

# AI Hw3 report

112550097 楊弘奕

## 1、CNN

### 1. Data Loading

#### a. load\_train\_dataset:

Using a string array as path for each iteration to go through folders in train folder, and for images in each folder, set idx as their label number, idx goes from 0 to 4.

```
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 = ['elephant', 'jaguar', 'lion', 'parrot', 'penguin']  
    idx = 0  
    img_labels = []  
    for label in labels:  
        label_path = os.path.join(path, label)  
        for image_name in os.listdir(label_path):  
            image_path = os.path.join(label_path, image_name)  
            images.append(image_path)  
            img_labels.append(idx)  
        idx += 1  
    return images, img_labels
```

#### b. load\_test\_dataset:

Join the image file name with folder path, and return the path of the image.

```
def load_test_dataset(path: str='data/test/')->List:  
    # (TODO) Load testing dataset from the given path, return images  
    images = []  
    for image_name in os.listdir(path):  
        image_path = os.path.join(path, image_name)  
        images.append(image_path)  
    return images
```

### 2. Design Model Architecture

#### a. `_init_()` : Architecture of CNN model.

- (1) The model consists of 3 convolutional layers with 32, 64, and 128 filters respectively, each using a kernel size of 3×3, stride 1, and padding 1 if out of boarding.
- (2) Each convolution layer includes ReLU and MaxPool with size 2x2 and
- (3) The input image of size 224×224×3 is transformed through the convolutional and pooling operations as follows:

- Conv1 → ReLU → Pooling: 112×112×32
- Conv2 → ReLU → Pooling: 56×56×64
- Conv3 → ReLU → Pooling: 28×28×128

The resulting feature map is then flattened into a 1D vector of size 28×28×128 to serve as the input to the first fully connected layer.

- (4) fc1 maps the 28x28x128 1D vector input to a 1D vector of size 512
- (5) fc2 maps the 512 1D vector input to 5 labels
- (6) softmax converts raw logits into class probabilities for prediction with formula:

$$\text{Softmax}(z_i) = e^{z_i} / \sum_{j=1}^C e^{z_j}, z = \text{score of class } i, C = \text{number of classes}$$

```
def __init__(self, num_classes=5):
    # (TODO) Design your CNN, it can only be less than 3 convolution layers
    super(CNN, self).__init__()
    # convolution layers
    self.conv1 = nn.Conv2d(3, 32, kernel_size=3, stride=1, padding=1)
    self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)
    self.conv3 = nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1)

    self.relu = nn.ReLU()
    self.pool = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)

    # fully connected layers
    # input size 224x224x3
    # data 224x224x3 -> 112x112x32 -> 56x56x64 -> 28x28x128
    self.fc1 = nn.Linear(128 * 28 * 28, 512)
    self.fc2 = nn.Linear(512, num_classes)
    self.softmax = nn.Softmax(dim=1)
```

#### b. forward(): Process of prediction

- (1) Do 3 layers of convolution
- (2) Call ReLU and MaxPool at the end of each convolution layer

- (3) Flatten X from third layer of convolution to a 1D vector
- (4) The first fully connected layer (fc1) reduces the flattened feature vector from  $128 \times 28 \times 28 = 100352$  to 512 dimensions, followed by a ReLU activation function to introduce non-linearity.
- (5) The second fully connected layer (fc2) maps the 512 dimension vector to 5 output values, which represent the logit scores of the 5 classes.
- (6) A softmax layer converts these logits into probabilities, allowing the model to output a prediction over the 5 classes.

```
def forward(self, x):  
    # (TODO) Forward the model  
    x = self.conv1(x)  
    x = self.relu(x)  
    x = self.pool(x)  
    x = self.conv2(x)  
    x = self.relu(x)  
    x = self.pool(x)  
    x = self.conv3(x)  
    x = self.relu(x)  
    x = self.pool(x)  
  
    # flatten -> fully connected layer  
    x = x.view(x.size(0), -1)  
    x = self.fc1(x)  
    x = self.relu(x)  
    x = self.fc2(x)  
    x = self.softmax(x)  
    return x
```

### 3. Define function train(), validate() and test()

- a. **train()**: Train the model using first 80% of training dataset and update the model parameters based on the calculated gradients.

- (1) Take data with a batch\_size = 32 (define in main.py)
- (2) Send data to device
- (3) Clear the accumulated gradients from the previous iteration using optimizer.zero\_grad().
- (4) Feed images in and get output from the CNN model
- (5) Calculate the cross-entropy loss between the model's output and the ground truth label by taking the negative log of the predicted probability for the true class.

$$P(c_i) = e^{z_i} / \sum_{j=1} e^{z_j} \text{ (by softmax)}$$

$$\text{Loss} = -\log(P(\text{true class}))$$

- (6) Perform backpropagation using `loss.backward()` to compute the gradients of the loss with respect to all learnable parameters in the model.
- (7) Update the model's parameters using `optimizer.step()`, based on the computed gradients.
- (8) Accumulate the total training loss using `loss × batch_size` for each batch.
- (9) Compute the average training loss by dividing the total loss by the total number of training samples.

```
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 t
    model.train()
    total_loss = 0
    total_sample_sz = 0
    for images, labels in tqdm(train_loader, desc="Training \ \ \"):
        # move data to device
        images, labels = images.to(device), labels.to(device)
        # zero the parameter gradients for each batch
        optimizer.zero_grad()
        # forward data
        outputs = model(images)
        # calculate cross entropy loss between outputs and labels
        loss = criterion(outputs, labels)
        # back propagation
        loss.backward()
        # update weights
        optimizer.step()

        total_loss += loss.item() * labels.size(0)
        total_sample_sz += labels.size(0)
    avg_loss = total_loss / total_sample_sz
    return avg_loss
```

**b. validate():** Validate the model using last 20% of training dataset

- (1) Switch to evaluation mode using `model.eval()` to disable dropout
- (2) Disable gradient computation with `torch.no_grad()` to accelerate the process
- (3) Compute the average validation loss using the same logic as in `train()`, but without updating weights.
- (4) Select the prediction with the highest probability (i.e. argmax from softmax output) as the final predicted class.
- (5) Sum up correct prediction to get overall accuracy

```
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
    # no dropout and no gradient calculation
    model.eval()
    total_loss = 0
    total_samples = 0
    correct = 0
    with torch.no_grad():
        for images, labels in tqdm(val_loader, desc="Validation \ \ \"):
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            loss = criterion(outputs, labels)
            total_loss += loss.item() * labels.size(0)
            total_samples += labels.size(0)
            _, predicted = torch.max(outputs.data, 1)
            for i in range(labels.size(0)):
                if predicted[i] == labels[i]:
                    correct += 1
    avg_loss = total_loss / total_samples
    accuracy = correct / total_samples
    return avg_loss, accuracy
```

c. **test()**: Test pretrained model on test dataset, store result to csv file.

- (1) Same as in validate(), switch to evaluation mode using model.eval() and disable gradient computation to speed up the process.
- (2) Select the class with the highest predicted probability as the final predicted label.
- (3) Store each prediction and corresponding image into CNN.csv, following the required format with column headers ['id', 'prediction'].

```
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()
    res = []
    with torch.no_grad():
        for batch in tqdm(test_loader, desc="Testing \ \ \"):
            images, names = batch
            images = images.to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            for i in range(len(names)):
                res.append({'id': names[i], 'prediction': predicted[i].item()})
    # store in CSV
    with open('CNN.csv', mode='w', newline='') as csv_file:
        header = ['id', 'prediction']
        writer = csv.DictWriter(csv_file, fieldnames=header)
        writer.writeheader()
        writer.writerows(res)
    print("save to CNN.csv")
    return
```

## 4. Printing Training Logs

- a. For each epoch, print
  - (1) Current epoch and total number of epochs
  - (2) Average training loss
  - (3) Average validation loss
  - (4) Validation accuracy
- b. If the current validation accuracy surpasses the previously recorded best accuracy, save the model as a checkpoint (best\_model.pth) and update the best accuracy.

```
for epoch in range(EPOCHS): #epoch
    train_loss = train(model, train_loader, criterion, optimizer, device)
    val_loss, val_acc = validate(model, val_loader, criterion, device)

    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
    logger.info(f"Epoch {epoch + 1} of {EPOCHS}, Train_Loss: {train_loss:.4f}, Validation_Loss: {val_loss:.4f}, Validation_Accuracy: {val_acc:.4f}")
    if val_acc > max_acc:
        max_acc = val_acc
        # save best model
        torch.save(model.state_dict(), "best_model.pth")
        logger.info(f"Model saved at epoch {epoch + 1}")

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

## 5. Plot Training and Validation Loss

- a. X-axis: epoch index
- b. Y-axis: loss
- c. Use different color to show legend of Train Loss and Validation Loss
- d. Save as loss.png

```
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'
    #     title: 'Training and Validation Loss'
    epochs = []
    for i in range(len(train_losses)):
        epochs.append(i+1)
    plt.figure(figsize=(10, 10))
    plt.plot(epochs, train_losses, Label='Train Loss', marker='o')
    plt.plot(epochs, val_losses, Label='Validation Loss', marker='o')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.title('Training and Validation Loss')
    plt.legend()
    plt.grid(True)
    plt.tight_layout()
    plt.savefig('loss.png')
    print("Save the plot to 'loss.png'")
    return
```

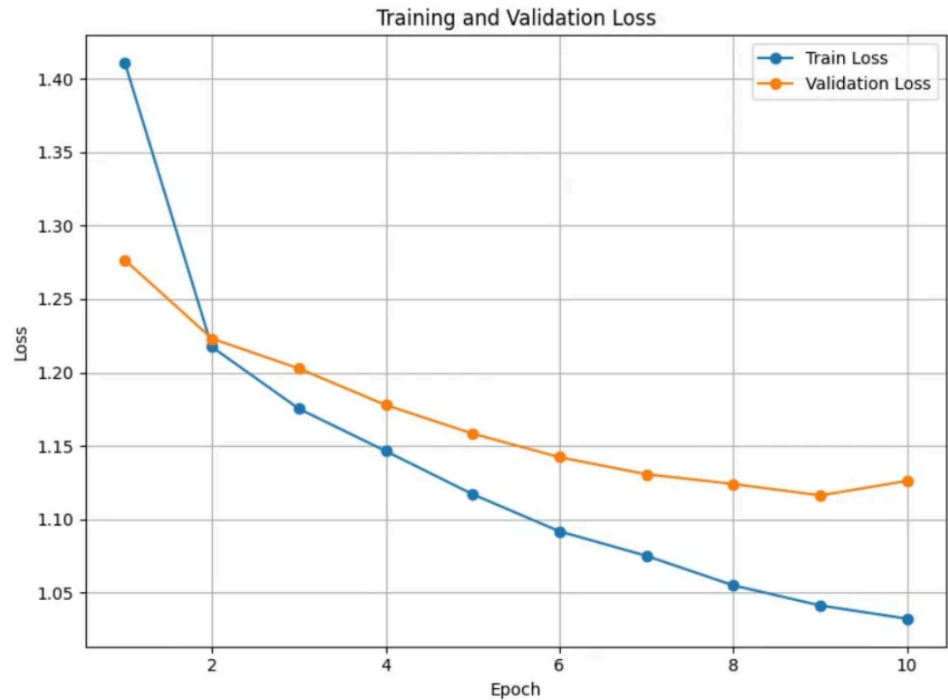
## 6. Experiments

### a. Without handling overfitting

#### i. When did overfitting happened?

1. From epoch 1 to 6, both training loss and validation loss steadily decrease, indicating that the model is learning patterns that generalize well to novel data.
2. After epoch 6, the training loss continues to decrease, but the validation loss gradually no longer decrease and even slightly increases (epoch 9–10), which suggests:
  - a. The model is still improving on the training data.
  - b. It is no longer improving, and even worsening, on the validation data.
  - c. This divergence between the two losses indicates that the model starts to memorize training data rather than learning generalizable patterns.

#### ii. loss plot



**b. Dropout:**

In the fully connected layers, dropout is used to prevent neurons from relying too heavily on specific features. By randomly disabling some neurons during training, the network is forced to learn more robust and general representations, as each neuron must learn to work with a wider variety of inputs.

i. implementation

Add a dropout layer with probability 0.3 in the `__init__()` function, and apply it after the first fully connected layer.



```

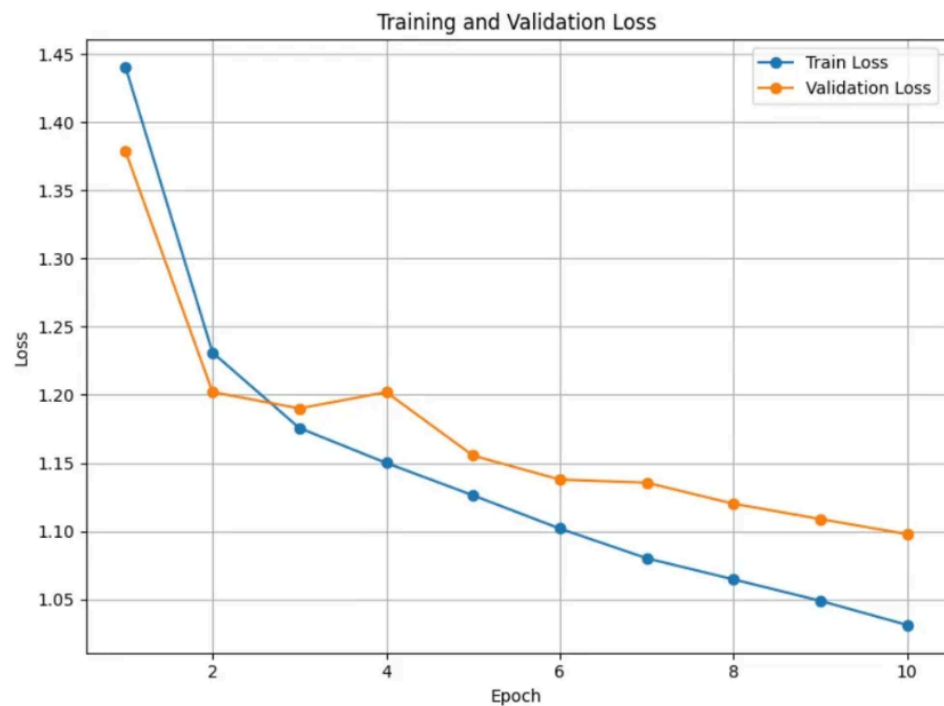
# dropout
self.dropout = nn.Dropout(0.3)

def forward(self, x):
    # (TODO) Forward the model
    x = self.conv1(x)
    x = self.relu(x)
    x = self.pool(x)
    x = self.conv2(x)
    x = self.relu(x)
    x = self.pool(x)
    x = self.conv3(x)
    x = self.relu(x)
    x = self.pool(x)

    # flatten -> fully connected layer
    x = x.view(x.size(0), -1)
    x = self.fc1(x)
    x = self.relu(x)
    x = self.dropout(x)
    x = self.fc2(x)
    x = self.softmax(x)
    return x

```

ii. loss plot



c. Data Augment:

By introducing slight variations, the model is exposed to a wider range of input conditions, making it less likely to memorize specific features of the training data. Instead, it learns more generalized patterns, leading to better performance on unseen validation or test data.

i. implementation

I implement data augmentation only on train\_dataset, so I add an additional parameter to decide whether to implement or not.

```
train_dataset = TrainDataset(train_images, train_labels, True)
val_dataset = TrainDataset(val_images, val_labels)
```

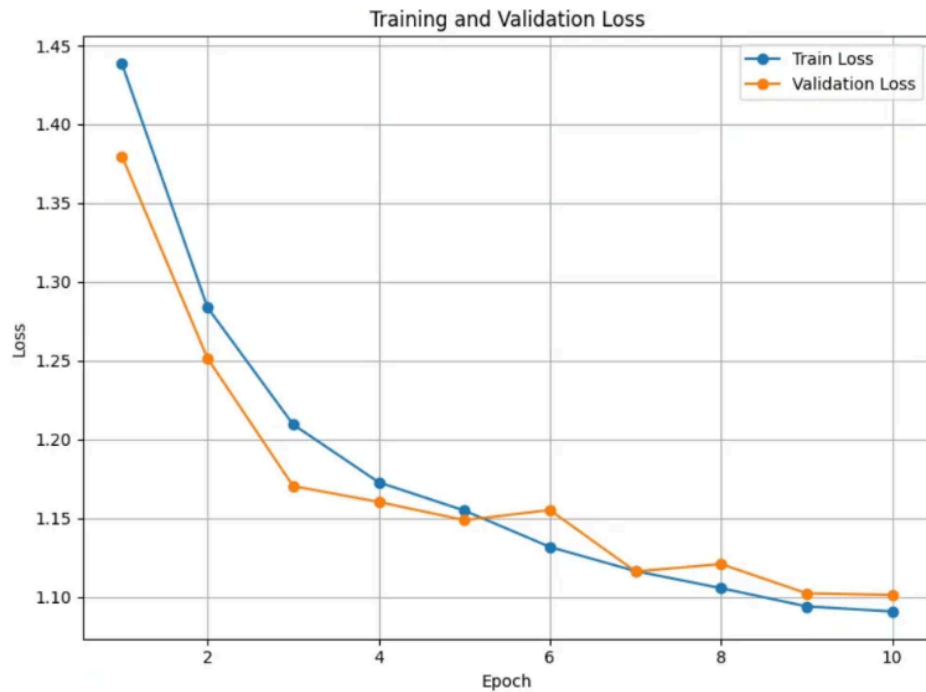
I implement data augmentation in the `__init__()` function of the `TrainDataset` class. When the transform flag is set to `True`, two augmentation techniques are applied:

1. `RandomResizedCrop(224, scale=(0.8, 1.0), ratio=(0.75, 1.0))`: randomly crops and resizes the image to simulate different viewpoints and object scales.
2. Flips the image horizontally with a 30% probability to introduce mirrored variations.

```
def __init__(self, images, labels, transform=False):
    if transform:
        self.transform = transforms.Compose([
            transforms.RandomResizedCrop(224, scale=(0.8, 1.0), ratio=(0.75, 1.0)),
            transforms.RandomHorizontalFlip(p=0.3),
            transforms.Grayscale(num_output_channels=3),
            transforms.ToTensor()
        ])
    else:
        self.transform = transforms.Compose([
            transforms.Grayscale(num_output_channels=3),
            transforms.Resize((224, 224)),
            transforms.ToTensor()
        ])

    self.images, self.labels = images, labels
```

ii. loss plot



d. Accuracy:

As shown in the Kaggle results, the model that applies dropout and data augmentation to prevent overfitting achieves better performance when dealing with unseen images.

Specifically, the model without any overfitting prevention techniques reached an accuracy of 0.801, while the models using dropout and data augmentation both achieved 0.819.

This demonstrates that incorporating regularization strategies helps the model generalize better to novel data.

✓	CNN.csv Complete · 21h ago · without handling overfitting	0.801
✓	CNN.csv Complete · 1d ago · dropout	0.819
✓	CNN.csv Complete · 1d ago · data augmentation	0.819

## 2、Decision Tree

### 1. Feature\_Extraction

a. **get\_features\_and\_labels()**: Extract features of images using a pretrained ConvNet model. These feature vectors, along with the ground truth labels, are used to train the decision tree.

- i. Switch model to evaluation mode, to disable dropout
- ii. Avoid gradient calculation to accelerate the process
- iii. First send images to device, get the output logits from pretrained ConvNet model
- iv. The model output is transferred back to the CPU and converted to a NumPy array by `.cpu().numpy()`
- v. The feature vectors and their corresponding labels are appended into features and labels lists respectively
- vi. The function returns a pandas.DataFrame containing the extracted feature vectors and a corresponding NumPy array of ground truth labels.

```
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()  
    features, labels = [], []  
    with torch.no_grad():  
        for imgs, label in tqdm(dataloader, desc="Extracting features & labels"):  
            imgs = imgs.to(device)  
            # convert tensor to numpy array  
            out = model(imgs).cpu().numpy()  
            # append to Logist from model to features  
            features.extend(out)  
            # append to Labels from dataloader to Labels  
            labels.extend(label.numpy())  
    return pd.DataFrame(features), np.array(labels)
```

b. **get\_features\_and\_paths()**: Extract features of test images, and pass them to the trained Decision Tree to make predictions. The predicted labels and corresponding image paths are then saved to a CSV file.

- i. Switch model to evaluation mode, to disable dropout
- ii. Avoid gradient calculation to accelerate the process
- iii. First send images to device, get the output logits from pretrained ConvNet model
- iv. The outputs are converted to NumPy arrays using `.cpu().numpy()` and appended to the feature list. Corresponding image paths are appended to the path list

- v. The function returns a pandas.DataFrame of extracted feature vectors and a list of corresponding image paths, which can be used to store predictions in a CSV file later.

```
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 paths
    model.eval()
    features, paths = [], []
    with torch.no_grad():
        for imgs, path in tqdm(dataloader, desc="Extracting features & paths"):
            imgs = imgs.to(device)
            out = model(imgs).cpu().numpy()
            features.extend(out)
            paths.extend(path)
    return pd.DataFrame(features), paths
```

## 2. Model Architecture

- a. **\_build\_tree()**: A recursively constructs the decision tree by splitting the dataset based on the feature and threshold that yields the highest information gain. Terminates and returns a leaf node when a stopping condition is met.
  - i. Terminate condition:  
Stop building the tree and return a leaf node if any of the following is true:
    - 1.  $\text{depth} \geq \text{max\_depth}$  – the maximum tree depth is reached.
    - 2.  $y$  contains only one unique class label.
    - 3. The number of samples is less than 2 ( $X.\text{shape}[0] < 2$ ).
  - ii. Use **\_best\_split()** to determine the optimal feature\_index and threshold that maximizes information gain.
  - iii. If **\_best\_split()** returns None, indicating no valid feature to split, create a leaf node with the majority class label.
  - iv. Use **\_split\_data()** to divide the dataset into left and right subsets based on the best feature and threshold.
  - v. If either the left or right subset is empty after the split, return a leaf node. This suggests the current node's samples cannot be further separated meaningfully.
  - vi. Add node with feature\_index, threshold, and left, right child node
  - vii. Update progressbar

```

def _build_tree(self, X: pd.DataFrame, y: np.ndarray, depth: int):
    # (TODO) Grow the decision tree and return it
    # terminate
    maxDepth = depth >= self.max_depth
    same_class = len(np.unique(y)) == 1
    min_samples = X.shape[0] < 2
    if maxDepth or same_class or min_samples:
        return {'label': int(np.bincount(y).argmax())}

    # use best_split to split data
    best_feature_idx, best_threshold = DecisionTree._best_split(X, y)
    # if no best feature
    if best_feature_idx is None:
        return {'label': int(np.bincount(y).argmax())}
    # split data
    left_X, left_y, right_X, right_y = DecisionTree._split_data(X, y, best_feature_idx, best_threshold)
    # if no split
    if left_X.shape[0] == 0 or right_X.shape[0] == 0:
        return {'label': int(np.bincount(y).argmax())}

    # create node
    node = {
        'feature_index': best_feature_idx,
        'threshold': best_threshold,
        'left': self._build_tree(left_X, left_y, depth + 1),
        'right': self._build_tree(right_X, right_y, depth + 1)
    }
    self.progress.update(1)
    return node

```

b. **predict():** Iterates over each input sample and traverses the decision tree using `_predict_tree()` to generate prediction labels.

- i. Traverse each sample, for each row in the input DataFrame X, use `_predict_tree()` to traverse the pre-built decision tree and determine the predicted class label.
- ii. Store prediction and return tensor

```

def predict(self, X: pd.DataFrame) -> np.ndarray:
    # (TODO) Call _predict_tree to traverse the decision
    predictions = []
    for _, row in X.iterrows():
        prediction = self._predict_tree(row, self.tree)
        predictions.append(prediction)
    return np.array(predictions)

```

- iii. Because the return datatype here is already numpy array, there's no need to convert to numpy array in main.py

```

results = []
for image_name, prediction in zip(test_paths, test_predictions):
    results.append({'id': image_name, 'prediction': prediction})
df = pd.DataFrame(results)
df.to_csv('DecisionTree.csv', index=False)
print(f"Predictions saved to 'DecisionTree.csv'")

```

- c. **\_predict\_tree()**: Recursive function that navigates the decision tree from root to leaf based on feature thresholds. Returns the predicted label when a leaf node is reached.
- i. If `tree_node` contains a 'label' key, it indicates a leaf node. The function returns this label as the prediction.
  - ii. The function uses `feature_index` and `threshold` stored in the node to determine whether the sample `x` should go to the left or right subtree:
    - 1. If `x.iloc[feature_index] <= threshold`, go to the left subtree.
    - 2. Otherwise, go to the right subtree.

```
def _predict_tree(self, x, tree_node):  
    # (TODO) Recursive function to traverse the decision tree  
    # Leaf node  
    if 'label' in tree_node:  
        return tree_node['label']  
    # Split data  
    feature_index = tree_node['feature_index']  
    threshold = tree_node['threshold']  
    # Left or right  
    if x.iloc[feature_index] <= threshold:  
        return self._predict_tree(x, tree_node['left'])  
    else:  
        return self._predict_tree(x, tree_node['right'])
```

- d. **\_split\_data()**: Splits the dataset into left and right subsets based on whether each sample's feature value is less than or greater than a given threshold.
- i. First convert dataframe to numpy array to accelerate computation
  - ii. A mask is created to indicate whether each sample's value in the selected feature is less than or equal to the given threshold
  - iii. Split the data based on mask, append `features_val <= threshold` to the `left_dataset`, while others to `right_dataset`

```
def _split_data(X: pd.DataFrame, y: np.ndarray, feature_index: int, threshold: float):
    # (TODO) split one node into left and right node

    data = X.values # convert to Numpy
    feature_val = data[:, feature_index]
    mask = feature_val <= threshold
    left_dataset_X = pd.DataFrame(data[mask], columns=X.columns).reset_index(drop=True)
    right_dataset_X = pd.DataFrame(data[~mask], columns=X.columns).reset_index(drop=True)
    left_dataset_y = y[mask]
    right_dataset_y = y[~mask]
    return left_dataset_X, left_dataset_y, right_dataset_X, right_dataset_y
```

- e. **\_best\_split()**: For each feature, tries all possible thresholds between unique values and computes the corresponding information gain. Selects the feature index and threshold that maximize the gain.
- i. First convert dataframe to numpy array to accelerate computation
  - ii. The function computes the entropy of the entire dataset y before any split. This value serves as the baseline to compare potential splits:
  - iii. For each feature column, extract all unique values, if there is only one unique value, skip this feature as it cannot be used to split the data.
  - iv. Try each midpoints between adjacent values to use as candidate thresholds
  - v. For each threshold:
    1. Split the dataset into left\_y and right\_y based on whether feature values are less than or greater than the threshold
    2. Compute the entropy for each subset and use it to calculate the information gain
    3. If the current information gain exceeds the best one found so far, update the best feature index and threshold
  - vi. Eventually we'll get the best\_threshold for best feature\_idx to get maximum information gain



```
def _best_split(X: pd.DataFrame, y: np.ndarray):
    # (TODO) Use Information Gain to find the best split for a dataset
    data = X.values # convert to Numpy
    origin_entropy = DecisionTree._entropy(y)
    best_info_gain = 0
    best_feature_index = None
    best_threshold = None
    feature_sz = X.shape[1]
    for i in range(feature_sz):
        feature_values = np.unique(data[:, i])
        # if all values are the same, skip this feature
        if len(feature_values) <= 1:
            continue

        thr_candidates = (feature_values[:-1] + feature_values[1:]) / 2
        for threshold in thr_candidates:
            # 直接到ndarray型別做split_data
            mask = data[:, i] <= threshold
            left_y = y[mask]
            right_y = y[~mask]
            # calculate entropy
            left_entropy = DecisionTree._entropy(left_y)
            right_entropy = DecisionTree._entropy(right_y)
            # calculate information gain
            info_gain = origin_entropy - (len(left_y) / len(y)) * left_entropy - (len(right_y) / len(y)) * right_entropy
            if info_gain > best_info_gain:
                best_info_gain = info_gain
                best_feature_index = i
                best_threshold = threshold

    return best_feature_index, best_threshold
```

f. **\_entropy()**: Calculates the entropy of the label distribution in a dataset.




- i. The np.unique() function is used to identify all unique class labels in y, and return\_counts=True provides the frequency of each label. The result counts is an array of class frequencies.
- ii. The class frequencies are divided by the total number of samples to obtain the probability distribution of the labels.
- iii. The entropy is calculated using the Shannon entropy formula:

$$H(y) = - \sum_{j=1}^n p_i \log_2(p_i), p_i > 0$$

```
def _entropy(y: np.ndarray)->float:
    # (TODO) Return the entropy
    _, counts = np.unique(y, return_counts=True)
    probs = counts / counts.sum()
    entropy = 0.0
    # 累加每個類別的貢獻，避免 log2(0)
    for p in probs:
        if p > 0:
            entropy += -p * np.log2(p)
    return entropy
```

### 3. Experiment

#### a. Accuracy on kaggle

	DecisionTree.csv Complete · 1m ago · depth 9	0.778	<input type="checkbox"/>
	DecisionTree.csv Complete · 12m ago · depth 5	0.749	<input type="checkbox"/>
	DecisionTree.csv Complete · 4h ago · depth 7	0.787	<input type="checkbox"/>

## b. Why?

I think that as the decision tree depth increases from 5 to 7, the model improves its ability to learn more patterns from the features extracted from ConvNet, resulting in better prediction performance.


However, when the depth increases to 9, the accuracy slightly drops. This suggests that the model starts to memorize the dataset instead of learning generalizable feature patterns. As a result, its ability to predict unseen data declines. Therefore, depth 7 achieves a better balance between fitting the training data and generalizing to new inputs.

# 3、Kaggle

## 1.CNN

115	112550097		0.825	15	2h
-----	-----------	---	-------	----	----

## 2.Decision Tree

57	112550097		0.794	16	3h
----	-----------	---	-------	----	----