HW3

112550109 楊榮竣

1 CNN

Data Loading

In this loading data part, the main idea is using ***os.listdir()*** to read the lsit under the floder with corresponding path, and do some condition detect for different function.

In the part of ***load_train_dataset()***, firstly, I try to read the floders from the given path, and using their name to condsider which label they belong, and create a dictionary(**label_convert**) to translate the name of floder into label index. After that, I use ***os.listdir()*** and ***os.path.join()*** to get the path of all files and store them into list with their label.

In the ***laod_test_dataset()***, the part of reading files is same as ***load_train_dataset()***, I store all the path of images in a list, and then I use for loop and ***endswith()*** to check every file wether is image. However, I need to sort the index of files to keep answer will correct sequence. Because of that, I use ***.sort()*** to sort this files with their name.

```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_convert = {"elephant":0,"jaguar":1,"lion":2,"parrot":3,"penguin":4}

    for class_name in os.listdir(path):
        class_path = os.path.join(path, class_name)
        for img_name in os.listdir(class_path):
            img_path = os.path.join(class_path, img_name)
            images.append(img_path)
            labels.append(label_convert[class_name])

    return images, labels
```

load_train_dataset

```python
def load_test_dataset(path: str='data/test/')->List:
    # (TODO) Load testing dataset from the given path, return images
    images = []
    files = [f for f in os.listdir(path) if f.lower().endswith('.jpg')]
    files.sort(key=lambda x: int(os.path.splitext(x)[0]))
    for img_name in files:
        img_path = os.path.join(path, img_name)
        images.append(img_path)

    return images
```

load_test_dataset

Design Model Architecture

__init__(self,num_class=5)

In this CNN model, I design three convolutional layer and three full connected layer and a output layer for classfication.

Convolutional Layer

conv1: 3 inputs, 64 outputs, kernel_size = 5, dilation = 1

conv2: 64 inputs, 128 outputs, kernel_size = 5, dilation = 2

conv3: 128 inputs, 256 outputs, kernel_size = 3, dilation = 1

Full Connected Layer

fc1: 256 * 24 * 24 inputs, 128 outputs

fc2: 128 inputs, 64 outputs

fc3: 64 inputs, 32 outputs

out_layer: 32 inputs, 5 outputs

Others Layer

relu: using nn.ReLU layer.

pool: using nn.MaxPool2d layer with kernel_size = 2.

flatten: using nn.Flatten layer

```python
def __init__(self, num_classes=5):
    # (TODO) Design your CNN, it can only be less than 3 convolution layers
    super(CNN, self).__init__()
    n1 = 64
    n2 = 128
    n3 = 256
    fc1_input_size = n3 * (24**2)
    fc1_output = 128
    fc2_output = 64
    fc3_output = 32

    self.relu = nn.ReLU()
    self.pool = nn.MaxPool2d(kernel_size=2)
    self.flatten = nn.Flatten()

    self.conv1 = nn.Conv2d(3, n1, kernel_size=5,dilation=1,stride=1)
    self.conv2 = nn.Conv2d(n1,n2, kernel_size=5,dilation=2,stride=1)
    self.conv3 = nn.Conv2d(n2,n3, kernel_size=3,dilation=1,stride=1)

    self.fc1 = nn.Linear(fc1_input_size,fc1_output)
    self.fc2 = nn.Linear(fc1_output,fc2_output)
    self.fc3 = nn.Linear(fc2_output,fc3_output)
    self.out_layer = nn.Linear(fc3_output,num_classes)
```

__init__(self,num_class=5)

forward(self,x)

Convolution Layer:

First convolutional layer: conv1 processes the image. Followed by a ReLU activation (relu) to introduce non-linearity. Then a pooling layer (pool), usually max pooling, which downsamples the feature maps.

Full connected Layer:

Firstly, ***self.flatten(x)*** flattens the 2D features map into 1D vector. This flattened vector is then passed into the first fully connected layer, **fc1(x)**. After this, use **self.relu(x)** to help model learn complex pattern. This process is repeated across three fully connected layers. In the end, **out_layer** map he learned features to number of classes.

```python
def forward(self, x):
    # (TODO) Forward the model
    # original forward
    x = self.pool(self.relu(self.conv1(x)))
    x = self.pool(self.relu(self.conv2(x)))
    x = self.pool(self.relu(self.conv3(x)))
    x = self.flatten(x)
    x = self.relu(self.fc1(x))
    x = self.relu(self.fc2(x))
    x = self.relu(self.fc3(x))
    x = self.out_layer(x)
    return x
```

forward(self,x)

Define function

train(model, train_loader, criterion, optimizer, device)

Using ***model.train()*** to make dropout layer and batch normalization layer activite. And, use ***tqdm()*** to store train_loader in loop and show current state in running code. In the loop do folowing step to train the model.

1. **optimizer.zero_grad() : Clear last gradient.**
2. **model(image) : put image into model and get the prediction.**
3. **criterion(output, label) : Comparing labels and prediction results, using the loss function to calculate loss.**
4. **loss.backward() : Compute the gradient of loss**
5. **optimizer.step() : update the model weight using optimizer and the gradient.**

When training model, use sample_number to track the total number of element when excuting this train function and total_loss to track the loss of history. After each round of train, batch_size will get the size of current batch and loss.item will get the loss of current train. Finally, calculate the average loss and retrun it.

```python
def train(model: nn.Module, train_loader: DataLoader, criterion, optimizer, device) -> float:
    model.train()
    total_loss = 0.0
    sample_number = 0
    loop = tqdm(train_loader,desc="Traning",colour='#00CACA')
    for (image, label) in loop:
        image,label=image.to(device),label.to(device)
        batch_size = image.size(0)
        optimizer.zero_grad()
        output = model(image)
        loss = criterion(output, label)
        loss.backward()
        optimizer.step()
        total_loss += loss.item() * batch_size
        sample_number += batch_size
    avg_loss = total_loss / sample_number
    return avg_loss
```

train

validate(model, val_loader, criterion, device)

In the ***validate()***, the main step is similar to the train function, but with some differences. First, using ***model.val()*** instead of ***model.train()*** to disable the dropout layer and batch normalization layer, because we want to make sure that the data of model will not update in this validation step. Next, using ***with torch.no_no_grad()*** make the device skip gradient calculations to save memory and speed up evaluation. Then using ***torch.max(outputs, 1)*** to find the hightest score label for each images. Finally, compare the prediction and labels, adding the value of correction into total_correct. At the end of function, we return the avg_loss and accuracy.

```python
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 = 0.0
    total_correct = 0
    sample_number = 0
    with torch.no_grad():
        loop = tqdm(val_loader,desc="Validating")
        for (images,labels) in loop:
            images,labels = images.to(device),labels.to(device)
            outputs = model(images)
            batch_size = images.size(0)
            loss = criterion(outputs, labels)
            total_loss += loss.item() * batch_size
            _, preds = torch.max(outputs, 1)
            total_correct += (preds == labels).sum().item()
            sample_number += batch_size

    avg_loss = total_loss / sample_number
    accuracy = float(total_correct) / sample_number
    return avg_loss, accuracy
```

validate

test(model,test_loader,criterion,device)

Like the Validation part, we don't want to update the model internal parameters during this step, so we use ***model.eval()*** to disable dropout and batch normalization layer. Create a loop for test_loader and using tqdm to track the process. Then, any image in test_loader I put it into the model(***model(images)***), get the prediction(***torch.max(output,1)***). After calculating prediction, using ***preds.cpu().numpy()*** to get the prediction from cpu and change it into numpy type. Then, putting index and prediction respectively into ids and predictions. Finishing calculation, each prediction and their index is store in predictions and indexs. Then, use ***f.write()*** to write results into CNN.csv file.

```
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()
    predictions = []
    ids = []
    index = 1
    with torch.no_grad():
        loop = tqdm(test_loader, desc="Testing")
        for images, _ in loop:
            images = images.to(device)
            outputs = model(images)
            _, preds = torch.max(outputs, 1)
            preds = preds.cpu().numpy()
            for p in preds:
                predictions.append(p)
                ids.append(index)
                index += 1
    # CSV
    with open('CNN.csv', mode='w', newline='') as f:
        f.write('id,prediction\n')
        for id,label in zip(ids,predictions):
            f.write(f"{id},{label}\n")

    print(f"Predictions saved to 'CNN.csv'")
    return
```

test

Printing Training Logs

In each round of training, I use ***logger.info*** to print the information of training process and use ***torch.save()*** to save data of model in **'model.pth'** file. After begining traing model, I use **load_check_point** to change the mode of loading saving model.

```python
# (TODO) Print the training log to help you monitor the training process
#        You can save the model for future usage
max_acc = max(max_acc,val_acc)

torch.save({
    'epoch': epoch,
    'model_state_dict': model.state_dict(),
    'optimizer_state_dict': optimizer.state_dict(),
    'train_losses': train_losses,
    'val_losses':val_losses,
    'max_acc':max_acc
}, save_path)

logger.info(f"Round {epoch+1}: train_loss : {train_loss:.4f} , val_loss: {val_loss:.4f} , Accuracy : {val_acc:.4f}")
```

printing training logs and saving model

```python
load_check_point = False
start_epoch = 0
save_path = 'model.pth'
if load_check_point:
    checkpoint = torch.load(save_path)
    model.load_state_dict(checkpoint['model_state_dict'])
    optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
    start_epoch = checkpoint['epoch'] + 1
    train_losses = checkpoint['train_losses']
    val_losses = checkpoint['val_losses']
    max_acc = checkpoint['max_acc']
```

loading saving model

Plot Traning and Validation Loss

plt.figure() : create figure to draw.

plt.plot(): draw the train_losses and val_losses on the figure.

plt.xlabel(), ***plt.ylabel()***: to modify the label of x-axis and y-axis. I use

plt.title() : to add title on figure.

plt.legend() : Displays a legend on the figure, which helps identify different lines.

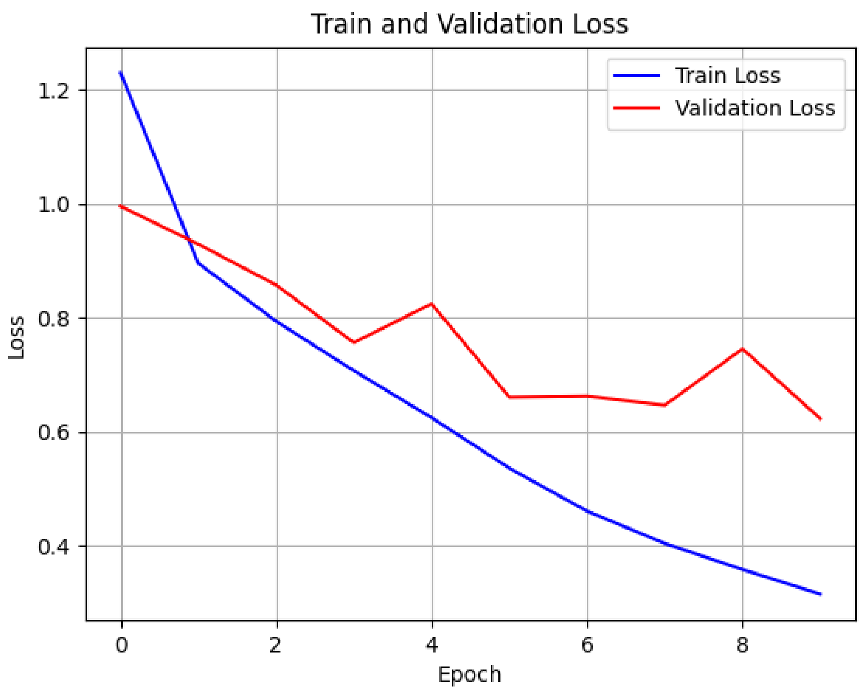
plt.grid(True) : Adds a grid to the plot background, making it easier to read values.

plt.savefig('loss.png') : save the figure with plotting as loss.png.

```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'
    fig = plt.figure()
    plt.plot(train_losses,'b-',label='Train Loss')
    plt.plot(val_losses,'r-',label='Validation Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.title('Train and Validation Loss')
    plt.legend()
    plt.grid(True)
    plt.savefig('loss.png')

    print("Save the plot to 'loss.png'")
    return
```

plot



loss.png

Experiments

In the loss.png, the validation loss is increasing when epoch is equal to 4 or 8, so overfitting happen in epoch 4 and epoch 8.

Method to Solve Overfitting

1. **Add Dropout layer and batchNorm layer.**

2. **Add weight_decay in optimizer.**

Implement

I add **BatchNorm** layer in convolutional layer and full connected layer. Additionally, I use dropout layer aftre relu layer finished. The **dropout_p** is a probability of node of full connected layer need to dropout. I set **dropout_p** as 0.5. Dorpout layer can randomly the output of some neurons to zero during training, which helps the model avoid overlearning or memorizing the training data too much.

```
self.dropout = nn.Dropout(p=dropout_p)

self.batch1 = nn.BatchNorm2d(n1,affine=True)
self.batch2 = nn.BatchNorm2d(n2,affine=True)
self.batch3 = nn.BatchNorm2d(n3,affine=True)

self.fc_batch1 = nn.BatchNorm1d(fc1_output,affine=True)
self.fc_batch2 = nn.BatchNorm1d(fc2_output,affine=True)
self.fc_batch3 = nn.BatchNorm1d(fc3_output,affine=True)
```

__init__

```
x = self.pool(self.relu(self.batch1(self.conv1(x))))
x = self.pool(self.relu(self.batch2(self.conv2(x))))
x = self.pool(self.relu(self.batch3(self.conv3(x))))
x = self.flatten(x)
x = self.dropout(self.relu(self.fc_batch1(self.fc1(x))))
x = self.dropout(self.relu(self.fc_batch2(self.fc2(x))))
x = self.dropout(self.relu(self.fc_batch3(self.fc3(x))))

x = self.out_layer(x)
return x
```

forward function

I change the original optimizer into new optimizer with weight_decay to add a small penalty to large weight values during training, which discourages the model from relying too heavily on any one feature or neuron.

```
# optimizer = optim.Adam(base_params, lr=1e-4) # original optimizer
optimizer = optim.Adam(base_params, lr=1e-4,weight_decay=1e-4) # new optimizer
```

Result of New Model

loss.png of new model

Model	**Original Model**	**New Model**
Accuracy	0.813	0.871

Accuracy of Model

In this loss picture, we can see that there is not any overfitting happen in 10 epoches. Moreover, the accuracy of new model is better than original model. Threrfore, the method is signigicantly helpful to make model avoid overfitting problem.

2 Decision Tree

Feature Extraction

The reason is same as mentioned aboving, we don't want to chnage model parameter when extracting feature. Therefore, using ***model.eval()*** to avoid that condition. Read all data in the dataloader and sent images into correct device(e.g. GPU) by ***image.to(device)***. Then, using ***feature = model(immage)*** we obtain the feature of imgae by passing through model. After calculation of features, we use ***torch.cat()*** to concatenate all features from each batches and ***.numpy()*** change data into numpy format. Finally, this function will return a features and labels with two ndarray type.

```
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 image , label in dataloader:
            image,label = image.to(device),label.to(device)
            feature = model(image)
            features.append(feature.cpu())
            labels.append(label.cpu())
    features = torch.cat(features).numpy()
    labels = torch.cat(labels).numpy()
    return features, labels
```

get_features_and_labels

The main purpose of this function is to get the test data feature and the index with correspond image. Therefore, I iterate through the data, pass image in dataloader throught correct device, and use ***model(images)*** to extract features of images. Because the data is sequential by the index, I can use **idx** as the index of images. Finally, I put the result into back of features and paths.

```python
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()
    features = []
    paths = []
    idx = 1
    with torch.no_grad():
        for images,path in dataloader:
            images = images.to(device)
            feature = model(images)
            features.append(feature.cpu())
            for i in path:
                paths.append(idx)
                idx += 1
    features = torch.cat(features).numpy()
    return features, paths
```

get_features_and_paths

Model Architecture

Tree Node Structure

I use dictionary to design the tree node. Every dictionary have following elements.

- **feature : the index of feature to make a decision in current tree node.**
- **threshold : the threshold value used to split the data based on the selected feature.**
- **left : a dictionary which represent the left child node of current node.**
- **right : a dictionary which represent the right child node of current node.**
- **label : this key only exist in leaf node to represent the most common label in the split dataset.**

_build_tree()

Firstly, I obtain **feature**(ie. index of target feature) and **threshold** by ***self._best_split(X,y)***. Then, it checks wether feature is None and wether the condition of procession end is met. If aboving condition is satisfied, this node become a leaf node, so it will count the most common label in current subset of data and return node dictionary which only contain 'label'. Otherwise, it splits data to left node and right node and it store feature, threshold, left, and right in dictionary and return it.

```python
def _build_tree(self, X: np.ndarray, y: np.ndarray, depth: int):
    # (TODO) Grow the decision tree and return it
    feature , threshold = self._best_split(X,y)
    if feature is None or depth == self.max_depth or len(np.unique(y)) == 1 or self.progress.n == self.progress.total:
        labels , counts = np.unique(y,return_counts=True)
        max_count , common_label = 0 , 0
        for label , count in zip(labels,counts):
            if count > max_count:
                common_label = label
                max_count = count
        return {"label": common_label}

    self.progress.update(1)
    left_x , left_y , right_x , right_y = self._split_data(X,y,feature,threshold)
    left = self._build_tree(left_x,left_y,depth+1)
    right = self._build_tree(right_x,right_y,depth+1)
    return {"feature": feature, "threshold": threshold, "left": left, "right": right}
```

build_tree

predict()

This function will iterate over each row in X and put the row into ***_predict_tree()*** method which will predict the label by input features. Then, merge the results and change list of result into **ndarray** as final answer.

```python
def predict(self, X: np.ndarray)->np.ndarray:
    # (TODO) Call _predict_tree to traverse the decision tree to return the classes of the testing dataset
    return np.array([self._predict_tree(x, self.tree) for x in X])
```

predict

_predict_tree()

In this function, I compare the target feature with threshold store in current tree node. If the feature value is larger than threshold, function moves to the right node keep computing until function arrive the leaf node. Otherwise, it moves to left node. In the leaf node, **tree_node** contains a ”label” key that represent the most common label after doing some sequencial decisions.

```python
def _predict_tree(self, x, tree_node):
    # (TODO) Recursive function to traverse the decision tree
    if "label" in tree_node:
        return tree_node["label"]
    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"])
```

_predict_tree

_split_data()

In this function, I need to split data according to feature_index and threshold. First, I use ***X[:,feature_index]*** to get the column of target feature. Then, I comapare all target features values with **thresold** and store result into **left_mask**. ***left_mask = feature_value <= threshold***, this line of code will get the boolean array with corresponded element wether is less than or equal than threshold. **right_mask** use inverter to invert all element in left_mask to get the values which is larger than threshold. Finally, return the split data **X[left_mask]**, **y[left_mask]**, **X[right_mask]**, **y[right_mask]**. X[left_mask] will selects the rows from X where the corresponding value in left_mask is True.

```
def _split_data(self,X: np.ndarray, y: np.ndarray, feature_index: int, threshold: float):
    # (TODO) split one node into left and right node
    feature_value = X[:,feature_index]
    left_mask = feature_value <= threshold
    right_mask = ~left_mask
    return  X[left_mask],y[left_mask],X[right_mask],y[right_mask]
```

_split_data

_best_split()

best_gain : best information current founded.

best_feature : index of feature with best information gain.

best_threshold : a value of feature with best information gain.

feature_number : the number of features.

$$\text{Information Gain} = \text{Entropy(parent)} - (\text{weighted average} \times \text{Entropy(children)})$$

best_split_data function will go through all features and find the best information gain. In the start of each round, ***values = X[:,idx]*** can transform all values of same feature into a ndarray. Then, Using ***np.unique()*** to obtain all unique values in **values** and going through every threshold in **thresholds**, I use the left_mask to store the pos of value which is less than or equal to threshold. Then, right_mask is equal to inverter of left_mask to represent the feature which larger than threshold. And ***y[left_mask]*** and ***y[right_mask]*** to calculate the result of labels of splite data, determine the information gain of result, and compare best information gain and new information gain. Moreover, I use ***p == 0 or p == 1*** to detect wether the data of left node or right node is zero to avoid zero divisor in entropy function.

```python
def _best_split(self,X: np.ndarray, y: np.ndarray):
    # (TODO) Use Information Gain to find the best split for a dataset
    best_gain = -1
    best_feature = None
    best_threshold = 0
    current_entropy = self._entropy(y)
    feature_number = X.shape[1]
    for idx in tqdm(range(feature_number),desc='best split',leave=False):
        values = X[:,idx]
        thresholds = np.unique(values)
        for threshold in thresholds:
            left_mask = values <= threshold
            right_mask = ~left_mask
            p = float(np.sum(left_mask))/len(y)
            if p==0 or p==1:
                continue
            gain = current_entropy - p * self._entropy(y[left_mask]) - (1-p)*self._entropy(y[right_mask])
            if gain > best_gain:
                best_gain = gain
                best_feature = idx
                best_threshold = threshold
    return best_feature, best_threshold
```

_best_split

_entropy()

In this part, I use ***np.unique*** to count the number of each term and divide them with total. Then, calculate val by this equation.

$$H(y) = -\sum_{i=1}^{n} p_i \log_2(p_i)$$

```python
def _entropy(self,y: np.ndarray)->float:
    # (TODO) Return the entropy
    val = 0.0
    total = len(y)
    _ , counts = np.unique(y,return_counts=True)
    for count in counts:
        p = count/total
        val -= p*np.log2(p)
    return val
```

entropy

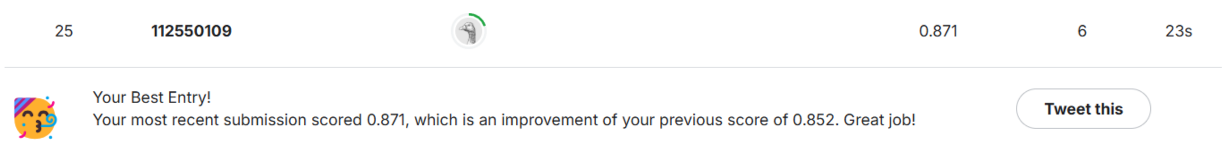
Experiment

max_depth	5	7	9
Experiment 1	0.7440	0.7974	0.7805
Experiment 2	0.7564	0.7783	0.7820
Experiment 3	0.7410	0.7644	0.8061
Average	0.7471	0.7800	0.7895

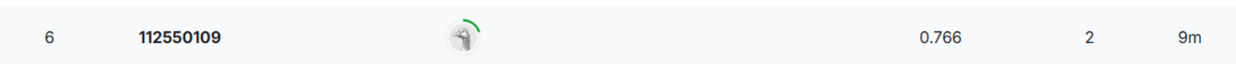
Experiment Results : Validation Accuracy

According aboving table, increasing max depth from 5 to 7 can significantly improve validation accuracy. Therefore, the model with max depth 7 can catch more features of images than model with depth 5. However, increasing max depth from 7 to 9 only have small improvement in accuracy. I think the reason of this result may be that the model with depth 9 encounters overfitting problems in training. Therefore, the rate of increasing in higher max depth is not larger.

3 Kaggle Scoring



CNN score



Decision Tree score