Machine Learning, Spring 2020

PyTorch Tutorial Part II

How to Build Your Deep Learning Project from Scratch

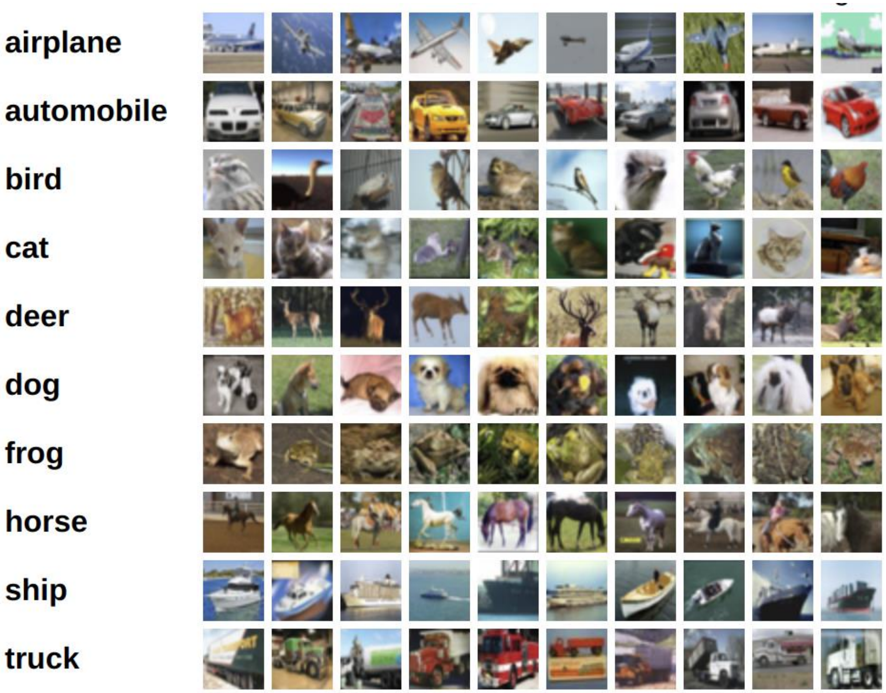
Yunxiao

What We Will Cover Today

- Build an image classifier by implementing LeNet
- Train and evaluate our model on the CIFAR-10 dataset
- Utilities to monitor training, error analysis and visualization

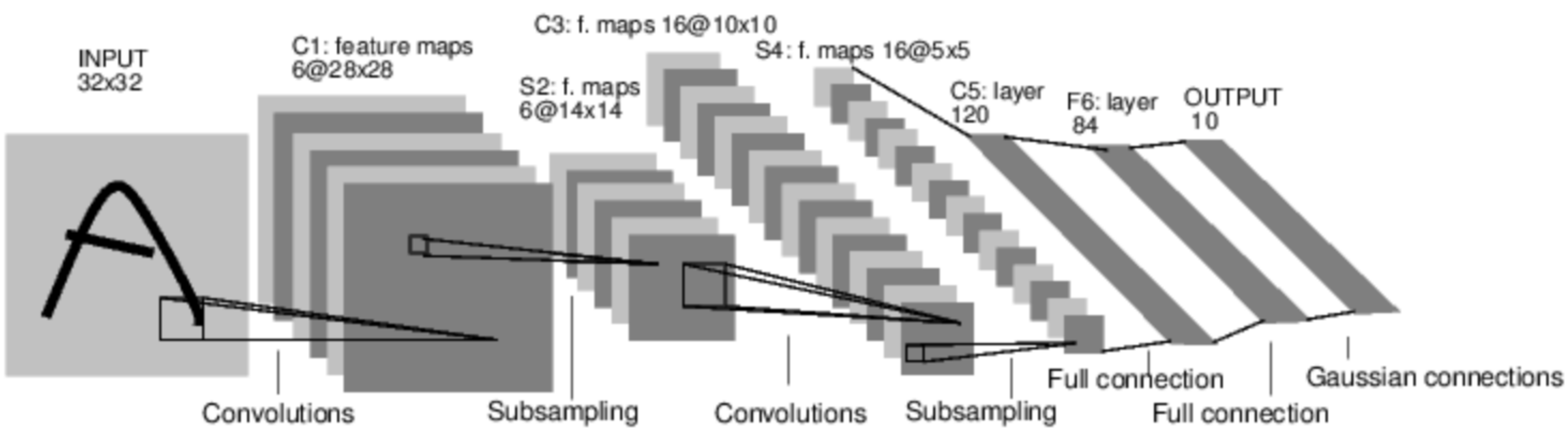
CIFAR-10 Dataset

- The CIFAR-10 dataset is a labeled subset of 80 million tiny images
- CIFAR-10 contains 60,000 32x32x3 color images in 10 classes, with 6,000 images per class
- 50,000 images for training and the rest 10,000 for testing



LeNet

- LeNet was developed by Yann Lecun *et. al* in the 1990s to classify hand-written digits
- The first Convolutional Neural Network (CNN) used on a large scale
- Very simple architecture (by today's standard)



LeNet on CIFAR-10

To build a deep learning (computer vision) project,

1. Know your data well – customize your own dataset class (torch.utils.data.Dataset)

2. Know your model well – implement your model correctly (torch.nn)

3. Design task-specfic loss function (torch.nn), optimizer (torch.optim)

4. Training

5. Evaluation

Customize your own dataset

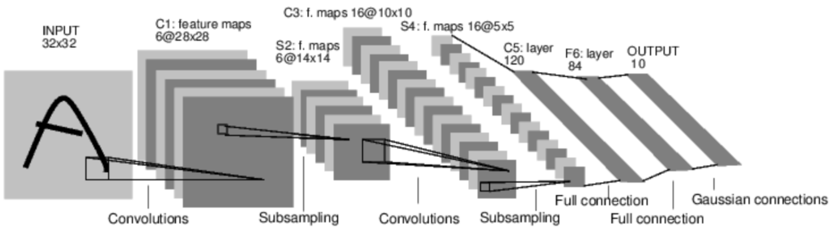
- Our customized dataset should be able to correctly return the training instance and its corresponding label when looping over the actual data

- We do this by implementing the following three special Python methods:
  - __init__()
  - __getitem__()
  - __len__()

```python
class CIFAR10:

    def __init__(self, root, train=True, transform=None):
        # your code
        pass


    def __getitem__(self, index):
        # your code
        pass


    def __len__(self):
        # your code
        pass
```

Implement LeNet

- LeNet's has a simple structure: 3 convs, 2 pools, and 2 fcs
- We implement our LeNet class by inheriting PyTorch's nn.Module class and overload its forward() method




```python
class LeNet(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, 10)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.pool(self.relu(self.conv1(x)))
        x = self.pool(self.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

Design Task-Specific Loss Function and Optimizer

- Have a clear mind about the task in front of you. Classification? Regression?
- We are dealing with a typical single-input, multi-class classification problem
- Cross-entropy loss is the de-facto choice
- We can easily implement the cross-entropy loss by leveraging PyTorch's torch.nn class
- Constructing the optimizer is just as easy by using torch.optim

```
# PyTorch has already implemented many common loss functions for us
criterion = nn.CrossEntropyLoss()
# Construct an optimizer is very easy in PyTorch
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```

Train Your Model

- To train a model in a supervised learning setting (like ours), it typically consists of following steps:
  1. Loop over training set in batches to inputs
  2. Zero the parameter gradients
  3. Forward pass the inputs to our model to get predictions
  4. Compute loss
  5. Back-propagate to update model parameters

```
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
        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+1) % 2000 == 0:    # print every 2000 mini-batches
            print('[%d, %5d] loss: %.3f' %
                  (epoch+1, i+1, running_loss/2000.))
            running_loss = 0.0
```

Evaluation

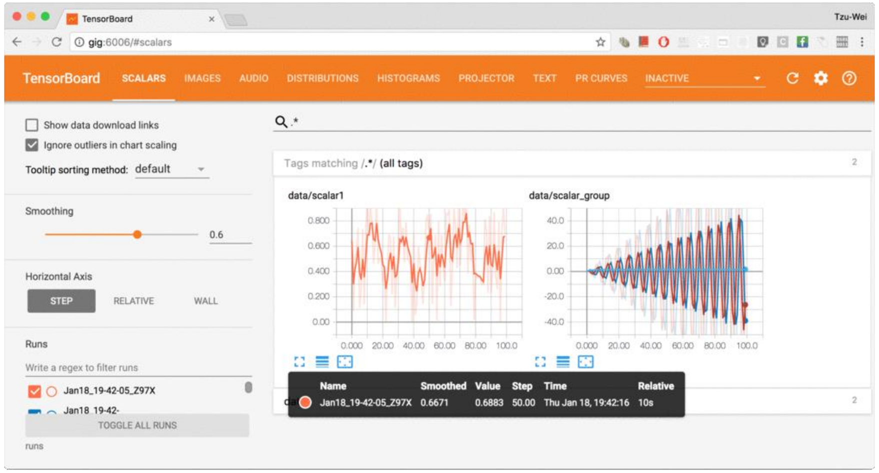
- Forward pass test set inputs to the trained model to get predictions
- Compute error metrics / fitness metrics against labels
- For our task we compute how many test cases are correctly classified to get model accuracy

```
correct = 0
total = 0
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

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

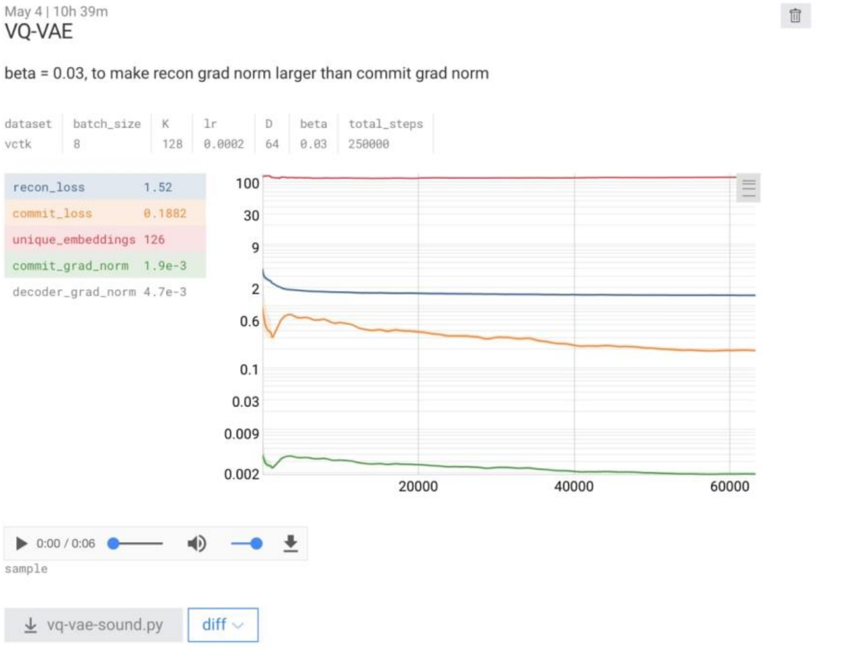
Utilities to Monitor Training

- Have a visualized interface to monitor training is helpful (loss curve, accuracy curve)
- Helps us know for instance if the training plateaus (learning rate decay)
- There are utilities out there that help monitor training in PyTorch

tensorboardX



lera.ai



Error Analysis and Visualization

- Inevitably our model will make mistakes on unseen test data
- For problems like image classification, we can conduct statistics of model decisions to have possible diagnosis
- Plot intermediate feature maps is another way to see if our model is learning properly

```
class_correct = list(0. for i in range(10))
class_total = list(0. for i in range(10))
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs, 1)
        c = (predicted == labels).squeeze()
        for i in range(4):
            label = labels[i]
            class_correct[label] += c[i].item()
            class_total[label] += 1


for i in range(10):
    print('Accuracy of %5s : %2d %%' % (classes[i], 100 * class_correct[i] / class_total[i]))
```

References

LeNet paper, Yann Lecun et al.

CIFAR-10 Dataset

PyTorch Docs

tensorboardX

lera.ai