How the train_model.py works

This training model trains a deep learning model using PyTorch to classify an image into 2 different categories i.e. safe vs unsafe content using a pre-trained ResNet-50 convolutional neural network (CNN).This pre-trained model works on transfer learning , where the pre trained model's weights are reused and only a connected layer is trained for a specific task. To improve the training efficiency early stopping and learning rate scheduling are used.

1. **Imports and dependencies**:

Torch library used for tensor operations and neural networks.

Torch.optim contains optimization algorithms.

Torchvision.datasets provides features to load image datasets.

Tqdm displays a progress bar during training.

os is used for file path operations.

2.**Configurations**:

DATA_DIR: Path to the dataset, expected to contain train and validation subdirectories.

MODEL_SAVE_PATH: Where the trained model weights will be saved.

EPOCHS: Number of times the model iterates over the entire training dataset (set to 5).

BATCH_SIZE: Number of images processed in one forward/backward pass (32 images per batch).

LEARNING_RATE: Step size for the optimizer (0.001, typical for fine-tuning)

For the data verification, the file path checks if the **train** and **val** directories exist in **DATA_DIR**. If either of them misses it raises an error which ensures that the dataset is properly structured.

3. Below are the data transformations required for the dataset:

transforms.Resize((224, 224)): Resizes images to 224x224 pixels, the input size required by ResNet-50.

- **transforms.RandomHorizontalFlip()**: Randomly flips images horizontally (only for training), a form of data augmentation to improve generalization.
- **transforms.ToTensor()**: Converts PIL images to PyTorch tensors and scales pixel values to [0, 1].

- **transforms.Normalize(mean, std)**: Normalizes the tensor using the mean and standard deviation of the ImageNet dataset (since ResNet-50 was pre-trained on ImageNet). This ensures the input data distribution matches the pre-trained model's expectations.

Different transforms are used to increase the dataset diversity and to prevent overfitting.

4. **Dataset and Dataloader**:

Here the **imageFolder** loads images from **train** and **val** directories.

BATCH_SIZE is nothing but grouping the datasets of 32 images.

Shuffle=True used for training the order of images to improve

Learning the dataset.

5**. Setting up of Model**:

Firstly, selection of the device takes place. Mostly uses GPU (cuda)

If available, otherwise CPU.

The model we are using is a pre-trained ResNet-50 model.

weights=ResNet50_Weights.DEFAULTindicates ImageNet weights.

requires_grad = False parameter is used to freeze the layers by preventing the pre-trained layers from being updated during training and this process is said to be transfer learning.

- nn.Linear(2048, 256): Reduces the feature dimension from 2048 to 256.
- nn.ReLU(): Applies ReLU activation for non-linearity.
- nn.Dropout(0.4): Randomly drops 40% of neurons during training to prevent overfitting.
- nn.Linear(256, 2): Outputs logits for two classes.

criterion: CrossEntropyLoss combines log-softmax and negative log-likelihood loss, suitable for multi-class classification (here, binary).

optimizer: Adam optimizer updates only the fc layer parameters (since other layers are frozen), with a learning rate of 0.001.

scheduler: ReduceLROnPlateau reduces the learning rate by a factor of 0.1 if the validation loss doesn't improve for 2 epochs (patience=2). This helps fine-tune learning when the model plateaus.

Initialization:

best_acc: Tracks the highest validation accuracy to save the best model.

patience and counter: Used for early stopping if performance doesn't improve.

Training Mode: model.train() enables dropout and batch normalization for training.

Batch Loop:

Moves images and labels to the device.

Clears previous gradients (optimizer.zero_grad()).

Forward pass: Computes model outputs (outputs = model(images)).

Computes loss (criterion(outputs, labels)).

Backward pass: Computes gradients (loss.backward()).

Updates weights (optimizer.step()).

Rate Scheduling:

Model Checkpointing:

If validation accuracy improves, saves the model weights to **MODEL_SAVE_PATH** and resets **counter**.

Uses **model.state_dict()** to save only the model parameters.

Early Stopping:

Increments **counter** if accuracy doesn't improve.

Stops training if **counter** reaches **patience** (3 epochs), preventing overfitting.

print (f" Model saved to {MODEL_SAVE_PATH}")

Indicates that the training is complete and the best model is saved.

Key Concepts:

- **Transfer Learning**: Leverages ResNet-50's pre-trained weights to extract features, training only the final layer for the new task.
- **Data Augmentation**: Random horizontal flips increase training data diversity.
- **Early Stopping**: Prevents overfitting by stopping training when performance plateaus.
- **Learning Rate Scheduling**: Dynamically adjusts the learning rate to improve convergence.
- **Binary Classification**: The model outputs logits for two classes, optimized using cross-entropy loss.

How the prediction_model.py works

1. Loading the Image Checker (Model)

- **Imagine:** You have a special tool, like a machine, that has been trained to look at pictures and decide if they are "Safe" or "Unsafe." This code starts by *finding* and *loading* this tool into the computer's memory.
- **What it does:**
    - **model = models.resnet50()**: This line is like saying, "Get the special tool!"
    - The next few lines fine-tune the tool a little, making it extra good at its specific job of deciding if images are safe or unsafe.
    - **model.load_state_dict(...)**: This line is like "Loading the instructions" of the special tool, which are stored in a file named "**moderation_model.pth**".
    - **model.eval()**: This line tells the tool to be in "evaluation mode," meaning it's ready to give its judgments.
    - If the file with the tool's instructions (**moderation_model.pth**) isn't found or something goes wrong, you will get an error message, and the program will stop.

2. Defining the Possible Results (Labels)

- **Imagine:** The tool can only give two answers: "Safe" or "Unsafe".
- **What it does:**
    - **class_names** = ['Safe', 'Unsafe']: This line simply creates a list of the possible answers.

3. Preparing the Picture for the Checker

- **Imagine:** Before the tool can look at a picture, you need to get it ready. You need to make sure it's the right size and format.
- **What it does:**
    - transform = **transforms.Compose**([...]): This is a list of instructions on how to get the image ready.
        - **transforms.Resize**((224, 224)): Make all the pictures a certain size (like 224 pixels wide and 224 pixels tall).
        - **transforms.ToTensor**(): Convert the picture into a format the tool can understand.
        - **transforms.Normalize**(...): Adjust the color values in the picture.

4. The Image Checker in Action

- **Imagine:** This part of the code feeds the prepared picture to your special tool, which then examines it and gives its verdict.
- **What it does (inside the predict_and_annotate function):**
    - Image.open(image_path).convert("RGB"): Opens the image.
    - ImageOps.exif_transpose(...): Corrects image orientation if needed.
    - transform(pil_img).unsqueeze(0): Applies the preparation steps (resizing, converting, normalizing).
    - with **torch.no_grad()**: output = model(img): Shows the prepared image to the tool and gets the results.
    - probs = F.softmax(output, dim=1): The tool gives a probability(the possibility) for both of the results (Safe or Unsafe). For example, there might be an 80% probability of the image being "Unsafe" and a 20% probability of it being "Safe."
    - pred = **torch.argmax**(probs).item(): The program finds the prediction with the highest probability.
    - confidence = probs[0][pred].item(): The confidence (percentage) of the tool's judgment.
    - label = **class_names**[pred]: The actual verdict ("Safe" or "Unsafe")
    - The code then prints the results to the screen, including the probabilities for each label (e.g., "Safe: 20.0%, Unsafe: 80.0%").

5. Showing the Result and Adding a Label (Annotating)

- **Imagine:** This is where the code takes the tool's answer and shows it in a visually understandable way.
- **What it does:**
    - orig = np.array(pil_img): Converts the image to a format that can be modified.
    - annotated = cv2.cvtColor(orig, cv2.COLOR_RGB2BGR): Makes the image in color.
    - if label == 'Unsafe': cv2.rectangle(...): If the tool said "Unsafe", it draws a red rectangle around the entire image to indicate it's unsafe.
    - text = f"{label} ({confidence*100:.1f}%)": Creates text like "Unsafe (80.0%)" to show the verdict and the confidence level.
    - cv2.putText(...): Adds the text to the image.
    - cv2.imshow("Moderation Result", annotated): Shows the image with the label.
    - cv2.waitKey(0): Waits until a key is pressed.

- cv2.destroyAllWindows(): Closes the image window.
- print(f"Moderation Result: {label} with {confidence*100:.1f}% confidence"): Finally, prints the final result in the terminal.

6. Trying It Out (The Last Line)

- **Imagine:** This last line tells the program to use all the above steps to analyze a *specific* image file that you provide.
- **What it does:**
  - predict_and_annotate(r"D:\RTP\Content Moderation\full data\Graphically Safe Images\human faces images_99.jpeg"): This tells the program to open and analyze the image at the given address. The program will then display the result.

The code loads a pre-trained tool to classify images. It prepares an image, gets the tool's verdict, and then shows the image with a label and a possible red box (if it is "Unsafe"). The final result is also printed in the terminal.