University of Macau

Faculty of Science and Technology

Department of Computer and Information Science

澳門大學

UNIVERSIDADE DE MACAU

UNIVERSITY OF MACAU

CISC3024 (2024-1 001)

Pattern Recognition

Project report

(Classification of Street View House Numbers Using PyTorch)

Mai Jiajun, D-C1-2785-3

(Group Partner: Huang Yanzhen, D-C1-2673-2)

05-NOV-2024

Contents

0 GitHub repo website

- Please be noticed that to reduce the excessive use of images and graphs, **some charts that we got in the experiments**(some Receiver Operating Characteristic (ROC) curves and Area Under the Curve (AUC) metrics) **will be omitted** in the following content for Experiment 2, 3, and 4.
- Please visit our GitHub repository at the following URL for **all curves and graphs**: https://github.com/hyz-courses/CISC3024-Pattern-Recognition.

  Jupiter one: https://github.com/hyz-courses/CISC3024-Pattern-Recognition/blob/master/Final_Project/main.ipynb

1 Data Processing and Augmentation

- From loading both "`train_32x32.mat`" and "`test_32x32.mat`", we have a total of **over 77,000** SVHN images with their labels as input and output, in each experiment we will **split the data from "`train_32x32.mat`" randomly into `train_data` and `valid_data` with the ratio of 8 : 2**.
- Each image has a size of **32 * 32 pixels and 3 channels**. Therefore, preprocessing the images and extracting features from them is a crucial step before training.

1.1 Transforms used on images

(1). `ContrastEnhanceTransform(factor1)`: This transform is a custom transform that adjusts the contrast of an image according to the parameter factor1, simulating images under different lighting conditions.

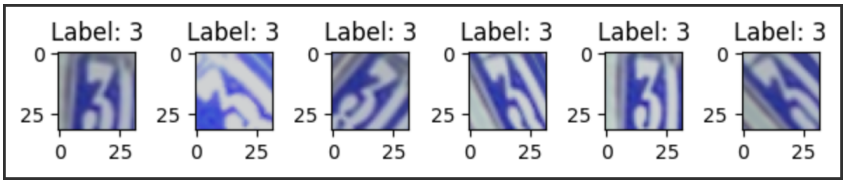

Figure 1.1.1: different factor1 on Contrast Transform

(2). `A.RandomResizedCrop` is used to randomly crop the image and then resize it to a 32x32 specification.

(3). `A.Rotate` randomly rotates the cropped image by a certain angle, either counterclockwise or clockwise.

(4). Finally, normalization and `ToTensorV2()` operations are performed to convert the image into a format that PyTorch can use as input for learning.

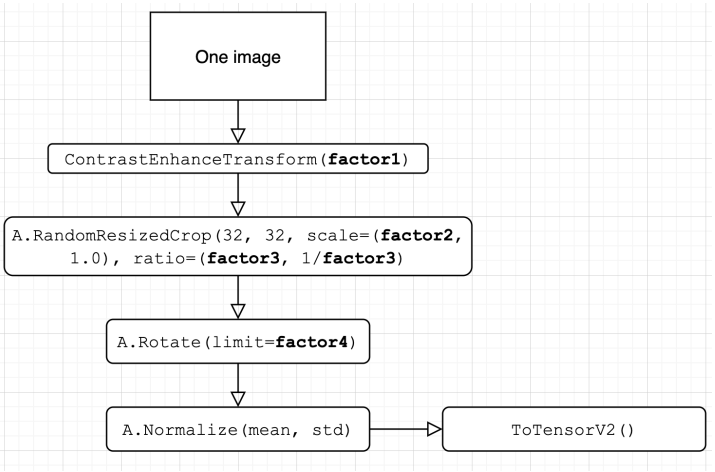

Figure 1.1.2: Determined transform steps after brainstorming.

1.2 Brief: ContrastEnhanceTransform(factor1)

The `ContrastEnhanceTransform` is a very simple custom transform that enhances **all pixels by multiplying them by a specific factor (factor1)** and then clipping the values between 0 and 255.

In subsequent experiments, we will experiment with different values of factor1.

```
4 个用法  麦家俊 1
class ContrastEnhanceTransform:
    麦家俊 1
    def __init__(self, factor: Union[float, Tuple[float, float]]) -> None:
        if isinstance(factor, tuple):
            self.factor_min = factor[0]
            self.factor_max = factor[1]
        else:
            self.factor_min = 1 / factor  # TODO
            self.factor_max = factor

    Yanzhen Huang 1
    def __call__(self, img: np.ndarray) -> np.ndarray:
        _dtype = img.dtype
        contrast_factor = random.uniform(self.factor_min, self.factor_max)
        img = np.clip(img * contrast_factor, a_min: 0, a_max: 255)  # apply contrast enhancement
        return img.astype(_dtype)
```

Figure 1.2.1: the logic of ContrastEnhanceTransform

1.3 Brief: A.Normalize(mean, std) and get_meanstd()

Due to the contrast adjustment, the entire training set of images may have different mean and standard deviation (std) compared to before the contrast adjustment. In the class SVHNDataset, we have customized a method called **get_meanstd to conveniently obtain the current mean and std** of the dataset.

```python
麦家俊 1
def get_meanstd(self, contrast_factor=None):
    if contrast_factor is not None:
        random_cf = random.uniform(1 / contrast_factor, contrast_factor)
        images_ = []
        for i in range(len(self.images)):
            image = self.images[i]
            image = np.clip(image.astype(np.float32) * random_cf, a_min: 0.0, a_max: 255.0)
            images_.append(image.astype(np.uint8))
        images_ = np.array(images_)
    else:
        images_ = self.images

    images_ = images_.astype(np.float32) / 255.0

    images_ = np.transpose(images_, axes: (3, 0, 1, 2))
    mean = [np.mean(x) for x in images_]
    std = [np.std(x, ddof=0) for x in images_]

    return mean, std
```

Figure 1.3.1: get_meanstd in class SVHNDataset(Dataset)

1.4 Experiments on hyperparameters (factors)

We will use PyTorch's Neural Network to train our classification model and conduct **4 extensive hyperparameter experiments**. These experiments will focus on the contrast, crop area ratio, crop aspect ratio, random rotation ratio, and various hyperparameters of the neural network.

We will also use **grid search** to try to find the best value combinations for these parameters.

2 Neural Network Setup

- In this project, **we will design a simplified VGG model, named SmallVGG**, which will replicate the convolutional and fully connected layers of the traditional VGG architecture.
- In Experiment 4-1, we will also explore neural networks beyond VGG to satisfy our curiosity about different architectures and conduct analyses.

2.1 class SmallVGG(nn.Module):

```
Yanzhen Huang 1
class SmallVGG(nn.Module):
    Yanzhen Huang 1
    def __init__(self, frame_size=32):
        super(SmallVGG, self).__init__()
        self.frame_size = frame_size
        self.conv_layers = nn.Sequential(
            nn.Conv2d( in_channels: 3,  out_channels: 8, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.Conv2d( in_channels: 8,  out_channels: 16, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),  # 16x16

            nn.Conv2d( in_channels: 16,  out_channels: 32, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.Conv2d( in_channels: 32,  out_channels: 32, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),  # 8x8

            nn.Conv2d( in_channels: 32,  out_channels: 32, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.Conv2d( in_channels: 32,  out_channels: 32, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),  # 4x4
        )

        self.fc_layers = nn.Sequential(
            nn.Linear(frame_size * 4 * 4,  out_features: 256),
            nn.ReLU(),
            nn.Linear( in_features: 256,  out_features: 10)
        )

    Yanzhen Huang
    def forward(self, x):
        x = self.conv_layers(x)
        x = x.view(x.size(0), -1)
        x = self.fc_layers(x)
        return x
```

Figure 2.1.1: based model: SmallVGG we used in the project

This neural network undergoes **3 times convolution and pooling** operations, and then flattens the multi-dimensional data into one-dimensional data before entering the fully connected layer.

3 Coding Training and Evaluation Functions

- In this section, we will **experiment with different parameter combinations** and analyze the comparative results. We will use a simple grid search to identify the parameter combinations worth retaining.
- Try to perform validation after each experiment and compute the **Receiver Operating Characteristic (ROC) curves and Area Under the Curve (AUC) metrics**.

3.1 Hyperparameters

We designed four sets of experiments targeting different hyperparameters:

1. Experiment Set 1 (**Optimizer**): Attempting to find an optimal optimizer.
2. Experiment Set 2 (**Epoch, Learning Rate, Batch Size**): Aiming to find suitable values for the number of epochs and learning rate, and to determine the appropriate batch size for the project.
3. Experiment Set 3 (**Transform** Hyperparameters): Appropriate image transformations can enhance the model's fitting ability, but excessive transformations can lead to underfitting.
4. Experiment Set 4 (Additional Experiments): Using models other than VGG, including LeNet-5, AlexNet, and GoogLeNet.

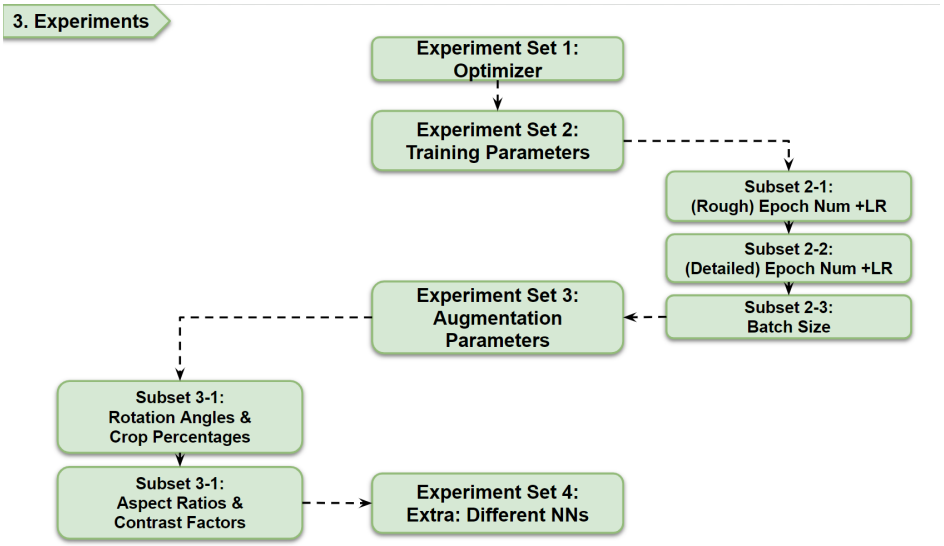

Figure 3.1.1: Experiments road map below

3.2 Experiment 1(**Optimizer**):

We identified **6 different optimizers** suitable for image processing (as shown in the figure below) and collected their final results, which are represented in the **train-validation loss graph**, **Confusion Matrix**, **Precision-Recall Curve**, and **ROC-AUC Curve**.

```
exp1_models = [SmallVGG().to(device) for _ in range(0,6)]

candidate_optimizers = [
    optim.Adam(exp1_models[0].parameters(), lr=exp1_hyperparams['lr']),
    optim.SGD(exp1_models[1].parameters(), lr=exp1_hyperparams['lr'], momentum=0.9),
    optim.RMSprop(exp1_models[2].parameters(), lr=exp1_hyperparams['lr']),
    optim.AdamW(exp1_models[3].parameters(), lr=exp1_hyperparams['lr'], weight_decay=0.01),
    optim.Adagrad(exp1_models[4].parameters(), lr=exp1_hyperparams['lr']),
    optim.SGD(exp1_models[5].parameters(), lr=exp1_hyperparams['lr'], momentum=0.9, nesterov=True)]

for model in exp1_models:
    print(id(model), end=", ")
```

```
1825706848512, 1825708732080, 1825706883920, 1825708226832, 1826082552032, 1826082550592,
```

Figure 3.2.0: All candidate optimizers

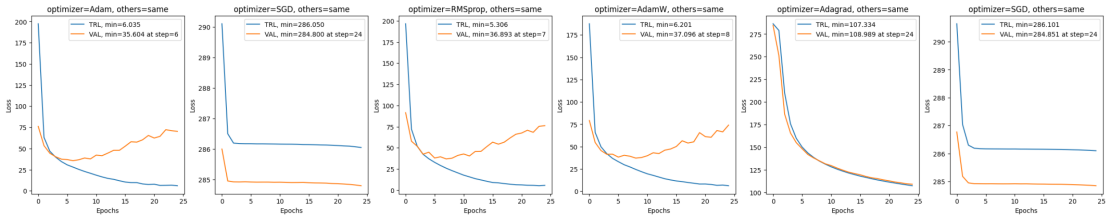

Figure 3.2.1: Train loss – valid loss graph of different optimizers

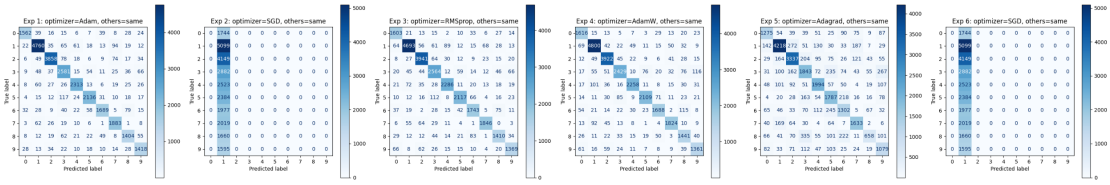

Figure 3.2.2: Confusion Matrix of different optimizers

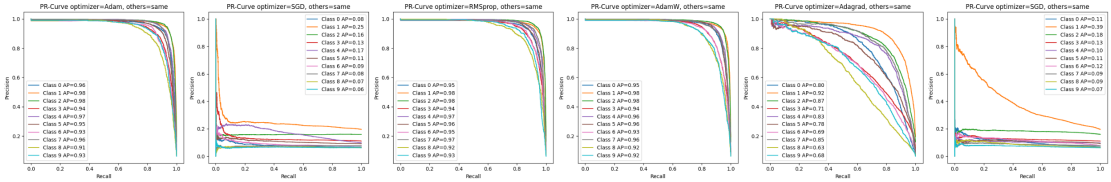

Figure 3.2.3: Precision-Recall Curve of different optimizers

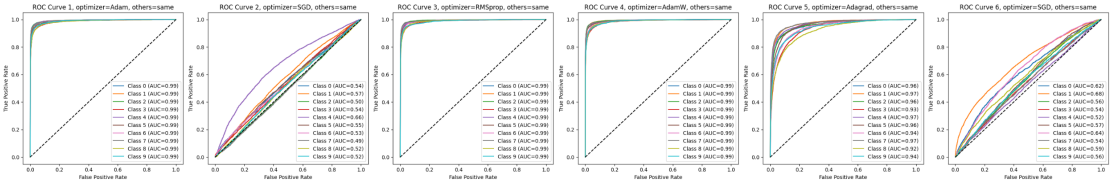

Figure 3.2.4: ROC-AUC Curve (Micro) of different optimizers

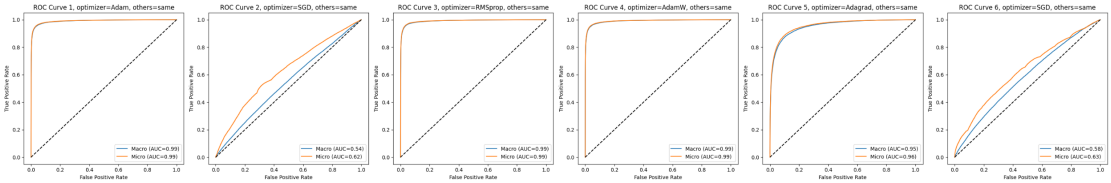

Figure 3.2.5: ROC-AUC Curve (Macro) of different optimizers

Result:

1. Optimizer "**Adam**" has the highest **avg F1 score: 0.899**.
2. We can find that all optimizers are not good at classify number '8' correctly.

3.3 Experiment 2-1(**Epoch, Learning Rate**):

To enable the neural network to fit faster without fluctuations in loss, we prioritize performing a **grid search on both the epoch and learning rate** together.

```
candidate_epochs = [10, 15, 20, 25]
candidate_lr = [1e-3, 1e-4, 1e-5, 1e-6]
```

Figure 3.3.0: candidates of epochs and lr

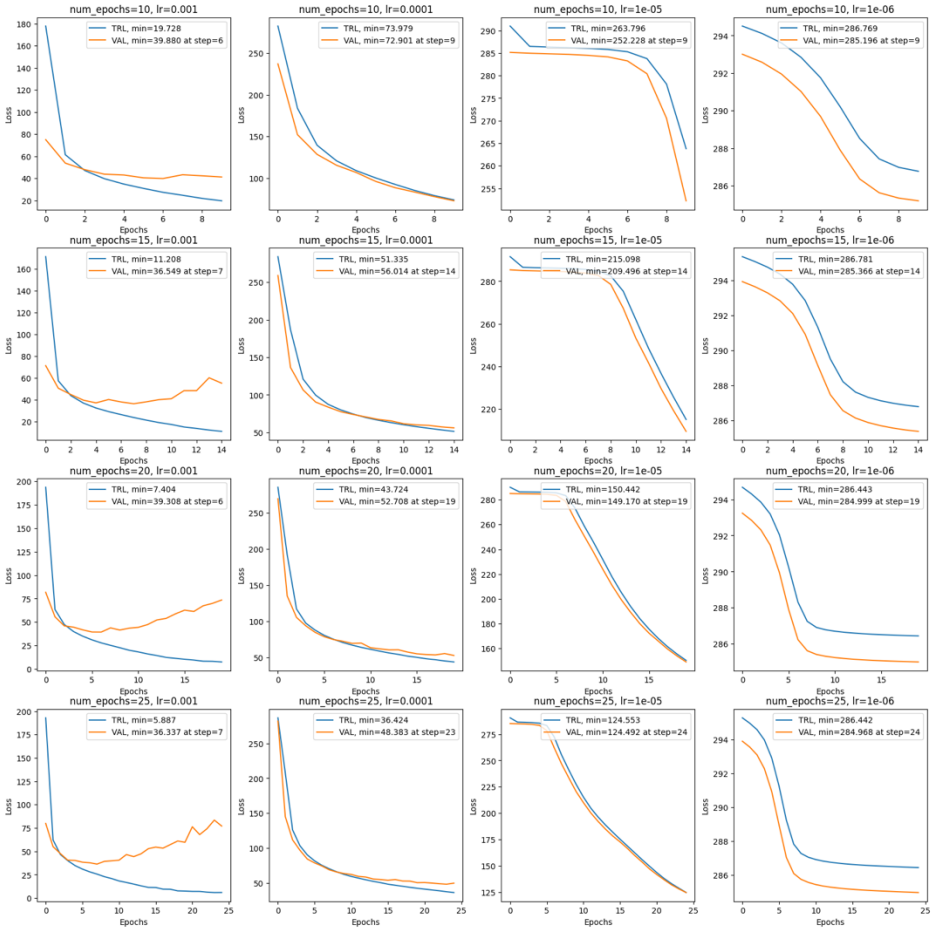

Figure 3.3.1: Train loss – valid loss graph of different epochs & lr

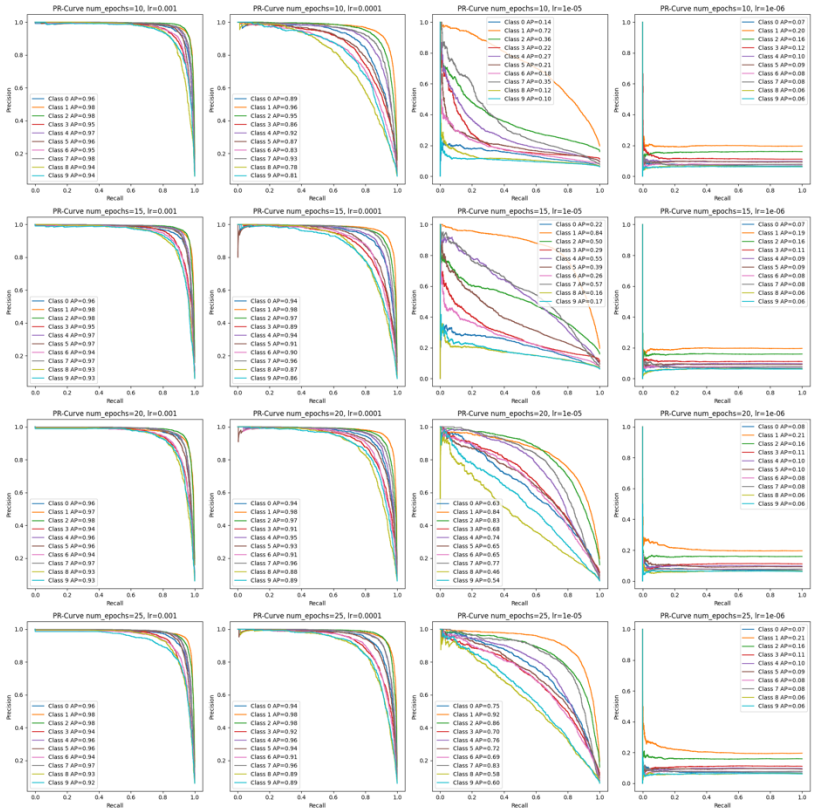

Figure 3.3.2: Precision-Recall Curve of different epochs and lr

Result:

1. Combination "**Epoch = 15**" and "**lr = 1e-3**" has the highest **avg F1 score: 0.909**.
2. However, the issue is that **we cannot confirm that this combination is optimal** because it is located at the upper right corner of the grid search. Therefore, we plan to further investigate the optimal value of the learning rate.

3.4 Experiment 2-2(Deeper on **Learning Rate**):

We will experiment with determining the learning rate in a proportional range from **5e-3 to 1e-4**, while keeping the epoch fixed at 15.

```
# More detailed candidate learning rates around 1e-3, that is 10e-4.
exp2_2_candidate_lr = np.geomspace(1e-4, 5e-3, 8)
print(exp2_2_candidate_lr)
```

```
[0.0001     0.00017487 0.00030579 0.00053472 0.00093506 0.00163512
 0.0028593  0.005     ]
```

Figure 3.4.1: Secondary candidates of learning rates (from 5e-3 to 1e-4)

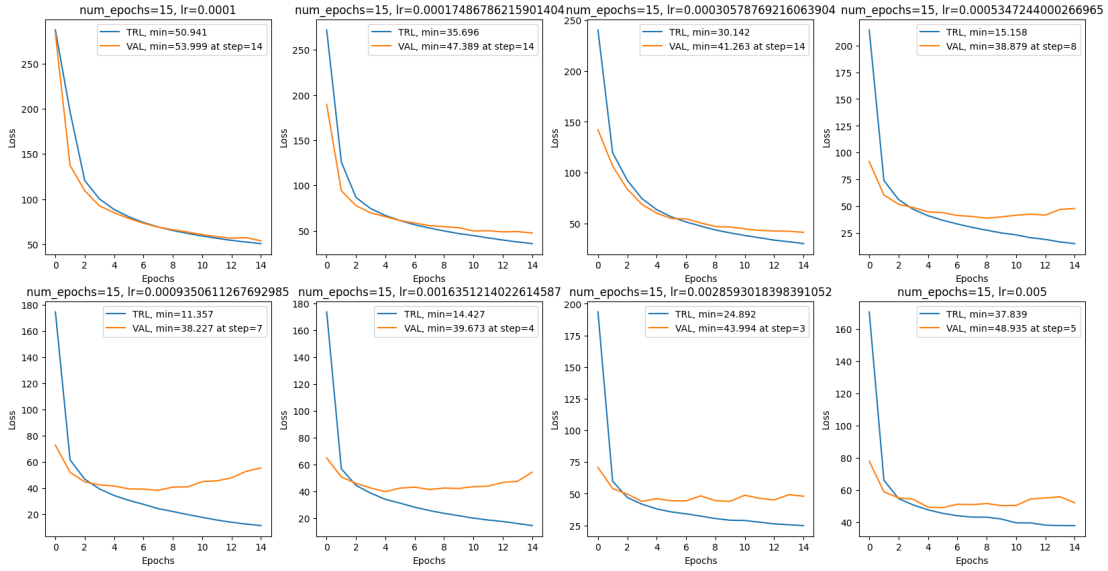

Figure 3.4.2: Train loss – valid loss graph of detailed learning rates

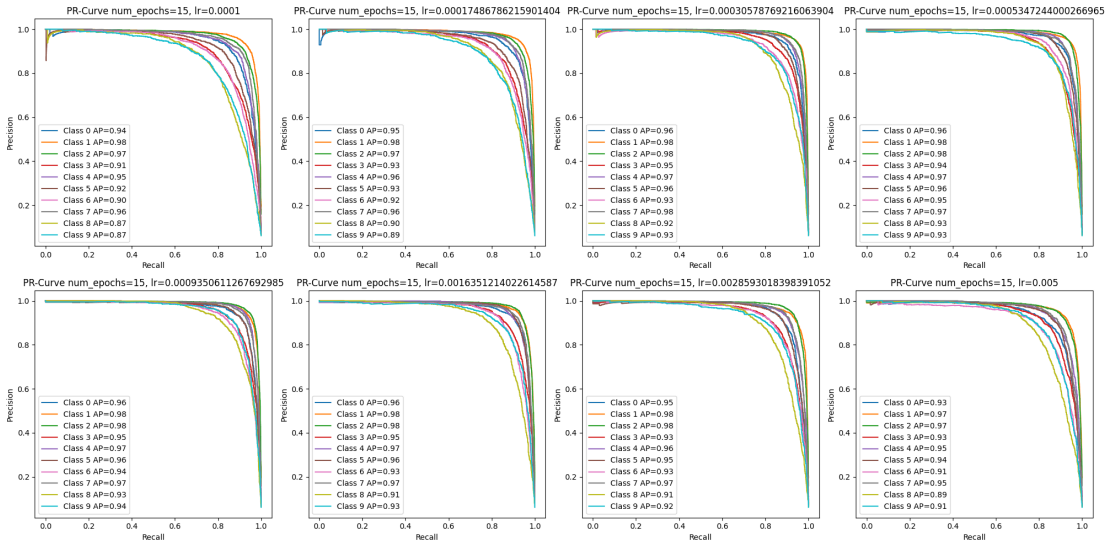

Figure 3.4.3: Precision-Recall Curve of detailed learning rates

Result:

1. It is obvious that "lr = 0.000935" performs way better than others, it has the highest **avg F1 score: 0.905** in the Precision-Recall check.
2. So, we decided to choose "**lr = 0.001**". We found the fact that **when the learning rate is reduced by an order of magnitude, the number of epochs required for the model to fit will increase by approximately an order of magnitude**, too.

3.5 Experiment 2-3(**Batch_size**):

```python
exp2_3_hyperparams = {
    "num_epoch": 100,
    "lr": 1e-3,
    "criterion": nn.CrossEntropyLoss(),
    "optimizer": optim.Adam,
}

candidate_batch_sizes = [16, 32, 64, 128]
```

Figure 3.5.1: hyper Param and candidates of batch size.

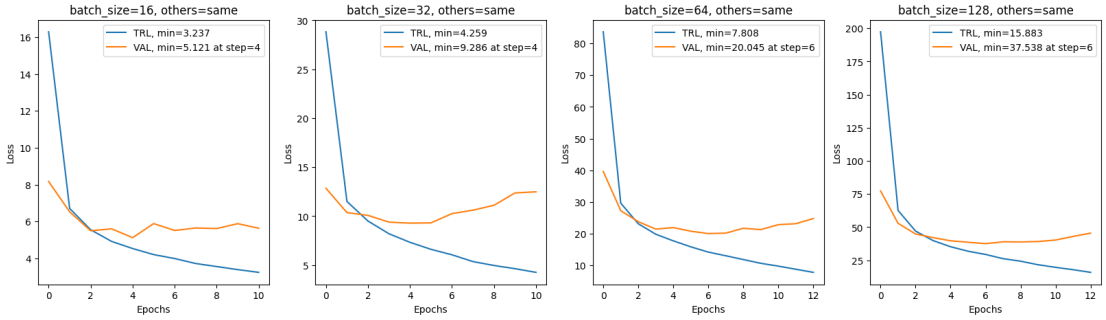

Figure 3.5.2: Train loss – valid loss graph of 4 kinds of batch size

```
Accuracies:
0.911 0.914 0.911 0.911

F1 Score Lists:
0.907 0.936 0.938 0.876 0.931 0.913 0.885 0.916 0.857 0.876 | Avg F1=0.904, Std F1=0.02698171524438727
0.914 0.937 0.939 0.891 0.924 0.924 0.897 0.913 0.871 0.862 | Avg F1=0.907, Std F1=0.024938019187106995
0.906 0.937 0.944 0.882 0.923 0.917 0.879 0.923 0.858 0.849 | Avg F1=0.902, Std F1=0.031413271954473965
0.913 0.942 0.943 0.868 0.929 0.911 0.895 0.922 0.852 0.850 | Avg F1=0.902, Std F1=0.03326424565964047
Best: 2-th
```

Figure 3.5.3: Accuracies and F1 scores for 4 kinds of batch size

Result:

1. This time, 4 different of batch size **didn't pull ahead**. In later tests, their scores were also very close in each aspect.
2. After discussion, we gave the reason that it was the large amount of input data and the optimization of the GPU in computer that led the batch size less active in all result.
3. We choose "**batch_size = 128**" as it can speed up a little bit in the experiments later.

3.6 Experiment 3-1(**Rotation Angles and Crop Percentages**):

There are many ways to deform an image. Among the 4 deformation parameters we predefined(rotate angle, crop percentage, crop ratio, and contrast factor), we decided to first

use the combination of (Rotation Angles and Crop Percentages) as the target for the first **grid search** and completed **4x4 sub-experiments**.

```
# Group 1
candidate_angles = [15, 30, 45, 60]
candidate_crops = [0.08, 0.24, 0.40, 0.60] # Left Boundary
```

```
def run_exp3_1(angles, crops, hyper_params, train_dataset, valid_loader):
    combinations = list(itertools.product(angles, crops))
    experiments = []
    for i, combo in enumerate(combinations):
        angle, crop = combo

        print(f"Running Exp {i+1}: angles={angle}, crop={crop}")
        this_model = SmallVGG().to(device)
        num_epochs = hyper_params['num_epochs']
        lr = hyper_params['lr']
        criterion = hyper_params['criterion']
        optimizer = hyper_params['optimizer'](this_model.parameters(), lr=lr)

        # Define Transform
        this_transform = A.Compose([
            A.RandomResizedCrop(32, 32, scale=(crop, 1.0)),
            A.Rotate(limit=angle),
            A.Normalize(mean=exp3_1_mean, std=exp3_1_std),
            ToTensorV2()
        ])

        # Generate Dataset
        print(f"Exp {i+1}: Generating dataset from transform")
        train_dataset.transform = this_transform

        train_loader = DataLoader(train_dataset, batch_size=128, shuffle=True)

        # Train Model
        train_losses, valid_losses = train_and_evaluate(this_model,
                                                        train_loader,
                                                        valid_loader,
                                                        criterion,
                                                        optimizer,
                                                        num_epochs,
                                                        stop_early_params={
                                                            "min_delta": 0.01,
                                                            "patience": 5
                                                        })
```

Figure 3.6.2: partial code of experiment 3-1

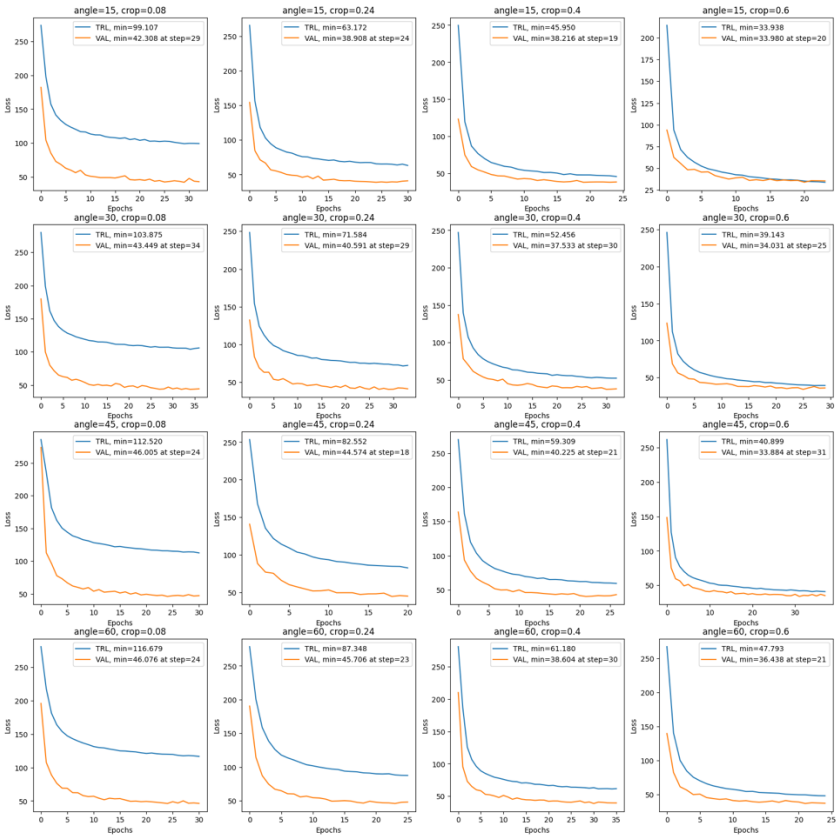

Figure 3.6.3: Train loss – valid loss graph of 4x4 combination of angle & crop

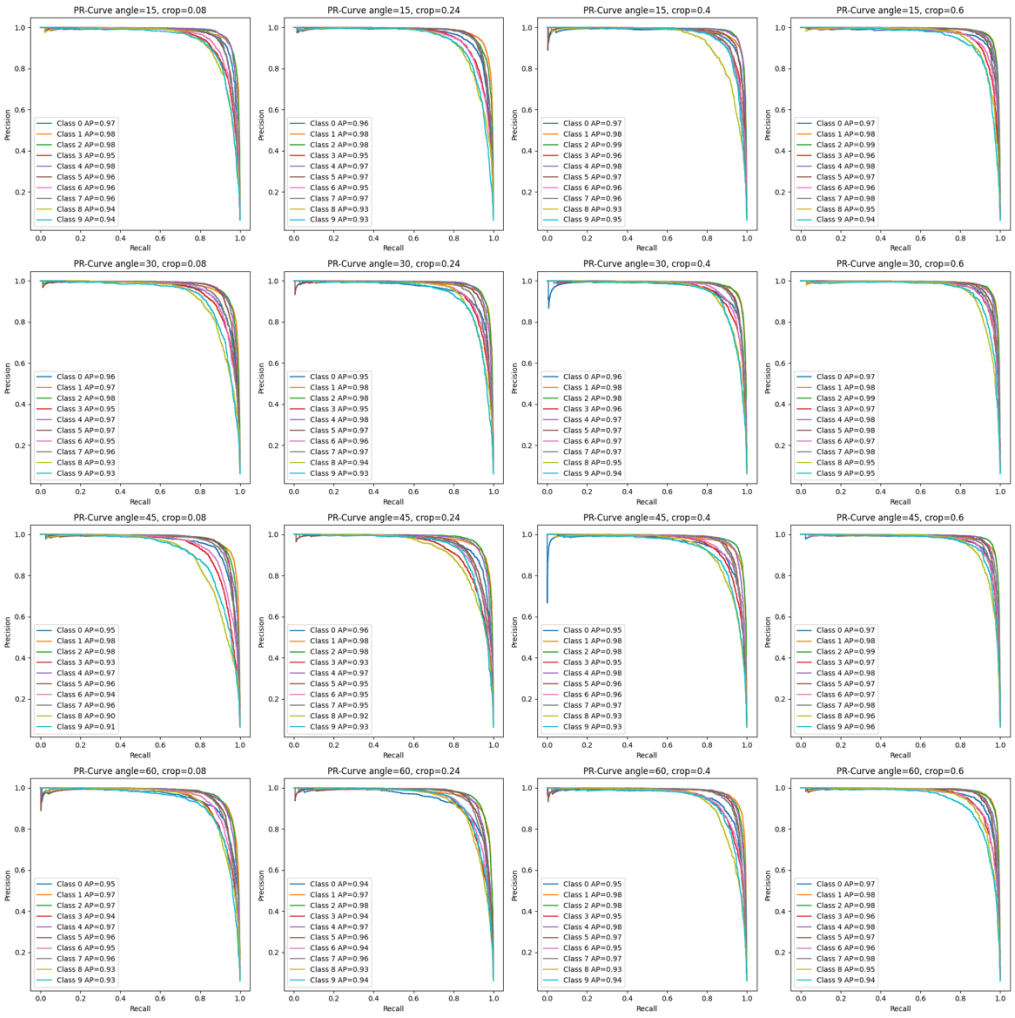

Figure 3.6.4: Precision-Recall Curve of angle & crop

```
Accuracies:
0.909 0.902 0.915 0.926 0.901 0.908 0.920 0.927 0.894 0.889 0.906 0.929 0.894 0.892 0.909 0.918

F1 Score Lists:
0.918 0.936 0.937 0.867 0.934 0.907 0.888 0.901 0.858 0.873 | Avg F1=0.902, Std F1=0.028146078028585732
0.902 0.930 0.923 0.860 0.913 0.913 0.885 0.911 0.857 0.863 | Avg F1=0.896, Std F1=0.02593225413077912
0.913 0.933 0.949 0.905 0.929 0.924 0.878 0.897 0.837 0.891 | Avg F1=0.906, Std F1=0.030528486926295145
0.918 0.951 0.948 0.904 0.937 0.925 0.908 0.938 0.885 0.874 | Avg F1=0.919, Std F1=0.024707654109350188
0.875 0.927 0.926 0.881 0.923 0.910 0.886 0.893 0.842 0.858 | Avg F1=0.892, Std F1=0.027922326916596706
0.893 0.935 0.927 0.879 0.926 0.921 0.888 0.905 0.873 0.863 | Avg F1=0.901, Std F1=0.024139022374309215
0.903 0.948 0.940 0.891 0.938 0.927 0.904 0.918 0.870 0.880 | Avg F1=0.912, Std F1=0.025248490682396124
0.921 0.947 0.947 0.909 0.944 0.927 0.918 0.924 0.885 0.883 | Avg F1=0.920, Std F1=0.02202325661064801
0.895 0.935 0.917 0.862 0.918 0.904 0.878 0.905 0.811 0.803 | Avg F1=0.883, Std F1=0.04265224570562562
0.899 0.927 0.921 0.837 0.911 0.885 0.877 0.883 0.839 0.830 | Avg F1=0.881, Std F1=0.033704954984753885
0.875 0.929 0.944 0.879 0.935 0.910 0.882 0.914 0.836 0.864 | Avg F1=0.897, Std F1=0.03311069193894167
0.903 0.946 0.956 0.912 0.944 0.936 0.912 0.925 0.891 0.897 | Avg F1=0.922, Std F1=0.021479968814335 38
0.881 0.923 0.919 0.850 0.918 0.893 0.874 0.889 0.863 0.856 | Avg F1=0.887, Std F1=0.025284746933263623
0.867 0.925 0.922 0.848 0.915 0.898 0.865 0.885 0.849 0.861 | Avg F1=0.884, Std F1=0.028327484028501 03
0.889 0.939 0.933 0.880 0.929 0.914 0.884 0.915 0.860 0.866 | Avg F1=0.901, Std F1=0.027214294349668898
0.910 0.946 0.946 0.891 0.919 0.919 0.894 0.931 0.884 0.860 | Avg F1=0.910, Std F1=0.026501694953201833
Best: 12-th
```

Figure 3.6.5: Accuracies and F1 scores for 16 sub-experiments of angle & crop

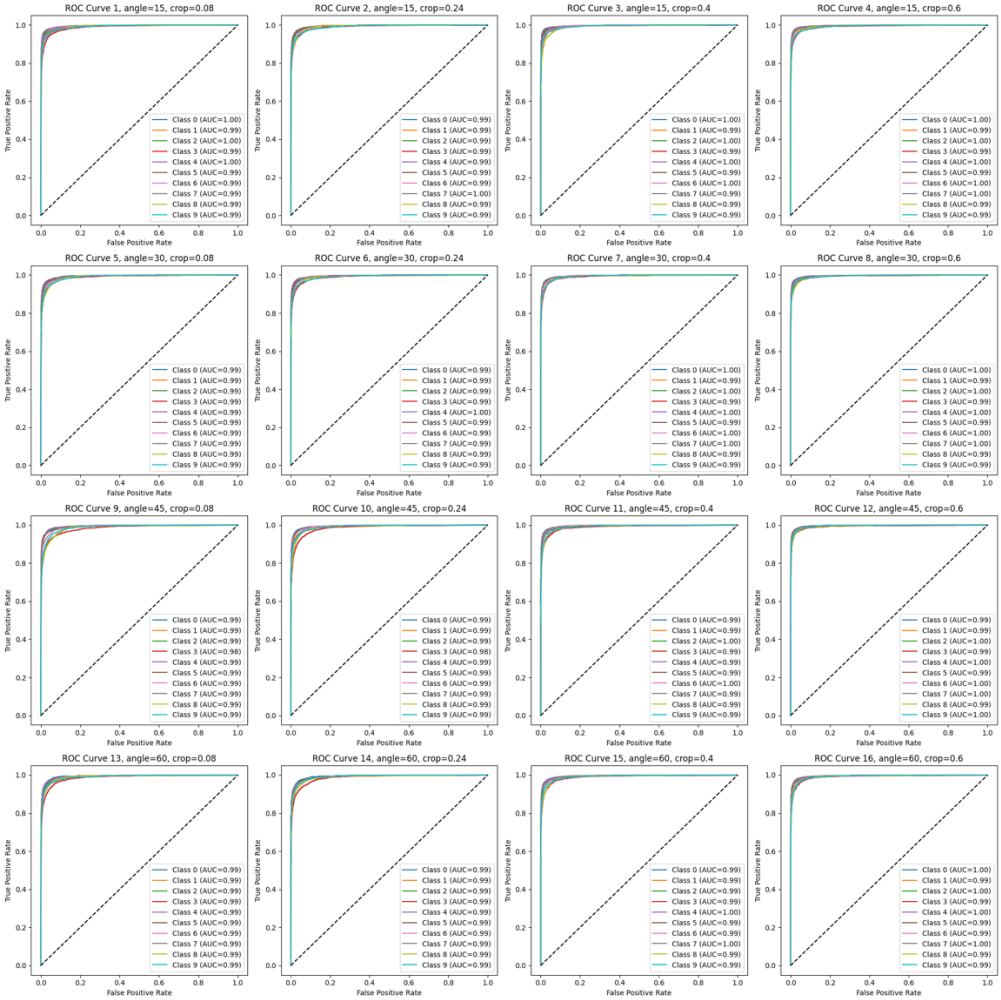

Figure 3.6.6: ROC Curve (Micro) of different combination of angles and crops

Result:

The best result obtained from this experiment is the combination (**angle = 45**, **crop = 0.8**), which means that each of our images will be cropped to 0.8 to 1.0 times their original size and then randomly rotated between -45 to 45 degrees. **Both random processes follow a uniform distribution**.

3.7 Experiment 3-2(**Aspect Ratios & Contrast Factors**):

For the second set of experiments on transform, we combined **aspect ratio and contrast factor**, as these 2 hyperparameters have a stronger transformation and generalization capability for images compared to the 3-1 experiment. We used grid search to test and analyze 4x4 combinations separately.

- **Aspect Ratios**: **Range of aspect ratios of the random crop**. For example, (0.75, 1.3333) allows crop aspect ratios from 3:4 to 4:3. Default: (0.75, 1.3333333333333333)
- Since we first performed contrast adjustment in our experiment, we **need to use the new train mean and train std** generated by the corresponding factor for normalization after the ContrastEnhanceTransform. The steps to obtain the train mean and train std are done before the transformation, which means it is **an offline preprocessing step**.

```
# Group 2
candidate_ratios = [0.25, 0.42, 0.58, 0.75]
candidate_contrast_factors = [1.2, 1.4, 1.6, 1.8]
```

Control candidates for different variables

```
def run_exp3_2(ratios, contrast_factors, hyper_params, train_dataset, valid_dataset):
    combinations = list(itertools.product(ratios, contrast_factors))
    experiments = []
    for i, combo in enumerate(combinations):
        ratio, cf = combo

        print(f"Running Exp {i+1}: ratio={ratio}, contrast_factor={cf}")
        this_model = SmallVGG().to(device)
        num_epochs = hyper_params['num_epochs']
        lr = hyper_params['lr']
        criterion = hyper_params['criterion']
        optimizer = hyper_params['optimizer'](this_model.parameters(), lr=lr)

        # Define Transform
        this_mean, this_std = train_dataset.get_meanstd(contrast_factor=cf)
        this_train_transform = A.Compose([
            A.Lambda(image=lambda img, **kwargs: ContrastEnhanceTransform(cf)(img)),  # Lambda customized transform block
            A.RandomResizedCrop(32, 32, scale=(hyper_params['crop'], 1.0), ratio=(ratio, 1.0 / ratio)),
            A.Rotate(limit=hyper_params['angle']),
            A.Normalize(mean=this_mean, std=this_std),
            ToTensorV2()
        ])

        this_valid_transform = A.Compose([
            A.Normalize(mean=this_mean, std=this_std),
            ToTensorV2()
        ])
```

Figure 3.7.1: partial code of experiment 3-2.

offline preprocessing step is under the line "`# Define Transform`"

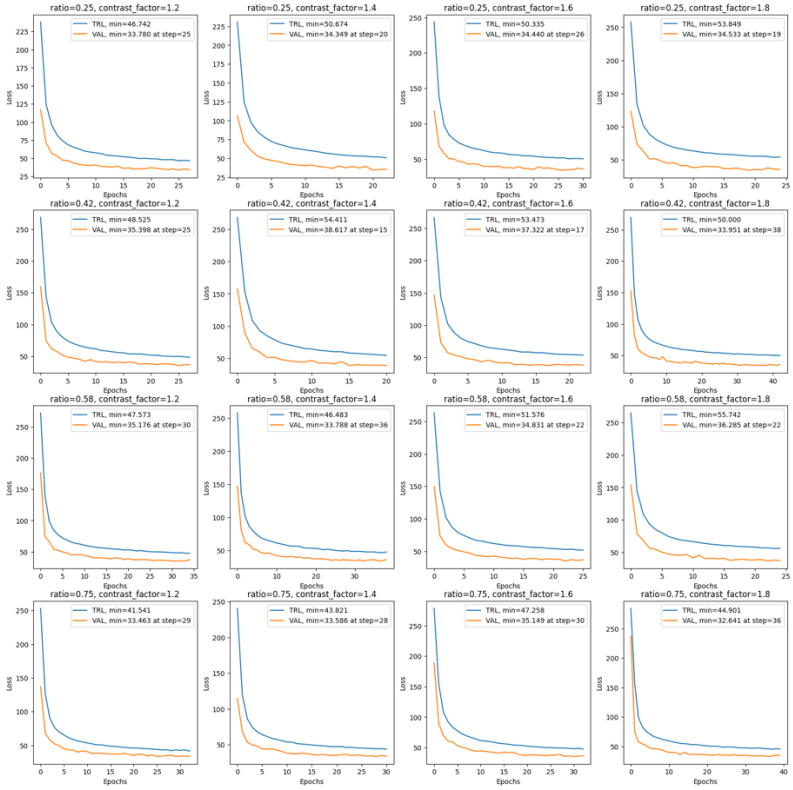

Figure 3.7.2: Train loss – valid loss graph of 4x4 combination of aspect ratios & contrast factors

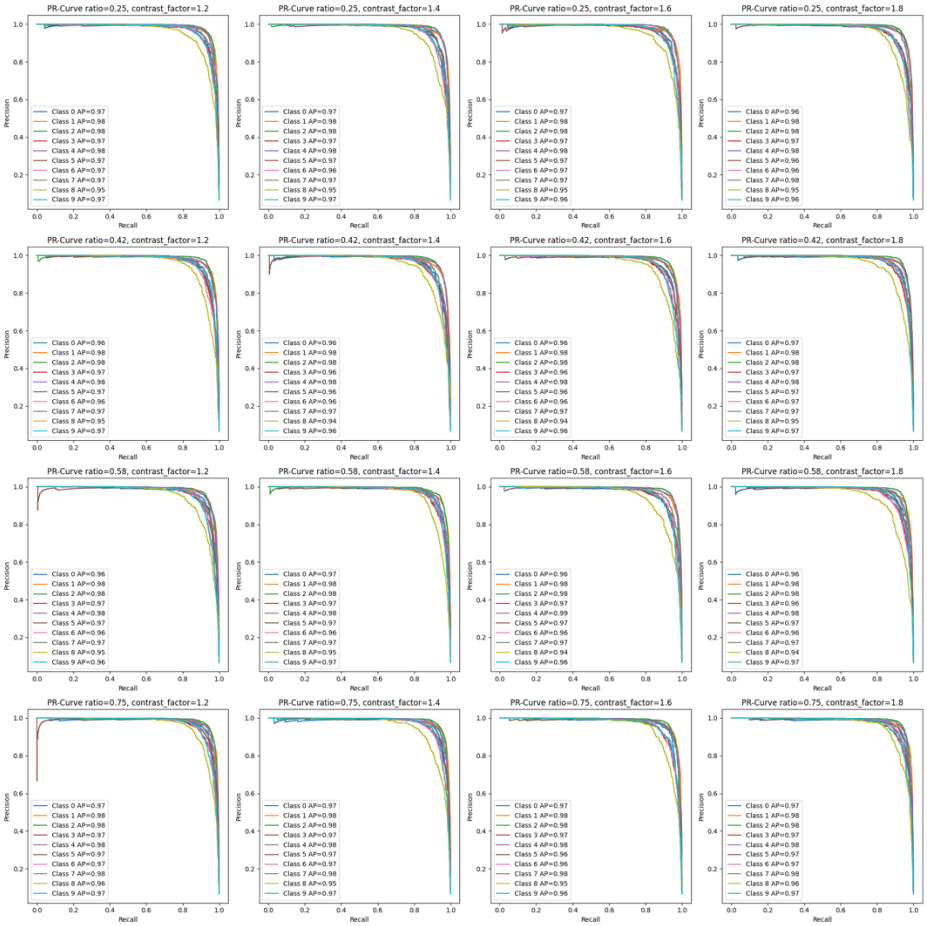

Figure 3.7.3: Precision-Recall Curve of aspect ratios & contrast factors

```
Accuracies:
0.920 0.918 0.917 0.917 0.918 0.911 0.913 0.922 0.914 0.916 0.913 0.913 0.923 0.923 0.917 0.922

F1 Score Lists:
0.894 0.937 0.947 0.909 0.939 0.912 0.896 0.926 0.880 0.912 | Avg F1=0.915, Std F1=0.02067976693819648
0.908 0.934 0.939 0.906 0.934 0.915 0.901 0.925 0.869 0.904 | Avg F1=0.913, Std F1=0.019974054656940087
0.925 0.939 0.929 0.904 0.940 0.916 0.908 0.916 0.852 0.902 | Avg F1=0.913, Std F1=0.02390683851734753
0.890 0.934 0.946 0.898 0.935 0.904 0.907 0.935 0.883 0.894 | Avg F1=0.912, Std F1=0.021508105940274098
0.912 0.936 0.946 0.912 0.930 0.908 0.887 0.922 0.873 0.910 | Avg F1=0.914, Std F1=0.0205745618350115
0.904 0.927 0.939 0.890 0.937 0.900 0.897 0.904 0.866 0.900 | Avg F1=0.907, Std F1=0.02124979354773411
0.912 0.932 0.936 0.888 0.930 0.904 0.889 0.926 0.872 0.898 | Avg F1=0.909, Std F1=0.02080427179136081
0.923 0.937 0.947 0.907 0.930 0.922 0.898 0.922 0.883 0.912 | Avg F1=0.918, Std F1=0.017568940244675372
0.904 0.933 0.936 0.904 0.932 0.908 0.906 0.905 0.872 0.890 | Avg F1=0.909, Std F1=0.019079743671399765
0.919 0.932 0.938 0.901 0.925 0.913 0.903 0.924 0.860 0.908 | Avg F1=0.912, Std F1=0.021048150026752813
0.907 0.933 0.946 0.899 0.938 0.890 0.899 0.920 0.862 0.886 | Avg F1=0.908, Std F1=0.024824526478269996
0.912 0.931 0.944 0.901 0.924 0.894 0.882 0.927 0.864 0.905 | Avg F1=0.908, Std F1=0.0230442781174334
0.920 0.939 0.945 0.910 0.921 0.915 0.906 0.928 0.889 0.912 | Avg F1=0.919, Std F1=0.015365852898010962
0.929 0.942 0.945 0.912 0.928 0.908 0.907 0.936 0.865 0.911 | Avg F1=0.918, Std F1=0.022282154155352875
0.925 0.933 0.943 0.895 0.935 0.903 0.901 0.924 0.871 0.905 | Avg F1=0.914, Std F1=0.021094009578053304
0.927 0.940 0.944 0.896 0.936 0.912 0.920 0.924 0.879 0.910 | Avg F1=0.919, Std F1=0.01913420145325484
Best: 16-th
```

Figure 3.7.4: Accuracies and F1 scores for 16 sub-experiments of aspect ratios & contrast factors

Result:

1. The experiment yielded unexpectedly but reasonably results (**aspect ratios = 0.75 & contrast factors = 1.8**).
2. Specifically, this means that the values of **all pixels in the images were randomly multiplied by 0.5556 to 1.8** (contrast transform)and clipped within the integer range of [0, 255], while the cropped aspect ratio was between (3/4, 4/3).
3. We believe that a high contrast factor allows the model to **better recognize data under different lighting conditions**, leading to a more fitting effect.

3.8 Experiment 4-1(**NN Structure**):

- In the final phase of the project, we tried various types of neural network architectures: **VGG**, **LeNet-5**, **2012 AlexNet**, and **2014 GoogLeNet** (their structures and code are presented below).
- We also used four different activation functions (**ReLU**, **ELU**, **Leaky ReLU**, and **SiLU**), resulting in 4x4=16 groups of neural network structure experiments.
- In the coding part, we defined a method mix_seq_and_act(model, activation function) to generate new structure of convolution layer and fc layer.
- Different neural networks have their own built-in drop rates.

```
candidate_seq: List[Tuple[TypingOrderedDict[str, Optional[nn.Module]], TypingOrderedDict[str, Optional[nn.Module]]]] = [
    (OrderedDict([  # first struct: SmallVGG
        ('conv1', nn.Conv2d(3, 8, kernel_size=3, padding=1)),
        ('*1', None),
        ('conv2', nn.Conv2d(8, 16, kernel_size=3, padding=1)),
        ('*2', None),
        ('max1', nn.MaxPool2d(kernel_size=2, stride=2)),  # 16x16

        ('conv3', nn.Conv2d(16, 32, kernel_size=3, padding=1)),
        ('*3', None),
        ('conv4', nn.Conv2d(32, 48, kernel_size=3, padding=1)),
        ('*4', None),
        ('max2', nn.MaxPool2d(kernel_size=2, stride=2)),  # 8x8

        ('conv5', nn.Conv2d(48, 56, kernel_size=3, padding=1)),
        ('*5', None),
        ('conv6', nn.Conv2d(56, 64, kernel_size=3, padding=1)),
        ('*6', None),
        ('max3', nn.MaxPool2d(kernel_size=2, stride=2))  # 4x4
    ]), OrderedDict([
        ('fc1', nn.Linear(64 * 4 * 4, 512)),
        ('*1', None),
        ('fc2', nn.Linear(512, 256)),
        ('*2', None),
        ('fc3', nn.Linear(256, 10))
    ])),
```

Figure 3.8.1: First candidate model: SmallVGG

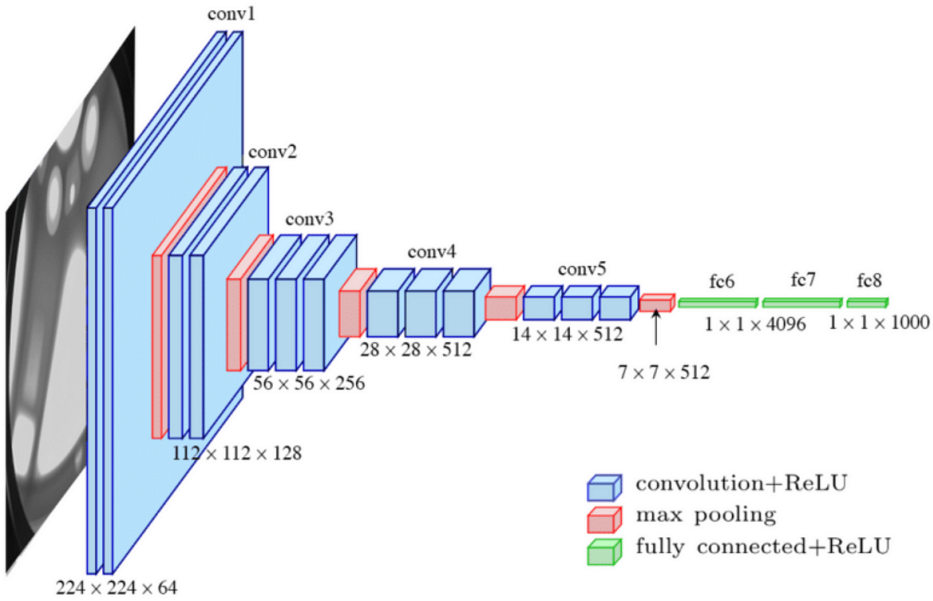

Figure 3.8.2: Diagram of a traditional VGG

```
(OrderedDict([  # second struct: LeNet-5
    ('conv1', nn.Conv2d(3, 12, kernel_size=5, stride=1, padding=2)),
    ('*1', None),
    ('avg1', nn.AvgPool2d(kernel_size=2, stride=2)),

    ('conv2', nn.Conv2d(12, 32, kernel_size=5)),
    ('*2', None),
    ('avg2', nn.AvgPool2d(kernel_size=2, stride=2)),
]), OrderedDict([
    ('fc1', nn.Linear(32 * 6 * 6, 256)),
    ('*3', None),
    ('fc2', nn.Linear(256, 128)),
    ('*4', None),
    ('fc3', nn.Linear(128, 10))
])),
```

Figure 3.8.3: Second candidate model: LeNet-5

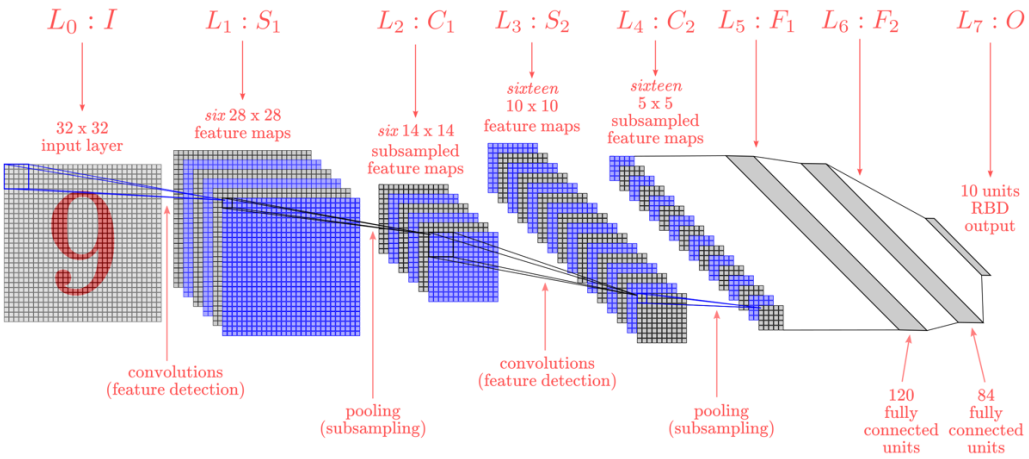

Figure 3.8.4: Diagram of LeNet-5 on 32x32x1 image

```
(OrderedDict([  # third struct: 2012AlexNet
    ('conv1', nn.Conv2d(3, 64, kernel_size=5, stride=1, padding=2)),  # 32x32
    ('*1', None),
    ('max1', nn.MaxPool2d(kernel_size=2, stride=2)),  # 16x16

    ('conv2', nn.Conv2d(64, 192, kernel_size=5, padding=2)),
    ('*2', None),
    ('max2', nn.MaxPool2d(kernel_size=2, stride=2)),  # 8x8

    ('conv3', nn.Conv2d(192, 384, kernel_size=3, padding=1)),
    ('*3', None),

    ('conv4', nn.Conv2d(384, 256, kernel_size=3, padding=1)),
    ('*4', None),

    ('conv5', nn.Conv2d(256, 256, kernel_size=3, padding=1)),
    ('*5', None),
    ('max3', nn.MaxPool2d(kernel_size=2, stride=2))  # 4x4
]), OrderedDict([
    ('fc1', nn.Linear(256 * 4 * 4, 4096)),  # 256 * 4 * 4 = 4096
    ('*6', None),
    ('dropout1', nn.Dropout()),

    ('fc2', nn.Linear(4096, 4096)),
    ('*7', None),
    ('dropout2', nn.Dropout()),

    ('fc3', nn.Linear(4096, 10))
])),
```

Figure 3.8.5: Third candidate model: 2012 AlexNet

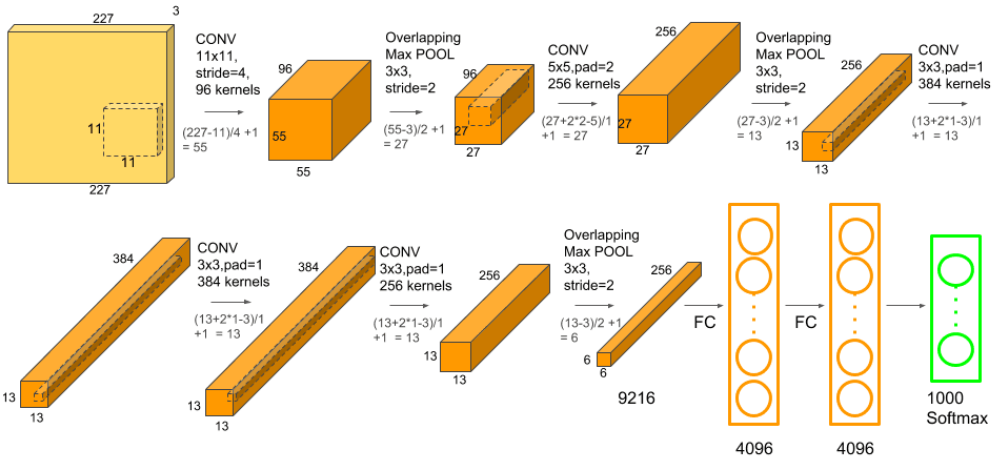

Figure 3.8.6: Diagram of AlexNet on 227x227x3 image

```
(OrderedDict([  # fourth struct: 2014GoogLeNet
    ('conv1', nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3)),
    ('max1', nn.MaxPool2d(kernel_size=3, stride=2, padding=1)),

    ('conv2', nn.Conv2d(64, 64, kernel_size=1)),
    ('conv3', nn.Conv2d(64, 192, kernel_size=3, padding=1)),
    ('max2', nn.MaxPool2d(kernel_size=3, stride=2, padding=1)),

    # Inception 模块
    ('inception3a', Inception(192, 64, 96, 128, 16, 32, 32)),
    ('inception3b', Inception(256, 128, 128, 192, 32, 96, 64)),
    ('max3', nn.MaxPool2d(kernel_size=3, stride=2, padding=1)),

    ('inception4a', Inception(480, 192, 96, 208, 16, 48, 64)),
    ('inception4b', Inception(512, 160, 112, 224, 24, 64, 64)),
    ('inception4c', Inception(512, 128, 128, 256, 24, 64, 64)),
    ('inception4d', Inception(512, 112, 144, 288, 32, 64, 64)),
    ('inception4e', Inception(528, 256, 160, 320, 32, 128, 128)),
    ('max4', nn.MaxPool2d(kernel_size=3, stride=2, padding=1)),

    ('inception5a', Inception(832, 256, 160, 320, 32, 128, 128)),
    ('inception5b', Inception(832, 384, 192, 384, 48, 128, 128)),

    ('avg1', nn.AdaptiveAvgPool2d((1, 1))),
    ('dropout1', nn.Dropout(0.4)),
]), OrderedDict([
    ('fc1', nn.Linear(1024, 10))
]))
```

Figure 3.8.7: Fourth candidate model: 2014 GoogLeNet

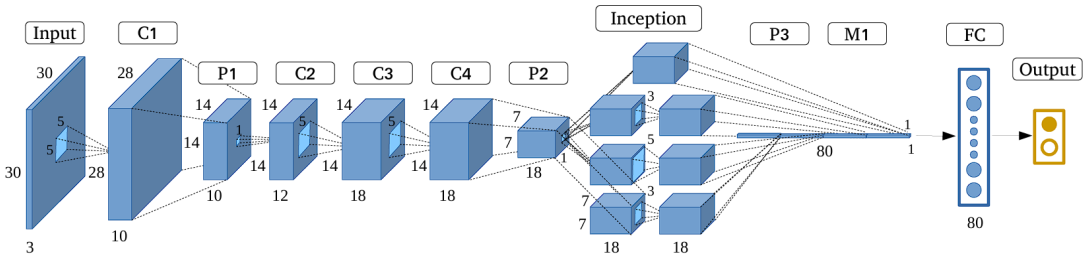

Figure 3.8.8: Diagram of GoogLeNet on 30x30x3 image

```python
def run_exp4_1(sequence_with_name: Tuple[List[str], List[Tuple]],
               activations: List,
               hyper_params: Dict[str, Any],
               train_loader: DataLoader,
               valid_loader: DataLoader) -> List[Dict[str, Union[List[float], dict, float, int]]]:

    combinations = list(itertools.product(sequence_with_name, activations))
    experiments = []

    for i, combo in enumerate(combinations):
        (seq_name, seq), activations = combo

        print(f"Running Exp {i + 1}: shape={seq_name}, activation func={activations.__class__.__name__}")

        this_model = SmallVGG()
        conv, fc = mix_seq_and_act(seq, activations)
        this_model.conv_layers = conv  # new conv_layers
        this_model.fc_layers = fc  # new fc_layers
        this_model = this_model.to(device)

        num_epochs = hyper_params['num_epochs']
        lr = hyper_params['lr']
        criterion = hyper_params['criterion']
        optimizer = hyper_params['optimizer'](this_model.parameters(), lr=lr)

        # Train Model
        print(f"Exp {i + 1}: Generating dataset from transform")
        train_losses, valid_losses = train_and_evaluate(this_model,
                                                        train_loader, valid_loader,
                                                        criterion, optimizer, num_epochs,
                                                        stop_early_params={
                                                            "min_delta": 0.01,
                                                            "patience": 5
                                                        })
```

replace with new conv/fc layers

Figure 3.8.9: partial code of experiment 4-1

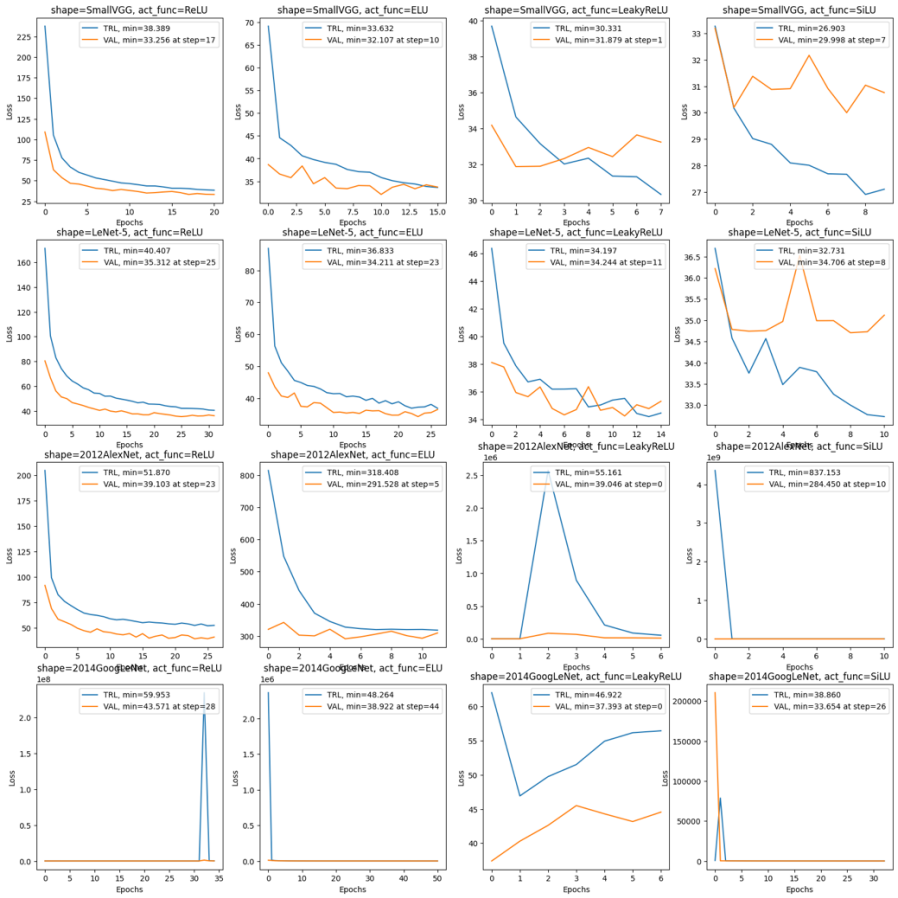

Figure 3.8.10: Train loss – valid loss curves of different combination in 4-1 experiments

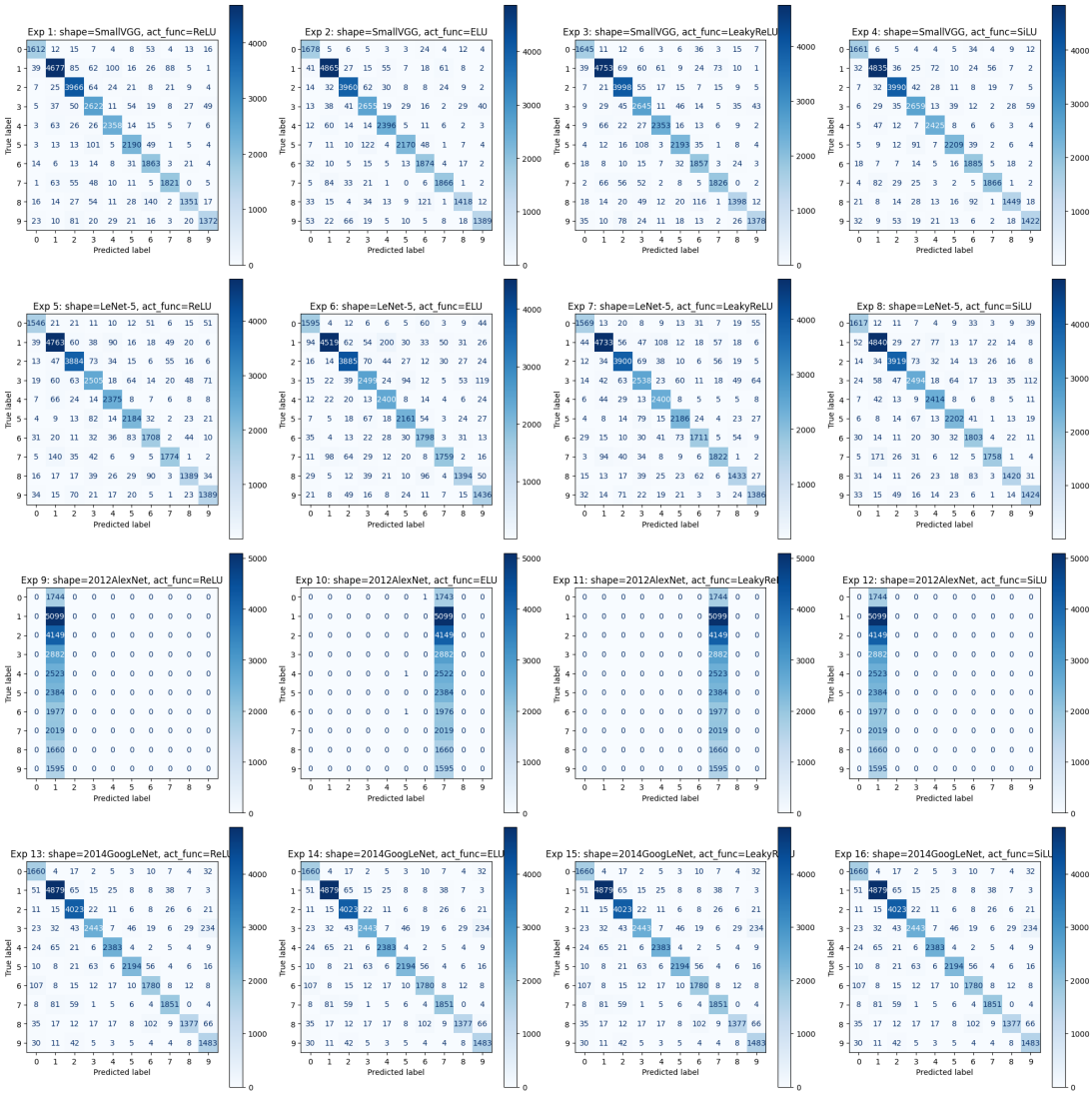

Figure 3.8.11: Confusion Matrix of different combinations of model and activation function

Result:

1. **(SmallVGG, SiLU)** had best performance among all sub-experiments.
2. In this experiment, we found that **SmallVGG is the most stable model** during training. Meanwhile, the **SiLU activation function allows each model to fit more quickly**. However, the downside is that SiLU exhibits significant fluctuations in loss calculation.
3. For each group of sub-experiments, we implemented an early stopping mechanism, which halts training after approximately 5 epochs once the model's validation loss reaches its minimum. If a lower validation loss is found, the program will continue to extend the epochs.

4 Analysis and Conclusion

- Through the multiple experiments conducted, we observed that neural networks show significant effectiveness in classifying images of individuals. Additionally, different hyperparameters can influence the model's training results to varying degrees.

4.1 Final hyperparameters:

The final model we determined is **`SmallVGG`**:
(3 multi-layers each consisting of 2 convolutional layers followed by 1 max pooling layer, plus 2 fully connected layers).

The optimizer used is **`optim.Adam()`**, and the criterion is **`nn.CrossEntropyLoss()`**.

The modified hyperparameter combinations include {

`Contrast Factor = 1.8`,
`Image Rotation = 45`,
`Random Cropping = 0.8`,
`Aspect Ratio Change = (0.75, 1.33)`
}.

The experimental hyperparameter combinations are {

`Epochs = 25`,
`Learning Rate = 0.001`,
`Batch Size = 128`
}.

4.2 Improvement can be done:

(1) Is there anything that needs improvement in each experiment?
The answer is **Yes**:
(a) We found that grid search is unnecessary in most cases. After brainstorming the tuning order of the parameters, we can **conduct individual experiments one by one for each hyperparameters**. This approach helps **avoid the excessive time costs** associated with grid search.
(b) In Experiment 4, we conducted experiments with different types of models. However, **we might focus our attention on one specific model**, such as SmallVGG, and adjust the number of convolutional layers, fully connected layers, and the quantity of input and output channels. At the same time, we can

try different pooling operations (MaxPool, AvgPool, or Adaptive AvgPool) as well as different dropout rates.

(2) About Application?

We have saved the trained .pth model, and in future studies, we can **use the Gradio library to create an interface for real-time interaction**, allowing us to better validate and utilize our training results.

4.3 What I want to say

- I am convinced that I learned a lot from the entire process of this project, and I also enjoyed it because I had met the beauty and fun of adjusting parameters.
- From the perspective of methodology, I still have many flaws and places that can be improved. Maybe next time I can sort out the goals and methods before executing.
- **Thank you for reading! I hope you can also enjoy the charm of adjusting parameter in machine learning.**