Machine Learning Final Project

Group 1 : 110511109李崇碩、110511277蔡東宏

2024/06/21

Outline

- Data Augmentation
- Model Choosing
- ShuffleNet Model
- Final Model
- Batch Size
- Optimizer Choosing
- Dynamic adjusting Learning Rate
- Reference

Data Augmentation

- **Data Augmentation is used to avoid overtraining**
  - **Resize (256 x 256)**
  - **RandomHorizontalFlip、RandomVerticalFlip**
  - **RandomRotation**
  - **RandomErasing**
  - **ColorJitter (未採用)**
  - **Normalize**

李崇碩、蔡東宏

Model Choosing

- **Comparing Different Model (parameters, FLOPs, Testing Accuracy)**

Model	Parameter	FLOPs	Testing Accuracy
ResNet18	11.7M	2.382G	0.9266
ResNet50	25.56M	5.4G	0.9266
ShuffleNet_v2	1.367M	57.9M	0.8349
EfficientNet	5.289M	641.64G	0.9083
MobileNet_v2	3.505M	427.346G	0.9174
VGG16	138.36M	20.168G	0.9174
VGG19	143.667M	25.604G	0.9174
DenseNet121	7.98M	3.78G	0.9266

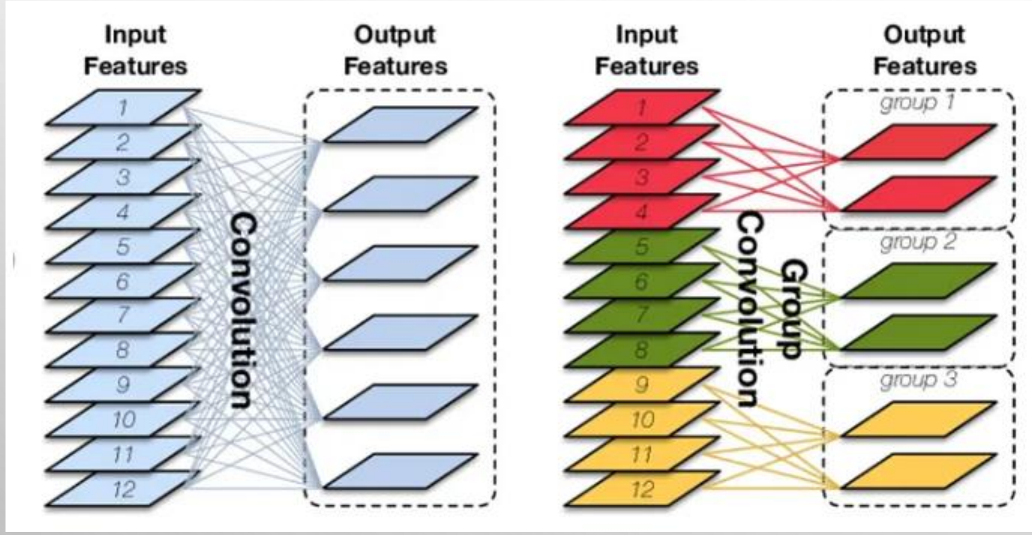
Model Choosing

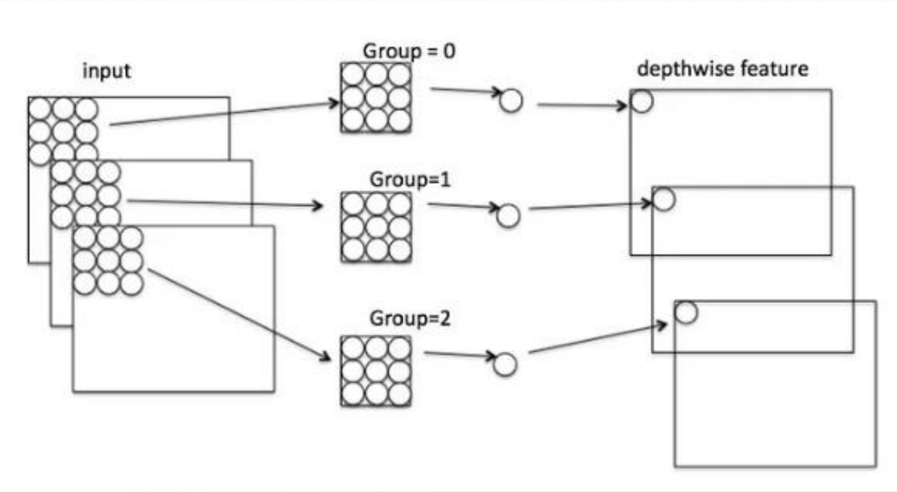
- **Without Modifying ShuffleNet_v2**
  - **Parameters : 1.376M 、 FLOPs : 57.9M、Testing Accuracy : 0.8349**

- **Modifying ShuffleNet_v2**
  - **Removing the latest two layers**
  - **Adding two full connected layers**
  - **Adding some dropout to avoid overtraining**
  - **Parameters : 0.143M 、 FLOPs : 44M、Testing Accuracy : 0.9**

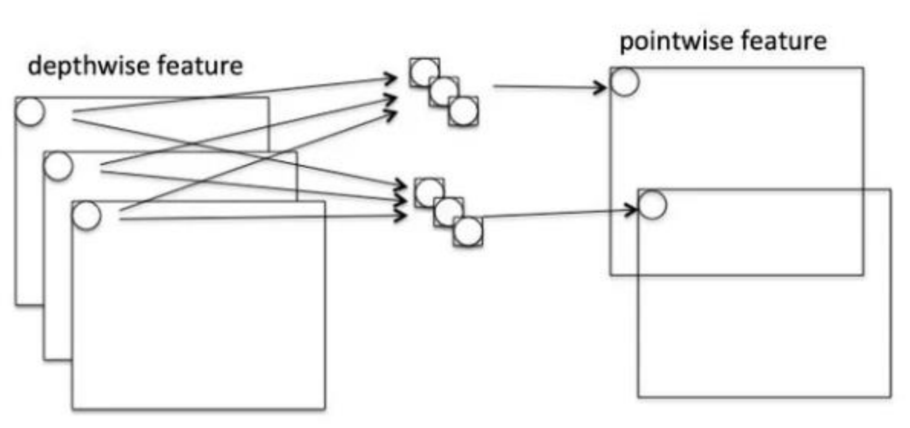
ShuffleNet Model

- Design concept
  - Group Convolution
  - Depthwise and Pointwise Convolution





上圖為depthwise feature，下圖為pointwise feature



李崇碩、蔡東宏

ShuffleNet V1 Model

- ShuffleNet V1 is design based on the bottleneck of ResNet.
- Using group Convolution and Channel Shuffle to reduce computation and parameter size.
- Channel Shuffle allows information exchange between channels

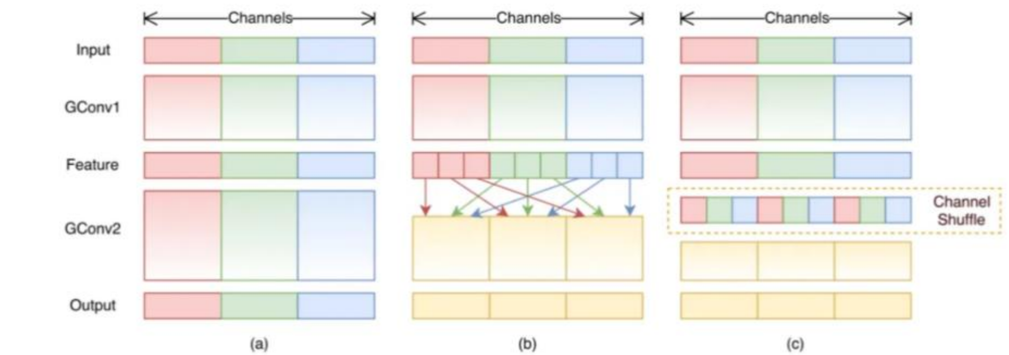


Figure 1. Channel shuffle with two stacked group convolutions. GConv stands for group convolution. a) two stacked convolution layers with the same number of groups. Each output channel only relates to the input channels within the group. No cross talk; b) input and output channels are fully related when GConv2 takes data from different groups after GConv1; c) an equivalent implementation to b) using channel shuffle.

Channel Shuffle示意圖

李崇碩、蔡東宏

ShuffleNet V1 VS ShuffleNet V2

- Computational efficiency: ShuffleNet_V2 > ShuffleNet_V1.
- ShuffleNet_V2 is simpler and more.
- ShuffleNet_V2 introduces new computational units (such as SE modules)
- ShuffleNet_V2 allocates channels differently based on actual computational characteristics.

Self-designed Model

nn.Conv2d:

- **Function:** Effectively extracts local features from images. Stacking multiple convolutional layers can progressively extract higher-level features.
- **Design Reason:** After each convolutional layer, the number of channels (features) gradually increases (from 3 (RGB) to 32, 32 to 64, and then to 128). This is because we want to capture more complex features as the layers deepen.

nn.ReLU:

- **Function:** The ReLU function introduces non-linearity, allowing the model to learn more complex features.
- **Design Reason:** Avoid the limitations of a linear model. Additionally, ReLU is computationally efficient and helps solve the vanishing gradient problem.

```python
class SimpleCNN(nn.Module):
    def __init__(self, num_classes=2):
        super(SimpleCNN, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 32, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(32, 64, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(64, 128, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
        self.classifier = nn.Sequential(
            nn.Dropout(p=0.5),
            nn.Linear(128 * 6 * 6, 200),  # Adjusted for
            nn.ReLU(inplace=True),
            nn.Dropout(p=0.5),
            nn.Linear(200, num_classes)
        )

    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), -1)
        x = self.classifier(x)
        return x
```


李崇碩、蔡東宏

Self-designed Model

nn.MaxPool2d:

•**Function:** Reduces the size of the feature map by taking the maximum value within a window, preserving the most important features.

•**Design Reason:** Reduce computational load and the number of parameters. Through downsampling , they make the feature map more invariant and increase the receptive field.

nn.Dropout:

•**Function:** Randomly drops some neurons during training.

•**Design Reason:** prevent overfitting and improve the model's generalization ability.

```
class SimpleCNN(nn.Module):
    def __init__(self, num_classes=2):
        super(SimpleCNN, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 32, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(32, 64, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(64, 128, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
        self.classifier = nn.Sequential(
            nn.Dropout(p=0.5),
            nn.Linear(128 * 6 * 6, 200),  # Adjusted for
            nn.ReLU(inplace=True),
            nn.Dropout(p=0.5),
            nn.Linear(200, num_classes)
        )

    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), -1)
        x = self.classifier(x)
        return x
```


李崇碩、蔡東宏

Self-designed Model

■**Result :**

```
FLOPs: 24146320.0, Params: 1015450.0
```

■ **Conclusion :** found that even with a simple method without stacking many layers, the parameters was already suboptimal. Therefore, We decided not to continue testing it and instead replaced it with ShuffleNet.

```
Epoch 53/100, Loss: 0.2989, Accuracy: 0.8786, Test Loss: 0.6106, Test Accuracy: 0.7167
Epoch 54/100, Loss: 0.2693, Accuracy: 0.8946, Test Loss: 0.6460, Test Accuracy: 0.6917
Epoch 55/100, Loss: 0.2715, Accuracy: 0.8911, Test Loss: 0.6245, Test Accuracy: 0.6917
Epoch 56/100, Loss: 0.2708, Accuracy: 0.8911, Test Loss: 0.6230, Test Accuracy: 0.7250
Epoch 57/100, Loss: 0.2830, Accuracy: 0.8804, Test Loss: 0.6079, Test Accuracy: 0.7167
Epoch 58/100, Loss: 0.2778, Accuracy: 0.8821, Test Loss: 0.6180, Test Accuracy: 0.6833
Epoch 59/100, Loss: 0.2763, Accuracy: 0.9000, Test Loss: 0.6277, Test Accuracy: 0.7000
Epoch 60/100, Loss: 0.2850, Accuracy: 0.9036, Test Loss: 0.6211, Test Accuracy: 0.7083
Epoch 61/100, Loss: 0.2715, Accuracy: 0.8946, Test Loss: 0.6094, Test Accuracy: 0.6917
Epoch 62/100, Loss: 0.2768, Accuracy: 0.8946, Test Loss: 0.5993, Test Accuracy: 0.7083
Epoch 63/100, Loss: 0.2744, Accuracy: 0.8982, Test Loss: 0.6210, Test Accuracy: 0.7000
Epoch 64/100, Loss: 0.2795, Accuracy: 0.8857, Test Loss: 0.6162, Test Accuracy: 0.7083
Epoch 65/100, Loss: 0.2695, Accuracy: 0.8946, Test Loss: 0.6280, Test Accuracy: 0.7000
Epoch 66/100, Loss: 0.2693, Accuracy: 0.9036, Test Loss: 0.6373, Test Accuracy: 0.6917
Epoch 67/100, Loss: 0.2821, Accuracy: 0.8911, Test Loss: 0.6073, Test Accuracy: 0.7000
Epoch 68/100, Loss: 0.2933, Accuracy: 0.8821, Test Loss: 0.6103, Test Accuracy: 0.7083
Epoch 69/100, Loss: 0.2852, Accuracy: 0.8839, Test Loss: 0.6318, Test Accuracy: 0.7083
Epoch 70/100, Loss: 0.2856, Accuracy: 0.8768, Test Loss: 0.6266, Test Accuracy: 0.7000
Epoch 71/100, Loss: 0.2726, Accuracy: 0.8911, Test Loss: 0.6260, Test Accuracy: 0.7000
Epoch 72/100, Loss: 0.2694, Accuracy: 0.8893, Test Loss: 0.6320, Test Accuracy: 0.7000
Epoch 73/100, Loss: 0.2616, Accuracy: 0.9000, Test Loss: 0.6227, Test Accuracy: 0.7167
Epoch 74/100, Loss: 0.2653, Accuracy: 0.8982, Test Loss: 0.6032, Test Accuracy: 0.7167
Epoch 75/100, Loss: 0.2762, Accuracy: 0.8929, Test Loss: 0.6217, Test Accuracy: 0.7000
Epoch 76/100, Loss: 0.2660, Accuracy: 0.9018, Test Loss: 0.5976, Test Accuracy: 0.7333
Epoch 77/100, Loss: 0.2811, Accuracy: 0.8982, Test Loss: 0.6120, Test Accuracy: 0.7083
Epoch 78/100, Loss: 0.2861, Accuracy: 0.9089, Test Loss: 0.6007, Test Accuracy: 0.7167
Epoch 79/100, Loss: 0.2730, Accuracy: 0.9000, Test Loss: 0.6171, Test Accuracy: 0.7167
Epoch 80/100, Loss: 0.2798, Accuracy: 0.8946, Test Loss: 0.6179, Test Accuracy: 0.7167
Epoch 81/100, Loss: 0.2676, Accuracy: 0.9000, Test Loss: 0.5936, Test Accuracy: 0.7083
Epoch 82/100, Loss: 0.2839, Accuracy: 0.8821, Test Loss: 0.6056, Test Accuracy: 0.7083
Epoch 83/100, Loss: 0.2705, Accuracy: 0.8964, Test Loss: 0.6238, Test Accuracy: 0.7250
Epoch 84/100, Loss: 0.2702, Accuracy: 0.8946, Test Loss: 0.5975, Test Accuracy: 0.7083
Epoch 85/100, Loss: 0.2748, Accuracy: 0.9036, Test Loss: 0.5957, Test Accuracy: 0.7000
Epoch 86/100, Loss: 0.2780, Accuracy: 0.8982, Test Loss: 0.6026, Test Accuracy: 0.7167
Epoch 87/100, Loss: 0.2757, Accuracy: 0.8964, Test Loss: 0.6325, Test Accuracy: 0.7000
Epoch 88/100, Loss: 0.2872, Accuracy: 0.8911, Test Loss: 0.6124, Test Accuracy: 0.7250
Epoch 89/100, Loss: 0.2577, Accuracy: 0.9107, Test Loss: 0.6264, Test Accuracy: 0.6917
Epoch 90/100, Loss: 0.2671, Accuracy: 0.8893, Test Loss: 0.6187, Test Accuracy: 0.7083
Epoch 91/100, Loss: 0.2709, Accuracy: 0.8982, Test Loss: 0.6158, Test Accuracy: 0.7167
Epoch 92/100, Loss: 0.2562, Accuracy: 0.8929, Test Loss: 0.6230, Test Accuracy: 0.7083
Epoch 93/100, Loss: 0.2725, Accuracy: 0.9071, Test Loss: 0.6152, Test Accuracy: 0.7083
Epoch 94/100, Loss: 0.2688, Accuracy: 0.8964, Test Loss: 0.6153, Test Accuracy: 0.7167
Epoch 95/100, Loss: 0.2740, Accuracy: 0.9000, Test Loss: 0.6295, Test Accuracy: 0.6917
Epoch 96/100, Loss: 0.2821, Accuracy: 0.8982, Test Loss: 0.6078, Test Accuracy: 0.7250
Epoch 97/100, Loss: 0.2834, Accuracy: 0.8804, Test Loss: 0.6199, Test Accuracy: 0.7083
Epoch 98/100, Loss: 0.2771, Accuracy: 0.8875, Test Loss: 0.6018, Test Accuracy: 0.7250
```


李崇碩、蔡東宏

Final Model

- Removing the latest two layer of ShuffleNet_v2 Model
- Adding global average pool to reduce the parameters of model
- Adding a dropout layer with a probability of 0.2 to randomly drop some neurons

```python
class Network(nn.Module):
    def __init__(self, num_classes=2):
        super(Network , self).__init__()
        self.model = torchvision.models.shufflenet_v2_x0_5(pretrained=True)

        self.model = nn.Sequential(*list(self.model.children())[:-2])
        self.model.add_module('global_avg_pool', nn.AdaptiveAvgPool2d(1))
        self.model.add_module('flatten', nn.Flatten())
        self.model.add_module('dropout', nn.Dropout(p=0.2))
        self.model.add_module('fc', nn.Linear(in_features=192, out_features=2))


        self.dropout = nn.Dropout(p=0.02)
        self.fc = nn.Linear(in_features=2, out_features=num_classes)

    def forward(self, x):
        x = self.model(x)
        x = self.dropout(x)
        x = self.fc(x)
        return x
```


李崇碩、蔡東宏

Final Model

- **Adding a full connected layer (in_features = 192, out_features = 2)**

- **Adding a dropout layer with a probability of 0.02 to randomly drop some neurons**

- **Adding a full connected layer (in_features = 2, out_features = 2)**

```python
class Network(nn.Module):
    def __init__(self, num_classes=2):
        super(Network , self).__init__()
        self.model = torchvision.models.shufflenet_v2_x0_5(pretrained=True)

        self.model = nn.Sequential(*list(self.model.children())[:-2])
        self.model.add_module('global_avg_pool', nn.AdaptiveAvgPool2d(1))
        self.model.add_module('flatten', nn.Flatten())
        self.model.add_module('dropout', nn.Dropout(p=0.2))
        self.model.add_module('fc', nn.Linear(in_features=192, out_features=2))


        self.dropout = nn.Dropout(p=0.02)
        self.fc = nn.Linear(in_features=2, out_features=num_classes)

    def forward(self, x):
        x = self.model(x)
        x = self.dropout(x)
        x = self.fc(x)
        return x
```


李崇碩、蔡東宏

Calculate Performance

- Use the profile function from thop package to calculate FLOPs and parameters.
- Additionally, use the summary function from torchsummary package to print out the parameter count, layer type, and output shape of each layer.

```
from thop import profile
from torchsummary import summary
model_for_profiling = copy.deepcopy(model)
# Profile the cloned model without modifying the original model's state_dict
model_for_profiling.eval()  # Set the cloned model to evaluation mode
with torch.no_grad():
    flops, params = profile(model_for_profiling, inputs=(random_tensor,), verbose=False)
print(f"FLOPs: {flops}, Params: {params}")
summary(model,(3,256,256))
```

Seed

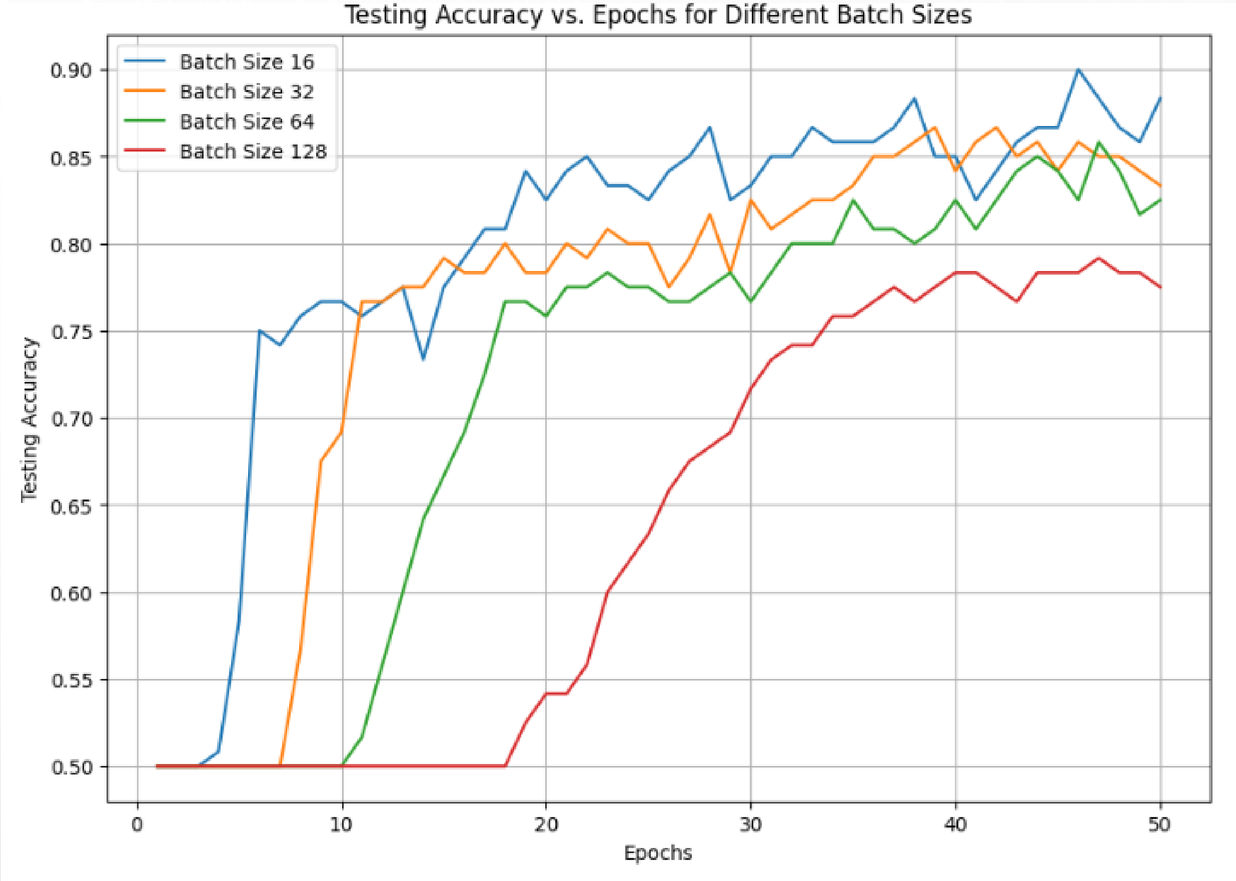
- To ensure that the testing environment remains consistent
- results before and after code optimizations can achieve accurate comparison

```python
def same_seeds(seed):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed(seed)
        torch.cuda.manual_seed_all(seed)
    torch.backends.cudnn.benchmark = False
    torch.backends.cudnn.deterministic = True

same_seeds(48763)
```

Batch Size

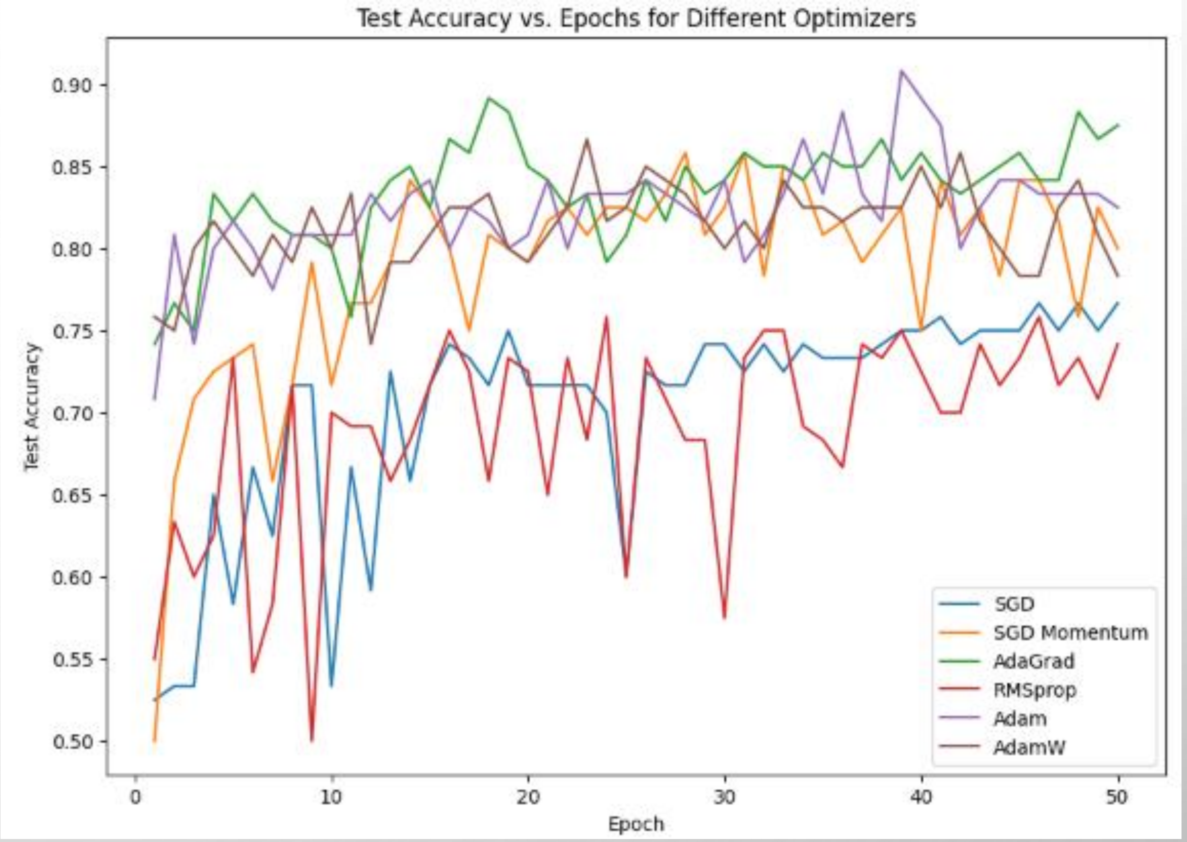
- Comparing Different Batch size
  - A smaller batch size converges faster and results in higher accuracy.
- We finally using 16 as our batch size



李崇碩、蔡東宏

Optimizer Choosing

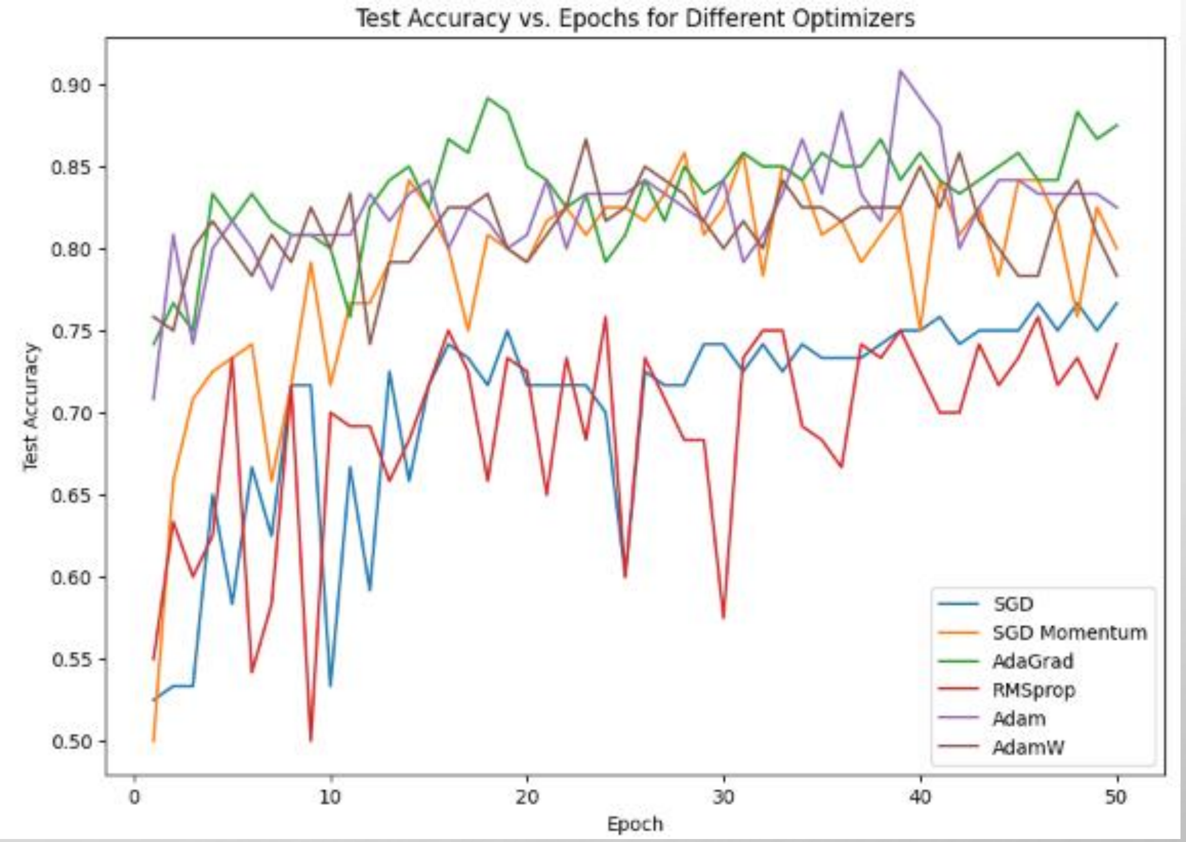
- Comparing Different Optimizer
  - SGD
  - SGD Momentum
  - AdaGrad
  - RMSprop
  - Adam
  - AdamW



Optimizer Choosing

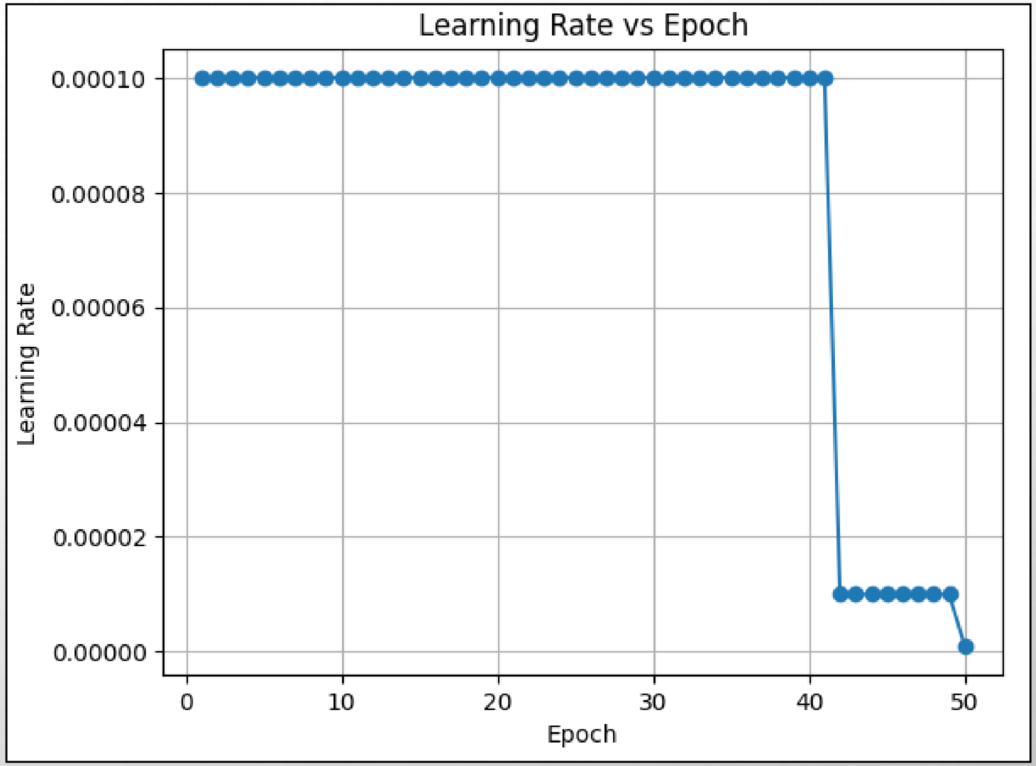
- **Through many times testing , We find that set Learning Rate = 0.0001 can achieve the best accuracy**

- **Although AdaGrad surpassed Adam in the final few epochs, Adam achieved a maximum testing accuracy of 90%. Therefore, we chose Adam as the optimizer.**



Dynamic adjusting Learning Rate

- Using ReduceLROnPlateau to dynamic adjusting learning rate.
- We can observe that in first 40 epochs, the learning rate is 0.0001. However, in final few epochs, the learning rate is reduced.



李崇碩、蔡東宏

Dynamic adjusting Learning Rate

- **After lowering the learning rate, the large-scale instability and oscillations were significantly reduced.**

Before Adjusting

```
Epoch 40/50, Loss: 0.1276, Accuracy_train: 0.9625, Accuracy_test: 0.8417,
Epoch 41/50, Loss: 0.1223, Accuracy_train: 0.9625, Accuracy_test: 0.8417,
Epoch 42/50, Loss: 0.1410, Accuracy_train: 0.9589, Accuracy_test: 0.8417,
Epoch 43/50, Loss: 0.1193, Accuracy_train: 0.9661, Accuracy_test: 0.8250,
Epoch 44/50, Loss: 0.1028, Accuracy_train: 0.9750, Accuracy_test: 0.8333,
Epoch 45/50, Loss: 0.1113, Accuracy_train: 0.9589, Accuracy_test: 0.8250,
Epoch 46/50, Loss: 0.0831, Accuracy_train: 0.9786, Accuracy_test: 0.8417,
Epoch 47/50, Loss: 0.0933, Accuracy_train: 0.9661, Accuracy_test: 0.8250,
Epoch 48/50, Loss: 0.0842, Accuracy_train: 0.9786, Accuracy_test: 0.8500,
Epoch 49/50, Loss: 0.0785, Accuracy_train: 0.9732, Accuracy_test: 0.8250,
Epoch 50/50, Loss: 0.0778, Accuracy_train: 0.9821, Accuracy_test: 0.8333,
```

After Adjusting

```
Epoch 00041: reducing learning rate of group 0 to 1.0000e-05.
Learning Rate: 0.000010
Epoch 42/50, Loss: 0.2302, Accuracy_train: 0.9393, Accuracy_test: 0.8500
Learning Rate: 0.000010
Epoch 43/50, Loss: 0.2496, Accuracy_train: 0.9107, Accuracy_test: 0.8583
Learning Rate: 0.000010
Epoch 44/50, Loss: 0.2406, Accuracy_train: 0.9286, Accuracy_test: 0.8500
Learning Rate: 0.000010
Epoch 45/50, Loss: 0.2355, Accuracy_train: 0.9304, Accuracy_test: 0.8583
Learning Rate: 0.000010
Epoch 46/50, Loss: 0.2483, Accuracy_train: 0.9214, Accuracy_test: 0.8583
Learning Rate: 0.000010
Epoch 47/50, Loss: 0.2367, Accuracy_train: 0.9268, Accuracy_test: 0.8583
Learning Rate: 0.000010
Epoch 48/50, Loss: 0.2319, Accuracy_train: 0.9321, Accuracy_test: 0.8500
Learning Rate: 0.000010
Epoch 49/50, Loss: 0.2348, Accuracy_train: 0.9321, Accuracy_test: 0.8583
Epoch 00049: reducing learning rate of group 0 to 1.0000e-06.
Learning Rate: 0.000001
Epoch 50/50, Loss: 0.2224, Accuracy_train: 0.9304, Accuracy_test: 0.8583
Training complete
```

Recording Best Accuracy Model

- Record the Testing Accuracy for each run, then load the model with the best Testing Accuracy for the final demo to ensure optimal accuracy.

```python
model = Network(num_classes=2)
model.load_state_dict(torch.load('best_model.pth'))
model = model.to(device)
model.eval()

correct_test = 0
total_test = 0
with torch.no_grad():
    for inputs, labels in test_loader:
        inputs, labels = inputs.to(device), labels.to(device)
        outputs = model(inputs)
        _, predicted = torch.max(outputs.data, 1)
        total_test += labels.size(0)
        correct_test += (predicted == labels).sum().item()

    epoch_acc_test = correct_test / total_test
    print(f'Accuracy_test: {epoch_acc_test:.4f}')
```

Reference

- https://hackmd.io/@allen108108/H1l4zqtp4
- https://medium.com/%E5%AD%B8%E4%BB%A5%E5%BB%A3%E6%89%8D/note-shufflenet-%E5%AD%B8%E7%BF%92%E5%BF%83%E5%BE%97-38f2f715e19a