`CustomImageDataset` class is doing.

Summary of `CustomImageDataset`

1. **Initialization (`__init__`)**
   - Takes the dataset folder (`root_dir`) and optional transforms.
   - Finds all **class folders** (ignores hidden ones).
   - Creates a mapping:

     ```
     {"class1": 0, "class2": 1, ...}
     ```

   - Calls `_load_images()` to collect all image file paths and their class labels.
   - Stores them in `self.images`.

---

2. **Image Collection (`_load_images`)**
   - Scans each class folder inside `root_dir`.
   - For every valid image file (`.png`, `.jpg`, `.jpeg`, ...), saves:

     ```
     (image_path, class_index)
     ```

   - Example:

     ```
     [("root_dir/class1/img1.jpg", 0),
      ("root_dir/class1/img2.jpg", 0),
      ("root_dir/class2/img3.jpg", 1)]
     ```

---

3. **Length (`__len__`)**
   - Returns the total number of images found.
   - Used by `len(dataset)` and DataLoader.

---

4. **Get Item (`__getitem__`)**
   - Given an index `idx`:
     - Looks up `(img_path, label)` from `self.images`.
     - Opens the image and **converts it to grayscale (`'L'`) → 1 channel**.
     - Applies transforms if provided.

- Returns `(image, label)`.

✅ In short:

This class:

1. Scans your dataset directory once,
2. Maps each image to its class label,
3. Loads images as **grayscale**,
4. Applies transforms,
5. Returns `(image_tensor, label)` when accessed.

👉 The `Dataset` base class in PyTorch is **just an interface** — it only defines that your custom class must implement:

- `__len__(self)`
- `__getitem__(self, idx)`

That's it. It does **not** handle class detection or label mapping for you.

🔎 Where does mapping happen then?

- In your code, **you wrote it manually** here:

```
self.classes = [d for d in os.listdir(root_dir) if os.path.isdir(os.path.join(root_dir, d)
self.classes.sort()
self.class_to_idx = {cls_name: i for i, cls_name in enumerate(self.classes)}
```

So:

- `self.classes` = list of folder names (`["cats", "dogs"]`)
- `self.class_to_idx` = mapping (`{"cats": 0, "dogs": 1}`)

Later, when `_load_images` runs, you use this mapping to assign each image a label.

⚡ But in `torchvision.datasets.ImageFolder`

Yes — the mapping *does* happen automatically.

- `ImageFolder` assumes your dataset is structured like:

```
root/
  ├── class1/
  │     ├── img1.jpg
```

```
    ├── class2/
            ├── img2.jpg
```

- It automatically:
  - Detects the class folders.
  - Sorts them.
  - Creates `class_to_idx`.
  - Returns `(image_tensor, label)` when you access an item.

So your `CustomImageDataset` is basically **re-implementing what `ImageFolder` already does**, but with your own tweaks (like grayscale conversion).

✅ **Answer in one line:** No, the mapping doesn't happen automatically in a custom `Dataset` — you must implement it (as you did). But if you use `torchvision.datasets.ImageFolder`, then yes, mapping happens automatically.

Do you want me to show you how your class could be simplified just by using **`ImageFolder` + a small transform** (instead of writing `_load_images` etc.)?

ResNet50

1. Mount Google Drive

- Mounts Google Drive to access dataset stored in `/content/drive/MyDrive/Project2/Project 1 Data`.

---

2. Device Configuration

- Checks for GPU availability (`cuda`) and sets the device for training.

---

3. Data Augmentation and Normalization

- **Basic Transform:** Resize, center crop, convert to tensor, normalize using ImageNet stats.
- **Augmentation Transform:** Random horizontal flip, rotation, color jitter, affine transforms for creating diverse samples.

---

4. Load and Split Dataset

- Loads images using `ImageFolder`.
- Extracts labels for stratified splitting.

- Splits the dataset into **80% training** and **20% validation** using `StratifiedShuffleSplit`.

---

5. Dataset Balancing

- Computes class counts for the training subset.
- Determines maximum class count and calculates **augmentation needed per class**.
- Defines a **custom `BalancedDataset`** class:
  - Original samples use `basic_transform`.
  - Additional samples are augmented using `augment_transform` + `basic_transform`.
- Verifies that the training dataset is now balanced.

---

6. Prepare Validation Dataset

- Defines a **`SubsetWithTransform`** class to apply `basic_transform` to validation data.

---

7. Create DataLoaders

- Creates **train_loader** and **val_loader** with batch size 32.
- Shuffles training data, does not shuffle validation data.
- Uses multiple workers (`num_workers=4`) for faster data loading.

---

8. Model Setup (Transfer Learning)

- Loads a pre-trained **ResNet50** model using `timm`.
- Replaces the final fully connected layer to match **5 classes**.
- Moves model to GPU or CPU.

---

9. Loss Function and Optimizer

- Uses **cross-entropy loss** for multi-class classification.
- Uses **Adam optimizer** with learning rate `3e-5` and L2 regularization (`weight_decay=1e-5`).
- Adds a **StepLR scheduler** to decrease learning rate every 7 epochs.

---

10. Training and Validation Function

- Loops over `num_epochs`:
  - **Training Phase:** forward pass, compute loss, backpropagate, update weights.
  - **Validation Phase:** compute validation loss and accuracy without gradient.
- Tracks **best validation accuracy** and saves the model if it improves.

- Prints epoch-wise statistics: train loss, validation loss, validation accuracy, and time.

---

11. Model Saving

- Saves the **best performing ResNet model** as `best_res_net_model.pth`.

✅ **In short:** You are **loading, augmenting, and balancing a dataset**, training a **pre-trained ResNet50** with a custom classifier, tracking validation performance, and saving the best model.

ViT base Model

Here's a **pointwise summary** of what your code is doing:

1. **Device Configuration:**
    - Checks if GPU is available and sets the device to `cuda` or `cpu`.
2. **Data Augmentation and Normalization:**
    - Defines `basic_transform` for resizing, cropping, tensor conversion, and normalization.
    - Defines `augment_transform` for image augmentation (flip, rotation, color jitter, affine transform).
3. **Load and Split Dataset:**
    - Loads images from Google Drive using `ImageFolder`.
    - Splits dataset into 80% training and 20% validation using stratified shuffle split.
4. **Class Balancing:**
    - Calculates class counts in training subset.
    - Determines how many augmented samples are needed per class to balance dataset.
    - Defines `BalancedDataset` class to generate augmented samples for underrepresented classes.
    - Creates a balanced training dataset.
5. **Prepare Validation Dataset:**
    - Wraps validation subset in `SubsetWithTransform` class to apply `basic_transform`.
6. **Create DataLoaders:**
    - Defines `DataLoader` for training (`shuffle=True`) and validation (`shuffle=False`) with batch size 32 and 4 workers.
7. **Model Setup (Transfer Learning - ViT):**

   - Loads pre-trained Vision Transformer (`vit_base_patch16_224`).
   - Replaces the classification head with a new linear layer to match `num_classes = 5`.
   - Moves the model to the configured device.

8. **Loss Function, Optimizer, Scheduler:**
   - Uses `CrossEntropyLoss`.
   - Uses `Adam` optimizer with weight decay for L2 regularization.
   - Uses `StepLR` scheduler to decay learning rate every 7 epochs.

9. **Training and Validation Function (`train_model`):**
   - Loops over epochs:
     - **Training Phase:** Sets model to train mode, computes loss, backpropagates, and updates weights.
     - **Validation Phase:** Sets model to eval mode, computes loss and accuracy.
   - Tracks best validation accuracy and saves model whenever it improves.
   - Prints epoch-wise train loss, validation loss, validation accuracy, and epoch time.

10. **Model Saving:**
    - The best model (with highest validation accuracy) is saved as `'best_vit_model.pth'`.

Vit FineTune

1. Device Setup

- Using GPU if available (`cuda`), otherwise CPU.

2. Data Preparation

- **Basic transforms:** Resize to 256, center crop to 224, convert to tensor, normalize with ImageNet stats.
- **Augmentation:** Random horizontal flip, rotation, color jitter, affine transforms to generate additional samples.

3. Dataset Split

- Training and validation sets are created with a **balanced validation set** (~10% of data).
- `Subset` and custom dataset classes are used to handle transforms and maintain class balance.

4. Class Balancing

- Compute class counts in training set.
- Determine how many augmented samples are needed for each class to match the **maximum class count**.
- `BalancedDataset` applies augmentation to under-represented classes while keeping original samples.

5. DataLoaders

- Training and validation DataLoaders created with batch size 32 and multiple workers.

6. Model Setup (ViT)

- Pre-trained ViT model (`vit_base_patch16_224`) is loaded.
- **Classification head replaced** for 5 classes with added dropout:

```
model.head = nn.Sequential(
    nn.Dropout(p=0.5),
    nn.Linear(model.head.in_features, num_classes)
)
```

- **Fine-tuning type:**
  - **Full fine-tuning:** All layers, including pre-trained ones, are updated during training because all parameters are passed to the optimizer.
  - Classification head is initialized randomly, learning from scratch.

7. Loss, Optimizer, Scheduler

- Loss: `CrossEntropyLoss`.
- Optimizer: `Adam` with learning rate 3e-5 and L2 regularization.
- Learning rate scheduler: `StepLR` (reduces LR every 7 epochs by 0.1 factor).

8. Training and Validation

- **Training loop:**
  - Model set to train mode.
  - Forward pass → compute loss → backward → optimizer step.
- **Validation loop:**
  - Model set to eval mode.
  - Compute loss and accuracy without gradient computation.

- Best model saved based on **highest validation accuracy**.

9. Output

- Trains for 20 epochs.
- Saves the best model (`best_vit_model.pth`).
- Reports **best validation accuracy** at the end.

✅ In short:

- Full ViT model is fine-tuned on a small, balanced, and augmented dataset with a replaced classification head for 5 classes.
- Augmentation and balancing are used to improve generalization.
- All layers, including pre-trained ones, are updated, so this is **full fine-tuning** rather than just head training.

Start coding or generate with AI.