Intro-to-AI HW03:

Part 1. CNN:

- Load_training_dataset():

```python
"""Implement load_train_dataset process: Load training dataset from given path, and give every image their corresponding label
1. Load the image from the path "data/train/", the subfolders have already been classified
2. Create correspomding dictionary for the labels
3. Scanning all the subfolders from "data/train/", for every subfolder
    a. Get the label of the subfolder(like elephant)
    b. Use dictionary to turn the label into number,0->elephant, 1->jaguar, 2->lion, 3->parrot, 4->penguin
    c. Run over the images in the subfolder, and get the path of every image
"""
def load_train_dataset(path: str='content/data/train/')->Tuple[List, List]:
    # (TODO) Load training dataset from the given path, return images and labels
    images = []# All the training images corresponding path
    labels = []# Labels to the number,
    # Because we know the name and we want to find the corresponding number, so we design to let name be the key
    label_dict = {
        'elephant': 0,
        'jaguar': 1,
        'lion': 2,
        'parrot': 3,
        'penguin': 4
    }
    # Use os.walk() to scan all the subfolders, os.walk() can handle more than 1 layer
    for root, _, files in os.walk(path):
        """
        root: Current folder's path
        dirs: sub folders list
        files: All files in current folder
        """
        # Ignore the ./daata/train/, we want it's subfolder,there have images
        if root == path:
            continue
        #Extract the last name of folder
        label_name = os.path.basename(root)
        #Loop through all the files in this folder
        for file in files:
            # Filter the file to make sure only read .jpg and .png file
            if file.endswith(('.jpg', '.png')):
                # Put path and corresponding label in each list
                labels.append(label_dict[label_name])
                # Use os.path.join function to paste path and file's name together
                images.append(os.path.join(root, file))
    return images, labels
```

- Load_test_dataset():

```python
"""Implementation of load_data_set
1. Run all the images in the given folder
2. Store the path of each image to images and return it
"""
def load_test_dataset(path: str='content/data/test/')->List:
    # (TODO) Load testing dataset from the given path, return images
    images = []
    for root, _, files in os.walk(path):
        for file in files:
            if file.endswith(('.jpg', '.png')):
                images.append(os.path.join(root, file))
    return images

"""Implementation of plot function:
1. Plot the training loss vs epoch and validation loss vs epoch
2. Set the x-axis label to 'Epoch' and y-axis label to 'Loss'
3. Use blue line for training loss abd red line for validation loss
4. Save the plot to 'loss.png'
```

- __init__():

```
    def __init__(self, num_classes=5):
        # (TODO) Design your CNN, it can only be less than 3
convolution layers
        super(CNN, self).__init__()
        # Use nn.Conv2d to design the convolution layer,
nn.conv2d(in_channels, out_channels, kernel_size, stride, padding)
        #First convolution layer
        self.conv1 = nn.Conv2d(3, 32, kernel_size = 3, padding = 1)
# kernal_size = 3, means 3*3 kernal. padding=1, means input and
output kernal size is the same
        # Second convo;ution layer
        self.conv2 = nn.Conv2d(32, 64, kernel_size = 3, padding =
1)
        # Third convolution layer
        self.conv3 = nn.Conv2d(64, 128, kernel_size = 3, padding =
1)
        # Use torch.nn.MaxPool2d(kernal_size, stride=None,
padding=0, dilation=1) to design the maxpooling layer)
        self.pool = nn.MaxPool2d(kernel_size = 2)
        # Create a global average pool, which count eevery
channel's average value, means the intensity of the channel
        # Input: (128, 56, 56)-> Output: (128, 1, 1), take every
channel 56*56 matrix, and get the average value of every channel
        self.global_avg_pool = nn.AdaptiveAvgPool2d(1)
        #Create a fully connected layer, combining class score
based on the importance of each channel
        # Use torch.nn.Linear(in_features, out_featires,
bias=True), channel means in_features, out_features means the
number of classes
        self.fc = nn.Linear(128, num_classes)
```

- Foward():

```python
    def forward(self, x):
        # (TODO) Forward the model
        # Implement the structure of CNN: (Conv2d-> ReLU->MaxPool2d)*N -> Flatten -> Fully Connected Layer -> Softmax

        #Utilize F.relu to help CNN learn complicated patterns, and improve traning stability

        # x means the input of the model, which is a batch of images, and the shape is [B, 3, 224, 224], B means batch size, 3 means RGB channel, 224 means image size

        # First layer: [B,3,224,224] -> [B, 32, 224, 224]
        x = F.relu(self.conv1(x))
        x = self.pool(x) # [B, 32, 224, 224]->[B, 32, 112, 112]

        # Second layer:-> [B, 64, 112, 112]
        x = F.relu(self.conv2(x))
        x = self.pool(x)# ->[B, 64, 56, 56]

        # Third layer: -> [B, 128, 56, 56]
        x = F.relu(self.conv3(x))

        # Global average it:[B, 128, 56, 56] -> [B, 128, 1, 1]
        x = self.global_avg_pool(x)

        # Flatten it becaues Linear isn matrix multiplication and input take[batch_size, feature]
        x = x.view(x.size(0), -1)
        # x.view means reshape the tensor, x.size(0) means batch size, -1 means the rest of the dimension, which is 128*1*1=128

        #Categorize it 128->5
        s = self.fc(x)
        return s
```

- Train()

```python
"""Implmentation of train function: To update the model's weight
and bias, and return the average loss of the data
1. Set the model to the traning mode, and set the model to the
device(GPU)
2. Keep cycling over the training dataset.
3. Clear the gradient of optimizer, or else the gradient will
accumulate.
4. Foward pass the model, and get the output of the model.
5. Calculate the loss of the model, and backward pass the model to
get the gradient of th emodel
6. Update the model's parameter by optimizer.step()
7. Accumulate the loss of the model, and return the average loss of
the data

"""
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()


    total_loss = 0
    progress_bar = tqdm(train_loader, desc="Training")
    for images, labels in train_loader:
        # Take the data to the GPU
        images, labels = images.to(device), labels.to(device)

        # Clear the gradient of optimizer
        optimizer.zero_grad()

        # Foward pass the model, and get the output of the model
        outputs = model(images)

        # Use criterion(loss function) to calculate the difference
between the output and the label
        loss = criterion(outputs, labels)

        # Backward pass the model to get the gradient of the model
        loss.backward()
        # Use optimizer (optimizer.step() function) to update the
model's parameter
        optimizer.step()

        # Use loss.item() instead of loss is becauseto break the
graph, and get the value of the loss
        # loss.item() means the value of the loss, and it is a
float number
        total_loss += loss.item()
        progress_bar.set_postfix(loss=loss.item())
    avg_loss = total_loss / len(train_loader)
    return avg_loss
```

- Validate():

```python
"""Implementation of validate: Test the model on validation dataset
1. Set the model to the evaluation mode, and set the model to the
device(GPU)
2. Disable the gradient calculation, because we don't need to
update the model's parameter
3. Keep cycling over the validation dataset.
4. Calculate the loss of the model, and calculate the accuracy of
the model
5. Return the average loss and accuracy of the data
"""
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()
    total_loss, correct = 0, 0

    # Disable the gradient calculation, can save the meomry and
speed up the process
    with torch.no_grad():
        for images, labels in val_loader:
            # Take the data to the GPU
            images, labels = images.to(device), labels.to(device)
            # Foward pass the model, and get the output of the
model
            outputs = model(images)
            # Use criterion(loss function) to calculate the
difference between the output and label
            loss = criterion(outputs, labels)
            # Use loss.item() to break the graph, and get the value
of the loss
            total_loss += loss.item()

            # Calculate the accuracy of the model, the return value
of torch.max() is the maximum value and the index of the maximum
value
            # torch.max(output, dim), dim means the which direction
we want to find maximum value, 0 means the vertical direction, 1
means the horizontal direction
            _, predicted = torch.max(outputs, 1) # We want to check
maximum value in horizontal direction, so dim = 1
            # Calculate the number of correct prediction, and
accumulate it
            # .sum is to count the number of true in tensor, and
.item() is to transform the tensor to a float number
            # predicted and labels both value are tensor, so we
don't need to use .numpy() to transform it to numpy array
            correct += (predicted == labels).sum().item()

    # len(val_holder) means the number of epochs in the validation
dataset, and len(val_loader.dataset) means the number of samples in
the validation dataset
    avg_loss, accuracy = total_loss/len(val_loader), correct/
len(val_loader.dataset)
    return avg_loss, accuracy
```

- Test()

```
"""Implementation of test function: Test the model on testing
dataset and write the result to 'CNN.csv'
1. Set the model to evaluation mode
2. Disable the gradient calculation, because we don't need to
update the model's parameter
3. Create Dataloader for test dataset, because we need to test the
model on the test dataset
4. Keep cycling over the test dataset.
5. Foward pass the model, and get the output of the model

"""
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()
    result = []

    with torch.no_grad():
        for images, img_ids in test_loader:
            # Take the data to GPU
            images = images.to(device)
            # Foward pass the model, and get the output of the
model

            output = model(images)# The type of output is tensor,
and the shape is [B, 5], B means batch size, 5 means the number of
classes

            # Get the predicted class of the model, and get the
index of the maximum value
            _, predicted = torch.max(output, 1)
            # Append the predicted class and image id to the result
list
            #Use zip is to combine the result together, and the
result is a list of tuple, each tuple is (image_id, predicted
result)
            for img_id, pred in zip(img_ids,
predicted.cpu().numpy()):# .cpu() is to transfer the tensor to CPU,
and .numpy() is to transform the tensor to numpy array
                result.append({"id": img_id, "prediction":
int(pred)})
    # Turn the result to a pandas dataframe, and save it to
'CNN.csv'
    df = pd.DataFrame(result) #Turn the result to a pandas table
    df.to_csv('CNN.csv', index=False) # Save the result to
'CNN.csv'
    print(f"Predictions saved to 'CNN.csv'")
    return df
```

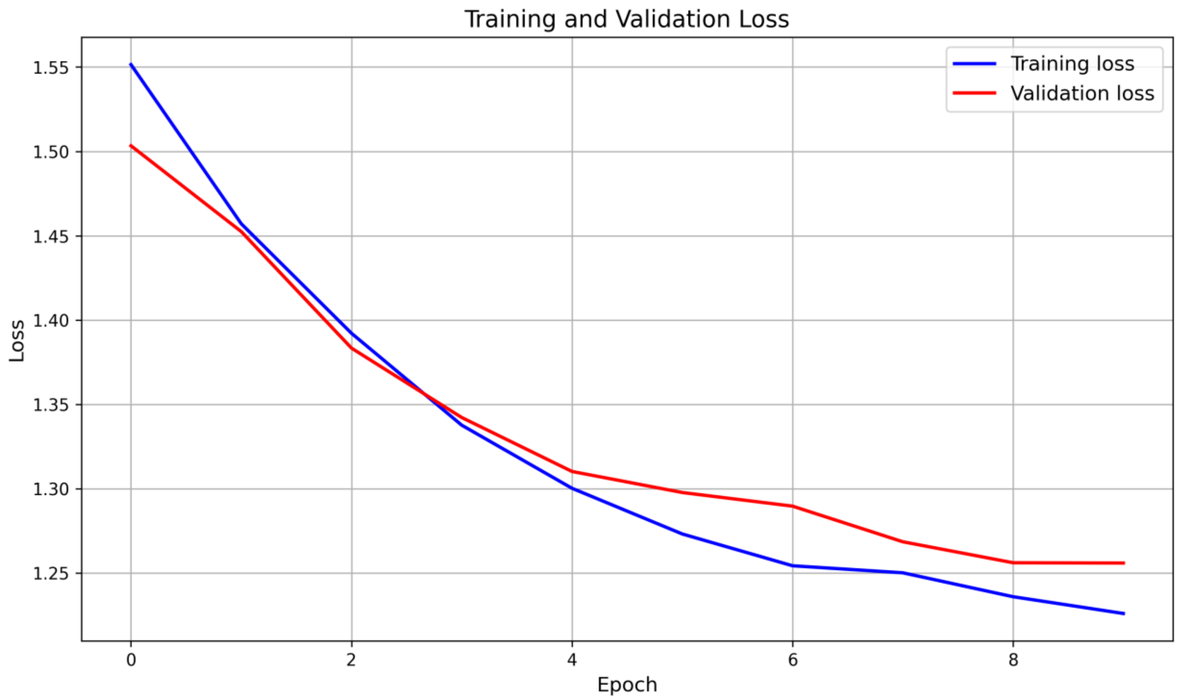
- Printing training log

```python
    for epoch in range(EPOCHS): #epoch
        """ Design the training log, to help us monitor and keep
track of the training process
        1. Print each epoch's time and training loss and validation
loss and validation accuracy
        2. Keep track of the best accuracy and sace the model if
the accuracy is better than the previous best accuracy

        """
        train_loss = train(model, train_loader, criterion,
optimizer, device)
        val_loss, val_acc = validate(model, val_loader, criterion,
device)

        # Append the result into each list
        train_losses.append(train_loss)
        val_losses.append(val_loss)
        val_accuracies.append(val_acc)

        # (TODO) Print the training log to help you monitor the
training process
        #        You can save the model for future usage

        # If the accuracy is better than the previous best
accuracy, save the model
        if val_acc > max_acc:
            max_acc = val_acc
            # Use torch.save() tp save the model, torch.save(obj,
f), other parameter just use default value
            # obj in here means the model, and f means the path to
save the model, better way is to store dict in obj
            #, we can save the model and other information in the
dict
            torch.save({
                'epoch': epoch, # Save times of training
                'model_state_dict': model.state_dict(), # Save all
the parameters of ther model, most important part
                'val_accuracy': val_acc # Save the accuracy of the
model
           }, 'best_model.pth')


        # Learning rate means how many change we wamt to make to
the model's parameter for every epoch
        # Higher learning rate means we wan to make more change to
the model's parameter for every epoch
        lr = optimizer.param_groups[0]['lr']

        # Print the training log use logger.info()
        # Add f is to let us get variable value in the string, and
we can use {} to get the variable value
        logger.info(
            f"Epoch: {epoch+1}/{EPOCHS} | "
            f"Time: {start_time}s | "
            f" Learning Rate: {lr:.2e} | " # use {:.2e} to print
the learning rate in scientific notation
            f"Train Loss: {train_loss:4f} | " #use {:.4f} to print
the loss in 4 decimal places
            f"Val Loss: {val_loss:.4f} | "
            f"Val Acc: {val_acc:.2%} | " # use{.2%} is to print
tyhe accuracy in percentage in 2 decimal places
            f"Best Acc: {max_acc:.2%}"
        )


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

- Plot()

```python
"""Implementation of plot function:
1. Plot the training loss vs epoch and validation loss vs epoch
2. Set the x-axis label to 'Epoch' and y-axis label to 'Loss'
3. Use blue line for training loss abd red line for validation loss
4. Save the plot to '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'

    plt.figure(figsize=(10, 6), dpi=300)
    # Plot the training loss and validation loss
    plt.plot(train_losses, label="Training loss", color="blue",
linewidth=2)
    plt.plot(val_losses, label="Validation loss",
color="red",linewidth=2)

    # Add title and labels
    plt.xlabel("Epoch", fontsize=12)
    plt.ylabel("Loss", fontsize=12)
    plt.title("Training and Validation Loss", fontsize=14)
    # Add grid and legend
    plt.grid(True)
    plt.legend(fontsize=12)

    plt.tight_layout()

    # Save the plot to 'loss.png'
    plt.savefig("loss.png")
    plt.close()  # Close the plot to free memory
    print("Save the plot to 'loss.png'")
    return
```

- Loss.png:



Part 2. Decision Tree

- Get_feature_and_labels():

```python
"""Implementation of get_features_and_labels():Use CNN to extract
the features from dataloaders
1. Set the model to evaluation mode, and disable the gradient
calculation
2. Loop over the dataloader, and get the featires and labels for
images
3. Turn the torch tensor to numpy array, and return the features
and 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
    features, labels = [], []

    model.eval()
    with torch.no_grad():
        for images, label in dataloader:
            # Take the data to GPU. label is just number don;t need
to transfer it to GPU
            images = images.to(device)
            # Foward pass the model, and get the output of the
model
            outputs = model(images)# Output type is tensor, [B, 5],
B means batch size, 5 means the number of classes

            # Append the result roeach list
            # Use .cpu() to move back to cpu, and numpy to turn the
tensor to numpy array
            features.append(outputs.cpu().numpy())
            labels.append(label.numpy())
    # Know the features is like [(B, 5), (B, 5), (B, 5)], so we
need to use np.concatenate to combine them together
    # Batch means every training, we have to sent B images to
model, and epoch means we run through all the images

    # We have to stack all the features together, the system will
automatically recognize each images's feature

    # Use if else is to prevent the error when the features is
empty
    # np.vstack is to deal with 2D array's stacking, and
np.concatenate is to deal with 1D array's stacking
    features = np.vstack(features) if features else np.array([])
    labels = np.concatenate(labels) if labels else np.array([])
    return features, labels
```

- Get_features_and_path():

```python
"""Implementation of get_features_and_paths(): Use CNN to extracct features from dataloaders
1. Set the model to evaluation mode, and disable the gradient calculation
2. Loop over the dataloader, and get the features and path of images
3. Turn the torch tensor value into numpy array, and return the features and path
"""
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
    features, paths = [], []

    model.eval()
    # Make sure model is on device
    model.to(device)

    with torch.no_grad():
        # path is the string from TestDataset's base name
        for images, path in dataloader:
            # Take the images to GPU
            images = images.to(device)
            # Use forward pass the model, an嵌套結構d get the result of the model
            outputs = model(images)
            # Append the result to each list
            features.append(outputs.cpu().numpy())

            # Reason why we use extend instead of append is to prevent nested structure
            """For example:
            batch_paths=['images1.jpg', 'images2.jpg', 'images3.jpg']
            path=[]
            path.append(batch_paths) -> path=[['images1.jpg', 'images2.jpg', 'images3.jpg']]
            path.extend(batch_paths) -> path=['images1.jpg', 'images2.jpg', 'images3.jpg'
            """
            paths.extend(path)
    # features is like[(B, 5), (B,5), (B,5)], so we stack them together to satrisfy the system's requirement
    # Use np.vstack to stack the 2D array.
    features = np.vstack(features) if features else np.array([])
    return features, paths
```

- _build_tree():

```python
    def _build_tree(self, X: pd.DataFrame, y: np.ndarray, depth:
int):
        # (TODO) Grow the decision tree and return it
        """Implementation of the _build_tree function:
        1. Every node has to have feature_index, thershold, left,
right, and class
        2. In the recursive, stop condition is:
            a. Reach max_depth
            b. All the data in the node are the same class
            c. No more feature to split
        3. Loop throuh all the features and calculate the best
split use function _best_split
        4. Split the data into left and right node use function
_split_data(Split the data basd on the best feature and threshold)
        5. When feature's value<thershold, go to left node, else go
to right node
        6. Keep recursively calling _build_tree to grow left and
right tree until meet the stop condition
        7. Return the tree node that include feature_index,
thershold, left, right and class(only for leaf node to get
predicted class
        """
        # Check if the depth is greater than max_depth, if so
return None
        if(depth>=self.max_depth or len(np.unique(y))==1):#
len(np.unique(y))==1 means all the data in node are in the same
class
            return {'class': np.argmax(np.bincount(y))} # Return
the class of the node, np.bincount(y) is to count the number of
each class in y, and np.argmax is to get the index of the maximum
value

        # Find the best feature index and threshold to split the
data
        feature, threshold = self._best_split(X,y)
        if feature is None: # Can't find the best feature to split
the data, so return None
            return {'class': np.argmax(np.bincount(y))} # Return
the most common class in the node
        # Split the data into left and right node
        X_left, y_left, X_right, y_right = self._split_data(X, y,
feature, threshold)        # Update the progress bar
        self.progress.update(1)

        # Keep recursively calling _build_tree to build left and
right tree
        return {
            'feature_index': feature,
            'threshold': threshold,
            'left': self._build_tree(X_left, y_left, depth+1),
            'right': self._build_tree(X_right, y_right, depth+1),
        }
```

- _predict:

```
    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
        """ Implementation of the predict function:
        X is the DataFrame of the testing dataset, (n_samples,
n_features) we use X.uterrows to traverse the dataset
        1. Use X.iterrows to traverse the dataset to get the
features
        2. Use _predict_tree to traverse the all the features and
put the result into array
        """
        return np.array([self._predict_tree(x, self.tree) for _, x
in X.iterrows()])
```

- _predict_tree():

```
    def _predict_tree(self, x, tree_node):
        # (TODO) Recursive function to traverse the decision tree

        """Implementation of the _predict_tree function:
        x is the features of the dataset, and tree_node is the
current node of the tree
        1. Check if the node is leaf node, if so return the class
of the node
        2. Check node's feature_index and threshold, if the
featire's value < threshold, go to left_tree, else go to right tree
        """
        if 'class' in tree_node: # 'class' is the jey of leaf-node,
so when we encounter means the node is leaf-node
             return tree_node['class']

        if x[tree_node['feature_index']] < tree_node['threshold']:
            return self._predict_tree(x, tree_node['left']) # Go to
left tree
        else:
            return self._predict_tree(x, tree_node['right']) # Go
to right tree
```

- _split_data():

```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
        """Implementation of the _split_data function:
        1. Use thershold as the split point to split tge
feature_index into left and right node
        2. If the feature's calue<thershold, go to left node, else
go to tight node
        """
        # pd.DataFrame is like dict, and we can use [] to get the
value of the key
        """Example of extract feature_index from X:
        X=np.array([[1,2,3],[4,5,6],[7,8,9]])
        X[:,1], :means choose all the rows, and 1 means choose the
second column
        X[:,1] = [2,5,8]
        """
        # Use X[:feature_index]] to get the feature index column
        feature_values = X.iloc[:, feature_index]
        # Make a bool to check if the feature's value is less than
the threshold
        left_mask = feature_values < threshold # Implementation of
boolean mask, left_mask is a list of bool value
        """Example of boolean mask
        X = [
        [1.2, 3.4, 5.6],  # 第0行 → 保留 (mask=True)
        [2.1, 0.5, 7.8],  # 第1行 → 保留 (mask=True)
        [5.6, 2.3, 1.2]   # 第2行 → 捨棄 (mask=False)
        ]
        feature_values=X[:,0] # X[:,0] is the first column, and
feature_values is [1.2, 2.1, 5.6]
        thershold=2.5
        left_mask=feature_values<threshold # left_mask is [True,
True, False]
        X[left_mask] only remain the line when left_mask is True,
so the result is [[1.2, 3.4, 5.6], [2.1, 0.5, 7.8]]
        """
        # Use the boolean mask to split the data into left and
right node
        left_dataset_X, right_dataset_X = X[left_mask],
X[~left_mask]
        left_dataset_y, right_dataset_y = y[left_mask],
y[~left_mask]
        return left_dataset_X, left_dataset_y, right_dataset_X,
right_dataset_y
```

- Best_split():

```python
    def _best_split(self, X: pd.DataFrame, y: np.ndarray):
        # (TODO) Use Information Gain to find the best split for a dataset
        """
        X is the feature of the dataset, y is the label of the dataset
        Implementation of the _best_split function:
        1. Loop through all the features and calculate the best split
        2. Best split means to maximize the information gain, gain=entropy(parent) - (weighted average of the entropy of the children)
        """
        best_gain = -1
        best_feature_index, best_treshold = None, None
        parent_entropy = self._entropy(y)

        n_features = X.shape[1] # Get the number of features in the dataset

        # Take all the only value of the feature, and take it as candidate threshold
        for feature_index in range(n_features):
            # Get the unique value of the feature
            thresholds = np.unique(X[:, feature_index])
            for threshold in thresholds:
                # Split the data to left node and right node
                left_x, left_y, right_x, right_y = self._split_data(X, y, feature_index, threshold)
                # Check if the left and right node is empty, if empty, continue the next feature
                # Check y instead of X, because we want to check the label of the dataset
                if len(left_y)==0 or len(right_y)==0:
                    continue
                # Calculate the left and right node's entropy
                left_entropy = self._entropy(left_y)
                right_entropy = self._entropy(right_y)

                # Calculate the weighted average of the entropy of the children
                # len(left_y) and len(right_y) represent the number of samples in left and right node
                total_len = len(y)
                weighted_entropy = (len(left_y)/total_len)*left_entropy + (len(right_y)/total_len)*right_entropy
                # Calculate the information gain
                gain = parent_entropy-weighted_entropy

                # Check if the gain is better than the best gain, if so update the best gain and best feature index and best threshold
                if gain > best_gain:
                    best_gain, best_feature_index, best_threshold = gain, feature_index, threshold

        return best_feature_index, best_threshold
```

- _entropy():

```python
    def _entropy(self, y: np.ndarray)->float:
        # (TODO) Return the entropy
        """Implementation of entropy function:
        1. np.ndarray is an array, and to calculate the entropy we
need to use the formula: Entropy(S) = -Σ (p_i * log₂(p_i))
        2. Based on the formula, count every class's number in the
dataset, and calculate the probability of each class
        3. Use the probability to calculate the entropy of the
dataset
        4. Return the entropy dataset
        """
        # Use vals, counts=np.unique(y, return_count=true) to count
the number of each class in data set
        """Example of using np.unique(y, return_counts=True):
        y=[1,2,2,3,1,4]
        vals, counts = np.unique(y, return_counts=True)
        vals=[1,2,3,4] #vals is all the non-repeated values in y
        counts=[2,2,1,1]# count is the number of each y
        """
        if len(y) == 0:
            return 0.0
        _, counts = np.unique(y, return_counts=True)
        # Calculate the probability of each class, and use np.log2
to calculate the logarithm of the probability
        probs = counts/len(y)
        entropy = -np.sum(probs*np.log2(probs+1e-10)) # Add small
value to prevent log(0)

        return entropy
```

- Experiment:

- **max_depth = 5**: The model is too shallow and may **underfit** the data, leading to **lower validation accuracy**.
- **max_depth = 9**: The model is deeper and may **overfit** the training data, so the **validation accuracy may increase or decrease** depending on how well it generalizes.