Comparing Three Models' Performances on the Fine -grained Bird Classification

Presenter: Anni Li, Ruoxuan Li

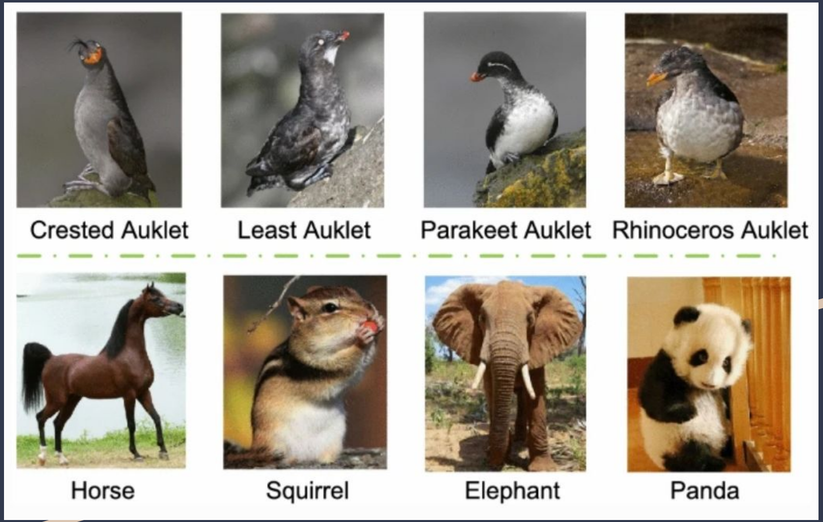
Github link:
https://github.com/AnniLi1212/Fine_Grained_Bird_Classification.git

Goals

- Design our CNN models to classify images according to different bird species
- Compare the performances of our models with a well-established model
- Gain a deeper understanding of the implementation of Pytorch, Tensorflow and other related tools for deep learning

Backgrounds

Fine-grained classification (top) v.s. general classification (bottom)



https://link.springer.com/article/10.1007/s13042-021-01473-8

- **Fine-Grained Image Classification** is a task in computer vision where the goal is to classify images into subcategories within a larger category. For example, classifying different species of birds or different types of flowers.

- This task is considered to be fine-grained because it requires the model to distinguish between subtle differences in visual appearance and patterns, making it more challenging than regular image classification tasks.

Data

- The dataset we chose comes from the Kaggle platform (https://www.kaggle.com/datasets/veeralakrishna/200-bird-species-with-11788-images?resource=download) and is named Caltech-UCSD Birds-200-2011. This dataset is an extended version of the CUB-200 dataset and consists of 200 bird species with 11,788 images. In this dataset, each image has 15 part locations, 312 binary attributes, and 1 bounding box.

```python
# split the dataset into training, validation, and testing and create data loaders
train_data = data[data["is_train"] == 1]
train_data = train_data.reset_index(drop=True)
train_data, val_data = train_test_split(train_data, test_size=0.2, random_state=42)

train_data = train_data.reset_index(drop=True)
trainset = CUBdata2(train_data, parts_locs, transform=transform)
trainloader = DataLoader(trainset, batch_size=8, shuffle=True)

val_data = val_data.reset_index(drop=True)
valset = CUBdata2(val_data, parts_locs, transform=transform)
valloader = DataLoader(valset, batch_size=8, shuffle=False)

test_data = data[data["is_train"] == 0]
test_data = test_data.reset_index(drop=True)
testset = CUBdata2(test_data, parts_locs, transform=transform)
testloader = DataLoader(testset, batch_size=8, shuffle=False)
```

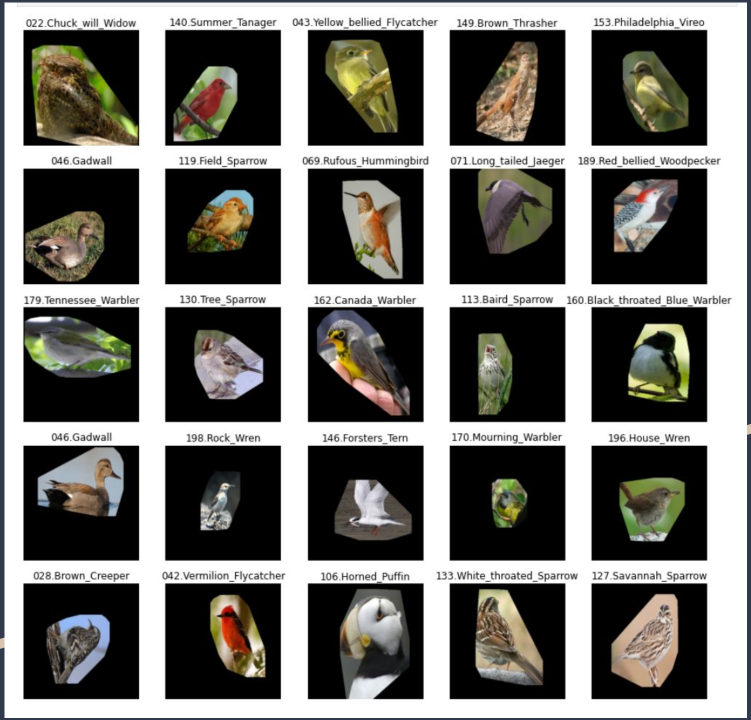
Model 1

```python
# define model 1: regular CNN
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()

        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, padding=1)

        self.pool = nn.MaxPool2d(2,2)

        self.fc1 = nn.Linear(32 * 112 * 112, 200)

        self.flatten = nn.Flatten()

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.flatten(x)
        x = F.relu(self.fc1(x))
        return x
```

- This model is a convolutional neural network model that is trained on the full bird images.

- This CNN is constructed with two convolutional layers, a max-pooling layer, and a fully connected layer. We used ReLU as the activation function in this basic neural network.

Model 2



- This model is a convolutional neural network model that is trained images which only include bird bodies using the information from parts.txt.
- We masked out the background according to the outermost part points and ignore one outermost point to prevent outlier points.

Model 2, continued

```python
# define model 2: CNN with mask
class Net2(nn.Module):
    def __init__(self):
        super(Net2, self).__init__()

        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(16)

        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm2d(32)

        self.conv3 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.bn3 = nn.BatchNorm2d(64)

        self.pool = nn.MaxPool2d(2,2)

        self.fc1 = nn.Linear(64 * 56 * 56, 512)
        self.fc2 = nn.Linear(512, 200)

        self.flatten = nn.Flatten()
        self.leaky_relu = nn.LeakyReLU(0.1)
        self.dropout = nn.Dropout(0.5)

    def forward(self, x):
        x = self.pool(self.leaky_relu(self.bn1(self.conv1(x))))
        x = self.pool(self.leaky_relu(self.bn2(self.conv2(x))))
        x = self.pool(self.leaky_relu(self.bn3(self.conv3(x))))

        x = self.flatten(x)
        x = self.dropout(x)
        x = self.leaky_relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)
        return x
```

- In addition to using the masked input, model 2 also has some improvements. It has three convolutional layers followed by batch normalization.

- After this, each convolution layer is processed by a leaky relu and finally the output is flattened and dropped out and go to a fully connected layer, followed by a dropout and fc again.

Model 3

```
# define the model 3: ResNet50
class Net3(nn.Module):
    def __init__(self, num_classes=200):
        super(Net3, self).__init__()
        self.resnet = resnet50(pretrained=True)
        self.resnet.fc = nn.Linear(self.resnet.fc.in_features, num_classes)

    def forward(self, x):
        return self.resnet(x)
```

- This model uses the ResNet50 model that is pre-trained on the ImageNet dataset, which contains more than one million images across 1000 different classes.

- By leveraging the knowledge gained from this pre-training, the ResNet-50 model can be fine-tuned on smaller datasets, such as the CUB-200-2011 dataset

Training (Part I)

```python
# define the hyperparameter learning tool
def objective2(trial):
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    net2 = Net2()
    net2.to(device)
    loss_func = nn.CrossEntropyLoss().to(device)

    lr = trial.suggest_loguniform("lr", 1e-5, 1e-1)
    momentum = trial.suggest_float("momentum", 0.5, 0.99)
    opt = optim.SGD(net2.parameters(), lr=lr, momentum=momentum)

    num_epoch = 5

    for epoch in range(num_epoch):
        running_loss = 0.0
        for i, trainstat in enumerate(trainloader, 0):
            inputs, ytrue = trainstat
            inputs, ytrue = inputs.to(device), ytrue.to(device)
            opt.zero_grad()

            outputs = net2(inputs)
            loss = loss_func(outputs, ytrue)
            loss.backward()
            opt.step()
            running_loss += loss.item()

        avg_loss = running_loss / len(trainloader)

    return avg_loss
```

```python
# learn the hyperparameters
study2 = optuna.create_study(direction="minimize")
study2.optimize(objective2, n_trials=5)
```

- We applied optuna for the first two models to select the best hyperparameters. Since the process takes too long, we set optuna to run 5 epochs with 6 trials

- Through manual practices, we found the cross entropy loss is the best loss function and the SGD is better than Adam as Adam would cause more severe over-fitting

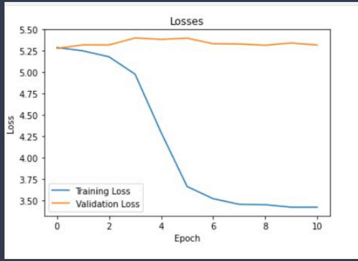
```python
loss_func = nn.CrossEntropyLoss().to(device)
opt = optim.SGD(net2.parameters(),
        lr=best_params2["lr"], momentum=best_params2["momentum"])
```

Training (Part II)

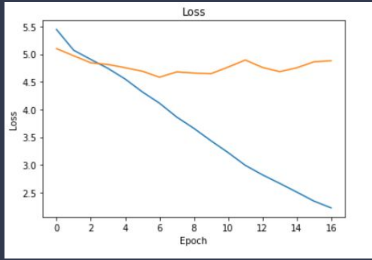
```python
# begin training
train_avg_losses = []
val_avg_losses = []
num_epoch = 100
patience = 10
best_val_loss = float('inf')
best_model = None
counter = 0

for epoch in range(num_epoch):
    net2.train()
    train_running_loss = 0.0
    train_num_iterations = 0

    for i, trainstat in enumerate(trainloader, 0):
        inputs, ytrue = trainstat
        inputs, ytrue = inputs.to(device), ytrue.to(device)
        opt.zero_grad()

        outputs = net2(inputs)
        loss = loss_func(outputs, ytrue)
        loss.backward()
        opt.step()

        train_running_loss  += loss.item()
        train_num_iterations += 1

    train_avg_loss = train_running_loss / train_num_iterations
    train_avg_losses.append(train_avg_loss)

    # validation
    net2.eval()
    val_running_loss  = 0.0
    val_num_iterations  = 0

    with torch.no_grad():
        for i, valstat in enumerate(valloader, 0):
            inputs, ytrue = valstat
            inputs, ytrue = inputs.to(device), ytrue.to(device)

            outputs = net2(inputs)
            loss = loss_func(outputs, ytrue)

            val_running_loss += loss.item()
            val_num_iterations += 1

    val_avg_loss = val_running_loss / val_num_iterations
    val_avg_losses.append(val_avg_loss)

    print(f'Epoch: {epoch + 1}, Training Loss: {train_avg_loss:.3f}, Validation Loss: {val_avg_loss:.3f}')

    # early stopping
    if val_avg_loss < best_val_loss:
        best_val_loss = val_avg_loss
        best_model = copy.deepcopy(net2)
        counter = 0
    else:
        counter += 1
        print(f'EarlyStopping counter: {counter} out of {patience}')
        if counter >= patience:
            print("Early stopping triggered. Stopping training.")
            break

print('Finished Training')
net2 = best_model
```

- The learning rate and momentum are chosen by optuna. For model 1, lr=0.0036 and momentum=0.9129; for model 2, lr = 0.0066 and momentum=0.6489.
- During the training and validation loop, we applied an early stopping with patience=10 to prevent the model from overfitting.
- For model 3, we chose Adam as the optimizer with learning rate=0.001 and weight decay=0.0001 to prevent overfitting because optuna took too long to finish.

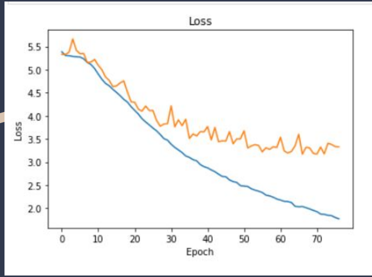
Results



Model 1
Accuracy: 1%
Precision: 0.001
Recall: 0.014



Model 2
Accuracy: 7%
Precision: 0.075
Recall: 0.070



Model 3
Accuracy: 28%
Precision: 0.334
Recall: 0.290

- Traditional CNNs with simple structure could not do fine-grained classification well, because their ability to effectively extract important features is limited.

- The results of our experiment and comparison indicated that the ResNet50 was the most suitable model for the fine-grained bird classification task.

Limitation

- We might have underestimated the complexity of the task and we failed to discuss the feasibility of this project with the instructor team while planning.

- We ran into huge difficulties while training the dataset due to the heavy GPU requirements from all three models.

- The results we obtained might not be an accurate representation of the true performance of the models as we could not compare the performances of the models on the same basis

Thank you!