312554012 王偉誠 Lab3: Leukemia Classification

1. Introduction

For this assignment, we are tasked with utilizing ResNet models, specifically ResNet18, ResNet50, and ResNet152, to conduct a classification analysis on Leukemia. The classification involves two distinct categories. Our primary objective is to implement these three models for classification purposes.

In addition, we are required to implement the getitem() function within the dataloader. This function serves a dual purpose: not only does it return the images themselves, but it also undertakes preprocessing steps aimed at enhancing the accuracy of model predictions.

2. Implementation Details:

A. The detail of your model (ResNet)

ResNet employs a technique known as residual learning to address the challenges associated with training deep convolutional neural network (CNN) models, particularly the degradation problem.

Degradation problem

The phenomenon of deeper networks sometimes yielding poorer results is known as the degradation problem. As the depth of a network increases, there can be a saturation or even a decline in network accuracy. This phenomenon is evident in the diagram below, where the 56-layer network performs worse than the 20-layer network. Importantly, this is not attributable to overfitting, as the training error for the 56-layer network remains high. It is acknowledged that deep networks encounter challenges like gradient vanishing or explosion, rendering the training of deep learning models arduous.

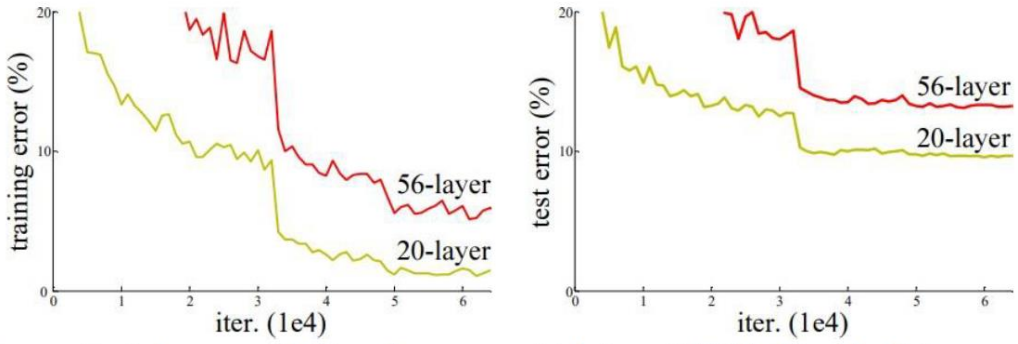
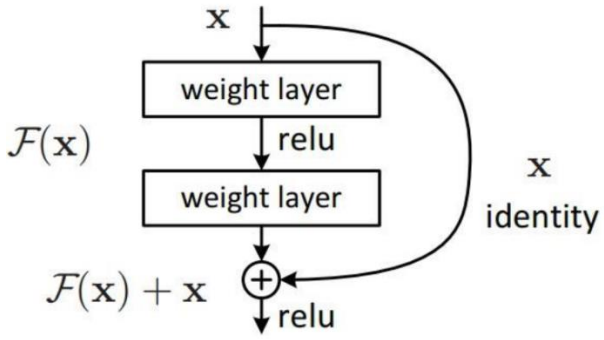


Figure. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer “plain” networks. The deeper network has higher training error, and thus test error.

Skip/Shortcut connection

To address the challenges posed by vanishing or exploding gradients, a skip or shortcut connection has been introduced. This connection involves adding the input data x to the output after traversing several weight layers, as illustrated in the diagram below.



Residual learning

Residual learning primarily addresses the aforementioned Degradation problem associated with training deep networks. In the context of deep neural networks, where the training process becomes challenging, the concept of residual learning aims to counteract this issue. The core idea of residual learning is rooted in the scenario where attempting to construct a deep network by stacking new layers may lead to a situation where the additional layers fail to learn anything meaningful. Instead, they replicate the features of the earlier network. In this context, these new layers are referred to as identity mappings. This extreme case can be addressed by the notion of identity mapping, where the accuracy would at least be comparable to that of a shallower network, preventing degradation.

To apply this idea to convolutional neural networks (CNNs), the technique of Skip/Shortcut connections is utilized. As depicted in the diagram provided, the input is denoted as x, and the learned features are represented by H(x). The residual, denoted as F(x) = H(x) - x, is also learned. Consequently, the learned features can be expressed as H(x) = F(x) + x. This means that when the residual is close to zero, the stacked layers effectively perform identity mapping. By incorporating the residual learning approach, the network avoids the Degradation problem. As a result, deep networks can be trained more effectively, mitigating challenges associated with vanishing gradients and facilitating smoother optimization.

Why ResNet can avoid vanishing gradient problem?

$$y_l = h(x_l) + F(x_l, W_l)$$
$$x_{l+1} = f(y_l)$$

The notations x_ℓ and $x_{\ell+1}$ respectively denote the input and output of the ℓ-th residual unit, where each residual unit typically consists of multiple layers. F represents the residual function, capturing the learned residual, while $h(x_\ell) = x_\ell$ signifies the identity mapping, and f represents the ReLU activation function. Therefore, the process of learning features from the shallow layer ℓ to the deep layer $\mathcal{L}$ can be expressed as follows:

$$x_L = x_l + \sum_{i=l}^{L-1} F(x_i, W_i)$$

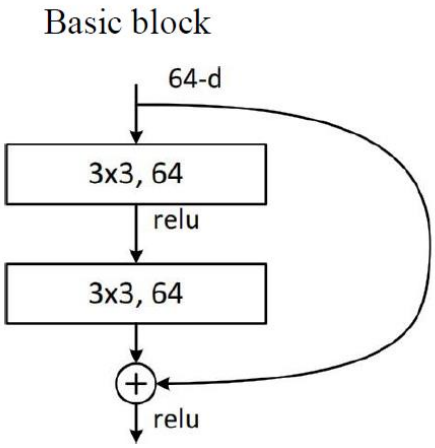
By employing the Chain Rule, we can compute the gradients during the backward propagation process.

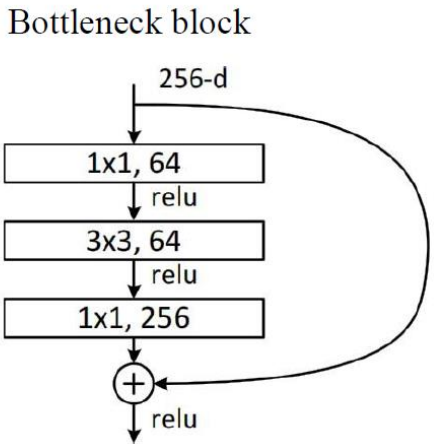
$$\frac{\partial loss}{\partial x_l} = \frac{\partial loss}{\partial x_L} \cdot \frac{\partial x_L}{\partial x_l} = \frac{\partial loss}{\partial x_L} \cdot \left(1 + \frac{\partial}{\partial x_l} \sum_{i=l}^{L-1} F(x_i, W_i)\right)$$

The first term of the equation, $\frac{\partial \text{loss}}{\partial x_\mathcal{L}}$, represents the gradient of the loss function with respect to the output of the L-th layer. On the other hand, the gradient of the residual needs to traverse through the weight layers, and the presence of the term within the parentheses, 1, signifies the ability of the shortcut mechanism to propagate gradients without loss, preventing gradient vanishing. As a result, residual learning facilitates more straightforward training due to its inherent capability to alleviate gradient-related challenges.

Building residual basic block and bottleneck block:

The concepts described above correspond to the fundamental building block depicted in the lower-left diagram. ResNet is composed of these basic blocks. As the network becomes deeper, such as in the case of ResNet50, ResNet introduces bottleneck blocks. These blocks decrease the width of the 3x3 convolution, significantly reducing computational requirements.





Implementation

Basic Block

```python
class Block(nn.Module):
    """
    ## Block
    this is a simple block of ResNet
    args:
        - in_channel: 3
        - out_channel: 64
        - i_downsample: None
        - stride: 1
    """

    expansion = 1

    def __init__(self, in_channel, out_channel, i_downsample=None, stride=1):
        super(Block, self).__init__()
        self.expansion = 1
        self.conv1 = nn.Conv2d(
            in_channel, out_channel, kernel_size=3, stride=1, padding=1
        )
        self.bn1 = nn.BatchNorm2d(out_channel)
        self.conv2 = nn.Conv2d(
            out_channel, out_channel, kernel_size=3, stride=stride, padding=1
        )
        self.bn2 = nn.BatchNorm2d(out_channel)
        self.relu = nn.ReLU()
        self.i_downsample = i_downsample
        self.stride = stride

    def forward(self, x):
        identity = x.clone()
        x = self.relu(self.bn1(self.conv1(x)))
        x = self.bn2(self.conv2(x))
        if self.i_downsample is not None:
            identity = self.i_downsample(identity)
        x += identity
        x = self.relu(x)
        return x
```

Bottleneck Block

```python
class Bottleneck(nn.Module):
    """
    ## Bottleneck
    This is a simple Bottleneck of ResNet
    args:
        - in_channel: 3
        - out_channel: 64
        - i_downsample: None
        - stride: 1
    """

    expansion = 4

    def __init__(self, in_channel, out_channel, i_downsample=None, stride=1):
        super(Bottleneck, self).__init__()
        self.expansion = 4
        self.conv1 = nn.Conv2d(
            in_channel, out_channel, kernel_size=1, stride=1, padding=0
        )
        self.bn1 = nn.BatchNorm2d(out_channel)
        self.conv2 = nn.Conv2d(
            out_channel, out_channel, kernel_size=3, stride=stride, padding=1
        )
        self.bn2 = nn.BatchNorm2d(out_channel)
        self.conv3 = nn.Conv2d(
            out_channel,
            out_channel * self.expansion,
            kernel_size=1,
            stride=1,
            padding=0,
        )
        self.bn3 = nn.BatchNorm2d(out_channel * self.expansion)
        self.relu = nn.ReLU()
        self.i_downsample = i_downsample
        self.stride = stride

    def forward(self, x):
        identity = x.clone()
        x = self.relu(self.bn1(self.conv1(x)))
        x = self.relu(self.bn2(self.conv2(x)))
        x = self.bn3(self.conv3(x))
        if self.i_downsample is not None:
            identity = self.i_downsample(identity)
        x += identity
        x = self.relu(x)
        return x
```

Main ResNet

```python
class ResNet(nn.Module):
    """
    ## ResNet
    This is a simple ResNet
    When you use this model, you can use ResNet18, ResNet34, ResNet50, ResNet101, ResNet152
    args:
        - block: Block or Bottleneck
        - layers: [2, 2, 2, 2] or [3, 4, 6, 3] or [3, 8, 36, 3]
        - image_channel: 3
        - num_classes: 2
    """

    def __init__(self, block, layers, image_channel, num_classes):
        super(ResNet, self).__init__()
        self.in_channel = 64
        self.conv1 = nn.Conv2d(image_channel, 64, kernel_size=7, stride=2, padding=3)
        self.bn1 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU()
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.layer1 = self._make_layer(block, layers[0], out_channel=64, stride=1)
        self.layer2 = self._make_layer(block, layers[1], out_channel=128, stride=2)
        self.layer3 = self._make_layer(block, layers[2], out_channel=256, stride=2)
        self.layer4 = self._make_layer(block, layers[3], out_channel=512, stride=2)
        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(512 * block.expansion, num_classes)

    def forward(self, x):
        x = self.relu(self.bn1(self.conv1(x)))
        x = self.maxpool(x)

        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)

        x = self.avgpool(x)
        x = x.reshape(x.shape[0], -1)
        x = self.fc(x)
        return x

    def _make_layer(self, block, num_residual_blocks, out_channel, stride):
        identity_downsample = None
        layers = []

        if stride != 1 or self.in_channel != out_channel * block.expansion:
            identity_downsample = nn.Sequential(
                nn.Conv2d(
                    self.in_channel,
                    out_channel * block.expansion,
                    kernel_size=1,
                    stride=stride,
                ),
                nn.BatchNorm2d(out_channel * block.expansion),
            )

        layers.append(block(self.in_channel, out_channel, identity_downsample, stride))
        self.in_channel = out_channel * block.expansion

        for i in range(num_residual_blocks - 1):
            layers.append(block(self.in_channel, out_channel))

        return nn.Sequential(*layers)

    def ResNet18(img_channel=3, num_classes=2):
        return ResNet(Block, [2, 2, 2, 2], img_channel, num_classes)

    def ResNet50(img_channel=3, num_classes=2):
        return ResNet(Bottleneck, [3, 4, 6, 3], img_channel, num_classes)

    def ResNet152(img_channel=3, num_classes=2):
        return ResNet(Bottleneck, [3, 8, 36, 3], img_channel, num_classes)
```

In the implementation of the ResNet architecture, I followed the block structure provided in the original paper and created two classes: Basic Block and Bottleneck Block. For each block, the forward method involves creating a clone of the input x to facilitate the implementation of the residual connection.

In the main ResNet structure, I designed the _make_layer function to dynamically construct different ResNet architectures based on the required number of blocks. For instance, ResNet18 utilizes the Basic Block, while layer1, layer2, layer3, and layer4 require [2, 2, 2, 2] blocks respectively.

B. The details of your Dataloader

The aspect of the assignment to be implemented involves the getitem() function within the DataLoader. Subsequently, the torch.utils.data.DataLoader can retrieve the corresponding data based on the implemented getitem() function.

As depicted in the diagram below, the implementation of the getitem() function follows the provided step-by-step hints. This entails initially retrieving the image and label based on the provided path and index. Subsequently, some preprocessing is applied to the data before being returned. The primary purpose of this preprocessing is to format the image to a suitable input type for the model and introduce additional variability and diversity to the image data.

A key consideration is that the transformations applied during training and testing should differ. Random cropping and flipping are employed during training to enhance data diversity and increase the training difficulty. Conversely, during testing, it is essential to simplify the process by eliminating random cropping and flipping. This adjustment is conducive to improving test accuracy. Consequently, distinct preprocessing steps are required for testing data, while avoiding actions that might cause distortion or deformation.

```python
def __getitem__(self, index):
    """something you should implement here"""

    """
       step1. Get the image path from 'self.img_name' and load it.
              hint : path = root + self.img_name[index] + '.jpeg'

       step2. Get the ground truth label from self.label

       step3. Transform the .jpeg rgb images during the training phase, such as resizing, random flipping,
              rotation, cropping, normalization etc. But at the beginning, I suggest you follow the hints.

              In the testing phase, if you have a normalization process during the training phase, you only need
              to normalize the data.

              hints : Convert the pixel value to [0, 1]
                      Transpose the image shape from [H, W, C] to [C, H, W]

       step4. Return processed image and label
    """

    train_transform = transforms.Compose(
        [
            transforms.RandomRotation(degrees=20),
            transforms.RandomHorizontalFlip(),
            transforms.RandomVerticalFlip(),
            transforms.ToTensor(),
            transforms.Normalize(
                mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]
            ),
        ]
    )

    test_transform = transforms.Compose(
        [
            transforms.ToTensor(),
            transforms.Normalize(
                mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]
            ),
        ]
    )

    # step 1
    path = self.root + self.img_name[index]
    # step 2
    if self.mode == "test":
        image = Image.open(path).convert("RGB")
        img = test_transform(image)
        return img
    else:
        label = self.label[index]

    # step 3
    image = Image.open(path).convert("RGB")
    if self.mode == "train":
        img = train_transform(image)
    else:
        img = test_transform(image)

    return img, label
```

- RandomRotation(degrees=d): This transformation randomly rotates the image within the range of (-d, d) degrees. Rotation may result in areas around the image's edges being filled with black pixels.
- RandomHorizontalFlip: Applying RandomHorizontalFlip(p=0.5) indicates that the image has a 0.5 probability of undergoing horizontal flipping.
- RandomVerticalFlip: Similarly, RandomVerticalFlip(p=0.5) indicates a 0.5 probability of the image undergoing vertical flipping.
- ToTensor: This transformation converts a PIL image or ndarray into a tensor, and it scales the pixel values to the range [0, 1]. Additionally, the image dimensions (H x W x C) are transposed to (C x H x W).
- Normalize: The Normalize transformation requires the mean and standard

deviation values for the images. Based on training with the ImageNet dataset, the calculated mean and standard deviation are mean = [0.485, 0.456, 0.406] and std = [0.229, 0.224, 0.225], respectively. ◦

C. Describing your evaluation through the confusion matrix

The confusion matrix is a commonly used tool for describing the performance of a classification model. It provides valuable insights into the model's predictions and errors across different classes. To visualize the confusion matrix, you can conveniently use the ConfusionMatrixDisplay function from the sklearn.metrics module. By providing the true labels and the model's prediction results, along with specifying normalize=True, you can generate a normalized confusion matrix that offers a clear representation of the model's classification performance. This visual representation aids in assessing the model's accuracy and potential areas of improvement.

Parameters:

y_true : *array-like of shape (n_samples,)*
True labels.

y_pred : *array-like of shape (n_samples,)*
The predicted labels given by the method `predict` of an classifier.

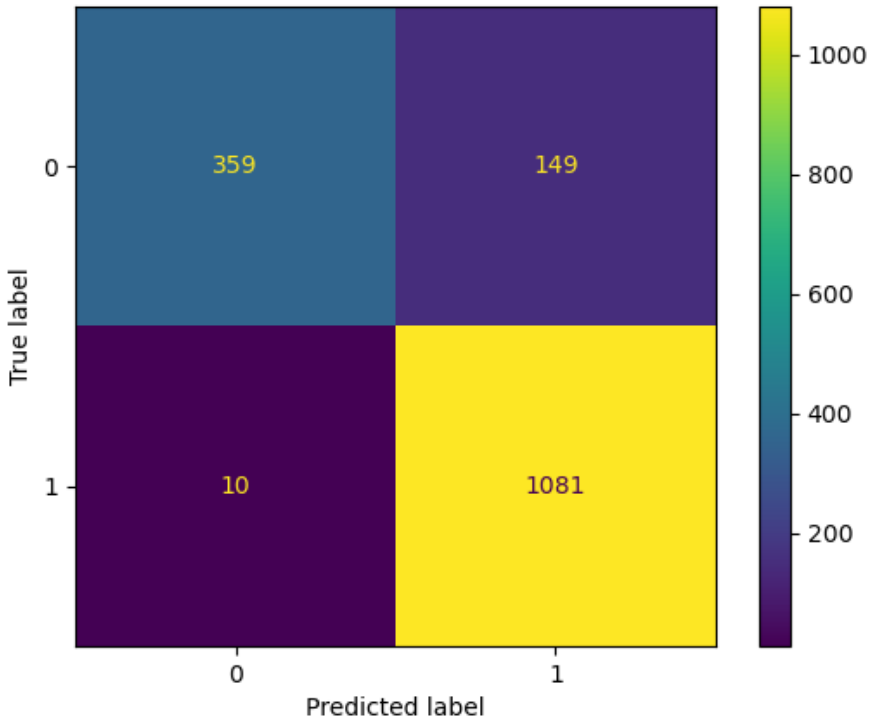
labels : *array-like of shape (n_classes,), default=None*
List of labels to index the confusion matrix. This may be used to reorder or select a subset of labels. If `None` is given, those that appear at least once in `y_true` or `y_pred` are used in sorted order.

sample_weight : *array-like of shape (n_samples,), default=None*
Sample weights.

normalize : *{'true', 'pred', 'all'}, default=None*
Either to normalize the counts display in the matrix:

- if `'true'`, the confusion matrix is normalized over the true conditions (e.g. rows);
- if `'pred'`, the confusion matrix is normalized over the predicted conditions (e.g. columns);
- if `'all'`, the confusion matrix is normalized by the total number of samples;
- if `None` (default), the confusion matrix will not be normalized.

I will use the following confusion matrix illustration to explain: The vertical axis represents the true labels, while the horizontal axis represents the predicted labels. For example, in the provided diagram, the value 359 in the top-left cell indicates that there are 359 instances where the prediction is 0 and the actual label is also 0. The value 10 in the cell denotes that there are 10 instances where the prediction is 0, but the true label is 1.



3. Data Preprocessing (20%)

A. How you preprocessed your data?

In the earlier implementation of the dataloader, preprocessing steps were introduced to prepare the images for compatibility with the model and to enhance the diversity of the image data. The preprocessing involved the utilization of techniques such as RandomCrop, RandomRotation, RandomHorizontalFlip, and RandomVerticalFlip. These techniques introduced random cropping, rotation, and horizontal or vertical flipping with a certain probability. Subsequently, the images were transformed into tensors and normalized, enabling the torch.utils.data.DataLoader to access the corresponding preprocessed data. This approach ensured that the data underwent preprocessing to capture a diverse range of variations.

During the testing phase, it is advisable to disable the operations that introduce image diversity, such as RandomRotation, RandomHorizontalFlip, and RandomVerticalFlip. This adjustment is made to enhance test accuracy. Additionally, the testing transform pipeline should focus on resizing the images proportionally, followed by a CenterCrop operation to extract the central pixels. Finally, the images are converted into tensors and normalized.

B. What makes your method special?

What sets my preprocess method apart is its comprehensive approach to preparing image data for compatibility with the model and enhancing its diversity. I introduce various preprocessing techniques, including RandomCrop, RandomRotation, RandomHorizontalFlip, and RandomVerticalFlip, which are applied with certain probabilities. This process ensures that the images undergo random cropping, rotation, and flipping, resulting in a diverse range of variations that can help improve the model's ability to generalize.

During the testing phase, I take a distinct approach by strategically disabling operations that introduce image diversity, such as RandomRotation, RandomHorizontalFlip, and RandomVerticalFlip. This adjustment aims to enhance the accuracy of testing results. My testing transform pipeline prioritizes proportional resizing of the images, followed by a CenterCrop operation to extract central pixels. Finally, the images are converted into tensors and normalized, allowing the torch.utils.data.DataLoader to access the appropriately preprocessed data for testing purposes. This careful balance between diversity-enhancing preprocessing during training and accuracy-focused preprocessing during testing showcases the uniqueness of my preprocess method.

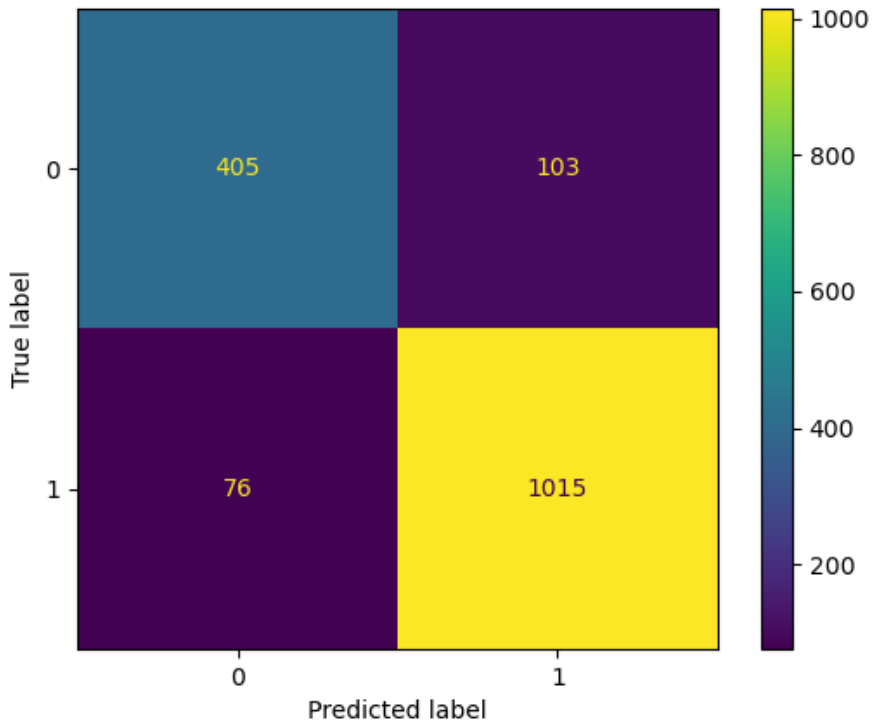
4. Experimental results

• Batch size= 32 for resnet18 / Batch size= 16 for resnet50 / Batch size= 8 for resnet152
• Learning rate = 1e-4
• Epochs = 25
• Optimizer: SGD
• Momentum = 0.9
• Weight Decay = 5e-3
• Loss function: Cross Entropy Loss

A. The highest testing accuracy

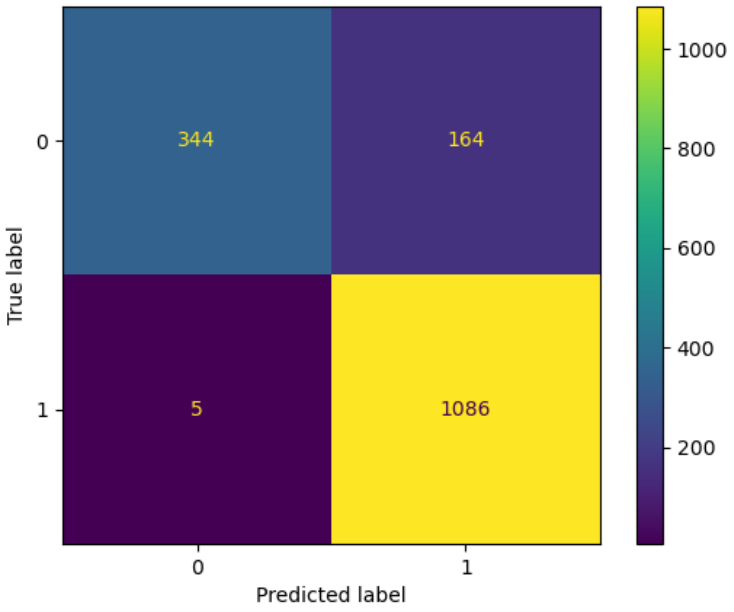
ResNet18

```
> Found 1599 images...
Accuracy of the ResNet18 on the valid images: 89.93120700437774 %
```



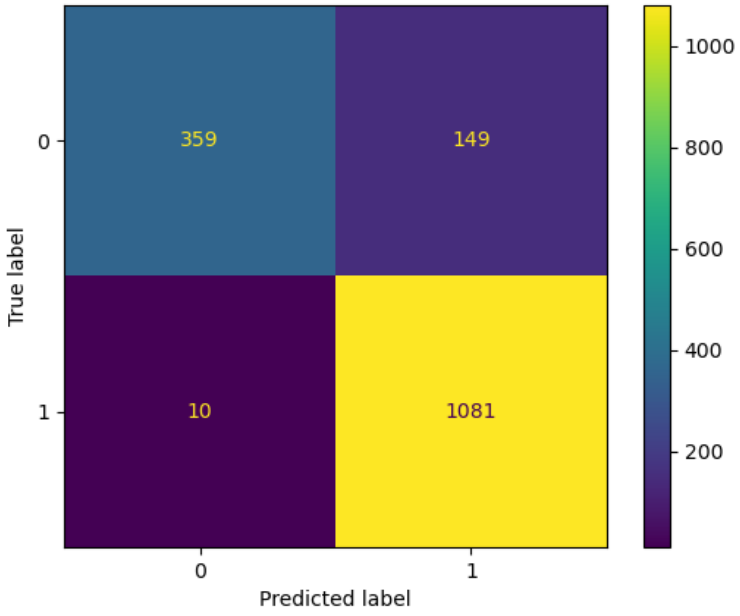
ResNet50

```
> Found 1599 images...
Accuracy of the ResNet50 on the valid images: 89.4308943089431 %
```

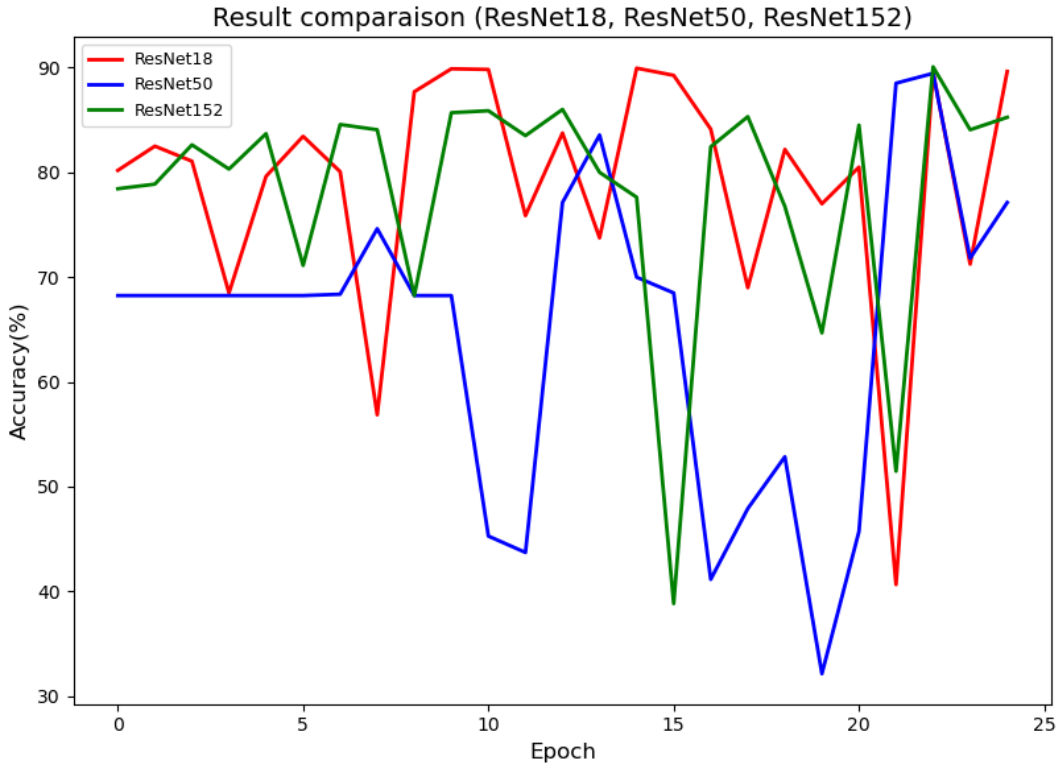


ResNet152

```
> Found 1599 images...
Accuracy of the ResNet152 on the valid images: 90.0562851782364 %
```



B. Comparison figures



From the comparison graph provided above, it is evident that the accuracy curve exhibits significant fluctuations. This phenomenon could potentially stem from suboptimal learning rate configuration. As the training progresses, it is advisable to gradually decrease the learning rate. I believe that the implementation of an lr_scheduler could be beneficial in addressing this situation. By incorporating an lr_scheduler, it is possible to optimize the learning rate schedule and potentially enhance the accuracy further.

5. Discussion

A. Augmentation

In this assignment, due to the limited availability of training data for certain categories, it is beneficial to enhance the dataset through various data augmentation techniques. These techniques include applying rotations, resizing, adjusting brightness, altering color temperature, and similar processes. Although these modifications remain recognizable to the human eye as the same images, they introduce novel variations to the machine, addressing the challenge of insufficient data volume. Data augmentation involves the modification and

transformation of existing images within the dataset to create additional examples, facilitating improved machine learning outcomes.

Furthermore, there are several augmentation strategies that can be employed, such as:

1. Data Normalization: Normalization can be applied either on a sample-wise basis (per batch) or feature-wise (across the entire dataset).

2. Whitening: ZCA Whitening, a technique that reduces data redundancy, can be employed.

3. Image Manipulation: Techniques like flipping, rotating, cropping, resizing, and shifting can be applied.

Below are various augmentation methods:

1. Cropping:

- Random Crop: transforms.RandomCrop
- Center Crop: transforms.CenterCrop
- Random Aspect Ratio Crop
- Five-Crop: transforms.FiveCrop
- Ten-Crop with Flipping: transforms.TenCrop

2. Flipping and Rotation:

- Random Horizontal Flip: transforms.RandomHorizontalFlip
- Random Vertical Flip: transforms.RandomVerticalFlip
- Random Rotation: transforms.RandomRotation

3. Image Transformations:

- Resize: transforms.Resize
- Standardization: transforms.Normalize
- Convert to Tensor: transforms.ToTensor
- Padding: transforms.Pad

- Adjust Brightness, Contrast, and Saturation: transforms.ColorJitter

- Convert to Grayscale: transforms.Grayscale

- Linear Transformation: transforms.LinearTransformation()

- Affine Transformation: transforms.RandomAffine

- Randomly Convert to Grayscale: transforms.RandomGrayscale

- Convert Data to PIL Image: transforms.ToPILImage

- Lambda Transformation: transforms.Lambda

4. Flexible Transformation Composition:

- Random Choice from a Set of Transforms: transforms.RandomChoice(transforms)

- Apply a Transform with Probability: transforms.RandomApply(transforms, p=0.5)

- Randomly Shuffle Transform Order: transforms.RandomOrder

By thoughtfully applying these augmentation techniques, the training data can be enriched and diversified, ultimately leading to the training of more robust and accurate models.

B. Batch size

1. In addition to the gradients themselves, the batch size and learning rate directly influence the weight updates of a model, making them crucial parameters for optimizing performance and convergence.

   A larger batch size improves memory utilization and enhances the parallelization efficiency of large matrix multiplications. As the batch size increases, the number of iterations required to complete one epoch decreases, leading to faster processing for the same amount of data. It also provides a more accurate direction for gradient descent, resulting in smaller training oscillations. However, achieving the same level of accuracy may require more epochs. Several recent studies have successfully trained models on the ImageNet dataset within one hour using large batch sizes. Stable gradient computation leads to smoother model training curves, and large batch sizes can yield favorable results during fine-tuning. Nevertheless, blindly

increasing the batch size might compromise the model's generalization ability, underscoring the importance of prudent batch size selection.

2. The relationship between learning rate and batch size is noteworthy. Typically, when we increase the batch size by a factor of N, to ensure equivalent weight updates, the learning rate should be scaled up by the same factor, following linear scaling rules. However, if maintaining the same weight variance is desired, the learning rate should be increased by a factor of sqrt(N). Both strategies have been studied, with the former being more commonly employed.

   The interplay between batch size and learning rate plays a critical role in shaping the training dynamics and convergence behavior of a neural network model. Careful consideration of these parameters is pivotal for achieving optimal training performance and generalization.