DL Project

EL Image Classification

과목명	딥러닝	**학과**	컴퓨터학부
교수명	정우환	**학번**	2020052551
제출일	2024.11.28.	**이름**	성주원

1. Analysis of the Problem

The purpose of our project is to classify Electroluminescene(EL) images to detect fault cell or to evaluate the quality of solar panels. The dataset input size is 100*200. Since we need to classify into 2 classes(fault or normal), we will use binary classification.

2. List of Modification

Used Model: ResNet18, Used Weight: IMAGENET1K_V1

Loss function: BCEWithLogitLoss

Optimizer: Adamw

Else: K_fold, Xavier initialization, batch normalization, ReduceonLRplateau, dropout

3. Reasons for the usage

I've decided to use ResNet18, because it has pretrained weight(IMAGENET1K_V1). Also, instead of building my model, I thought using ResNet18 has more appropriate model for analyzing images. I've tried to use deeper model(32,50, etc), but their models all exceeds the 60MB. So for me, it was best to use pretrained model from ResNet18.(The default for ResNet18 is IMAGENET1K_V1)

```
# torchvision model
model = resnet18(weights=ResNet18_Weights.DEFAULT)
```

As we've analyzed the problem, we know this is binary classification problem so it would be best to use binary cross entropy loss function. Because we need to use sigmoid as the output function, we will just use BCEWithLogitLoss(Binary Cross Entropy Loss+Sigmoid) function.

```
def train(args, k_fold_loader, model):
    criterion = torch.nn.BCEWithLogitsLoss()
```

To reduce the vanishing gradient problem, we will use Adam, which is combination of momentum and RMSprop. The Adamw includes the L2 normalization, which helps model to prevent the overfitting problem. I've gave model parameters and learning rate(0.1) and weight decay as parameters.

```
optimizer = torch.optim.AdamW(model.parameters(), lr=args.learning_rate, weight_decay=1e-4)
```

To resolve the over-fitting problem and increase the performance, I've implemented K-Fold. The data_loader function is located in _utils.py. I've divided train, validation, test size 64%, 16%(args.k_folds=5), 20% respectively. First, we divide test and train/validation set into 20% and 80%. To prevent class imbalance problem, I've gave stratify parameter by dataset.target. ImageFolder library divides the class based on the folders. So this prevents the class imbalance problem. Within the 80%, which is train and validation data set, since I've set k_fold value into 5, so the train data set will be 64% and validation data set will be 16% and as folds iterate, the fold changes.

```python
def make_data_loader(args):
    # Get Dataset
    dataset = datasets.ImageFolder(args.data, transform=custom_transform)

    # 1. Divide dataset
    labels=dataset.targets
    train_val_indices, test_indices = train_test_split(
        range(len(dataset)),
        test_size=0.2,
        random_state=42,
        stratify=labels
    )

    train_val_dataset = Subset(dataset, train_val_indices)
    test_dataset = Subset(dataset, test_indices)

    # 2. Dataloader for test set
    test_loader = DataLoader(test_dataset, batch_size=args.batch_size, shuffle=False)

    # 3. Divide train and validation set for K-Fold
    kfold = KFold(n_splits=args.k_folds, shuffle=True, random_state=42)

    kfold_loaders = []

    fold_idx = 0

    for train_idx, val_idx in kfold.split(train_val_dataset.indices):
        print(f"Training fold {fold_idx + 1}/{kfold.get_n_splits()}")

        train_subset = Subset(train_val_dataset, train_idx)
        val_subset = Subset(train_val_dataset, val_idx)

        # 4. Create dataloader for train and validation set
        train_loader = DataLoader(train_subset, batch_size=args.batch_size, shuffle=True)
        val_loader = DataLoader(val_subset, batch_size=args.batch_size, shuffle=False)

        kfold_loaders.append((train_loader, val_loader))

        fold_idx += 1

    # 5. Return k-fold loaders(test, validation) and test loader
    return kfold_loaders, test_loader
```

At the training function, k_fold_loader will be passed as parameter and will be used to enumerate n_splits(args.k_folds=5)

```
for fold, (train_loader, val_loader) in enumerate(k_fold_loader):
    print(f"[Fold {fold + 1}] Training the model...")
```

For the best start, we have 2 choices for the weight initialization. He or Xavier initialization. But Since we are using Sigmoid function in the BCEWithLogitLoss function, Xavier is more efficient when using Sigmoid function. So I've used Xavier initialization for weight initialization. We will initialize bias into 0.

```
def init_weights(model):
    if isinstance(model, nn.Linear):
        init.xavier_uniform_(model.weight)
        if model.bias is not None:
            init.zeros_(model.bias)

def train(args, k_fold_loader, model):
    criterion = torch.nn.BCEWithLogitsLoss()

    model.apply(init_weights)
```

By using ResNet18, the batch normalization is implemented. As we can see from the picture, the batch normalization is located in between convolutional layer and activation function to normalize the means and variance before using Relu function. By using this, we can prevent vanishing gradient problem.
(part of ResNet function)

```
_log_api_usage_once(self)
if norm_layer is None:
    norm_layer = nn.BatchNorm2d
self._norm_layer = norm_layer
self.conv1 = nn.Conv2d(3, self.inplanes, kernel_s
self.bn1 = norm_layer(self.inplanes)
self.relu = nn.ReLU(inplace=True)
 x = self.conv1(x)
 x = self.bn1(x)
 x = self.relu(x)
 x = self.maxpool(x)
```

To dynamically change the learning rate parameter, I've used ReduceonLRplateau.This function is used when the validation accuracy does not increase. When the validation is not getting better for 3 consecutive times, it will divide the learning rate in 10.

```
scheduler = ReduceLROnPlateau(optimizer, mode='min', factor=0.1, patience=3, verbose=True)
```

To prevent overfitting problem, I've implmented dropout function within ResNet18. It will dropouut with probability of 0.3.

(Part of ResNet function)

```
        self.relu = nn.ReLU(inplace=True)
        self.dropout = nn.Dropout(0.3)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
```

(Part of forward function)

```
        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.dropout(x)
```

4. How Learning works

When we run train.py first, we will add path to our model and image as argument. If we don't have GPU that we can use to train our model, we will use CPU instead. We can change hyper parameter values.

```python
if __name__ == '__main__':

    parser = argparse.ArgumentParser(description='2024 DL Term Project')
    parser.add_argument('--save-path', default='checkpoints/', help="Model's state_dict")
    parser.add_argument('--data', default='test_image/', type=str, help='data folder')
    args = parser.parse_args()

    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
    args.device = device
    num_classes = 1

    # hyperparameters
    args.epochs = 30
    args.learning_rate = 0.1
    args.batch_size = 256
    args.k_folds = 5
```

We will load our data but to prevent different input size, we will resize the image into 100*200. After, to apply k-fold, we will divide our dataset into test and train, validation data set. After, we will return them.(Detailed explanation is in the 3)
(This is part of _utils.py)

```python
custom_transform = transforms.Compose([
    transforms.Resize((100, 200)),
    transforms.ToTensor()
])


def make_data_loader(args):
    # Get Dataset
    dataset = datasets.ImageFolder(args.data, transform=custom_transform)

    # 1. Divide dataset
    labels=dataset.targets
    train_val_indices, test_indices = train_test_split(
        range(len(dataset)),
        test_size=0.2,
        random_state=42,
        stratify=labels
    )

    train_val_dataset = Subset(dataset, train_val_indices)
    test_dataset = Subset(dataset, test_indices)

    # 2. Dataloader for test set
    test_loader = DataLoader(test_dataset, batch_size=args.batch_size, shuffle=False)

    # 3. Divide train and validation set for K-Fold
    kfold = KFold(n_splits=args.k_folds, shuffle=True, random_state=42)

    kfold_loaders = []

    fold_idx = 0

    for train_idx, val_idx in kfold.split(train_val_dataset.indices):
        print(f"Training fold {fold_idx + 1}/{kfold.get_n_splits()}")

        train_subset = Subset(train_val_dataset, train_idx)
        val_subset = Subset(train_val_dataset, val_idx)

        # 4. Create dataloader for train and validation set
        train_loader = DataLoader(train_subset, batch_size=args.batch_size, shuffle=True)
        val_loader = DataLoader(val_subset, batch_size=args.batch_size, shuffle=False)

        kfold_loaders.append((train_loader, val_loader))

        fold_idx += 1

    # 5. Return k-fold loaders(test, validation) and test loader
    return kfold_loaders, test_loader
```

We will load our divided data set and bring the ResNet18 model and its weight. After, we will change 512 output into 1 output to make into binary classification. Then, we will train out model based on our configuration.

```
# Make Data loader and Model
k_fold_loader,_ = make_data_loader(args)

# torchvision model
model = resnet18(weights=ResNet18_Weights.DEFAULT)

num_features = model.fc.in_features
model.fc = nn.Sequential(
    nn.Linear(num_features, 1)
)
model.to(device)
print(model)


# Training The Model
train(args, k_fold_loader, model)
```

(Before going into train function)

We will print our accuracy based on binary classification. If the prediction is larger than 0.5, we will classift the prediction as 1. Else, we will consider it as 0. Then, we will return the accuracy. We will initialize the weight and bias by using init_weight function.

```
def acc(pred, label):
    pred = (pred >= 0.5).float()
    return torch.sum(pred == label).item()


def init_weights(model):
    if isinstance(model, nn.Linear):
        init.xavier_uniform_(model.weight)
        if model.bias is not None:
            init.zeros_(model.bias)
```

We will use BCEWithLogitLoss function as loss function. We will use AdamW optimizer and use ReduceLROnPlateau function to dynamically change our learning rate. To keep track of best accuracy, we will use best_val_acc variable.

```
def train(args, k_fold_loader, model):
    criterion = torch.nn.BCEWithLogitsLoss()

    optimizer = torch.optim.AdamW(model.parameters(), lr=args.learning_rate, weight_decay=1e-4)

    scheduler = ReduceLROnPlateau(optimizer, mode='min', factor=0.1, patience=3)

    best_val_acc = 0
```

If we save our model with optimizer and scheduler status, it exceeds over 100mb so I've decided to separate the weight only model and model for learning. If the model is found successfully, it will run from where we stopped. If not, it will just start from beginning. But before just starting, we will use our init_weight function for best start.

```
try:
    checkpoint = torch.load(f'{args.save_path}/model(learn).pth')
    model.load_state_dict(checkpoint['model_state_dict'])
    optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
    scheduler.load_state_dict(checkpoint['scheduler_state_dict'])
    print("Previously saved model loaded successfully.")
except FileNotFoundError:
    init_weights(model)
    print("No previously saved model found, training from scratch.")
```

We will iterate our training with k folds and epoch. We will set model into training mode and we will do back-propagation and forward operation. Then, we will calculate our accuracy with our acc function.

```
for fold, (train_loader, val_loader) in enumerate(k_fold_loader):
    print(f"[Fold {fold + 1}] Training the model...")

    for epoch in range(args.epochs):
        train_losses = []
        train_acc = 0.0
        total=0
        print(f"[Epoch {epoch+1} / {args.epochs}]")

        model.train()
        pbar = tqdm(train_loader)
        for i, (x, y) in enumerate(pbar):
            image = x.to(args.device)
            label = y.to(args.device).float().squeeze()
            optimizer.zero_grad()


            output = model(image).squeeze()
            loss = criterion(output, label)
            loss.backward()
            optimizer.step()

            train_losses.append(loss.item())
            total += label.size(0)

            train_acc += acc(output, label)
```

To evaluate out model with validation set, we will set our model into evaluation mode. After, we will check the prediction accuracy based on our model prediction.

```python
model.eval()
val_losses = []
val_acc = 0.0
val_total = 0

with torch.no_grad():
    for x, y in tqdm(val_loader, desc='Validation'):
        image = x.to(args.device)
        label = y.to(args.device).float().squeeze()
        label = label.squeeze()

        output = model(image).squeeze()
        loss = criterion(output, label)

        val_losses.append(loss.item())
        val_total += label.size(0)

        preds = (output >= 0.5).float()
        val_acc += (preds == label).sum().item()
```

After each evaluation, we need to forward our scheduler to help it check the prediction accuracy. If it doesn't improve 3 times consecutively, it will factor learning rate by 10. After, if validation accuracy is higher than any other validation accuracy, we will save our weight only model and learning model(includes optimizer and scheduler state dict). We will continue this iteration until the end

```python
        scheduler.step(epoch_val_loss)

        if epoch_val_acc > best_val_acc:
            best_val_acc = epoch_val_acc
            torch.save(model.state_dict(), f'{args.save_path}/model.pth')
            torch.save({
                'model_state_dict': model.state_dict(),
                'optimizer_state_dict': optimizer.state_dict(),
                'scheduler_state_dict': scheduler.state_dict(),
            }, f'{args.save_path}/model(learn).pth')
            print(f"Saved best model with validation accuracy: {best_val_acc:.2f}%")
    print(f"Last best model with validation accuracy: {best_val_acc:.2f}%")
```