# COMPSCI 389
# Introduction to Machine Learning

**Classification Example**

Prof. Philip S. Thomas (pthomas@cs.umass.edu)

Note: This presentation covers (and provides additional context/information regarding) `23 Classification Example.ipynb`

# CIFAR-100

- Produced by the *Canadian Institute For Advances Research* (CIFAR).
- One of the most widely used data sets for testing ML methods for computer vision.
- Contains 60,000 images
  - Each 32x32 pixels (color!)
- Two variants, CIFAR-10 (10 classes) and CIFAR-100 (100 classes)
  - CIFAR-10 has 6,000 images of each class.
  - CIFAR-100 has 600 images of each class.
- CIFAR-10 classes: airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks

# Loading CIFAR-100

- When installing PyTorch we installed `torchvision` along with `torch`.
  - This includes methods for loading common data sets like CIFAR-100

```
trainset = torchvision.datasets.CIFAR100(root='./data', train=True,
                                          download=True, transform=transform)
```

- This downloads the training data to a “data” subdirectory.
- The provided transforms modify the original data slightly.

```
trainset = torchvision.datasets.CIFAR100(root='./data', train=True,
                                         download=True, transform=transform)
```

# Loading CIFAR-100

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

> transforms.ToTensor() converts the images, represented as NumPy arrays, into PyTorch tensors. It scales pixel intensities from $[0, 255]$ to $[0.0, 1.0]$, and changes the order of dimensions from (heigth, width, channels) to (channels, height, width).

> transforms.Normalize adjusts the color channels of the images (now tensors), performing a form of normalization. It re-scales the red, green, and blue channels from $[0, 1]$ to $[-1, 1]$. The first argument, (0.5, 0.5, 0.5), indicates that $0.5$ will be subtracted from each channel (red, green, and blue). The second argument, also $(0.5, 0.5, 0.5)$ indicates that each channel should be divided by $0.5$ (i.e., mulitplied by two).

# Loading CIFAR-100

```
trainset = torchvision.datasets.CIFAR100(root='./data', train=True,
                                         download=True, transform=transform)
```

```
testset = torchvision.datasets.CIFAR100(root='./data', train=False,
                                        download=True, transform=transform)
```

- Create data loaders (each with 2 threads).

```
trainloader = torch.utils.data.DataLoader(trainset, batch_size=16,
                                          shuffle=True, num_workers=2)
testloader = torch.utils.data.DataLoader(testset, batch_size=16,
                                         shuffle=False, num_workers=2)
```

You can experiment with different batch sizes!

Conv2d represents a convolutional layer for a 2-dimensional image.

The first argument is the number of channels (3 for red, green, blue)

The second argument is the number of filters (output channels)

The third argument is the patch size (kernel size)

```
class Net(nn.Module):
    def __init__(self):
        super(Net, 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, 100)

    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
```

You can experiment with different network architectures!

Conv2d represents a convolutional layer for a 2-dimensional image.

The first argument is the number of channels (3 for red, green, blue)

The second argument is the number of filters (output channels)

The third argument is the patch size (kernel size)

```
class Net(nn.Module):
    def __init__(self):
        super(Net, 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, 100)

    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
```

# Prepare for Training (nothing new!)

Create the network.

```python
net = Net()
```

Python

Select the loss function and optimizer.

You can experiment with different optimizers! Changing this to:

```python
optimizer = optim.Adam(net.parameters(), lr=0.001)
```

resulted in lower accuracy.

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

Python

Set up for GPU training:

```python
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
display(device)
net.to(device)
```

Python

```
device(type='cuda', index=0)
```

9

# Train the network (nothing new!)

```
for epoch in range(100):  # 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)

        # 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
```

This took 23 minutes on my RTX 2070

The code prints the loss every 2000 mini-batches within each epoch. With smaller batches this will have multiple lines per epoch:
[1, 2000]
[1, 4000]
...
[2, 2000]
[2, 4000]
...

```
[1,   2000] loss: 4.601
[2,   2000] loss: 4.143
[3,   2000] loss: 3.808
[4,   2000] loss: 3.555
[5,   2000] loss: 3.355
[6,   2000] loss: 3.219
[7,   2000] loss: 3.101
[8,   2000] loss: 3.005
[9,   2000] loss: 2.907
[10,  2000] loss: 2.842
[11,  2000] loss: 2.777
[12,  2000] loss: 2.714
[13,  2000] loss: 2.672
[14,  2000] loss: 2.622
[15,  2000] loss: 2.578
[16,  2000] loss: 2.525
[17,  2000] loss: 2.498
[18,  2000] loss: 2.446
[19,  2000] loss: 2.414
[20,  2000] loss: 2.394
[21,  2000] loss: 2.357
[22,  2000] loss: 2.320
[23,  2000] loss: 2.292
[24,  2000] loss: 2.258
[25,  2000] loss: 2.226
...
[97,  2000] loss: 1.549
[98,  2000] loss: 1.533
[99,  2000] loss: 1.526
[100,  2000] loss: 1.539
```

# Evaluate (nothing new!)

```python
correct = 0
total = 0
with torch.no_grad():
    for data in testloader:
        inputs, labels = data[0].to(device), data[1].to(device)
        outputs = net(inputs)
        _, 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} %')
```

Python

```
Accuracy of the network on the 10000 test images: 25 %
```

# Is 25% accuracy good?

- Significantly better than random (1% accuracy)
- Far worse than state of the art (~80% in 2010, high-90% today)

# End