Final model in part1 description:

DenseNet-like Architecture with Transfer Learning

Architecture: Custom DenseNet

- **Dense Layer**: Each dense layer consists of Batch Normalization, ReLU activation, and a Convolutional layer. The output is concatenated with the input feature maps.
- **Dense Block**: A sequence of dense layers, where each layer receives inputs from all preceding layers.
- **Transition Layer**: Contains Batch Normalization, ReLU activation, a Convolutional layer, and an Average Pooling layer to reduce dimensionality.
- **Custom DenseNet Structure**:
    - Initial Convolutional layer with 64 filters.
    - Four dense blocks with a growth rate of 32 and layers configuration [6, 12, 24, 16].
    - Transition layers between dense blocks to downsample the feature maps.
    - Global average pooling followed by a fully connected layer for classification.

Transfer Learning:

- **Pre-trained on ImageNet**: The model was initialized with weights from pre-trained models on ImageNet.
- **Ensemble Model**: Three individual models were trained separately, and their predictions were combined to form an ensemble model. Each model was fine-tuned on the specific dataset.

Training Strategy:

- **Optimizer**: AdamW with learning rate scheduling.
- **Early Stopping**: Implemented to halt training when the validation loss stopped improving.

PART 2: Data Augmentation and Regularization Techniques

Augmentation and Regularization:

- **MixCut Augmentation**: A data augmentation technique that combines parts of two images to create a new training sample, which helps the model generalize better.
- **Criterion**: CrossEntropyLoss was used as the loss function.
- **Regularization Methods**: Various regularization techniques, including dropout, weight decay, and gradient clipping, were applied to prevent overfitting.

Observations:

- When the DenseNet-like architecture from PART 1 was used in PART 2 without transfer learning, it resulted in lower accuracy.
- Conversely, applying MixCut augmentation and regularization methods from PART 2 to the transfer learning model in PART 1 also led to suboptimal results.

Final model in part1 description:

1. Model Architecture

- **Base Model:** ResNet-18
  - Pre-trained on ImageNet (weights: `IMAGENET1K_V1`)
  - Final fully connected layer adjusted to output 43 classes (specific to the problem at hand).

2. Class Balancing

- **Class Weights:**
  - Computed using `sklearn.utils.class_weight.compute_class_weight` to handle class imbalance in the training dataset.
  - Applied these weights to the CrossEntropyLoss function to ensure balanced learning.

3. Data Augmentation

- **Mixup Augmentation:**
  - Implemented a mixup strategy to augment the training data.
  - Combines pairs of training examples with a mixup ratio drawn from a Beta distribution (alpha=0.2).
  - Helps in regularizing the model and improving its robustness by encouraging it to predict a linear combination of labels.

4. Training Enhancements

- **Gradient Clipping:**
  - Applied gradient clipping (`torch.nn.utils.clip_grad_norm_`) with a max norm of 1.0 to prevent exploding gradients during backpropagation.

5. Optimization and Learning Rate Scheduling

- **Optimizer:**
  - Used `AdamW` optimizer, which is Adam with weight decay to improve regularization.
- **Learning Rate Schedulers:**
  - **ReduceLROnPlateau:**
    - Monitors validation loss and reduces learning rate by a factor of 0.1 if no improvement is seen for 3 epochs.
  - **CosineAnnealingLR:**

    - Adjusts the learning rate following a cosine annealing schedule over a period (`T_max=10`), with a minimum learning rate (`eta_min=1e-6`).

6. Early Stopping

- **Early Stopping Mechanism:**
    - Implemented to halt training when validation loss stops improving for 15 consecutive epochs.
    - Restores the best model weights observed during training to avoid overfitting.

7. Hyperparameter Tuning

- **Learning Rates:**
    - Experimented with learning rates of 0.0001 and 0.0003.
- **Weight Decays:**
    - Tested with weight decay values of `1e-2` and `1e-4`.
- **Schedulers:**
    - Combined each learning rate and weight decay with both `ReduceLROnPlateau` and `CosineAnnealingLR` schedulers to find the optimal configuration. (`CosineAnnealingLR shows better result`)

8. Evaluation Metrics

- **Accuracy and Loss:**
    - Tracked both training and validation accuracy and loss to monitor the performance and generalization ability of the model.

Results Overview

- Conducted multiple training experiments with different hyperparameter settings. (`lr=0.0001, weight_decay=0.01`)
- Each experiment's performance in terms of validation accuracy and loss was recorded.
- The best performing model and hyperparameter configuration were identified based on maximum validation accuracy achieved. (`Epoch [14/25] Train Loss: 0.5971, Train Acc: 0.9058, Val Loss: 0.8160, Val Acc: 0.8277`)

Our project faced several challenges, including the loss of many initial experiment runs. When we attempted to rerun these experiments, some data remained irretrievably lost. We used a custom data loader instead of the one provided. Initially, labels were binary (0 and 1) instead of product names, which was later corrected.

We experimented with various hyperparameters and built multiple models. Despite our efforts, the accuracy was disappointingly low. For example, validation accuracy for untrained models

was around 20%, excluding ResNet. To improve accuracy, we increased model complexity, but saw no significant improvement. We then implemented a basic ResNet model, achieving a validation accuracy of around 30-40%.

Then we decided to experiment with several data augmentations. While most attempts yielded lower accuracy, the augmentations we retained provided the best validation accuracies. To diagnose our low validation accuracy, we developed a function to find the most optimal model. We discovered that incorrect labels were common, particularly in classes with the most data, indicating a “data distribution” issue.

To solve this problem, we came up with one idea, and it was Batch Sampling (it means selecting balanced subsets of data during training to ensure fair representation of all classes). This method, designed to balance data distribution, did not improve results. So we used 3 other methods, and only the last one worked out the way we wanted it to:
1. Mixed-Up Augmentation: A data augmentation technique that creates new training examples by combining pairs of examples and their labels.

2. Mixed-Up Criterion: A loss function that combines the losses of mixed-up samples, promoting better generalization.

3. Gradient Clipping: A technique to prevent the exploding gradient problem by capping the gradients during backpropagation.

Our model was overfitting some data and underfitting others, so we tried Ensemble Learning (which is combining multiple models to improve overall performance). We built three individual models with various subclasses based on dataset sizes:

1. Subclass 1: classes with huge datasets

2. Subclass 2: classes with small datasets

3. Subclass 3: two versions:

- Randomly picked 36 data points (average of data over all datasets) from each class.
- Classes with neither huge, nor small datasets, but a reasonable amount.

Unfortunately, this, too, did not yield desired accuracy improvements.

We then made our complexity go higher, and realized that there are some papers suggesting that using DenseNet will give us better results. So we implemented a DenseNet-like convolutional architecture, which increased accuracy from 15% to 30-40%.

Later on we experimented with three ensemble learning methods:

1. Voting: An ensemble method where each model votes for a class, and the majority vote is chosen

2. Weighted average: Combining models by giving different weights to their predictions.

3. Ensemble model complex: An ensemble method that combines multiple models and add fully-connected layers to their outputs to enhance prediction accuracy. (This provided the best results)

We also used Boosting, which is a technique that combines weak models sequentially to improve performance, but similar to ensemble learning, did not yield good results.

Our ensemble model combined simple models with fully connected layers, achieving a validation accuracy of 49-50%.

We used various optimizers like SGD (an optimizer that updates model parameters using random subsets of data, promoting faster convergence) and ADAM (an optimizer that combines the advantages of two other extensions of SGD, providing fast and adaptive learning rates) and also different schedule learners like Reduce LR on Plateau (reduces the learning rate when a metric has stopped improving, to allow finer adjustments), Step LR (decays the learning rate by a factor every few epochs to enable gradual fine-tuning) and One Cycle LR (adjusts the learning rate according to a cyclical schedule that varies between lower and upper bounds, often leading to better model performance) to adjust learning rates during training, which improved accuracy only by 3-4%.

Our final solution was Transfer Learning, which is a technique where a pre-trained model is fine-tuned on a new task. We trained a custom DenseNet on ImageNet data, achieving an accuracy of 57-58%. By training for 4-5 epochs instead of 1-2, we reached 60% accuracy, meeting our project goal.

For the second part of our project, we used ResNet with provided weights, achieving a maximum validation accuracy of 78%. Using mixed-up methods, we increased this to 80% by further tuning hyperparameters.

Despite numerous challenges and setbacks, including lost data and initially low accuracy, we achieved our target accuracy through persistent experimentation and innovative techniques such as ensemble learning, DenseNet architecture, and transfer learning. Our final model met the project's accuracy requirements, demonstrating significant improvement from our initial attempts.