DLP Lab4 Report

▼ 310551178 資科工碩 穆冠蓁

Introduction

The purpose of this lab is to make us learn the concept of Resnet and build Residual block and bottleneck block. Also compare the performance between Resnet18 and Resent50. Beside, we need to customize a suitable Dataloader for our Diabetic Retinopathy Detection, which include some necessary data preprocessing steps.

Experiment Setup

1. The details of your model

   In the Deep Residual Learning for Image recognition, it has mentioned that while the layer is smaller than 50, we use Basic block as the residual block, while the layer is bigger than 50 we use Bottleneck block instead.

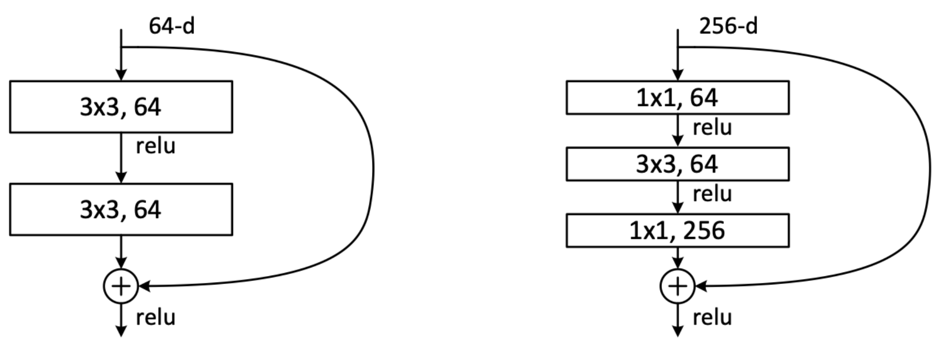


Figure 5. A deeper residual function $\mathcal{F}$ for ImageNet. Left: a building block (on 56×56 feature maps) as in Fig. 3 for ResNet-34. Right: a "bottleneck" building block for ResNet-50/101/152.

   a. Basic block & Bottleneck block

      The architecture of basic block and bottleneck block is reference from above Fig.5.

```
class BasicBlock(nn.Module):
    def __init__(self, in_channels, out_channels, downsample_stride):
        super(BasicBlock, self).__init__()
        if downsample_stride is None:
            self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
            self.downsample = None
        else:
            self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
            self.downsample = downsample(in_channels, out_channels, downsample_stride)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)

    def forward(self, x):
        ori = x
        out = self.bn1(self.conv1(x))
        out = self.relu(out)
        out = self.bn2(self.conv2(out))
        if self.downsample is not None:
            ori = self.downsample(ori)
        out = self.relu(out+ori)
        return out
```

```
class Bottleneck(nn.Module):
    def __init__(self, in_channels, mid_channels, out_channels, downsample_stride):
        super(Bottleneck, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, mid_channels, kernel_size=(1, 1), stride=(1, 1), bias=False)
        self.bn1 = nn.BatchNorm2d(mid_channels)
        if downsample_stride is None:
            self.conv2 = nn.Conv2d(mid_channels, mid_channels, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
            self.downsample = None
        else:
            self.conv2 = nn.Conv2d(mid_channels, mid_channels, kernel_size=(3, 3), stride=downsample_stride, padding=(1, 1), bias=F
            self.downsample = downsample(in_channels, out_channels, downsample_stride)
        self.bn2 = nn.BatchNorm2d(mid_channels)
        self.conv3 = nn.Conv2d(mid_channels, out_channels, kernel_size=(1, 1), stride=(1, 1), bias=False)
        self.bn3 = nn.BatchNorm2d(out_channels)
```

```
        self.relu = nn.ReLU(inplace=True)

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

b. Resnet18 & Resnet50

The architecture of every single block follows the following table.

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Table 1. Architectures for ImageNet. Building blocks are shown in brackets (see also Fig. 5), with the numbers of blocks stacked. Down-sampling is performed by conv3_1, conv4_1, and conv5_1 with a stride of 2.

```
class ResNet18(nn.Module):
    def __init__(self) -> None:
        super().__init__()
        self.conv1 = nn.Conv2d(3, 64, kernel_size=(7, 7), stride=(2,2), padding=(3,3), bias=False)
        self.bn1 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.layer1 = nn.Sequential(
            BasicBlock(64, 64, None),
            BasicBlock(64, 64, None))
        self.layer2 = nn.Sequential(
            BasicBlock(64, 128, (2,2)),
            BasicBlock(128, 128, None))
        self.layer3 = nn.Sequential(
            BasicBlock(128, 256, (2, 2)),
            BasicBlock(256, 256, None))
        self.layer4 = nn.Sequential(
            BasicBlock(256, 512, (2, 2)),
            BasicBlock(512, 512, None))
        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(512, 5)

    def forward(self, inputs):
        outputs = self.conv1(inputs)
        outputs = self.bn1(outputs)
        outputs = self.relu(outputs)
        outputs = self.maxpool(outputs)
        outputs = self.layer1(outputs)
        outputs = self.layer2(outputs)
        outputs = self.layer3(outputs)
        outputs = self.layer4(outputs)
        outputs = self.avgpool(outputs)
        outputs = self.fc(outputs.reshape(outputs.shape[0], -1))
        return outputs
```

```
class ResNet18(nn.Module):
    def __init__(self) -> None:
        super().__init__()
        self.conv1 = nn.Conv2d(3, 64, kernel_size=(7, 7), stride=(2,2), padding=(3,3), bias=False)
        self.bn1 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.layer1 = nn.Sequential(
            BasicBlock(64, 64, None),
            BasicBlock(64, 64, None))
        self.layer2 = nn.Sequential(
            BasicBlock(64, 128, (2,2)),
            BasicBlock(128, 128, None))
        self.layer3 = nn.Sequential(
            BasicBlock(128, 256, (2, 2)),
            BasicBlock(256, 256, None))
        self.layer4 = nn.Sequential(
            BasicBlock(256, 512, (2, 2)),
            BasicBlock(512, 512, None))
        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(512, 5)

    def forward(self, inputs):
        outputs = self.conv1(inputs)
        outputs = self.bn1(outputs)
```

```
        outputs = self.relu(outputs)
        outputs = self.maxpool(outputs)
        outputs = self.layer1(outputs)
        outputs = self.layer2(outputs)
        outputs = self.layer3(outputs)
        outputs = self.layer4(outputs)
        outputs = self.avgpool(outputs)
        outputs = self.fc(outputs.reshape(outputs.shape[0], -1))
        return outputs

# Resnet50
class ResNet50(nn.Module):
    def __init__(self):
        super(ResNet50, self).__init__()
        self.conv1 = nn.Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
        self.bn1 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.layer1 = nn.Sequential(
            Bottleneck(64, 64, 256, (1, 1)),
            Bottleneck(256, 64, 256, None),
            Bottleneck(256, 64, 256, None))
        self.layer2 = nn.Sequential(
            Bottleneck(256, 128, 512, (2, 2)),
            Bottleneck(512, 128, 512, None),
            Bottleneck(512, 128, 512, None),
            Bottleneck(512, 128, 512, None))
        self.layer3 = nn.Sequential(
            Bottleneck(512, 256, 1024, (2, 2)),
            Bottleneck(1024, 256, 1024, None),
            Bottleneck(1024, 256, 1024, None),
            Bottleneck(1024, 256, 1024, None),
            Bottleneck(1024, 256, 1024, None),
            Bottleneck(1024, 256, 1024, None))
        self.layer4 = nn.Sequential(
            Bottleneck(1024, 512, 2048, (2, 2)),
            Bottleneck(2048, 512, 2048, None),
            Bottleneck(2048, 512, 2048, None))
        self.avgpool = nn.AdaptiveAvgPool2d(output_size=(1, 1))
        self.fc = nn.Linear(2048, 5)

    def forward(self, x):
        outputs = self.relu(self.bn1(self.conv1(x)))
        outputs = self.maxpool(outputs)
        outputs = self.layer1(outputs)
        outputs = self.layer2(outputs)
        outputs = self.layer3(outputs)
        outputs = self.layer4(outputs)
        outputs = self.avgpool(outputs)
        outputs = self.fc(outputs.reshape(outputs.shape[0], -1))
        return outputs
```

c. train

```
def train(model_type, device, model, train_loader, test_loader, optimizer, criterion, epoch_num, ckpt_path):
    os.makedirs(ckpt_path, exist_ok=True)
    scheduler = StepLR(optimizer, step_size= 50, gamma=0.95)
    batch_count = 0
    epoch_pbar = tqdm(range(1, epoch_num+1))
    for epoch in epoch_pbar:
        model.to(device)
        model.train()
        epoch_loss = 0
        correct = 0.0
        total = 0.0
        avg_loss = 0.0
        batch_pbar = tqdm(train_loader)
        for i, (images, labels) in enumerate(batch_pbar):
            images = images.to(device)
            labels = labels.to(device)
            optimizer.zero_grad()
            outputs = model(images)
            loss = criterion(outputs, labels)
            avg_loss += loss.item()
            loss.backward()
            optimizer.step()
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
            batch_count += 1
            epoch_loss += loss.item()
            batch_pbar.set_description(f'[train] [epoch:{epoch:>4}/{epoch_num}] [batch: {i+1:>5}/{len(train_loader)}] loss: {loss.i
        scheduler.step()
        epoch_pbar.set_description(f'[train] [epoch:{epoch:>4}/{epoch_num}] loss: {epoch_loss/len(train_loader):.4f}')
        acc = 100 * correct / total
        avg_loss /= len(train_loader)
        print('Train accuracy : {:.2f} %, Train loss : {:.4f}'.format(acc, avg_loss))
        torch.save(model.state_dict(), f'{ckpt_path}{model_type}_epoch{epoch}.ckpt')
        evaluate(model_type, model, device, test_loader, criterion, epoch)
```

2. The details of your Dataloader

- `__init__` : initialize some parameter and also to transformation
- `__len__` : return the size of the dataset
- `__getitem__` : get the provided item data and label(in .pt format).

```
class RetinopathyLoader(data.Dataset):
    def __init__(self, root, mode):
```

```
        means = [0.485, 0.456, 0.406]
        stds = [0.229, 0.224, 0.225]
        self.root = root
        self.img_name, self.label = getData(mode)
        self.mode = mode
        self.size = [512, 512]
        self.new_path = "./data/new_test"
        self.transform = transforms.Compose([
            transforms.CenterCrop(2560),
            transforms.Resize(self.size),
            transforms.RandomHorizontalFlip(p=0.5),
            transforms.ToTensor(),
            transforms.Normalize(means, stds)
        ])
        print("> Found %d images..." % (len(self.img_name)))

    def __len__(self):
        """'return the size of dataset"""
        return len(self.img_name)

    def __getitem__(self, index):
        image_name = self.img_name[index] + '.pt'
        path = os.path.join(self.root, image_name)
        img = torch.load(path)
        label = self.label[index]
        return img, label

    def save_tensor(self, index):
        image_name = self.img_name[index] + '.jpeg'
        path = os.path.join(self.root, image_name)
        img = Image.open(path)
        img = self.transform(img)
        new_path = os.path.join(self.new_path, self.img_name[index] + '.pt')
        torch.save(img, new_path)
```

3. Describing your evaluation through the confusion matrix

   We load the model we trained to predict the testing image, and compare the predicted `outputs` with the ground truth `labels`, also compute the loss to update the parameters.

   After computing the loss and the accuracy, we put the predicted results and the ground truth into the `plt_confusion_matrix` to draw the confusion matrix.

```
def evaluate(model_type, model, device, test_loader, criterion, epoch):
    predict = []
    ground_truth = []
    model.eval()
    with torch.no_grad():
        correct = 0
        total = 0
        avg_loss = 0
        batch_pbar = tqdm(test_loader)
        for i, (images, labels) in enumerate(batch_pbar):
            images = images.to(device)
            labels = labels.to(device)
            outputs = model(images)
            loss = criterion(outputs, labels)
            avg_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            predict += predicted.cpu().numpy().tolist()
            ground_truth += labels.cpu().numpy().tolist()
            correct += (predicted == labels).sum().item()
            batch_pbar.set_description(f'[test] [batch: {i+1:>5}/{len(test_loader)}] loss: {loss.item():.4f}')
        avg_loss /= len(test_loader)
        acc = 100 * correct / total
        print('Test accuracy : {:.2f} %, Test loss : {:.4f}'.format(acc, avg_loss))
        plt_confusion_matrix(ground_truth, predict, title=model_type+'_'+str(epoch), normalize='true')
```

```
def plt_confusion_matrix(ground_truth, predicted, normalize='all', title='Confusion matrix', cmap=plt.cm.Blues):
    matrix = confusion_matrix(ground_truth, predicted, normalize=normalize)
    cm_display = metrics.ConfusionMatrixDisplay(confusion_matrix = matrix, display_labels = ['0', '1', '2', '3', '4'])
    cm_display.plot(cmap=cmap)
    plt.title(title)
    plt.savefig(f'./plot/{title}.png')
```

Data Preprocessing

I do centercrop at first and do resize and randomflip and do the normalization.

Due to the dataset is large, I have design a new member function called `save_tensor` to do transform, and save the result into torch tensor.And also customize the `__getitem__` not to read the jpeg data, instead of reading the augmentation and normalize tensor we save.

Experimental results

1. The highest testing accuracy

Experiment settings	Training accuracy	Testing accuracy

Experiment settings	Training accuracy	Testing accuracy
Resnet18 with pretrained	83.20	81.95
Resnet18 w/o pretrained	73.51	73.49
Resnet50 with pretrained	**88.79**	**83.25**
Resnet50 w/o pretrained	73.49	72.90

The best training accuracy and testing accuracy is the setting of Resnet50 without pretrained.

hyperparameter:

epoch: 10

batch size: 8

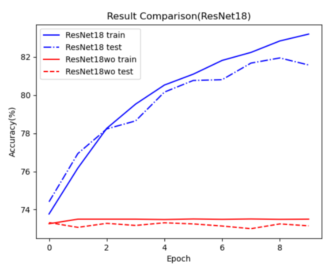
learning rate:0.001

loss function: cross entropy loss

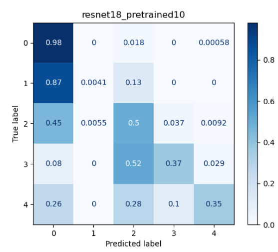
optimizer:sgd

2. Comparison figure

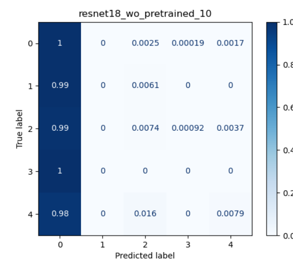
- Resnet 18



Resnet18 accuracy comparison

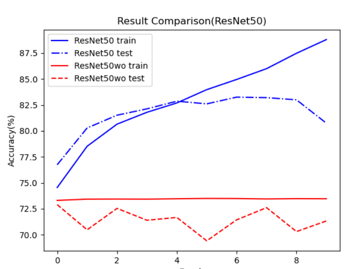


Resnet18 pretrained confusion matrix

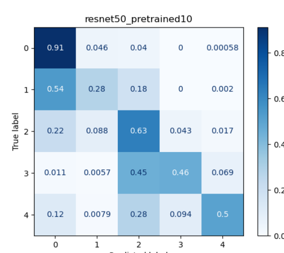


Resnet18 without pretrained confusion matrix

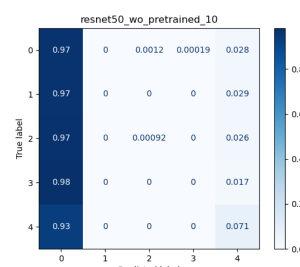
- Resnet 50



Resnet18 accuracy comparison



Resnet50 pretrained confusion matrix



Resnet50 without pretrained confusion matrix

Discussion

From the comparison figure above, we can see that there is a large gap in accuracy between training from scratch and from pretrained.It seems that the pretrained model has learned a general features from the original dataset, so it can outperforms the model which is train from scratch.

Reference

TRANSFORMING AND AUGMENTING IMAGES: https://pytorch.org/vision/main/transforms.html#transforming-and-augmenting-images