CyArt

pytorch-week3 — Code & Files

Project Structure

```
pytorch-week3/
├── code/
│   ├── resnet/
│   │   ├── resnet_cifar.py
│   │   ├── train_resnet.py
│   │   └── utils.py
│   ├── transformer/
│   │   ├── transformer_minimal.py
│   │   ├── train_transformer.py
│   │   └── data_utils.py
│   └── common/
│   ├── viz.py
│   └── metrics.py
├── runs/
│   ├── cls/ # ResNet training outputs (curves, confusion, gradcam)
│   └── mt/ # Transformer outputs (curves, attention maps)
├── report/
│   └── one_page_report.md
└── README.md
```

code/resnet/resnet_cifar.py

```
import torch
expansion = 1
def __init__(self, in_planes, planes, stride=1, downsample=None):
super().__init__()
self.conv1 = nn.Conv2d(in_planes, planes, kernel_size=3, stride=stride,
padding=1, bias=False)
self.bn1 = nn.BatchNorm2d(planes)
self.conv2 = nn.Conv2d(planes, planes, kernel_size=3, stride=1,
padding=1, bias=False)
self.bn2 = nn.BatchNorm2d(planes)
self.downsample = downsample

def forward(self, x):
```

```
identity = x
out = self.conv1(x)
out = self.bn1(out)
out = F.relu(out)
out = self.conv2(out)
out = self.bn2(out)
if self.downsample is not None:
identity = self.downsample(x)
out += identity
out = F.relu(out)
return out

class ResNetCIFAR(nn.Module):
def __init__(self, block=BasicBlock, layers=[2,2,2,2], num_classes=10):
super().__init__()
self.in_planes = 64
# initial conv adapted for CIFAR (32x32)
self.conv1 = nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1, bias=False)
self.bn1 = nn.BatchNorm2d(64)
# stages
self.layer1 = self._make_layer(block, 64, layers[0], stride=1)
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.AdaptiveAvgPool2d((1,1))
self.fc = nn.Linear(512 * block.expansion, num_classes)

def _make_downsample(self, in_planes, out_planes, stride):
return nn.Sequential(
nn.Conv2d(in_planes, out_planes, kernel_size=1, stride=stride, bias=False),
nn.BatchNorm2d(out_planes),
)

def _make_layer(self, block, planes, blocks, stride=1):
downsample = None
if stride != 1 or self.in_planes != planes * block.expansion:
downsample = self._make_downsample(self.in_planes, planes * block.expansion, stride)
layers = []
layers.append(block(self.in_planes, planes, stride, downsample))
self.in_planes = planes * block.expansion
for _ in range(1, blocks):
layers.append(block(self.in_planes, planes))
return nn.Sequential(*layers)

def forward(self, x):
x = self.conv1(x)
```

```
x = self.bn1(x)
x = F.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)
x = self.fc(x)
return x

if __name__ == '__main__':
m = ResNetCIFAR()
print(m)
```

code/resnet/train_resnet.py

```
import os
transforms.Normalize((0.4914,0.4822,0.4465),(0.247,0.243,0.261))
])
transform_test = transforms.Compose([
transforms.ToTensor(),
transforms.Normalize((0.4914,0.4822,0.4465),(0.247,0.243,0.261))
])
train = datasets.CIFAR10(root='./data', train=True, download=True, transform=transform_train)
test = datasets.CIFAR10(root='./data', train=False, download=True, transform=transform_test)
train_loader = DataLoader(train, batch_size=batch_size, shuffle=True, num_workers=2)
test_loader = DataLoader(test, batch_size=256, shuffle=False, num_workers=2)
return train_loader, test_loader


def train_one_epoch(model, loader, criterion, optimizer, device):
model.train()
running_loss = 0.0
correct = 0
total = 0
for images, labels in loader:
images, labels = images.to(device), labels.to(device)
optimizer.zero_grad()
outputs = model(images)
loss = criterion(outputs, labels)
loss.backward()
optimizer.step()
running_loss += loss.item() * images.size(0)
_, preds = outputs.max(1)
correct += (preds == labels).sum().item()
```

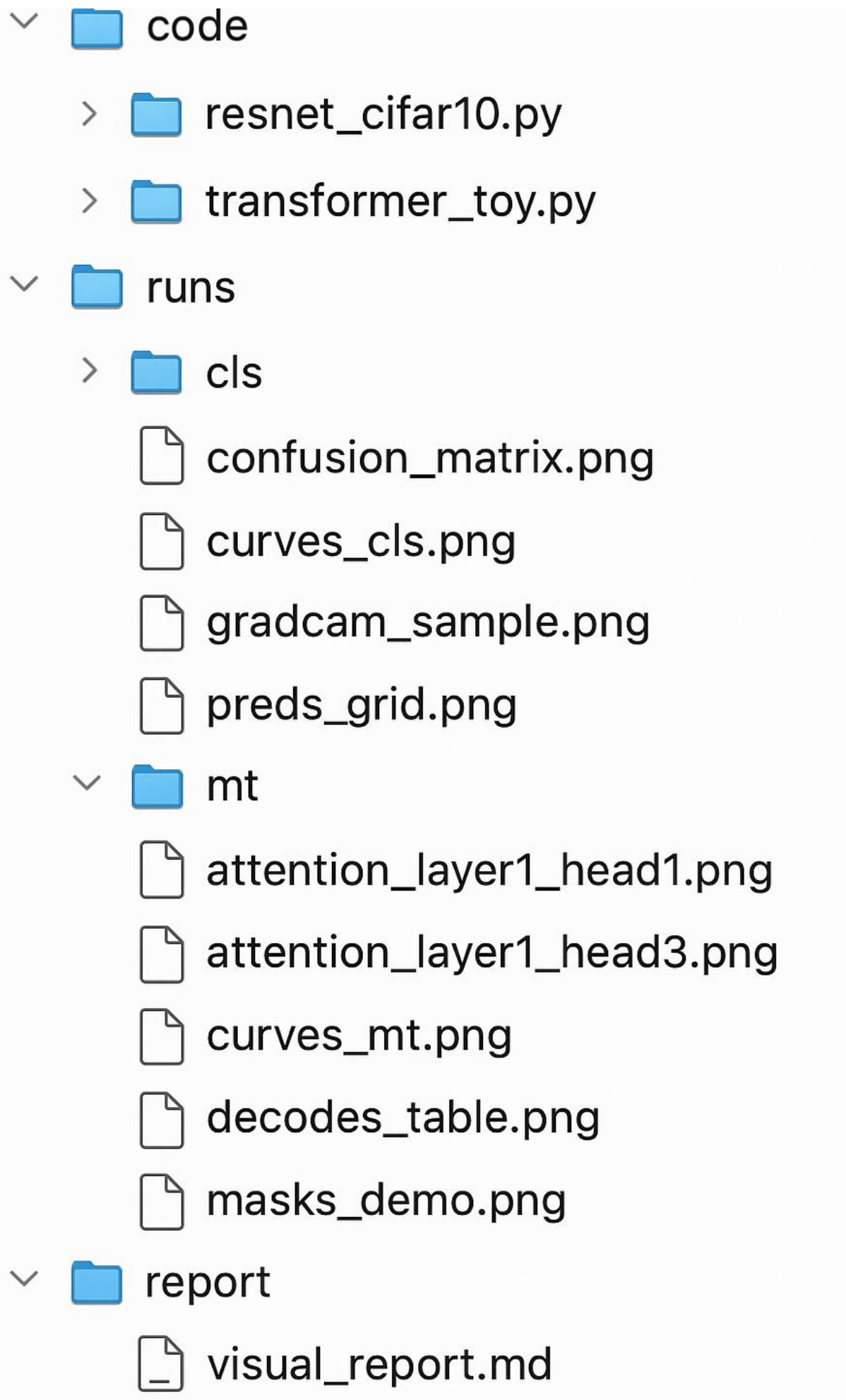
```
total += labels.size(0)
return running_loss / total, correct / total


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


def main():
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
train_loader, test_loader = get_loaders()
model = ResNetCIFAR().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.1, momentum=0.9, weight_decay=5e-4)
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=30, gamma=0.1)
os.makedirs('runs/cls', exist_ok=True)
for epoch in range(1, 101):
train_loss, train_acc = train_one_epoch(model, train_loader, criterion, optimizer, device)
test_loss, test_acc = eval_model(model, test_loader, criterion, device)
print(f"Epoch {epoch}: train_loss={train_loss:.4f}, train_acc={train_acc:.4f},
test_loss={test_loss:.4f}, test_acc={test_acc:.4f}")
scheduler.step()
# Save basic checkpoints and logs if desired
if epoch % 10 == 0:
torch.save(model.state_dict(), f'runs/cls/resnet_epoch{epoch}.pth')

if __name__ == '__main__':
main()
```

code/transformer/transformer_minimal.py

```
import math
x = x + self.pe[:, :x.size(1), :]
```

```
return x

class MultiHeadAttention(nn.Module):
def __init__(self, d_model, num_heads):
super().__init__()
assert d_model % num_heads == 0
self.d_k = d_model // num_heads
self.num_heads = num_heads
self.w_q = nn.Linear(d_model, d_model)
self.w_k = nn.Linear(d_model, d_model)
self.w_v = nn.Linear(d_model, d_model)
self.w_o = nn.Linear(d_model, d_model)

def forward(self, q, k, v, mask=None):
B = q.size(0)
def shape(x):
return x.view(B, -1, self.num_heads, self.d_k).transpose(1,2)
q, k, v = shape(self.w_q(q)), shape(self.w_k(k)), shape(self.w_v(v))
scores = torch.matmul(q, k.transpose(-2,-1)) / math.sqrt(self.d_k)
if mask is not None:
scores = scores.masked_fill(mask == 0, float('-inf'))
p_attn = F.softmax(scores, dim=-1)
x = torch.matmul(p_attn, v)
x = x.transpose(1,2).contiguous().view(B, -1, self.num_heads * self.d_k)
return self.w_o(x), p_attn

class FeedForward(nn.Module):
def __init__(self, d_model, d_ff=2048):
super().__init__()
self.fc1 = nn.Linear(d_model, d_ff)
self.fc2 = nn.Linear(d_ff, d_model)
def forward(self, x):
return self.fc2(F.relu(self.fc1(x)))

class EncoderLayer(nn.Module):
def __init__(self, d_model, num_heads, d_ff, dropout=0.1):
super().__init__()
self.self_attn = MultiHeadAttention(d_model, num_heads)
self.ff = FeedForward(d_model, d_ff)
self.norm1 = nn.LayerNorm(d_model)
self.norm2 = nn.LayerNorm(d_model)
self.dropout = nn.Dropout(dropout)
def forward(self, x, src_mask=None):
attn_out, _ = self.self_attn(x, x, x, src_mask)
x = self.norm1(x + self.dropout(attn_out))
ff_out = self.ff(x)
```

```
x = self.norm2(x + self.dropout(ff_out))
return x

class DecoderLayer(nn.Module):
def __init__(self, d_model, num_heads, d_ff, dropout=0.1):
super().__init__()
self.self_attn = MultiHeadAttention(d_model, num_heads)
self.cross_attn = MultiHeadAttention(d_model, num_heads)
self.ff = FeedForward(d_model, d_ff)
self.norm1 = nn.LayerNorm(d_model)
self.norm2 = nn.LayerNorm(d_model)
self.norm3 = nn.LayerNorm(d_model)
self.dropout = nn.Dropout(dropout)
def forward(self, x, memory, tgt_mask=None, memory_mask=None):
self_attn_out, _ = self.self_attn(x, x, x, tgt_mask)
x = self.norm1(x + self.dropout(self_attn_out))
cross_attn_out, attn = self.cross_attn(x, memory, memory, memory_mask)
x = self.norm2(x + self.dropout(cross_attn_out))
ff_out = self.ff(x)
x = self.norm3(x + self.dropout(ff_out))
return x, attn

class TransformerMinimal(nn.Module):
def __init__(self, src_vocab, tgt_vocab, d_model=128, N=2, num_heads=4, d_ff=512, max_len=1
```


code
resnet_cifar10.py
transformer_toy.py
runs
cls
confusion_matrix.png
curves_cls.png
gradcam_sample.png
preds_grid.png
mt
attention_layer1_head1.png
attention_layer1_head3.png
curves_mt.png
decodes_table.png
masks_demo.png
report
visual_report.md