0. Introduction

name: 蔡懷恩
ID: 112550020

1. Implementation

1.1. Data Loading

builds label mappings and loads data based on folder names.

```python
def load_train_dataset(path: str='data/train/')->Tuple[List, List]:
    # (TODO) Load training dataset from the given path, return images and labels
    images = []
    labels = []
    label_dict = {0: "elephant", 1: "jaguar", 2: "lion", 3: "parrot", 4: "penguin"} # number and name

    for label_idx, label_name in label_dict.items():
        folder_path = os.path.join(path, label_name)        # folder_path
        if not os.path.isdir(folder_path):
            continue
        for filename in os.listdir(folder_path):            # find filename
            if filename.lower().endswith(('jpg', 'jpeg', 'png')): # make name lowercase ,check if it is the image
                images.append(os.path.join(folder_path, filename)) # path and filename save into images
                labels.append(label_idx)                    # index save into labels

    return images, labels

def load_test_dataset(path: str='data/test/')->List:
    # (TODO) Load testing dataset from the given path, return images
    images = []
    for filename in os.listdir(path):
        if filename.lower().endswith(('jpg', 'jpeg', 'png')): # make name lowercase ,check if it is the image
            images.append(os.path.join(path, filename)) # path and filename save into images
    return images
```

1.2. Design Model Architecture

design CNN's forward: Conv → BN → ReLU → Pooling → Conv → BN → ReLU → Pooling → flatten → 128-dimensional feature vector → ReLU → dropout → 5 logits

```python
class CNN(nn.Module):
    def __init__(self, num_classes=5):
        # (TODO) Design your CNN, it can only be less than 3 convolution layers
        super(CNN, self).__init__()
        # 1st Convolution Layer：input 3 channel (RGB) , output 16 feature map , kernel = 3x3 , padding = 1 →  keep size (224x224)
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=16, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(16) # Normalize output values, speed up training and improve stability
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2) # For each non-overlapping 2x2 block, take max → size halved (112x112)
        # 2nd Convolution Layer: input 16 channels, output 32 channels
        self.conv2 = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm2d(32) # Normalize output values, speed up training and improve stability
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2) # For each non-overlapping 2x2 block, take max → size halved (56x56)

        # two pooling layers: 224 -> 112 -> 56, resulting in 32 x 56 x 56
        self.fc1 = nn.Linear(32 * 56 * 56, 128) # make 128-dimensional feature vector
        self.dropout = nn.Dropout(0.5)          # Randomly set 50% of neurons to zero
        self.fc2 = nn.Linear(128, num_classes)  # make 5 logits

    def forward(self, x):
        # (TODO) Forward the model
        x = self.pool(F.relu(self.bn1(self.conv1(x))))  # Each block: Conv → BN → ReLU → Pooling
        x = self.pool(F.relu(self.bn2(self.conv2(x))))  # Each block: Conv → BN → ReLU → Pooling
        x = x.view(x.size(0), -1)                       # make flatten
        x = F.relu(self.fc1(x))                         # make 128-dimensional and ReLU
        x = self.dropout(x)                             # Randomly set 50% of neurons to zero
        x = self.fc2(x)                                 # make 5 classes
        return x
```

1.3. Define function train(), validate() and test()

train(): Runs training in batches with loss computation and gradient updates.

validate(): Evaluates the model on validation data, returning average loss and accuracy.

test(): Runs inference on the test set and writes predictions to CSV.

```
def train(model: CNN, train_loader: DataLoader, criterion, optimizer, device)->float:
    # (TODO) Train the model and return the average loss of the data, we suggest use tqdm to know the progress
    model.train()                                       # Set model to training mode
    running_loss = 0.0                                  # Accumulate total loss
    for images, labels in tqdm(train_loader, desc="Training", leave=False):
        images, labels = images.to(device), labels.to(device) # Move data to device

        optimizer.zero_grad()                           # Reset gradients before backward pass
        outputs = model(images)                         # CNN Forward pass
        loss = criterion(outputs, labels)               # Compute loss
        loss.backward()                                 # Backward pass to compute gradients
        optimizer.step()                                # Update model parameters(according to gradients)

        running_loss += loss.item() * images.size(0)    # Accumulate loss

    avg_loss = running_loss / len(train_loader.dataset) # Compute average loss over entire dataset
    return avg_loss
```

```
def validate(model: CNN, val_loader: DataLoader, criterion, device)->Tuple[float, float]:
    # (TODO) Validate the model and return the average loss and accuracy of the data, we suggest use tqdm to know the progress
    model.eval()                                        # Set the model to evaluation mode
    # initialize
    running_loss = 0.0
    correct = 0
    total = 0

    with torch.no_grad():                               # Disable gradient computation (faster and saves memory)
        for images, labels in tqdm(val_loader, desc="Validating", leave=False):
            images, labels = images.to(device), labels.to(device) # Move data to device

            outputs = model(images)                     # CNN Forward pass
            loss = criterion(outputs, labels)           # Compute loss
            running_loss += loss.item() * images.size(0)# accumulate loss

            _, predicted = torch.max(outputs, 1)        # Get the predicted class

            # Update total and correct prediction count
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    # Calculate average loss and accuracy
    avg_loss = running_loss / len(val_loader.dataset)
    accuracy = correct / total
    return avg_loss, accuracy
```

```python
def test(model: CNN, test_loader: DataLoader, criterion, device):
    # (TODO) Test the model on testing dataset and write the result to 'CNN.csv'
    model.eval()                                    # Set the model to evaluation mode
    predictions = []
    ids = []

    with torch.no_grad():                           # Disable gradient computation (faster and saves memory)
        for images, image_ids in tqdm(test_loader, desc="Testing", leave=False):
            images = images.to(device)              # Move data to device
            outputs = model(images)                 # CNN Forward pass
            _, predicted = torch.max(outputs, 1)    # Get the predicted class
            predictions.extend(predicted.cpu().numpy()) # Store predictions
            ids.extend(image_ids)                   # Store image IDs

    # Create a DataFrame to store results in the CSV
    df = pd.DataFrame({"id": ids, "prediction": predictions})
    df.to_csv("CNN.csv", index=False)
    print(f"Predictions saved to 'CNN.csv'")
    return
```

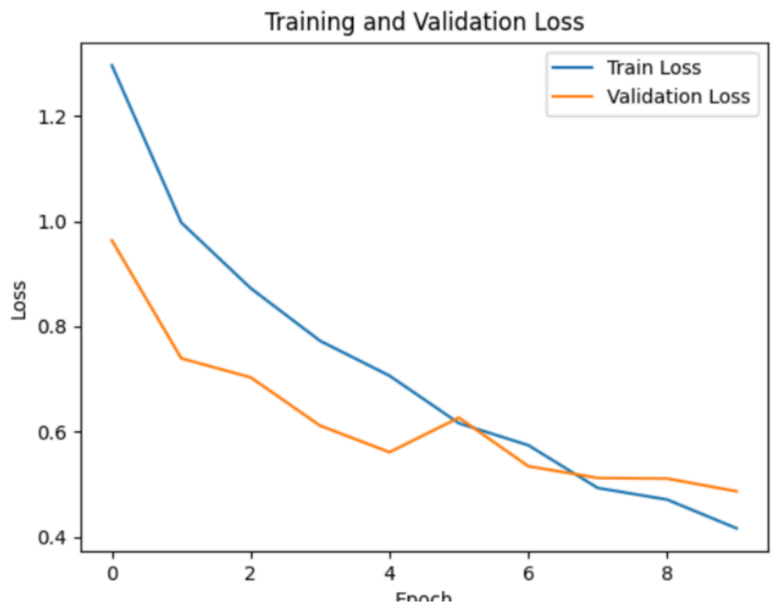
1.4. Printing Training Logs

```python
for epoch in range(EPOCHS): #epoch
    logger.info(f"Epoch {epoch+1}/{EPOCHS}") # add by myself
    train_loss = train(model, train_loader, criterion, optimizer, device) # train
    val_loss, val_acc = validate(model, val_loader, criterion, device) # Evaluate on validation set

    logger.info(f"Train Loss: {train_loss:.4f} | Val Loss: {val_loss:.4f} | Val Acc: {val_acc:.4f}") # add by myself

    # Store training and validation loss for plotting
    train_losses.append(train_loss)
    val_losses.append(val_loss)

    # (TODO) Print the training log to help you monitor the training process
    #        You can save the model for future usage
    # Save the model if it achieves the best validation accuracy so far
    if val_acc > max_acc:
        max_acc = val_acc
        torch.save(model.state_dict(), "best_cnn.pth") # Save best model weights
        logger.info("Best model updated")

logger.info(f"Best Accuracy: {max_acc:.4f}")
```

1.5. Plot Training and Validation Loss

```python
def plot(train_losses: List, val_losses: List):
    # (TODO) Plot the training loss and validation loss of CNN, and save the plot to 'loss.png'
    #        xlabel: 'Epoch', ylabel: 'Loss'
    plt.figure()
    plt.plot(train_losses, label='Train Loss')
    plt.plot(val_losses, label='Validation Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()
    plt.title('Training and Validation Loss')
    plt.savefig('loss.png')
    print("Save the plot to 'loss.png'")
```

1.6. Experiments



From the loss curve (loss.png), overfitting started to occur **epoch 5**.
method 1 **Dropout:** 0.5 dropout layer was added, which prevents the model from co-adapting too much to training data, reducing validation loss.

```python
def forward(self, x):
    # (TODO) Forward the model
    x = self.pool(F.relu(self.bn1(self.conv1(x))))  # Each block: Conv → BN → ReLU → Pooling
    x = self.pool(F.relu(self.bn2(self.conv2(x))))  # Each block: Conv → BN → ReLU → Pooling
    x = x.view(x.size(0), -1)                       # make flatten
    x = F.relu(self.fc1(x))                         # make 128-dimensional and ReLU
    x = self.dropout(x)                             # Randomly set 50% of neurons to zero
    x = self.fc2(x)                                 # make 5 classes
    return x
```

method 2 **Best model saving:** ensuring the final result uses the most generalized version of the model.

```python
# Save the model if it achieves the best validation accuracy so far
if val_acc > max_acc:
    max_acc = val_acc
    torch.save(model.state_dict(), "best_cnn.pth") # Save best model weights
    logger.info("Best model updated")
```

2. Decision Tree

2.1. Feature Extraction

This converts raw images into structured vectors for decision tree learning.

```python
def get_features_and_labels(model: ConvNet, dataloader: DataLoader, device)->Tuple[List, List]:
    # (TODO) Use the model to extract features from the dataloader, return the features and labels
    model.eval() # Set the model to evaluation mode
    features = []
    labels = []
    with torch.no_grad(): # Disable gradient computation (faster and saves memory)
        for images, batch_labels in tqdm(dataloader, desc="Extracting features"):
            images = images.to(device) # Move data to device
            output = model(images) # ConvNet Forward pass
            features.extend(output.cpu().numpy()) # Move outputs to CPU, convert to NumPy, and append
            labels.extend(batch_labels.numpy()) # Convert labels to NumPy and append
    return features, labels

def get_features_and_paths(model: ConvNet, dataloader: DataLoader, device)->Tuple[List, List]:
    # (TODO) Use the model to extract features from the dataloader, return the features and path of the images
    model.eval() # Set the model to evaluation mode
    features = []
    paths = []
    with torch.no_grad(): # Disable gradient computation (faster and saves memory)
        for images, image_ids in tqdm(dataloader, desc="Extracting features"):
            images = images.to(device) # Move data to device
            output = model(images) # ConvNet Forward pass
            features.extend(output.cpu().numpy()) # Move outputs to CPU, convert to NumPy, and append
            paths.extend(image_ids) # Store image identifiers for result mapping
    return features, paths
```

2.2. Model Architecture

_build_tree(): Recursively grows the tree.
predict(): For each sample x in X, recursively traverse the tree to get a prediction
_predict_tree(): Recursive function to traverse the decision tree
_split_data(): Use Information Gain to find the best split for a dataset
_entropy(): Calculate the entropy
_best_split(): Selects the threshold with best info gain.
_information_gain(): Measures improvement from a split.
_majority_class(): Used to determine the predicted class label at a leaf node

```python
def _build_tree(self, X: pd.DataFrame, y: np.ndarray, depth: int):
    # (TODO) Grow the decision tree and return it
    if len(set(y)) == 1 or depth == self.max_depth: # If all samples have the same label or max depth is reached
        return {'label': self._majority_class(y)}

    best_feature, best_threshold = self._best_split(X, y)

    if best_feature is None:                    # If no valid split is found, return a leaf node
        return {'label': self._majority_class(y)}

    self.progress.update(1)                     # Update the training progress

    # Split the dataset into left and right subsets
    left_X, left_y, right_X, right_y = self._split_data(X, y, best_feature, best_threshold)

    # Return a non-leaf node containing split info and two child subtrees
    return {
        'feature': best_feature,
        'threshold': best_threshold,
        'left': self._build_tree(left_X, left_y, depth + 1),
        'right': self._build_tree(right_X, right_y, depth + 1)
    }
```

```python
def predict(self, X: pd.DataFrame)->np.ndarray:
    # (TODO) Call _predict_tree to traverse the decision tree to return the classes of the testing dataset
    # For each sample x in X, recursively traverse the tree to get a prediction
    # Collect all predictions into a NumPy array and return
    return np.array([self._predict_tree(x, self.tree) for x in X])
```

```python
def _predict_tree(self, x, tree_node):
    # (TODO) Recursive function to traverse the decision tree
    if 'label' in tree_node:                    # Base case: if the current node is a leaf, return its label
        return tree_node['label']

    # If the feature value is less than or equal to the threshold, go to the left subtree
    if x[tree_node['feature']] <= tree_node['threshold']:
        return self._predict_tree(x, tree_node['left'])
    else:
        return self._predict_tree(x, tree_node['right'])
```

```python
def _split_data(self, X: pd.DataFrame, y: np.ndarray, feature_index: int, threshold: float):
    # (TODO) split one node into left and right node
    left_dataset_X, left_dataset_y, right_dataset_X, right_dataset_y = [], [], [], []
    for xi, yi in zip(X, y):
        if xi[feature_index] <= threshold:          # If the selected feature value is less than or equal to the threshold
            left_dataset_X.append(xi)
            left_dataset_y.append(yi)
        else:
            right_dataset_X.append(xi)
            right_dataset_y.append(yi)
    return left_dataset_X, left_dataset_y, right_dataset_X, right_dataset_y
```

```python
def _best_split(self, X: pd.DataFrame, y: np.ndarray):
    # (TODO) Use Information Gain to find the best split for a dataset
    best_gain = -1                              # Track the highest information gain found
    best_feature_index = None                   # Index of the best feature for splitting
    best_threshold = None                       # Threshold value for the best split
    n_features = len(X[0])                      # Number of features in the dataset

    for feature_index in range(n_features):
        thresholds = sorted(set([x[feature_index] for x in X])) # Extract all unique values for this feature

        # Optional optimization: limit the number of thresholds to at most 20
        if len(thresholds) > 20:
            step = max(1, len(thresholds) // 20)
            thresholds = thresholds[::step]

        for threshold in thresholds:
            # Split y into left and right subsets based on threshold
            left_y = [yi for xi, yi in zip(X, y) if xi[feature_index] <= threshold]
            right_y = [yi for xi, yi in zip(X, y) if xi[feature_index] > threshold]

            # Compute information gain from this split
            gain = self._information_gain(y, left_y, right_y)

            # Update best split if a higher gain is found
            if gain > best_gain:
                best_gain = gain
                best_feature_index = feature_index
                best_threshold = threshold

    return best_feature_index, best_threshold
```

```python
def _entropy(self, y: np.ndarray)->float:
    # (TODO) Return the entropy
    counts = np.bincount(y)                     # Count the number of occurrences of each class label
    probs = counts / len(y)                     # Convert counts to probabilities by dividing by total number of samples
    return -np.sum([p * np.log2(p) for p in probs if p > 0])
```

```python
# add by myself
def _information_gain(self, parent_y, left_y, right_y):
    if len(left_y) == 0 or len(right_y) == 0:       # If either child is empty, the split is useless → return 0 gain
        return 0
    parent_entropy = self._entropy(parent_y)        # Entropy before the split
    left_entropy = self._entropy(left_y)            # Entropy of the left subset
    right_entropy = self._entropy(right_y)          # Entropy of the right subset
    p_left = len(left_y) / len(parent_y)            # Proportion of samples going to the left child
    p_right = len(right_y) / len(parent_y)          # Proportion of samples going to the right child
    return parent_entropy - (p_left * left_entropy + p_right * right_entropy) # Parent entropy − Weighted child entropy
```

```python
# add by myself
def _majority_class(self, y):
    # np.bincount(y): counts the number of occurrences for each integer label
    # argmax(): returns the index (i.e., the class label) with the highest count
    return np.bincount(y).argmax()
```

2.3. Experiment

```
tree_model = DecisionTree(max_depth=7)

PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS                                    bash - hw3

Predictions saved to 'DecisionTree.csv'
wayne777aa@MSI:~/work/AI/hw3$ python3 main.py
2025-04-20 23:07:35.738 | INFO     | __main__:main:24 - Start loading data
2025-04-20 23:07:35.750 | INFO     | __main__:main:94 - Start training Decision Tree
Extracting features: 100%|| 171/171 [00:53<00:00,  3.20it/s]
Growing tree:  96%|| 123/128 [01:42<00:04,  1.20it/s]
Extracting features: 100%|| 43/43 [00:12<00:00,  3.37it/s]
2025-04-20 23:10:24.915 | INFO     | __main__:main:110 - Validation Accuracy: 0.7652
Extracting features: 100%|| 54/54 [00:16<00:00,  3.27it/s]
Predictions saved to 'DecisionTree.csv'
```

depth = 7

```
tree_model = DecisionTree(max_depth=5)

PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS                                    bash - hw3

Predictions saved to 'DecisionTree.csv'
wayne777aa@MSI:~/work/AI/hw3$ python3 main.py
2025-04-20 23:02:49.207 | INFO     | __main__:main:24 - Start loading data
2025-04-20 23:02:49.238 | INFO     | __main__:main:94 - Start training Decision Tree
Extracting features: 100%|| 171/171 [01:01<00:00,  2.77it/s
Growing tree:  97%|| 31/32 [01:03<00:02,  2.05s/it
Extracting features: 100%|| 43/43 [00:13<00:00,  3.12it/s
2025-04-20 23:05:08.815 | INFO     | __main__:main:110 - Validation Accuracy: 0.7352
Extracting features: 100%|| 54/54 [00:16<00:00,  3.22it/s
Predictions saved to 'DecisionTree.csv'
wayne777aa@MSI:~/work/AI/hw3$
```

depth = 5: Validation accuracy decreases.
It makes fewer splits (less expressive power)
Some complex or borderline samples may be misclassified due to insufficient depth.

```
tree_model = DecisionTree(max_depth=9)

PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS                                    bash - hw3

Predictions saved to 'DecisionTree.csv'
wayne777aa@MSI:~/work/AI/hw3$ python3 main.py
2025-04-20 23:13:47.751 | INFO     | __main__:main:24 - Start loading data
2025-04-20 23:13:47.763 | INFO     | __main__:main:94 - Start training Decision Tree
Extracting features: 100%|| 171/171 [00:52<00:00,  3.26it/s]
Growing tree:  50%|| 254/512 [02:09<02:11,  1.96it/s]
Extracting features: 100%|| 43/43 [00:12<00:00,  3.55it/s]
2025-04-20 23:17:02.053 | INFO     | __main__:main:110 - Validation Accuracy: 0.7996
Extracting features: 100%|| 54/54 [00:15<00:00,  3.45it/s]
Predictions saved to 'DecisionTree.csv'
```

depth = 9: Validation accuracy increases.
The tree becomes more complex and may memorize the training data, which may cause overfitting.