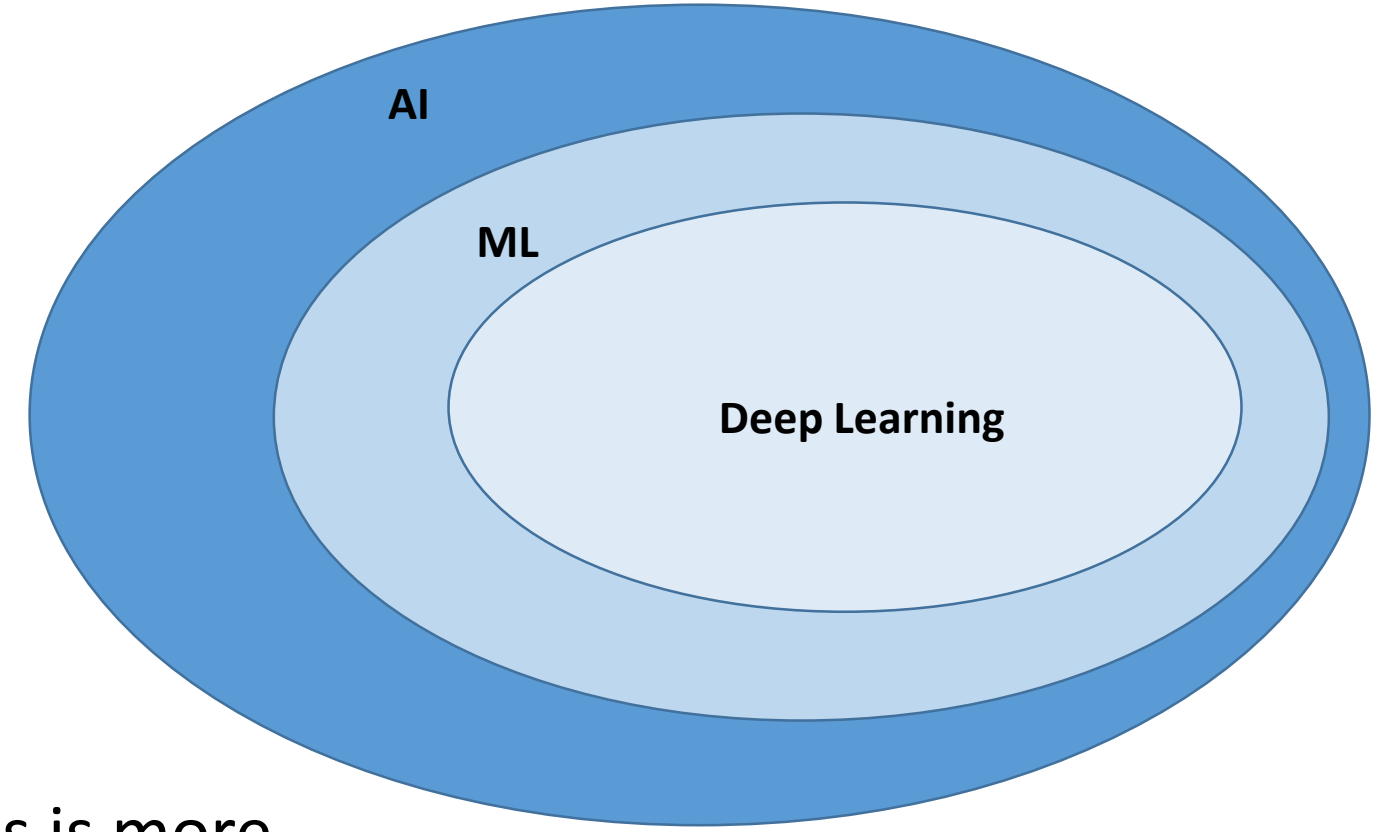
Deploying Neural Network Architectures using PyTorch

Data Mining Lectures

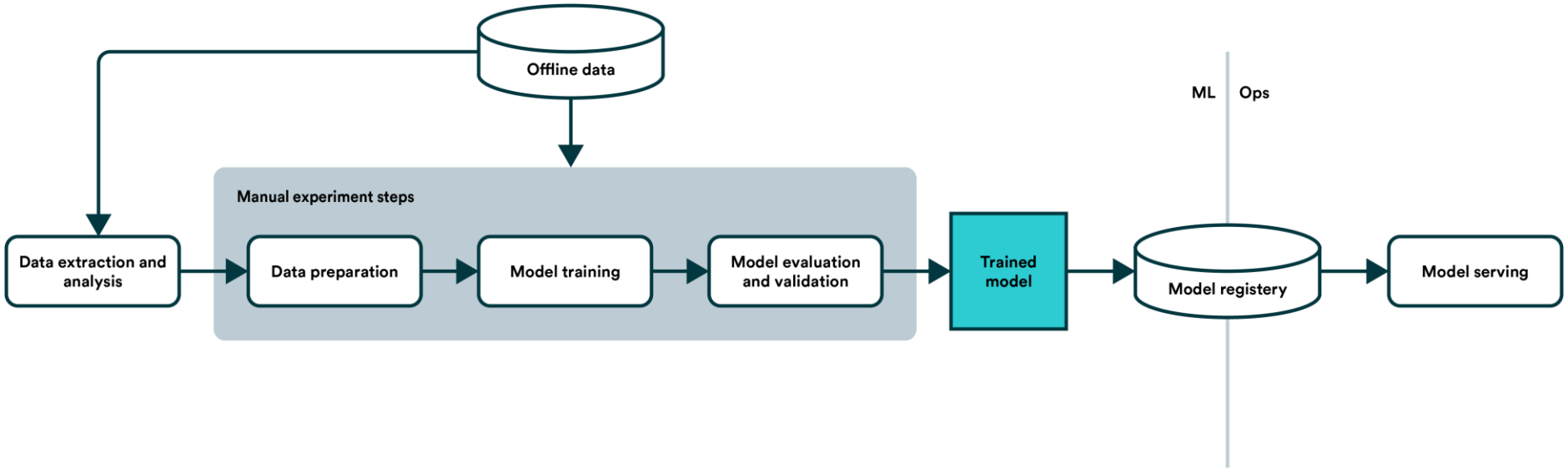
Machine Learning & Deep Learning



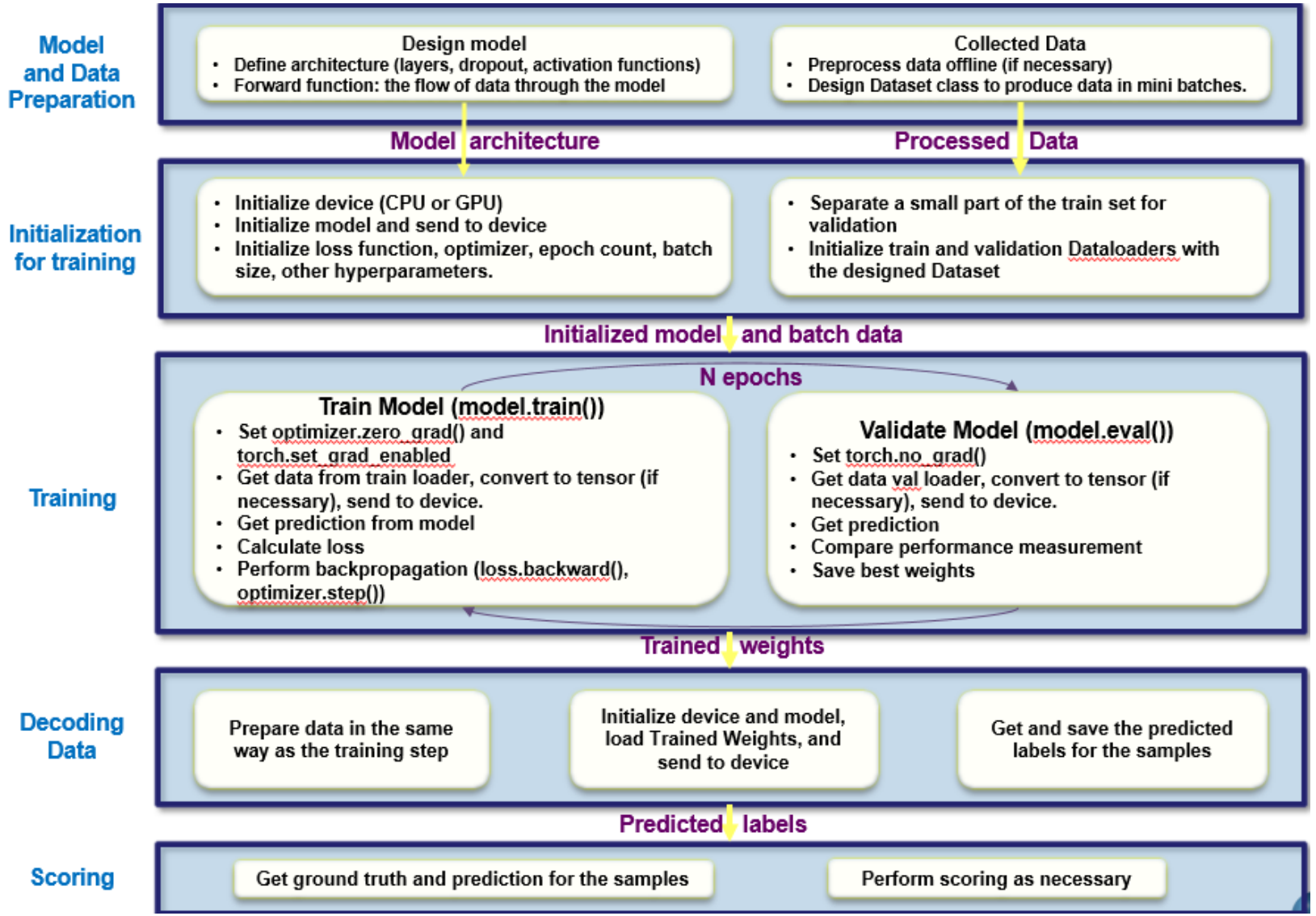
- Less is more
- ML; structured data, DL; unstructured data

Machine Learning/Deep Learning Pipeline

- **To successfully design and implement a machine learning model or deep learning model, researchers often follow the steps below:**
  - **Data collection**
  - **Data Preparation**
  - **Designing and training model**
  - **Evaluation**
- **We will explore these steps using PyTorch.**



Machine Learning/Deep Learning Pipeline

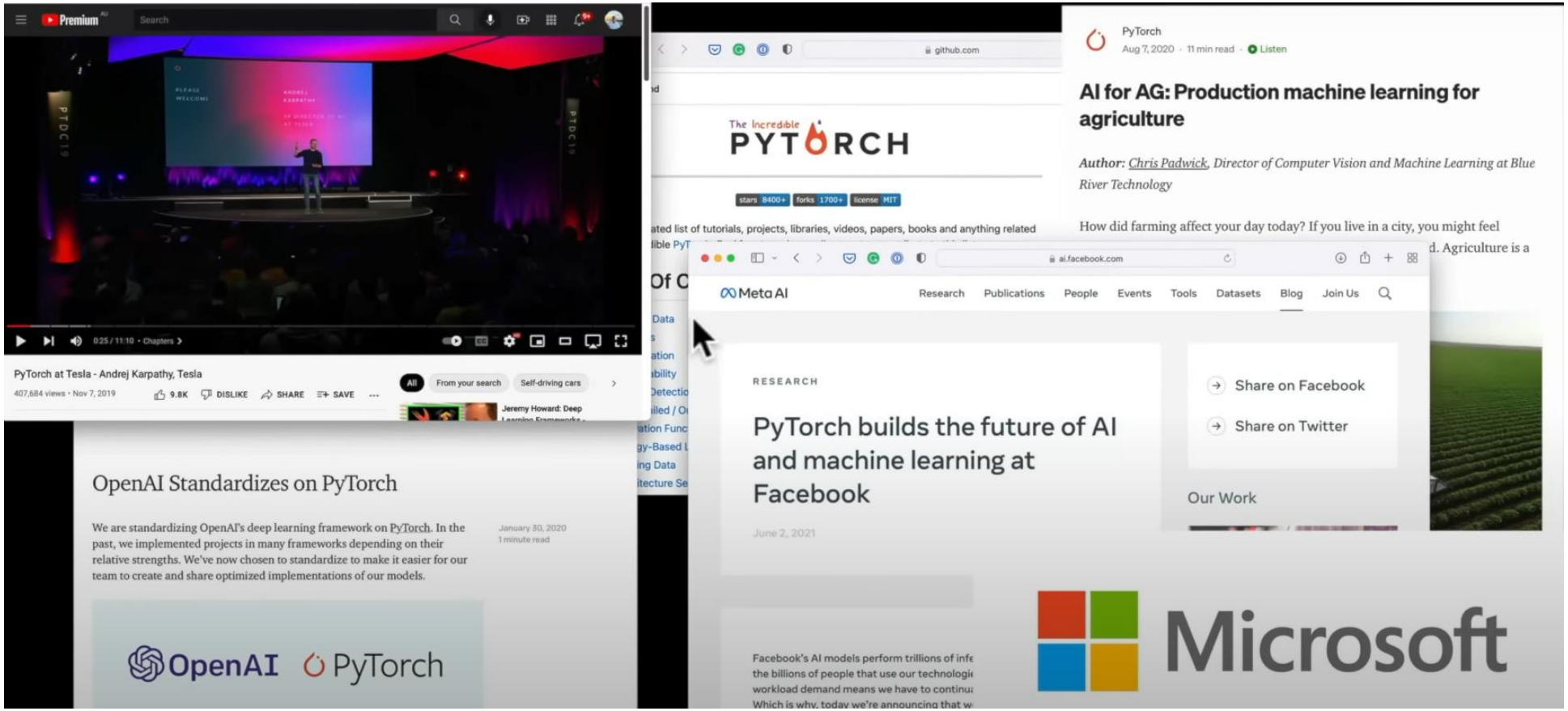


What is PyTorch?

- PyTorch is a machine and deep learning library by Facebook's AI Research Lab (FAIR).
- PyTorch has gained **popularity** among the research community as it is **easy** to develop and debug machine learning models in PyTorch.
- Nowadays, all **state-of-the-art** models and more are available in PyTorch and easy to integrate with any pipeline.
- With proper seeding, PyTorch can generate **reproducible** models.
- Like Tensorflow, a machine learning library by Google, PyTorch works with **tensors** which can be thought of as matrices with higher dimensions. These are equivalent to *ndarrays* in *NumPy*.
    - *A replacement for NumPy to use the power of GPUs*
- There are other deep learning framework available. For example, MATLAB can be used to design and train models.
- Keras is a high-level API library that uses TensorFlow as backend and suitable for beginners. It is straightforward to build and test models using Keras. However, it can be difficult to debug in Keras.

Why PyTorch?

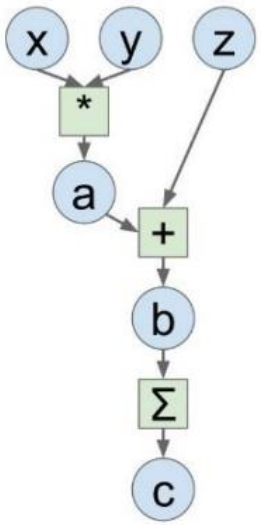
- Most popular & easier to develop and debug models



Why PyTorch?

- Faster

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()
```

Why PyTorch?

- PyTorch enables users to leverage a GPU through an interface called CUDA, which is a parallel computing platform and API.
  - A GPU is a graphics processing unit that was originally designed for video games, but it's very **fast** at crunching numbers.
  - **CUDA** allows software to use certain types of graphics processing units for **general-purpose computing**.
- PyTorch leverages CUDA to enable users to run their machine learning code on NVIDIA GPUs.
- TPUs, or tensor processing units, are another option for running PyTorch code, but GPUs are far more popular in practice.

Run a PyTorch process on GPU

- Anaconda is popular platform to deploy various deep learning and data science libraries. It facilitates the usage of separate environments for different setups.
- The appropriate conda environment should be activated with:

  “conda activate <env_name>”
- Source the .bashrc script from the home directory.
- The available GPU might need to be set up as an environment variable (it may vary in different setups): export CUDA_VISIBLE_DEVICES=‘0’
- The PyTorch script needs to recognize the available devices:

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

  This code checks if a GPU is available. If it is not available, the code utilizes CPU.
  - Sending data/model/objects is easy: obj.to(device)
- Alternate: cuda = True if torch.cuda.is_available() else False
  - However, this requires branching when an entity is being sent to the device.

What is a tensor in PyTorch?

- To use a neural network, **input** data must be **numerically** encoded.
- The numerical encoding is passed to the neural network to **learn patterns.**
- The **output** is a representation that can be **converted** to a **human-readable** form.
- Tensors are the fundamental building block of PyTorch.
- Tensors can represent almost any type of numerical data.
- The torch tensor is a key component of PyTorch.

Tensor in PyTorch vs Numpy Array

- A PyTorch Tensor is basically the same as a *numpy array*: *it does not know anything about deep learning or computational graphs or gradients, and is just a generic n-dimensional array to be used for arbitrary numeric computation.*
- The biggest difference between a numpy array and a PyTorch Tensor is that a PyTorch Tensor can run on either CPU or GPU. To run operations on the GPU, just cast the Tensor to a cuda datatype

PyTorch tensors

- By default, the 'requires_grad' attribute of a tensor in PyTorch is set to False, making it a non-trainable parameter.
- You can turn on the 'requires_grad' attribute using 't.requires_grad_()' or by setting it to True explicitly when creating the tensor.
- To access the value of the tensor, you can use 't.data'.
- To access the gradient of the tensor, you can use 't.grad'.
- The 'grad_fn' attribute keeps track of the history of operations for automatic differentiation (autograd) in PyTorch.
- The 'grad_fn' attribute represents the function that generated the tensor and is used to compute the gradients during backpropagation.
    - The gradient of a tensor represents the derivative of that tensor with respect to some other tensor.

```python
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()

print(c.grad_fn)
print(x.data)
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]])
```

Loading Data, Devices and CUDA

- Numpy arrays to PyTorch tensors
  - torch.from_numpy(x_train)
  - Returns a cpu tensor!
- PyTorchtensor to numpy
  - t.numpy()
- Using GPU acceleration
  - t.to()
  - Sends to whatever device (cudaor cpu)
- Fallback to cpu if gpu is unavailable:
  - torch.cuda.is_available()
- Check cpu/gpu tensor OR numpyarray ?
  - type(t)or t.type() returns
    - numpy.ndarray
    - torch.Tensor
      - CPU - torch.cpu.FloatTensor
      - GPU - torch.cuda.FloatTensor

Autograd

- Autograd is a PyTorch package for automatic differentiation
- Computes gradients without worrying about partial differentiation or chain rule
- Use 'backward()' method for computing gradients during backpropagation
- Gradients are accumulated for each step by default
- Zero out gradients after each update to prevent accumulation of gradients from previous computations
- 'tensor.grad_zero()' method sets gradients of the tensor to zero, ensuring that only current computation gradients are accumulated

```
# 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
```

Optimizer and Loss

- Optimizer:
  - Adam, SGD, etc.
  - Takes the parameters we want to update, the learning rate, and other hyper-parameters
  - Performs updates to minimize the loss function
- Loss:
  - Measure of how well the model is performing on the training data
  - Scalar value that is minimized during training
  - Various predefined loss functions to choose from in PyTorch
    - L1 loss (MAE), computes absolute difference between predicted and true value
    - MSE loss, computes square of the difference between predicted and true value
    - Cross-entropy loss, measures difference between predicted probability distribution and true probability distribution
    - Commonly used in classification tasks where the model output is a probability distribution over classes

Optimizer and Loss

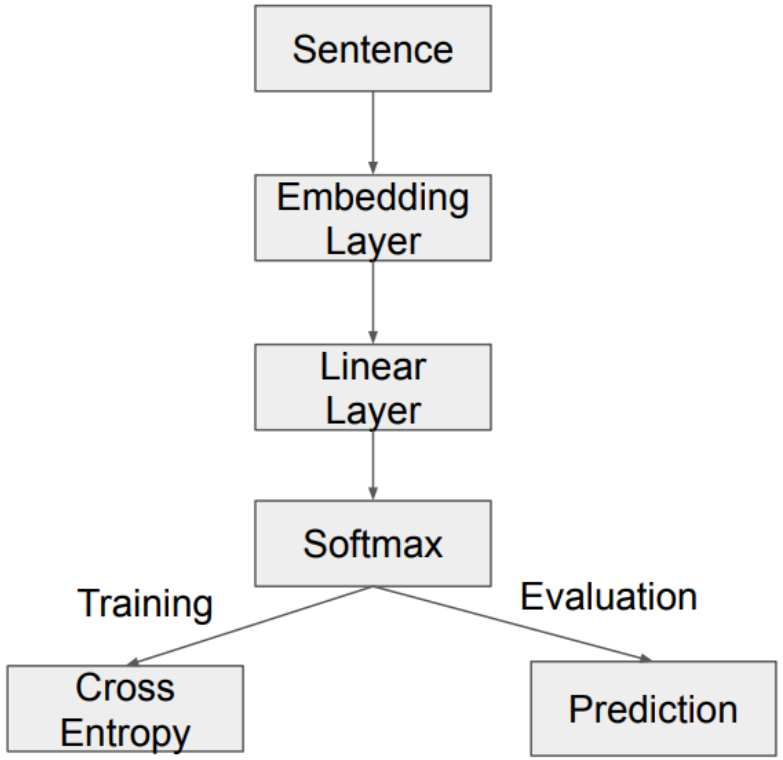
```
a = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)
b = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)


# Defines a SGD optimizer to update the parameters
optimizer = optim.SGD([a, b], lr=lr)

for epoch in range(n_epochs):
    yhat = a + b * x_train_tensor
    error = y_train_tensor - yhat
    loss = (error ** 2).mean()

    loss.backward()

    optimizer.step()

    optimizer.zero_grad()

print(a, b)
```

PyTorch Model

- In PyTorch, a model is represented by a regular Python class that inherits from the Module class.
  - Two components
    - __init__(self): it defines the parts that make up the model- in our case, two parameters, a and b
    - forward(self, x) : it performs the actual computation, that is, it outputs a prediction, given the inputx

```
class ManualLinearRegression(nn.Module):
    def __init__(self):
        super().__init__()
        # To make "a" and "b" real parameters of the model, we need to wrap them with nn.Parameter
        self.a = nn.Parameter(torch.randn(1, requires_grad=True, dtype=torch.float))
        self.b = nn.Parameter(torch.randn(1, requires_grad=True, dtype=torch.float))

    def forward(self, x):
        # Computes the outputs / predictions
        return self.a + self.b * x
```

Neural bag-of-words (ngrams) text classification



Design Model

- Initialize modules.
- Use linear layer here.
- Can change it to RNN,
- CNN, Transformer etc.
- Randomly initialize parameters
- Foward pass

```
import torch.nn as nn
import torch.nn.functional as F
class TextSentiment(nn.Module):
    def __init__(self, vocab_size, embed_dim, num_class):
        super().__init__()
        self.embedding = nn.EmbeddingBag(vocab_size, embed_dim, sparse=True)
        self.fc = nn.Linear(embed_dim, num_class)
        self.init_weights()

    def init_weights(self):
        initrange = 0.5
        self.embedding.weight.data.uniform_(-initrange, initrange)
        self.fc.weight.data.uniform_(-initrange, initrange)
        self.fc.bias.data.zero_()

    def forward(self, text, offsets):
        embedded = self.embedding(text, offsets)
        return self.fc(embedded)
```

Preprocess

- Build preprocessed dataset
- Build vocab

```
import torch
import torchtext
from torchtext.datasets import text_classification
NGRAMS = 2
import os
if not os.path.isdir('./.data'):
    os.mkdir('./.data')
train_dataset, test_dataset = text_classification.DATASETS['AG_NEWS'](
    root='./.data', ngrams=NGRAMS, vocab=None)
BATCH_SIZE = 16
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

```
VOCAB_SIZE = len(train_dataset.get_vocab())
EMBED_DIM = 32
NUN_CLASS = len(train_dataset.get_labels())
model = TextSentiment(VOCAB_SIZE, EMBED_DIM, NUN_CLASS).to(device)
```

Preprocess

- One example of dataset:

```
print(train_dataset[0])
```

```
(2, tensor([    572,     564,       2,    2326,   49106,     150,      88,       3,
            1143,      14,      32,      15,      32,      16,  443749,       4,
             572,     499,      17,      10,  741769,       7,  468770,       4,
              52,    7019,    1050,     442,       2,   14341,     673,  141447,
          326092,   55044,    7887,     411,    9870,  628642,      43,      44,
             144,     145,  299709,  443750,   51274,     703,   14312,      23,
         1111134,  741770,  411508,  468771,    3779,   86384,  135944,  371666,
            4052]))
```

- Create batch (Used in SGD)
- Choose pad or not (Using [PAD])

```
def generate_batch(batch):
    label = torch.tensor([entry[0] for entry in batch])
    text = [entry[1] for entry in batch]
    offsets = [0] + [len(entry) for entry in text]
    # torch.Tensor.cumsum returns the cumulative sum
    # of elements in the dimension dim.
    # torch.Tensor([1.0, 2.0, 3.0]).cumsum(dim=0)

    offsets = torch.tensor(offsets[:-1]).cumsum(dim=0)
    text = torch.cat(text)
    return text, offsets, label
```

Training each epoch

Iterable batches

Before each optimization, make previous gradients zeros

Forward pass to compute loss

Backforward propagation to compute gradients and update parameters

After each epoch, do learning rate decay (optional)

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

def train_func(sub_train_):

    # Train the model
    train_loss = 0
    train_acc = 0
    data = DataLoader(sub_train_, batch_size=BATCH_SIZE, shuffle=True,
                      collate_fn=generate_batch)
    for i, (text, offsets, cls) in enumerate(data):
        optimizer.zero_grad()
        text, offsets, cls = text.to(device), offsets.to(device), cls.to(device)
        output = model(text, offsets)
        loss = criterion(output, cls)
        train_loss += loss.item()
        loss.backward()
        optimizer.step()
        train_acc += (output.argmax(1) == cls).sum().item()

    # Adjust the learning rate
    scheduler.step()

    return train_loss / len(sub_train_), train_acc / len(sub_train_)
```

Test process

Do not need back propagation or parameter update !

```
def test(data_):
    loss = 0
    acc = 0
    data = DataLoader(data_, batch_size=BATCH_SIZE, collate_fn=generate_batch)
    for text, offsets, cls in data:
        text, offsets, cls = text.to(device), offsets.to(device), cls.to(device)
        with torch.no_grad():
            output = model(text, offsets)
            loss = criterion(output, cls)
            loss += loss.item()
            acc += (output.argmax(1) == cls).sum().item()

    return loss / len(data_), acc / len(data_)
```

The whole training process

- Use CrossEntropyLoss() as the criterion.
- The input is the output of the model.
- First do logsoftmax, then compute cross-entropy loss.
  - Use SGD as optimizer.
  - Use exponential decay to decrease learning rate

```
import time
from torch.utils.data.dataset import random_split
N_EPOCHS = 5
min_valid_loss = float('inf')

criterion = torch.nn.CrossEntropyLoss().to(device)
optimizer = torch.optim.SGD(model.parameters(), lr=4.0)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, 1, gamma=0.9)

train_len = int(len(train_dataset) * 0.95)
sub_train_, sub_valid_ = \
    random_split(train_dataset, [train_len, len(train_dataset) - train_len])

for epoch in range(N_EPOCHS):

    start_time = time.time()
    train_loss, train_acc = train_func(sub_train_)
    valid_loss, valid_acc = test(sub_valid_)

    secs = int(time.time() - start_time)
    mins = secs / 60
    secs = secs % 60

    print('Epoch: %d' %(epoch + 1), " | time in %d minutes, %d seconds" %(mins, secs))
    print(f'\tLoss: {train_loss:.4f}(train)\t|\tAcc: {train_acc * 100:.1f}%(train)')
    print(f'\tLoss: {valid_loss:.4f}(valid)\t|\tAcc: {valid_acc * 100:.1f}%(valid)')
```

Evaluation with test dataset or random news

```
print('Checking the results of test dataset...')
test_loss, test_acc = test(test_dataset)
print(f'\tLoss: {test_loss:.4f}(test)\t|\tAcc: {test_acc * 100:.1f}%(test)')
```

```
import re
from torchtext.data.utils import ngrams_iterator
from torchtext.data.utils import get_tokenizer

ag_news_label = {1 : "World",
                 2 : "Sports",
                 3 : "Business",
                 4 : "Sci/Tec"}

def predict(text, model, vocab, ngrams):
    tokenizer = get_tokenizer("basic_english")
    with torch.no_grad():
        text = torch.tensor([vocab[token]
                             for token in ngrams_iterator(tokenizer(text), ngrams)])
        output = model(text, torch.tensor([0]))
        return output.argmax(1).item() + 1
```

```
ex_text_str = "MEMPHIS, Tenn. – Four days ago, Jon Rahm was \
    enduring the season's worst weather conditions on Sunday at The \
    Open on his way to a closing 75 at Royal Portrush, which \
    considering the wind and the rain was a respectable showing. \
    Thursday's first round at the WGC-FedEx St. Jude Invitational \
    was another story. With temperatures in the mid-80s and hardly any \
    wind, the Spaniard was 13 strokes better in a flawless round. \
    Thanks to his best putting performance on the PGA Tour, Rahm \
    finished with an 8-under 62 for a three-stroke lead, which \
    was even more impressive considering he'd never played the \
    front nine at TPC Southwind."

vocab = train_dataset.get_vocab()
model = model.to("cpu")

print("This is a %s news" %ag_news_label[predict(ex_text_str, model, vocab, 2)])
```

CNN example

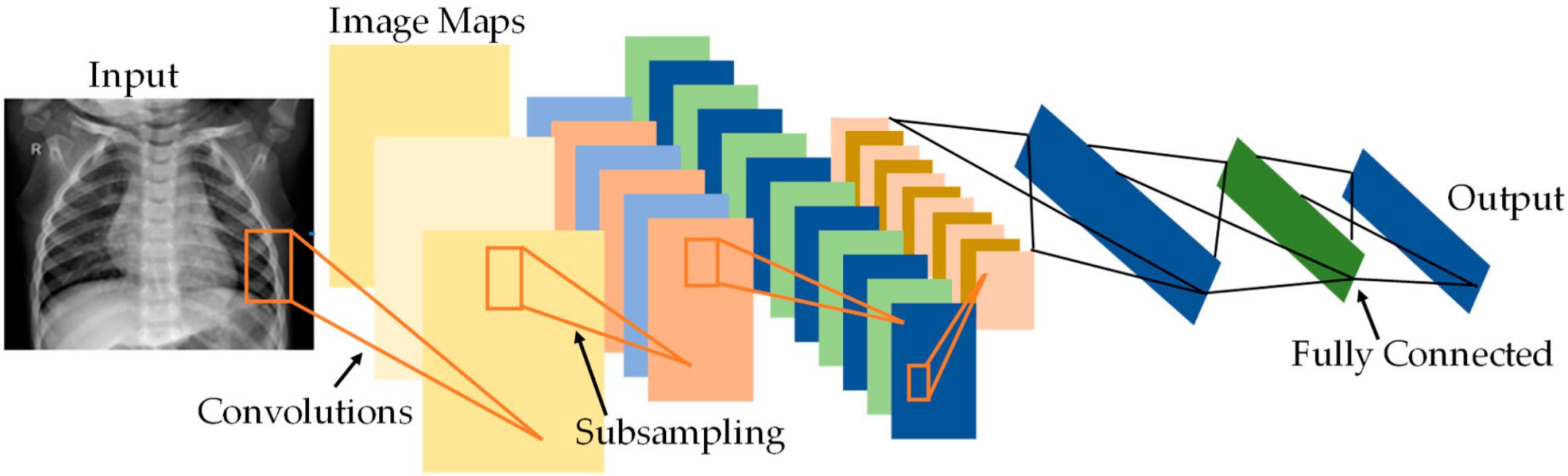


Image Maps
Input
Output
Fully Connected
Convolutions
Subsampling

Load Required Classes and Modules

```
import torch
```
To use the Torch in Python

```
import torch.nn as nn
import torch.nn.functional as F
```
To create a model by layers

```
import torch.optim as optim
from torch.optim import lr_scheduler
```
To set the optimization

```
import numpy as np
```
To manipulate arrays

```
import torchvision
from torchvision import datasets, models, transforms
```
To Process the data and use the existing Models

```
import os
import copy
```
To save the best model and get data files

Code Reference: https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html

Image Transformation

- We need to transform the images:
  1. Change size of all images to a unanimous value.
  2. Convert them to tensor. Tensor transfer the values from scale **0-255 to 0-1.**
  3. Normalize the image with mean and standard deviation for RGB values.

```
data_transforms = {
    'train': transforms.Compose([
        transforms.Resize(256),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
    'val': transforms.Compose([
        transforms.Resize(256),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
}
```

Code Reference: https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html

Image Normalization

- After converting image to tensor, every pixel value is in **range [0,1]**
- Then For every pixel, we apply the following formula to each of the channel values and **range will be [-1,1]**:

$$\frac{\text{Pixel\_Red\_Value} - \text{Red\_Channel\_Mean}}{\text{Red\_Channel\_Standard\_Deviation}}$$

$$\frac{\text{Pixel\_Red\_Value} - 0.486}{0.229}$$

$$\frac{\text{Pixel\_Green\_Value} - \text{Green\_Channel\_Mean}}{\text{Green\_Channel\_Standard\_Deviation}}$$

$$\frac{\text{Pixel\_Green\_Value} - 0.456}{0.224}$$

$$\frac{\text{Pixel\_Blue\_Value} - \text{Blue\_Channel\_Mean}}{\text{Blue\_Channel\_Standard\_Deviation}}$$

$$\frac{\text{Pixel\_Blue\_Value} - 0.406}{0.225}$$

Why Image Normalization?

- In general , in order to handle noise in data, data can be transformed globally to change the scale or range of data (normalize).[1]

- In Convolutional Neural Network if we don't scale (normalize) the values, the range of different features (e.g. image channels) will be different.[2]

- Since the values are multiplied by learning rate, the features that have **larger scale** might be **over-compensated** and features with **smaller scale** might be **under-compensated**.[2]

1. https://www.coursera.org/lecture/data-genes-medicine/data-normalization-jGN7k
2. https://stats.stackexchange.com/questions/185853/why-do-we-need-to-normalize-the-images-before-we-put-them-into-cnn

More Data Preprocessing

- In addition to the mentioned data preprocessing, there are some transformation that are used mainly for **data augmentation**:
  - transforms.RandomHorizontalFlip()
  - transforms.RandomResizedCrop(224)
- **Data augmentation** is a strategy that enables practitioners to significantly increase the diversity of data available for training models, **without actually collecting new data.**[1]

1. https://bair.berkeley.edu/blog/2019/06/07/data_aug/

Mini Batch and Epoch

- Batch: Number of images which is propagated to a model iteration.
- Epoch: An epoch refers to one cycle through the full training dataset.[1]

```
batch_size = 4
num_epochs = 30
```

- Example:
  - ❖ Number of Images = 1024
  - ❖ Batch Size = 4
  - ❖ Number of Iterations in Every Epoch: 256

1. https://deepai.org/machine-learning-glossary-and-terms/epoch

Load Data and Set Device

Dataset Directory

```
data_dir = 'datasets/hw2'
```

Load Data

```
image_datasets = {x: datasets.ImageFolder(os.path.join(data_dir, x), data_transforms[x])
          for x in ['train', 'val']}


dataloaders = {x: torch.utils.data.DataLoader(image_datasets[x], batch_size=4, shuffle=True, num_workers=4)
      for x in ['train', 'val']}
```

```
dataset_sizes = {x: len(image_datasets[x]) for x in ['train', 'val']}


class_names = image_datasets['train'].classes
```

Get number of images and name of classes

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

Set the Device to GPU or CPU

Code Reference: https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html

Sample Network

- Here is an example of a PyTorch model

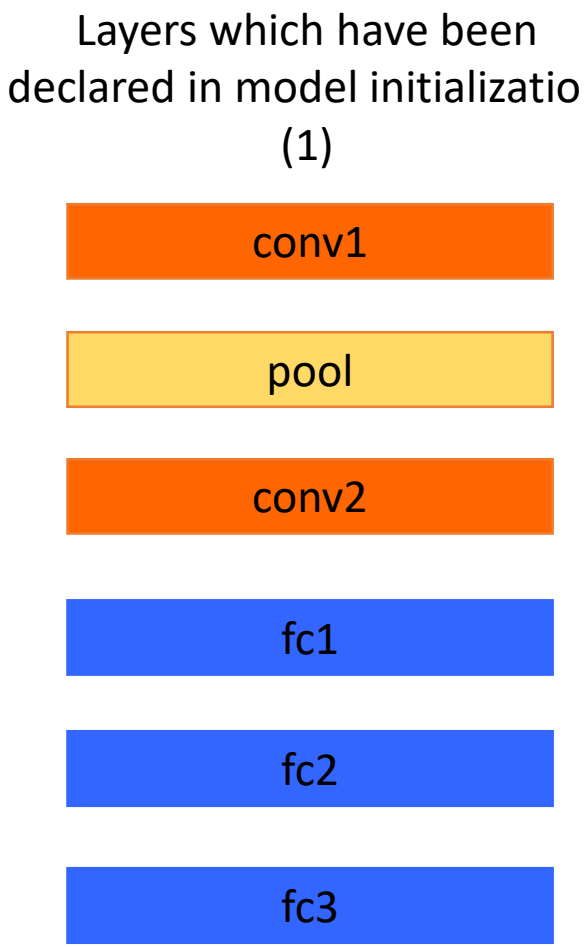
```
class Sample_Network(nn.Module):
    def __init__(self):
        super(Sample_Network, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 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 = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

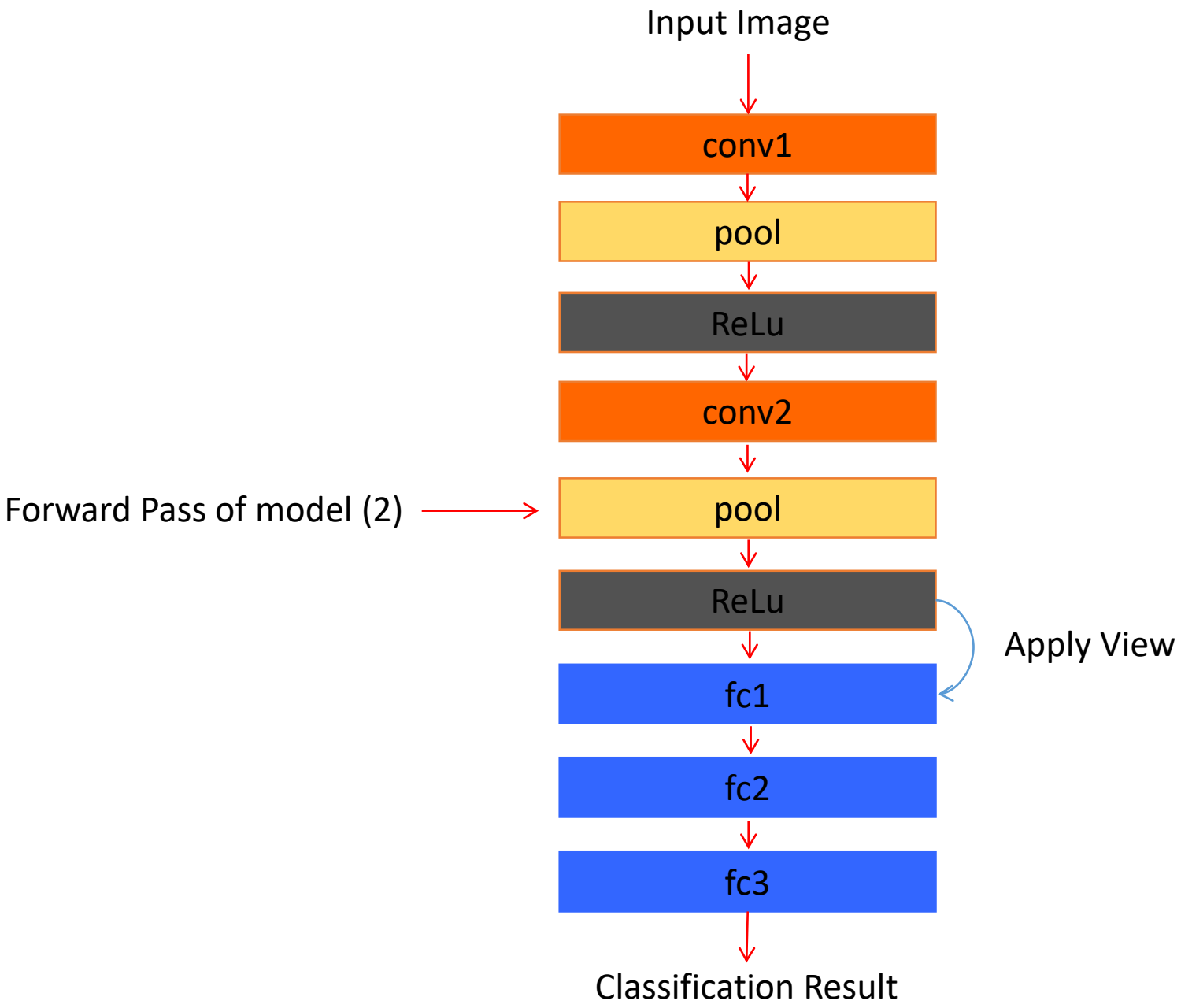
← Define the layers of model (1)

← Forward function is called during forward pass (2)

Code Reference:
https://github.com/pytorch/tutorials/blob/master/beginner_source/blitz/neural_networks_tutorial.py

Visualization of Sample Network




Input Image
conv1
pool
ReLu
conv2
pool
Forward Pass of model (2)
ReLu
Apply View
fc1
fc2
fc3
Classification Result

Before Start Training

- For starting the training process we need to
  1. Initialize an instance from the model which we have already defined
  2. Specify the criterion (loss) for evaluation of model
  3. Specify the setting of optimizer
  4. Specify the way learning rate changes during training

1
```
model = Sample_Network()
```
2
```
criterion = nn.CrossEntropyLoss()
```
3
```
optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
```
4
```
scheduler = lr_scheduler.StepLR(optimizer, step_size=7, gamma=0.1)
```

Code Reference: https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html

Save the Best Model Parameter

- We need to train the network for the specified number of epochs.
- Before training process, we save the initial weight as the best model weight and set the best accuracy as zero.
- In every epoch and after finishing the training process, we use the trained model to select the model which has best performance on the validation set.

```
best_model_wts = copy.deepcopy(model.state_dict())
best_acc = 0.0
```

Code Reference: https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html

Iterate Over Train and Validation Sets in every Epoch

- In every epoch we either train the model or just use it for evaluation.
- For training, we need to set the model to **train** mode and for test we need to set to **eval** mode.

```
for phase in ['train', 'val']:
        if phase == 'train':
                model.train()
        else:
                model.eval()
```

Code Reference: https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html

Iterate Over every Minibatch

- We use the data loader which we have created in previous slides to go thorough the data.
- What we get from data loader are tensors for **images (inputs)** and **labels** and we need to transfer them to the device which we have created before.
- Note: Phase here is ‘train’ and ‘test’

```
for inputs, labels in dataloaders[phase]:
        inputs = inputs.to(device)
        labels = labels.to(device)
```

Code Reference: https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html

Prediction and Back Propagation

```
optimizer.zero_grad()
```
← Zero the gradient before start of a new mini batch

```
outputs = model(inputs)
```
← Apply Forward Function and get logit

```
_, preds = torch.max(outputs, 1)
```
← Get the highest logic as prediction

```
loss = criterion(outputs, labels)
```
← Compute the loss based on predicted value

```
if phase == 'train':
        loss.backward()
        optimizer.step()
```
← Back propagate if we are in train phase

```
running_loss += loss.item() * inputs.size(0)
```
← Sum the loss of batch with all loss values

```
running_corrects += torch.sum(preds == labels.data)
```
↖ Sum correctly predicted values in batch with all loss values

Code Reference: https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html

Finish Iterating over Data in One Epoch

- When iteration over all data finished then we need to compute the loss and save the best model.

Scheduler setting (e.g. learning rate) needs to be updates

```
if phase == 'train':
    scheduler.step()
```

Loss and accuracy needs to be computed at the end of epoch

```
epoch_loss = running_loss / dataset_sizes[phase]
epoch_acc = running_corrects.double() / dataset_sizes[phase]
```

```
if phase == 'val' and epoch_acc > best_acc:
    best_acc = epoch_acc
    best_model_wts = copy.deepcopy(model.state_dict())
    torch.save(best_model_wts , 'best_model_weight.pth')
```

Save the best model

Code Reference: https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html

Load Data for Test

Transform the test images

```
data_transforms = {
'test': transforms.Compose([
        transforms.Resize(256),
        transforms.ToTensor()
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
]),
}
```

```
data_dir = 'datasets/hw2'


image_datasets = {x: datasets.ImageFolder(os.path.join(data_dir, x), data_transforms[x])
                  for x in ['test']}

dataloaders = {x: torch.utils.data.DataLoader(image_datasets[x], batch_size=4, shuffle=True, num_workers=4)
               for x in ['test']}

dataset_sizes = {x: len(image_datasets[x]) for x in ['test']}
```

Load the data and get the dataset size

Code Reference: https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html

Test the Loaded Data

```
model.eval()
```

Set the model in evaluation mode

```
phase = 'test'

for inputs, labels in dataloaders[phase]:
        inputs = inputs.to(device)
        labels = labels.to(device)


        outputs = model(inputs)
        _, preds = torch.max(outputs, 1)
        loss = criterion(outputs, labels)


        running_loss += loss.item() * inputs.size(0)
        running_corrects += torch.sum(preds == labels.data)
```

Iterate over test data and compute loss and correctly predicted values

```
test_loss = running_loss / dataset_sizes[phase]
test_acc = running_corrects.double() / dataset_sizes[phase]
```

Compute the loss and Accuracy over all data

Code Reference: https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html