Overview of workflow of DL

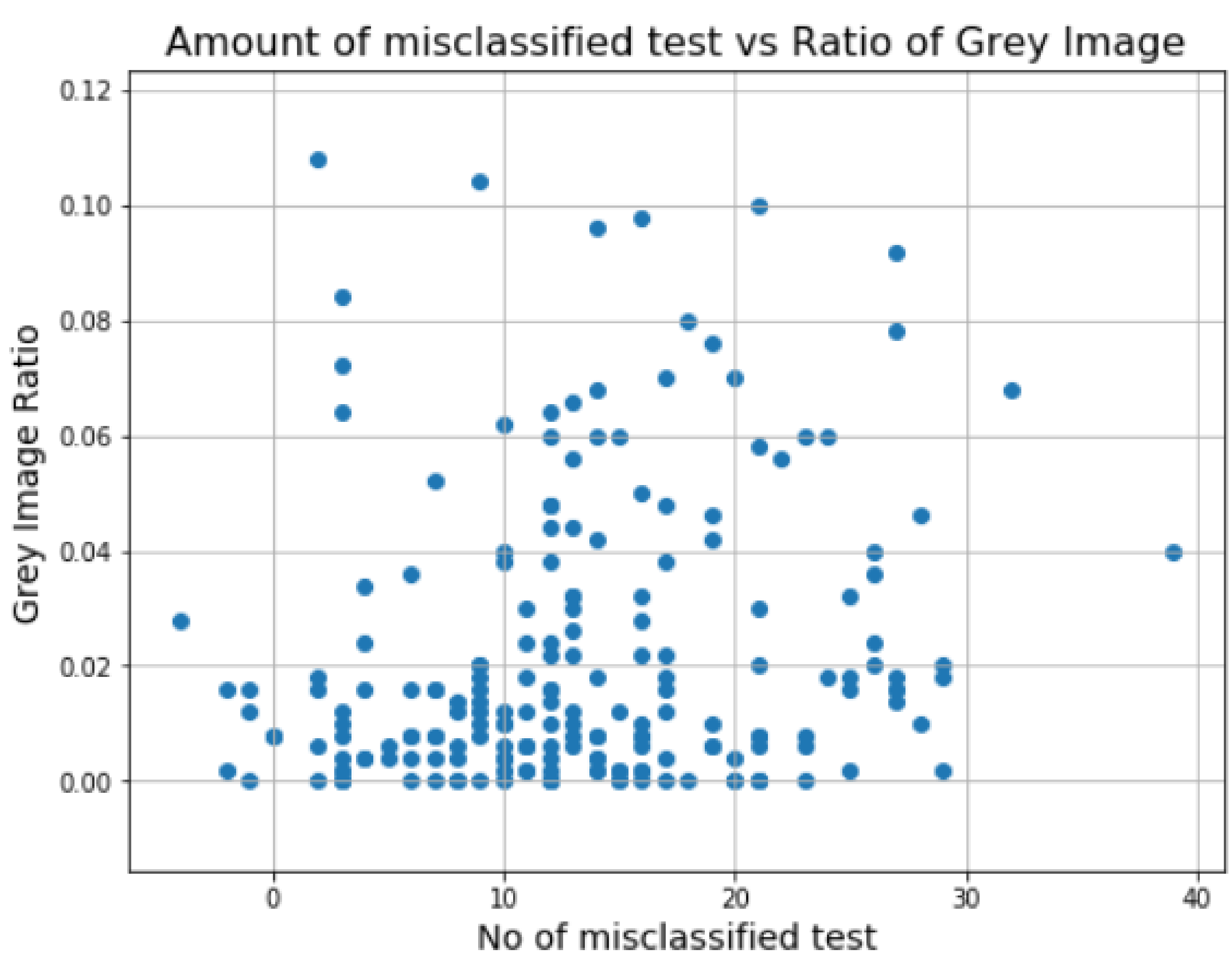
1. Gathering Data

All training and testing images are stored in `the-identification-game.zip` . In *Kaggle notebook*, the server will store data automatically. So we only need to enter correct directory and load required data. While in *Colab*, we need to mount *Google Drive* first and then download (unzip) file.

2. Data pre-processing

2.1. Analysis of grey image

Through experiments, we found there exists grey images. Since we are predicting a 3 RGB channel colored image, inputs with greyscale image will decrease the accuracy of the output because it only replicate the infomation on other 2 channels. Therefore, we did an investigation on the proportion of grey images. The result is shown as below:



More details can be found in [GreyImage_Ratio_Analysis.ipynb (GreyImage_Ratio_Analysis.ipynb)](GreyImage_Ratio_Analysis.ipynb). We can see that some class can have less than 10% grey images. So we try to avoid training grey images:

```
if (grey_image):
  skip
```

However, we got memory error due to pre-trained network only accept the entire dataset as input and it's hard to bundle new dataset without grey images.

In all, we accept/keep grey image in the input bundle because:

1. we found no significant correlation between amount of grey image and classification performance
2. the amount of grey image ratio is less than 10% for all classes
3. technical difficulties to separate them from given input

2.2. Data Augmentation (DA)

The extracted data can be constructed to `ImageFolder` object with `ToTensor()` transform. In order to apply normalization to dataset, we can calculate mean and standard deviation (std) through converting the object to `numpy.ndarray`.

- 90% training set and 10% validation set:
  - mean = [0.48038346, 0.44819227, 0.39750773]
  - std = [0.27704743, 0.26908568, 0.28212458]
- full training set:
  - mean = [0.44245052, 0.44358176, 0.44349745]
  - std = [0.29067352, 0.2874385, 0.28603515]
- In transfer learning, pre-trained model *Wide ResNet 50-2 and 101-2*: (from official document)
  - mean = [0.485, 0.456, 0.406]
  - std = [0.229, 0.224, 0.225]

In early stage, we construct network by ourselves (like LeNet-5, VGG-16, etc. see section 3 for more details) so that we can input images with 3x64x64. Then we have to use the former two mean and std to train (but in transfer learning, we have to use the last mean and std because those pre-trained models are trained with 224x224 images with specific mean and std). In this case, we tried to apply different DA strategies with `torchvision.transforms`:

1. `RandomHorizontalFlip()`
2. `RandomRotation(10)` + `RandomHorizontalFlip()`
3. `RandomRotation((20,30))` + `RandomHorizontalFlip()`
4. `RandomAffine(degrees=(0, 10), transalate=(0.1, 0.1))` + `RandomHorizontalFlip()`
5. `RandomApply([transforms=AddGaussianNoise(3e-7, 1.)], p=0.5)` + `RandomHorizontalFlip()`

The `AddGaussianNoise class` can be seen below:

In [ ]:
```
class AddGaussianNoise(object):
    def __init__(self, mean=0., std=1.):
        self.std = std
        self.mean = mean

    def __call__(self, tensor):
        # Add noise with Gaussian distribution
        return tensor + torch.randn(tensor.size()) * self.std + self.mean

    def __repr__(self):
        return self.__class__.__name__ + '(mean={0}, std={1})'.format(self.mean, self.std)
```

Through experiments, we found all strategies cannot achieve distinct imporvement. The situation is worse when we apply transfer learning. We tried different models (like VGG-19, ResNet-18) as baseline, after applying some of above strategies, the accuracy will decrease. This may be caused by the features of images:

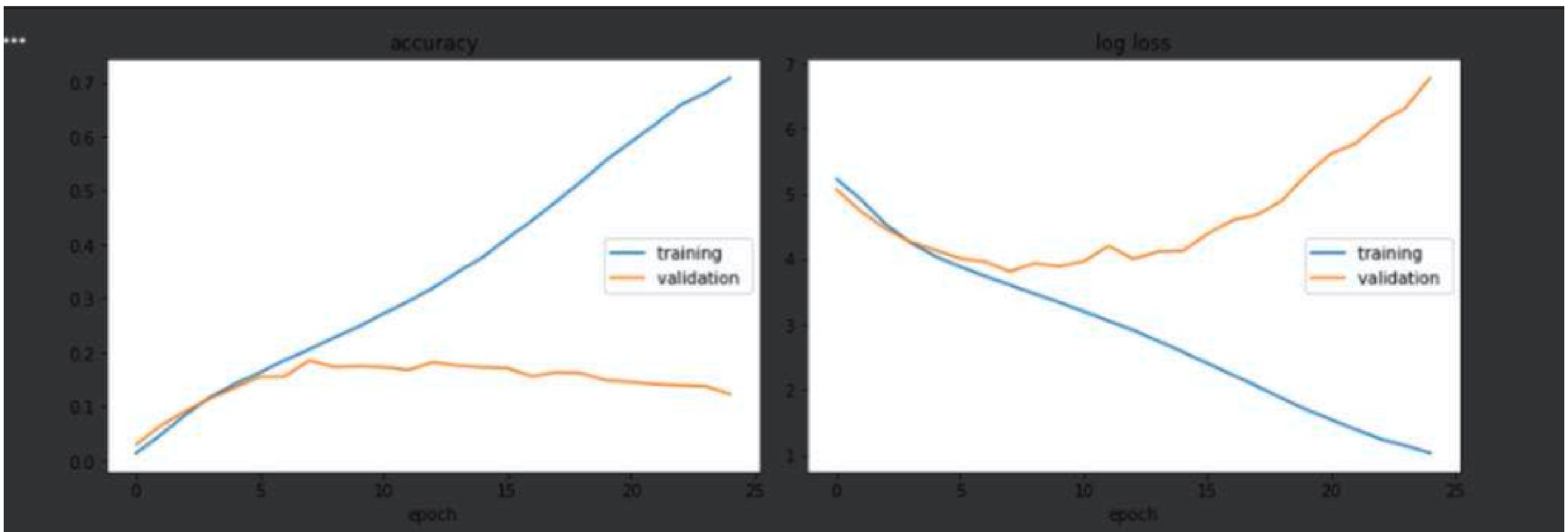
- Through obversation, we found there are some features located at the edge of the figure. Therefore, random rotation may destory those features
- Also, we found the orientation of images varies differently. For example, the class 'snail' has different shooting angle. Applying random rotation and horizontal flip will increase the training difficulty.
- Adding Gaussian noise may not have benefits as well because the pixel of the image is too small.

We can get the conclusion that due to the complex diversity of the dataset, apply DA randomly may not help in this case. For detailed analysis, please see [Investigate_input_for_data_augmentation (./Investigate_input_for_data_augmentation)](./Investigate_input_for_data_augmentation).

3. Best model research for data type

As a start, we construct LeNet5 with following architecture, but we got bad result:

	Size of input image n	Number of input channels	f	p	s	Size of output image (n+2p-f)/s+1	Number of output channels or filters	Number of output neurons	Size of Filter + 1	Number of Parameters
Conv1	64	3	9	0	1	56	6	18816	244	1464
MaxPool2	56	6	2	0	2	28	6	4704		
Conv3	28	6	9	0	1	20	16	6400	487	7792
MaxPool4	20	16	2	0	2	10	16	1600		
	Size of input							Number of output neurons		
FC5	1600							800	1601	1280800
FC6	800							400	801	320400
Output	400							200	401	80200
							Total Neurons	32920	Total Parameters	1690656

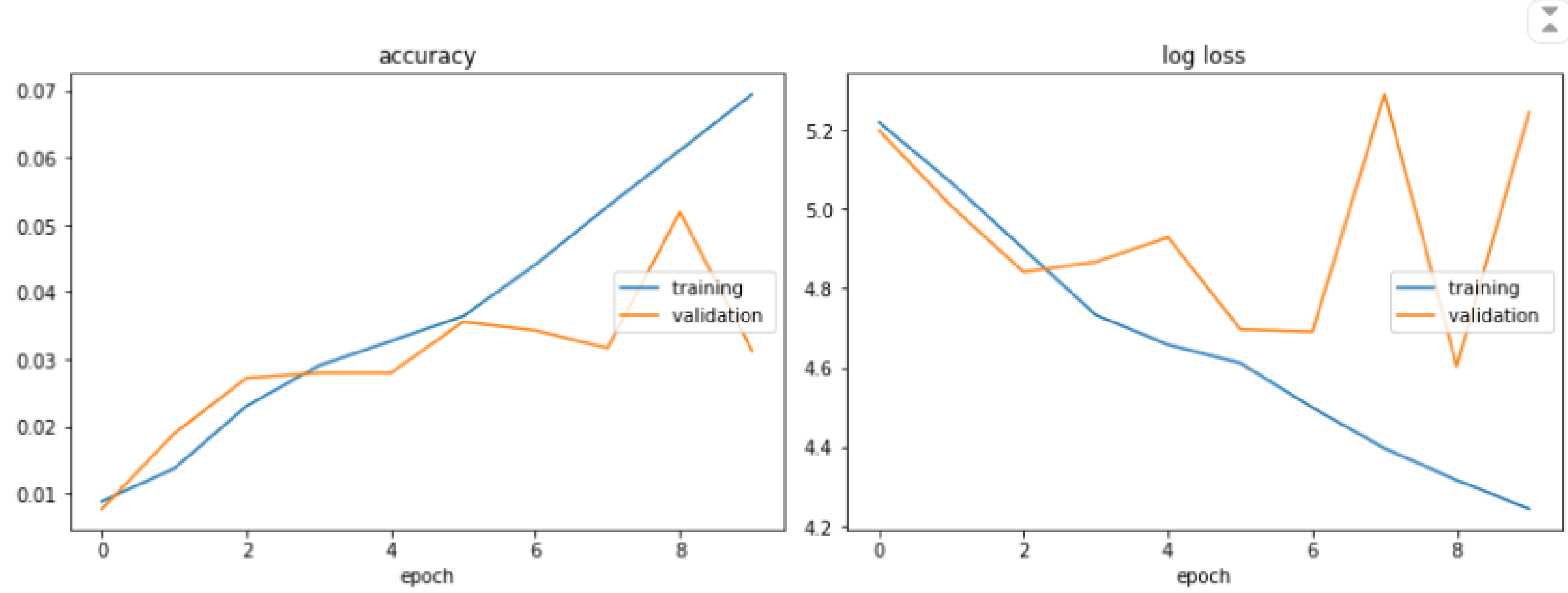



```
accuracy
        training             (min:    0.013, max:    0.709, cur:    0.709)
        validation           (min:    0.028, max:    0.185, cur:    0.123)
log loss
        training             (min:    1.026, max:    5.231, cur:    1.026)
        validation           (min:    3.807, max:    6.777, cur:    6.777)
```

Adjusting the hyperparameters cannot improve validation accuracy (around 0,2), so we try to build another more complex model VGG-16:

	Size of input image n	Number of input channels	f	p	s	Size of output image (n+2p-f)/s+1	Number of output channels or filters	Number of output neurons	Size of Filter + 1	Number of Parameters
Conv11	64	3	3	1	1	64	64	262144	28	1792
Conv12	64	64	3	1	1	64	64	262144	577	36928
MaxPool1	64	64	2	0	2	32	64	65536		
Conv21	32	64	3	1	1	32	128	131072	577	73856
Conv22	32	128	3	1	1	32	128	131072	1153	147584
MaxPool2	32	128	2	0	2	16	128	32768		
Conv31	16	128	3	1	1	16	256	65536	1153	295168
Conv32	16	256	3	1	1	16	256	65536	2305	590080
Conv33	16	256	3	1	1	16	256	65536	2305	590080
MaxPool3	16	256	2	0	2	8	256	16384		
Conv41	8	256	3	1	1	8	512	32768	2305	1180160
Conv42	8	512	3	1	1	8	512	32768	4609	2359808
Conv43	8	512	3	1	1	8	512	32768	4609	2359808
MaxPool4	8	512	2	0	2	4	512	8192		
Conv51	4	512	3	1	1	4	512	8192	4609	2359808
Conv52	4	512	3	1	1	4	512	8192	4609	2359808
Conv53	4	512	3	1	1	4	512	8192	4609	2359808
MaxPool5	4	512	2	0	2	2	512	2048		
	Size of input							Number of output neurons		
FC1	2048							2048	2049	4196352
FC2	2048							2048	2049	4196352
FC3	2048							200	2049	409800
							Total Neurons	1208520	Total Parameters	23517192

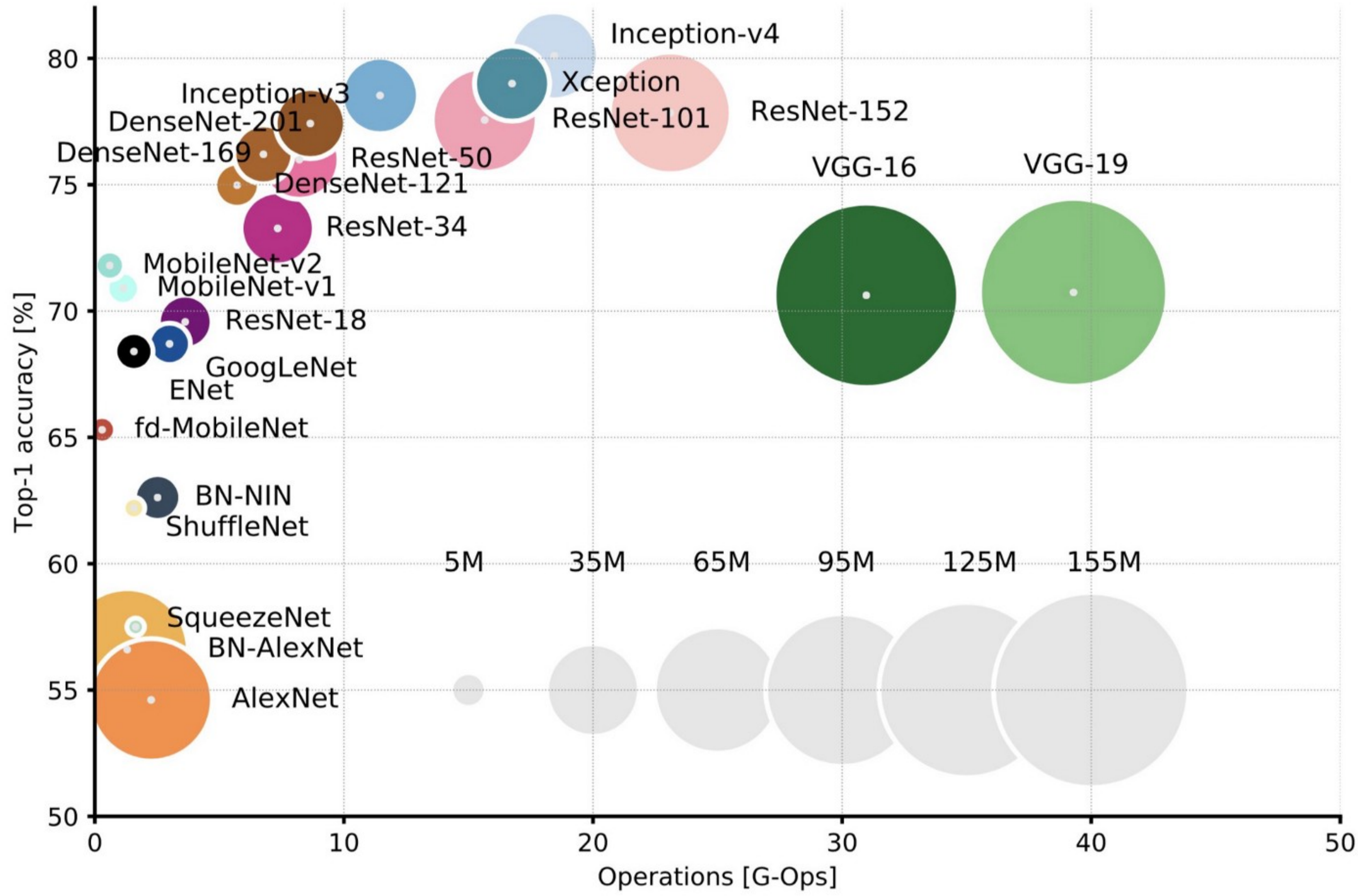
```
vgg16_model = train_model()
```



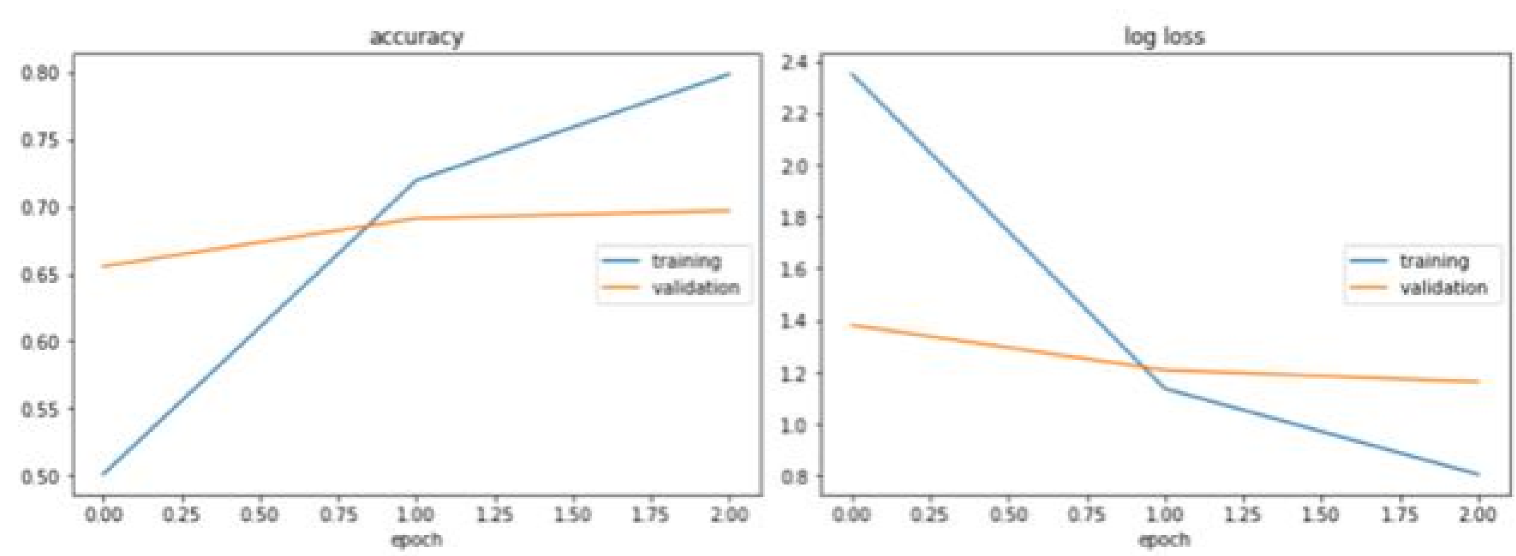

```
accuracy
        training             (min:    0.009, max:    0.069, cur:    0.069)
        validation           (min:    0.008, max:    0.052, cur:    0.031)
log loss
        training             (min:    4.243, max:    5.220, cur:    4.243)
        validation           (min:    4.603, max:    5.290, cur:    5.245)
```

+ Code   + Markdown

Through we can observe some improvements, that's still far away from the task of Image Classification. To solve such problem, transfer learning can be a good solution because `Pytorch` provide many models with low error rate on ImageNet, those models use 224x224x3-channel image as input and output 1000 classes, which can cover our 200 classes well. The below figure shows a rank for models on ImageNet:



As can be seen, VGG family models are extremely data-hungry. ResNet adpots skip-connection (see introduction section in [WideResNet_Algorithm (./WideResNet_Algorithm.pdf)](./WideResNet_Algorithm.pdf), which can adaptive adjustment of model complexity and avoid over-fitting. So we chose ResNet-18 as a baseline, which actually showed a much better performance:




```
accuracy
        training             (min:    0.501, max:    0.799, cur:    0.799)
        validation           (min:    0.656, max:    0.697, cur:    0.697)
log loss
        training             (min:    0.806, max:    2.350, cur:    0.806)
        validation           (min:    1.163, max:    1.381, cur:    1.163)
```

Note that we modified the last full connection layer parameters with following attempts:

1. directly change out 1000 to 200: `model_rn.fc = nn.Linear(model_rn.fc.in_features, 200)`
2. Add dropout with different probabilities (p=0.3, 0.4, 0.5, 0.6):

```
model_rn.fc = nn.Sequential(nn.Dropout(),
              nn.Linear(model_rn.fc.in_features, 200))
```

3. Add another fc layer:

```
model_rn.fc = nn.Sequential(nn.Linear(model_rn.fc.in_features, 1000)
              nn.Linear(1000, 200))
```

4. Add two dropout and another fc layer:

```
        model_rn.fc = nn.Sequential(nn.Dropout(),
                    nn.Linear(model_rn.fc.in_features, 1000)
                    nn.Dropout(),
                    nn.Linear(1000, 200))
```

Through experiments, we found that option 1 and option 3 showed good performance. The reason they work well may be that they keep the pre-training information of fc layer. Besides, applying dropout or batchnorm didn't show a significant improvement. So we abandoned. With this foundation, we tried ResNet-50 with around 0.76 validation accuracy and its improved version - Wide ResNet-50_2 (WRNs), which is more emphasize on widening rather than deepening the network. So it has benefit from having less gradient diminishing problem than resnet from the same number of parameters. WRNs can have more advantages because our dataset is not suitable for very deep neural network training. Finally, we tried WRN-101, which showed best performance with 0.817 validation accuracy. More details of the WRN-101 algorithm can be seen in [WideResNet_Algorithm (./WideResNet_Algorithm.pdf)](./WideResNet_Algorithm.pdf).

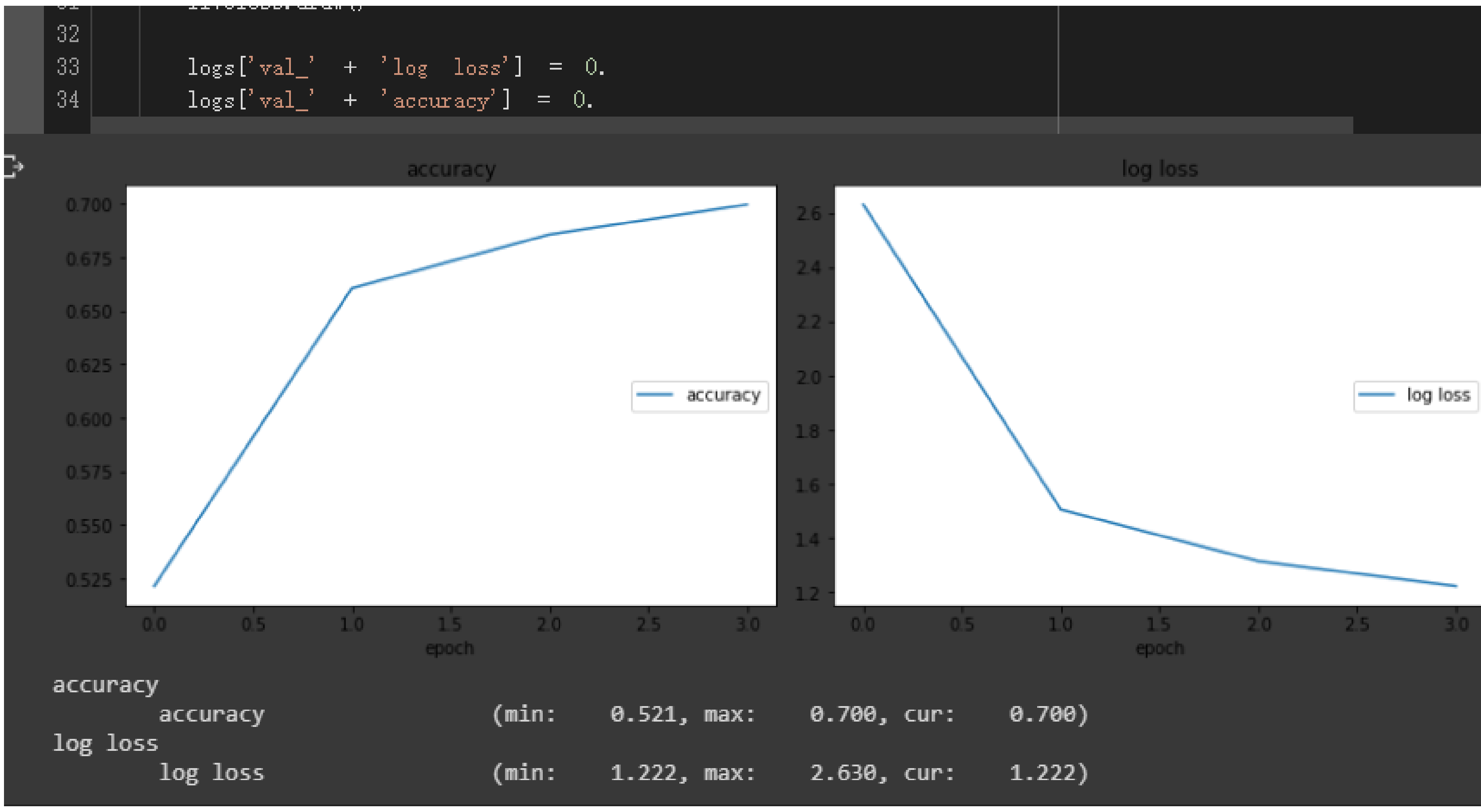
Besides, we tried ensemble learning, which combined ResNet-50 and WRN-50_2, however, the performance didn't improve. See the code below:

In [ ]:
```
class CustomEnsemble(nn.Module):
    def __init__(self, modelA, modelB, nb_classes=200):
        super(CustomEnsemble, self).__init__()
        self.modelA = modelA
        self.modelB = modelB
        # Remove last linear layer
        self.modelA.fc = nn.Identity()
        self.modelB.fc = nn.Identity()

        # Create new classifier
        self.classifier = nn.Linear(2048+2048, nb_classes)

    def forward(self, x):
        x1 = self.modelA(x.clone())  # clone to make sure x is not changed by inplace methods
        x1 = x1.view(x1.size(0), -1)
        x2 = self.modelB(x)
        x2 = x2.view(x2.size(0), -1)
        x = torch.cat((x1, x2), dim=1)

        x = self.classifier(F.relu(x))
        return x
```

Also, we need to freeze the two models:

In [ ]:
```
for param in modelA.parameters():
    param.requires_grad_(False)

for param in modelB.parameters():
    param.requires_grad_(False)
```

However, the result is not so good:



We want to try the combination of WRN-50_2 and WRN-101_2, but it takes more than three hours to run an epoch. So we didn't succeed to complete.

In conclusion, We can discover that the accuracy of these model is corresponding to the information showed in the Figure1, so from that we can confirm that our models decision is correct.

4. Training and testing the model on data

4.1. Hyper-parameters Selection

Here we just use the code provided by the implement lectures for training and testing. For the hyperparameters choose, we generally fix others then change one for several times(the strategy of trying one hyperparameter is using Bisection method)

1. Learning rate(lr). For ReLU activation method, lr should be small (as we discussed in ACSE-8 coursework 1). We selected lr=1e-2 as baseline. Through we found through small lr could improve accuracy, it would take more time to train, especially in deep CNNs. It took more than 2.5 hours to run an epoch with lr=1e-4 when training WRN-101_2. So we used baseline lr

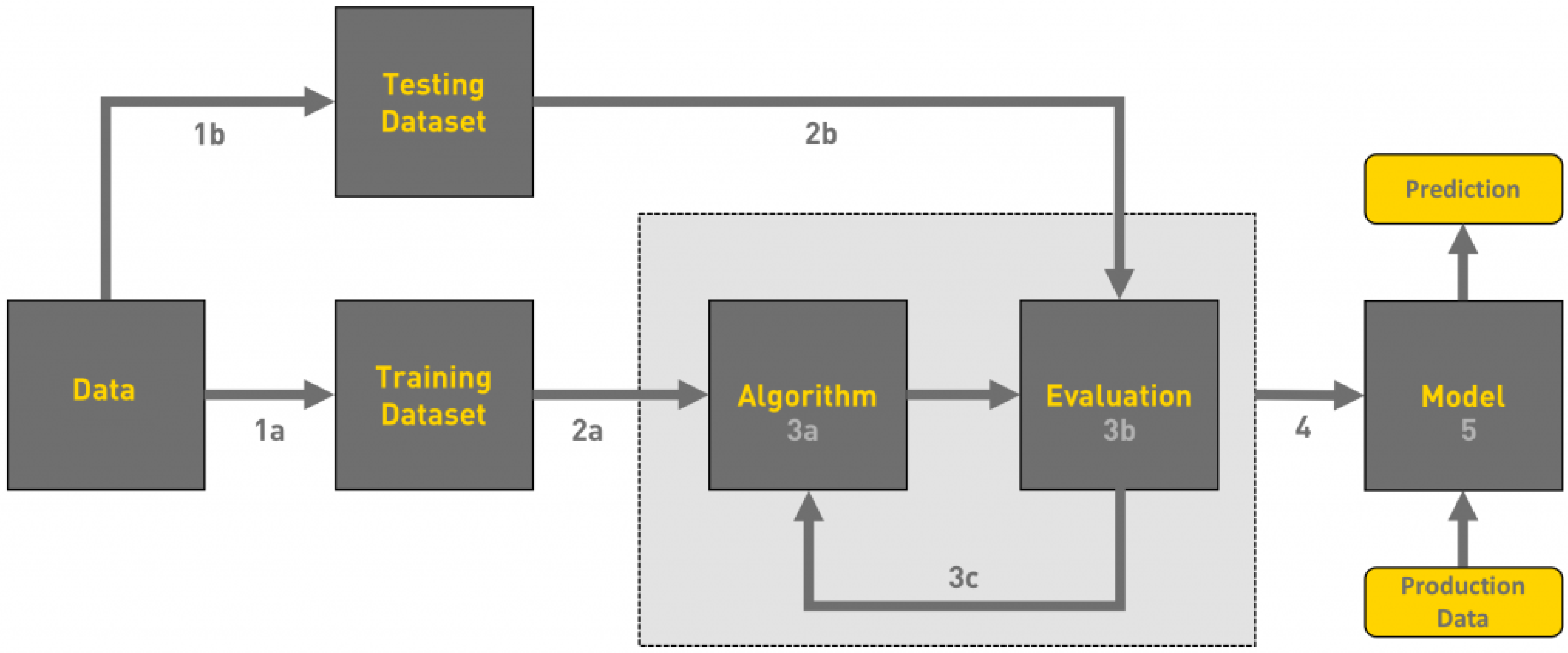
2. Weight decay. In our tests, weight_decay=1e-5 shows the best performance if we use weight_decay. But there is no significant improvement compared the optimizer without weight_decay

3. Momentum. We didn't change momentum because in the ACSE-8 afternoon exercise, a medium momentum value is reasonable

4. Batchsize. For training batch size, we selected 64. Normally, we chose size range between 32 and 128. So we didn't change training batch size. For validation batch size, we selected 500 instead of 1000 to avoid the hardware running out of memory and crashing

5. Epochs. To avoid overfitting, we only trained less than 5 epochs. We tried epoch 3 and 4 as final submission. However, we could still observe improving trend at epoch 4

4.2. Training, validation and testing



Through the figures of validation and training accuracy and loss, we can canmpare the performence of our temporary model and judge whether the hyperparameters are good enough. When we have got the optimal hyperparamters on temporary model, we will train full training set and then test this model through uploading the predictions of test data. Finally, we will compare the scores with the former one , and analysis which model we should choose next time until we get the final best score on Kaggle.

When training the model, we also observed overfitting, which is expected because the model is relatively large compared to the data. So we try to regularize by implement DA and dropout. However, the result slightly improves the overfit but worsen the validation accuracy. Therefore, from such tradeoff, we decided not to implement DA or dropout, but use small number of epoch to avoid overfitting.

In the process of training and testing for each iteration, we will save model one time to make sure our weights not loss when the network broken up.

5. Evaluation

Since we don't have labels of the test set, we consider model performance form validation loss and accuracy. Also, we check the rate of over-fitting.

The evaluation metric for this competition is Mean F1-Score. The F1 score, commonly used in information retrieval, measures accuracy using the statistics precision p and recall r. Precision is the ratio of true positives (tp) to all predicted positives (tp + fp). Recall is the ratio of true positives to all actual positives (tp + fn). The F1 score is given by:

$$F1 = 2\frac{p \cdot r}{p + r} \quad \text{where } \ p = \frac{tp}{tp + fp}, \ r = \frac{tp}{tp + fn}$$

The F1 metric weights recall and precision equally, and a good classification algorithm will maximize both precision and recall simultaneously. Thus, moderately good performance on both will be favored over extremely good performance on one and poor performance on the other.

Conclusion

In all, we used WRN-101_2, which shows satisfying performance. However, it shows high possibility to overfit. In further, we may try three-model ensemble learning with ResNet, GoogleNet and DenseNet because they don't need large dataset to support and show good performance in the rank figure. Also, we can try different optimizer like `Adam` with different lr (i.e. 1e-4).