Q2 readme

a. DataLoader Design

- **Standardization:** Images were resized to 224x224 pixels to conform to the input size required by the pre-trained ResNet18 model. Following resizing, we applied normalization using means [0.485, 0.456, 0.406] and standard deviations [0.229, 0.224, 0.225], which are the standard normalization values for models pre-trained on the ImageNet dataset.
- **Label Definition:** Labels were defined based on the filename prefixes, converted to one-hot encoded format to suit the multi-class classification objective.
- **Shuffling:** Data shuffling was employed (shuffle=True) to ensure that each batch seen by the model during training contains a random assortment of all classes, which helps prevent the model from learning spurious patterns.
- **Batch Size:** We chose a batch size of 32, as it is a standard size that balances computational efficiency and model update granularity.

b. Code screenshots related to (a)

```python
class WeatherDataset(Dataset):
    def __init__(self, data_dir):
        self.data_dir = data_dir
        self.image_files = os.listdir(data_dir)
        self.transform = transforms.Compose([
            transforms.Resize((224, 224)),    # Resize the image to 224x224
            transforms.ToTensor(),    # Convert to tensor
            transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])    # Normalize
        ])

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

    def __getitem__(self, idx):
        image_name = self.image_files[idx]
        image_path = os.path.join(self.data_dir, image_name)
        image = Image.open(image_path).convert('RGB')
        image_tensor = self.transform(image)
        # Extract label information
        label_name = image_name.split('.')[0]    # Remove file extension
        label = self.get_label(label_name)

        return image_tensor, label

    def get_label(self, label_name):
        # Return one-hot encoded label based on label name
        if label_name.startswith('Cloudy'):
            return torch.tensor([1, 0, 0, 0, 0])
        elif label_name.startswith('Foggy'):
            return torch.tensor([0, 1, 0, 0, 0])
        elif label_name.startswith('Rainy'):
            return torch.tensor([0, 0, 1, 0, 0])
        elif label_name.startswith('Snowy'):
            return torch.tensor([0, 0, 0, 1, 0])
        elif label_name.startswith('Sunny'):
            return torch.tensor([0, 0, 0, 0, 1])
        else:
            raise ValueError('Invalid label name')
```

```python
# Set batch size and shuffle
batch_size = 32
shuffle = True
```

c. A brief introduction to model along with relevant code screenshots

- The core of our classification model is the ResNet18 architecture, renowned for its performance on image data. The model was initialized with weights pre-trained on the ImageNet dataset to leverage the representational learning from a wide and general dataset. We modified the final fully connected layer of the ResNet18 to output five classes, corresponding to our weather classification task: Cloudy, Foggy, Rainy, Snowy, and Sunny.

```python
class WeatherClassifier(nn.Module):
    def __init__(self, num_classes):
        super(WeatherClassifier, self).__init__()
        self.base_model = models.resnet18(pretrained=True)
        num_features = self.base_model.fc.in_features
        self.base_model.fc = nn.Linear(num_features, num_classes)

    def forward(self, x):
        x = self.base_model(x)
        return x
```

```
base_model.conv1: Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
base_model.layer1.0.conv1: Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
base_model.layer1.0.conv2: Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
base_model.layer1.1.conv1: Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
base_model.layer1.1.conv2: Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
base_model.layer2.0.conv1: Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
base_model.layer2.0.conv2: Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
base_model.layer2.0.downsample.0: Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
base_model.layer2.1.conv1: Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
base_model.layer2.1.conv2: Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
base_model.layer3.0.conv1: Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
base_model.layer3.0.conv2: Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
base_model.layer3.0.downsample.0: Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
base_model.layer3.1.conv1: Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
base_model.layer3.1.conv2: Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
base_model.layer4.0.conv1: Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
base_model.layer4.0.conv2: Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
base_model.layer4.0.downsample.0: Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
base_model.layer4.1.conv1: Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
base_model.layer4.1.conv2: Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
base_model.fc: Linear(in_features=512, out_features=5, bias=True)
Output Shape: [5]
```

- To optimize the model, we used the Adam optimizer with a learning rate of 0.001 and selected the loss function CrossEntropyLoss, which is appropriate for multi-class classification problem.

```python
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

- The model was trained for 50 epochs, and at each epoch, the training loss and accuracy were recorded to monitor the learning progress. And the training and evaluation stages were called separately to ensure the correctness of the model's behavior.

```python
# Train the model
for epoch in range(num_epochs):
    model.train()  # Make sure the model is in training mode
    running_loss = 0.0
    correct = 0
    total = 0

    for images, labels in dataloader:
        images = images.to(device)
        labels = labels.to(device)

        # Forward pass
        outputs = model(images)
        loss = criterion(outputs, torch.argmax(labels, dim=1))

        # Backward pass and optimization
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # Calculate accuracy
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == torch.argmax(labels, dim=1)).sum().item()

        running_loss += loss.item()

    # Print loss and accuracy for each epoch
    epoch_loss = running_loss / len(dataloader)
    epoch_acc = correct / total * 100
    print(f"Epoch [{epoch+1}/{num_epochs}]\t Loss: {epoch_loss:.4f}\t Accuracy: {epoch_acc:.2f}%")
```

```
# Evaluate the model on the entire training dataset
model.eval()  # Make sure the model is in evaluation mode
total = 0
correct = 0
```

d. Screenshots depicting the training process

```
Epoch [1/50]   Loss: 0.8685   Accuracy: 67.20%
Epoch [2/50]   Loss: 0.5025   Accuracy: 84.40%
Epoch [3/50]   Loss: 0.2452   Accuracy: 89.20%
Epoch [4/50]   Loss: 0.1419   Accuracy: 94.40%
Epoch [5/50]   Loss: 0.1959   Accuracy: 95.20%
Epoch [6/50]   Loss: 0.2013   Accuracy: 92.40%
Epoch [7/50]   Loss: 0.1565   Accuracy: 94.80%
Epoch [8/50]   Loss: 0.0788   Accuracy: 96.80%
Epoch [9/50]   Loss: 0.1003   Accuracy: 96.80%
Epoch [10/50]  Loss: 0.1160   Accuracy: 96.80%
Epoch [11/50]  Loss: 0.1152   Accuracy: 96.00%
Epoch [12/50]  Loss: 0.1153   Accuracy: 95.60%
Epoch [13/50]  Loss: 0.0863   Accuracy: 97.20%
Epoch [14/50]  Loss: 0.0522   Accuracy: 98.00%
Epoch [15/50]  Loss: 0.0443   Accuracy: 97.60%
Epoch [16/50]  Loss: 0.0192   Accuracy: 99.20%
Epoch [17/50]  Loss: 0.1000   Accuracy: 96.80%
Epoch [18/50]  Loss: 0.0552   Accuracy: 97.60%
Epoch [19/50]  Loss: 0.0469   Accuracy: 98.40%
Epoch [20/50]  Loss: 0.0730   Accuracy: 97.60%
Epoch [21/50]  Loss: 0.0684   Accuracy: 98.00%
Epoch [22/50]  Loss: 0.0536   Accuracy: 97.60%
Epoch [23/50]  Loss: 0.0682   Accuracy: 97.20%
Epoch [24/50]  Loss: 0.0663   Accuracy: 97.20%
Epoch [25/50]  Loss: 0.0616   Accuracy: 98.00%
```

```
Epoch [26/50]  Loss: 0.0218   Accuracy: 99.20
Epoch [27/50]  Loss: 0.0315   Accuracy: 98.40
Epoch [28/50]  Loss: 0.0215   Accuracy: 99.20
Epoch [29/50]  Loss: 0.0124   Accuracy: 99.60
Epoch [30/50]  Loss: 0.0127   Accuracy: 99.60
Epoch [31/50]  Loss: 0.0086   Accuracy: 99.60
Epoch [32/50]  Loss: 0.0082   Accuracy: 99.60
Epoch [33/50]  Loss: 0.0094   Accuracy: 99.60
Epoch [34/50]  Loss: 0.0074   Accuracy: 99.60
Epoch [35/50]  Loss: 0.0076   Accuracy: 99.60
Epoch [36/50]  Loss: 0.0068   Accuracy: 99.60
Epoch [37/50]  Loss: 0.0068   Accuracy: 99.20
Epoch [38/50]  Loss: 0.0063   Accuracy: 100.0
Epoch [39/50]  Loss: 0.0062   Accuracy: 99.60
Epoch [40/50]  Loss: 0.0056   Accuracy: 100.0
Epoch [41/50]  Loss: 0.0083   Accuracy: 99.20
Epoch [42/50]  Loss: 0.0058   Accuracy: 99.60
Epoch [43/50]  Loss: 0.0072   Accuracy: 99.60
Epoch [44/50]  Loss: 0.0051   Accuracy: 99.60
Epoch [45/50]  Loss: 0.0073   Accuracy: 99.60
Epoch [46/50]  Loss: 0.0087   Accuracy: 99.60
Epoch [47/50]  Loss: 0.0070   Accuracy: 99.20
Epoch [48/50]  Loss: 0.0093   Accuracy: 99.60
Epoch [49/50]  Loss: 0.0047   Accuracy: 100.0
Epoch [50/50]  Loss: 0.0052   Accuracy: 99.60
```

e. Accuracy on the training set (with screenshots)

```
Accuracy on training set: 99.60%
```

References:

[1] ResNet-18 实现 Cifar-10 图像分类 https://blog.csdn.net/sunqiande88/article/details/80100891