Lab 10

Aim: To implement ResNet101 CNN Architecture

```
import os
import time

import numpy as np
import pandas as pd

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

from torch.utils.data import Dataset
from torch.utils.data import DataLoader
from torch.utils.data.dataset import Subset

from torchvision import datasets
from torchvision import transforms

import time

import matplotlib.pyplot as plt
from PIL import Image


if torch.cuda.is_available():
    torch.backends.cudnn.deterministic = True

# Hyperparameters
RANDOM_SEED = 1
LEARNING_RATE = 0.01
NUM_EPOCHS = 25

# Architecture
NUM_CLASSES = 10
BATCH_SIZE = 128
DEVICE = 'cuda' if torch.cuda.is_available() else 'cpu'
GRAYSCALE = False

train_indices = torch.arange(0, 49000)
valid_indices = torch.arange(49000, 50000)


train_and_valid = datasets.CIFAR10(root='data',
                                   train=True,
                                   transform=transforms.ToTensor(),
```

```
                                   download=True)

train_dataset = Subset(train_and_valid, train_indices)
valid_dataset = Subset(train_and_valid, valid_indices)


test_dataset = datasets.CIFAR10(root='data',
                                train=False,
                                transform=transforms.ToTensor())


train_loader = DataLoader(dataset=train_dataset,
                          batch_size=BATCH_SIZE,
                          num_workers=8,
                          shuffle=True)

valid_loader = DataLoader(dataset=valid_dataset,
                          batch_size=BATCH_SIZE,
                          num_workers=8,
                          shuffle=False)

test_loader = DataLoader(dataset=test_dataset,
                         batch_size=BATCH_SIZE,
                         num_workers=8,
                         shuffle=False)

# Checking the dataset
for images, labels in train_loader:
    print('Image batch dimensions:', images.shape)
    print('Image label dimensions:', labels.shape)
    break

for images, labels in test_loader:
    print('Image batch dimensions:', images.shape)
    print('Image label dimensions:', labels.shape)
    break

for images, labels in valid_loader:
    print('Image batch dimensions:', images.shape)
    print('Image label dimensions:', labels.shape)
    break
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to
data/cifar-10-python.tar.gz

100%|██████████| 170498071/170498071 [00:13<00:00, 12606907.54it/s]

Extracting data/cifar-10-python.tar.gz to data

/usr/local/lib/python3.10/dist-packages/torch/utils/data/
dataloader.py:558: UserWarning: This DataLoader will create 8 worker
```

```
processes in total. Our suggested max number of worker in current
system is 2, which is smaller than what this DataLoader is going to
create. Please be aware that excessive worker creation might get
DataLoader running slow or even freeze, lower the worker number to
avoid potential slowness/freeze if necessary.
  warnings.warn(_create_warning_msg(
```

```
Image batch dimensions: torch.Size([128, 3, 32, 32])
Image label dimensions: torch.Size([128])
Image batch dimensions: torch.Size([128, 3, 32, 32])
Image label dimensions: torch.Size([128])
Image batch dimensions: torch.Size([128, 3, 32, 32])
Image label dimensions: torch.Size([128])
```

```
def conv3x3(in_planes, out_planes, stride=1):
    """3x3 convolution with padding"""
    return nn.Conv2d(in_planes, out_planes, kernel_size=3,
stride=stride,
                     padding=1, bias=False)


class Bottleneck(nn.Module):
    expansion = 4

    def __init__(self, inplanes, planes, stride=1, downsample=None):
        super(Bottleneck, self).__init__()
        self.conv1 = nn.Conv2d(inplanes, planes, kernel_size=1,
bias=False)
        self.bn1 = nn.BatchNorm2d(planes)
        self.conv2 = nn.Conv2d(planes, planes, kernel_size=3,
stride=stride,
                               padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(planes)
        self.conv3 = nn.Conv2d(planes, planes * 4, kernel_size=1,
bias=False)
        self.bn3 = nn.BatchNorm2d(planes * 4)
        self.relu = nn.ReLU(inplace=True)
        self.downsample = downsample
        self.stride = stride

    def forward(self, x):
        residual = x

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)
        out = self.relu(out)
```

```
        out = self.conv3(out)
        out = self.bn3(out)

        if self.downsample is not None:
            residual = self.downsample(x)

        out += residual
        out = self.relu(out)

        return out


class ResNet(nn.Module):

    def __init__(self, block, layers, num_classes, grayscale):
        self.inplanes = 64
        if grayscale:
            in_dim = 1
        else:
            in_dim = 3
        super(ResNet, self).__init__()
        self.conv1 = nn.Conv2d(in_dim, 64, kernel_size=7, stride=2,
padding=3,
                               bias=False)
        self.bn1 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2,
padding=1)
        self.layer1 = self._make_layer(block, 64, layers[0])
        self.layer2 = self._make_layer(block, 128, layers[1],
stride=2)
        self.layer3 = self._make_layer(block, 256, layers[2],
stride=2)
        self.layer4 = self._make_layer(block, 512, layers[3],
stride=2)
        self.avgpool = nn.AvgPool2d(7, stride=1, padding=2)
        #self.fc = nn.Linear(2048 * block.expansion, num_classes)
        self.fc = nn.Linear(2048, num_classes)

        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                n = m.kernel_size[0] * m.kernel_size[1] *
m.out_channels
                m.weight.data.normal_(0, (2. / n)**.5)
            elif isinstance(m, nn.BatchNorm2d):
                m.weight.data.fill_(1)
                m.bias.data.zero_()

    def _make_layer(self, block, planes, blocks, stride=1):
```

```python
        downsample = None
        if stride != 1 or self.inplanes != planes * block.expansion:
            downsample = nn.Sequential(
                nn.Conv2d(self.inplanes, planes * block.expansion,
                          kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(planes * block.expansion),
            )

        layers = []
        layers.append(block(self.inplanes, planes, stride,
downsample))
        self.inplanes = planes * block.expansion
        for i in range(1, blocks):
            layers.append(block(self.inplanes, planes))

        return nn.Sequential(*layers)

    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.maxpool(x)

        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)

        #x = self.avgpool(x)
        x = x.view(x.size(0), -1)
        logits = self.fc(x)
        probas = F.softmax(logits, dim=1)
        return logits, probas



def resnet101(num_classes, grayscale):
    """Constructs a ResNet-101 model."""
    model = ResNet(block=Bottleneck,
                   layers=[3, 4, 23, 3],
                   num_classes=NUM_CLASSES,
                   grayscale=grayscale)
    return model

torch.manual_seed(RANDOM_SEED)
model = resnet101(NUM_CLASSES, GRAYSCALE)
model.to(DEVICE)

optimizer = torch.optim.Adam(model.parameters(), lr=LEARNING_RATE)
```

```python
def compute_accuracy(model, data_loader, device):
    correct_pred, num_examples = 0, 0
    for i, (features, targets) in enumerate(data_loader):

        features = features.to(device)
        targets = targets.to(device)

        logits, probas = model(features)
        _, predicted_labels = torch.max(probas, 1)
        num_examples += targets.size(0)
        correct_pred += (predicted_labels == targets).sum()
    return correct_pred.float()/num_examples * 100


start_time = time.time()

# use random seed for reproducibility (here batch shuffling)
torch.manual_seed(RANDOM_SEED)

for epoch in range(NUM_EPOCHS):

    model.train()

    for batch_idx, (features, targets) in enumerate(train_loader):

        ### PREPARE MINIBATCH
        features = features.to(DEVICE)
        targets = targets.to(DEVICE)

        ### FORWARD AND BACK PROP
        logits, probas = model(features)
        cost = F.cross_entropy(logits, targets)
        optimizer.zero_grad()

        cost.backward()

        ### UPDATE MODEL PARAMETERS
        optimizer.step()

        ### LOGGING
        if not batch_idx % 120:
            print (f'Epoch: {epoch+1:03d}/{NUM_EPOCHS:03d} | '
                   f'Batch {batch_idx:03d}/{len(train_loader):03d} |'
                   f' Cost: {cost:.4f}')

    # no need to build the computation graph for backprop when
computing accuracy
    with torch.set_grad_enabled(False):
        train_acc = compute_accuracy(model, train_loader,
device=DEVICE)
        valid_acc = compute_accuracy(model, valid_loader,
```

```
device=DEVICE)
        print(f'Epoch: {epoch+1:03d}/{NUM_EPOCHS:03d} Train Acc.:
{train_acc:.2f}%'
              f' | Validation Acc.: {valid_acc:.2f}%')

    elapsed = (time.time() - start_time)/60
    print(f'Time elapsed: {elapsed:.2f} min')

elapsed = (time.time() - start_time)/60
print(f'Total Training Time: {elapsed:.2f} min')
```

```
Epoch: 001/025 | Batch 000/383 | Cost: 2.7585
Epoch: 001/025 | Batch 120/383 | Cost: 3.7965
Epoch: 001/025 | Batch 240/383 | Cost: 2.2732
Epoch: 001/025 | Batch 360/383 | Cost: 2.0641
Epoch: 001/025 Train Acc.: 29.08% | Validation Acc.: 30.80%
Time elapsed: 1.30 min
Epoch: 002/025 | Batch 000/383 | Cost: 1.8997
Epoch: 002/025 | Batch 120/383 | Cost: 1.6565
Epoch: 002/025 | Batch 240/383 | Cost: 1.6737
Epoch: 002/025 | Batch 360/383 | Cost: 1.7384
Epoch: 002/025 Train Acc.: 41.41% | Validation Acc.: 41.20%
Time elapsed: 2.57 min
Epoch: 003/025 | Batch 000/383 | Cost: 1.4844
Epoch: 003/025 | Batch 120/383 | Cost: 1.5908
Epoch: 003/025 | Batch 240/383 | Cost: 1.4972
Epoch: 003/025 | Batch 360/383 | Cost: 1.4840
Epoch: 003/025 Train Acc.: 48.25% | Validation Acc.: 49.10%
Time elapsed: 3.84 min
Epoch: 004/025 | Batch 000/383 | Cost: 1.1930
Epoch: 004/025 | Batch 120/383 | Cost: 1.4712
Epoch: 004/025 | Batch 240/383 | Cost: 1.4109
Epoch: 004/025 | Batch 360/383 | Cost: 1.2905
Epoch: 004/025 Train Acc.: 55.51% | Validation Acc.: 53.20%
Time elapsed: 5.13 min
Epoch: 005/025 | Batch 000/383 | Cost: 1.2403
Epoch: 005/025 | Batch 120/383 | Cost: 1.1947
Epoch: 005/025 | Batch 240/383 | Cost: 1.3734
Epoch: 005/025 | Batch 360/383 | Cost: 1.0939
Epoch: 005/025 Train Acc.: 61.72% | Validation Acc.: 59.40%
Time elapsed: 6.41 min
Epoch: 006/025 | Batch 000/383 | Cost: 1.0849
Epoch: 006/025 | Batch 120/383 | Cost: 1.0589
Epoch: 006/025 | Batch 240/383 | Cost: 1.4794
Epoch: 006/025 | Batch 360/383 | Cost: 1.1895
Epoch: 006/025 Train Acc.: 61.20% | Validation Acc.: 59.20%
Time elapsed: 7.69 min
Epoch: 007/025 | Batch 000/383 | Cost: 1.1170
Epoch: 007/025 | Batch 120/383 | Cost: 1.0913
Epoch: 007/025 | Batch 240/383 | Cost: 1.1495
```

```
Epoch: 007/025 | Batch 360/383 | Cost: 0.9294
Epoch: 007/025 Train Acc.: 68.34% | Validation Acc.: 66.30%
Time elapsed: 8.98 min
Epoch: 008/025 | Batch 000/383 | Cost: 0.8882
Epoch: 008/025 | Batch 120/383 | Cost: 0.9053
Epoch: 008/025 | Batch 240/383 | Cost: 1.0006
Epoch: 008/025 | Batch 360/383 | Cost: 0.7615
Epoch: 008/025 Train Acc.: 74.02% | Validation Acc.: 70.10%
Time elapsed: 10.27 min
Epoch: 009/025 | Batch 000/383 | Cost: 0.7237
Epoch: 009/025 | Batch 120/383 | Cost: 1.0414
Epoch: 009/025 | Batch 240/383 | Cost: 0.8668
Epoch: 009/025 | Batch 360/383 | Cost: 0.7295
Epoch: 009/025 Train Acc.: 78.16% | Validation Acc.: 71.40%
Time elapsed: 11.56 min
Epoch: 010/025 | Batch 000/383 | Cost: 0.7486
Epoch: 010/025 | Batch 120/383 | Cost: 0.8835
Epoch: 010/025 | Batch 240/383 | Cost: 0.7277
Epoch: 010/025 | Batch 360/383 | Cost: 0.8684
Epoch: 010/025 Train Acc.: 74.77% | Validation Acc.: 69.10%
Time elapsed: 12.84 min
Epoch: 011/025 | Batch 000/383 | Cost: 0.6268
Epoch: 011/025 | Batch 120/383 | Cost: 0.6856
Epoch: 011/025 | Batch 240/383 | Cost: 0.7181
Epoch: 011/025 | Batch 360/383 | Cost: 0.5709
Epoch: 011/025 Train Acc.: 78.31% | Validation Acc.: 72.00%
Time elapsed: 14.12 min
Epoch: 012/025 | Batch 000/383 | Cost: 0.6416
Epoch: 012/025 | Batch 120/383 | Cost: 0.7250
Epoch: 012/025 | Batch 240/383 | Cost: 0.7191
Epoch: 012/025 | Batch 360/383 | Cost: 0.6804
Epoch: 012/025 Train Acc.: 81.89% | Validation Acc.: 72.30%
Time elapsed: 15.41 min
Epoch: 013/025 | Batch 000/383 | Cost: 0.5234
Epoch: 013/025 | Batch 120/383 | Cost: 0.7396
Epoch: 013/025 | Batch 240/383 | Cost: 0.5967
Epoch: 013/025 | Batch 360/383 | Cost: 0.8753
Epoch: 013/025 Train Acc.: 81.08% | Validation Acc.: 74.80%
Time elapsed: 16.70 min
Epoch: 014/025 | Batch 000/383 | Cost: 0.4325
Epoch: 014/025 | Batch 120/383 | Cost: 0.5298
Epoch: 014/025 | Batch 240/383 | Cost: 0.5962
Epoch: 014/025 | Batch 360/383 | Cost: 0.6495
Epoch: 014/025 Train Acc.: 85.73% | Validation Acc.: 75.50%
Time elapsed: 17.99 min
Epoch: 015/025 | Batch 000/383 | Cost: 0.4743
Epoch: 015/025 | Batch 120/383 | Cost: 0.4527
Epoch: 015/025 | Batch 240/383 | Cost: 0.3980
Epoch: 015/025 | Batch 360/383 | Cost: 0.4704
```

```
Epoch: 015/025 Train Acc.: 87.51% | Validation Acc.: 75.80%
Time elapsed: 19.27 min
Epoch: 016/025 | Batch 000/383 | Cost: 0.3573
Epoch: 016/025 | Batch 120/383 | Cost: 0.5161
Epoch: 016/025 | Batch 240/383 | Cost: 0.8183
Epoch: 016/025 | Batch 360/383 | Cost: 0.5599
Epoch: 016/025 Train Acc.: 84.63% | Validation Acc.: 73.50%
Time elapsed: 20.55 min
Epoch: 017/025 | Batch 000/383 | Cost: 0.3803
Epoch: 017/025 | Batch 120/383 | Cost: 1.0083
Epoch: 017/025 | Batch 240/383 | Cost: 0.6728
Epoch: 017/025 | Batch 360/383 | Cost: 0.5373
Epoch: 017/025 Train Acc.: 87.59% | Validation Acc.: 75.90%
Time elapsed: 21.84 min
Epoch: 018/025 | Batch 000/383 | Cost: 0.3428
Epoch: 018/025 | Batch 120/383 | Cost: 0.3178
Epoch: 018/025 | Batch 240/383 | Cost: 0.3678
Epoch: 018/025 | Batch 360/383 | Cost: 0.4563
Epoch: 018/025 Train Acc.: 91.10% | Validation Acc.: 75.30%
Time elapsed: 23.13 min
Epoch: 019/025 | Batch 000/383 | Cost: 0.2925
Epoch: 019/025 | Batch 120/383 | Cost: 0.2200
Epoch: 019/025 | Batch 240/383 | Cost: 0.3575
Epoch: 019/025 | Batch 360/383 | Cost: 0.2210
Epoch: 019/025 Train Acc.: 93.76% | Validation Acc.: 74.90%
Time elapsed: 24.41 min
Epoch: 020/025 | Batch 000/383 | Cost: 0.1384
Epoch: 020/025 | Batch 120/383 | Cost: 0.3544
Epoch: 020/025 | Batch 240/383 | Cost: 0.5099
Epoch: 020/025 | Batch 360/383 | Cost: 0.2977
Epoch: 020/025 Train Acc.: 93.00% | Validation Acc.: 77.70%
Time elapsed: 25.69 min
Epoch: 021/025 | Batch 000/383 | Cost: 0.2436
Epoch: 021/025 | Batch 120/383 | Cost: 0.3992
Epoch: 021/025 | Batch 240/383 | Cost: 0.6025
Epoch: 021/025 | Batch 360/383 | Cost: 0.2882
Epoch: 021/025 Train Acc.: 92.24% | Validation Acc.: 76.00%
Time elapsed: 26.98 min
Epoch: 022/025 | Batch 000/383 | Cost: 0.2165
Epoch: 022/025 | Batch 120/383 | Cost: 0.2758
Epoch: 022/025 | Batch 240/383 | Cost: 0.1705
Epoch: 022/025 | Batch 360/383 | Cost: 0.3327
Epoch: 022/025 Train Acc.: 95.40% | Validation Acc.: 77.40%
Time elapsed: 28.26 min
Epoch: 023/025 | Batch 000/383 | Cost: 0.1869
Epoch: 023/025 | Batch 120/383 | Cost: 0.0938
Epoch: 023/025 | Batch 240/383 | Cost: 0.2034
Epoch: 023/025 | Batch 360/383 | Cost: 0.3387
Epoch: 023/025 Train Acc.: 95.31% | Validation Acc.: 77.80%
```

```
Time elapsed: 29.54 min
Epoch: 024/025 | Batch 000/383 | Cost: 0.1274
Epoch: 024/025 | Batch 120/383 | Cost: 0.1440
Epoch: 024/025 | Batch 240/383 | Cost: 0.1691
Epoch: 024/025 | Batch 360/383 | Cost: 0.1815
Epoch: 024/025 Train Acc.: 96.22% | Validation Acc.: 76.30%
Time elapsed: 30.82 min
Epoch: 025/025 | Batch 000/383 | Cost: 0.0593
Epoch: 025/025 | Batch 120/383 | Cost: 0.0517
Epoch: 025/025 | Batch 240/383 | Cost: 0.1567
Epoch: 025/025 | Batch 360/383 | Cost: 0.1563
Epoch: 025/025 Train Acc.: 96.09% | Validation Acc.: 76.80%
Time elapsed: 32.12 min
Total Training Time: 32.12 min
```

```
with torch.set_grad_enabled(False): # save memory during inference
    print('Test accuracy: %.2f%%' % (compute_accuracy(model,
test_loader, device=DEVICE)))
```

```
Test accuracy: 74.04%
```