WEEK4: Convolution Neural Network Tutorial

Department of Computer Science and Information Engineering
Ubiquitous Sensing and Cloud Computing Lab

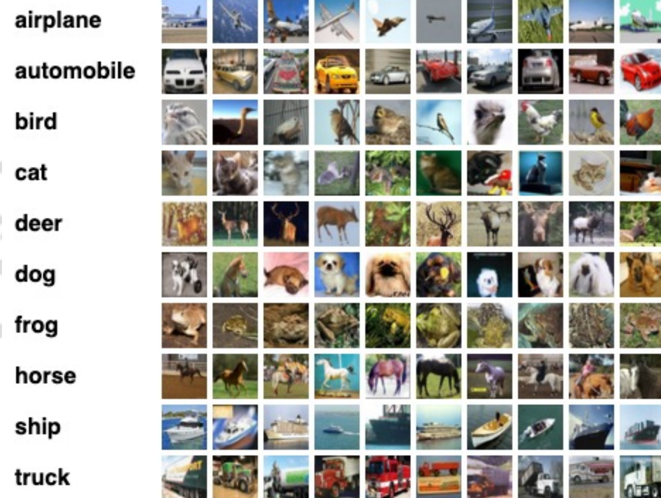
Training an image classifier

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

Cifar10

- The CIFAR-10 dataset consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images.



Load and normalize CIFAR10

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

```
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)
```

CONV2D

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.

In the simplest case, the output value of the layer with input size (N, C_{in}, H, W) and output $(N, C_{\text{out}}, H_{\text{out}}, W_{\text{out}})$ can be precisely described as:

$$\text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) + \sum_{k=0}^{C_{\text{in}}-1} \text{weight}(C_{\text{out}_j}, k) \star \text{input}(N_i, k)$$

where $\star$ is the valid 2D cross-correlation operator, N is a batch size, C denotes a number of channels, H is a height of input planes in pixels, and W is width in pixels.

Examples

```
>>> # With square kernels and equal stride
>>> m = nn.Conv2d(16, 33, 3, stride=2)
>>> # non-square kernels and unequal stride and with padding
>>> m = nn.Conv2d(16, 33, (3, 5), stride=(2, 1), padding=(4, 2))
>>> # non-square kernels and unequal stride and with padding and dilation
>>> m = nn.Conv2d(16, 33, (3, 5), stride=(2, 1), padding=(4, 2), dilation=(3, 1))
>>> input = torch.randn(20, 16, 50, 100)
>>> output = m(input)
```

MAXPOOL2D

CLASS `torch.nn.MaxPool2d(kernel_size, stride=None, padding=0, dilation=1, return_indices=False, ceil_mode=False)` [SOURCE]

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

In the simplest case, the output value of the layer with input size (N, C, H, W), output (N, C, H_{out}, W_{out}) and `kernel_size` (kH, kW) can be precisely described as:

$$out(N_i, C_j, h, w) = \max_{m=0,\ldots,kH-1} \max_{n=0,\ldots,kW-1} \text{input}(N_i, C_j, \text{stride}[0] \times h + m, \text{stride}[1] \times w + n)$$

If `padding` is non-zero, then the input is implicitly padded with negative infinity on both sides for `padding` number of points. `dilation` controls the spacing between the kernel points. It is harder to describe, but this link has a nice visualization of what `dilation` does.

Examples:

```
>>> # pool of square window of size=3, stride=2
>>> m = nn.MaxPool2d(3, stride=2)
>>> # pool of non-square window
>>> m = nn.MaxPool2d((3, 2), stride=(2, 1))
>>> input = torch.randn(20, 16, 50, 32)
>>> output = m(input)
```

TORCH.FLATTEN

`torch.flatten(input, start_dim=0, end_dim=- 1) → Tensor`

Flattens `input` by reshaping it into a one-dimensional tensor. If `start_dim` or `end_dim` are passed, only dimensions starting with `start_dim` and ending with `end_dim` are flattened. The order of elements in `input` is unchanged.

Unlike NumPy's flatten, which always copies input's data, this function may return the original object, a view, or copy. If no dimensions are flattened, then the original object `input` is returned. Otherwise, if input can be viewed as the flattened shape, then that view is returned. Finally, only if the input cannot be viewed as the flattened shape is input's data copied. See `torch.Tensor.view()` for details on when a view will be returned.

Example:

```
>>> t = torch.tensor([[[1, 2],
...                    [3, 4]],
...                   [[5, 6],
...                    [7, 8]]])
>>> torch.flatten(t)
tensor([1, 2, 3, 4, 5, 6, 7, 8])
>>> torch.flatten(t, start_dim=1)
tensor([[1, 2, 3, 4],
        [5, 6, 7, 8]])
```

Build Convolution Neural Network

```
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.network = nn.Sequential(
            nn.Conv2d(in_channels=3, out_channels=6, kernel_size=3, stride=2, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),
        )

        self.linear = nn.Sequential(
            nn.Linear(in_features=6 * 8 * 8, out_features=72),
            nn.ReLU(),
            nn.Linear(in_features=72, out_features=10),
        )
    def forward(self, x):
        x = self.network(x)
        # print(x.shape)
        x = torch.flatten(x, 1)
        return self.linear(x)
```

Training on GPU

Just like how you transfer a Tensor onto the GPU, you transfer the neural net onto the GPU.

Let's first 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)
```

Out:

```
cuda:0
```

Training on GPU

The rest of this section assumes that `device` is a CUDA device.

Then these methods will recursively go over all modules and convert their parameters and buffers to CUDA tensors:

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

Remember that you will have to send the inputs and targets at every step to the GPU too:

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

Tasks

Tasks

1. Implement your Convolution Neural Network.
2. Train your Neural Network on GPU.
3. Try to use different hyperparameters(e.g. batch_size, epoch), you need to have three different results.
4. Write your experimentation result on Google Docs or Word and upload to moodle `WEEK4_EXPERIMENTATION`
    - include:
        - Convolution Neural Network code
        - difficulties which you encountered
        - Produce different results by adjust hyperparameters, loss or transform(at least 3 results)
        - Conclusion or your thoughts(at least 100 words)

https://colab.research.google.com/drive/1Y1fkJHFm4SwXnAX45hDkIEew2BwYZspP?usp=sharing

Reference

- https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html
- https://pytorch.org/tutorials/beginner/finetuning_torchvision_models_tutorial.html
- https://pytorch.org/vision/stable/transforms.html