ECE449 Final Project Report
3080Ti

Zhongkai Shangguan, Yue Zhao, Jingwen Wang

April 25, 2020

1 Method Description:

1.1 Augmentation and Preprocessing Method:

- Cropping: Crop, pad and rescale the image into (224,224).
- Rotation: Rotate the image between -10 degree and +10 degree with 50% chance.
- Flip: Do horizontal flip with 50% chance.
- Image contrast augmentation: Do one of the three image augmentation below with 50% chance.
  - RandomGamma: do random gamma between the limit of (60, 120).
  - HueSaturationValue: color saturation changes with hue change in range(-10,10) and saturation change in range (-20,20).
  - RandomBrightnessContrast: brightness change in range (-0.2,0.2).
- Blur: Smooth the image with with with 50% chance with three different filter:
  - GaussianBlur: with blur_limit=4.
  - MotionBlur: with blur_limit=4.

- MedianBlur: with blur_limit=4.

- CoarseDropout: Cover the original image with up to five 8×8 black squares with 50% chance.

By doing image augmentation, we can have a better performance when dealing with limited image dataset and avoid overfitting, i.e. robust to validation and test set.

1.2 Network Architecture:

2-a Backbones

We test and modified different Deep-CNN architectures, including ResNet34, PNASNet-5-large, Efficientnet-b7 and SE-ResNeXt101. All models are initialized from pre-trained models on ImageNet. ResNet34 and PNASNet-5 is used as our baseline, so here we list Efficientnet-b7 and SE-ResNeXt101(Figure 1) which is used as benchmark.

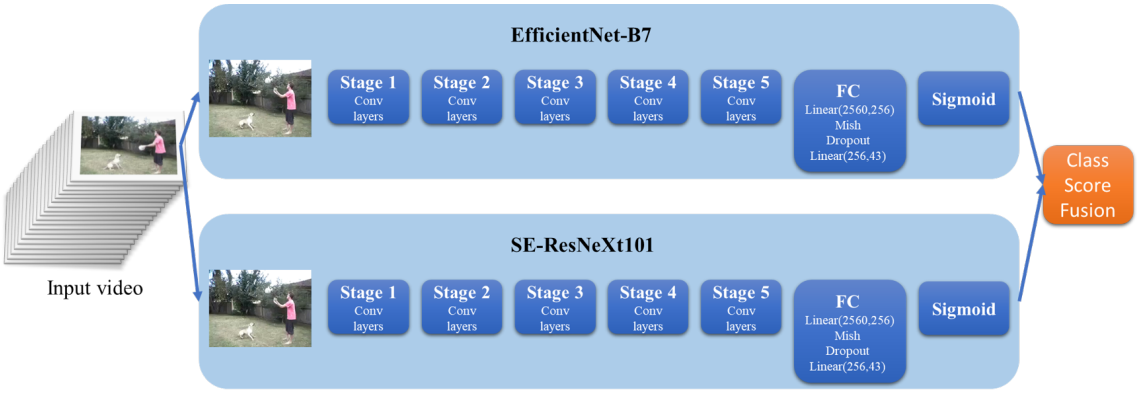

Figure 1: *The architectures we use for our training method*

- EfficientNet-B7 [1]
  The base of EfficientNet-B7 is EfficientNet-B0 (Figure 2), of which architecture is simple and clean, so it is easier to scale and generalize. From the base architecture, EfficientNet-B7 can be generated by the following two steps:

a. Fix $\Phi = 1$, assuming twice more resources available, and set $\alpha = 1.2$, $\beta = 1.1$, $\gamma = 1.15$, under constraint of $\alpha \cdot \beta^2 \cdot \gamma^2 \approx 2$.

b. Fix α, β, γ as constants and scale up baseline network with different Φ.

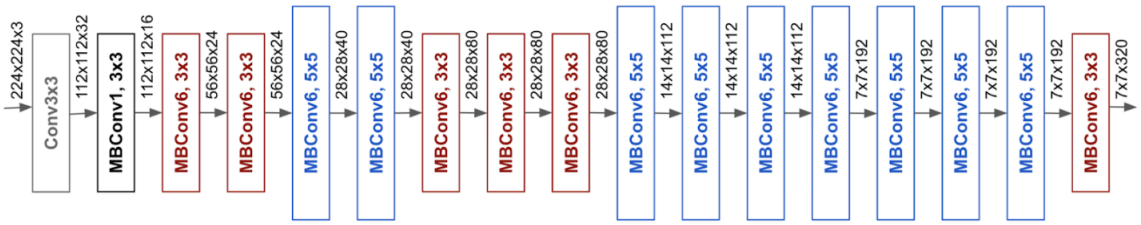

Figure 2: *The architecture for our baseline network EfficientNet-B0*

- SE-ResNeXt101 [2]

  ResNeXt is a simple architecture which adopts VGG/ResNets' strategy of repeating layers, while exploiting the split-transform-merge strategy in an easy, extensible way. The innovation lies in the proposed aggregate transformations, which replaces the original ResNet's three-layer convolution block with a parallel stack of blocks of the same topology structure (Figure 3), which improves the accuracy of the model without significantly increasing the magnitude of the parameter.

  Then apply SE (Squeeze-and-Excitation) blocks to ResNext we get SE-ResNext model, the SE block is shown in (Figure 4), the main idea of SE block is by giving different channels a weight which make the model also learn feature of channels. We notice that after convolution, we first apply global maxpooling to squeeze the 2-d channel into a number, then using fc layers learn the weight of each channel, finally multiply the weights to the original 2-d feature.

2-b Fully Connected Layer

We also add a FC layer (fully connected layer, also known as 'dense layer') after the two architectures, respectively. This FC layer firstly applies a linear transformation to the incoming data with 2560 input size and 256 out putsize. Then we replace the original ReLU to Mish as the activation function, and it is followed by a Dropout with p=0.5 and another linear transformation with the output size of 43.

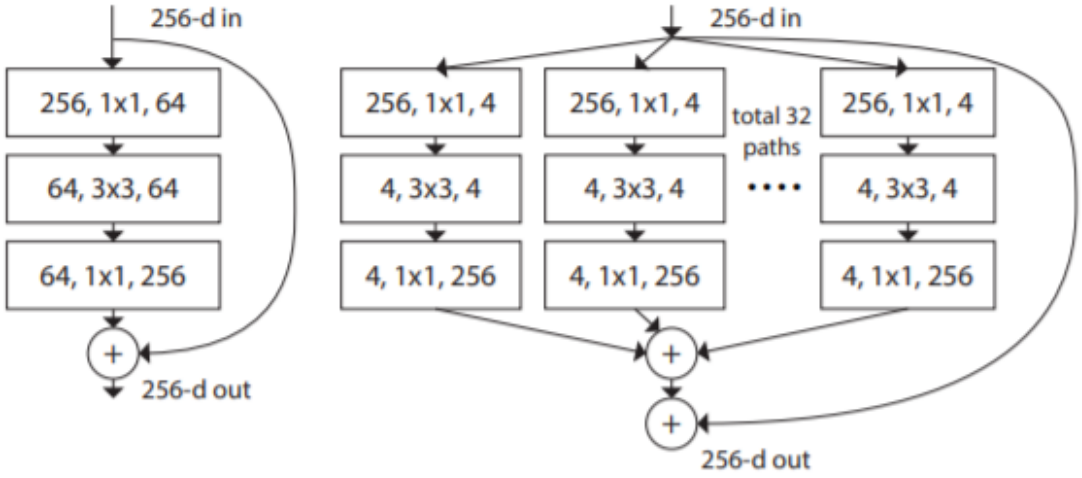

Figure 3: *(Left) A block of ResNet; (Right) A block in RexNeXt with Cardinality=32. A layer is shown as (in_channel, filter_size, out_channel). Cardinality is the size of the set of transformations.*

```
nn.Linear(2560, 256),
Mish(),
nn.Dropout(p=0.5),
nn.Linear(256, num_classes)
```

Our presented architecture is better performed with the activation function Mish. It is firstly introduced in "Mish: A Self Regularized Non-Monotonic Neural Activation Function" [3], and the network with Mish had an increase in F1-scoree by around 2% as compared to the same network. Being unbounded above, Mish has the desirable

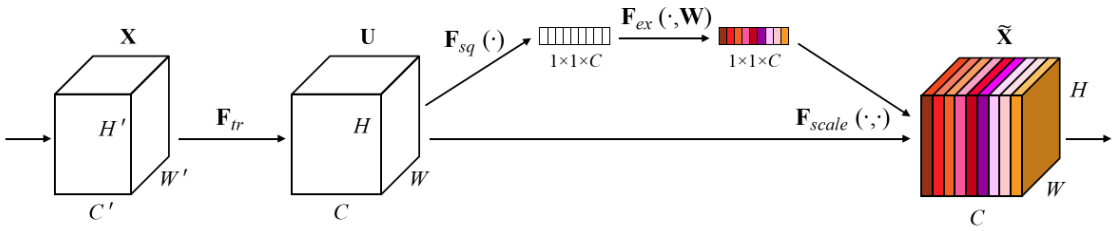

Figure 4: *Squeeze-and-Excitation (SE) Block*

property for any activation function, because it avoids saturation and increases the speed of training. Mish is also bounded below so that it has strong regularization advantageous. Moreover, Mish can improve expressivity and gradient flow as its non-monotonic property allows small negative inputs to be preserved as negative outputs. The order of continuity is also a benefit for Mish over ReLU, since it can efficiently avoid some undesired problems in gradient-based optimization.

The definition and graph (Figure 5) of Mish activation function are shown below:

$$f(x) = x \cdot tanh(ln(1 + e^x))$$

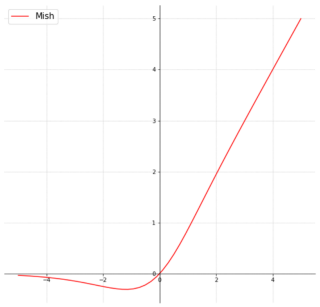

Figure 5: *Mish Activation Function*

1.3 Losses and Accuracy:

We use nn.BCEWithLogtsLoss() as our loss function.

$$Loss = \{l_1, l_2, \cdots, l_N\}, l_n = -w_n[y_n \cdot log\sigma(x_n) + (1 - y_n) \cdot log(1 - \sigma(x_n))]$$

The total loss is the loss from primary net.

1.4 Optimization Method:

The batch size we use for EfficientNet-B7 is 24 with accumulator=3, while for SE-ResNeXt101, we train both batch size 80 and 64 with accumulator=1. We use stochastic gradient descent (SGD) as the optimization of the model. SGD is one of

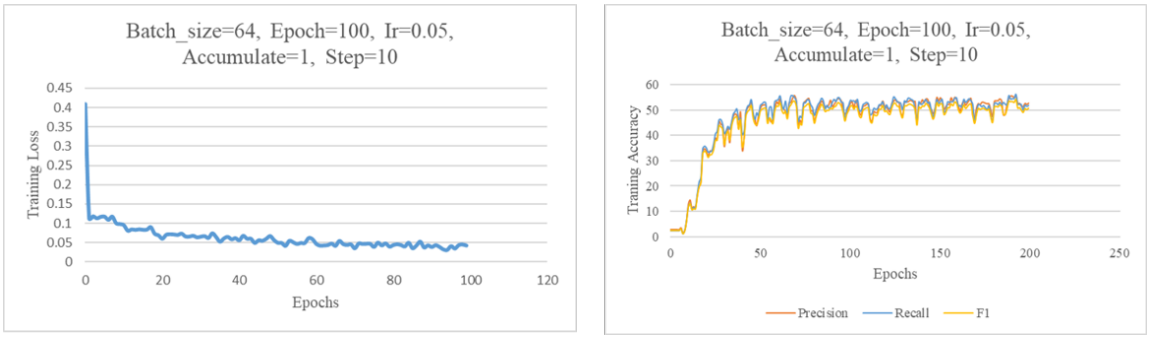

Figure 6: *(Left)Training Loss for EfficientNet-B7;(Right)Training Accuracy for EfficientNet-B7*

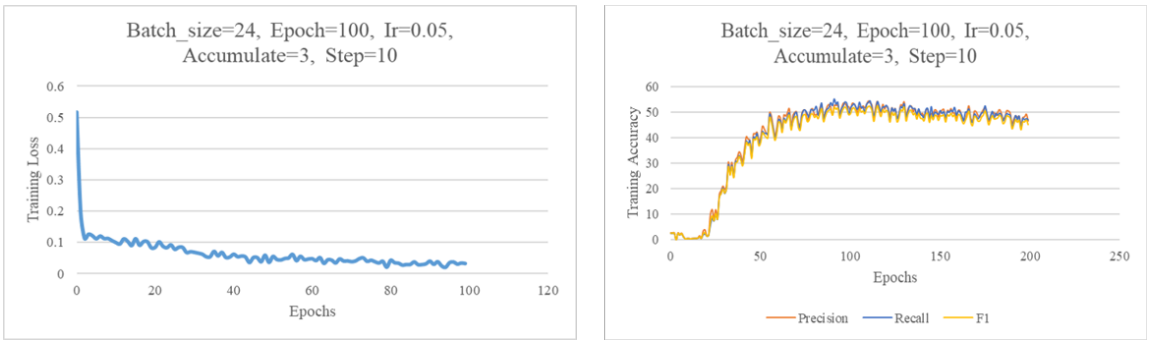

Figure 7: *(Left)Training Loss for SE-ResNeXt101;(Right)Training Accuracy for SE-ResNeXt101*

the most popular optimization methods, which update the parameters during each iteration after calculating the gradient of mini-batch. Yet it has some flaws, such as difficulties when choosing the proper learning rate and easily being stuck at the local optimal (saddle) point.

Therefore, we also use Adam (Adaptive Moment Estimation) [4] as the optimization method. Its advantages lie in the fact that after bias-correction, learning rate in each iteration has a specific range, which stabilized the parameters. In other words, it has different adaptive learning rates for different parameters.

Theoretically, Adam should have better performance, however, according to our training results, SDG has a better and faster convergence performance than Adam. Therefore, we choose SDG as our optimization method.

```
optimizer = optim.SGD(model.parameters(), lr=0.05, momentum=0.9,
    weight_decay=0.00005)
```

For learning rate scheduler, 'torch.optim.lr_scheduler' provides several methods to adjust the learning rate based on the number of epochs. In this case, we firstly choose CosineAnnealingLR to train the model.

CosineAnneaingLR (Figure 8) takes the cosine function as the cycle and reset the learning rate at the maximum value of each iteration (The initial learning rate is the maximum learning rate, here we have it as lr=0.05).

```
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max
    =150, eta_min=5e-6, last_epoch=-1)
```

where T_max stands for the number of iterations in one learning rate cycle, which means after T_max of epoch, learning rate is resetted. And eta_min is the minimum learning rate.

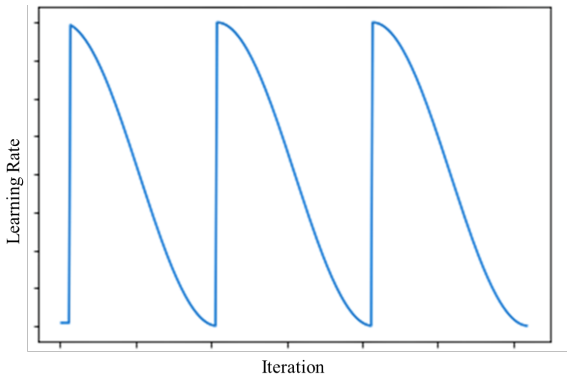

Figure 8: *Cosine Annealing LR Scheduler*

However, the cosine structure of this method leads to a bad performance on convergence. In other words, this method doesn't provide stable convergence and sometimes it still decreases after it already reaches to its peak. However, we can still use this fast converging method to find the approximate optimal.

Then, we pick another method called MultiStepLR, which can help adjust the learning rate in a set of time period (epoch). This method is suitable for the post-debugging and customize the time to adjust the learning rate for each experiment.

```
scheduler = torch.optim.lr_scheduler.MultiStepLR(optimizer, [5, 10, 20,
    40, 75], gamma=0.25)
```

where for milestones(list) is a list of time when each element adjust learning rate and gamma is the adjust parameter (here we set it as 0.25, which means decrease 25 times). For example, here we have 25 times decrease at the 5^{th} epoch, 10^{th} epoch, 20^{th} epoch, 40^{th} epoch and 75^{th} epoch, respectively.

The training is a Mixed Precision process, so we also import a training accelerator 'amp' from Apex. It contains two computing type: FP16 (Half-precision 16-bit floating-point) and FP32 (Single-precision 32-bit floating-point). In this case, we use the opt_level="O1", which means the computing method is automatically chosen during the training process.

```
model, optimizer = amp.initialize(model, optimizer, opt_level="O1",
    verbosity=0)
```

In order to increase the robustness of the model, we add an initial random seed at the beginning of each epoch. This step adds the randomness to the original pseudo-random number generated automatically by the computer, which means avoiding the generated random number being the optimal one.

```
def seed_everything(seed=SEED):
    random.seed(seed)
    os.environ['PYHTONHASHSEED'] = str(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    torch.backends.cudnn.deterministic = True
```

```
seed_everything(SEED+epoch)
```

1.5 Number of iterations/epochs of convergence:

$$Number\_of\_iterations = \frac{epochs\_of\_convergence \times the\_amount\_of\_data}{batch\_size \times accumulate}$$

For EfficientNet-B7, the model is converged roughly around 87 and the number of iteration is $\frac{87 \times 4752}{24 \times 3} = 5742$.

For SE-ResNeXt101, the model is converged roughly around 44 and the number of iteration is $\frac{44 \times 4800}{64} = 3300$.

1.6 Hyperparameters:

Table 1: Hyperparameters

Model	Batch size	Learning rate	Accumulate
EfficientNet-B7	24	0.05	3
SE-ResNeXt101	64	0.05	1

2 Summary

2.1 Novelty

The novelty of our method lies in several aspects.

Firstly, it is important to select a proper backbone for this problem based on journal paper and experience. We also customize the FC Layer of the architecture so that the model can well adjust the problem.

Secondly, we apply different data augmentation method to increase the diversity of the data and performance of the model. In spatial level, we use rotation and horizontal flip. And for pixel level, we choose image contrast augmentation (gamma transform, color saturation change and brightness change), image blurring (motion blur and median blur) and coarse dropout (Figure 9). These enhance the robustness of the model.

Thirdly, we propose several methods to smooth the training process. In this part, we select the proper optimizer (SGD) based on experiment. And we use the learning rate scheduler to help adjust learning rate during the training process, which can increase the convergence speed to some extent. It is a two stage process for training when using different scheduler. Also, seed-everything enables randomness of the generated seed, and thus also enhance the robustness of our model. More importantly, we apply

Figure 9: *Coarse Dropout*

the concept of the accumulator, which is equivalent to updating the weights in every $batch\_size \times accumulate\_number$ during each backpropogation updating process. This is a good way to reach appropriate batch size.

2.2 Things don't work

According to your experiment, Too many dense layers may decrease the performance of precision and F1 score accordingly.

Some data augmentation methods are not suitable in this situation, for instance, vertical flip and image transpose, as the generated images make no sense in recognizing the action. Also, image sharpening, which means compensate the outline of the image and enhance the edge of the image, is also unrelated to the action capturing.

As for the training method, it is shown from the experiment that Adam doesn't work so well even though theoretically it should be. And if we use too big batch size or accumulator, this model might not show good performance as well. Similarly, too big or too small learning rate may cause bad convergence performance or even no convergence.

2.3 Performance Results and Comparison

Our all finetured results are shown in Table 2 below:

Table 2: Performance Comparison

Model	Train Set			Validation Set		
	Precision	Recall	F1	Precision	Recall	F1
Resnet34	/	/	/	48.4	46.0	45.8
PNANet 5 large*	/	/	/	51.3	54.3	50.7
Efficientnet-B7	79.2	77.3	77.1	54.1	54.0	52.4
SE-ResNeXt101	71.7	71.8	71.2	58.9	59.8	57.5
Efficientnet-B7 (Mish Aug)	78.9	75.7	76.0	56.3	58.9	56.1
SE-ResNeXt101 (Mish Aug)	84.8	82.8	83.0	60.6	61.1	59.2

(* indicate not finetune)

2.4 Future Work

We could try new models and activation methods and meanwhile analyze why SE-ResNeXt shows better performance than Efficientnet-B7 in multi-label classification. It will also be a good choice if we blend models and see the generated performance. There are also a few data augmentation methods that we would like to try: large size input and other image filter. Moreover, we could combine the two learning rate into one. There is also, still an unsolved puzzle for the fact that Adam doesn't work well in this problem, so we would like to make a deep dive into this problem.

References

[1] Mingxing Tan and Quoc V Le. Efficientnet: Rethinking model scaling for convolutional neural networks. *arXiv preprint arXiv:1905.11946*, 2019.

[2] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1492–1500, 2017.

[3] Diganta Misra. Mish: A self regularized non-monotonic neural activation function. *arXiv preprint arXiv:1908.08681*, 2019.

[4] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. international conference on learning representations (2015), 2015.