

## Q2 Weather Recognition

For Image classification problem, it is always the prior choice to use a well pretrained model(typically on Imagenet) to fine-tuning on our datasets. Therefore, I choose the ResNet34 as my backbone and add a 3-layer MLP linear protocol to perform the downstream tasks.

### Design of DataLoader

Firstly, I create a **Q2Dataset** object which is the **sub object** of the `torch.utils.data.Dataset`. During the construction, I first used the regular expression to extract the labels hided in the file name. Then I converted the label into the **one-hot encoding label** for subsequent classification process.

Then, I use the **PIL** library which is very suitable for image processing to load the image and convert them into PIL Image format. After that, I use `torchvision.transforms` to do some **image augmentation**(Like random rotation) and convert all the PIL Image into Pytorch Tensor. Notice that I **resized** all the Image to (255,255) with respect to height and width of the iamge. And there is no image augmentation for the data in the test datasets(when testing on train dataset in our case)

Finally, we can get our DataLoader. We **do the shuffle operation during training** to avoid the order of the image becoming a feature. Batch size is set to 16, and use 2 workers to load the data.

Below is the screen shot of my DataLoader and Dataset.

#### DataLoader:

```
# initialization
dataset = Q2Dataset(data_dir="../../Data/Q2/train_data/")
dataloader = DataLoader(dataset=dataset, batch_size=batch_size, shuffle=True, num_workers=2)
```

#### Dataset

```

class Q2Dataset(Dataset):
    def __init__(self, data_dir, transform: Optional[transforms.Compose] = None, train: Optional[bool] = True) -> None:
        super().__init__()
        self.data_dir = data_dir
        self.transform = transform
        self.train = train

        if os.path.exists(self.data_dir):
            files = os.listdir(self.data_dir)
            file_absolute_paths = []
            labels = []
            for file in files:
                labels.append(re.split(r"\d+", file)[0])
                file_absolute_path = self.data_dir + file
                file_absolute_paths.append(file_absolute_path)
            self.data = pd.get_dummies(labels, dtype=int)
            self.data["file"] = file_absolute_paths
        else:
            raise FileNotFoundError

    def __len__(self):
        return len(self.data)

    def __getitem__(self, index):
        image = Image.open(self.data.iloc[index, -1]).convert("RGB")
        label = self.data.iloc[index, :-1]
        label = torch.tensor(label, dtype=torch.float32)

        if self.transform is None:
            if self.train:
                self.transform = transforms.Compose([
                    transforms.RandomRotation(15),
                    transforms.Resize((255, 255)),
                    transforms.ToTensor(),
                    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225]) # mean and variance in ImageNet
                ])
            else:
                self.transform = transforms.Compose([
                    transforms.Resize((255, 255)),
                    transforms.ToTensor(),
                    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225]) # mean and variance in ImageNet
                ])

        img = self.transform(image)
        return img, label

```

## Design of Model

I use the ResNet34 as my backbone to extract the features of image and project it into the embedding layer and use a 3-layer MLP as a linear protocol to deal with the classification problem.

ResNet34 is easy with bunch of Convolution layers and use the skip connection in each block. The benefit of this design is that it can avoid gradient vanishing which allows us to optimize and train a much deeper network.

The 3-layer MLP is also very simple. Each Linear block contains a linear projection layer, a batch normalization layer and a ReLU activation layer. With 3 linear blocks, we can learn a non-linear mapping to the output space.

Here is the screenshot of my network architecture:

```

class LinearBlock(nn.Module):
    def __init__(self, in_features, out_features, act: Optional[bool] = True, last: bool = False, *args, **kwargs) -> None:
        super().__init__(*args, **kwargs)
        self.in_features = in_features
        self.out_features = out_features
        self.act = act
        self.last = last

        self.linear = nn.Sequential(
            nn.Linear(in_features=in_features, out_features=out_features, bias=True if not self.last else False),
            nn.BatchNorm1d(out_features) if not last else nn.Identity(),
            nn.ReLU() if act else nn.Identity()
        )

    def forward(self, x):
        return self.linear(x)

class MyResNet(nn.Module):
    def __init__(self,
                 num_classes: int,
                 in_features: int = 1000,
                 hidden_features: Optional[list] = [512, 256, 64],
                 *args, **kwargs) -> None:
        super().__init__(*args, **kwargs)
        self.num_classes = num_classes
        self.hidden_features = hidden_features

        self.backbone = pretrained_model.resnet34(pretrained=True) # weights=pretrain_model.ResNet34_Weights.DEFAULT
        for parameter in self.backbone.parameters():
            parameter.requires_grad = False

        self.classifier = nn.Sequential(
            LinearBlock(in_features=in_features, out_features=hidden_features[0]),
            LinearBlock(in_features=hidden_features[0], out_features=hidden_features[1]),
            LinearBlock(in_features=hidden_features[1], out_features=hidden_features[2]),
            LinearBlock(in_features=hidden_features[2], out_features=self.num_classes, act=False, last=True)
        )

        layers = OrderedDict([
            ("backbone", self.backbone),
            ("classifier", self.classifier)
        ])

        self.model = nn.Sequential(layers)

    def forward(self, x):
        return self.model(x)

```

## Training process and Result

I carefully tuned the hyper parameters, and I found that it is important to have a proper learning rate. In this case, I set the learning rate to  $3e-5$  and reach 98% accuracy on the training dataset.

Here is the screen shot of my process and results.

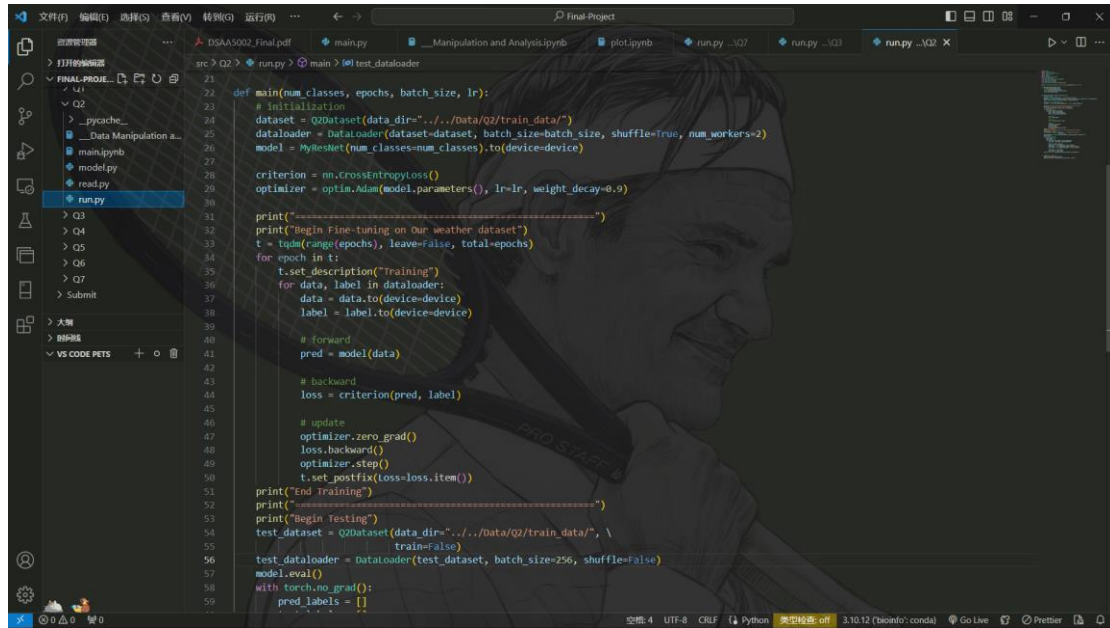
```

[1]: from run import main, seed_everything

[2]: seed_everything(random_state=42)
      main(num_classes=5, epochs=50, batch_size=16, lr=3e-5)

=====
Begin Fine-tuning on Our weather dataset
=====
End Training
=====
Begin Testing
Accuracy: 0.98

```



```
21:
22: def main(num_classes, epochs, batch_size, lr):
23:     # Initialization
24:     dataset = Q2Dataset(data_dir="../../Data/Q2/train_data/")
25:     dataloader = DataLoader(dataset=dataset, batch_size=batch_size, shuffle=True, num_workers=2)
26:     model = PyResNet(num_classes=num_classes).to(device=device)
27:
28:     criterion = nn.CrossEntropyLoss()
29:     optimizer = optim.Adam(model.parameters(), lr=lr, weight_decay=0.9)
30:
31:     print("=====")
32:     print("Begin Fine-tuning on Our weather dataset")
33:     t = tqdm(range(epochs), leave=False, total=epochs)
34:     for epoch in t:
35:         t.set_description("Training")
36:         for data, label in dataloader:
37:             data = data.to(device=device)
38:             label = label.to(device=device)
39:
40:             # forward
41:             pred = model(data)
42:
43:             # backward
44:             loss = criterion(pred, label)
45:
46:             # update
47:             optimizer.zero_grad()
48:             loss.backward()
49:             optimizer.step()
50:             t.set_postfix(loss=loss.item())
51:
52:     print("End Training")
53:     print("=====")
54:     print("Begin Testing")
55:     test_dataset = Q2Dataset(data_dir="../../Data/Q2/train_data/", \
56:                             train=False)
57:     test_dataloader = DataLoader(test_dataset, batch_size=256, shuffle=False)
58:     model.eval()
59:     with torch.no_grad():
60:         pred_labels = []
```