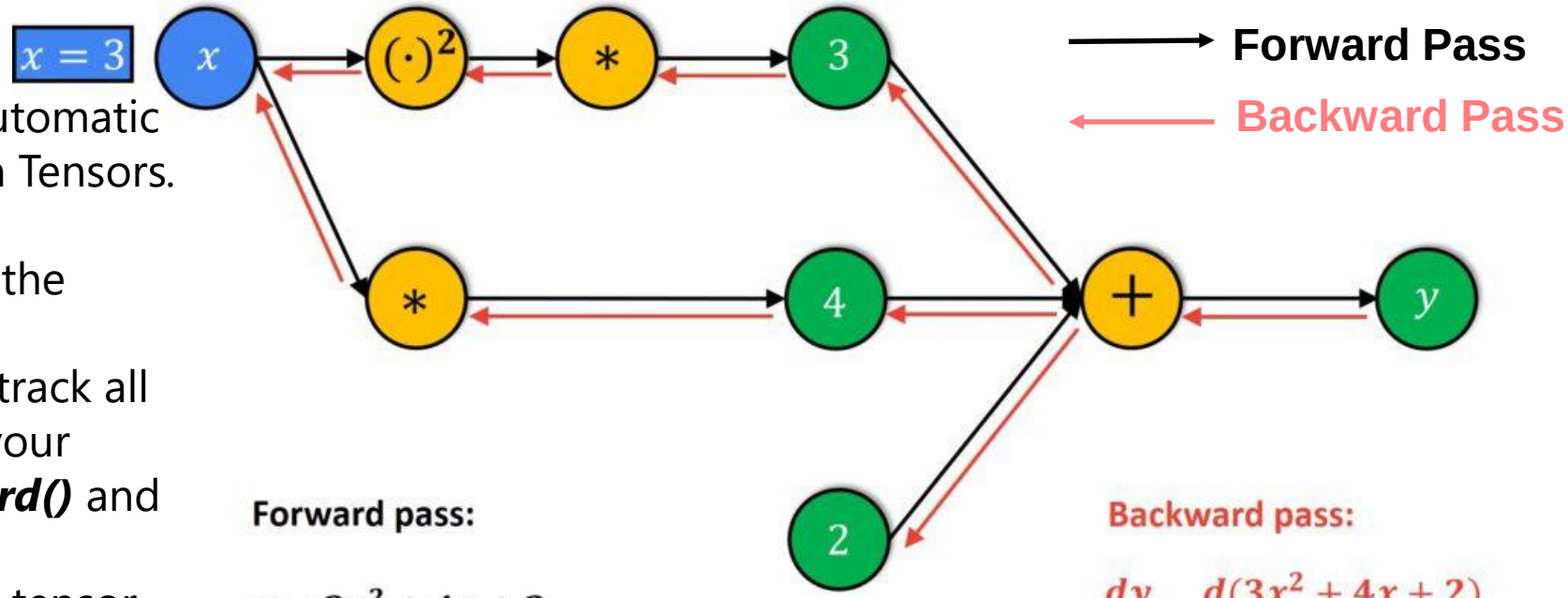


Pic courtesy: https://deeplearning.neuromatch.io/tutorials/W1D2_LinearDeepLearning/student/W1D2_Tutorial1.html

Gradient Descent & Backward Pass

Computational graph and Autograd with Pytorch

- ✓ The ***autograd*** package provides automatic differentiation for all operations on Tensors.
- ✓ ***torch.Tensor*** is the central class of the package. If you set its attribute ***.requires_grad*** as True, it starts to track all operations on it. When you finish your computation you can call ***.backward()*** and have all the gradients computed automatically. The gradient for this tensor will be accumulated into .grad attribute.
- ✓ To stop a tensor from tracking history, you can call ***.detach()*** to detach it from the computation history, and to prevent future computation from being tracked.



Forward pass:

$$y = 3x^2 + 4x + 2$$

$$y = 3 \cdot 9 + 4 \cdot 3 + 2$$

$$y = 41$$

Backward pass:

$$\frac{dy}{dx} = \frac{d(3x^2 + 4x + 2)}{dx}$$

$$\frac{dy}{dx} = 2 \cdot 3x + 4 = 6x + 4$$

$$\frac{dy}{dx} = 18 + 4 = 22$$

https://datahacker.rs/004-computational-graph-and-autograd-with-pytorch/

Gradient Descent & Backward Pass

Computational graph and Autograd with Pytorch

1. Autograd

```
import torch

x = torch.tensor(3., requires_grad=True)
y = 3*x**2 + 4*x + 2
print(y)

y.backward()
print(x.grad)
```

2. How to turn off the gradient calculations?

```
x = torch.tensor(3., requires_grad=True)
print(x)

x = x.requires_grad_(False)
print(x)

x = x.detach()
print(x)
```

3. Gradient accumulation effect

```
x = torch.tensor(3., requires_grad=True)

for epoch in range(3):
  y = 3*x**2 + 4*x + 2
  y.backward()

  print(x.grad)
```

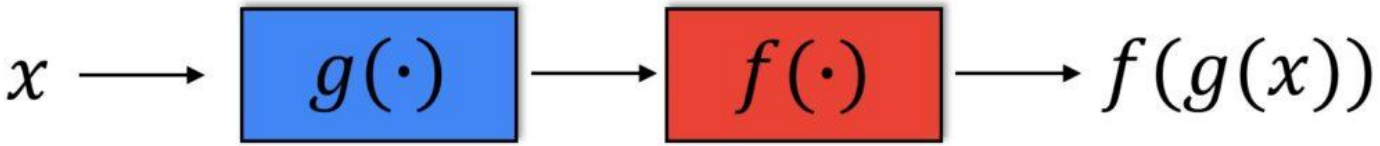
```
x = torch.tensor(3., requires_grad=True)
for epoch in range(3):
   y = 3*x**2 + 4*x + 2
   y.backward()
   print(x.grad)
   x.grad.zero_()
```

Gradient Descent & Backward Pass

Computational graph and Autograd with Pytorch

The Chain Rule

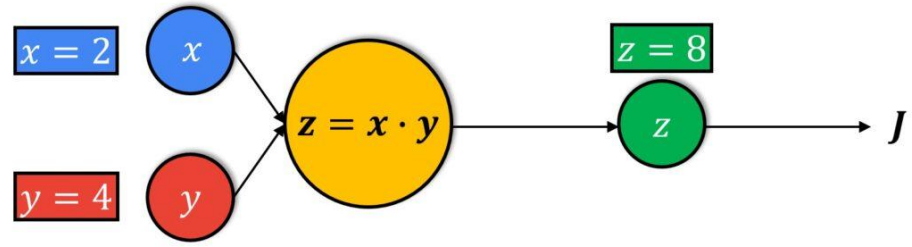
$$g = g(x) \qquad f = f(x)$$



$$\frac{dg}{dx} \qquad \frac{df}{dg}$$

$$\frac{d}{dx} f(g(x)) = \frac{dg}{dx}\frac{df}{dg}$$

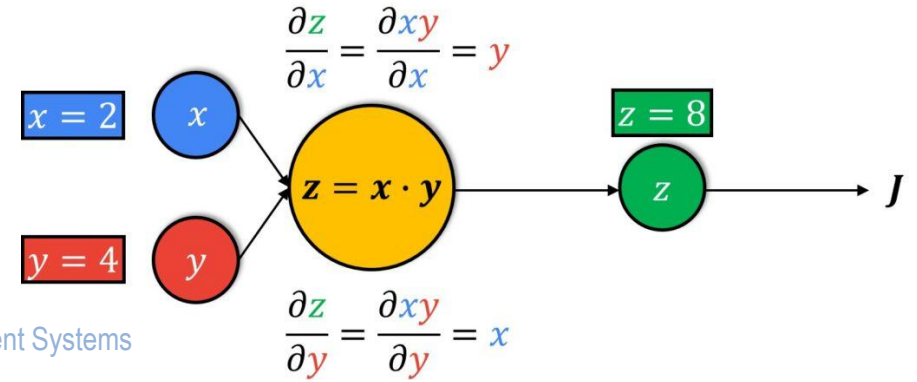
Forward propagation



$$z = x \cdot y$$

$$z = 2 \cdot 4 = 8$$

Backward propagation



Gradient Descent & Backward Pass

Computational graph and Autograd with Pytorch

4. Optimizing parameters with Autograd

Now, let's see how we can apply the chain rule in PyTorch.

```
import torch

x = torch.tensor([1.,2.,3.], requires_grad=True)
y = x * 2 + 3
z = y ** 2
print(z)
# To to calculate the gradients, we would need a scalar value of z. WHY?
out = z.mean()
out.backward()

print(out)
print(x.grad)
```

Now let's see what should we do if instead of calculating the gradients for a mean value of z, we want to calculate the gradient for each z individually?

```
v = torch.tensor([1.,1.,1.])

z.backward(v)

print(x.grad/len(x))
```

Simple Neural Network

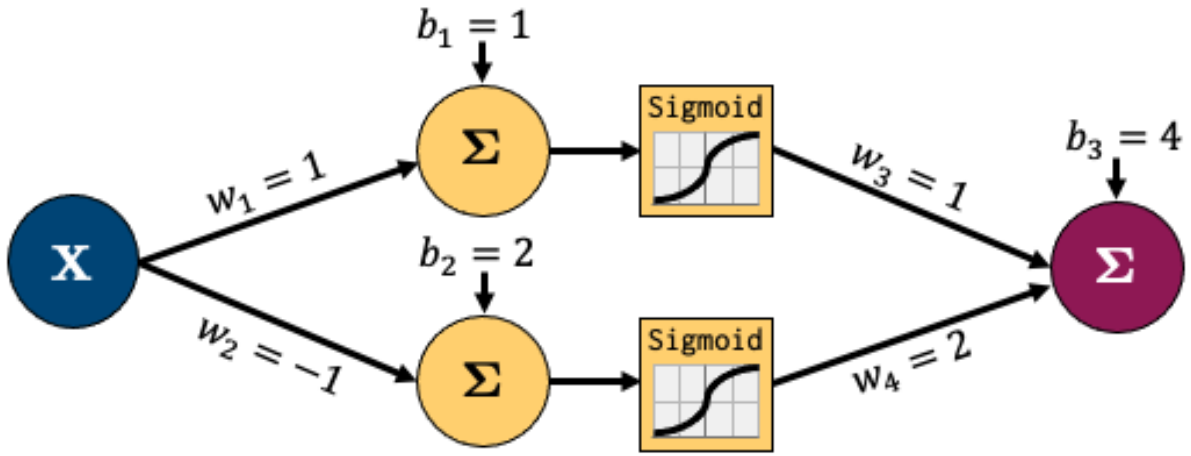
```
import numpy as np
import torch
from torch import nn
from torchvision import transforms, datasets, utils
from torch.utils.data import DataLoader, TensorDataset
```

1. Defining NN model

```
class network(torch.nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super().__init__()
        self.hidden = torch.nn.Linear(input_size, hidden_size)
        self.output = torch.nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = self.hidden(x)
        x = torch.sigmoid(x)
        x = self.output(x)
        return x
```

2. Manual Weight and Bias Initialization

```
model = network(1, 2, 1)  # make an instance of our network
# fix the weights manually as in the earlier figure
model.state_dict()['hidden.weight'][:] = torch.tensor([[1], [-1]])
model.state_dict()['hidden.bias'][:] = torch.tensor([1, 2])
model.state_dict()['output.weight'][:] = torch.tensor([[1, 2]])
model.state_dict()['output.bias'][:] = torch.tensor([-1])

x, y = torch.tensor([1.0]), torch.tensor([3.0])  # our x, y data

print(model.output.bias.grad)
```



3. Loss Function

```
criterion = torch.nn.MSELoss()
```

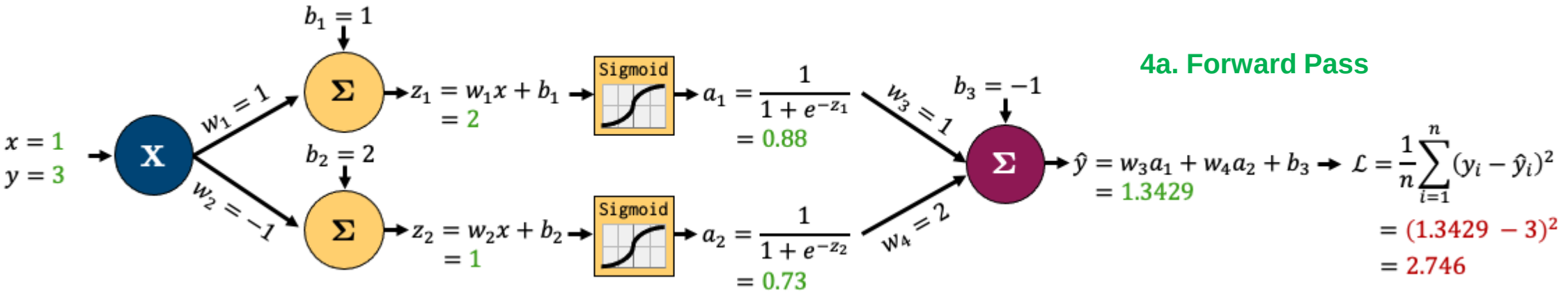
4. Forward and Backward Pass

```
loss = criterion(model(x), y)
loss.backward()  # calculate gradients!
```

5. Bias Grad

```
print(model.output.bias.grad)
```

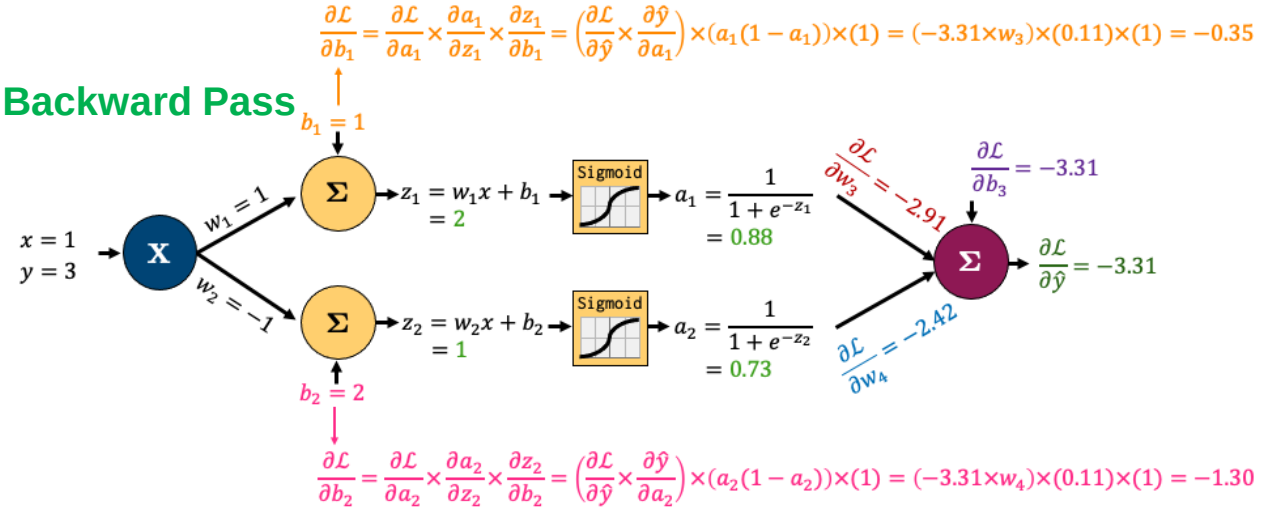
Simple Neural Network



4a. Forward Pass

```
print("Hidden Layer Gradients")
print("Bias:", model.hidden.bias.grad)
print("Weights:", model.hidden.weight.grad.squeeze())
print()
print("Output Layer Gradients")
print("Bias:", model.output.bias.grad)
print("Weights:", model.output.weight.grad.squeeze())

model.state_dict()
```

4b. Backward Pass



Output

```
OrderedDict([('hidden.weight',
              tensor([[ 1.],
                      [-1.]])),
             ('hidden.bias', tensor([1., 2.])),
             ('output.weight', tensor([[1., 2.]])),
             ('output.bias', tensor([-1.]))])
```

Simple Neural Network

6. Optimizer and Gradient Descent

```
optimizer = torch.optim.SGD(model.parameters(),
lr=0.1)
optimizer.step()

# Model weight and bias will be different now.
model.state_dict()
```

```
OrderedDict([('hidden.weight',
              tensor([[ 1.0348],
                      [-0.8697]])),
             ('hidden.bias', tensor([1.0348, 2.1303])),
             ('output.weight', tensor([[1.2919, 2.2423]])),
             ('output.bias', tensor([-0.6686]))])
```

7. Weight updating over Iterations

```
optimizer.zero_grad()  # Why?. Try without this command
# After 5 iterations
for _ in range(1, 6):
    loss = criterion(model(x), y)
    loss.backward()
    print(f"b3 gradient after call {_} of loss.backward():", model.hidden.bias.grad)
```

```
b3 gradient after call 1 of loss.backward(): tensor([-0.1991, -0.5976])
b3 gradient after call 2 of loss.backward(): tensor([-0.3983, -1.1953])
b3 gradient after call 3 of loss.backward(): tensor([-0.5974, -1.7929])
b3 gradient after call 4 of loss.backward(): tensor([-0.7966, -2.3906])
b3 gradient after call 5 of loss.backward(): tensor([-0.9957, -2.9882])
```

Logistic Regression

Logistic regression is a type of regression that predicts the probability of an event.

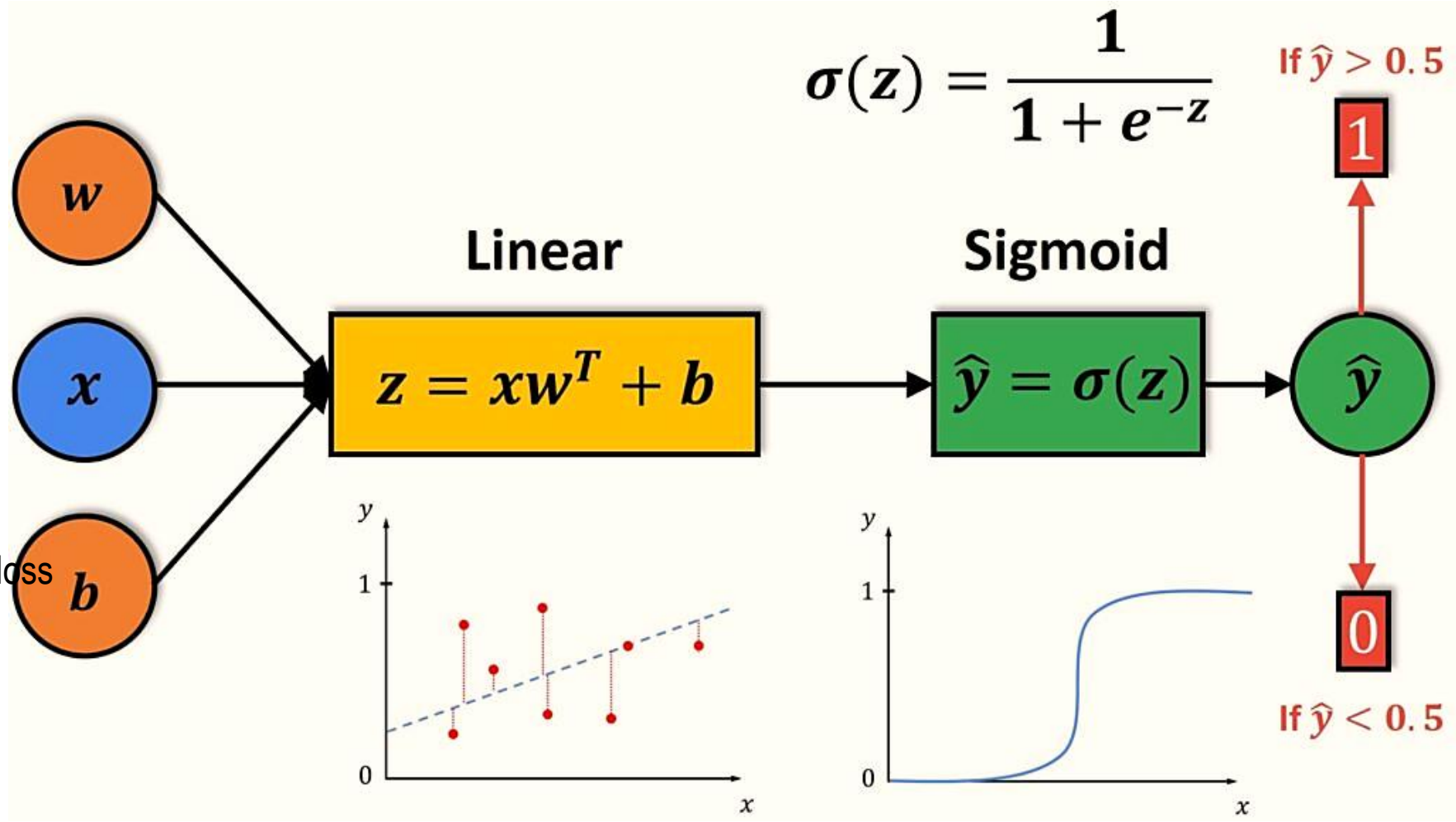
Steps for PyTorch Training

1) Data Preparation
2) Design model (input, output, forward pass with different layers)
3) Construct loss and optimizer
4) Training loop

   # Forward = compute prediction and loss

   # Backward = compute gradients

   # Update weights



https://datahacker.rs/005-pytorch-logistic-regression-in-pytorch/

Logistic Regression

```
import torch
import torch.nn as nn
import numpy as np
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
```

1) Data Preparation

```
bc = datasets.load_breast_cancer()
X, y = bc.data, bc.target

n_samples, n_features = X.shape

X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=1234)

# scale
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)

X_train = torch.from_numpy(X_train.astype(np.float32))
X_test = torch.from_numpy(X_test.astype(np.float32))
y_train = torch.from_numpy(y_train.astype(np.float32))
y_test = torch.from_numpy(y_test.astype(np.float32))

y_train = y_train.view(y_train.shape[0], 1)
y_test = y_test.view(y_test.shape[0], 1)
```

2) Model Definition

```
# Linear model f = wx + b , sigmoid at the end
class Model(nn.Module):
    def __init__(self, n_input_features):
        super(Model, self).__init__()
        self.linear = nn.Linear(n_input_features, 1)

    def forward(self, x):
        y_pred = torch.sigmoid(self.linear(x))
        return y_pred

model = Model(n_features)
```

3) Loss Function and Optimizer

```
num_epochs = 100
learning_rate = 0.01
criterion = nn.BCELoss()
optimizer = torch.optim.SGD(model.parameters(),
lr=learning_rate)
```

Logistic Regression

4) Training Loop

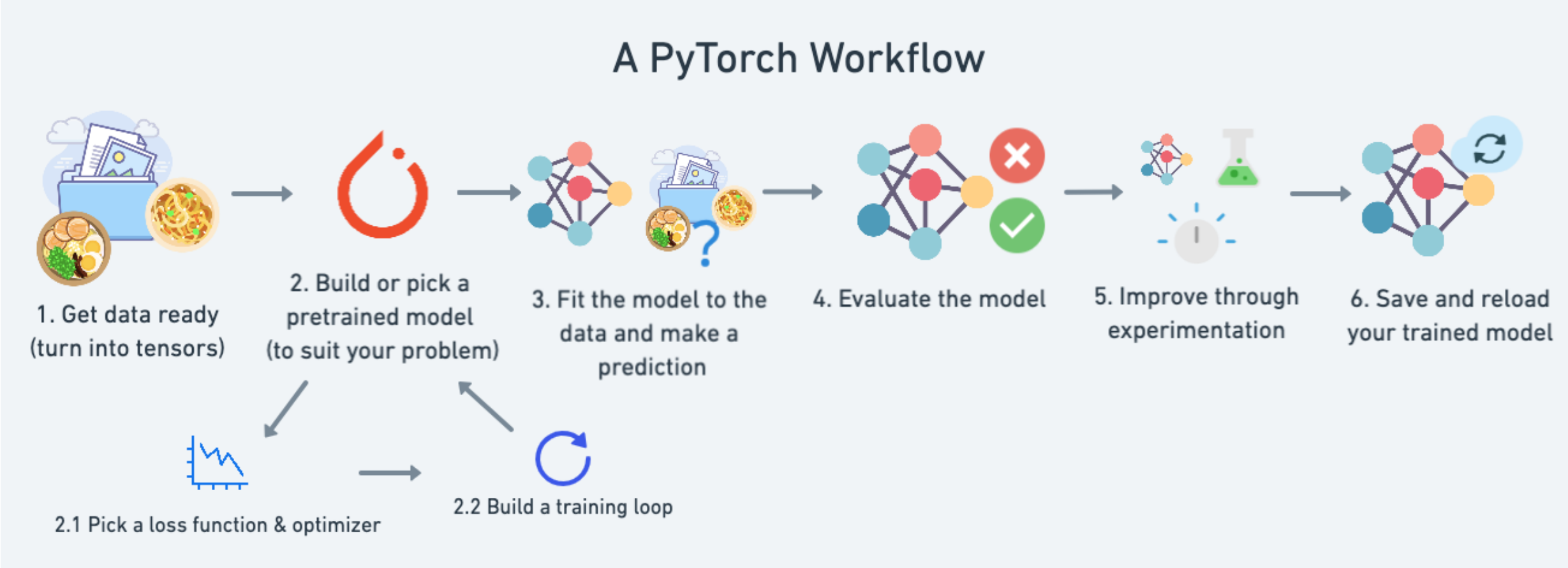
```
for epoch in range(num_epochs):
    # Forward pass and loss
    y_pred = model(X_train)
    loss = criterion(y_pred, y_train)

    # Backward pass and update
    loss.backward()
    optimizer.step()

    # zero grad before new step
    optimizer.zero_grad()

    if (epoch+1) % 10 == 0:
        print(f'epoch: {epoch+1}, loss = {loss.item():.4f}')
```

Testing and Performance Analysis

```
with torch.no_grad():
    y_predicted = model(X_test)
    y_predicted_cls = y_predicted.round()
    acc = y_predicted_cls.eq(y_test).sum() / float(y_test.shape[0])
    print(f'accuracy: {acc.item():.4f}')
```

PyTorch Workflow for DL

Various Class Definitions required for Building ANN/CNN Models



https://www.learnpytorch.io/

PyTorch Workflow for DL

Create DataLoader & Data-Transformer


```
import torch
from torch.utils.data import Dataset, DataLoader
import numpy as np

class CustomDataset(Dataset):
    def __init__(self, data, labels):
        """
        Args:
            data (numpy array or torch.Tensor): Input features.
            labels (numpy array or torch.Tensor): Corresponding labels.
        """
        self.data = torch.tensor(data, dtype=torch.float32)
        self.labels = torch.tensor(labels, dtype=torch.long)

    def __len__(self):
        """Returns the total number of samples."""
        return len(self.data)

    def __getitem__(self, index):
        """Retrieves a sample at the given index."""
        return self.data[index], self.labels[index]
```

PyTorch Workflow for DL

Create DataLoader & Data-Transformer

Feature	PyTorch DataLoader	PyTorch Transformers (Transforms)
Purpose	Efficiently loads and batches dataset	Applies data augmentation and preprocessing
Class Location	`torch.utils.data.DataLoader`	`torchvision.transforms` (for images)
Primary Function	Iterates through dataset with batching, shuffling, and parallel processing	Modifies dataset samples (e.g., resizing, normalization)
Usage	Used in training loops to load data efficiently	Applied to each sample before it is returned by the dataset
Works With	Any `Dataset` object	Image datasets (`torchvision`), tabular/text data
Key Parameters	`batch_size` , `shuffle` , `num_workers` , `drop_last`	`Resize` , `Normalize` , `RandomCrop` , `ToTensor` , etc.
Example Usage	`dataloader = DataLoader(dataset, batch_size=32, shuffle=True)`	`transform = transforms.Compose([transforms.Resize(256), transforms.ToTensor()])`
Supports Custom Datasets?	✅ Yes, via `torch.utils.data.Dataset`	✅ Yes, via `__getitem__` in custom datasets
Parallel Processing?	✅ Yes (`num_workers > 0` speeds up loading)	❌ No, applied within dataset class
Application in Deep Learning	Handles mini-batches, making training efficient	Prepares data for better model generalization

```
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt

# Device configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

```
# MNIST dataset
train_dataset = torchvision.datasets.MNIST(root='./data',
                                           train=True,
                                           transform=transforms.ToTensor(),
                                           download=True)

test_dataset = torchvision.datasets.MNIST(root='./data',
                                          train=False,
                                          transform=transforms.ToTensor())

# Data loader
train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                           batch_size=100,
                                           shuffle=True)

test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
                                          batch_size=batch_size,
                                          shuffle=False)
```

PyTorch Workflow for DL

Creating custom DataLoader

```
import torch
from torch.utils.data import Dataset, DataLoader
import torchvision.transforms as transforms
import torchvision.datasets as datasets
import matplotlib.pyplot as plt
import numpy as np
```

```
# Device configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

- **ToTensor():** Converts PIL images to PyTorch tensors.
- **Normalize()**: Scales pixel values to range [−1,1][−1,1] for better convergence.

```
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])
```

```
# Load training dataset with custom dataset class
train_dataset = CustomMNISTDataset(train=True, transform=transform)
train_loader = DataLoader(train_dataset, batch_size=8, shuffle=True)
```

```
class CustomMNISTDataset(Dataset):
    def __init__(self, train=True, transform=None):
        self.mnist_data = datasets.MNIST(root='./data', train=train, download=True,
                                         transform=transform)

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

    def __getitem__(self, idx):
        image, label = self.mnist_data[idx]
        return image, label
```

```
# Get a batch of data
data_iter = iter(train_loader)
images, labels = next(data_iter)

# Function to display images
def imshow(img):
    img = img / 2 + 0.5  # Unnormalize
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)), cmap="gray")
    plt.axis("off")
    plt.show()

# Display batch of images
imshow(torchvision.utils.make_grid(images))
print("Labels:", labels.numpy())
```

PyTorch Workflow for DL

DataLoader – Splitting Dataset

Split	Purpose	Amount of total data	How often is it used?
Training set	The model learns from this data (like the course materials you study during the semester).	~60-80%	Always
Validation set	The model gets tuned on this data (like the practice exam you take before the final exam).	~10-20%	Often but not always
Testing set	The model gets evaluated on this data to test what it has learned (like the final exam you take at the end of the semester).	~10-20%	Always

1. Training Set (80%) 🏋️

- Used to **train** the model by adjusting weights through backpropagation.
- The model learns patterns and representations from this dataset.

2. Validation Set (10%) 🧪

- Used to **fine-tune hyperparameters** (e.g., learning rate, batch size, dropout rate).
- Helps in preventing **overfitting**, ensuring the model does not memorize the training data.
- **Early stopping** is often based on validation loss improvement.

3. Test Set (10%) 📊

- Used **only after** the model is finalized to evaluate its performance.
- Measures how well the model generalizes to **completely unseen data**.
- Ensures unbiased model assessment before deployment.

PyTorch Workflow for DL

DataLoader – Splitting Dataset

```python
from torch.utils.data import random_split

# Define dataset
full_dataset = CustomMNISTDataset(train=True, transform=transform)

total_size = len(full_dataset)
train_size = int(0.8 * total_size)
val_size = int(0.1 * total_size)
test_size = total_size - train_size - val_size  # Ensures no data loss

# Split dataset
train_dataset, val_dataset, test_dataset = random_split(full_dataset, [train_size, val_size, test_size])

# Create DataLoaders
train_loader = DataLoader(train_dataset, batch_size=100, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=100, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=100, shuffle=False)
```

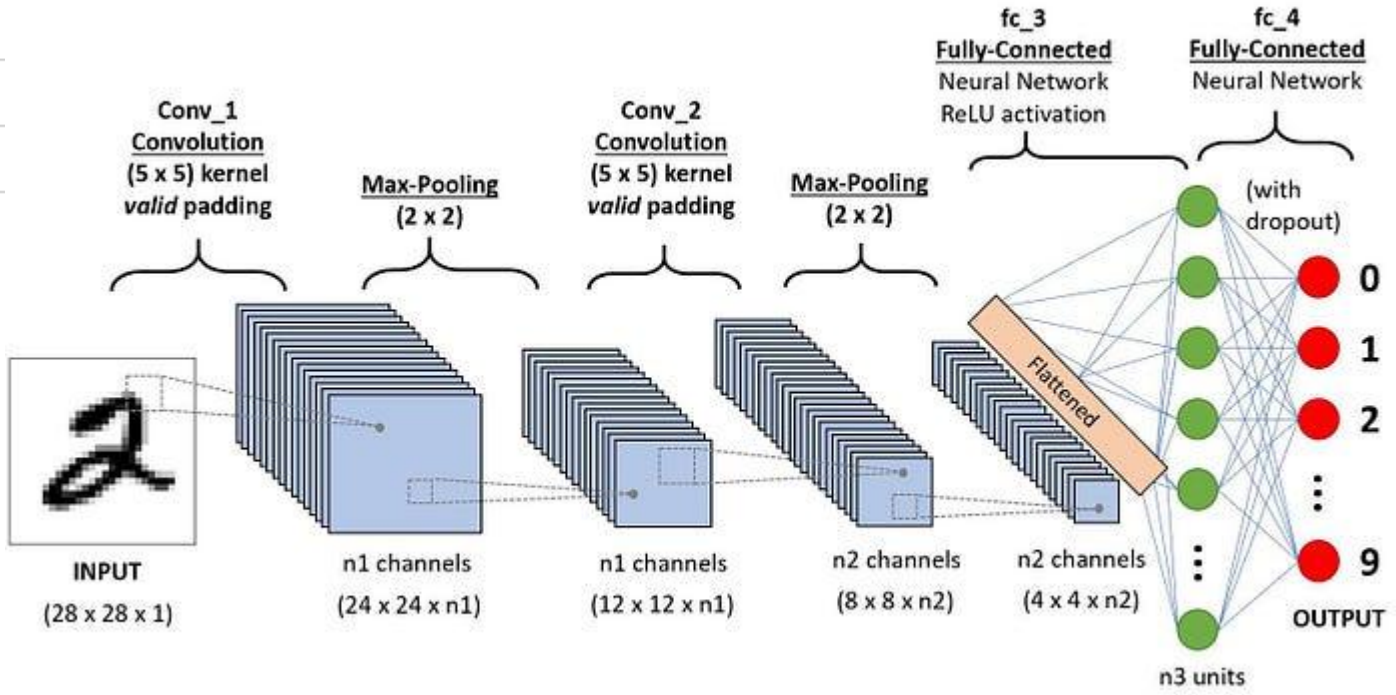
PyTorch Workflow for DL

Model Definition

- **Architecture Summary**

Layer	Details
Input	$28 \times 28 \times 1$
Conv2D (1st layer)	5×5 filter, 32 filters, ReLU, MaxPooling (2×2)
Conv2D (2nd layer)	5×5 filter, 64 filters, ReLU, MaxPooling (2×2)
Flatten 1	Converts feature maps into a 1D vector
Fully Connected Layer (fc1)	256 neurons, ReLU Activation
Dropout	Dropout probability 0.5
Fully Connected Layer (fc2)	10-class output

```python
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import torch.nn.functional as F

from torch.utils.data import DataLoader
```



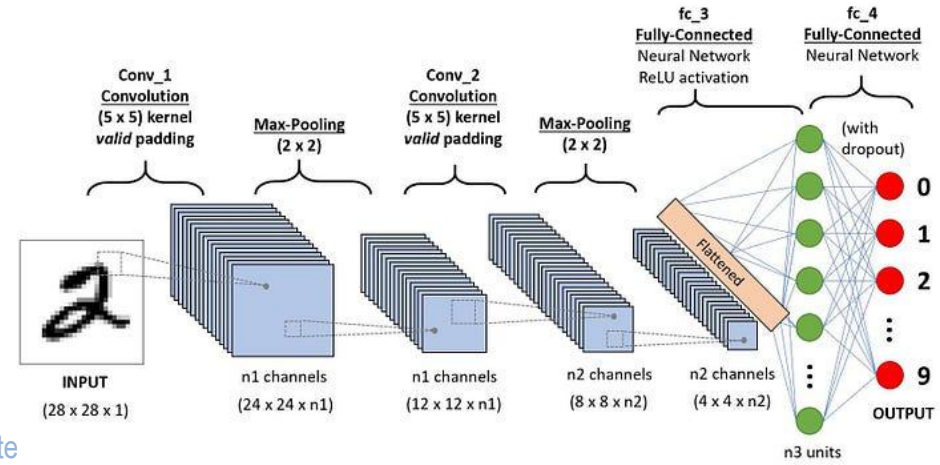
PyTorch Workflow for DL

Model Definition

```python
class CustomCNN(nn.Module):
    def __init__(self, num_classes=10):
        super(CustomCNN, self).__init__()

        # First Conv Layer: 1 input channel, 32 filters, kernel size 5x5
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=32, kernel_size=5)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)

        # Second Conv Layer: 32 input channels, 64 filters, kernel size 5x5
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=5)

        # Fully Connected Layers
        self.fc1 = nn.Linear(64 * 4 * 4, 256)  # Adjusted for input size after convolutions
        self.fc2 = nn.Linear(256, num_classes)

        # Dropout
        self.dropout = nn.Dropout(0.5)
    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))  # Conv1 -> ReLU -> MaxPool
        x = self.pool(F.relu(self.conv2(x)))  # Conv2 -> ReLU -> MaxPool

        x = torch.flatten(x, start_dim=1)  # Flattening layer (Flatten1)
        x = F.relu(self.fc1(x))  # Fully connected layer with ReLU activation
        x = self.dropout(x)  # Dropout layer

        x = self.fc2(x)  # Final output layer
        return x
```

The base class for all neural network modules, all the building blocks for neural networks are subclasses. If you're building a neural network in PyTorch, your models should subclass `nn.Module`

All `nn.Module` subclasses require a `forward()` method, this defines the computation that will take place on the data passed to the particular `nn.Module`.



PyTorch Workflow for DL

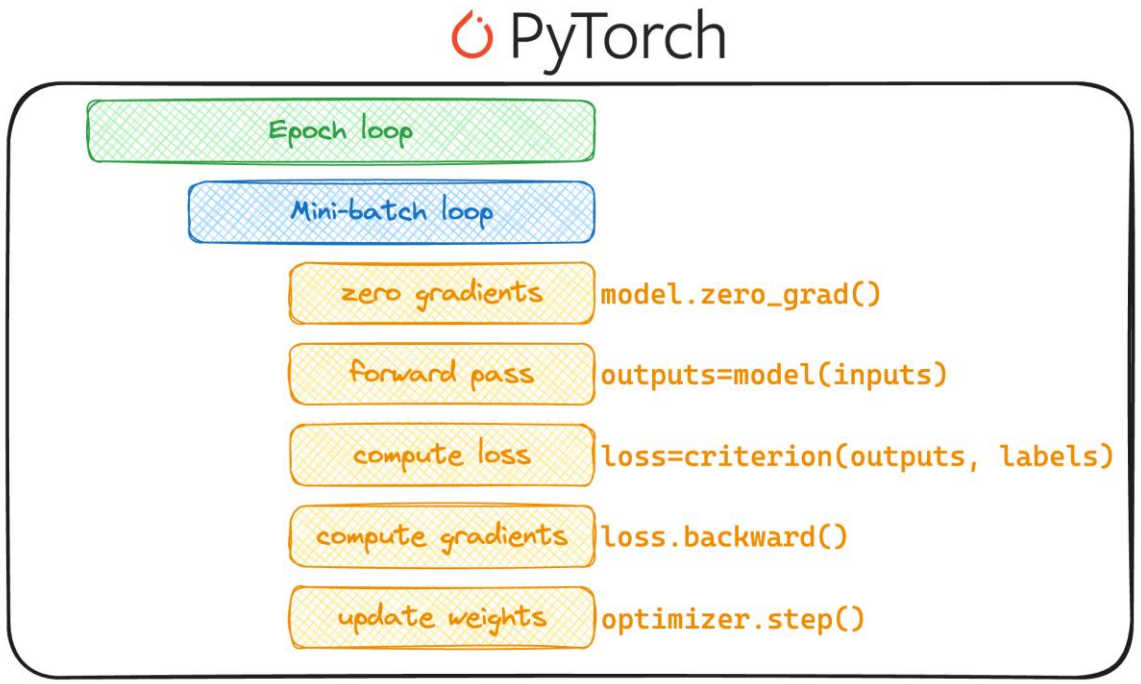
Training Loop

Step name	What does it do?	Code example
Forward pass	The model goes through all of the training data once, performing its **`forward()`** function calculations.	`model(x_train)`
Calculate the loss	The model's outputs (predictions) are compared to the ground truth and evaluated to see how wrong they are.	`loss = loss_fn(y_pred, y_train)`
Zero gradients	The optimizers gradients are set to zero (they are accumulated by default) so they can be recalculated for the specific training step.	`optimizer.zero_grad()`
Perform backpropagation on the loss	Computes the gradient of the loss with respect for every model parameter to be updated (each parameter with **`requires_grad=True`**). This is known as backpropagation, hence "backwards".	`loss.backward()`
Update the optimizer (gradient descent)	Update the parameters with **`requires_grad=True`** with respect to the loss gradients in order to improve them.	`optimizer.step()`

PyTorch Workflow for DL

Training Loop

```
# Initialize model
model = CustomCNN()

# Define optimizer
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Define loss function
criterion = nn.CrossEntropyLoss()
```



Training loop

```
for epoch in range(5):
    model.train()
    running_loss = 0.0

    for data in trainloader:

        inputs, labels = data
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()

    accuracy = evaluate(model)

    print(f"""Epoch {epoch + 1},
              Loss: {running_loss / len(trainloader)},
              Accuracy: {accuracy * 100:.2f}%
          """)
```

PyTorch Workflow for DL

Model Evaluation

Evaluate model

```
def evaluate(model):

    model.eval()
    correct, total = 0, 0

    with torch.no_grad():
        for data in testloader:
            inputs, labels = data
            outputs = model(inputs)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    return correct / total
```

CNN Complete Model

Step 1) Import required packages and libraries

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import torch.nn.functional as F

from torch.utils.data import DataLoader
```

Step 2) Load the dataset

```
# 1) Data Transformer
transform = transforms.Compose([transforms.ToTensor(),
                                transforms.Normalize((0.5,), (0.5,))])

# 2) Create Train Dataset
trainset = torchvision.datasets.MNIST(root='./data', train=True,
                                      download=True, transform=transform)
trainloader = DataLoader(trainset, batch_size=64, shuffle=True)

# 3) Create Test Dataset
testset = torchvision.datasets.MNIST(root='./data', train=False,
                                     download=True, transform=transform)
testloader = DataLoader(testset, batch_size=64, shuffle=False)
```

Step 3) Define the PyTorch Model

Architecture

Forward pass

```
class PyTorchNet(nn.Module):

    def __init__(self):
        super(PyTorchNet, self).__init__()
        self.fc1 = nn.Linear(28*28, 512)
        self.fc2 = nn.Linear(512, 1024)
        self.fc3 = nn.Linear(1024, 128)
        self.fc4 = nn.Linear(128, 10)

    def forward(self, x):
        x = x.view(-1, 28*28)
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = torch.relu(self.fc3(x))
        x = self.fc4(x)
        return x
```

Step 4) Initialize the model and define the loss function and optimizer

```
# Initialize model
model = PyTorchNet()

# Define optimizer
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Define loss function
criterion = nn.CrossEntropyLoss()
```

CNN Complete Model


```python
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import torch.nn.functional as F

from torch.utils.data import DataLoader
```

```python
class PyTorchNet(nn.Module):

    def __init__(self):
        super(PyTorchNet, self).__init__()
        self.fc1 = nn.Linear(28*28, 512)
        self.fc2 = nn.Linear(512, 1024)
        self.fc3 = nn.Linear(1024, 128)
        self.fc4 = nn.Linear(128, 10)

    def forward(self, x):
        x = x.view(-1, 28*28)
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = torch.relu(self.fc3(x))
        x = self.fc4(x)
        return x
```

```python
# 1) Data Transf
transform = tran

# 2) Create Trai
trainset = torch

trainloader = Da

# 3) Create Test
testset = torchv

testloader = Dat
```

```python
# Initialize
model = PyTor

# Define opti
optimizer = o

# Define loss
criterion = n
```

Training loop

```python
for epoch in range(5):
    model.train()
    running_loss = 0.0

    for data in trainloader:

        inputs, labels = data
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()

    accuracy = evaluate(model)

    print(f"""Epoch {epoch + 1},
              Loss: {running_loss / len(trainloader)},
              Accuracy: {accuracy * 100:.2f}%
          """)
```

Step 5) Train the model

CNN Complete Model

Indian Institute of Technology Kharagpur
Department of Electrical Engineering | Signal Processing and Machine Learning

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms
import torch.nn.functional as

from torch.utils.data import
```

```
class PyTorchNet(nn.Module):

    def __init__(self):
        super(PyTorchNet, se
        self.fc1 = nn.Linear
        self.fc2 = nn.Linear
        self.fc3 = nn.Linear
        self.fc4 = nn.Linear

    def forward(self, x):
        x = x.view(-1, 28*28
        x = torch.relu(self.
        x = torch.relu(self.
        x = torch.relu(self.
        x = self.fc4(x)
        return x
```

```
# 1) Data Transformer
transform = transforms.Compose([transforms.ToTensor(),
                                transforms.Normalize((0.5,), (0.5,))])
```

Evaluate model

```
def evaluate(model):

    model.eval()
    correct, total = 0, 0

    with torch.no_grad():
        for data in testloader:
            inputs, labels = data
            outputs = model(inputs)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    return correct / total
```

Training loop

Step 6) Define the evaluation method

```
, labels)
n()

print(f"""Epoch {epoch + 1},
          Loss: {running_loss / len(trainloader)},
          Accuracy: {accuracy * 100:.2f}%
      """)
```

PyTorch Workflow for DL

Model Inference

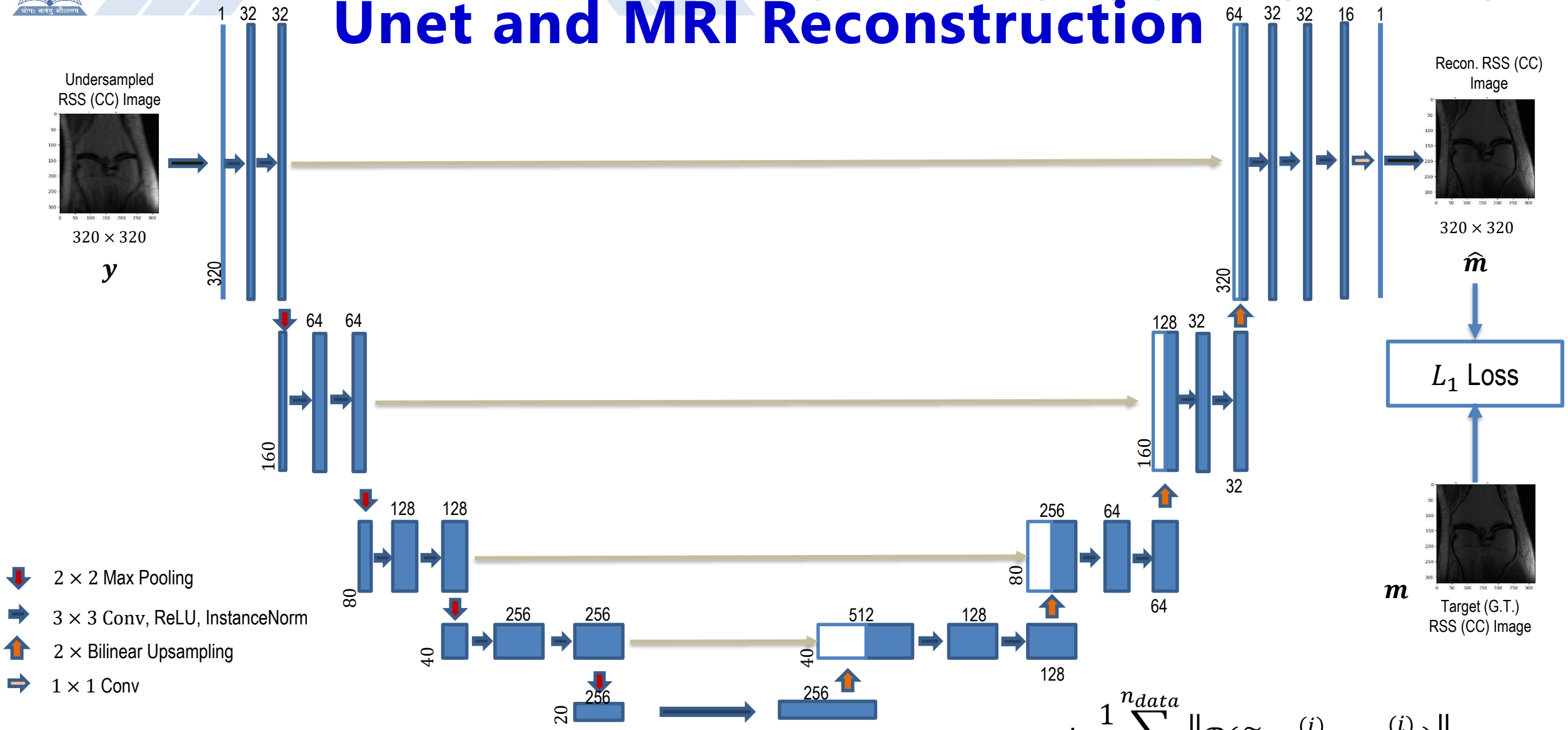
PyTorch method	What does it do?
torch.save	Saves Model, tensors and various other Python objects like dictionaries can be saved using torch.save.
torch.load	Loads Python object files (like models, tensors or dictionaries) into memory. You can also set which device to load the object to (CPU, GPU etc).
torch.nn.Module.load_state_dict	Loads a model's parameter dictionary (model.state_dict()) using a saved state_dict() object.

```python
# Save the trained model
torch.save(model.state_dict(), 'model.pth')


# Load the model
model_loaded = PyTorchNet()
model_loaded.load_state_dict(torch.load('model.pth'))
model_loaded.eval()
```

```python
# Inference: Example of prediction
with torch.no_grad():
    for data in testloader:
        inputs, labels = data
        outputs = model_loaded(inputs)
        _, predicted = torch.max(outputs, 1)
        print(f"Predicted: {predicted}, Ground Truth: {labels}")
        break  # Just use one batch for demonstration
```

```python
# Evaluate the loaded model
accuracy = evaluate(model_loaded)
print(f"Loaded model accuracy: {accuracy * 100:.2f}%")
```

Unet and MRI Reconstruction



$$\boldsymbol{L_1}\ \textbf{Loss:}\ \min_{\theta} \frac{1}{2} \sum_{i=0}^{n_{data}} \left\| \mathcal{B}(\tilde{m}_{rss}{}^{(i)} - m_{rss}^{(i)}) \right\|_1$$

Model Complexity Analysis

1. torchsummary

```
pip install torchsummary
```

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
from torchsummary import summary

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.conv2_drop = nn.Dropout2d()
        self.fc1 = nn.Linear(320, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
        x = x.view(-1, 320)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        x = self.fc2(x)
        return F.log_softmax(x, dim=1)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = Net().to(device)

summary(model, (1, 28, 28))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1          [-1, 10, 24, 24]             260
            Conv2d-2          [-1, 20, 8, 8]             5,020
         Dropout2d-3          [-1, 20, 8, 8]                 0
            Linear-4                [-1, 50]            16,050
            Linear-5                [-1, 10]               510
================================================================
Total params: 21,840
Trainable params: 21,840
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.06
Params size (MB): 0.08
Estimated Total Size (MB): 0.15
----------------------------------------------------------------
```

Model Complexity Analysis

2. GFLOP and GMAC

```
pip install ptflops
```

GFLOP: **Giga Floating-Point Operations per Second**
GMAC: **Giga Multiply-Add Operations per Second**

1 GFLOP = 2* GMAC

$$GMAC = \frac{(Number\ of\ Multiply - Add\ Operations)}{10^9}$$

$$GFLOP = \frac{(Number\ of\ Floating - Point\ Operations)}{(Elapsed\ Time\ in\ Seconds)} \over (10^9)$$

```python
import torchvision.models as models
import torch
from ptflops import get_model_complexity_info
import re

#Model thats already available
net = models.densenet161()
macs, params = get_model_complexity_info(net, (3, 224, 224), as_strings=True,
print_per_layer_stat=True, verbose=True)
# Extract the numerical value
flops = eval(re.findall(r'([\d.]+)', macs)[0])*2
# Extract the unit
flops_unit = re.findall(r'([A-Za-z]+)', macs)[0][0]

print('Computational complexity: {:<8}'.format(macs))
print('Computational complexity: {} {}Flops'.format(flops, flops_unit))
print('Number of parameters: {:<8}'.format(params))
```

Computational complexity: 7.82 GMac
Computational complexity: 15.64 GFlops
Number of parameters: 28.68 M

https://medium.com/@ajithkumarv/how-to-calculate-gmac-and-gflops-with-python-62004a753e3b

Model Complexity Analysis

https://pytorch.org/blog/zeus/

3. Deep Learning Energy Measurement

```
pip install zeus-ml==0.7.0
```

```python
from zeus.monitor import ZeusMonitor

# All four GPUs are measured simultaneously.
monitor = ZeusMonitor(gpu_indices=[0,1,2,3])

# Measure total time and energy within the window.
monitor.begin_window("training")

for e in range(100):
    # Measurement windows can arbitrarily be overlapped.
    monitor.begin_window("epoch")
    for x, y in train_dataloader:
        y_hat = model(x)
        loss = criterion(y, y_hat)
        loss.backward()
        optim.step()
    measurement = monitor.end_window("epoch")
    print(f"Epoch {e}: {measurement.time} s, {measurement.total_energy} J")

measurement = monitor.end_window("training")
print(f"Entire training: {measurement.time} s, {measurement.total_energy} J")
```

4. CodeCarbon

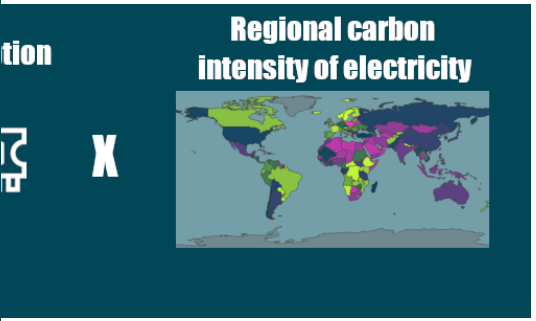
```
pip install co
```

Estimate ar

lyze their impact.

Estimates your
(GPU + CPU +
intensity of the r

```
from codecarbon import

@track_emissions
def training_loop():
    # Compute intensive
```



com/mlco2/codecarbon?tab=readme-ov-file

Thank you for your time

and attention …

Thanks to the organizers of National Symposium – 2025

Questions?

A Special Thanks to my Mentor:
Dr. Debdoot Sheet
Associate Professor, IIT Kgp.

https://www.linkedin.com/in/susant-panigrahi-b79b1396/

https://scholar.google.co.in/citations?user=1MjsFuUAAAAJ&hl=en

https://github.com/susant146?tab=repositories

It's all happening here:

- Kharagpur Learning, Imaging & Visualization (**KLIV**) research group: https://iitkliv.github.io/