Segmentation Report

Ge Wang a1880714
Tien

Introduction

An important application in self-driving cars is semantic segmentation, which is a key task in computer vision that assigns class labels to every pixel in an image. This process is vital for achieving precise scene understanding. The baseline model used for this task implements a simple convolutional neural network architecture for pixel-wise classification, utilizing a dataset that includes 150 training samples and 50 testing samples across 19 semantic classes. The objective is to enhance the baseline segmentation model by improving its accuracy and computational efficiency.

The baseline was a shallow CNN model following a traditional encoder-decoder structure with four downsampling stages using 3x3 convolutions and max pooling, followed by upsampling to restore spatial resolution. The model trained with CrossEntropyLoss using the Adam optimizer (learning rate 3e-4, batch size=4), helps convergence stable and fast. It achieves a low segmentation accuracy (mIoU = 28.6%) with a cheap GFLOPs computational cost (66.97%). From there, we can see the model can roughly segment objects but has many incorrect or imprecise predictions.

System Modifications and Ablation Study

To enhance the baseline model's performance, I implemented several modifications across data augmentation, training parameters, architecture, and loss functions, as detailed below:

1. Data Augmentation

We applied data augmentation techniques to address the limited training dataset (150 images) and enhance model generalization, particularly for autonomous driving scenarios. The following methods were used:

- Random Rotation (5°): Simulates varying object orientations.
- Gaussian Blur (kernel size 3): Mimics reduced image quality or focus.
- Color Jitter: Adjusts brightness, contrast, saturation, and hue to replicate diverse lighting conditions.

```
self.transformImg = tf.Compose([tf.ToPILImage(),
tf.RandomRotation(5), # random rotation data augmentation
tf.GaussianBlur(kernel_size=3), # Gaussian blur data augmentation
tf.ColorJitter(brightness=0.1, contrast=0.1, saturation=0.1, hue=0.05),#color jitter data
augmentation
tf.Resize((height, width)),
tf.ToTensor(),
tf.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225))
])
```

These techniques diversify the training data, exposing the model to a broader range of real-world conditions without increasing the dataset size.

2. Training Parameters

We optimized the training process by adjusting key hyperparameters:

- Optimizer: Replaced Adam with AdamW (weight decay 1e-4) to improve weight decay regulation and convergence.
- Batch Size: Reduced from 4 to 2 to accommodate larger architectures and enable finer gradient updates.
- Learning Rate Scheduler: Introduced cosine annealing (T_max=180 epochs) to dynamically adjust the learning rate, enhancing convergence stability and final performance.

```
optimizer = torch.optim.AdamW(params=segNet.parameters(), lr=learning_rate,
weight_decay=1e-4)

scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=epochs)
#Adding a learning rate scheduler
```

These changes aimed to improve training efficiency and model generalization.

3. Architectural Changes

We enhanced the convolutional neural network (CNN) architecture with the following modifications:

- Batch Normalization: Added after convolutional layers to stabilize training and accelerate convergence.
- Residual Connections: Incorporated to mitigate vanishing gradients, supporting deeper networks.

- Skip Connections: Integrated between encoder and decoder stages to preserve spatial details during upsampling.
- Dilated Convolutions: Applied in the bottleneck (dilation=2) to expand the receptive field without increasing parameters.

```
#Define the semantic segmentation network. Tips: a new network can be used
class SegNetwork(nn.Module):
def __init__(self,
n_class=19,
use_residual=False, #optional residual connection
use_batch_norm=False, #optional batch normalization
):
super(SegNetwork, self).__init__()
self.use_residual = use_residual #optional residual connection
self.relu = nn.ReLU(inplace=True)
#stage 1
self.conv1_1 = nn.Conv2d(3, 64, 3, padding=1)
if use_batch_norm:
self.bn1_1 = nn.BatchNorm2d(64, momentum=0.1)
self.relu1_1 = nn.ReLU(inplace=True)
self.conv1_2 = nn.Conv2d(64, 64, 3, padding=1)
if use_batch_norm:
self.bn1_2 = nn.BatchNorm2d(64)
self.relu1_2 = nn.ReLU(inplace=True)
self.pool1 = nn.MaxPool2d(2, stride=2, ceil_mode=True) #1/2

# Residual block for stage 1
self.res_block1 = nn.Sequential(
nn.Conv2d(3, 64, 3, padding=1),
nn.BatchNorm2d(64, momentum=0.1),
nn.ReLU(inplace=True)
)

#stage 2
self.conv2_1 = nn.Conv2d(64, 128, 3, padding=1)
if use_batch_norm:
self.bn2_1 = nn.BatchNorm2d(128)
self.relu2_1 = nn.ReLU(inplace=True)
self.conv2_2 = nn.Conv2d(128, 128, 3, padding=1)
if use_batch_norm:
self.bn2_2 = nn.BatchNorm2d(128)
```

```
self.relu2_2 = nn.ReLU(inplace=True)
self.drop2_1 = nn.Dropout2d(p=0.1) #optional dropout
self.pool2 = nn.MaxPool2d(2, stride=2, ceil_mode=True) #1/4
# Residual block for stage 2
self.res_block2 = nn.Sequential(
nn.Conv2d(64, 128, 1), # 1x1 conv to match channels
nn.BatchNorm2d(128, momentum=0.1),
nn.ReLU(inplace=True)
)

#stage 3
self.conv3_1 = nn.Conv2d(128, 256, 3, padding=1)
if use_batch_norm:
self.bn3_1 = nn.BatchNorm2d(256)
self.relu3_1 = nn.ReLU(inplace=True)
self.conv3_2 = nn.Conv2d(256, 256, 3, padding=1)
if use_batch_norm:
self.bn3_2 = nn.BatchNorm2d(256)
self.relu3_2 = nn.ReLU(inplace=True)
self.pool3 = nn.MaxPool2d(2, stride=2, ceil_mode=True) #1/8
# Residual block for stage 3
self.res_block3 = nn.Sequential(
nn.Conv2d(128, 256, 1), # 1x1 conv to match channels
nn.BatchNorm2d(256, momentum=0.1),
nn.ReLU(inplace=True)
)

#stage 4
self.conv4_1 = nn.Conv2d(256, 512, 3, padding=1)
if use_batch_norm:
self.bn4_1 = nn.BatchNorm2d(512)
self.relu4_1 = nn.ReLU(inplace=True)
self.conv4_2 = nn.Conv2d(512, 512, 3, padding=1)
if use_batch_norm:
self.bn4_2 = nn.BatchNorm2d(512)
self.relu4_2 = nn.ReLU(inplace=True)
self.pool4 = nn.MaxPool2d(2, stride=2, ceil_mode=True) #1/16
# Residual block for stage 4
self.res_block4 = nn.Sequential(
nn.Conv2d(256, 512, 1), # 1x1 conv to match channels
```

```
nn.BatchNorm2d(512, momentum=0.1),
nn.ReLU(inplace=True)
)

#stage 5
# Bottleneck: Input 512 channels, Output 512 channels
self.conv5_1 = nn.Conv2d(512, 512, 3, padding=2, dilation=2) # Dilated conv
self.bn5_1 = nn.BatchNorm2d(512, momentum=0.1) if use_batch_norm else None
self.relu5_1 = nn.ReLU(inplace=True)

# Upsampling path with skip connections
self.upsample_conv1 = nn.Conv2d(512, 256, 3, padding=1)
self.bn_up1 = nn.BatchNorm2d(256, momentum=0.1) if use_batch_norm else None
self.relu_up1 = nn.ReLU(inplace=True)

self.upsample_conv2 = nn.Conv2d(256, 128, 3, padding=1)
self.bn_up2 = nn.BatchNorm2d(128, momentum=0.1) if use_batch_norm else None
self.relu_up2 = nn.ReLU(inplace=True)

self.upsample_conv3 = nn.Conv2d(128, 64, 3, padding=1)
self.bn_up3 = nn.BatchNorm2d(64, momentum=0.1) if use_batch_norm else None
self.relu_up3 = nn.ReLU(inplace=True)

# Final convolution to produce segmentation map
self.final_conv = nn.Conv2d(64, n_class, 1)

def forward(self, x):
inp_shape = x.shape[2:] # (256, 864)
# Stage 1
shortcut1 = x
x = self.relu(self.conv1_1(x))
x = self.relu(self.conv1_2(x))
if self.use_residual and self.res_block1 is not None:
x = x + self.res_block1(shortcut1)
skip1 = x
x = self.pool1(x)

# Stage 2
shortcut2 = x
x = self.relu(self.bn2_1(self.conv2_1(x)))
```

```
x = self.relu(self.bn2_2(self.conv2_2(x)))
if self.use_residual and self.res_block2 is not None:
x = x + self.res_block2(shortcut2)
skip2 = x
x = self.pool2(x)

# Stage 3
shortcut3 = x
x = self.relu(self.bn3_1(self.conv3_1(x)))
x = self.relu(self.bn3_2(self.conv3_2(x)))
if self.use_residual and self.res_block3 is not None:
x = x + self.res_block3(shortcut3)
skip3 = x
x = self.pool3(x)

# Stage 4
shortcut4 = x
x = self.relu(self.bn4_1(self.conv4_1(x)))
x = self.relu(self.bn4_2(self.conv4_2(x)))
if self.use_residual and self.res_block4 is not None:
x = x + self.res_block4(shortcut4)
x = self.pool4(x)

# Bottleneck
x = self.conv5_1(x)
if self.bn5_1 is not None:
x = self.bn5_1(x)
x = self.relu5_1(x)

# Upsampling path
x = F.interpolate(x, size=skip3.shape[2:], mode="bilinear", align_corners=True)
x = self.upsample_conv1(x)
if self.bn_up1 is not None:
x = self.bn_up1(x)
x = self.relu_up1(x)
x = x + skip3

x = F.interpolate(x, size=skip2.shape[2:], mode="bilinear", align_corners=True)
x = self.upsample_conv2(x)
if self.bn_up2 is not None:
```

```
x = self.bn_up2(x)
x = self.relu_up2(x)
x = x + skip2

x = F.interpolate(x, size=skip1.shape[2:], mode="bilinear", align_corners=True)
x = self.upsample_conv3(x)
if self.bn_up3 is not None:
x = self.bn_up3(x)
x = self.relu_up3(x)
x = x + skip1

# Final convolution
x = self.final_conv(x)
x = F.interpolate(x, size=inp_shape, mode="bilinear", align_corners=True)
return x # [batch_size, 19, 256, 864]

#Get the predefined network
segNet = SegNetwork(n_class=19,
use_residual=True, #optional residual connection
use_batch_norm=True #optional batch normalization
).to(device)
```

- MobileNetV3 Backbone: Tested as a lightweight encoder, leveraging depthwise separable convolutions to reduce computational cost while maintaining accuracy.

```
import torchvision
# Load DeepLabV3 with MobileNetV3 backbone
segNet = torchvision.models.segmentation.deeplabv3_mobilenet_v3_large(
weights=None, # or 'DEFAULT' for pretrained on COCO
num_classes=19
).to(device)
```

These enhancements aimed to improve segmentation quality and training stability.

4. Loss Function

To address class imbalance and enhance segmentation quality, we experimented with a hybrid loss function combining CrossEntropyLoss and DiceLoss (alpha=0.5). This was intended to balance per-pixel accuracy and region-based overlap.

Ablation Table Study

The following ablation study summarizes the impact of each modification:

Data Augmentation	Training Params	Architecture	Loss Function	mIoU (%)	GFLOPs
None	Adam, lr=3e-4, BS=4	Baseline	CrossEntropy	0.286	66.97
Crop+Rot+Blur+Jitter	AdamW, lr=3e-4, BS=2, CosineAnneal	Baseline	CrossEntropy	0.1566	66.97
None	AdamW, lr=3e-4, BS=2, CosineAnneal	BatchNorm+Residual+Skip+Dilated	CrossEntropy	0.3421	204.83
Crop+Rot+Blur+Jitter	AdamW, lr=3e-4, BS=2, CosineAnneal	BatchNorm+Residual+Skip+Dilated	CrossEntropy	0.3110	204.83
Crop+Rot+Blur+Jitter	AdamW, lr=3e-4, BS=2, CosineAnneal	BatchNorm+Residual+Skip+Dilated	Hybrid	0.2125	204.84
None	AdamW, lr=3e-4, BS=2, CosineAnneal	MobileNetV3+BatchNorm+Skip	CrossEntropy	0.3377	18.10
Crop+Rot+Blur+Jitter	AdamW, lr=3e-4, BS=2, CosineAnneal	MobileNetV3+BatchNorm+Skip	CrossEntropy	0.3292	18.10
Crop+Rot+Blur+Jitter	AdamW, lr=3e-4, BS=2, CosineAnneal	MobileNetV3+BatchNorm+Skip	Hybrid	0.3318	18.10

The MobileNet V3 model saved computational costs, reducing GFLOPs from 204.83 to 18.10 while maintaining comparable accuracy (33.77% to 34.21% mIoU). This demonstrates the effectiveness of high performance for architectural semantic segmentation. Batch normalization and skip connection help to increase significant convergence speed and computational cost, and the important factor is the final accuracy. The results demonstrate that architectural improvements provide the most significant accuracy gains (28.6% to 34.21%), while the MobileNet V3 backbone offers the best

efficiency trade-off. The hybrid loss function decreased performance in our implementation due to suboptimal loss weighting.

Limitations/Conclusions:

There are some limitations in the study, the hybrid loss function does not meet expectations, more sophisticated weighting strategies or additional hyperparameter optimization may be needed. In addition, the dataset size remains a limitation with 150 training images and 50 testing images, which does not provide a fully robust performance estimate.

Modifications to the baseline semantic segmentation model markedly improved performance for autonomous driving applications, with the custom architecture (batch normalization, residual connections, skip connections, and dilated convolutions) achieving a top mIoU of 34.21%—a 19.6% gain over the baseline—but at a high computational cost of 204.83 GFLOPs, hindering real-time use.

The MobileNetV3-based model balanced efficiency and accuracy, delivering a 33.77% mIoU (18.1% improvement) with just 18.10 GFLOPs, ideal for resource-limited settings. However, a small dataset (150 training images) caused overfitting despite augmentation, and the hybrid loss function (CrossEntropyLoss + DiceLoss) underperformed at 21.25% mIoU due to poor weighting or class imbalance handling.

Future work should refine the loss function, explore lightweight architectures, and expand the dataset to boost generalization and efficiency.