**Deep Learning** 2024011303 Liu Hanzuo

# CP2 Report

## Training Techniques

**Model Structure** : from `Dropout` to `NoDropout` models, I only remove the dropout layers in the convolution layers. Which is mentioned in class – a dropout layer should not follows a BatchNorm layer.
From `Vgg` models to other models, I add residual connection layers, using a "same" padding and a kernel size of 3 (like DenseNet), leading to a roughly 1% improvement. (Since the model is not deep enough, the residual connection does not have a significant improvement.).
From `Deep` models to `Dropout` models, I change the connection ways: In the `Deep` architecture, there are 2 Resnets between convs: res1 and res2, the shedule is like:

$$x \to y = res1(x) + x \to res2(y) + y$$

However, this structure does not works well due to limited model size. Thus I change the structure to:

$$x \to y = \text{cat}[res1(x), res2(x)]$$

To describe it simply, the model's width also means much when the model's size is limited. Moreover, I use a linear layer for last several layer's result (such as project them to a 32/64-dimension space), after concat them, we finally use a linear layer to get the final result. After checking this idea in CNN-relative papers, I find out that this idea is similar to ResNeXt, which improve the model's performance for about 1%.

**Data Augmentation** : The following data augmentation techniques were applied during training to improve the model's robustness and performance:

- **Random Horizontal Flip**: Randomly flips the image horizontally.
- **Random Crop**: Crops the image to 128x128 pixels with a padding of 4 pixels.
- **Random Rotation**: Rotates the image by a random angle up to 30 degrees.
- **Color Jitter**: Randomly changes the brightness, contrast, saturation, and hue of the image.
- **Random Grayscale**: Converts the image to grayscale with a probability of 0.2.

- **Random Affine**: Applies random affine transformations with translation up to 25% of the image size.
- **Random Perspective**: Applies random perspective transformations with a distortion scale of 0.3 and a probability of 0.5.
- **Normalization**: Normalizes the image with mean [125/255, 124/255, 115/255] and standard deviation [60/255, 59/255, 64/255].
- **MixUp**: Use torchvision's MixUp augmentation with a alpha of 0.1→0.5.

In fact, data augmentation is a good way to prevent model from overfitting. However, the model's size is limited, thus the data augmentation could not be setted too hard (like RandomResizedCrop, RandomPerspective, etc.). Otherwise, the model will not converge. These parameters are setted after several tries.

# Specific Model structures

### ResBlock

```
def createManyResBlock(self, channels=64, BlockNum=3, kernel_size=3):
    self.cnt += 1
    manyResBlock = []
    for i in range(BlockNum):
        x = nn.Sequential(
            nn.Conv2d(channels, channels, kernel_size, padding=(kernel_size-1)//2),
            nn.BatchNorm2d(channels),
            nn.SiLU(),
            nn.Dropout2d(0.15 if channels < 128 else 0.25),
            nn.Conv2d(channels, channels, kernel_size, padding=(kernel_size-1)//2),
        )
        self.add_module(f'{self.cnt}_{i}', x)
        manyResBlock.append(x)
    return manyResBlock

def PassThrough(self, manyResBlock: list, x):
    for i in range(len(manyResBlock)):
        x = F.mish(x + manyResBlock[i](x))
        if i % 2:
            x = nn.Dropout2d(0.1)(x)
    return x
```

Here we show the ResBlock structure. The ResBlock is a basic block in the model. It is a combination of two convolution layers, with a BatchNorm layer and a SiLU activation

layer. The dropout layer is added after the first. In the model architecture, we only change the hyperparameter in the function.

**Model Architecture** Notation: In the following graph, here are some definitions.

1. inc: input channels, outc: output channels, c: channels in ResBlock, ks: kernel size, ps: padding size, s: stride size
2. bn: block number, dr: dropout rate
3. Concat: Concatenate the two inputs, ResConnect: enumerate before, pass the input through Resblocks consequently.

Table 1: Model Architecture Layers

| **Layers** | **Basic Configs** | **Vgg** | **Deep** | **Dropout** | **NoDropout** |
|---|---|---|---|---|---|
| Conv1 | inc=3, outc=64, ks=7, ps=3 | | | | |
| BatchNorm | c=64 | | | | |
| Maxpool | ks=2 | | | | |
| Dropout | dr=0.15 | | | | dr=0.0 |
| ResBlock11 | c=64, ks=3, bn=3 | x | | | |
| ResBlock12 | c=64, ks=5, bn=3 | x | ks=3 | | |
| Connection | | x | ResConnect | Concat | Concat |
| Conv2 | inc=outc=128, ks=3, ps=1, s=2 | | | | |
| BatchNorm | c=128 | | | | |
| Maxpool | ks=2 | | | | |
| Dropout | dr=0.25 | | | | dr=0.0 |
| ResBlock2 | c=128, ks=3, bn=5 | | | | |
| Conv3 | inc=128, outc=128, ks=5, ps=2 | | | | |
| BatchNorm | c=128 | | | | |
| Maxpool | ks=2 | | | | |
| Dropout | dr=0.25 | | | | dr=0.0 |
| ResBlock3 | c=128, ks=3, ps=1 | | | | |
| Conv4 | inc=128, outc=64, ks=3, ps=1 | | | | |
| BatchNorm | c=64 | | | | |
| Maxpool | ks=2 | | | | |
| Dropout | dr=0.25 | | | | dr=0.0 |
| ResBlock41 | c=64, ks=3, ps=1 | | | | |
| Linear | inc=64, outc=256 | | | | |
| BatchNorm | c=256 | | | | |
| GELU | | | | | |
| Dropout | dr=0.3 | | | | |

Continued on next page

**Table 1 – continued from previous page**

| Layers | Basic Configs | Vgg | Deep | Dropout | NoDropout |
|---|---|---|---|---|---|
| Linear | inc=256, outc=256 | | | | |
| BatchNorm | c=256 | | | | |
| GELU | | | | | |
| Dropout | dr=0.2 | | | | |
| Linear | inc=256, outc=10 | | | | |

# Training Curves

# Appendix

In this report, I will mainly enumerate about three model structures(also introducing some failure examples) and list their result datas.

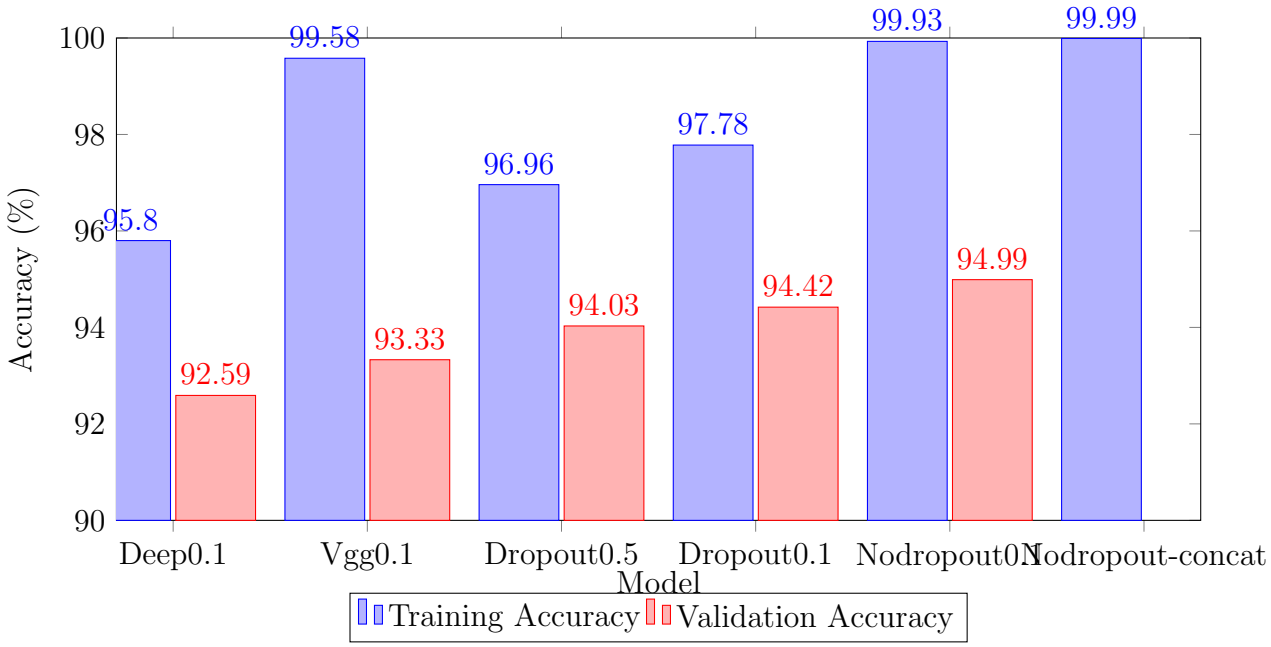


Figure 1: Comparison of Model Architectures

**Settings** The similarity between these models are convolution layers. Basically, we utilize `channels=64` and `channels=128`. Except for Vgg net, we utilize Resnet configuration, with "same" paddings. Also, we add a Maxpooling after each individual convolution layer(but not after residual layers.) Thus we could extract each range of features in the photo(similar to DenseNet). A Batch Normalization is added after each layer. Activation layers varies from SiLU, GELU, LeakyReLU (but does not affect the result a lot).

**Unsuccessful Tries – MoE** At the beginning, I tries to utilize MoE architecture to train the model. However, We mainly faces the following questions:

- Total Model size is limited by 5M, thus each expert's size is limited, leading to a poor performance.
- Classification tasks is not suitable for MoE, MoE has its advantage for a faster inference speed and tasks that could be seperated to different experts. Leading to a better performance in regression tasks.
- Need a extra gate to determine which expert to use, leading to a slower training speed.

**Conclusion** After implementing this coding project, some experience is gained:

- Architecture is important, especially need to be carefully chosen for a specific task and a limited model size.
- Data Augmentation is also important, but should not be setted too hard, in case of the poor ability of model to converge.
- The model's size is limited, thus the model's width also means much.
- Some tricks (such as no dropout after BatchNorm, etc.) should be considered.
- for a model that is not deep enough, residual connection does not have a significant improvement. However, it indeed increase the model's performance, and make the training progress more stable.

Afterall, the model's best performance on valid set is **94.99%**.