INDIVIDUAL REPORT: ACUTE LYMPHOBLASTIC LEUKEMIA (ALL) CLASSIFICATION

Stefani Guevara

Machine Learning II | Final Project

Introduction

In this project my team looked at image classification on the blood cells of Acute Lymphoblastic Leukemia (ALL) patients. To conduct the project, we looked at how others have approached the classification problem and experimented with similar approaches through the PyTorch framework. As we conducted the project, we experimented with different pretrained networks including resnet, densenet, googlenet, vgg and efficientnet.

Outline of shared work:

- Chris prepared the data-loading section of the code, thus allowing us to begin working on the experimentations. He also provided notes and input that went into preparing the Project Proposal. Additionally, Chris focused on experimenting with data augmentation for this project.
- Dylan investigated implementing certain pretrained networks including VGG and EfficientNet and helping in coordinating the project plan.
- I did research on what other papers have done on this dataset, prepped and finalized the Proposal, and helped coordinate the schedule for completing the project. I focused on experimenting with different pretrained networks, including EfficientNet via PyTorch, which obtained better results more consistently and provided the code to the team for each of us to attempt a different version of the pretrained model.

Description of Individual Work

We decided to use Exam2 as a template given that we chose to use the PyTorch framework. The majority of the script of Exam2 was left in tack as it didn't hinder the program compiling and we could just start testing different pretrained networks and finetuning.

Project Proposal

Gathered notes on the data and on papers (found through Papers with Code) that have looked at classification on this dataset, as well as Kaggle notebooks. Particularly took notes on Prellberg & Kramer's 2020 paper which helped provide information on what metrics to use and a baseline for the project.

Script

I updated the script to include the f1_weighted and coh metrics. Since the data is heavily unbalanced with the Hem cell images constituting half of ALL cell images, I doubled the number of instances of the minority class by extending the list of Hem cells by itself:

```
H.extend(train_dataset_0_hem)
H.extend(train_dataset_1_hem)
H.extend(train_dataset_2_hem)
H.extend(H)  # double the instances of the number of minority classes
```

The resulting population would be closer to equal but still somewhat unbalanced with cancer cells at 7,272 count and normal cells at 6,778 count, where the original split was 7,272 to 3,389. I used different pretrained networks including variations of resnet and densenet, as well as

googlenet and efficientnet. We saw a popular Kaggle notebook implemented Efficient-Net on this dataset, so I attempted to implement efficientnet through the PyTorch framework myself. The EfficientNet pretrained architecture for PyTorch was just released Oct 21, 2021 so little information is currently available online on how to use it, but through debugging I found the required output for a Linear layer was 1280, therefore used this for the classifier:

```
if pretrained == True:
    model = models.efficientnet_b0(pretrained=True)
    model.classifier = nn.Linear(1280, OUTPUTS_a)
```

Additionally implemented basic augmentations based on what had been done in Prellberg & Kramer's paper: random rotation, random vertical and horizontal flips and center cropping. Flips were done using RandomApply with a 50% probability, and rotations were done with a 30% probability. Center crops were done to 300x300 size from 400.

```
def augmentation_step(X, rotate=False):

    transform = transforms.RandomApply([transforms.RandomVerticalFlip(), transforms.RandomHorizontalFlip()], p=0.5)
    rotation = transforms.RandomApply([transforms.RandomRotation(degrees=(0, 270))], p=0.3)
    normalize = transforms.Compose([transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225)),
                                    transforms.CenterCrop(300)])

    X = transform(X)
    if rotate:
        X = rotation(X)
    X = normalize(X)

    return X
```

Models were finetuned on the learning rate, optimizer, image size, number of epochs, and batch_size. We used channel 3 as color is a factor when differentiating between ALL cells and normal cells.

Results

Densenet

Several experiments were conducted for each model. For densenet, with the original unbalanced data, 5 epochs and batch size of 30 obtained the following results:

```
# Epoch 4: 100%|██████████| 356/356 [41:22<00:00,  6.97s/it, Test Loss: 0.27253]
# Epoch 4: 100%|██████████| 63/63 [01:46<00:00,  1.69s/it, Test Loss: 0.61142]
# Epoch 4:  Train acc 0.88575 Train f1_weighted 0.88550 Train coh 0.73539 Train avg
0.62666 -  Test acc 0.66899 Test f1_weighted 0.67438 Test coh 0.29954 Test avg 0.41073
```

Multiple attempts were done varying the number of epochs and batch size as well as image size under the densenet pretrained model, however they did not result in more impressive scores.

The only data augmentation step included in these series of runs with densenet was an image normalization using transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225)).

Below are the results with the densenet pretrained model after doubling the number of instances of the normal hem cells and training with 3 epochs. Image size remained at 300x300, however the augmentation step this time included random vertical and horizontal flips using RandomApply with a 50% probability. As we can see, the scores were higher, between low and mid 70s.

```
# Epoch 2: 100%|██████████| 440/440 [49:43<00:00,  6.78s/it, Test Loss: 0.25944]
# Epoch 2: 100%|██████████| 59/59 [01:42<00:00,  1.74s/it, Test Loss: 0.55175]
# Epoch 2:  Train acc 0.88769 Train f1_weighted 0.88770 Train coh 0.77516 Train avg
0.63764 -  Test acc 0.74826 Test f1_weighted 0.75226 Test coh 0.46654 Test avg 0.49177
# The model has been saved!
# Epoch 3: 100%|██████████| 440/440 [48:50<00:00,  6.66s/it, Test Loss: 0.23834]
# Epoch 3: 100%|██████████| 59/59 [01:43<00:00,  1.76s/it, Test Loss: 0.68798]
# Epoch 3:  Train acc 0.89701 Train f1_weighted 0.89703 Train coh 0.79386 Train avg
0.64698 -  Test acc 0.70809 Test f1_weighted 0.71485 Test coh 0.42475 Test avg 0.46192
```

With this Densenet model and with the remaining models used to experiment on this classification problem, we left the number of instances of the minority class doubled on the training set, as mentioned previously.

Resnet

The results with resnet152 were consistently underwhelming with low scores ranging below 70s, even with higher epochs. Testing with the resnet pretrained model included the same procedure as that with the densenet pretrained network. Below are the results of the last run made with resnet using 3 epochs. As shown, the results on the test set were lower than with densenet.

```
# Epoch 0: 100%|██████████| 440/440 [1:42:09<00:00, 13.93s/it, Test Loss: 0.48582]
# Epoch 0: 100%|██████████| 59/59 [04:03<00:00,  4.13s/it, Test Loss: 0.58998]
# Epoch 0:  Train acc 0.76712 Train f1_weighted 0.76697 Train coh 0.53523 Train avg
0.51733 -  Test acc 0.68345 Test f1_weighted 0.68373 Test coh 0.30283 Test avg 0.41750
# The model has been saved!
# Epoch 1: 100%|██████████| 440/440 [1:42:30<00:00, 13.98s/it, Test Loss: 0.43937]
# Epoch 1: 100%|██████████| 59/59 [03:59<00:00,  4.07s/it, Test Loss: 0.84616]
# Epoch 1:  Train acc 0.79196 Train f1_weighted 0.79201 Train coh 0.58411 Train avg
0.54202 -  Test acc 0.48741 Test f1_weighted 0.47345 Test coh 0.09870 Test avg 0.26489
# Epoch 2: 100%|██████████| 440/440 [1:42:30<00:00, 13.98s/it, Test Loss: 0.41374]
# Epoch 2: 100%|██████████| 59/59 [04:01<00:00,  4.09s/it, Test Loss: 0.69911]
# Epoch 2:  Train acc 0.80349 Train f1_weighted 0.80354 Train coh 0.60719 Train avg
0.55355 -  Test acc 0.53830 Test f1_weighted 0.54091 Test coh 0.14611 Test avg 0.30633
```

Googlenet

The googlenet pretrained network with basic augmentation – normalization and random flips – provided better scores than resnet using the same number of epochs, and marginally higher than with densenet, reaching up to mid 70's after the second epoch.

```
# Epoch 0: 100%|██████████| 440/440 [19:40<00:00,  2.68s/it, Test Loss: 0.41123]
# Epoch 0: 100%|██████████| 59/59 [01:03<00:00,  1.07s/it, Test Loss: 1.77636]
# Epoch 0:  Train acc 0.80648 Train f1_weighted 0.80653 Train coh 0.61312 Train avg
```

```
0.55653 -  Test acc 0.65345 Test f1_weighted 0.54521 Test coh 0.03126 Test avg 0.30748
# The model has been saved!
# Epoch 1: 100%|██████████| 440/440 [20:58<00:00,  2.86s/it, Test Loss: 0.33766]
# Epoch 1: 100%|██████████| 59/59 [01:03<00:00,  1.07s/it, Test Loss: 0.47616]
# Epoch 1:  Train acc 0.84484 Train f1_weighted 0.84488 Train coh 0.68949 Train avg
0.59480 -  Test acc 0.75362 Test f1_weighted 0.75791 Test coh 0.48078 Test avg 0.49808
# The model has been saved!
# Epoch 2: 100%|██████████| 440/440 [20:55<00:00,  2.85s/it, Test Loss: 0.30488]
# Epoch 2: 100%|██████████| 59/59 [01:03<00:00,  1.08s/it, Test Loss: 0.64873]
# Epoch 2:  Train acc 0.86192 Train f1_weighted 0.86196 Train coh 0.72378 Train avg
0.61192 -  Test acc 0.69255 Test f1_weighted 0.69115 Test coh 0.31572 Test avg 0.42486
```

Efficientnet

Influenced by this Kaggle notebook by Ashish Goswami, we wanted to look at how EfficientNet would perform on this dataset, considering it's a much newer architecture than the previously attempted pretrained networks; moreover, Goswami approached this dataset with Keras while we used PyTorch.

Each member attempted a different version of torchvision's EfficientNet, thus I implemented efficientnet_b0. On the first attempt with efficientnet_b0 and basic augmentation with 7 epochs our model did better than the previous models, reaching up to high 70's on the test set:

```
# Epoch 5: 100%|██████████| 440/440 [22:46<00:00,  3.11s/it, Test Loss: 0.20562]
# Epoch 5: 100%|██████████| 59/59 [00:35<00:00,  1.67it/s, Test Loss: 0.50617]
# Epoch 5:  Train acc 0.91046 Train f1_weighted 0.91049 Train coh 0.82088 Train avg
0.66046 -  Test acc 0.77290 Test f1_weighted 0.77383 Test coh 0.50324 Test avg 0.51249
# Epoch 6: 100%|██████████| 440/440 [22:47<00:00,  3.11s/it, Test Loss: 0.17257]
# Epoch 6: 100%|██████████| 59/59 [00:35<00:00,  1.68it/s, Test Loss: 0.69576]
# Epoch 6:  Train acc 0.92484 Train f1_weighted 0.92486 Train coh 0.84960 Train avg
0.67483 -  Test acc 0.75469 Test f1_weighted 0.75276 Test coh 0.45081 Test avg 0.48956
```

This same model reached our highest score on its second epoch at an accuracy of 77.667% and weighted F1 score of 77.62%.

```
Epoch 2: 100%|██████████| 440/440 [22:47<00:00,  3.11s/it, Test Loss:
0.26712]
# Epoch 2: 100%|██████████| 59/59 [00:34<00:00,  1.69it/s, Test Loss:
0.53067]
# Epoch 2:  Train acc 0.88292 Train f1_weighted 0.88295 Train coh 0.76570
Train avg 0.63289 -  Test acc 0.77665 Test f1_weighted 0.77619 Test coh
0.50523 Test avg 0.51452
# The model has been saved!
```

Delving into details, this model used a batch size of 32, image size of 300x300, an Adam optimizer with the default learning rate, and a ReduceLROnPlateau scheduler. We left the factor for the scheduler at 0.5 and while we varied the patience in other tests, in this specific test we left it at 0. In addition, we used simple data augmentation of randomly applying vertical and horizontal flips with a probability of 0.5, followed by image normalization as done previously.

My next best model obtained an accuracy score of 76.96% and a weighted F1 score of 76.98%. This model used all else the same as the previous model with an additional center crop of 200x200 dimension, patience of 3 on the scheduler and 9 total epochs.

```
# Epoch 8: 100%|██████████| 440/440 [10:23<00:00,  1.42s/it, Test Loss: 0.18673]
# Epoch 8: 100%|██████████| 59/59 [00:17<00:00,  3.45it/s, Test Loss: 0.61700]
# Epoch 8:  Train acc 0.92441 Train f1_weighted 0.92443 Train coh 0.84876 Train avg
0.67440 -  Test acc 0.76968 Test f1_weighted 0.76985 Test coh 0.49257 Test avg 0.50803
# The model has been saved!
```

Additional variations on the above model – such as removing the scheduler, changing the learning rate, and increasing the number of epochs by as much as 10 – resulted in comparable if not slightly lower performances.

```
# Epoch 7:  Train acc 0.91132 Train f1_weighted 0.91134 Train coh 0.82256 Train avg
0.66131 -  Test acc 0.72094 Test f1_weighted 0.72079 Test coh 0.38363 Test avg 0.45634
# Epoch 8: 100%|██████████| 562/562 [13:20<00:00,  1.42s/it, Test Loss: 0.18451]
# Epoch 8: 100%|██████████| 75/75 [00:19<00:00,  3.77it/s, Test Loss: 0.78374]
# Epoch 8:  Train acc 0.92391 Train f1_weighted 0.92393 Train coh 0.84776 Train avg
0.67390 -  Test acc 0.68827 Test f1_weighted 0.69381 Test coh 0.34398 Test avg 0.43151
# Epoch 9: 100%|██████████| 562/562 [13:30<00:00,  1.44s/it, Test Loss: 0.16582]
# Epoch 9: 100%|██████████| 75/75 [00:19<00:00,  3.76it/s, Test Loss: 0.98224]
# Epoch 9:  Train acc 0.93132 Train f1_weighted 0.93134 Train coh 0.86260 Train avg
0.68131 -  Test acc 0.72309 Test f1_weighted 0.71831 Test coh 0.37059 Test avg 0.45300
```

Ultimately, among all the tested pretrained networks, the EfficientNet pretrained network did provide some of the best results on this classification problem.

Summary and Conclusions

In conclusion, my best model resulted from implementing EfficientNet's baseline architecture with rather minimal data augmentation. The key approach to my experimentation involved testing different pretrained networks, and fortunately, EfficientNet was just recently released into torchvision's models package. While my model did not outperform others that have worked on this dataset (namely Prellberg & Kramer, 88%, and Goswami, 93%) it did perform competitively with many other models submitted to Kaggle. For this dataset specifically, EfficientNet provided the best results likely due to its ability to capture more granular details in the images – necessary for this type of classification as, at least to the untrained eye, it is difficult to differentiate between ALL cells and Hem cells. EfficientNet also conveniently provided faster results that our other models.

Due to time constraints, we did not have the chance to test out customized learning rates, which our outperforming baseline models used. This could potentially improve the results on our own model and can be looked at for future work.

About 500 lines of code from the Exam2 template and the data aggregation and loading were used similar to code from the Leukemia Classification Kaggle Dataset, which took about 55 lines that replaced the original Exam2 template. I modified about 15-20 lines and added about 18-20 of my own lines.

(500-20)/(500+20)× *100 = ~92%*

References

Acute Lymphoblastic Leukemia Classification from Microscopic Images using Convolutional Neural Networks (2020 paper)
Acute Lymphoblastic Leukemia Detection from Microscopic Images Using Weighted Ensemble of Convolutional Neural Networks (2021 paper)
https://wiki.cancerimagingarchive.net/pages/viewpage.action?pageId=52758223#52758223bcab02c187174a288dbcbf95d26179e8
Best deep CNN architectures and their principles: from AlexNet to EfficientNet
Committed Towards Better Future: EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks
EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks (2020)