

- Data Loading

- ◆ load_train_dataset():

images, labels are the empty lists stored the two returned data. sw is the dict make change label easier. The first loop for each folder in data/train/, then second loop for each picture in that folder. Storing the picture path and label that come from folder name.

```
def load_train_dataset(path: str='data/train/')->Tuple[List, List]:
    # Load training dataset from the given path, return images and labels
    images = []
    labels = []
    sw = {"elephant":0, "jaguar":1, "lion":2, "parrot":3, "penguin":4}
    for sub in os.listdir(path):
        subpath = os.path.join(path, sub)
        for pic in os.listdir(subpath):
            picpath = os.path.join(subpath, pic)
            images.append(picpath)
            labels.append(sw[sub])

    #raise NotImplementedError
    return images, labels
```

- ◆ load_test_dataset():

Same as load_train_dataset, but without labels.

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

    return images
```

- Design Model Architecture

- ◆ __init__():

With 3 layers of convolution that kernel size is 5*5. Because the original pictures have 3 channels, I use 3 channels for the first layer input. I try the different numbers to output channel, then I find that using 32 to the first layer can get the best result. And the other channels are linear to n1. Then use the 2*2 kernel with stride 2 as the pool. The convolutional layers and pooling layer are the same as the sample in spec. Fully-Connected layer need to compute the input number and the output number. Equations are written in the picture. I searched the google that people say unit can start with 128, so I use 128 to start the training. I also use the dropout, for each training the model can only use half of features to

compute the score. Avoid over-reliance on certain features.

```
n1 = 32
n2, n3 = n1*2, n1*4
class CNN(nn.Module):
    代码解释 | 函数注释 | 调优建议 | 行间注释
    def __init__(self, num_classes=5):
        # Design your CNN, it can be no more than 3 convolution layers
        super(CNN, self).__init__()
        self.Conv_1 = nn.Conv2d(in_channels=3, out_channels=n1, kernel_size=5) #224-4 220/2
        self.Conv_2 = nn.Conv2d(in_channels=n1, out_channels=n2, kernel_size=5) #110-4 106/2
        self.Conv_3 = nn.Conv2d(in_channels=n2, out_channels=n3, kernel_size=5) #53-4 49/2 = 24
        self.Maxpool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.fc_1 = nn.Linear(24*24*n3, 128) #units=128
        self.fc_2 = nn.Linear(128, num_classes) #output [0 1 2 3 4] len=5
        self.dropout = nn.Dropout(p=0.5)
```

◆ Forward():

For the convolutional layers, I do the same things on them. 1. convoluting 2. ReLU 3. pooling. Before the first fully-connected layer, the data must be flattened. Then do the fully-connected, do the ReLU again. The steps can see in sample at spec. Then I dropout some features, go to the final layer.

```
def forward(self, x):
    # Forward the model
    x = self.Conv_1(x)
    x = torch.relu(x)
    x = self.Maxpool(x) #layer 1

    x = self.Conv_2(x)
    x = torch.relu(x)
    x = self.Maxpool(x) #layer 2

    x = self.Conv_3(x)
    x = torch.relu(x)
    x = self.Maxpool(x) #layer 3

    x = x.view(x.size(0), -1)
    x = self.fc_1(x)
    x = torch.relu(x) #fc1

    x = self.dropout(x)
    x = self.fc_2(x) #fc2
    return x
```

- Define function train(), validate() and test()

- ◆ train():

First, turn the train mode on. `all_loss`: computes the total loss over all batches. `batch`: total number of used batches. Starts a tqdm progress bar, total as step number, desc as title, unit for calculating the unit of 1 sec. Because I use the batch as total step, the update place at the last of loop of batch. Loop through the data loader for each images, labels in a batch. Move data to GPU or CPU. Clear previous calculated gradients. Train the model by images of this batch. Get the loss for outputs and labels, and computes gradients of loss by backward. Then the optimizer updates the model's weights based on that gradients. Add the loss to `all_loss`. The loss is tensor, so use the `.item()` to get the exactly number. I use `set_postfix()` to see if the loss reduce. After all, `all_loss/batch` to get the average loss.

```
def train(model: CNN, train_loader: DataLoader, criterion, optimizer, device) -> float:
    # Train the model and return the average loss of the data, we suggest use tqdm to know the progress
    model.train()

    all_loss = 0.0
    batch = len(train_loader)

    with tqdm(total=batch, desc="Training Progress", unit="batch") as pbar:
        for batch_idx, (images, labels) in enumerate(train_loader):
            images, labels = images.to(device), labels.to(device)

            optimizer.zero_grad()
            outputs = model(images)

            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            all_loss += loss.item()

            pbar.set_postfix(loss=loss.item())
            pbar.update(1)

    avg_loss = all_loss / batch
    return avg_loss
```

Also, I use the learning rate 0.001 not 1e-4.

```
optimizer = optim.Adam(base_params, lr=0.001)
```

- ◆ validate():

Just like the train(), but use eval mode. `num` is count the how many data it has, for compute the accuracy rate. And count the number of same prediction and label. Get the prediction by `torch.max(outputs, 1)` _ will be the maximum of each tensor, predicted will be the maximum index of each tensor. Then it can return the average loss and the accuracy rate.

```
def validate(model: CNN, val_loader: DataLoader, criterion, device) -> Tuple[float, float]:
    # Validate the model and return the average loss and accuracy of the data, we suggest use tqdm to know the progress
    model.eval()

    all_loss = 0.0
    correct = 0
    batch = len(val_loader)
    num = 0

    with tqdm(total=batch, desc="Validating Progress", unit="batch") as pbar:
        with torch.no_grad():
            for batch_idx, (images, labels) in enumerate(val_loader):
                images, labels = images.to(device), labels.to(device)
                num += len(images)
                outputs = model(images)

                loss = criterion(outputs, labels)
                all_loss += loss.item()

                _, predicted = torch.max(outputs, 1)
                correct += (predicted == labels).sum().item()

                pbar.set_postfix(loss=loss.item())
                pbar.update(1)

    avg_loss = all_loss / batch
    accuracy = 100 * correct / num

    return avg_loss, accuracy
```

◆ test():

Same as validate(), but not need to compute the loss and the accuracy rate. But it need the name of picture. The test data haven't shuffled, so I using the index of image back to the dataset to get its name.

```
def test(model: CNN, test_loader: DataLoader, criterion, device):
    # Test the model on testing dataset and write the result to 'CNN.csv'
    model.eval()
    data = []
    dataset = test_loader.dataset
    batch = len(test_loader)

    with tqdm(total=batch, desc="Testing Progress", unit="batch") as pbar:
        with torch.no_grad():
            for batch_idx, (images, l) in enumerate(test_loader):
                images = images.to(device)

                output = model(images)
                _, predicted = torch.max(output, 1)
                for i in range(len(images)):
                    idx = batch_idx*32 + i
                    path = dataset.image[idx]
                    path = path.replace('data/test/', '', 1)
                    path = path.replace('.jpg', '', 1)
                    data.append((path, predicted[i].item()))

                pbar.update(1)

    df = pd.DataFrame(data, columns=['id', 'prediction'])
    df.to_csv('CNN.csv', index=False)

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

● Printing Training Logs

maxe to see if the max_acc come at last. And save the model at the max_acc to test the test data. Decrease the time of training model.

```
train_losses = []
val_losses = []
max_acc = 0
maxe = 0
EPOCHS = 10
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)
    if val_acc > max_acc:
        max_acc = val_acc
        maxe = epoch+1
        torch.save(model, 'cnn_model.pt')

    train_losses.append(train_loss)
    val_losses.append(val_loss)
    # print(f"{train_loss:.4f}, {val_loss:.4f}, {val_acc:.4f}")
    # Print the training log to help you monitor the training process
    # You can save the model for future usage

    logger.info(f"Best Accuracy: {max_acc:.4f}#", at {maxe}th")
```

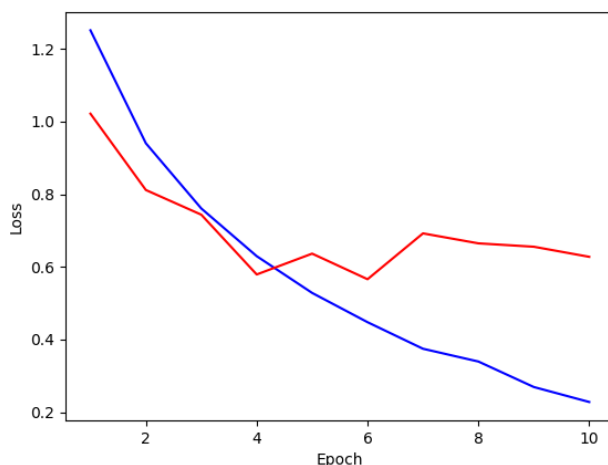
- Plot Training and Validation Loss

- ◆ plot():

I add a int (epochs) to test different EPOCH numbers easier.

```
def plot(train_losses: List, val_losses: List, epochs: int):  
    # Plot the training loss and validation loss of CNN, and save the plot to 'loss.png'  
    # xlabel: 'Epoch', ylabel: 'Loss'  
    x = list(range(1, epochs + 1))  
    plt.plot(x, train_losses, 'b')  
    plt.plot(x, val_losses, 'r')  
    plt.xlabel('Epoch')  
    plt.ylabel('Loss')  
    #raise NotImplementedError  
    plt.savefig('loss.png')  
    print("Save the plot to 'loss.png'")  
    return
```

The red one is val_loss, blue one is train_loss.



- Experiments

The overfitting happens at 4th and 6th epoch. When the train loss decreased, the val loss increased.

1. use the dropout after the first layer of fully-connected. Avoid over-reliance on certain features. So, it can reduce the chance of overfitting. Accuracy rate can be improved, but the loss it's not certainly decrease or increase.

```
self.dropout = nn.Dropout(p=0.5)
```

2. Save the optimal model, then use it at test. It's early stopping. It avoids the change of overfitting. Accuracy rate must be the optimal of this loop. loss won't change by that.

```
if val_acc > max_acc:  
    max_acc = val_acc  
    maxe = epoch+1  
    torch.save(model, 'cnn_model.pt')
```

```
model = torch.load('cnn_model.pt', weights_only=False)  
  
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)  
test(model, test_loader, criterion, device)
```

- Feature Extraction

- ◆ get_features_and_labels():

To get the features use the eval mode. Then using hook to store the data stream for the certain layer. I print the name of each layer of model. Then I chose the `global_pool.flatten` layer. I have tried the other layer, but none of them can get a better score than this layer. This layer has about 576 features. So, I use hook to get the features data of outputs, before each prediction. Then append them on the list. The most difficult thing is to figure out the data structure. The shape of `f[i]` is `[576,1,1]`, I use the `squeeze` to remove the dimension that length equals 1, and use `numpy` change the tensor to the `numpy` vector. Then change the features to `DataFrame`, labels to the `numpy` array. Make the other function process easier.

```
def get_features_and_labels(model: ConvNet, dataloader: DataLoader, device) -> Tuple[pd.DataFrame, np.ndarray]:
    # Use the model to extract features from the dataloader, return the features and labels
    model.eval()

    batch = len(dataloader)
    f, labels, features = [], [], []
    layer = dict(model.named_modules())['model.global_pool.flatten']

    代码解释 | 函数注释 | 调优建议 | 行间注释
    def hookf(module, input, output):
        f.append(output.detach())

    with torch.no_grad(), tqdm(total=batch, desc="Features label relation getting Progress", unit="batch") as pbar:
        h = layer.register_forward_hook(hookf)
        for batch_idx, (images, l) in enumerate(dataloader):
            images = images.to(device)

            outputs = model(images)
            .....

            f1 = f[len(f)-1]

            for i in range(len(images)):
                features.append(f1[i].squeeze().numpy())
                labels.append(l[i].item())

            pbar.update(1)

        f = []
        h.remove()
    features = pd.DataFrame(features)
    labels = np.array(labels)
    return features, labels
```

- ◆ get_features_and_paths():

Same as `get_features_and_labels`, but use `dataset` to get the path of pictures. I using the index of image back to the `dataset` to get its path.

```
def get_features_and_paths(model: ConvNet, dataloader: DataLoader, device) -> Tuple[pd.DataFrame, np.ndarray]:
    # Use the model to extract features from the dataloader, return the features and path of the images
    model.eval()

    dataset = dataloader.dataset
    batch = len(dataloader)
    f, paths, features = [], [], []

    layer = dict(model.named_modules())['model.global_pool.flatten']

    代码解释 | 函数注释 | 调优建议 | 行间注释
    def hookf(module, input, output):
        f.append(output.detach())

    with torch.no_grad(), tqdm(total=batch, desc="Features label relation getting Progress", unit="batch") as pbar:
        h = layer.register_forward_hook(hookf)
        for batch_idx, (images, l) in enumerate(dataloader):
            images = images.to(device)

            outputs = model(images)
            .....

            f1 = f[len(f)-1]

            for i in range(len(images)):
                idx = batch_idx*32 + i
                path = dataset.image[idx]
                features.append(f1[i].squeeze().numpy())
                paths.append(path)

            pbar.update(1)

        f = []
        h.remove()

    features = pd.DataFrame(features)
    paths = np.array(paths)
    return features, paths
```

- Model Architecture

- ◆ `_build_tree()`:

The prediction sets as the most frequency label of y. If the node reaches the maximum depth, return the node of type leaf and the prediction. Also, if the entropy of this node is very small will return, that means this node is almost the same label in y. And the length of y is very short will return, that means the split is not effectiveness. After these, find the best split. If the the `feature_index` of best split function equals -1, meaning the best split of this node is too small, so just return the leaf. Then do the split, return the node

```
def _build_tree(self, X: pd.DataFrame, y: np.ndarray, depth: int):
    # Grow the decision tree and return it
    v, c = np.unique(y, return_counts=True)

    predi = v[np.argmax(c)]

    #print(f'{predi}pre, {self._entropy(y):.4f}e, {depth}dep')
    if depth == self.max_depth or self._entropy(y) < 1e-3 or len(y)<5:
        self.progress.update(1)
        return {
            'type': 'leaf',
            'prediction': predi
        }

    bf, bt = self._best_split(X, y)

    if bf == -1:
        #print(f'bf d{depth} p{predi}')
        self.progress.update(1)
        return {
            'type': 'leaf',
            'prediction': predi
        }
    fn = X.columns[bf]
    lm = X[fn] <= bt
    rm = X[fn] > bt

    lt = self._build_tree(X[lm], y[lm], depth+1)
    rt = self._build_tree(X[rm], y[rm], depth+1)

    self.progress.update(1)
    return{
        'type': 'decision node',
        'feature_idx': bf,
        'threshold': bt,
        'left_tree': lt,
        'right_tree': rt
    }
```

with the info of type decision node, which feature needs to use at decision(`feature_idx`), the decide value (threshold), and the left tree node and the right tree node.

- ◆ `predict()` :

For each row of data X, to get the label from function `_predict_tree`, and turn the label list to numpy array.

```
def predict(self, X: pd.DataFrame)->np.ndarray:
    # Call _predict_tree to traverse the decision tree to return the classes of the testing dataset
    labels = []
    for x in X.itertuples(index=False):
        x = np.array(x)
        l = self._predict_tree(x, self.tree)
        labels.append(l)

    labels = np.array(labels)
    return labels
```

◆ `_predict_tree()`:

Search the decision tree just like binary search tree. So, only when reach the leaf will return, otherwise do the recursion.

```
def _predict_tree(self, x, tree_node):
    # Recursive function to traverse the decision tree
    l = -1
    if tree_node['type'] == 'leaf':
        return tree_node['prediction']
    else:
        nextnode = None
        if x[tree_node['feature_idx']] <= tree_node["threshold"]:
            nextnode = tree_node['left_tree']
        else:
            nextnode = tree_node['right_tree']

        l = self._predict_tree(x, nextnode)

    return l
```

◆ `_split_data()`:

Split the data X and y by the threshold and feature_index. If the value of X column data is greater than the value of threshold, then split them into the two data sets. Due to these two data structures can use the bool list to get the wanted data. So, can just get the separated data by `data[bool list]`.

```
def _split_data(self, X: pd.DataFrame, y: np.ndarray, feature_index: int, threshold: float):
    # split one node into left and right node
    feature_name = X.columns[feature_index]
    lm = X[feature_name] <= threshold
    rm = X[feature_name] > threshold

    left_dataset_X, left_dataset_y = X[lm], y[lm]
    right_dataset_X, right_dataset_y = X[rm], y[rm]

    return left_dataset_X, left_dataset_y, right_dataset_X, right_dataset_y
```

◆ `_best_split()`:

The part of most high time complexity in the decision tree. To find the best split from depth 0 to depth 1 need 4 hours. The length of X column is 576, and the longest length of X row can be 5764. For reduce the time of this part, I use unique to the value of each row of fixed column, make the same value won't be computed again. Although, the values aren't the continuous number, I still use the way used at continuous features. Because it can still reduce running time a little bit. The best split means the entropy gets smaller as much. So, compute the entropy from the present and the entropy from features separated data sets. The difference between the present one and the separated one. Also, to make sure that it won't split the unuse set, I set the original value is 1e-6; The data set could be separated, only if the difference is greater than this

value.

```
def _best_split(self, X: pd.DataFrame, y: np.ndarray):
    # Use Information Gain to find the best split for a dataset
    best_gain = 1e-6
    best_feature_index = -1
    best_threshold = -1.0
    e = self._entropy(y)

    #g = len(X.columns)
    #with tqdm(total=g, desc="bsplit", unit="column") as pbar:
    for f_idx, f in enumerate(X.columns):
        v = np.sort(X[f].unique())
        if len(v) >= 2:
            ts = (v[:-1]+v[1:])/2
            for t in ts:
                lx, ly, rx, ry = self._split_data(X, y, f, t)

                if len(ly) != 0 and len(ry) != 0:
                    pl = len(ly)/len(y)
                    pr = 1-pl
                    el = self._entropy(ly)
                    er = self._entropy(ry)

                    g = e - (pl*el+pr*er)

                    if g > best_gain:
                        best_gain = g
                        best_feature_index = f_idx
                        best_threshold = t

            #pbar.set_postfix(g=best_gain, i=best_feature_index)
            #pbar.update(1)

    return best_feature_index, best_threshold
```

◆ _entropy():

Same as the picture of the right side. c for count the number of each label in y. Then divide by len(y), it becomes the probability of each label. Now, use the entropy function, and maybe p is nearly 0, +1e-9 to avoid log2(0) .

```
def _entropy(self, y: np.ndarray)->float:
    # Return the entropy
    v, c = np.unique(y, return_counts=True)
    nv = len(v)
    n = len(y)

    p = c / n
    e = -np.sum(p * np.log2(p + 1e-9))

    return e
```

$$\text{Entropy} = -\sum p_j \log_2 p_j$$

● Experiment

For max_depth to 5, the validation accuracy will decrease. The tree becomes shallower, then its' capacity to capture patterns will decrease. It cannot separate classes well.

For max_depth to 9, the validation accuracy won't certainly increase or decrease. It could become overfitting. Or it has enough space to separate the classes by the other features.

● Kaggle Evaluation Baseline

◆ CNN:

129

109550072



0.829

1

3m

◆ Decision Tree:

The score of 0.829 is the cnn, I put in the wrong place. SO, the real score is 0.753. I don't know how to delete it.




DecisionTree.csv
Complete · 19h ago

0.753



This leaderboard is calculated with all of the test data.

#	Team	Members	Score	Entries	Last
32	109550072		0.829	8	19h