PyTorch:
Model Training

Steps in Model Training

Training a model with PyTorch involves several key steps

a) Data preparation and loading and processing
b) Model Definition
c) Loss Function and Optimizer Definition
d) Training Loop
e) Evaluation (per epoch or periodically)
f) Saving (and Loading the Model)

a) Data Preparation, Loading and Processing

- Load and preprocess your dataset often involves transformations such as resizing, normalization
- Split the data into training, validation (optional), and test sets.
- Create Dataset and DataLoader objects to efficiently load and batch the data during training.
  - Dataset retrieves the 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 the above complexity for us in an easy API.

Example: MNIST Image Dataset

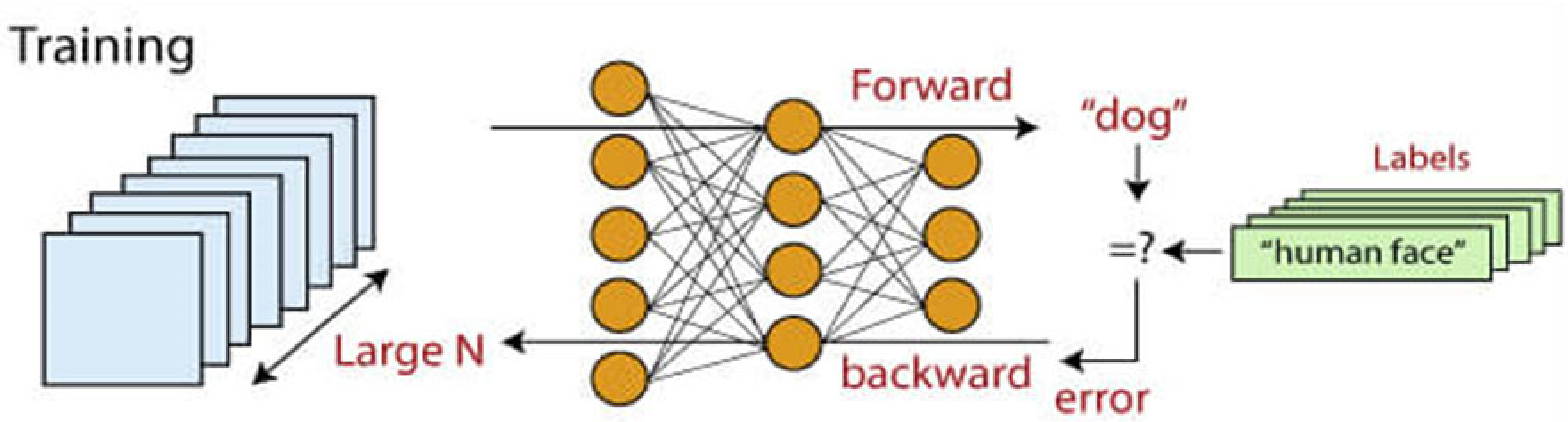
MNIST dataset for digit recognition

- **Modified National Institute of Standards and Technology** dataset
- a large database of handwritten digits
- grayscale images of handwritten single digits between 0 and 9.
- each of size 28x28.
- 60,000 training images and 10,000 testing images

(Wikipedia - Sample images from MNIST test dataset)

b) Model Definition

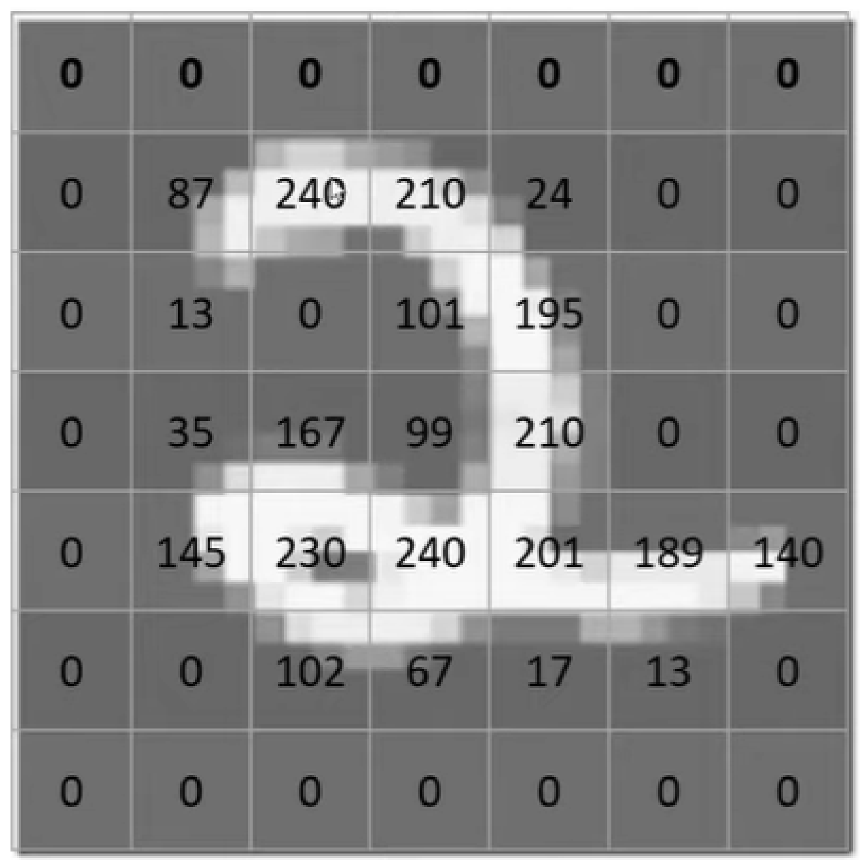
- Define the neural network architecture by inheriting from torch.nn.Module.
- Implement the __init__ method to define layers and parameters.
- Implement the forward method to specify how data flows through the network.



Flattening the Data

Consider the following handwritten image

- greyscale (0 to 255) with 28 x 28 pixel
- but assume 7x7 size



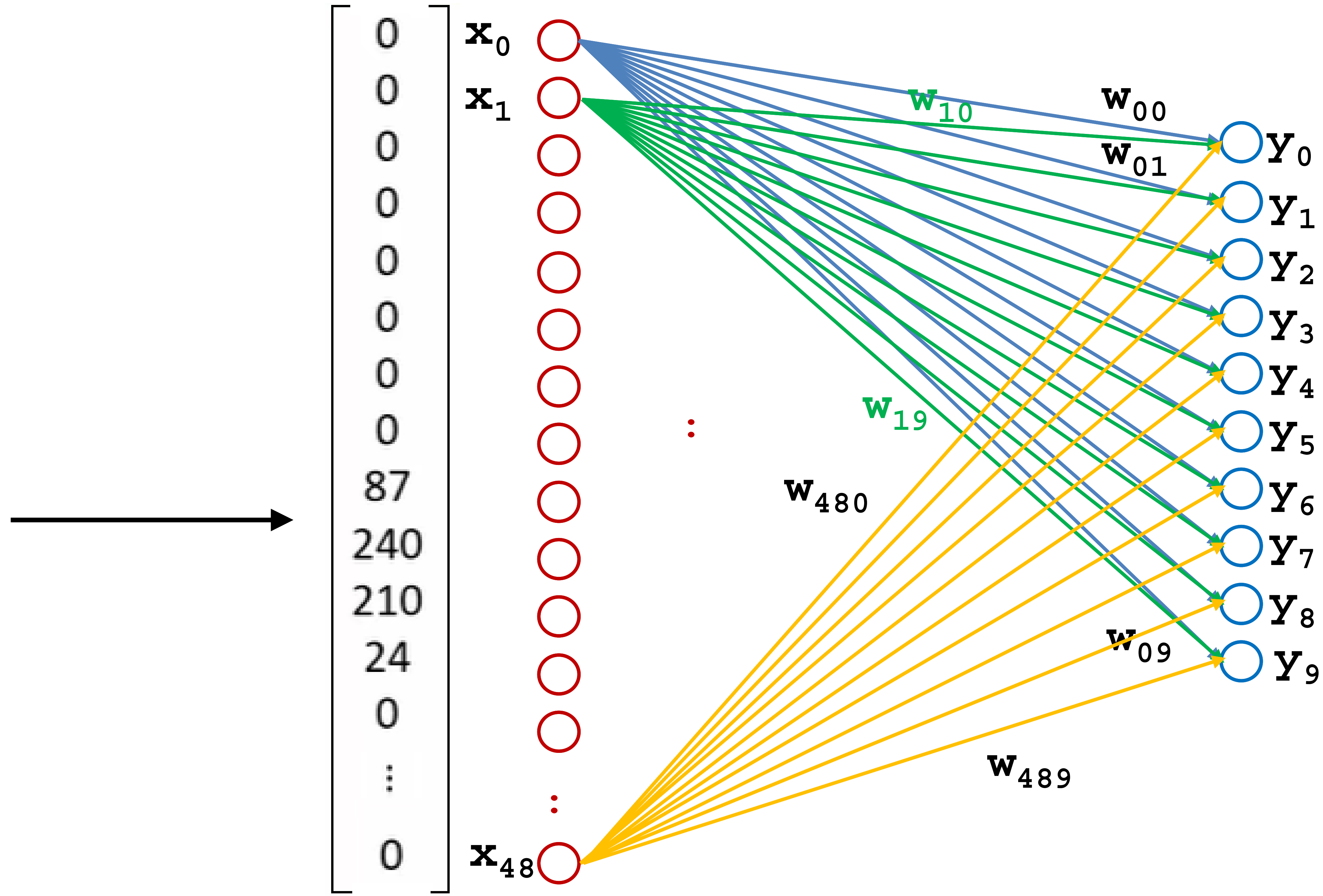
0	0	0	0	0	0	0
0	87	240	210	24	0	0
0	13	0	101	195	0	0
0	35	167	99	210	0	0
0	145	230	240	201	189	140
0	0	102	67	17	13	0
0	0	0	0	0	0	0

(Source: Youtube – codebasics)

Flattening the Data (cont.)

Flatten the 7x7 2D array to
1D array of size = 1x49

0	0	0	0	0	0	0
0	87	240	210	24	0	0
0	13	0	101	195	0	0
0	35	167	99	210	0	0
0	145	230	240	201	189	140
0	0	102	67	17	13	0
0	0	0	0	0	0	0



(Source: Youtube – codebasics)

c) Loss Function and Optimizer Definition

- Choose an appropriate loss function to quantify the difference between predictions and true labels
  - e.g., nn.MSELoss for regression

    nn.CrossEntropyLoss for classification
- Select an optimizer to update model parameters based on gradients
  - e.g., torch.optim.SGD

    torch.optim.Adam

    with Learning rate, Momentum

d) Training Loop (per Epoch)

- Set model to training mode: model.train()
- Iterate through batches:

  For each batch from the DataLoader:

  - Forward Pass: Pass the input data through the model to get prediction:
    outputs = model(inputs).
  - Calculate Loss: Compute the loss using the predictions and true labels:
    loss = loss_fn(outputs, labels).
  - Zero Gradients: Clear previously computed gradients from the optimizers
    optimizer.zero_grad().
  - Backpropagation: Compute gradients of the loss with respect to model
    parameters: loss.backward().
  - Optimizer Step: Update model parameters using the calculated gradient:
    optimizer.step().

e) Evaluation

- Set model to evaluation mode:
  model.eval()
- Disable gradient calculation:
  Use torch.no_grad() to save memory and speed up computation during evaluation.
- Evaluate the model's performance on the validation set (per epoch) or test set (after completion) using metrics like accuracy, confusion matrix, precision, recall, etc

f) Saving and Loading Model

- Save the trained model (for inference)
  i. save the whole architecture:

     torch.save(model, 'full_model.pth’)

  ii. save only necessary trained model's learned parameters (i.e. weights and biases) in trained model's state dictionary (against each layer)

     torch.save(model.state_dict(), ‘model.pth’)

- Load a saved model for inference
  i. torch.load(‘full_model.pth’)
  ii. load_state_dict(torch.load’model.pth’))
     - to a instantiated model