VRDL Homework 1: Bird Image Classification

Zhi-Yi Chin
joycenerd.cs09@nycu.edu.tw

November 3, 2021

1 Introduction

In this homework, we participated a bird image classification challenge hosted on CodaLab. This challenge provided a total of 6,033 bird images (3000 for training and 3033 for testing) belonging to 200 bird species. For the fairness of this challenge, external data is not allowed to train our model. We can use pre-trained models that have been released by other people online but are limited to pre-train on ImageNet. For the code implementation, please visit GitHub: https://github.com/joycenerd/bird-images-classification.

2 Methodology

2.1 Data pre-processing

Before the actual training, we first split the training data into 70% and 30% of the whole training data. Because we want to use some of the data for evaluation since the provided data does not have a validation set. We randomly split the data by uniform distribution. We wrote the splitting results into two text files, one for training and one for validation. In each line of the text files, we record the image data's full path and classification label, which we transformed into integer numbers.

Then we constructed a custom dataset class. In this class, we read in the training or validation text files above. We read in the image path and the label in each line. We located the image from the image path and applied transformation. For training data, the transformations we had applied are RandomAffine, RandomGrayscale, RandomHorizontalFlip, RandomPerspective, RandomVerticalFlip, ColorJitter, RandomShift, color transform, RandomRotation. For all the data, the transfomations we had applied are resize the image to 380×380, ToTensor, normalize to mean=[0.485,0.456,0.406], std=[0.229,0.224,0.225].

```
def _random_colour_space(x):
    output = x.convert("HSV")
    return output

class RandomShift(object):
    def __init__(self, shift):
        self.shift = shift
    @staticmethod
```

```
    def get_params(shift):
        """Get parameters for ``rotate`` for a random rotation.
        Returns:
            sequence: params to be passed to ``rotate`` for random rotation.
        """
        hshift, vshift = np.random.uniform(-shift, shift, size=2)
        return hshift, vshift
    def __call__(self, img):
        hshift, vshift = self.get_params(self.shift)
        return img.transform(img.size, Image.AFFINE, (1, 0, hshift, 0, 1, vshift),
        resample=Image.BICUBIC, fill=1)

def make_dataset(mode,data_root,img_size):
    colour_transform = transforms.Lambda(lambda x: _random_colour_space(x))
    transform = [
        transforms.RandomAffine(degrees=30, shear=50, fillcolor=0),
        transforms.RandomGrayscale(p=0.5),
        transforms.RandomHorizontalFlip(p=0.5),
        transforms.RandomPerspective(
            distortion_scale=0.5, p=0.5, fill=0),
        transforms.RandomVerticalFlip(p=0.5),
        transforms.ColorJitter(
            brightness=0.5, contrast=0.5, saturation=0.5, hue=0.5),
        transforms.Grayscale(num_output_channels=3),
        RandomShift(3),
        transforms.RandomApply([colour_transform]),
    ]
    data_transform_train = transforms.Compose([
        transforms.RandomResizedCrop(img_size),
        transforms.RandomApply(transform, p=0.5),
        transforms.RandomApply([transforms.RandomRotation(
            (-90, 90), expand=False, center=None)], p=0.5),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[
                                0.229, 0.224, 0.225])
    ])

    data_transform_dev = transforms.Compose([
        transforms.Resize((img_size, img_size)),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[
                                0.229, 0.224, 0.225])
    ])
    if(mode == "train"):
        data_set = BirdDataset(data_root, mode, data_transform_train)
    elif(mode == "eval"):
        data_set = BirdDataset(data_root, mode, data_transform_dev)
```

```
    return data_set
```

Listing 1: The implementation of the data augmentation.

2.2 Model Architecture

We used EfficientNet-b4[5] as our model. We did not implement the model by ourselves; instead, we installed the efficientnet-pytorch[3] package using pip and then imported it to our code. In this way, we can use EfficientNet as a standard API.

2.2.1 Why EfficientNet?

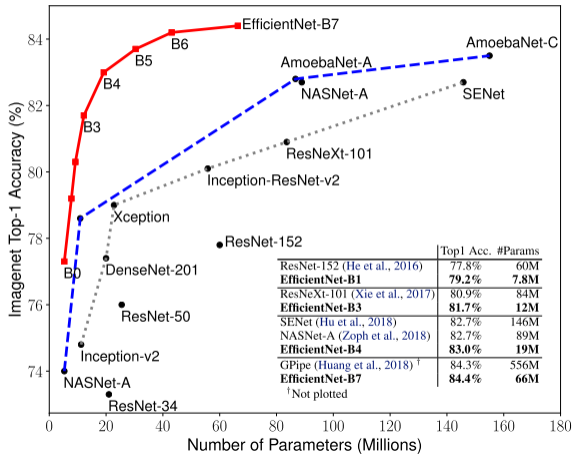

Figure 1: Model size v.s. ImageNet accuracy.

From Figure 1, we can see that EfficientNets significantly outperform other ConvNets. The great thing about EfficientNets is that not only do they have better accuracies compared to their counterparts, they are also lightweight and, thus, faster to run.

2.2.2 How EfficientNet?

The main idea of [5] is compound scaling. Before the EfficientNets came along, the most common way to scale up ConvNets was either by one of three dimensions: depth (number of layers), width (number of channels), or image resolution (image size). EfficientNet, on the other hand, performs compound scaling, that is, scales all three dimensions while maintaining a balance between all dimensions of the network.

First, the author of [5] used Neural Architecture Search (NAS) to get a mobile-size network that's very similar to MNasNet[6], and they named it EfficientNet-B0, which is the baseline network. Next, the authors scaled this baseline network by using the compound scaling technique to scale depth(d), width(w), and resolution(r) to get EfficientNet B1-B7.

```
from efficientnet_pytorch import EfficientNet
model=EfficientNet.from_pretrained('efficientnet-b4',num_classes=num_classes)
```

Listing 2: The implementation of EfficientNet-B4.

2.3 Hyperparameters

We use the cross-entropy loss as our loss function, stochastic gradient descent as our optimizer with an initial learning rate of 0.005, momentum 0.9, and weight decay 1e-5, and ReduceLROnPlateau as our learning rate scheduler to reduce the learning rate when evaluation loss has stopped improving; we set the factor to 0.5, patience to 4, and cooldown to 1. We set the batch size to 24. We train our model on 2 NVIDIA RTX 1080Ti for 200 epochs, and this process takes about 5-6 hours.

3 Summary

3.1 Results

Without a scheduler, we got accuracy 55.29% on the test set. After adding the scheduler, we got an accuracy of 72.53% on the test set.

3.2 Other ideas

3.2.1 Other optimizer

We have also tried using another optimizer. We have tried SAM[1] optimizer. However, we found that the results were not that great. The highest accuracy we can get after using the SAM optimizer is 53%. In [1], it did mention it uses cross-entropy loss with label smoothing as its loss function, and cosine annealed warm restart learning scheduler, so we also implemented with the SAM optimizer, but we get worse accuracy 48.89%.

3.2.2 Model ensembling

We have tried model ensembling. Model ensembling is to ensemble multiple models, and ideally, multiple models will perform better than one single model. We cannot ensemble EfficientNet-b4 with other models due to our hardware limitation since EfficientNet-b4 is too big. We choose the ResNet series. The first ensemble method is shown as Figure 2. We classify both modelA and modelB so that we will get two vectors with the size of num_classes. Then we use a learnable weight w to aggregate the two vectors to get the final results. We choose ResNet50[2] as our modelA and ResNest50[7] as our modelB. The implementation of our first ensemble method is as follow:

```
class EnsembleModel(nn.Module):
    def __init__(self,num_classes):
        super(EnsembleModel,self).__init__()
        # model A: resnet50
        self.modelA=torchvision.models.resnet50(pretrained=True)
        in_feat=self.modelA.fc.in_features
        self.modelA.fc=nn.Linear(in_feat,num_classes)
        nn.init.xavier_uniform_(self.modelA.fc.weight)

        # model B: resnest50
        self.modelB=torch.hub.load('zhanghang1989/ResNeSt', 'resnest50',
```

```
                                    pretrained=True)
        in_feat=self.modelB.fc.in_features
        self.modelB.fc=nn.Linear(in_feat,num_classes)
        nn.init.xavier_uniform_(self.modelB.fc.weight)

        self.w=nn.Parameter(torch.tensor(0.5,dtype=torch.double),
                            requires_grad=True)

    def forward(self,x):
        x1=self.modelA(x.clone())
        x1=x1.view(x1.size(0),-1)

        x2=self.modelB(x)
        x2=x2.view(x2.size(0),-1)

        # out=self.multi_layer(x1,x2)
        out=self.w*x1+(1-self.w)*x2
        return out
```

Listing 3: Implementation of the first model ensembling method.

Unfortunately, the first ensemble method does not yield very good results. The accuracy is 53.44%.

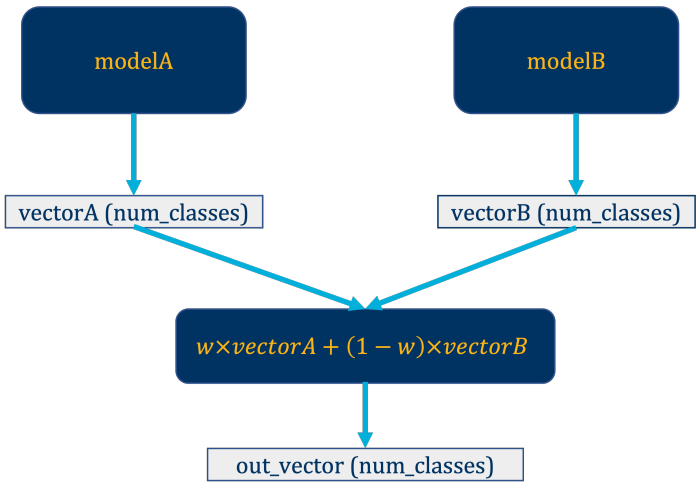

Figure 2: First ensemble method.

The second ensemble method is shown in Figure 3. We extract the feature vector before the classification layer from both modelA and modelB. Then we concatenate the two feature vectors and define our classification layer. Our classification layer has two fully connected layers and a ReLU activation layer in between. Same as the first ensemble method, we use ResNet50 as modelA and ResNest50 as modelB. The accuracy of this method is slightly higher than the first ensembling method, and it is 60.83%. Then we try ResNet101 and modelA and ResNest101 as modelB, and the accuracy is 66.24%. The implementation of our second ensemble method is as follow:

```python
class EnsembleModel(nn.Module):
    def __init__(self,num_classes,layer):
        super(EnsembleModel,self).__init__()
        # model A: resnet50
        if layer==50:
            self.modelA=torchvision.models.resnet50(pretrained=True)
        elif layer==101:
            self.modelA=torchvision.models.resnet101(pretrained=True)
        featA=self.modelA.fc.in_features
        self.modelA.fc=nn.Identity()

        # model B: resnest50
        if layer==50:
            self.modelB=torch.hub.load('zhanghang1989/ResNeSt', 'resnest50',
                                       pretrained=True)
        elif layer==101:
            self.modelB=torch.hub.load('zhanghang1989/ResNeSt', 'resnest101',
                                       pretrained=True)
        featB=self.modelB.fc.in_features
        self.modelB.fc=nn.Identity()

        # classifier
        self.classifier=nn.Sequential(
            nn.Linear(featA+featB,2048),
            nn.ReLU(),
            nn.Linear(2048,num_classes)
        )

    def forward(self,x):
        x1=self.modelA(x.clone())
        x1=x1.view(x1.size(0),-1)

        x2=self.modelB(x)
        x2=x2.view(x2.size(0),-1)

        out=torch.cat((x1,x2),dim=1)
        out=self.classifier(out)
        return out
```

Listing 4: The implementation of the second ensembling method.

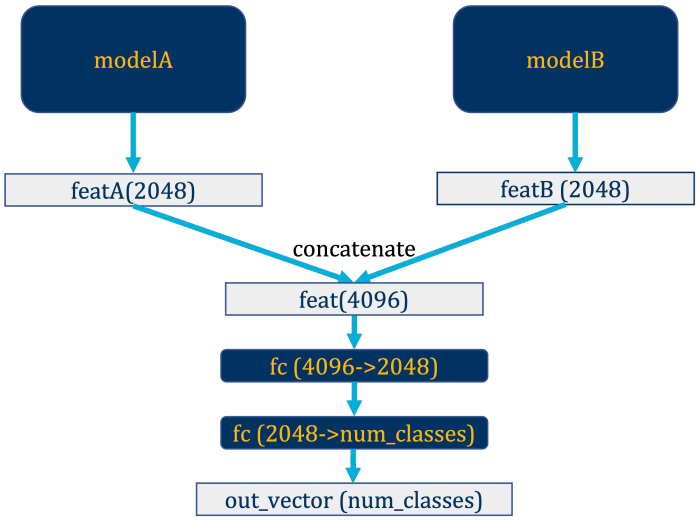

Figure 3: Second ensemble method.

3.3 Online hard example mining

Online hard example mining[4] is mainly used in object detection tasks, but it can apply to classification as well. The general idea of hard example mining is that once the loss (and gradients) are computed for every sample in the batch, we can sort batch samples in descending order of losses, then pick top-k samples from it, and only do backward pass only for those k samples. This ensures that sample which does not contribute much to the network's learning is ignored. Ideally, applying hard example mining can improve accuracy at the same time, decrease computational complexity.

We applied cyclical online hard example mining. At the beginning of the training, backpropagate all the samples ($k = \text{num\_of\_samples}$), then we linearly decrease the hard example mining ratio (k decreases). We do this for two-cycle. The change of ratio is shown in Figure 4.

The accuracy is 71.48% which is slightly lower than the best accuracy mentioned in Section 3.1. We think the reason is that our batch size is tiny (20). When k becomes smaller, then we may use only 1 or 2 samples to update the parameters; this is when our training becomes unstable. Unstable training causes the training loss to increase, so the whole system does not converge in the end.

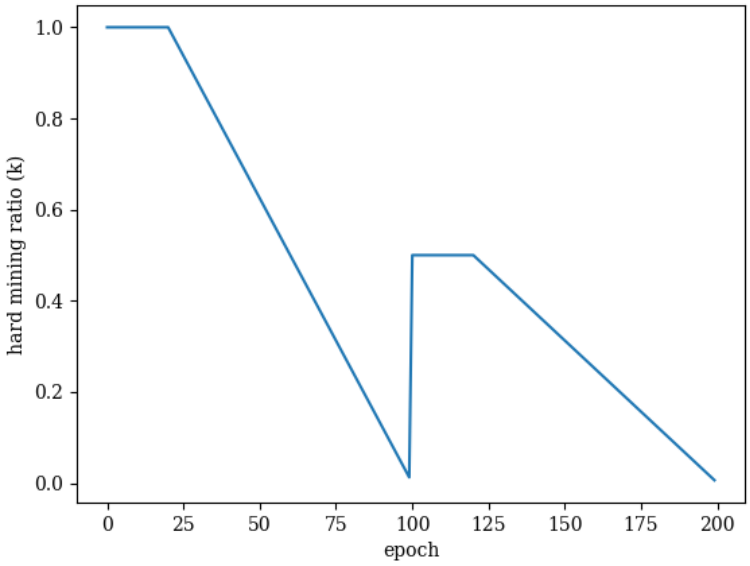

Figure 4: The hard mining example change of ratio (cyclical) during training.

```
class NLL_OHEM(torch.nn.NLLLoss):
    """ Online hard example mining.
    Needs input from nn.LogSotmax() """
    def __init__(self, ratio,device,total_ep):
        super(NLL_OHEM, self).__init__(None, True)
        self.ratio = ratio
        self.device=device
        self.total_ep=total_ep
    def forward(self, x, y, epoch,sched_ratio=True):
        num_inst = x.size(0)
        if sched_ratio:
            self.ratio_sched(epoch)
        else:
            self.ratio=1
        num_hns = int(self.ratio * num_inst)
        if num_hns>0:
            x_ = x.clone()
            inst_losses = torch.zeros(num_inst).to(self.device
            inst_losses = torch.autograd.Variable(inst_losses)
            for idx, label in enumerate(y.data):
                inst_losses[idx] = -x_.data[idx, label]
            _, idxs = inst_losses.topk(num_hns)
            x_hn = x.index_select(0, idxs)
            y_hn = y.index_select(0, idxs)
            loss=torch.nn.functional.nll_loss(x_hn, y_hn,reduction='mean')
        else:
            loss=torch.nn.functional.nll_loss(x,y,reduction='mean')
```

```
        return loss

    def ratio_sched(self,epoch):
        half=int(self.total_ep/2)
        max_range=int(half*0.2)
        if epoch<half:
            if epoch<max_range:
                self.ratio=1.0
            else:
                self.ratio=(half-epoch)/float(half-max_range)

        else:
            if epoch<(half+max_range):
                self.ratio=0.5
            else:
                self.ratio=0.5*(self.total_ep-epoch)/float(half-max_range)
```

Listing 5: The implementation of cyclical online hard example mining.

We make another attempt with online hard example mining. We schedule the negative sample ratio using steps. Since we think continuously changing the negative sample ratio makes our learning unstable, we are using the step method this time. We let our system learn at least 10 epochs until we change our ratio. We gradually decrease our negative sample ratio until it is 0.4. The change of ratio is shown in Figure 5 We get an accuracy of 72.5%. It is still lower than the best results we have.

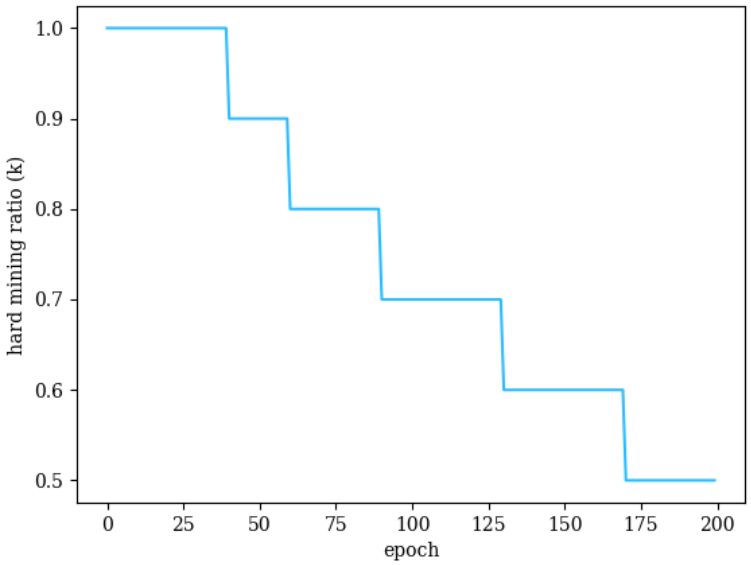

Figure 5: The hard mining example change of ratio (step) during training.

```
def step_ratio_sched(self,epoch):
    if epoch<40:
```

```
        self.ratio=1
    elif epoch>=40 and epoch<60:
        self.ratio=0.9
    elif epoch>=60 and epoch<90:
        self.ratio=0.8
    elif epoch>=90 and epoch<130:
        self.ratio=0.7
    elif epoch>=130 and epoch<170:
        self.ratio=0.6
    elif epoch>=170:
        self.ratio=0.5
```

Listing 6: Online hard negative mining ratio scheduler (step).

3.4 Conclusion

The following points are what we think contributed to our accuracy:

1. Many data augmentations: since we got a small dataset, we use data augmentations to enlarge our data diversity.
2. Bigger model (EfficientNet-B4) than ResNet50: this works well in our implementation because we have used data augmentation to enlarge our dataset.
3. Learning rate scheduler: using learning rate scheduler to let our model converge to a better place.

References

[1] P. Foret, A. Kleiner, H. Mobahi, and B. Neyshabur. Sharpness-aware minimization for efficiently improving generalization. In *International Conference on Learning Representations (ICLR)*, 2021.

[2] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, pages 770–778, 2016.

[3] L. Melas-Kyriazi. Efficientnet pytorch, 2020. URL https://github.com/lukemelas/EfficientNet-PyTorch.

[4] A. Shrivastava, A. Gupta, and R. Girshick. Training region-based object detectors with online hard example mining. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, pages 761–769, 2016.

[5] M. Tan and Q. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning (ICML)*, pages 6105–6114. PMLR, 2019.

[6] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2820–2828, 2019.

[7] H. Zhang, C. Wu, Z. Zhang, Y. Zhu, Z. Zhang, H. Lin, Y. Sun, T. He, J. Muller, R. Manmatha, M. Li, and A. Smola. Resnest: Split-attention networks. *arXiv preprint arXiv:2004.08955*, 2020.