Deep Learning – Lab 3

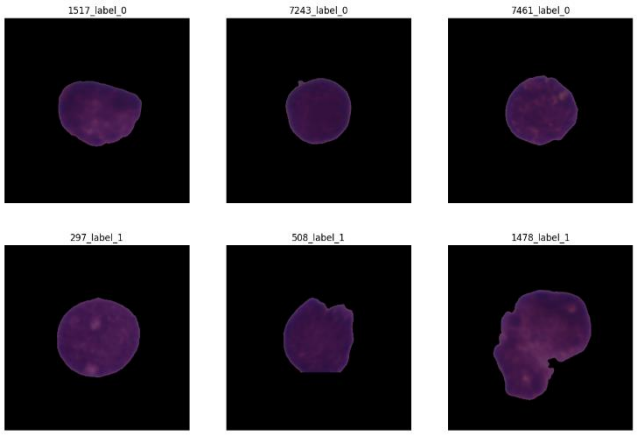
Leukemia Classification

I. Introduction

i. ResNet

- In the past, deep convolutional neural networks faced the issue of vanishing gradients, which led to the problem of feature disappearance when the networks became too deep. Therefore, prior to the release of ResNet, most network designs focused on being complex and intricate to achieve good accuracy with a relatively small number of layers, as seen in VGGNet from 2014.
- However, with the introduction of ResNet and its residual learning architecture, neural networks were able to overcome the problem of vanishing gradients associated with depth. This breakthrough allowed networks to be significantly deepened practically overnight. The deepened ResNet, with its simple design, easily outperformed previous networks that relied on complex architectures, resulting in a substantial improvement in performance.
- ResNet's accuracy outperformed other models by a significant margin. It can be clearly seen that ResNet is undoubtedly a milestone in the field of artificial intelligence.

ii. Data



- The data in this case is used for blood cancer recognition through white blood cells. In the image, the upper half represents normal white blood cells (0), and the lower half represents white blood cells with the disease (1).

II. Implementation Details

i. The details of your model (ResNet)

```python
class BasicBlock(nn.Module):
    def __init__(self, in_channel, out_channel):
        super().__init__()
        self.in_channel = in_channel
        self.out_channel = out_channel
        self.ReLU = nn.ReLU(inplace = True)
        first_stride = 1
        if in_channel != out_channel:
            first_stride = 2
            self.downsample = nn.Sequential(
                nn.Conv2d(in_channel, out_channel, kernel_size = (1, 1), stride = (first_stride, first_stride), bias = False),
                nn.BatchNorm2d(out_channel, eps = 1e-5, momentum = 0.1, affine = True, track_running_stats = True)
            )
        self.layer = nn.Sequential(
            nn.Conv2d(in_channel, out_channel, kernel_size = (3, 3), stride = (first_stride, first_stride),padding=(1, 1),  bias = False),
            nn.BatchNorm2d(out_channel, eps = 1e-5, momentum = 0.1, affine = True, track_running_stats = True),
            nn.ReLU(inplace = True),
            nn.Conv2d(out_channel, out_channel, kernel_size = (3, 3), stride = (1, 1), padding = (1, 1), bias = False),
            nn.BatchNorm2d(out_channel, eps = 1e-5, momentum = 0.1, affine = True, track_running_stats = True)
        )
    def forward(self, input):
        if self.in_channel != self.out_channel:
            return self.ReLU(self.layer(input) + self.downsample(input))
        return self.ReLU(self.layer(input) + input)
```

```python
class Bottleneck(nn.Module):
    def __init__(self, in_channel, cross_channel, out_channel, cross_stride = 1):
        super().__init__()
        self.in_channel = in_channel
        self.cross_channel = cross_channel
        self.out_channel = out_channel
        self.cross_stride = cross_stride
        self.ReLU = nn.ReLU(inplace = True)
        self.layer = nn.Sequential(
            nn.Conv2d(in_channel, cross_channel, kernel_size = (1, 1), stride = (1, 1), bias = False),
            nn.BatchNorm2d(cross_channel, eps = 1e-5, momentum = 0.1, affine = True, track_running_stats = True),
            nn.Conv2d(cross_channel, cross_channel, kernel_size = (3, 3), stride = (cross_stride, cross_stride), padding = (1, 1), bias = False),
            nn.BatchNorm2d(cross_channel, eps = 1e-5, momentum = 0.1, affine = True, track_running_stats = True),
            nn.Conv2d(cross_channel, out_channel, kernel_size = (1, 1), stride = (1, 1), bias = False),
            nn.BatchNorm2d(out_channel, eps = 1e-5, momentum = 0.1, affine = True, track_running_stats = True),
            nn.ReLU(inplace = True)
        )
        if in_channel != out_channel:
            self.downsample = nn.Sequential(
                nn.Conv2d(in_channel, out_channel, kernel_size = (1, 1), stride = (cross_stride, cross_stride), bias = False),
                nn.BatchNorm2d(out_channel, eps = 1e-5, momentum = 0.1, affine = True, track_running_stats = True)
            )
    def forward(self, input):
        if self.in_channel != self.out_channel:
            return self.ReLU(self.layer(input) + self.downsample(input))
        return self.ReLU(self.layer(input) + input)
```

- ResNet18 uses Basic Blocks as its basic building units, while ResNet50 and ResNet152 use Bottleneck building units. The fundamental concept in both cases remains the same, employing residual learning, which involves adding the identity that hasn't gone through the convolutional layers to the output that has.
- When the input channels and output channels of the convolutional layer differ, the identity must undergo a downsample process to adjust its dimensions to match those of the convolutional layer output. After this downsample process, the two can be element-wise added together. This downsample step ensures that the element-wise addition can be performed even when the dimensions are not the same.
- Such a structure allows ResNet to train deeper networks effortlessly. The

inclusion of residual learning ensures that deeper networks do not suffer from vanishing or exploding gradients, making the training process more stable and facilitating convergence.

ii. Dataloader

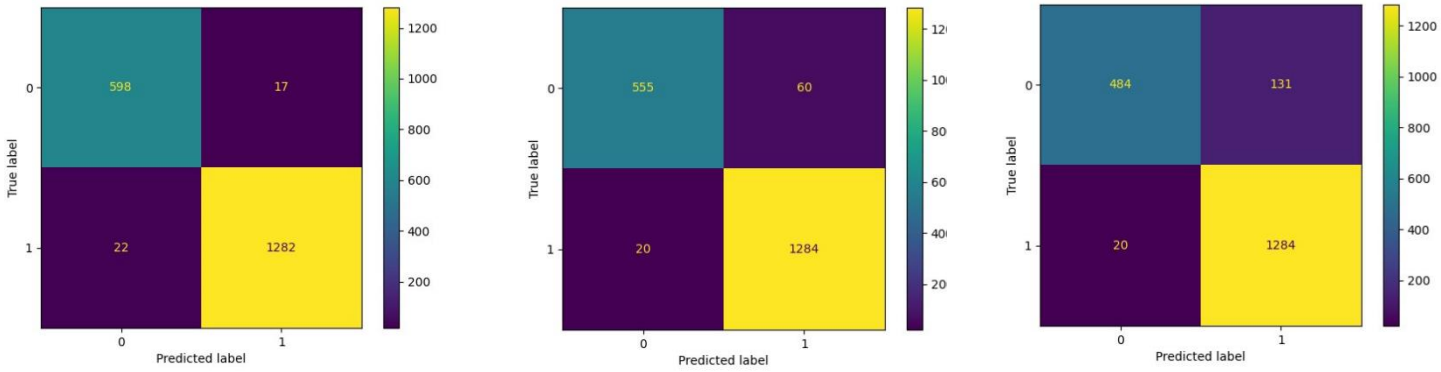
```
def get_image_path(path, base_data_path = base_data_path):
    image_path = os.path.join(base_data_path, path.split('./')[-1])
    return image_path
def get_image(image_path):
    image = cv2.imread(image_path)
    image_RGB = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    return image_RGB
```

- Firstly, my dataloader has two basic functions, get_image_path() and get_image(). They are just get image path and get image.

```
class LeukemiaDataset(Dataset):
    def __init__(self, df, is_test_model = False, transforms = None):
        self.df = df
        self.is_test_model = is_test_model
        self.transforms = transforms
    def __len__(self):
        return len(self.df)
    def __getitem__(self, index):
        image_id = self.df.iloc[index]['Path']
        image_path = get_image_path(image_id)
        image = get_image(image_path)
        if self.transforms:
            image = self.transforms(image = image)['image']
        if self.is_test_model:
            return image_id, image
        label = self.df.iloc[index]['label']
        return image_id, image, label
```

- Next is my main Dataset class. It will accept a dataframe and read the required data based on the index. If this dataset is for the test set, it will return the Image ID (Path) and the image. Otherwise, it will return the Image ID (Path), the image, and the Label for training purposes.

iii. Describing your evaluation through the confusion matrix



- In the images from left to right is ResNet18, ResNet50, and ResNet152. From the images, we can observe that, except for ResNet18, the models tend to predict class 0 as class 1 with a much higher probability compared to predicting class 1 as class 0. Interestingly, if we examine the dataset, we notice that class 1 samples slightly outnumber class 0 samples. Logically, the models should be more prone to misclassifying class 1 as class 0 since there are more opportunities for errors due to the larger number of class 1 samples.

- I believe the reason for this behavior lies in the training duration of ResNet50 and ResNet152, as I did not train them for many epochs due to their longer training times. It's possible that the models haven't fully converged and, therefore, haven't truly learned how to discriminate between the classes. Consequently, when the models are uncertain about the class of a data sample, they tend to guess class 1 (as it results in lower loss due to the higher probability of class 1). I expect that when the models are trained adequately, they should exhibit a more balanced error rate on both classes, similar to ResNet18.

III. Data Preprocessing

i. How you preprocessed your data?

```python
def get_train_transforms():
    return albumentations.Compose([
        albumentations.CenterCrop(cfg.image_size, cfg.image_size, p = 1.0),
        albumentations.HorizontalFlip(p = 0.5),
        albumentations.VerticalFlip(p = 0.5),
        albumentations.Transpose(p = 0.5),
        albumentations.Rotate(limit = (-180, 180), p = 0.5),
        albumentations.GaussianBlur(always_apply = False, blur_limit = 3, p = 0.2),
        albumentations.GaussNoise(var_limit = (30.0, 30.0), mean = 0, always_apply = False, p = 0.2),
        albumentations.Cutout(max_h_size = int(cfg.image_size * 0.125), max_w_size = int(cfg.image_size * 0.125), num_holes = 1, p = 0.5),
        albumentations.Normalize(mean = [0.485,0.456,0.406], std = [0.229,0.224,0.225], max_pixel_value = 255, p=1.0),
        ToTensorV2(p = 1.0)
    ])
```

- This is my data preprocessing, which includes basic data augmentations such as flipping, rotation, and transpose. For this specific task, I believe such augmentations should not have any significant impact on the model's recognition performance. Therefore, I set the probabilities for flipping and transpose to 0.5 each, meaning an equal chance of applying or not applying these augmentations.
  - For the same reason, I set the probability for rotation to 1 initially. However, I found that it was not as effective as setting it to 0.5. My personal conjecture for this observation is that rotation might cause some parts of the images to be extrapolated and stitched back, which could lead to difficulties in model training.
- To add noise and enhance model robustness during training, I incorporated Gaussian noise, Gaussian blur, and Cut out techniques. I believe that training a model is about competing to train it deeper and longer without overfitting. Introducing noise can help prevent overfitting and make the later stages of training more challenging, avoiding excessively low training loss that might hinder model updates.
- Finally, the Center crop step is intended to remove black background areas, allowing the model to focus more on the cell parts while reducing the input data size, which can help in memory usage reduction.

ii. What makes your method special?

- I choose basic transformations for data augmentation because I believe the main differences between Label 0 and 1 in this task lie in cell size, cell color (brightness), and possibly some aspects of cell shape. Therefore, I think that adding augmentations such as scaling or brightness transformations could

potentially make the model struggle in classification.

- Additionally, I have utilized the ttach library, which conceptually performs multiple flips and rotations on the images during the test process. For each transformation, the model makes a prediction, and the final result is integrated (resembling a k-fold cross-validation approach but applied through augmentation). This strategy helps improve the model's robustness and reduces the impact of potential biases in individual test samples.
  - Source of the ttach library: https://github.com/qubvel/ttach

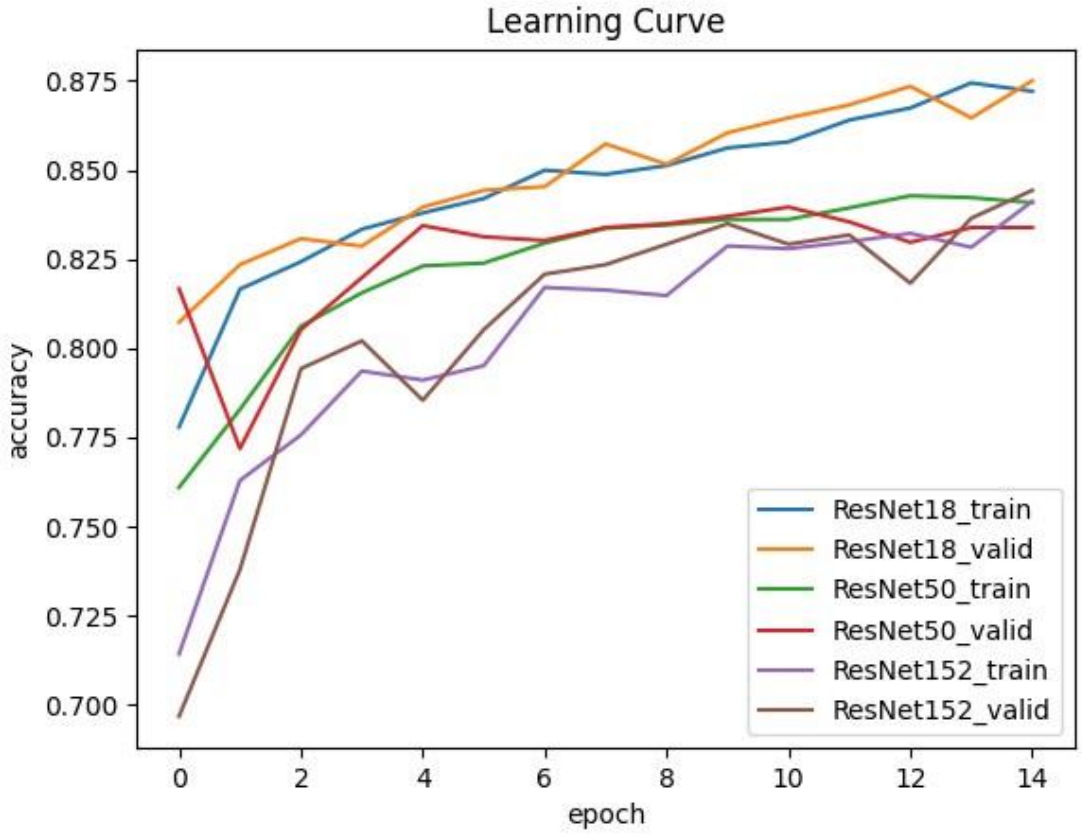
IV. Experimental results

i. The highest testing accuracy

```
average_accuracy=0.984
```

0.98313 ☐

ii. Comparison figures



- From the graph, it can be observed that ResNet18 trains faster compared to ResNet50 and ResNet152. I believe this is because ResNet18 has fewer parameters to learn, and the task at hand is relatively simple. Consequently, ResNet18 can quickly learn important features. On the other hand, ResNet50 and ResNet152 have more features to learn, requiring more time to get into the groove. However, with a sufficient number of epochs, ResNet50 or ResNet152 may eventually surpass ResNet18 in accuracy.
  - By the way, this graph was plotted under the condition of using Data Augmentation, which helps suppress the occurrence of Overfitting.

V. Discussion

i. The problem with model.eval()

- I have observed that when I include "model.eval()" during model training on the validation phase, the model's validation becomes very unstable. In the worst case scenario, it may even result in the model outputting all 1s. Initially, I suspected that there might be an issue with my model implementation, but I tested it with the ResNet18 load from PyTorch and encountered the same problem. Interestingly, some of my classmates faced identical issues. Even after completing the assignment, I still do not understand the root cause of this behavior.

- Strangely, I found that if I do not include "model.eval()" during validation but include it during testing, the model performs perfectly. Both sections of the code are almost identical, so I do not actually know how it happend…

ii. The problem with kfold

- In this Lab, I used k-fold cross-validation. Initially, I used the subset method provided by PyTorch's dataset to split the dataset (following methods like this: https://stackoverflow.com/questions/60883696/k-fold-cross-validation-using-dataloaders-in-pytorch). However, I encountered terrible overfitting issues with this approach. Surprisingly, when I used the same splitting strategy, but manually split the dataframe instead of using the built-in subset method, the overfitting problem disappeared. In my opinion, these two methods should yield the same results, but the actual outcomes were quite unexpected. I suspect that the issue may be related to the dataset's sampler, but I am not entirely sure.

iii. Optimizer Ranger21

- I used to use AdamW in combination with a scheduler(used to be CosineAnnealingWarmRestarts). However, since the release of Ranger21, I decided to give Ranger21 a try. The challenge I faced was that Ranger21 incorporates a warm-up mechanism that adjusts the learning rate internally. As a result, I am unsure how to properly combine it with a scheduler. I have not come across any discussions online regarding this topic either. Consequently, up until now, I have been training the model without using a scheduler, but I might need to figure out how to integrate the two in the future.

iv. Convert RGB to Gray

- When I initially examined the image data in this lab, I believed that the main differences between the images of class 0 and class 1 lie in cell size and cell brightness, with shape being of secondary importance. Additionally, the cell colors in both classes were predominantly purple. As a result, I thought that the colorfulness of the images would not be helpful for this task. Therefore, I attempted to convert the images from color to grayscale, which not only reduced the dimensionality significantly, leading to reduced computational load but also potentially allowed the model to focus more on brightness rather than color. Well, it turns in a very very very bad result QQ.

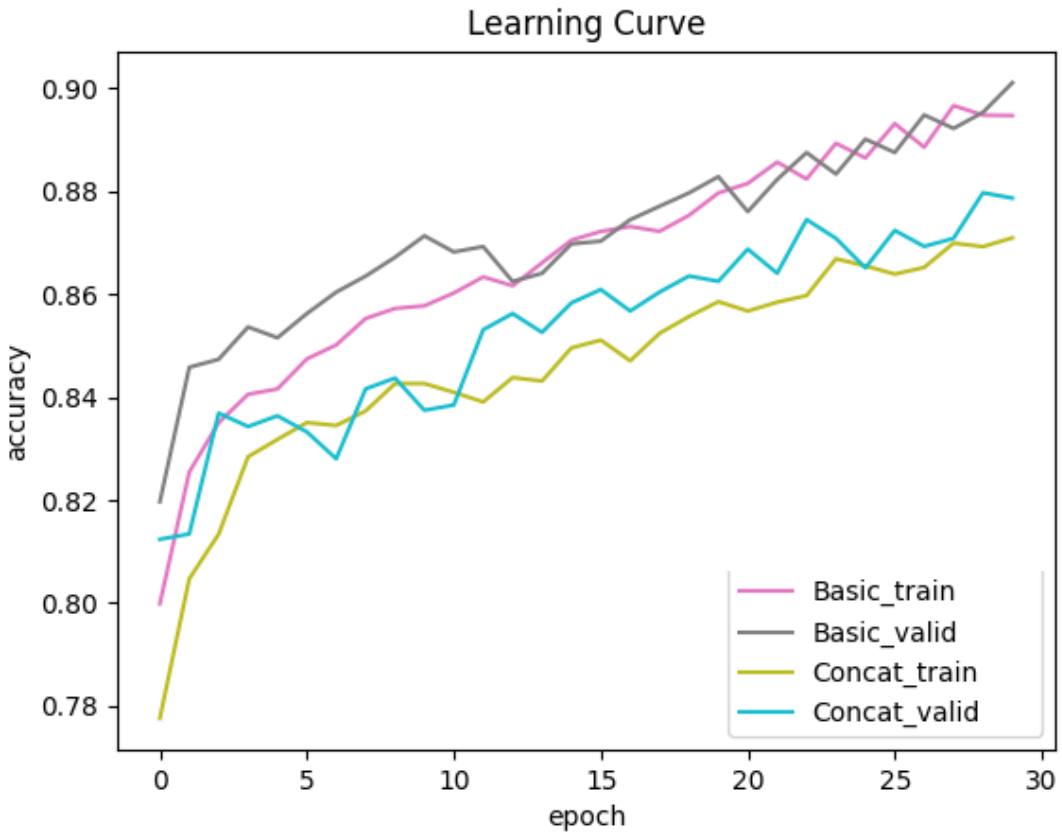
v. Addition vs Concatenate

- While implementing ResNet, I noticed that the shortcut connections are implement through addition. As I understand it, these shortcuts are introduced to prevent feature vanishing by passing previous layer features to subsequent layers. However, I think the direct addition of features to be somewhat unreasonable. The importance of past features may not necessarily be equal to the features produced by the current layer. Therefore, it would be more appropriate to apply weighted summation rather than a simple addition.

- This led me to think, why not concatenate the features and then apply a CNN for embedding? Consequently, I conducted this experiment.

```
class BasicBlock(nn.Module):
    def __init__(self, in_channel, out_channel):
        super().__init__()
        self.in_channel = in_channel
        self.out_channel = out_channel
        self.ReLU = nn.ReLU(inplace = True)
        first_stride = 1
        if in_channel != out_channel:
            first_stride = 2
            self.downsample = nn.Sequential(
                nn.Conv2d(in_channel, out_channel, kernel_size = (1, 1), stride = (first_stride, first_stride), bias = False),
                nn.BatchNorm2d(out_channel, eps = 1e-5, momentum = 0.1, affine = True, track_running_stats = True)
            )
        self.layer = nn.Sequential(
            nn.Conv2d(in_channel, out_channel, kernel_size = (3, 3), stride = (first_stride, first_stride),padding=(1, 1),  bias = False),
            nn.BatchNorm2d(out_channel, eps = 1e-5, momentum = 0.1, affine = True, track_running_stats = True),
            nn.ReLU(inplace = True),
            nn.Conv2d(out_channel, out_channel, kernel_size = (3, 3), stride = (1, 1), padding = (1, 1), bias = False),
            nn.BatchNorm2d(out_channel, eps = 1e-5, momentum = 0.1, affine = True, track_running_stats = True)
        )
        self.conv = nn.Sequential(
            nn.Conv2d(out_channel * 2, out_channel, kernel_size = (3, 3), stride = (1, 1), padding = (1, 1), bias = False),
            nn.ReLU()
        )
    def forward(self, input):
        if self.in_channel != self.out_channel:
            output = torch.cat((self.layer(input), self.downsample(input)), 1)
            return self.conv(output)
        output = torch.cat((self.layer(input), input), 1)
        return self.conv(output)
```

- Above is the modified ResNet18 network that I have implemented. The number of parameters has increased approximately twofold.



- Next is my experimental result. As it can be seen, the actual training results of the concatenated features were not as good as the addition method. I'm not entirely sure why, but my guess is that it might be due to the significantly increased number of parameters. The model needs to spend more time learning many less meaningful parameters, leading to slower model growth.
- Perhaps with more training time, it could achieve better results, but I feel that if that's the case, it would be more effective to deepen the network (e.g., directly using ResNet50).