ECE C147/C247

Introduction to PyTorch

Pan Lu, UCLA CS

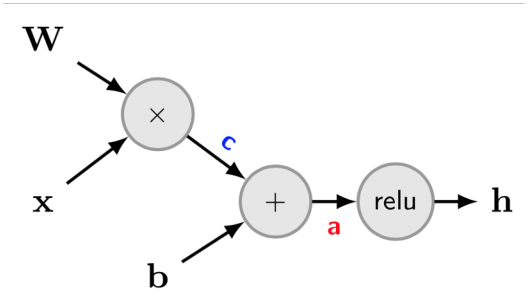
Introduction to PyTorch

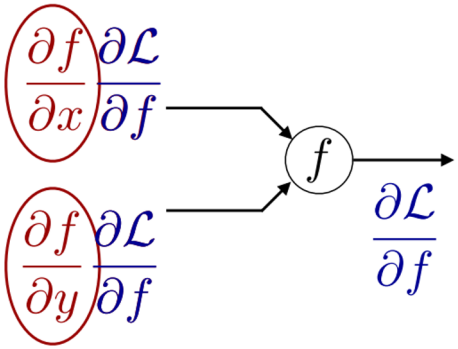
- What is PyTorch
- PyTorch basics
  - Tensors
  - Neural network layer
- PyTorch CNN example
  - Data preparation
  - Model construction
  - Model training
  - Model evaluation
- PyTorch advances
  - Useful tricks
  - Useful packages and tools

The Colab link for this lecture:
https://colab.research.google.com/drive/146XJEdXOguMQ41CbU-6TqS2gEuc061qI

The basic idea of deep learning libraries

- In deep learning libraries, you write code that builds a computational graph, akin to what we did in the backpropagation lectures.
- Deep learning libraries implement backpropagation for you, so that after the computational graph is built, you don't have to implement the backward pass!!
- Deep learning libraries also provide a straightforward API to do all training on GPUs.





A zoo of frameworks!

Caffe (UCB)

Caffe2 (Facebook)

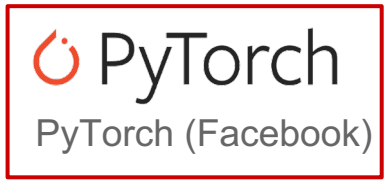
PaddlePaddle (Baidu)

Chainer

MXNet (Amazon)

CNTK (Microsoft)

Torch (NYU / Facebook)



PyTorch (Facebook)

We'll focus on these

JAX (Google)

Theano (U Montreal)

TensorFlow (Google)

And others...

PyTorch

Pytorch is a deep learning framework, i.e… set of functions and libraries which allow you to do higher-order programming designed for Python programming language based on Torch.

Pros:

- Based on Python: pythonic and shares commands with numpy
- Easy to get started: easy to install, simple syntax
- Community: active community, very good documents, including tutorial: https://pytorch.org/tutorials/.
- Dynamic computational graphs: the network behavior can be changed programmatically at runtime, easier model optimization
- Data parallelism: work among multiple CPU or GPU cores

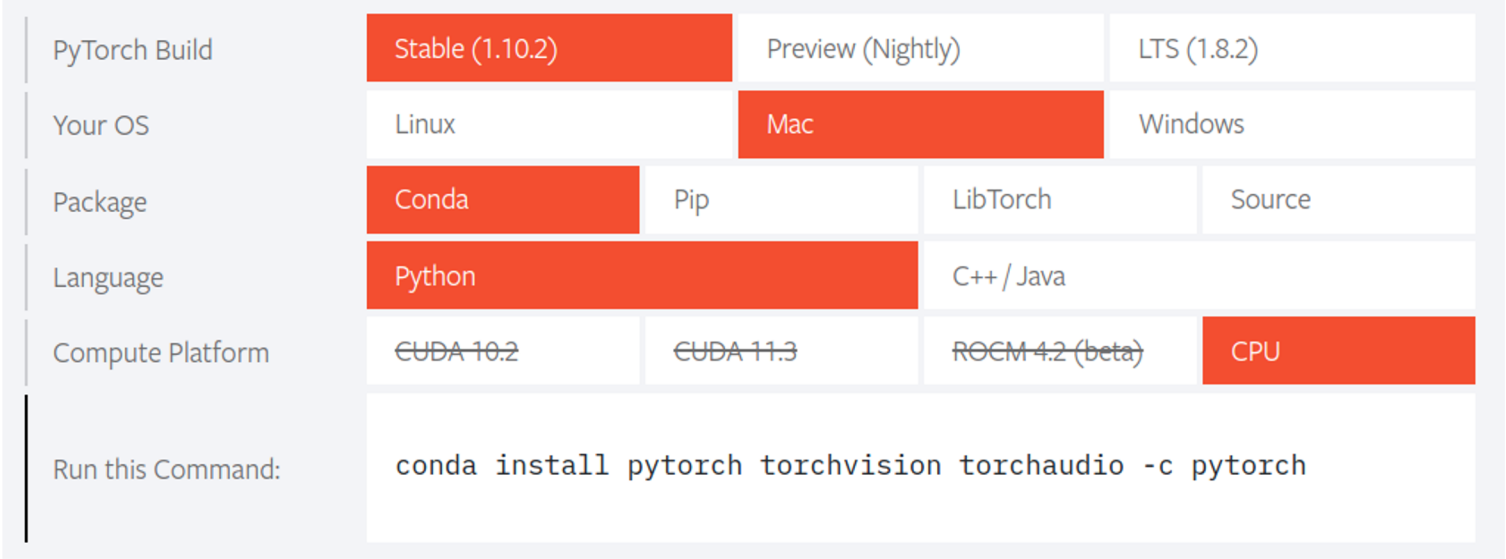
Cons:

- Limited monitoring and visualization interfaces
- Not as extensive as TensorFlow

PyTorch

Install PyTorch

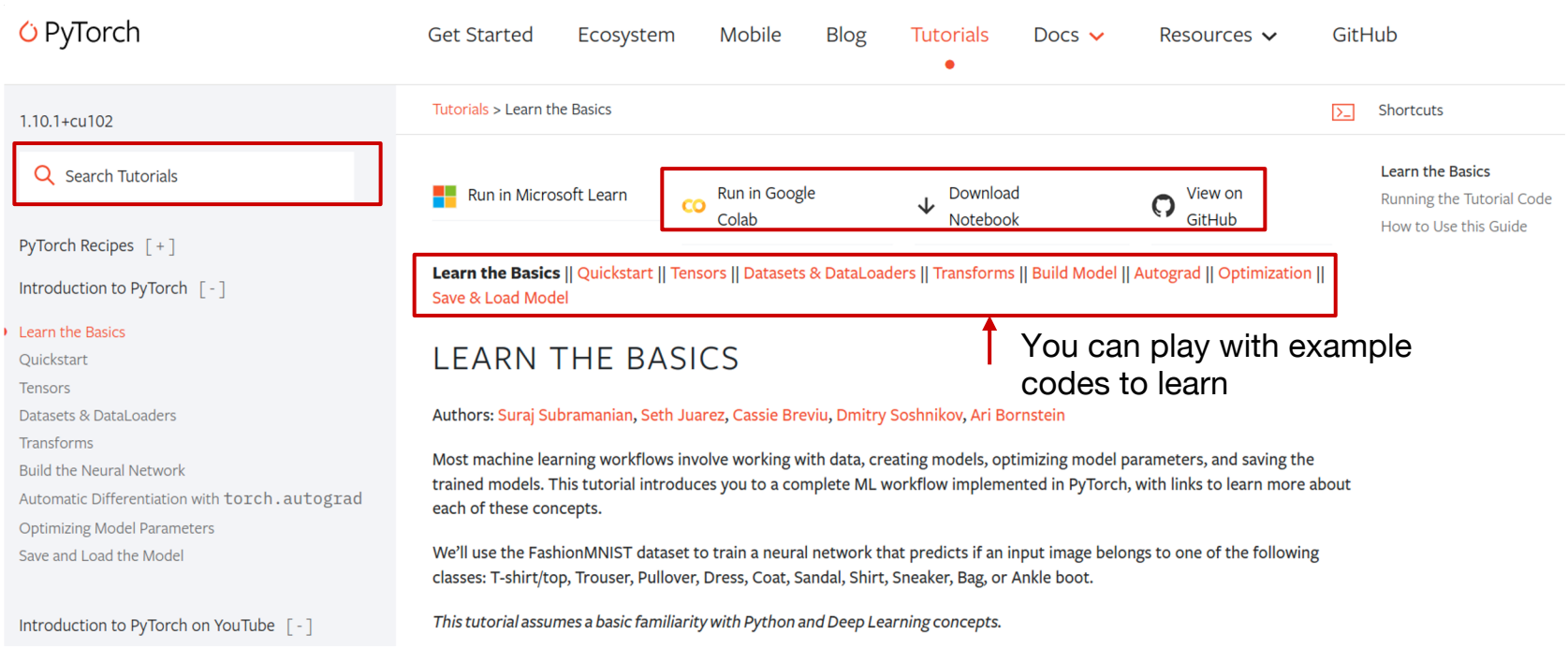
- https://pytorch.org/
- Mac:
  ```
  conda install pytorch torchvision torchaudio -c pytorch
  ```
- Windows:
  ```
  conda install pytorch torchvision torchaudio cpuonly -c pytorch
  ```



PyTorch

PyTorch tutorial

- https://pytorch.org/tutorials/
- PyTorch provide a very good online tutorials and documents



PyTorch Tensor

Tensors are a specialized data structure that are very similar to arrays and matrices. In PyTorch, we use tensors to encode the **inputs** and **outputs** of a model, as well as the model's **parameters**.

Tensors can be initialized in various ways.

- Directly from data:

```
data = [[1, 2],[3, 4]]
x_data = torch.tensor(data)
```

- From a NumPy array:

```
np_array = np.array(data)
x_np = torch.from_numpy(np_array)
```

- From another tensor:

```
x_ones = torch.ones_like(x_data) # retains the properties of x_data
```

- With random or constant values:

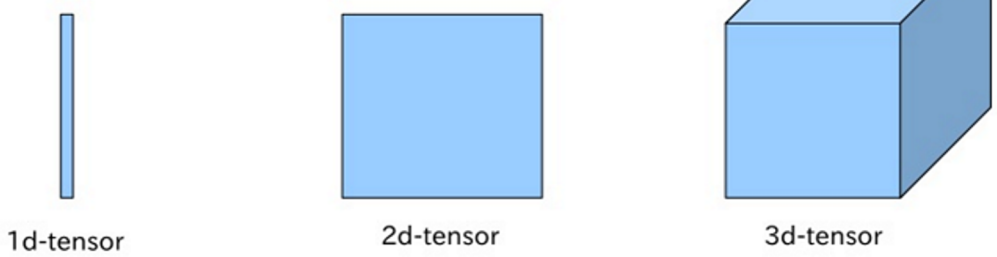
```
shape = (2,3,)
rand_tensor = torch.rand(shape)
ones_tensor = torch.ones(shape)
zeros_tensor = torch.zeros(shape)
```

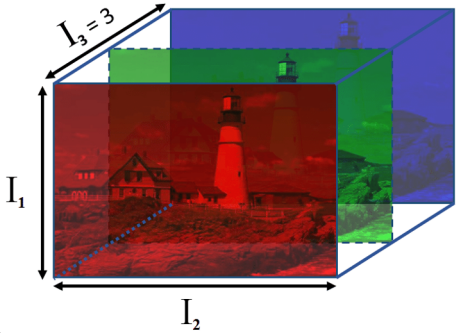
Aside: Tensors

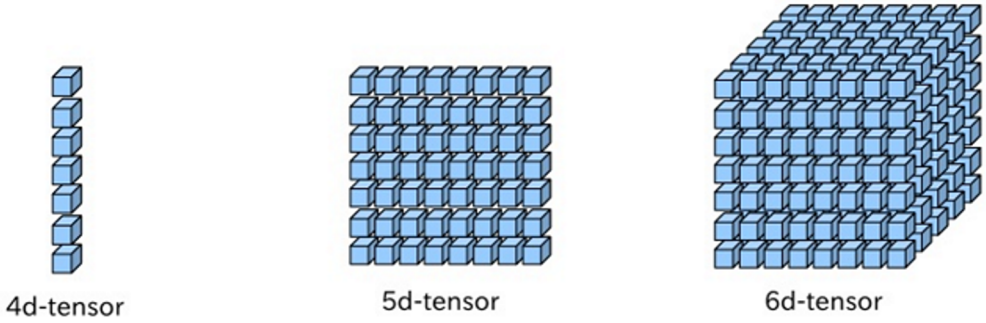
Tensors are multi-dimensional arrays with a uniform data type (called dtype)

- 1d-tensor is a vector, 2d-tensor is a matrix, 3d-tensor is a cube
- 2D color image = 3D tensor

Tensors are similar to numpy arrays, with the addition being that Tensors can also be used on a GPU to accelerate computing.





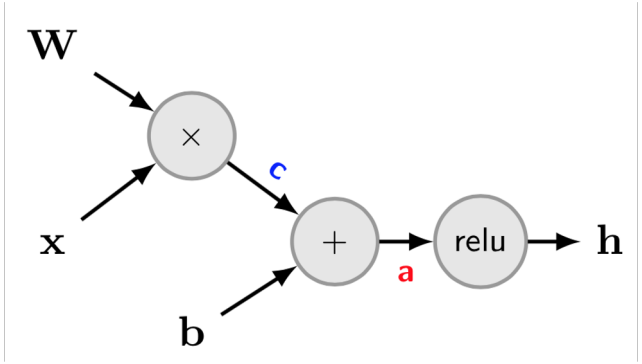


Neural network layer in NumPy

```
import numpy as np
```

```
num_in = 10
num_hidden = 50

# init numpy arrays
x = np.random.randn(num_in) # (10,)
W = np.random.randn(num_hidden, num_in) # (50,10)
b = np.random.randn(num_hidden) # (50,)

# forwad pass
a = np.matmul(W, x) + b # get affine output
h = np.maximum(a, np.zeros_like(a))
```

- This is what we're used to.
- It instantiates the following computational graph.



Neural network layer in NumPy

```python
import numpy as np
```

```python
num_in = 10
num_hidden = 50

# init numpy arrays
x = np.random.randn(num_in) # (10,)
W = np.random.randn(num_hidden, num_in) # (50,10)
b = np.random.randn(num_hidden) # (50,)

# forwad pass
a = np.matmul(W, x) + b # get affine output
h = np.maximum(a, np.zeros_like(a))
```

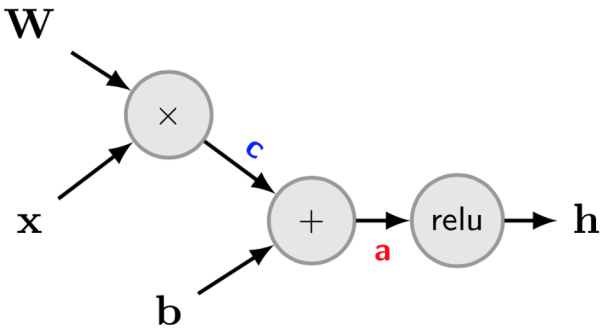
- We can think of each of these functions as implementing some forward pass that operates on inputs.
- e.g., the function `np.matmul(x,y)` has a defined forward pass that takes as input two matrices, `x` and `y`, and multiplies them together.

- But what about a backwards pass?

Neural network layer in NumPy

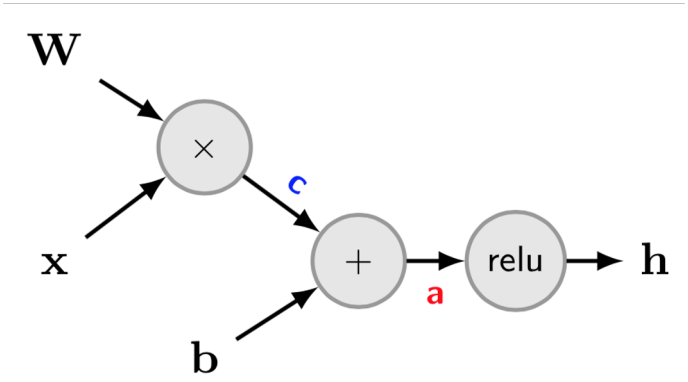
```
import numpy as np
```

```
num_in = 10
num_hidden = 50

# init numpy arrays
x = np.random.randn(num_in) # (10,)
W = np.random.randn(num_hidden, num_in) # (50,10)
b = np.random.randn(num_hidden) # (50,)

# forwad pass
a = np.matmul(W, x) + b # get affine output
h = np.maximum(a, np.zeros_like(a))
```



- As you're used to in the HW, in addition to writing the forward pass, you now have to write a backward pass

Neural network layer in NumPy

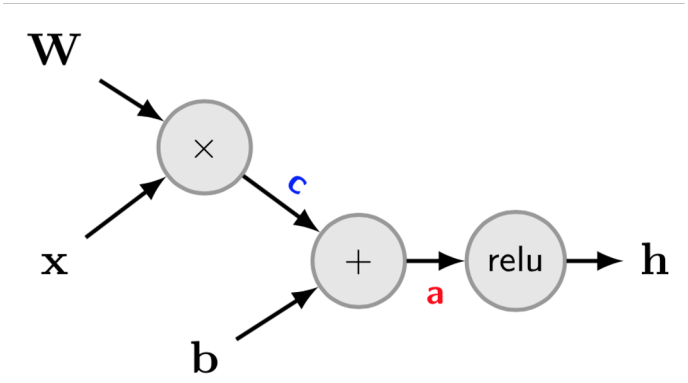


$$\frac{\partial \ell}{\partial \mathbf{a}} = \mathbb{I}(\mathbf{a} > 0) \odot \frac{\partial \ell}{\partial \mathbf{h}}$$
$$\frac{\partial \ell}{\partial \mathbf{c}} = \frac{\partial \mathbf{a}}{\partial \mathbf{c}} \frac{\partial \ell}{\partial \mathbf{a}} = \frac{\partial \ell}{\partial \mathbf{a}}$$
$$\frac{\partial \ell}{\partial \mathbf{x}} = \frac{\partial \mathbf{c}}{\partial \mathbf{x}} \frac{\partial \ell}{\partial \mathbf{c}} = \mathbf{W}^T \frac{\partial \ell}{\partial \mathbf{c}} = \mathbf{W}^T \frac{\partial \ell}{\partial \mathbf{a}}$$
$$\frac{\partial \ell}{\partial \mathbf{W}} = \frac{\partial \mathbf{c}}{\partial \mathbf{W}} \frac{\partial \ell}{\partial \mathbf{c}} = \frac{\partial \ell}{\partial \mathbf{c}} \mathbf{x}^T = \frac{\partial \ell}{\partial \mathbf{a}} \mathbf{x}^T$$

- Manually implement the backward pass.

```
# backward pass
dL = 1
da = np.expand_dims((a > 0) * dL, axis = 1) # (50,1)
dW = np.matmul(da, np.expand_dims(x, axis = 0)) # (50,1)*(1,10) = (50,10)
dx = np.matmul(W.T, da) # (10,50)*(50,1) = (10,1)
```

The computational graph



$$\frac{\partial \ell}{\partial \mathbf{a}} = \mathbb{I}(\mathbf{a} > 0) \odot \frac{\partial \ell}{\partial \mathbf{h}}$$

$$\frac{\partial \ell}{\partial \mathbf{c}} = \frac{\partial \mathbf{a}}{\partial \mathbf{c}} \frac{\partial \ell}{\partial \mathbf{a}} = \frac{\partial \ell}{\partial \mathbf{a}}$$

$$\frac{\partial \ell}{\partial \mathbf{x}} = \frac{\partial \mathbf{c}}{\partial \mathbf{x}} \frac{\partial \ell}{\partial \mathbf{c}} = \mathbf{W}^T \frac{\partial \ell}{\partial \mathbf{c}} = \mathbf{W}^T \frac{\partial \ell}{\partial \mathbf{a}}$$

$$\frac{\partial \ell}{\partial \mathbf{W}} = \frac{\partial \mathbf{c}}{\partial \mathbf{W}} \frac{\partial \ell}{\partial \mathbf{c}} = \frac{\partial \ell}{\partial \mathbf{c}} \mathbf{x}^T = \frac{\partial \ell}{\partial \mathbf{a}} \mathbf{x}^T$$

- Couldn't a software create the computational graph, store the local gradients, and thus calculate all the gradients with backpropagation for us?
- i.e., in addition to having a "forward" function, it has a "backward" function that accepts the upstream gradient, and multiplies it by the local gradient.
- This is fairly mechanical, and so we should be able to automate it.
- This is where deep learning libraries come in.

Implementation in PyTorch

```numpy
import numpy as np

num_in = 10
num_hidden = 50

# init numpy arrays
x = np.random.randn(num_in)
W = np.random.randn(num_hidden, num_in)
b = np.random.randn(num_hidden)

# forwad pass
a = np.matmul(W, x) + b # get affine output
h = np.maximum(a, np.zeros_like(a))
```

```pytorch
import torch

num_in = 10
num_hidden = 50

# init tensors
x = torch.randn(num_in)
W = torch.randn(num_hidden, num_in)
b = torch.randn(num_hidden)

# forwad pass
a = torch.matmul(W, x) + b # get affine output
h = torch.max(a, torch.zeros(num_hidden)) # relu
```

- In numpy, `x`, `W`, `b` are all numpy arrays.
- In PyTorch, these are called PyTorch Tensors, and they are declared via e.g., `torch.randn`, or `torch.zeros`, etc.
- A PyTorch Tensor is basically the same as a numpy array, but it can run on either CPU or GPU (casted to a `cuda` datatype).

Implementation in PyTorch

Further, it's easy to go between numpy arrays and PyTorch tensors.

```
import torch
import numpy as np

data = [[1, 2],[3, 4]]
np_array = np.array(data)

# numpy array to torch tensor
x_tensor = torch.from_numpy(np_array)

# torch tensor to numpy array
x_array =  x_tensor.numpy()

print('x_tensor is:', x_tensor)
print('x_array is', x_array)
```

```
x_tensor is: tensor([[1, 2],
        [3, 4]])
x_array is [[1 2]
 [3 4]]
```

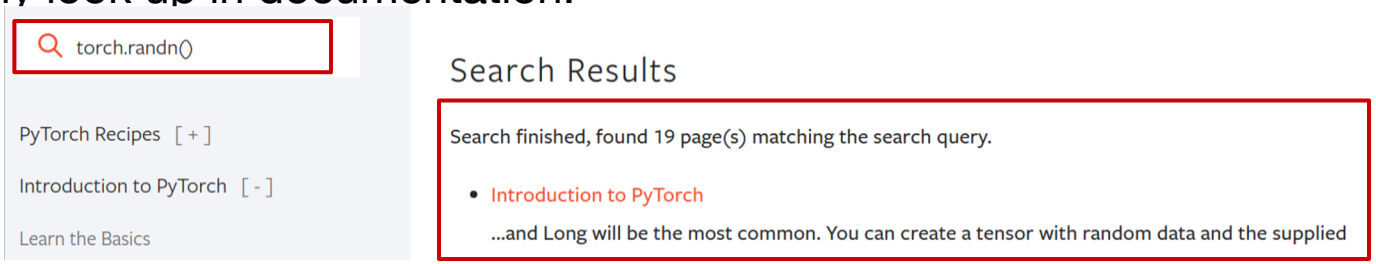
- Note: The torch Tensor and numpy array will share their **underlying memory** locations, and changing one will change the other!

Implementation in PyTorch

```
import torch
```

```
num_in = 10
num_hidden = 50

# init tensors
x = torch.randn(num_in)
W = torch.randn(num_hidden, num_in)
b = torch.randn(num_hidden)

# forwad pass
a = torch.matmul(W, x) + b # get affine output
h = torch.max(a, torch.zeros(num_hidden)) # relu
```

- Most of the operations that we have in numpy have PyTorch equivalents; usually look up in documentation.



- But now PyTorch also, for each of these functions, implements its **backward pass** so it can **automatically** do backprop and calculate gradients for you.

Implementation in PyTorch

```python
import torch
```

```python
num_in = 10
num_hidden = 50

# init tensors
x = torch.randn(num_in)
W = torch.randn(num_hidden, num_in)
b = torch.randn(num_hidden)

# forwad pass
a = torch.matmul(W, x) + b # get affine output
h = torch.max(a, torch.zeros(num_hidden)) # relu
```

```python
import torch
```

```python
num_in = 10
num_hidden = 50

# init tensors
x = torch.randn(num_in, requires_grad=True)
W = torch.randn(num_hidden, num_in, requires_grad=True)
b = torch.randn(num_hidden, requires_grad=True)

# forwad pass
a = torch.matmul(W, x) + b # get affine output
h = torch.max(a, torch.zeros(num_hidden)) # relu
```

- There is a core torch package called **`autograd`** for automatic differentiation.
- `requires_grad=True` signals that every operation on the tensor should be tracked.

Implementation in PyTorch

```
num_in = 10
num_hidden = 50

# init tensors with grad
x = torch.randn(num_in, requires_grad=True)
W = torch.randn(num_hidden, num_in, requires_grad=True)
b = torch.randn(num_hidden, requires_grad=True)

# forwad pass
a = torch.matmul(W, x) + b # get affine output
h = torch.max(a, torch.zeros(num_hidden)) # relu

print(x.data)
print(x.grad)
```

```
tensor([ 0.0156, -1.2956,  1.4077, -0.2591, -0.6188, -1.0828,  0.7685, -0.6054,
         1.2413,  1.0060])
None
```

- `x.data` contains the values of the PyTorch Tensor `x`.
- `x.grad` is another PyTorch **Tensor** containing the gradients of `x`.
- Therefore, `x.grad.data` contains the values of the gradients of `x`.

Implementation in PyTorch

```
num_in = 10
num_hidden = 50

# init tensors with grad
x = torch.randn(num_in, requires_grad=True)
W = torch.randn(num_hidden, num_in, requires_grad=True)
b = torch.randn(num_hidden, requires_grad=True)

# forwad pass
a = torch.matmul(W, x) + b # get affine output
h = torch.max(a, torch.zeros(num_hidden)) # relu

# calculate loss
loss = (torch.sum(h) - 1).pow(2)

# do backward
loss.backward()

print("x size: {}, x.grad size: {}".format(list(x.data.shape), list(x.grad.data.shape)))
print("W size: {}, W.grad size: {}".format(list(W.data.shape), list(W.grad.data.shape)))
print("b size: {}, b.grad size: {}".format(list(b.data.shape), list(b.grad.data.shape)))
```

```
x size: [10], x.grad size: [10]
W size: [50, 10], W.grad size: [50, 10]
b size: [50], b.grad size: [50]
```

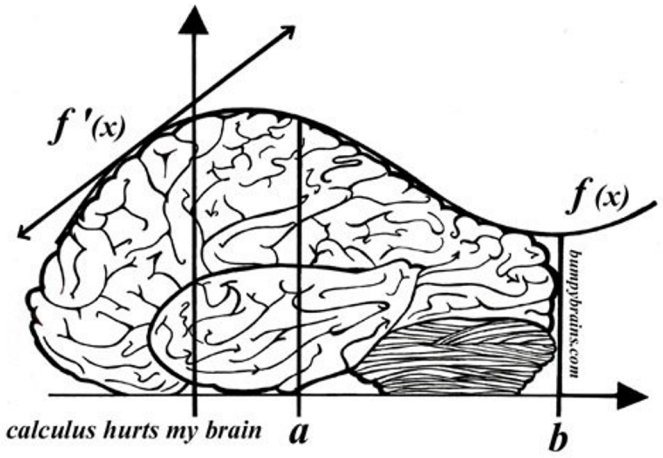
- "`.backward()`" takes the loss function and calculates the gradient for every variable that had a "`requires_grad=True`".
- This is one line code, i.e., the backward process required no cognitive effort!

Implementation in PyTorch

Why did we make you implement backpropagation?

It lets you know the call to "`.backward()`" is not magic:

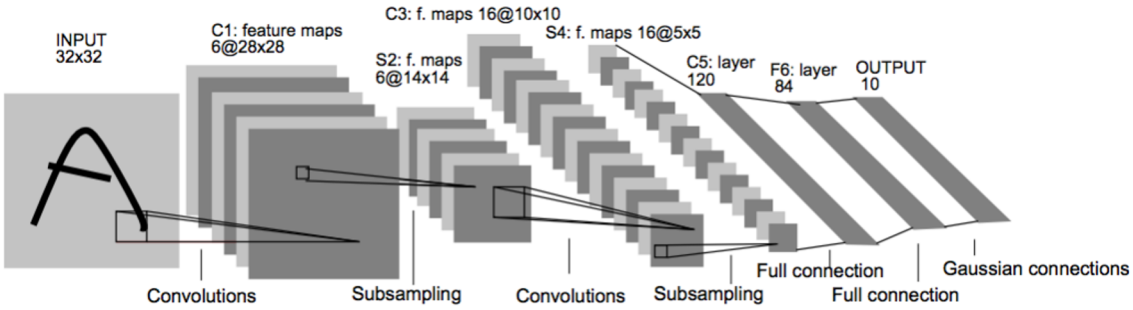
- making a computational graph
- knowing the local gradients at each node of the computational graph
- multiplying them by the upstream gradient

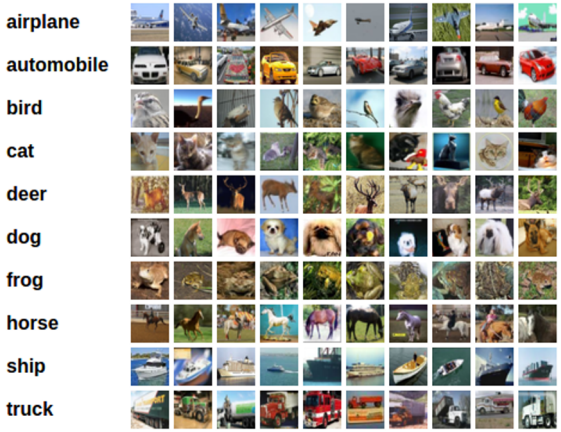


Example: an image classifier

Now with the basic ideas of PyTorch down, we'll go into detail about how to implement a simple neural network.

- Network: the LeNet-5 neural network architecture
- Dataset: CIFAR10, 10 classes, RGB images of size 3x32x32





Example: an image classifier

To learn an image classifier, we will do the following steps in order:

1. Load and normalize the CIFAR10 training and test datasets using `torchvision`
2. Define a Convolutional Neural Network
3. Define a loss function
4. Train the network on the training data
5. Test the network on the test data

Step 1: Load and normalize the dataset

Using `torchvision`, it's extremely easy to load CIFAR10.

```
import torch
import torchvision
import torchvision.transforms as transforms
```

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

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)

classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

Define the image transformation

Define the data loader

More details: https://pytorch.org/vision/stable/transforms.html

Step 2: Define a convolutional neural network

Modify the LeNet-5 neural network architecture to take 3-channel images

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

Import torch functions and modules

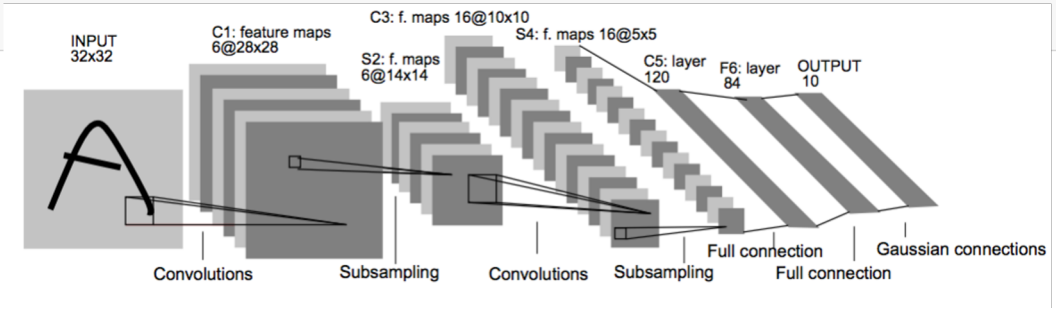
```
class Net(nn.Module):

    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)
```

Initialize the variables and network modules

```
    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)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

Define the forward pass

```
net = Net()
```



Step 2: Define a convolutional neural network

PyTorch incorporates **modules**, which are roughly equivalent to neural network layers.

- A **module** accepts Tensors as inputs and outputs.
- A **module** will also store internal Tensors with learnable weights.
- Examples of **modules** are `Conv2d`, `BatchNorm2d`, i.e., they implement the layers of a neural network.

The package `torch.nn` contains several modules that implement neural network layers, which you can compose together to make networks.

You can essentially think of the nn package as a high-level wrapper that makes it simple to write neural nets.

Step 2: Define a convolutional neural network

All documentation is available online, e.g.

CLASS `torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True, padding_mode='zeros', device=None, dtype=None)` [SOURCE]

Applies a 2D convolution over an input signal composed of several input planes.

Parameters

- **in_channels** (*int*) – Number of channels in the input image
- **out_channels** (*int*) – Number of channels produced by the convolution
- **kernel_size** (*int or tuple*) – Size of the convolving kernel
- **stride** (*int or tuple, optional*) – Stride of the convolution. Default: 1
- **padding** (*int, tuple or str, optional*) – Padding added to all four sides of the input. Default: 0
- **padding_mode** (*string, optional*) – `'zeros'`, `'reflect'`, `'replicate'` or `'circular'`. Default: `'zeros'`
- **dilation** (*int or tuple, optional*) – Spacing between kernel elements. Default: 1
- **groups** (*int, optional*) – Number of blocked connections from input channels to output channels. Default: 1
- **bias** (*bool, optional*) – If `True`, adds a learnable bias to the output. Default: `True`

More details: https://pytorch.org/docs/stable/nn.html

Step 2: Define a convolutional neural network

In practice, we can inspect the networks with python functions.

```
print(net)
```

```
Net(
  (conv1): Conv2d(3, 6, kernel_size=(5, 5), stride=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=400, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=10, bias=True)
)
```

The learnable parameters of a net are returned by `net.parameters()`.

```
params = list(net.parameters())
print(len(params))
print(params[0].size()) # conv1's .weight
```

```
10
torch.Size([6, 3, 5, 5])
```

Step 3: Define a loss function and optimizer

There are several different loss functions under the `nn` package:

- `nn.MSELoss`
- `nn.CrossEntropyLoss`

`torch.optim` is a package implementing various optimization algorithms:

- `optim.SGD`
- `optim.Adam`
- `optim.RMSprop`

Let's use a classification cross-entropy loss and SGD with momentum.

```
import torch.optim as optim

# define the cross-entropy loss
criterion = nn.CrossEntropyLoss()

# create the optimizer: SGD with momentum
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```

Step 4: Train the network

- We usually utilize scripts similar as follows for training and updating the net.
- We simply have to loop over our data iterator, and feed the inputs to the network and optimize.

```
for epoch in range(10): # 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

        # forward pass
        outputs = net(inputs)
        loss = criterion(outputs, labels)

        # backward + optimize
        loss.backward() # backward tp get gradient values
        optimizer.step() # does the update

        # zero the parameter gradients
        optimizer.zero_grad()

        # accumulate loss
        running_loss += loss.item()
```

Update net, get grad values, and zero grad

Step 4: Train the network

Question: where should I place `optimizer.zero_grad()`?

```
# zero the gradients
optimizer.zero_grad()

# forward pass
outputs = net(inputs)
loss = criterion(outputs, labels)

# backward + optimize
loss.backward() # backward
optimizer.step() # update
```

```
# forward pass
outputs = net(inputs)
loss = criterion(outputs, labels)

# zero the gradients
optimizer.zero_grad()

# backward + optimize
loss.backward() # backward
optimizer.step() # update
```

```
# forward pass
outputs = net(inputs)
loss = criterion(outputs, labels)

# backward + optimize
loss.backward() # backward
optimizer.step() # update

# zero the gradients
optimizer.zero_grad()
```

All approaches are valid for the standard use case, i.e. if you do not want to accumulate gradients for multiple iterations.

You can thus call `optimizer.zero_grad()` everywhere in the loop but not between the `loss.backward()` and `optimizer.step()` operation.

Step 5: Test the network on the test data

At test time, we set `torch.no_grad()` to disables gradient calculation.

Disabling gradient calculation is useful for inference, when you are sure that you will not call `Tensor.backward()`. It will reduce memory consumption for computations.

```
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}%')
```

Printing the summary of a model

Use the `torchsummary` package to print the summary of a model in PyTorch.

```
# !pip3 install torch-summary
from torchsummary import summary
summary(net, (3, 32, 32))
```

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
├─Conv2d: 1-1                            [-1, 6, 28, 28]           456
├─MaxPool2d: 1-2                         [-1, 6, 14, 14]           --
├─Conv2d: 1-3                            [-1, 16, 10, 10]          2,416
├─MaxPool2d: 1-4                         [-1, 16, 5, 5]            --
├─Linear: 1-5                            [-1, 120]                 48,120
├─Linear: 1-6                            [-1, 84]                  10,164
├─Linear: 1-7                            [-1, 10]                  850
==========================================================================================
Total params: 62,006
Trainable params: 62,006
Non-trainable params: 0
Total mult-adds (M): 0.65
==========================================================================================
Input size (MB): 0.01
Forward/backward pass size (MB): 0.05
Params size (MB): 0.24
Estimated Total Size (MB): 0.30
==========================================================================================
```

More details: https://github.com/sksq96/pytorch-summary

Saving and loading models

Save/Load `state_dict` (recommended)

- When saving a model for inference, it is only necessary to save the trained model's learned parameters.
- Remember that you must call `model.eval()` to set dropout and batch normalization layers to evaluation mode before running inference.

```
PATH = './cifar_net.pth'

# Save:
torch.save(net.state_dict(), PATH) # save state_dict

# Load:
model = Net()
model.load_state_dict(torch.load(PATH)) # load state_dict
model.eval() # sets model in evaluation (inference) mode
```

More details: https://pytorch.org/tutorials/beginner/saving_loading_models.html

Training on GPU

Define our device as the first visible cuda device if we have CUDA available:

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

# Assuming that we are on a CUDA machine, this should print a CUDA device:

print(device)
```

```
cuda:0
```

Transfer the data Tensor and the neural net onto the GPU:

```
net.to(device) # send the net model to GPU

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[0].to(device), data[1].to(device) # send data to GPU

        # zero the parameter gradients
        optimizer.zero_grad()

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

        # print statistics
        running_loss += loss.item()
```

More details: https://pytorch.org/tutorials/beginner/blitz/data_parallel_tutorial.html

Training on GPU

If you have CUDA available, you alternatively use `.cuda()`:

```
net.cuda() # send the net model to GPU

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[0].cuda(), data[1].cuda() # send data to GPU

        # zero the parameter gradients
        optimizer.zero_grad()

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

        # print statistics
        running_loss += loss.item()
```

More details: https://pytorch.org/tutorials/beginner/blitz/data_parallel_tutorial.html

How to adjust learning rate

`torch.optim.lr_scheduler` provides several methods to adjust the learning rate based on the number of epochs:

- `lr_scheduler.ExponentialLR`: decays learning rate by gamma every epoch
- `lr_scheduler.ReduceLROnPlateau`: reduce learning rate when a metric has stopped improving

Learning rate scheduling should be applied after optimizer's update.

```
optimizer = SGD(model, 0.1)
scheduler = ExponentialLR(optimizer, gamma=0.9) # Define the scheduler
for epoch in range(20):
    for input, target in dataset:
        optimizer.zero_grad()
        output = model(input)
        loss = loss_fn(output, target)
        loss.backward()
        optimizer.step()
    scheduler.step() # Learning rate scheduling
```

More details: https://pytorch.org/docs/stable/optim.html#how-to-adjust-learning-rate

Getting pre-trained models.

Import `torchvision` to get pre-defined models.

```
import torchvision

# load pre-defined models
alexnet = torchvision.models.alexnet(pretrained=False)

# modify the pre-defined model
my_resnet101 = torchvision.models.__dict__['resnet101'](pretrained=False, num_classes=10) # 10 classes
```

Import `torchvision` to get models with pre-trained parameters.

```
import torchvision

# load pre-pretained models
alexnet = torchvision.models.alexnet(pretrained=True)
vgg16 = torchvision.models.vgg16(pretrained=True)
resnet152 = torchvision.models.resnet152(pretrained=True)

# modify the pre-defined model
model = torchvision.models.resnet101(pretrained=True)
# get features after the last AvgPool
modules = list(model.children())[:-1] # output of pool5: [2048,1,1]
model = torch.nn.Sequential(*modules)
```

Build your module

You can combine multiple layers into one module:

```
class FCNet(nn.Module):

    def __init__(self, dims):
        super(FCNet, self).__init__()

        layers = []
        for i in range(len(dims)-2):
            in_dim = dims[i]
            out_dim = dims[i+1]
            layers.append(nn.Linear(in_dim, out_dim))
            layers.append(nn.ReLU())
            layers.append(nn.Dropout(p=0.5))
        layers.append(nn.Linear(dims[-2], dims[-1]))
        layers.append(nn.ReLU())
        layers.append(nn.Dropout(p=0.5))

        self.main = nn.Sequential(*layers)

    def forward(self, x):
        return self.main(x)


fcs = FCNet([16 * 5 * 5, 120, 84, 10])
print(fcs)
```

```
FCNet(
  (main): Sequential(
    (0): Linear(in_features=400, out_features=120, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=120, out_features=84, bias=True)
    (4): ReLU()
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=84, out_features=10, bias=True)
    (7): ReLU()
    (8): Dropout(p=0.5, inplace=False)
  )
)
```

Defining new autograd function

What if the function you want to implement is not in PyTorch?

You can write it yourself: defining the forward and backward pass, and then use it.

Example:

- A third order polynomial, trained to predict $y=g(x)$.

$$P_3(x) = \frac{1}{2}\left(5x^3 - 3x\right)$$

- We implement our own custom autograd function

$$P_3'(x) = \frac{3}{2}\left(5x^2 - 1\right)$$

More details: https://pytorch.org/tutorials/beginner/examples_autograd/two_layer_net_custom_function.html

Defining new autograd function

This implementation computes the forward pass using operations on PyTorch Tensors, and uses PyTorch autograd to compute gradients.

```
class LegendrePolynomial3(torch.autograd.Function):
    """
    We can implement our own custom autograd Functions by subclassing
    torch.autograd.Function and implementing the forward and backward passes
    which operate on Tensors.
    """

    @staticmethod
    def forward(ctx, input):
        """
        In the forward pass we receive a Tensor containing the input and return
        a Tensor containing the output. ctx is a context object that can be used
        to stash information for backward computation. You can cache arbitrary
        objects for use in the backward pass using the ctx.save_for_backward method.
        """
        ctx.save_for_backward(input)
        return 0.5 * (5 * input ** 3 - 3 * input)

    @staticmethod
    def backward(ctx, grad_output):
        """
        In the backward pass we receive a Tensor containing the gradient of the loss
        with respect to the output, and we need to compute the gradient of the loss
        with respect to the input.
        """
        input, = ctx.saved_tensors
        return grad_output * 1.5 * (5 * input ** 2 - 1)
```

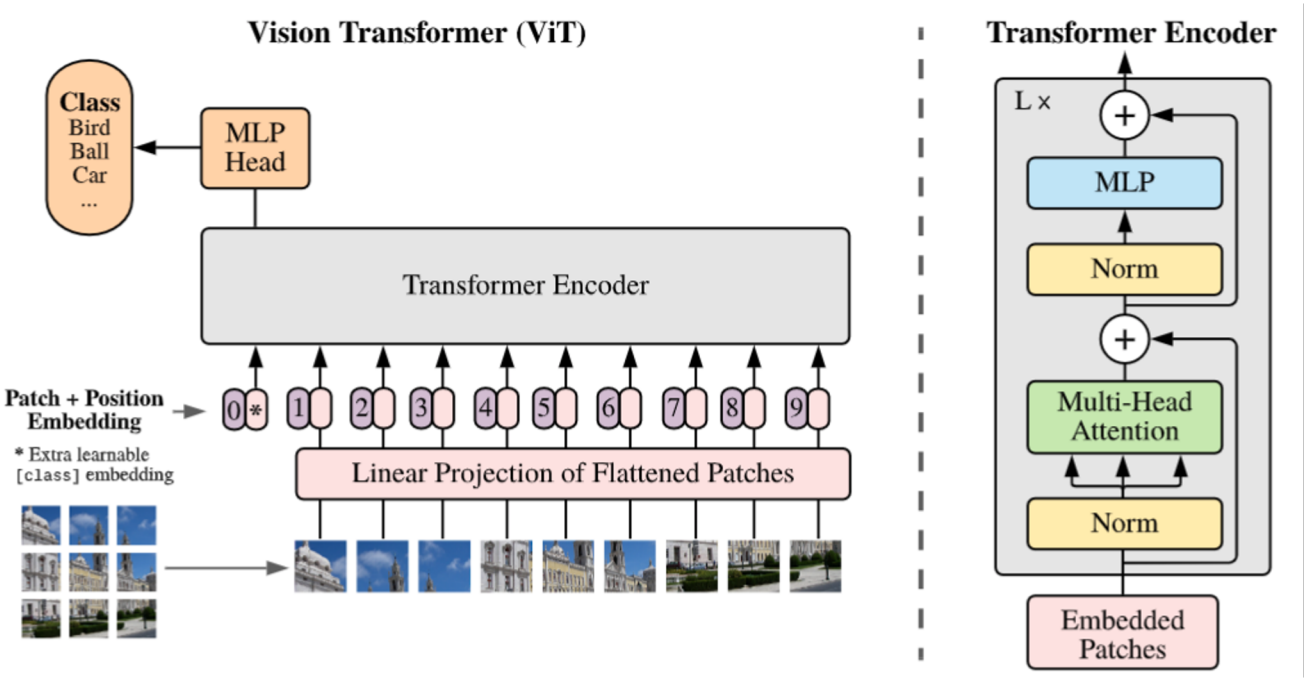
$$P_3(x) = \frac{1}{2}\left(5x^3 - 3x\right)$$

$$P_3'(x) = \frac{3}{2}\left(5x^2 - 1\right)$$

PyTorch Package: Hugging Face

Pretrained models are the state-of-the-art deep learning models.

Hugging Face provides thousands of pretrained models to perform tasks on different modalities such as text, vision, and audio.

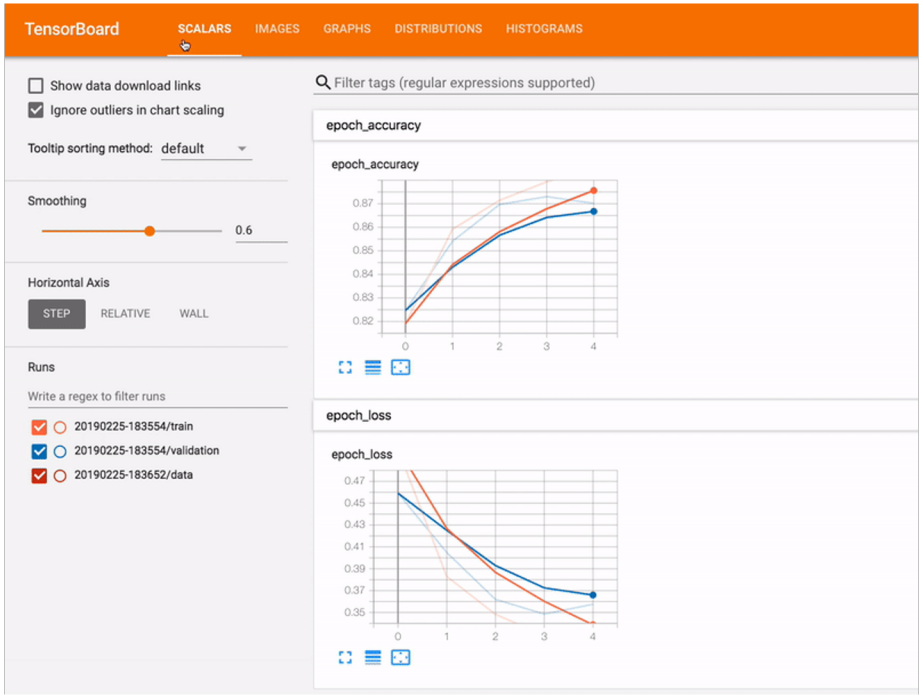


More details: https://huggingface.co/docs/transformers

TensorBoard

Visualization toolkit

- Metrics
- Model graphs
- Histograms of weights, biases
- Projecting embeddings
- Displaying images, text
- And much more



More details: https://pytorch.org/tutorials/recipes/recipes/tensorboard_with_pytorch.html

GPU Tool

Monitor the GPU status: `watch nvidia-smi`

```
Every 2.0s: nvidia-smi

Sun Feb 20 11:31:33 2022
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 470.103.01   Driver Version: 470.103.01   CUDA Version: 11.4     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  NVIDIA TITAN X ...  Off  | 00000000:01:00.0  On |                  N/A |
| 34%   59C    P0    58W / 250W |    702MiB / 12192MiB |      1%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   1  NVIDIA GeForce ...  Off  | 00000000:02:00.0 Off |                  N/A |
|  0%   34C    P8     4W / 350W |     10MiB / 24268MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|    0   N/A  N/A      1689      G   ...zoom-client/168/zoom/zoom       34MiB |
|    0   N/A  N/A      6696      G   /usr/lib/xorg/Xorg                 35MiB |
```

Take-away points

- PyTorch is one of the best deep learning frameworks.
- Pytorch official tutorials
  - https://pytorch.org/tutorials/
  - Feel free to check out the tutorial for more details!
- Useful packages
  - CV: `torchvision`, OpenMMLab
  - NLP: `torchtext`, Hugging Face, fairseq, AllenNLP
  - Audio: `torchaudio`
  - Tools: TensorBoard, `nvidia-smi`, `htop`
- Some potentially helpful resources:
  - UCB CS285 PyTorch Tutorial: https://www.youtube.com/watch?v=kPa6hU9prg4
  - PyTorch for Deep Learning: https://www.youtube.com/watch?v=GIsg-ZUy0MY
  - Deep Learning With PyTorch: https://www.youtube.com/watch?v=c36lUUr864M
  - https://d2l.ai/index.html
  - https://github.com/bharathgs/Awesome-pytorch-list

Thanks!