Nitin Chandrasekhar

Assignment 7

https://github.com/nchan18/ECGR_4105_Assignments.git

```
# cifar_experiments.py
import os
import time
import argparse
import csv
from functools import partial

import numpy as np
import matplotlib.pyplot as plt

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import DataLoader
import torchvision
import torchvision.transforms as transforms

# ----------------------
# Utilities
# ----------------------
def count_parameters(model):
return sum(p.numel() for p in model.parameters())

def model_size_bytes(model):
# rough estimate: sum of parameter dtype sizes (float32 -> 4 bytes)
return sum(p.numel() * p.element_size() for p in model.parameters())

def save_metrics(log_path, rows, header=None):
write_header = not os.path.exists(log_path)
with open(log_path, 'a', newline='') as f:
writer = csv.writer(f)
if write_header and header:
writer.writerow(header)
writer.writerows(rows)
```

```
def plot_learning(train_loss, val_loss, train_acc, val_acc, out_dir, title):
plt.figure(figsize=(12,5))
plt.subplot(1,2,1)
plt.plot(train_loss, label='train loss')
plt.plot(val_loss, label='val loss')
plt.xlabel("Epoch")
plt.legend()
plt.title("Loss")
plt.subplot(1,2,2)
plt.plot(train_acc, label='train acc')
plt.plot(val_acc, label='val acc')
plt.xlabel("Epoch")
plt.legend()
plt.title("Accuracy")
plt.suptitle(title)
os.makedirs(out_dir, exist_ok=True)
plt.savefig(os.path.join(out_dir, 'learning.png'))
plt.close()

# ----------------------
# Models
# ----------------------
class FCNet(nn.Module):
"""Simple fully connected network for CIFAR (flattened input)"""
def __init__(self, hidden_sizes=[1024, 512, 256], num_classes=10):
super().__init__()
layers = []
in_dim = 3*32*32
for h in hidden_sizes:
layers.append(nn.Linear(in_dim, h))
layers.append(nn.ReLU(inplace=True))
in_dim = h
layers.append(nn.Linear(in_dim, num_classes))
self.net = nn.Sequential(*layers)
def forward(self, x):
x = x.view(x.size(0), -1)
return self.net(x)

class BasicCNN(nn.Module):
"""Baseline CNN like lecture. Tune channels and FC for CIFAR10."""
def __init__(self, num_classes=10, dropout_p=0.0, use_batchnorm=False):
super().__init__()
ch1, ch2 = 32, 64
self.conv1 = nn.Conv2d(3, ch1, kernel_size=3, padding=1)
```

```
self.bn1 = nn.BatchNorm2d(ch1) if use_batchnorm else None
self.conv2 = nn.Conv2d(ch1, ch2, kernel_size=3, padding=1)
self.bn2 = nn.BatchNorm2d(ch2) if use_batchnorm else None
self.pool = nn.MaxPool2d(2,2)
self.dropout = nn.Dropout(dropout_p) if dropout_p>0 else None

# after two conv+pool (32x32 -> 16x16 -> 16x16 if only 1 pool; with two pools -> 8x8)
# We will do two pooling stages in forward
self.fc = nn.Sequential(
nn.Flatten(),
nn.Linear(ch2 * 8 * 8, 256),
nn.ReLU(inplace=True),
nn.Dropout(dropout_p) if dropout_p>0 else nn.Identity(),
nn.Linear(256, num_classes)
)

def forward(self, x):
x = F.relu(self.bn1(self.conv1(x)) if self.bn1 else self.conv1(x))
x = self.pool(x)
x = F.relu(self.bn2(self.conv2(x)) if self.bn2 else self.conv2(x))
x = self.pool(x) # now 8x8
return self.fc(x)

class ExtraConvCNN(BasicCNN):
"""Baseline CNN + one extra conv layer (Problem 1.b)"""
def __init__(self, num_classes=10, dropout_p=0.0, use_batchnorm=False):
super().__init__(num_classes=num_classes, dropout_p=dropout_p, use_batchnorm=use_batchnorm)
# override: add one more conv after conv2
ch3 = 128
self.conv3 = nn.Conv2d(64, ch3, kernel_size=3, padding=1)
self.bn3 = nn.BatchNorm2d(ch3) if use_batchnorm else None
# adjust fc input channels -> ch3
self.fc = nn.Sequential(
nn.Flatten(),
nn.Linear(ch3 * 4 * 4, 256), # we will add another pooling to reduce to 4x4
nn.ReLU(inplace=True),
nn.Dropout(dropout_p) if dropout_p>0 else nn.Identity(),
nn.Linear(256, num_classes)
)

def forward(self, x):
x = F.relu(self.bn1(self.conv1(x)) if self.bn1 else self.conv1(x))
x = self.pool(x) # 16x16
```

```
x = F.relu(self.bn2(self.conv2(x)) if self.bn2 else self.conv2(x))
x = self.pool(x) # 8x8
x = F.relu(self.bn3(self.conv3(x)) if self.bn3 else self.conv3(x))
x = self.pool(x) # 4x4
return self.fc(x)

# ResNet building blocks
class ResBlock(nn.Module):
def __init__(self, in_channels, out_channels, stride=1, use_batchnorm=False, dropout_p=0.0):
super().__init__()
self.use_batchnorm = use_batchnorm
mid_channels = out_channels
self.conv1 = nn.Conv2d(in_channels, mid_channels, kernel_size=3, stride=stride, padding=1, bias=not
use_batchnorm)
self.bn1 = nn.BatchNorm2d(mid_channels) if use_batchnorm else None
self.conv2 = nn.Conv2d(mid_channels, out_channels, kernel_size=3, padding=1, bias=not
use_batchnorm)
self.bn2 = nn.BatchNorm2d(out_channels) if use_batchnorm else None
self.relu = nn.ReLU(inplace=True)
self.dropout = nn.Dropout(dropout_p) if dropout_p>0 else None

if stride != 1 or in_channels != out_channels:
# projection to match dimensions
self.downsample = nn.Sequential(
nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=stride, bias=not use_batchnorm),
nn.BatchNorm2d(out_channels) if use_batchnorm else nn.Identity()
)
else:
self.downsample = nn.Identity()

def forward(self, x):
identity = self.downsample(x)
out = self.conv1(x)
if self.bn1: out = self.bn1(out)
out = self.relu(out)
if self.dropout: out = self.dropout(out)
out = self.conv2(out)
if self.bn2: out = self.bn2(out)
out += identity
out = self.relu(out)
return out

class ResNet10(nn.Module):
```

```
"""Simple configurable ResNet with 10 blocks total (count blocks as BasicBlocks)"""
def __init__(self, num_classes=10, use_batchnorm=False, dropout_p=0.0):
super().__init__()
self.in_channels = 16
self.conv = nn.Conv2d(3, self.in_channels, kernel_size=3, padding=1, bias=not use_batchnorm)
self.bn0 = nn.BatchNorm2d(self.in_channels) if use_batchnorm else None
self.relu = nn.ReLU(inplace=True)

# make layers with block counts: example distribution to reach ~10 blocks:
# layer1: 2 blocks (out 16)
# layer2: 2 blocks (out 32)
# layer3: 2 blocks (out 64)
# layer4: 4 blocks (out 128) -> total = 10 blocks (2+2+2+4 = 10)
self.layer1 = self._make_layer(16, 2, stride=1, use_batchnorm=use_batchnorm, dropout_p=dropout_p)
self.layer2 = self._make_layer(32, 2, stride=2, use_batchnorm=use_batchnorm, dropout_p=dropout_p)
self.layer3 = self._make_layer(64, 2, stride=2, use_batchnorm=use_batchnorm, dropout_p=dropout_p)
self.layer4 = self._make_layer(128, 4, stride=2, use_batchnorm=use_batchnorm, dropout_p=dropout_p)

self.avgpool = nn.AdaptiveAvgPool2d((1,1))
self.fc = nn.Linear(128, num_classes)

def _make_layer(self, out_channels, blocks, stride, use_batchnorm, dropout_p):
layers = []
layers.append(ResBlock(self.in_channels, out_channels, stride=stride, use_batchnorm=use_batchnorm,
dropout_p=dropout_p))
self.in_channels = out_channels
for _ in range(1, blocks):
layers.append(ResBlock(self.in_channels, out_channels, stride=1, use_batchnorm=use_batchnorm,
dropout_p=dropout_p))
return nn.Sequential(*layers)

def forward(self, x):
x = self.conv(x)
if self.bn0: x = self.bn0(x)
x = self.relu(x)
x = self.layer1(x)
x = self.layer2(x)
x = self.layer3(x)
x = self.layer4(x)
x = self.avgpool(x)
x = torch.flatten(x,1)
return self.fc(x)
```

```
# ----------------------
# Training / Eval loops
# ----------------------
def train_one_epoch(model, device, dataloader, criterion, optimizer):
model.train()
running_loss = 0.0
correct = 0
total = 0
start = time.time()
for inputs, targets in dataloader:
inputs, targets = inputs.to(device), targets.to(device)
optimizer.zero_grad()
outputs = model(inputs)
loss = criterion(outputs, targets)
loss.backward()
optimizer.step()

running_loss += loss.item() * inputs.size(0)
_, preds = outputs.max(1)
correct += preds.eq(targets).sum().item()
total += inputs.size(0)
epoch_time = time.time() - start
avg_loss = running_loss / total
acc = correct / total
return avg_loss, acc, epoch_time

def eval_model(model, device, dataloader, criterion):
model.eval()
running_loss = 0.0
correct = 0
total = 0
with torch.no_grad():
for inputs, targets in dataloader:
inputs, targets = inputs.to(device), targets.to(device)
outputs = model(inputs)
loss = criterion(outputs, targets)
running_loss += loss.item() * inputs.size(0)
_, preds = outputs.max(1)
correct += preds.eq(targets).sum().item()
total += inputs.size(0)
if total == 0:
return 0.0, 0.0
return running_loss / total, correct / total
```

```
# ----------------------
# Experiment runner
# ----------------------
def run_experiment(args):
device = 'cuda' if torch.cuda.is_available() else 'cpu'
print("Device:", device)

# Data transforms
transform_train = transforms.Compose([
transforms.RandomCrop(32, padding=4),
transforms.RandomHorizontalFlip(),
transforms.ToTensor(),
transforms.Normalize((0.4914,0.4822,0.4465), (0.2023,0.1994,0.2010))
])
transform_test = transforms.Compose([
transforms.ToTensor(),
transforms.Normalize((0.4914,0.4822,0.4465), (0.2023,0.1994,0.2010))
])

trainset = torchvision.datasets.CIFAR10(root=args.data_dir, train=True, download=True,
transform=transform_train)
testset = torchvision.datasets.CIFAR10(root=args.data_dir, train=False, download=True,
transform=transform_test)

trainloader = DataLoader(trainset, batch_size=args.batch_size, shuffle=True, num_workers=4)
testloader = DataLoader(testset, batch_size=args.batch_size, shuffle=False, num_workers=4)

# choose model
model_map = {
'fc': lambda: FCNet(num_classes=10),
'cnn': lambda: BasicCNN(num_classes=10, dropout_p=args.dropout_p, use_batchnorm=args.batchnorm),
'cnn_extra': lambda: ExtraConvCNN(num_classes=10, dropout_p=args.dropout_p,
use_batchnorm=args.batchnorm),
'resnet10': lambda: ResNet10(num_classes=10, use_batchnorm=args.batchnorm,
dropout_p=args.dropout_p)
}
model = model_map[args.model]().to(device)
print("Model:", args.model, "Params:", count_parameters(model), "Size (MB):",
model_size_bytes(model)/1024**2)

criterion = nn.CrossEntropyLoss()
# weight decay handled via optimizer for L2 regularization
```

```
optimizer = optim.SGD(model.parameters(), lr=args.lr, momentum=0.9,
weight_decay=args.weight_decay)

scheduler = None
if args.lr_step:
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=args.lr_step, gamma=0.1)

train_loss_history, val_loss_history = [], []
train_acc_history, val_acc_history = [], []
epoch_times = []
best_val_acc = 0.0

out_dir = os.path.join(args.out_dir, f"{args.model}_run")
os.makedirs(out_dir, exist_ok=True)

# CSV header
csv_path = os.path.join(out_dir, "log.csv")
if not os.path.exists(csv_path):
save_metrics(csv_path, [], header=["epoch","train_loss","train_acc","val_loss","val_acc","epoch_time"])

start_total = time.time()
for epoch in range(1, args.epochs+1):
t_loss, t_acc, e_time = train_one_epoch(model, device, trainloader, criterion, optimizer)
v_loss, v_acc = eval_model(model, device, testloader, criterion)

if scheduler:
scheduler.step()

train_loss_history.append(t_loss)
val_loss_history.append(v_loss)
train_acc_history.append(t_acc)
val_acc_history.append(v_acc)
epoch_times.append(e_time)

print(f"Epoch {epoch:03d}/{args.epochs} | train_loss={t_loss:.4f} acc={t_acc:.4f} | val_loss={v_loss:.4f}
acc={v_acc:.4f} | epoch_time={e_time:.2f}s")

save_metrics(csv_path, [[epoch, t_loss, t_acc, v_loss, v_acc, e_time]])

# save checkpoint if best
if v_acc > best_val_acc:
best_val_acc = v_acc
```

```
torch.save({
'epoch': epoch,
'model_state': model.state_dict(),
'optimizer_state': optimizer.state_dict(),
'val_acc': v_acc
}, os.path.join(out_dir, 'best_checkpoint.pth'))

total_time = time.time() - start_total
print("Total training time (s):", total_time)

# final evaluation & save final model
final_metrics = {
'best_val_acc': best_val_acc,
'final_val_acc': val_acc_history[-1],
'final_val_loss': val_loss_history[-1],
'params': count_parameters(model),
'model_size_MB': model_size_bytes(model)/1024**2,
'total_time_s': total_time
}
print("Final metrics:", final_metrics)

# plot & save
plot_learning(train_loss_history, val_loss_history, train_acc_history, val_acc_history, out_dir,
title=f"{args.model}")
torch.save(model.state_dict(), os.path.join(out_dir, 'final_model.pth'))
# summary CSV for run
summary_path = os.path.join(args.out_dir, "summary.csv")
save_metrics(summary_path, [[args.model, args.epochs, final_metrics['best_val_acc'],
final_metrics['final_val_acc'],
final_metrics['final_val_loss'], final_metrics['params'], final_metrics['model_size_MB'],
final_metrics['total_time_s']]],
header=["model","epochs","best_val_acc","final_val_acc","final_val_loss","params","model_size_MB","tota
l_time_s"])
return final_metrics

# ----------------------
# CLI
# ----------------------
def parse_args():
parser = argparse.ArgumentParser()
parser.add_argument('--data-dir', default='./data')
parser.add_argument('--model', choices=['fc','cnn','cnn_extra','resnet10'], default='cnn')
parser.add_argument('--batch-size', type=int, default=128)
```

```
parser.add_argument('--epochs', type=int, default=300)
parser.add_argument('--lr', type=float, default=0.1)
parser.add_argument('--weight-decay', type=float, default=0.0) # for L2 regularization
parser.add_argument('--dropout-p', type=float, default=0.0)
parser.add_argument('--batchnorm', action='store_true')
parser.add_argument('--out-dir', default='./runs')
parser.add_argument('--lr-step', type=int, default=100, help='StepLR step size (0 to disable)')
return parser.parse_args()

if __name__ == '__main__':
args = parse_args()
if args.lr_step == 0:
args.lr_step = None
run_experiment(args)
```