DenseNet161_Implementation

Summary

This is an implementation of DenseNet161 utilizing the CIFAR-100 dataset. This was originally trained on ImageNet, which has millions of images, however. The goal is to see the advantage of fine-tuning a pre-trained model vs. training a model from scratch. I ran into some issues with GPU constraints, however.

Implementation Overview

Imports

```
# Imports
import torch
from torch import nn
from torch.utils.data import DataLoader
import torchvision
from torchvision import datasets
from torchvision import transforms
import tqdm
import numpy as np
from sklearn.metrics import f1_score
```

Define Train/Test

```
# Define Training/Testing

# train
def train(dataloader, model, loss_fn, optimizer, device):
  model.train()
  # step being a training step
```

```
    for step, (X, y) in enumerate(dataloader):
      # send to CPU/GPU
      X = X.to(device)
      y = y.to(device)
      # model's prediction
      pred = model(X)
      # get loss
      loss = loss_fn(pred, y)
      # backprop
      optimizer.zero_grad()
      loss.backward()
      optimizer.step()

      # progress check
      if step % 100 == 0:
        print(f'step: {step}, loss: {loss.item()}')

  # validation
  def test(dataloader, model, loss_fn, device):
    num_steps = len(dataloader)
    model.eval()
    test_loss = 0
    correct = 0
    y_true = []
    y_predicted = []

    with torch.no_grad():
      for X, y in dataloader:
        X = X.to(device)
        y = y.to(device)
        pred = model(X)
        loss = loss_fn(pred, y)
        test_loss += loss.item()
        y_true.extend(y.cpu().numpy())
        y_hat = pred.argmax(1)
        y_predicted.extend(y_hat.cpu().numpy())
        correct_step = (y_hat == y).type(torch.float).sum().item()
        correct += correct_step

    test_loss /= num_steps # average loss
    correct = correct / (num_steps * batch_size)
    f1 = f1_score(y_true, y_predicted, average='macro')
  print()
  print(f'F1 Score: {f1}')
  print(f'Test Accuracy: {correct}')
```

Running the Model

Rather than implementing DenseNet161from scratch I am using the library. Call `model = torchvision.models.densenet161(pretrained=True).to(device)` and change `pretrained` to `False` to train it from scratch. DenseNet161 can take images as small as 29x29, so I don't need to resize anything.

Output w/o pretraining

Here, we train the model from scratch rather than fine-tuning with pretrained parameters. Given the length of time I cut the training short out of concern I'd run out of GPU time.

```
model = torchvision.models.densenet161(pretrained=False).to(device)
epochs = 20
lr = 5e-3
# optimize
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=lr)
```

```
25%|██▌       | 5/20 [1:34:37<4:44:29, 1138.00s/it]
F1 Score: 0.1749929852771643
Test Accuracy: 0.2045726837060703
```

Output w/ pretraining

Unfortunately, I ran out of GPU time very quickly after earlier experimentation with AlexNet. However, we can see the accuracy is significantly improved even though there was significantly less time spent training on the data. Note also how much faster fine-tuning is vs

```
15%|█▌        | 3/20 [08:10<46:29, 164.08s/it]
F1 Score: 0.5468895652485322
Test Accuracy: 0.5493210862619808
```