Implementation Report for Exercise 3

By: Katayoon Kobraei

In this exercise, we were tasked with predicting Intel's dataset using well-known models like ResNet and applying transfer learning. Here is a step-by-step breakdown of the implementation process:

Step 1: Data Preprocessing

After downloading and uploading the dataset, the first task was to make some modifications to the data. For example, we used transformation functions from the `transforms` module to augment the data. This step was crucial because the more data we have, the better our model's prediction accuracy becomes. Another essential preprocessing step was normalizing the data. This normalization was applied to both the training and testing data after the transformations.

Key Point:

During the transformation step, it was important to choose functions that didn't alter the number of input features, as this could cause issues during model execution.

```
data_transforms = {
  train: transforms.Compose([
    transforms.CenterCrop(224),
    transforms.RandomHorizontalFlip(0.5),
    transforms.RandomRotation(degrees=(0,180)),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
  ]),
  test: transforms.Compose([
    transforms.RandomResizedCrop(224),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
  ]),
```

Step 2: Model Selection

In the next step, we downloaded the desired models. As specified in the exercise, we used the `models` module to download and implement various models like ResNet and others. For transfer learning, we needed to freeze the model weights for the initial epochs so that the new data wouldn't disrupt the original model weights. The only modification required was replacing the final fully connected (FC) layers.

To redesign the FC layers, we used at least two linear layers, with intermediate layers utilizing the ReLU activation function and the final layer using either `LogSoftmax` or `Softmax` to achieve optimal results.

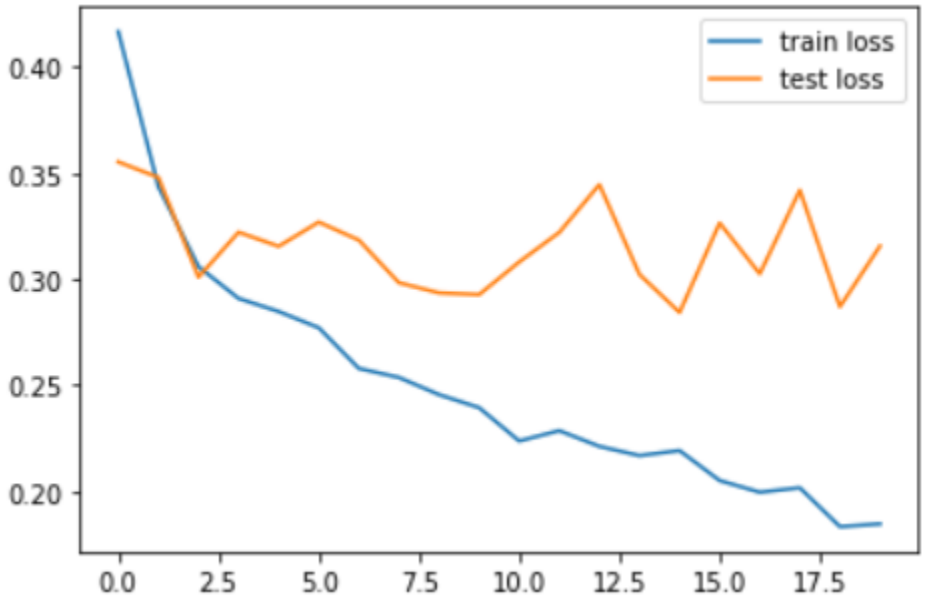
```
for param in resnet.parameters():
    param.requires_grad = False


resnet.fc = nn.Sequential(nn.Linear(2048, 1024),
                          nn.ReLU(),
                          nn.Dropout(0.25),
                          nn.Linear(1024, 256),
                          nn.ReLU(),
                          nn.Dropout(0.25),
                          nn.Linear(256, 6),
                          nn.LogSoftmax(dim=1))
```

Step 3: Model Training

After modifying the final layers, we moved the model to the available GPU for training. For all models, we used **cross-entropy loss** as the criterion and **Adam** as the optimizer. Initially, we set a learning rate of 0.1, but this was too high, causing the test and train losses to fluctuate significantly, preventing us from achieving optimal accuracy. We eventually reduced the learning rate to 0.001 and 0.0001, which stabilized the training process.

We then trained the models and described their performance using charts:

1. **Best Model (ResNet-101):**
   This model achieved the highest accuracy of **89.93%**. The training approach involved freezing the base model for the first 10-15 epochs to prevent the new data from disrupting the pre-trained weights. After this, we unfroze the entire model and trained it as a whole, leading to the best accuracy. The training loss at each epoch was recorded and visualized using a plot.



2. **Accuracy Calculation:**
   Using the written functions, we computed the accuracy for both training and testing datasets, along with the final accuracy.

```
accuracy(dataloader[test_data], model)
```

```
  0%|          | 0/94 [00:00<?, ?it/s]
Got 2689/3000 with accuracy 89.6333
```

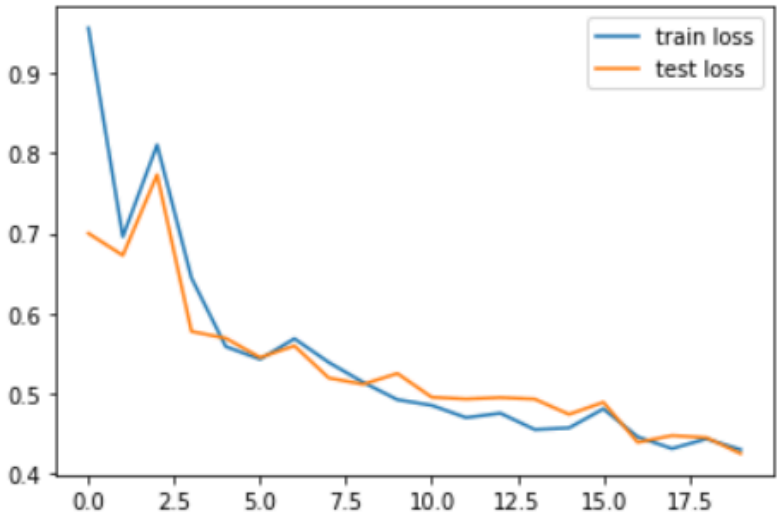
Step 4: Additional Optimization (Scheduler)

One additional technique we could employ was adding a learning rate scheduler to further improve the training process. However, our model required more experimentation for the scheduler to be fully effective. Still, the training loss at the start of each epoch showed noticeable improvement.

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=1e-4)
exp_lr_scheduler = lr_scheduler.StepLR(optimizer, step_size=7, gamma=0.1)
```

Step 5: Comparison with Other Models

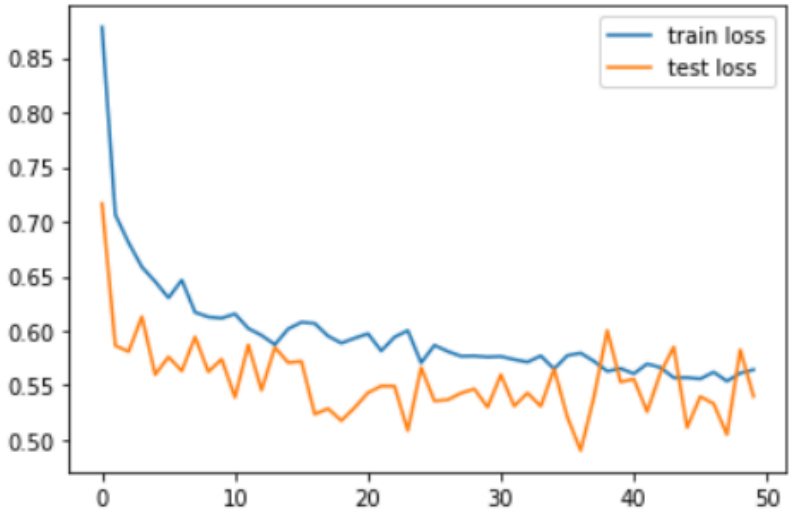
Another model we implemented was **DenseNet**, which also yielded good accuracy and was worth considering. The train and test losses were very close, indicating good model quality. After measuring the loss across epochs, we achieved an accuracy of **85.73%**.




```
In [53]: accuracy(test_loader, model)
  0%|          | 0/47 [00:00<?, ?it/s]
Got 2656/3000 with accuracy 88.5333

In [ ]:
```

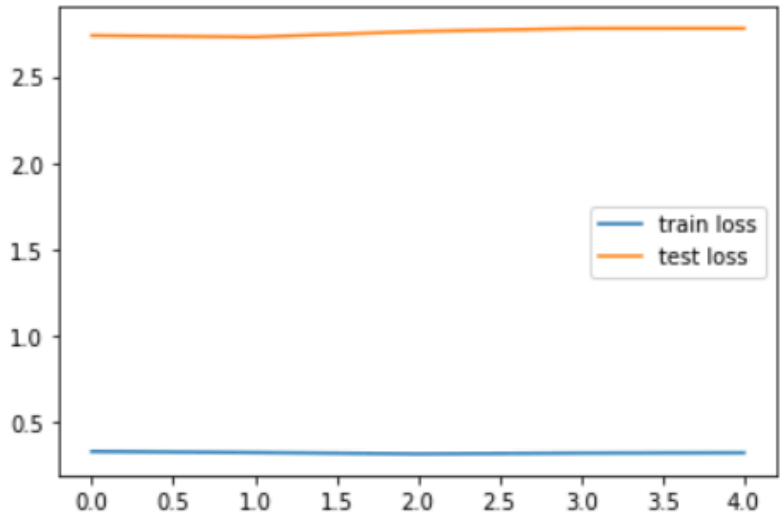
Next, we implemented **Inception-ResNet**, which, surprisingly, performed worse than DenseNet and ResNet-101. Its accuracy was significantly lower than expected.

```
In [83]: accuracy(dataloader[test], inception_resnet)
           0%|          | 0/94 [00:00<?, ?it/s]
         Got 2505/3000 with accuracy 83.5000

In [ ]:
```



Step 6: Overfitting Issues

In some models, we encountered overfitting issues, where the difference between train and test losses was too large. This problem was clearly identifiable through the plotted loss graphs.




```
Epoch 1/5, Train_Loss: 0.3340, \Test_Loss: 2.7370, Duration: 0:01:45.264610
Epoch 2/5, Train_Loss: 0.3278, \Test_Loss: 2.7281, Duration: 0:01:44.532685
Epoch 3/5, Train_Loss: 0.3204, \Test_Loss: 2.7605, Duration: 0:01:42.578312
Epoch 4/5, Train_Loss: 0.3247, \Test_Loss: 2.7776, Duration: 0:01:41.351926
Epoch 5/5, Train_Loss: 0.3264, \Test_Loss: 2.7779, Duration: 0:01:41.551378
```

Conclusion

By adjusting the training steps, tweaking hyperparameters, and properly loading the test and train data, we were able to resolve these issues and improve the models' performance.