CASE SCHOOL OF ENGINEERING
CASE WESTERN RESERVE UNIVERSITY

CSDS 451: Designing High Performant Systems for AI

Lecture 12

10/7/2025

Sanmukh Kuppannagari

sanmukh.kuppannagari@case.edu

https://sanmukh.research.st/

Case Western Reserve University

Outline

- Pytorch Basics

Announcements

- Midterm this Thursday - 10/9
  - Will cover topics till the previous lecture

- I will upload WA1 and WA2 solutions by tonight

Outline

- Pytorch Basics

Training

$$E(W):\ \min_{W} \sum (y_i\ - F(x_i : W))\text{^}2$$

$$W_{t+1} \leftarrow W_t - \alpha \frac{\delta E(W)}{\delta W}$$

Iteratively

$$\frac{\delta E(W)}{\delta W} = 2 \sum_i \underbrace{\left(y_i\ - F(x_i : W)\right)}_{\text{Error}} \times \frac{\delta F(x_i{:}W)}{\delta W}$$

For all samples, in each iteration

Training

L layers in CNN

Weights (Filters/Kernels)

$$W_1, W_2, W_3, \dots, W_L$$

$$O_1, O_2, O_3, \dots, O_L$$

Output of layer

$$2 * Error * \frac{\delta F(x_i : W_{1:L})}{\delta W_1}, \frac{\delta F(x_i : W_{1:L})}{\delta W_2}, \dots, \frac{\delta F(x_i : W_{1:L})}{\delta W_L}$$

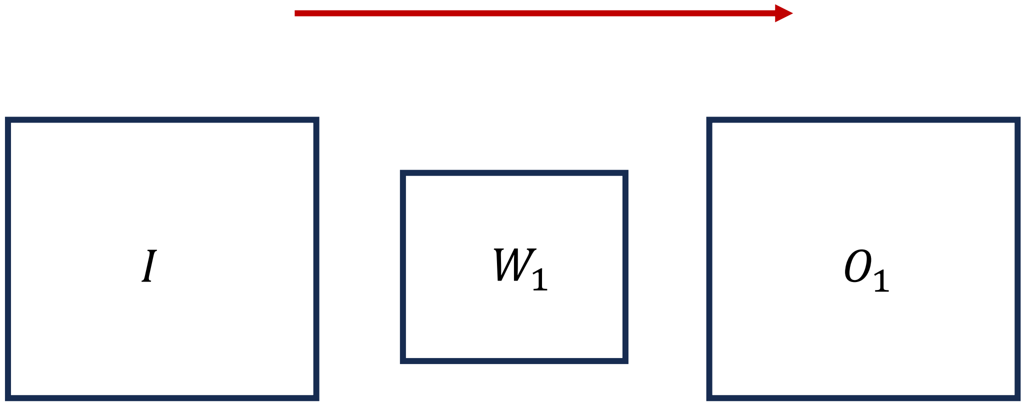
Compute these for all the samples

CNN Training: Forward Propagation

I

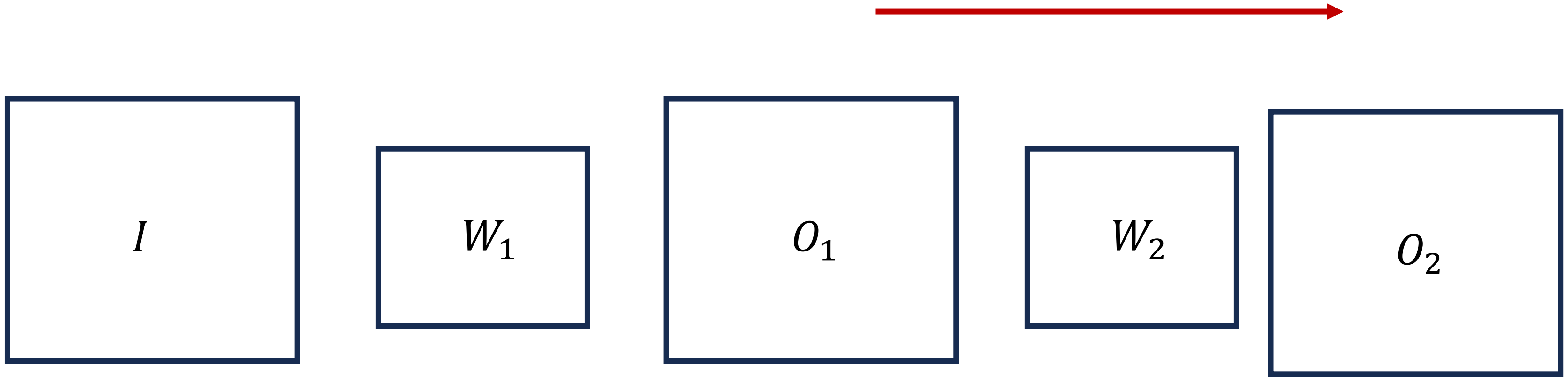
For a single image

CNN Training: Forward Propagation



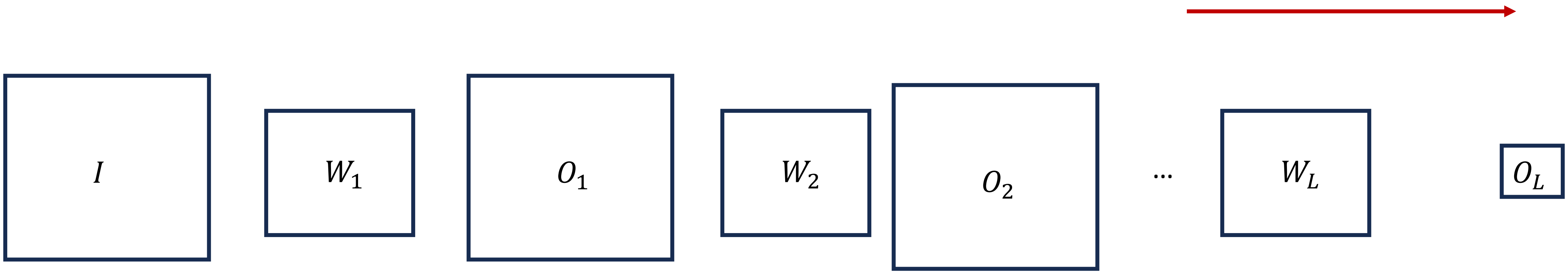
For a single image

CNN Training: Forward Propagation



For a single image

CNN Training: Forward Propagation



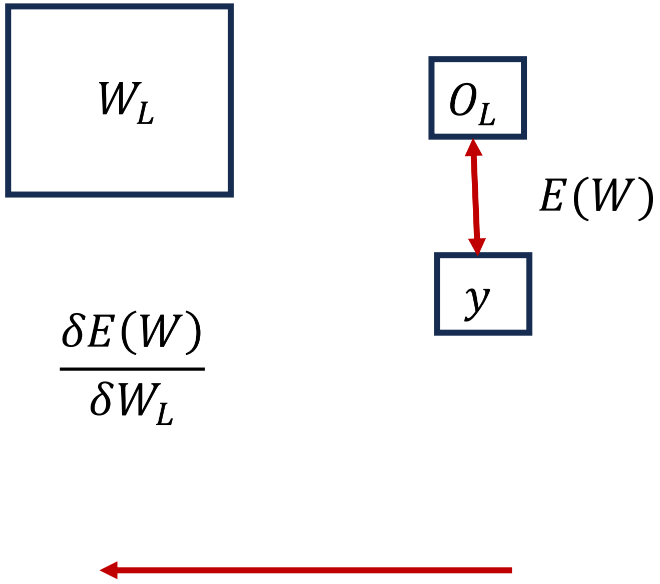
For a single image

CNN Training: Error Calculation

$$\begin{array}{c} \boxed{O_L} \\ \updownarrow \; E(W) \\ \boxed{y} \end{array}$$

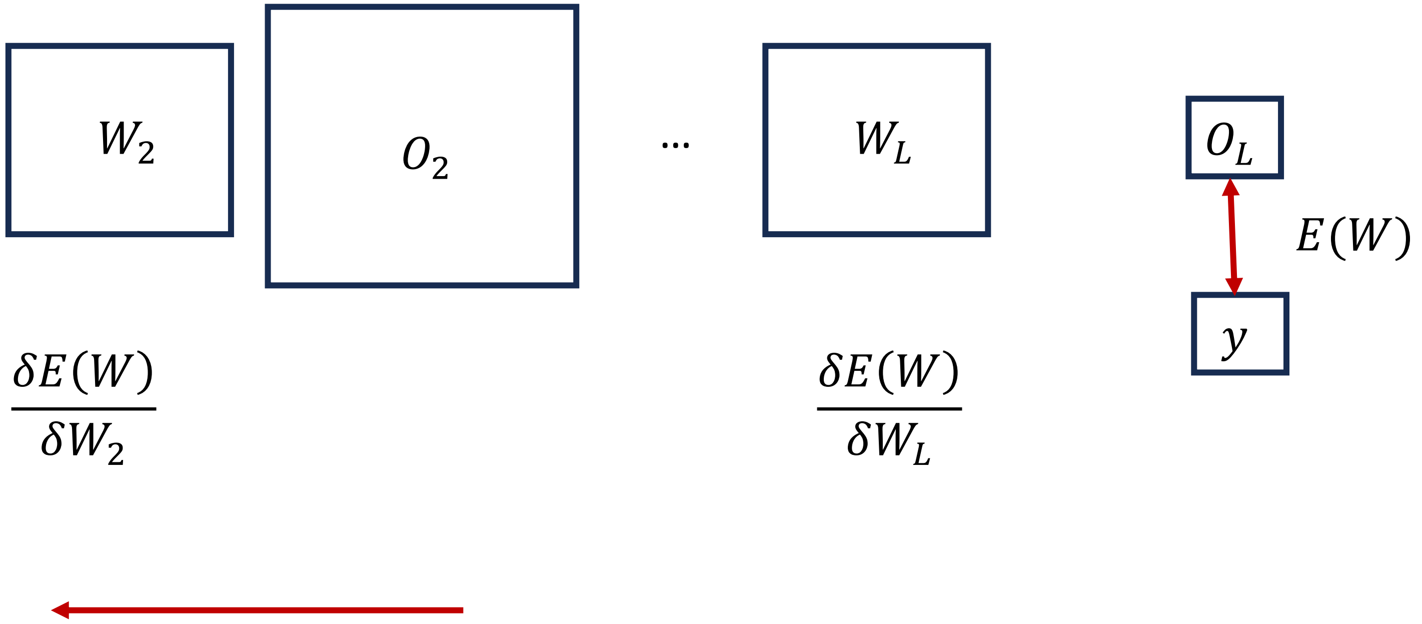
For a single image

CNN Training: Backward Propagation



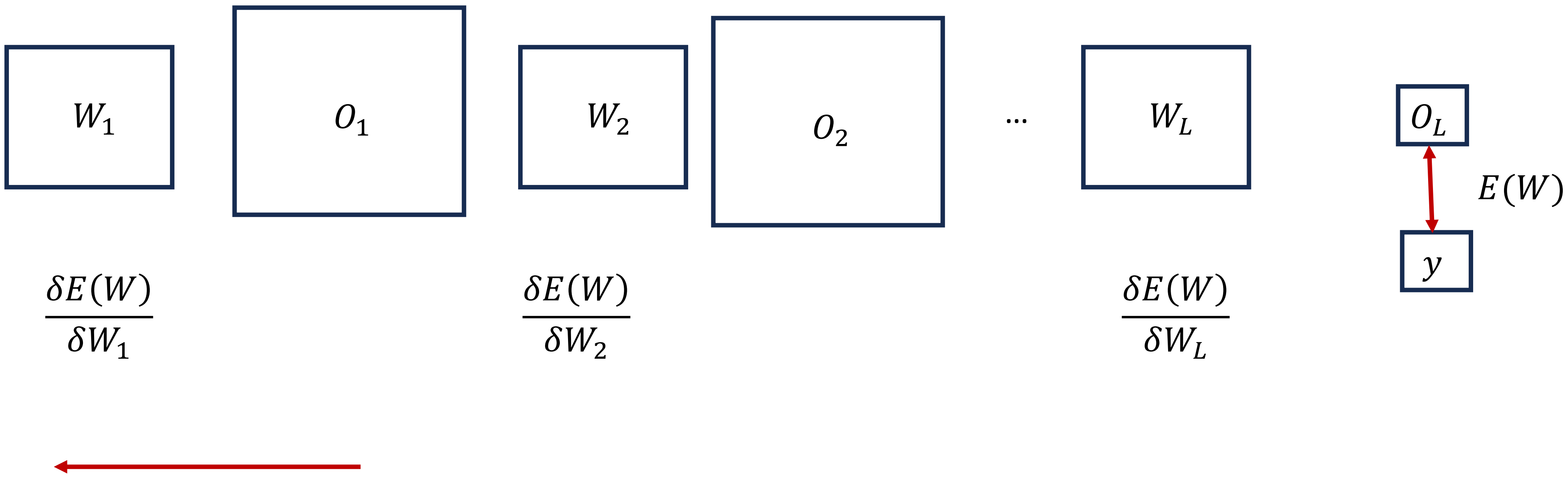
For a single image

CNN Training: Backward Propagation



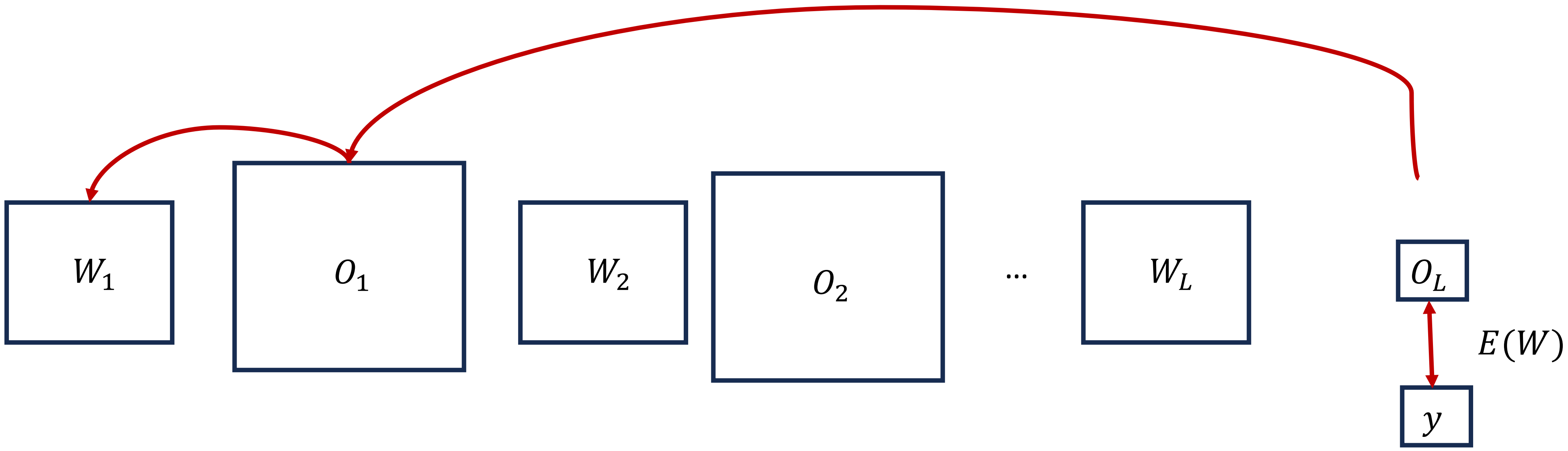
For a single image

CNN Training: Backward Propagation



For a single image

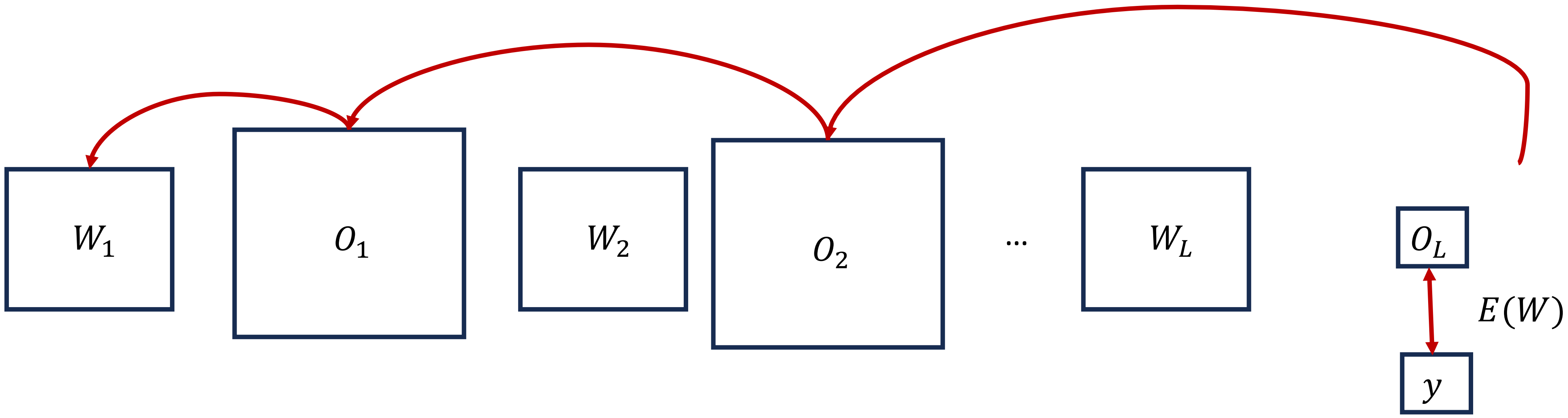
CNN Training: Backward Propagation



$$\frac{\delta E(W)}{\delta W_1} = \frac{\delta O_1}{\delta W_1} \times \frac{\delta E(W)}{\delta O_1}$$

For a single image

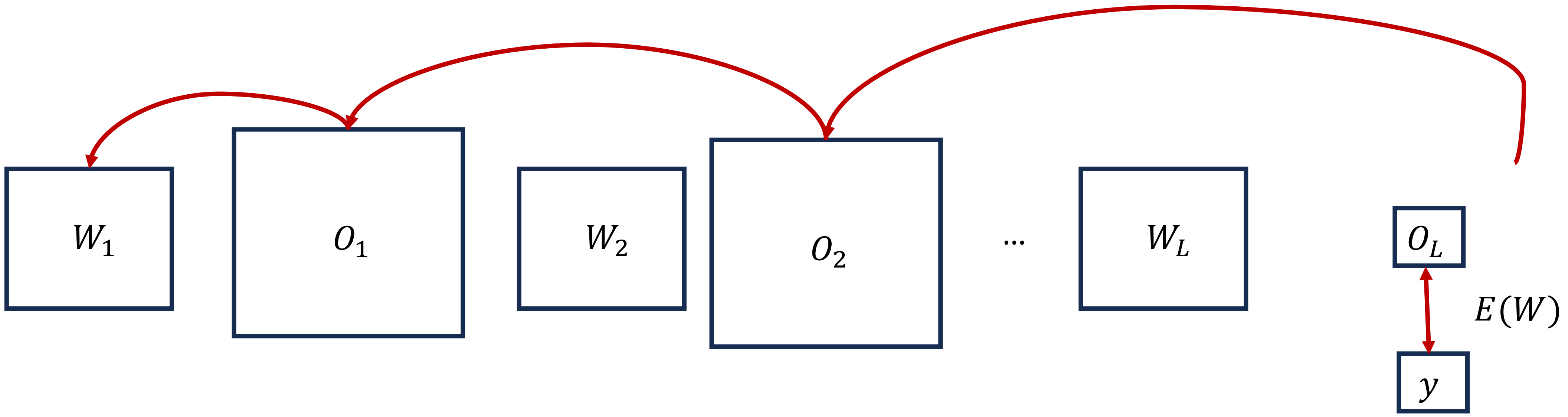
CNN Training: Backward Propagation



$$\frac{\delta E(W)}{\delta W_1} = \frac{\delta O_1}{\delta W_1} \times \frac{\delta O_2}{\delta O_1} \times \frac{\delta E(W)}{\delta O_2}$$

For a single image

CNN Training: Backward Propagation



$$\frac{\delta E(W)}{\delta W_1} = \frac{\delta O_1}{\delta W_1} \times \frac{\delta O_2}{\delta O_1} \times \frac{\delta O_3}{\delta O_2} \times \cdots \times \frac{\delta O_L}{\delta O_{L-1}} \times \frac{\delta E(W)}{\delta O_L}$$

For a single image

Training

$$E(W): \min_{W} \sum (y_i - F(x_i : W))^\wedge 2$$

$$W_{t+1} \leftarrow W_t - \alpha \frac{\delta E(W)}{\delta W}$$

Iteratively

$$\frac{\delta E(W)}{\delta W} = 2\sum \underbrace{\left(y_i - F(x_i : W)\right)} \times \frac{\delta F(x_i : W)}{\delta W}$$

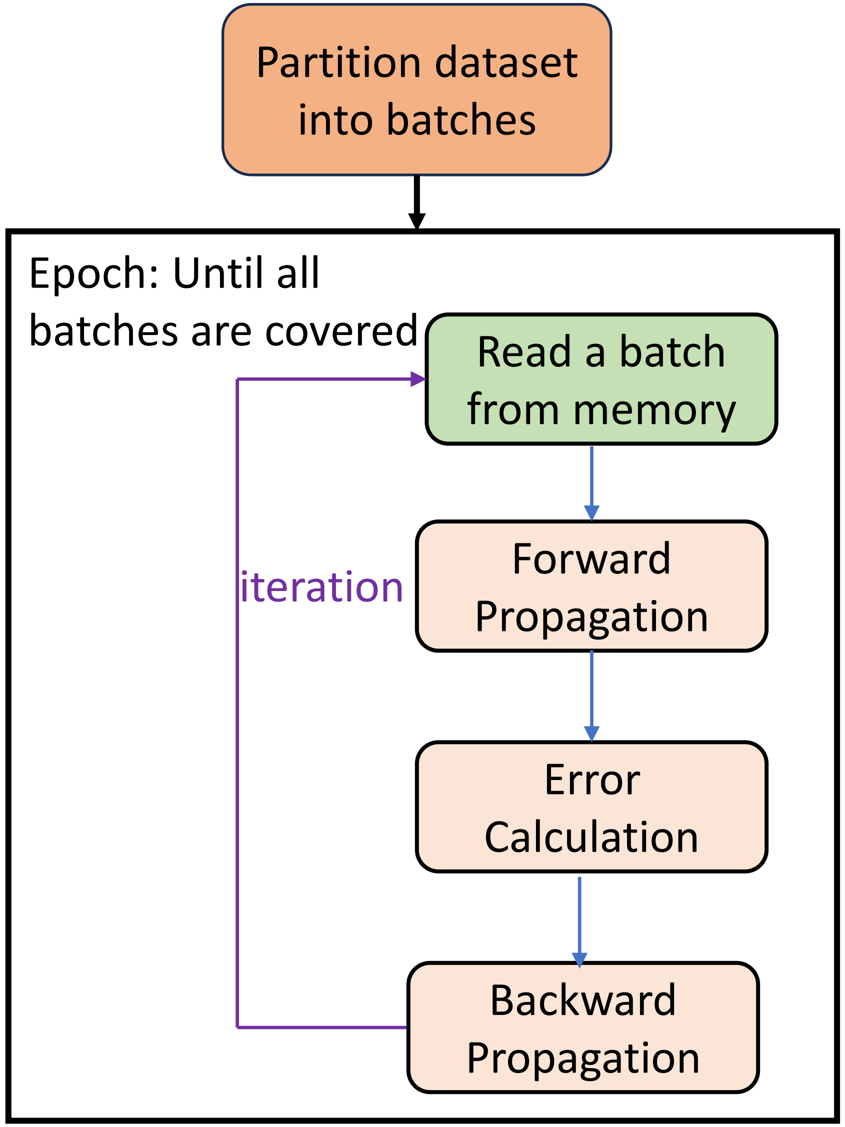
For all samples, in each iteration

Error

For multiple images in a batch: Simply sum up the gradients

$$\frac{\delta E(W)}{\delta W_l} = \boxed{\frac{\delta O_l}{\delta W_l} \times \frac{\delta O_{l+1}}{\delta O_l} \times \frac{\delta O_{l+2}}{\delta O_{l+1}} \times \cdots \times \frac{\delta O_L}{\delta O_{L-1}}} \times \boxed{\frac{\delta E(W)}{\delta O_L}}$$

Training a Machine Learning Model



Pytorch

- Python wrapper for Torch library
  - Open source library for working with tensors
- Great accelerator support.
  - Automatically links to device specific accelerated libraries such as cuDNN, hipBLAS.
- Abstracts away the hardware/system complexity of building and training AI/ML models

Pytorch - Workflows

- Inference: Use a pretrained model to perform inference on data

- Training: Train a model using a dataset
  - Fine-tune a pre-trained model using a dataset
  - Train a new custom model from scratch

Pytorch - Inference

- Pre-trained models available at pytorch
- Vision Models - https://docs.pytorch.org/vision/stable/models.html
- Transformer Models (for NLP) - https://docs.pytorch.org/text/stable/models.html

Pytorch - Inference

- Download the model weights from pytorch ‘cloud’
- Apply them to a model architecture
- Transform the image on which you need to perform inference based on the requirements of the model
- Perform inference using the model

Pytorch - Inference

- Download the model weights from pytorch ‘cloud’
- Apply them to a model architecture
- Transform the image on which you need to perform inference based on the requirements of the model
- Perform inference using the model

Pytorch - Inference

```
from torchvision.io import decode_image
from torchvision.models import resnet50, ResNet50_Weights

img = decode_image("test/assets/encode_jpeg/grace_hopper_517x606.jpg")

# Step 1: Initialize model with the best available weights
weights = ResNet50_Weights.DEFAULT
model = resnet50(weights=weights)
model.eval()

# Step 2: Initialize the inference transforms
preprocess = weights.transforms()

# Step 3: Apply inference preprocessing transforms
batch = preprocess(img).unsqueeze(0)

# Step 4: Use the model and print the predicted category
prediction = model(batch).squeeze(0).softmax(0)
class_id = prediction.argmax().item()
score = prediction[class_id].item()
category_name = weights.meta["categories"][class_id]
print(f"{category_name}: {100 * score:.1f}%")
```

Pytorch - Inference

- Download the model weights from pytorch ‘cloud’
- Apply them to a model architecture
- Transform the image on which you need to perform inference based on the requirements of the model
- Perform inference using the model

Pytorch - Inference

```
from torchvision.io import decode_image
from torchvision.models import resnet50, ResNet50_Weights

img = decode_image("test/assets/encode_jpeg/grace_hopper_517x606.jpg")

# Step 1: Initialize model with the best available weights
weights = ResNet50_Weights.DEFAULT
model = resnet50(weights=weights)
model.eval()

# Step 2: Initialize the inference transforms
preprocess = weights.transforms()

# Step 3: Apply inference preprocessing transforms
batch = preprocess(img).unsqueeze(0)

# Step 4: Use the model and print the predicted category
prediction = model(batch).squeeze(0).softmax(0)
class_id = prediction.argmax().item()
score = prediction[class_id].item()
category_name = weights.meta["categories"][class_id]
print(f"{category_name}: {100 * score:.1f}%")
```

Set model’s model to evaluation – We are telling the model that this is not training to avoid some training specific operations

Pytorch - Inference

- Download the model weights from pytorch ‘cloud’
- Apply them to a model architecture
- Transform the image on which you need to perform inference based on the requirements of the model
- Perform inference using the model

Pytorch - Inference

```
from torchvision.io import decode_image
from torchvision.models import resnet50, ResNet50_Weights

img = decode_image("test/assets/encode_jpeg/grace_hopper_517x606.jpg")

# Step 1: Initialize model with the best available weights
weights = ResNet50_Weights.DEFAULT
model = resnet50(weights=weights)
model.eval()

# Step 2: Initialize the inference transforms
preprocess = weights.transforms()

# Step 3: Apply inference preprocessing transforms
batch = preprocess(img).unsqueeze(0)

# Step 4: Use the model and print the predicted category
prediction = model(batch).squeeze(0).softmax(0)
class_id = prediction.argmax().item()
score = prediction[class_id].item()
category_name = weights.meta["categories"][class_id]
print(f"{category_name}: {100 * score:.1f}%")
```

Transformations can be cropping or resizing the image so that it is similar in dimensions to what model is expecting

Pytorch - Inference

- Download the model weights from pytorch ‘cloud’
- Apply them to a model architecture
- Transform the image on which you need to perform inference based on the requirements of the model
- Perform inference using the model

Pytorch - Inference

```
from torchvision.io import decode_image
from torchvision.models import resnet50, ResNet50_Weights

img = decode_image("test/assets/encode_jpeg/grace_hopper_517x606.jpg")

# Step 1: Initialize model with the best available weights
weights = ResNet50_Weights.DEFAULT
model = resnet50(weights=weights)
model.eval()

# Step 2: Initialize the inference transforms
preprocess = weights.transforms()

# Step 3: Apply inference preprocessing transforms
batch = preprocess(img).unsqueeze(0)

# Step 4: Use the model and print the predicted category
prediction = model(batch).squeeze(0).softmax(0)
class_id = prediction.argmax().item()
score = prediction[class_id].item()
category_name = weights.meta["categories"][class_id]
print(f"{category_name}: {100 * score:.1f}%")
```

For a batch of images, the model produces the prediction output.

Pytorch – Training

- Decide what dataset to use
- Create a Dataloader object – responsible for creating batches
- Define the model architecture
- Choose a loss function
- Training loop – actual testing occurs here
- Testing – periodically test to ensure model training is proceeding on the right direction

Pytorch – Training

- Datasets – List of common datasets are available on pytorch
- Vision - https://docs.pytorch.org/vision/stable/datasets.html
- Text - https://docs.pytorch.org/text/stable/datasets.html
- You can also use your own custom dataset - https://docs.pytorch.org/tutorials/beginner/data_loading_tutorial.html

Pytorch – Training

- Loading/Downloading a dataset from Pytorch

```
trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform)
```

Train = True/False -> training/testing portion
Transform = Applies the transform that is needed for the model that you wish to use (such as resizing, cropping)

```
testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform)
```

Pytorch – Training

- Creating a DataLoader object
  - Enables accessing data in batches
  - Performs random shuffling to improve AI/ML convergence

```
batch_size = 4

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,
                                          shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size,
                                         shuffle=False, num_workers=2)
```

Pytorch – Training

- Defining the model architecture
- Layers (and structure) need to be outlined in the __init__ method.
  - Specific attributes can be passed in here.
- Forward describes the data's movement through the model.
  - Activation functions tend to be applied here.
- Create an instance of the model.

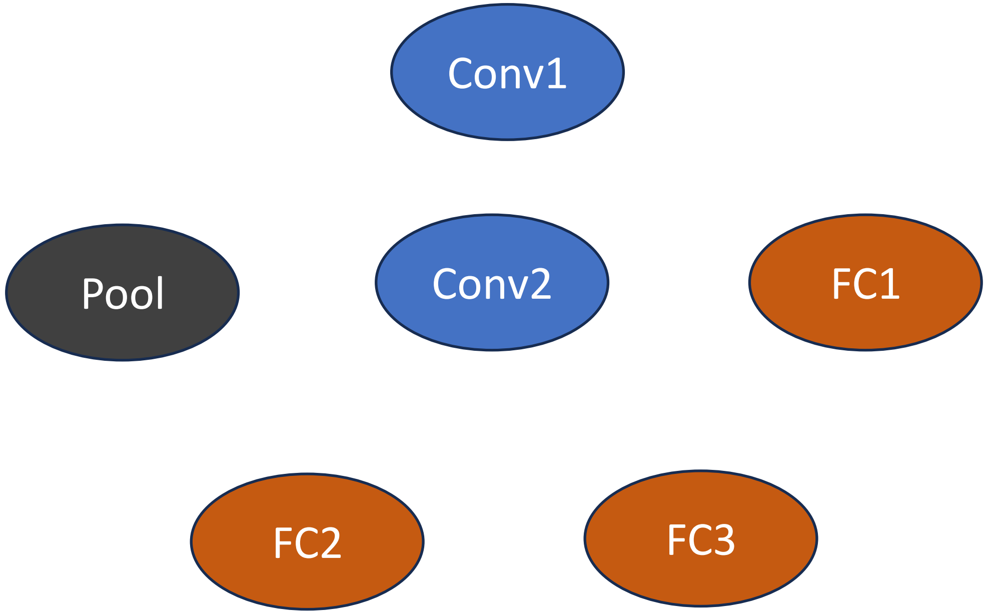
```
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x, 1) # flatten all dimensions except batch
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x


net = Net()
```

CASE SCHOOL OF ENGINEERING
CASE WESTERN RESERVE UNIVERSITY

Pytorch – Training

- The __init__ method is analogous to **defining types** of certain vertices within a task graph.
  - These are layers, but they also perform operations.
- However, there is nothing specifying quantity or order (how data flows through these).

```python
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)
```

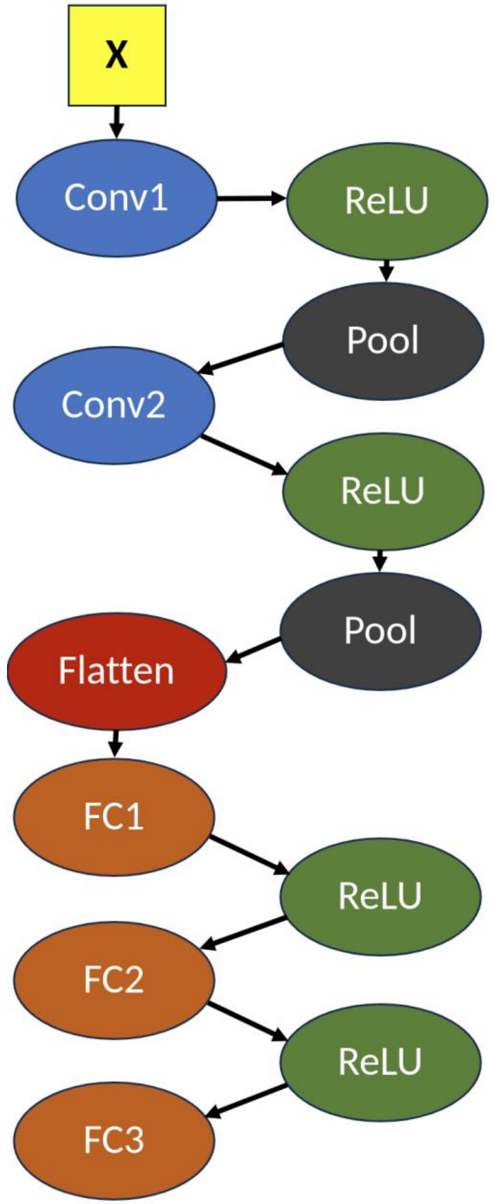


Pytorch – Training

- forward() specifies connectivity and data flow

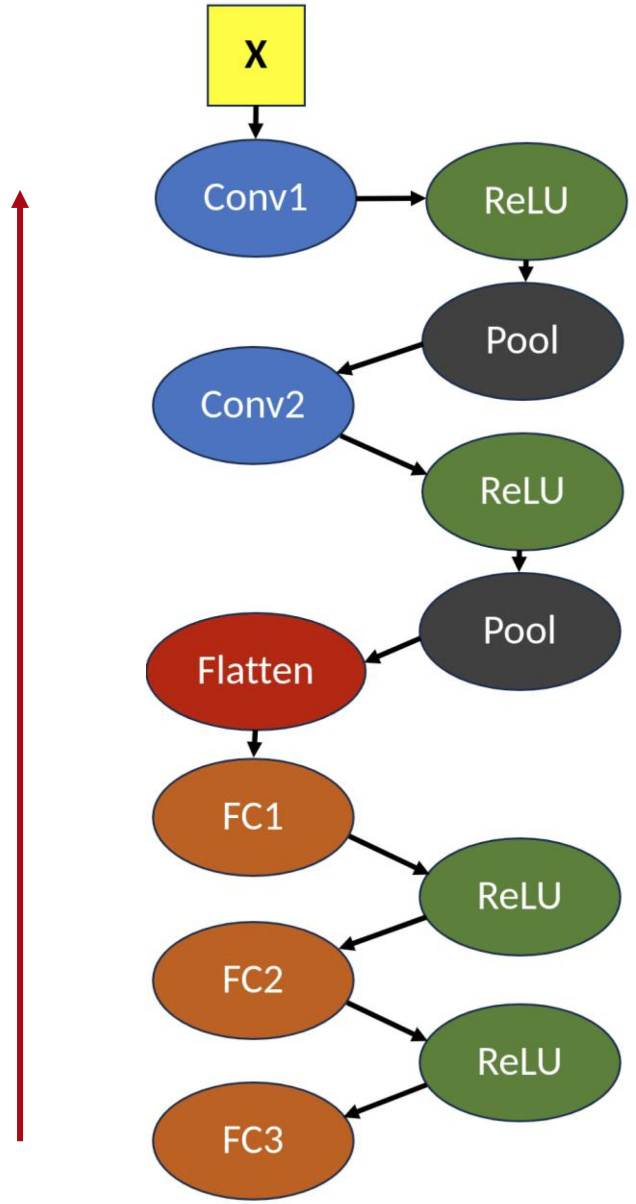
- During inference, when we call model(img), the forward function gets called

```
def forward(self, x):
    x = self.pool(F.relu(self.conv1(x)))
    x = self.pool(F.relu(self.conv2(x)))
    x = torch.flatten(x, 1) # flatten all dimensions except batch
    x = F.relu(self.fc1(x))
    x = F.relu(self.fc2(x))
    x = self.fc3(x)
    return x
```



Pytorch – Training

- In backward propagation, the gradients flow in the reverse direction of the connectivity
- Do we explicitly define the backward() function?
- No, **Autograd** functionality takes care of this in most circumstances



Pytorch – Training

- Choosing a loss function
- Appropriate loss function for your task, you can also build custom loss functions
- We also need to choose an optimizer
  - It applies the weight updates based on the gradients calculated in backpropagation
  - SGD, ADAM are popular optimizers

```
import torch.optim as optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```

Pytorch – Training

- Training Loop
- Epoch: Go over the entire dataset in batches to perform forward and backward propagation
- Stop training after a set number of epochs (or some other convergence criteria)

```python
for epoch in range(2):  # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 2000 == 1999:    # print every 2000 mini-batches
            print(f'[{epoch + 1}, {i + 1:5d}] loss: {running_loss / 2000:.3f}')
            running_loss = 0.0

print('Finished Training')
```

Pytorch – Training

- Testing Loop

- Test the performance of the model in an unknown data

```
correct = 0
total = 0
# since we're not training, we don't need to calculate the gradients for our outputs
with torch.no_grad():
    for data in testloader:
        images, labels = data
        # calculate outputs by running images through the network
        outputs = net(images)
        # the class with the highest energy is what we choose as prediction
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f'Accuracy of the network on the 10000 test images: {100 * correct // total} %')
```

Pytorch – Execution on Accelerators

- PyTorch defaults to the CPU, even if there are other resources available.

- Specifying a GPU is easy.
  - Indicate what device to use:

```
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
```

  - Send your model to the device:

```
net.to(device)
```

  - Send your data to the device:

```
inputs, labels = data[0].to(device), data[1].to(device)
```

Pytorch – Execution on Accelerators

- For simple networks, the speedup may not be noticeable.
  - However, on larger ones it can be quite substantial.
- Table is from the DarkNet repo (YOLO).
  - tkDNN is a neural network library built using cuDNN and tensorRT.

GeForce RTX 2080 Ti:

Network Size	Darknet, FPS (avg)	tkDNN TensorRT FP32, FPS	tkDNN TensorRT FP16, FPS	OpenCV FP16, FPS	tkDNN TensorRT FP16 batch=4, FPS	OpenCV FP16 batch=4, FPS	tkDNN Speedup
320	100	116	**202**	183	423	**430**	**4.3x**
416	82	103	**162**	159	284	**294**	**3.6x**
512	69	91	134	**138**	206	**216**	**3.1x**
608	53	62	103	**115**	150	**150**	**2.8x**
Tiny 416	443	609	**790**	773	**1774**	1353	**3.5x**
Tiny 416 CPU Core i7 7700HQ	3.4	-	-	42	-	39	**12x**

Next Class

- 10/9 Midterm
  - Best of luck!

- 10/14 Lecture 14
  - Pytorch-Lightning

Thank You

- Questions?

- Email: sanmukh.kuppannagari@case.edu