National Tsing Hua University

Fall 2023 11210IPT 553000

Deep Learning in Biomedical Optical Imaging

Report

CHUN-HAO LI[1]

1Ultrafast Photonics Lab, Institute of Photonics Technologies, Tsing Hua University.

Student ID:112066513

1. Complete the code for training and testing

When we first opened the report task code, it was incomplete, so I first chose one layer of convolution + one layer of fully connected layer of training neural network, and I didn't implement data augmentation in the first 4 training processes. While for the testing part I just simply load the training results and do what we usually do. So, next I'm going to dive deep into the algorithm I used for improving training as well as testing results.

2. Convolution Neural Network

2.1 *One Layer Convolutional Neural Network*

As I mentioned above, for the starting I chose simple one layer CNN as training, and the purpose is to see how complicated the dataset is. After training, the result is down below:

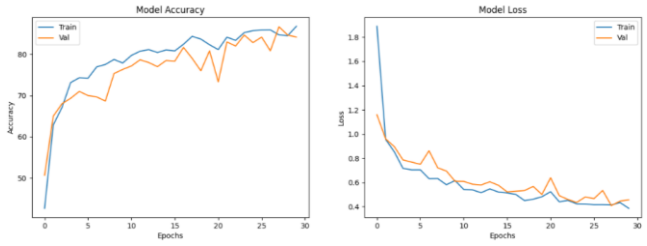


Fig.1. One Layer CNN

This seems quite good at the first glance, but you can find that the best accuracies aren't greater than 90%, although accuracies and losses seem just right, so I think there's still room for improvement. And the testing result proved what I thought, it was 84.83%

2.2 *Multiple Layers Convolution Neural Network*

Next, I tried even deeper CNN. For three layers, each layer I implement two convolutions with one batch normalization in between and one max pooling at the end, the parameters I make it larger then decrease like expand them then shrink to original, the code is in Fig.2.. And the results are shown below (Fig.3.).

```
self.net = nn.Sequential(
        nn.Conv2d(3, 32, kernel_size=3, stride=1, padding='same'),
        nn.BatchNorm2d(32),
        nn.ReLU(),
        nn.Conv2d(32, 64, kernel_size=3, stride=1, padding='same'),
        nn.BatchNorm2d(64),
        nn.ReLU(),
        nn.MaxPool2d(kernel_size=2, stride=2),

        nn.Conv2d(64, 128, kernel_size=7, stride=1, padding='same'),
        nn.BatchNorm2d(128),
        nn.ReLU(),
        nn.Conv2d(128, 256, kernel_size=7, stride=1, padding='same'),
        nn.BatchNorm2d(256),
        nn.ReLU(),
        nn.MaxPool2d(kernel_size=2, stride=2),

        nn.Conv2d(256, 64, kernel_size=5, stride=1, padding='same'),
        nn.BatchNorm2d(64),
        nn.ReLU(),
        nn.Conv2d(64, 32, kernel_size=5, stride=1, padding='same'),
        nn.BatchNorm2d(32),
        nn.ReLU(),
        nn.MaxPool2d(kernel_size=2, stride=2),

        nn.AdaptiveAvgPool2d(1),
        nn.Flatten(),
        nn.Linear(32, num_classes)
)
```

Fig.2. Deeper Convolution Network

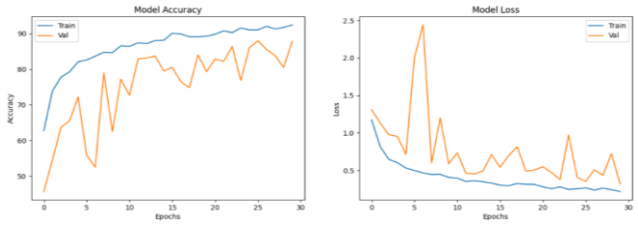


Fig.3. Two Layer CNN

When I compared with the previous work, it has slightly been improved, the testing result improved to 89.5%. But as I can see, the result is still unsatisfactory to me.

2.3 *Handcraft ResNet*

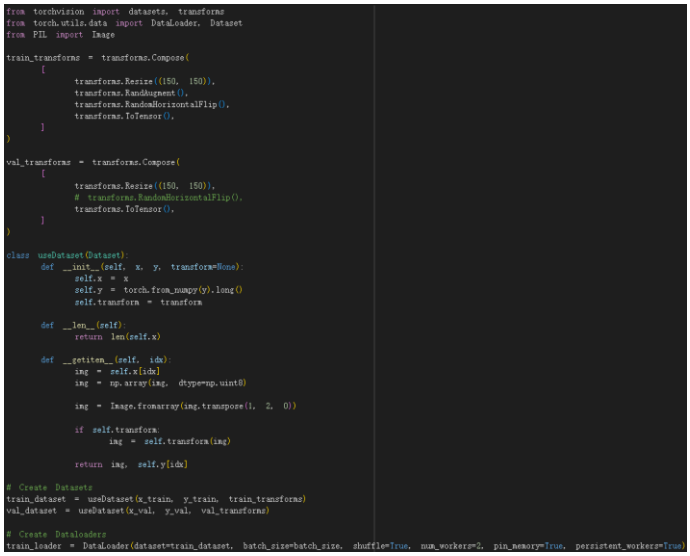
So, I implemented a handcraft Resnet from a website I found, because in my perspective, this model is built by someone else, absolutely much more complex than my build, and is a champion of a competition. So I trusted it would make a difference. I implemented a Resnet50 model. I encountered numerous errors while tuning this model to fit my training needs.

已註解 [1]: 引用

After I finished building this model, I trained for several times. But each time I got similar results: the best training accuracy is around 85%~92%, ant validation accuracy is around 82%~85%, which seems overfitting to me. And the testing result verified my suspect: around 80%~85%, lower than the convolutional network I built.

3. Data Augmentation

So, then I started thinking, was this the trouble of lack of database? And then I added the data augmentation method to expand my database.



```
from torchvision import datasets, transforms
from torch.utils.data import DataLoader, Dataset
from PIL import Image

train_transforms = transforms.Compose(
    [
        transforms.Resize((150, 150)),
        transforms.RandAugment(),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
    ]
)

val_transforms = transforms.Compose(
    [
        transforms.Resize((150, 150)),
        # transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
    ]
)

class useDataset(Dataset):
    def __init__(self, x, y, transform=None):
        self.x = x
        self.y = torch.from_numpy(y).long()
        self.transform = transform

    def __len__(self):
        return len(self.x)

    def __getitem__(self, idx):
        img = self.x[idx]
        img = np.array(img, dtype=np.uint8)

        img = Image.fromarray(img.transpose(1, 2, 0))

        if self.transform:
            img = self.transform(img)

        return img, self.y[idx]

# Create Datasets
train_dataset = useDataset(x_train, y_train, train_transforms)
val_dataset = useDataset(x_val, y_val, val_transforms)

# Create Dataloaders
train_loader = DataLoader(dataset=train_dataset, batch_size=batch_size, shuffle=True, num_workers=2, pin_memory=True, persistent_workers=True)
```

Fig.4. Data Augmentation

After some time of tuning the method, it is successfully functional in my code. I looked into the result after first training, and I found I was facing a huge problem: validation is better than training (this training was using ResNet50 model):

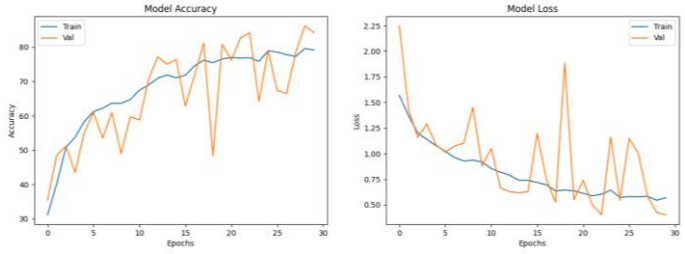


Fig.5. Data Augmentation

As you can see, this isn't anywhere near "better result", and I realized this is due to the validation dataset being simpler than the training dataset. So, I added "resize" to the validation dataset, and had another train. But the result was the same. I even got the same result for my convolutional network. Because I lacked experience for tuning data augmentation, I couldn't find a way to improve the training. Then I decided to put this method down, and started thinking if my convolutional network is too complex, that leads to overfitting.

4. Downgrade Convolutional Network

So, I started to modify the convolutional network, I modified the channels less than they were. And after a training process, this verified the thought.

```
# Convolutional Network
class ConvModel(nn.Module):
    def __init__(self):
        super().__init__()

        self.net = nn.Sequential(
            nn.Conv2d(3, 32, kernel_size=3, stride=1, padding='same'),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=3, stride=1, padding='same'),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),

            nn.Conv2d(64, 64, kernel_size=7, stride=1, padding='same'),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.Conv2d(64, 128, kernel_size=7, stride=1, padding='same'),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),

            nn.Conv2d(128, 64, kernel_size=5, stride=1, padding='same'),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.Conv2d(64, 64, kernel_size=5, stride=1, padding='same'),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),

            nn.AdaptiveAvgPool2d(1),
            nn.Flatten(),
            nn.Linear(64, num_classes)
        )
    def forward(self, x):
        x = self.net(x)
        return x
```

Fig.6. Modified Convolutional Network

The best training accuracy reaches 91.25%, training loss 0.2219; validation accuracy 90.5%, validation loss 0.2710. And for testing, I got 91.33%.

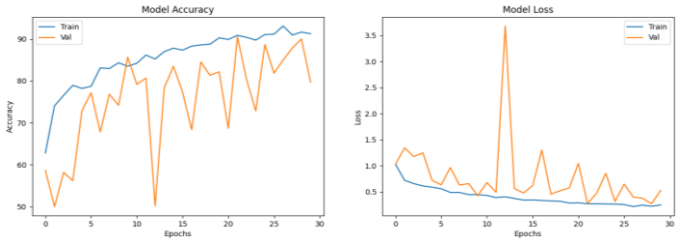


Fig.7. Result

5. Result and Discussion

Seeing the improvement of accuracy is really a cheerful moment, although it might be just a little bit of improving, it still uncovered a way for further improvement. In the process of making this report, I overcame many difficulties, but just like climbing a tall mountain, there will be twists and turns. And I learnt a lot from this report.