CIS6800: Project 6: SegFormer

Instructions:

- This is a group assignment with one submission per group. It is expected that each member of the group will contribute to solving each question. Be sure to specify your teammates when you submit to Gradescope! Collaborating with other groups is not permitted.
- There is no single answer to most problems in deep learning, therefore the questions will often be underspecified. You need to fill in the blanks and submit a solution that solves the (practical) problem. Document the choices (hyperparameters, features, neural network architectures, etc.) you made where specified.
- All the code should be written in Python.
- You may include any code used in previous projects.

Introduction

SegFormer is a simple architecture with some key innovations that allow it to acheive state of the art performance at much higher speeds and fewer parameters than other competing methods. In this assignment, you will be implementing the building blocks of SegFormer. An overview of the architecture is provided below.

It is highly encouraged for you to review the SegFormer paper, available here. In particular, Section 3 contains detailed descriptions of the modules you will be implementing.

[ ] ↳ 2 cells hidden

Data

For this assignment, we will use a simplified subset of ADE20K containg approximately 11,000 images with only four classes: "plant", "person/animal", "vehicle", and "background". The data in composed of a semantic class for each pixel - there is no distinction between different instances of the same object, unlike in previous assignments.

The following code downloads the original dataset, processes it, and visualizes an example.

[ ] ↳ 4 cells hidden

Modules

For this section, you will have to implement the base functionalities and modules for the SegFormer. As illustrated in the **Figure 2** and **Section 3** in the paper, we divide SegFormer into encoder-decoder sections. The encoder mainly consists of multiple layers of MixTransformerEncoderLayer, which then consists of OverlapPatchMergiing + (EfficientSelfAttention + MixFFN). While patch merging is relatively intuitive, you have to implement EfficientSelfAttention following **Equation 1** in the paper, and MixFFN as illustrated in **Equation 3**. On the other hand, the decoder section also consists of multiple decoder blocks, ending with a MLP. Refer to **Equation 4** for this part.

[ ] ↳ 5 cells hidden

Model and Training

We have provided you with a near-complete network template which you can indicate how a complete SegFormer is structured and how it calls the modules we have built. For validation step, you need to implement mean IoU as paper suggests.

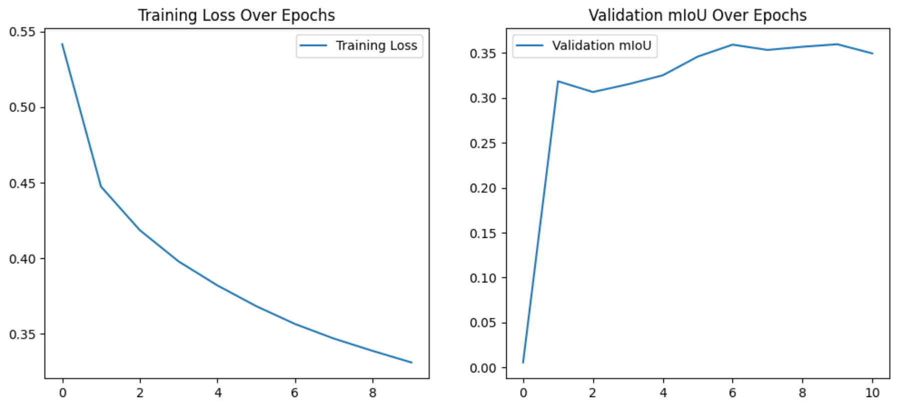
[ ] ↳ 3 cells hidden

Results

Show your training curves and results on several images from the validation set. Provide a qualitative description of the performance of your model. Don't forget to upsample your predicted segmentation mask to the same resolution as the image! Segformer indeed has achieved great performance, there is a short demo on Cityspaces-C corrupted dataset on YouTube.

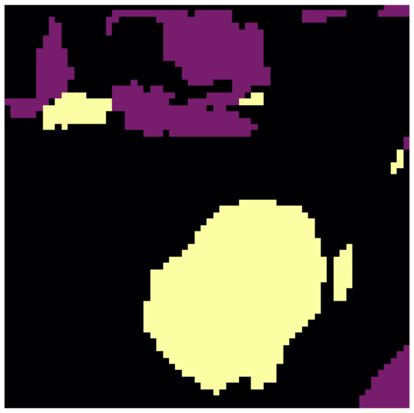
```
import matplotlib.pyplot as plt

plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(model.train_epoch, label='Training Loss')
plt.title('Training Loss Over Epochs')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(model.miou_epoch, label='Validation mIoU')
plt.title('Validation mIoU Over Epochs')
plt.legend()

# plt.show()
```

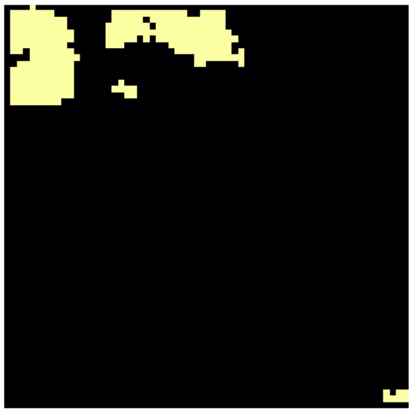
```
<matplotlib.legend.Legend at 0x79c09bc23280>
```



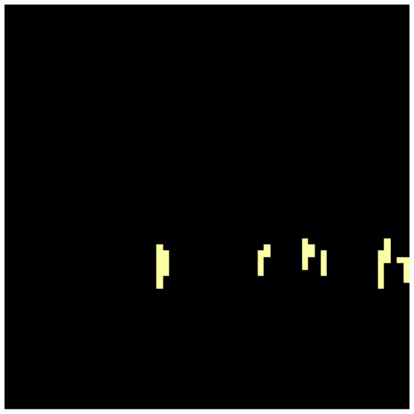
```
def denormalize(tensor, mean, std):
    if tensor.ndim == 4:  # Batch of images
        mean = mean[:, None, None]
        std = std[:, None, None]
    else:  # Single image
        mean = mean.view(-1, 1, 1)
        std = std.view(-1, 1, 1)
    return tensor * std + mean


import matplotlib.pyplot as plt
from torchvision.transforms.functional import to_pil_image

def visualize_predictions(model, val_loader, device, num_images=5):
    model.eval()  # Set the model to evaluation mode
    model.to(device)

    images, predictions, targets = [], [], []
    with torch.no_grad():
        for i, (image, target) in enumerate(val_loader):
            if i >= num_images:
                break
            image = image.to(device)
            pred = model(image)

            images.append(image.cpu())
            predictions.append(pred.cpu())
            targets.append(target.cpu())

    mean = torch.tensor([0.485, 0.456, 0.406])
    std = torch.tensor([0.229, 0.224, 0.225])

    for i in range(5):
        plt.figure(figsize=(18, 6))

        # Original Image (De-normalized)
        denormalized_img = denormalize(images[i][0], mean, std)
        plt.subplot(1, 3, 1)
        plt.imshow(to_pil_image(denormalized_img))
        plt.title('Original Image (De-normalized)')
        plt.axis('off')
        plt.axis('off')

        # Ground Truth Mask
        plt.subplot(1, 3, 2)
        plt.imshow(targets[i][0], cmap='inferno')
        plt.title('Ground Truth Mask')
        plt.axis('off')

        plt.subplot(1, 3, 3)
        pred_class = torch.argmax(predictions[i][0], dim=0)
        plt.imshow(pred_class, cmap='inferno')
        plt.title('Predicted Mask')
        plt.axis('off')

        plt.show()

# Call the function to visualize the predictions
visualize_predictions(model, val_loader, device='cuda' if torch.cuda.is_available() else 'cpu')
```

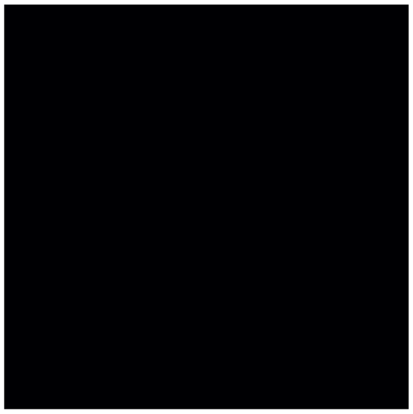
Original Image (De-normalized)

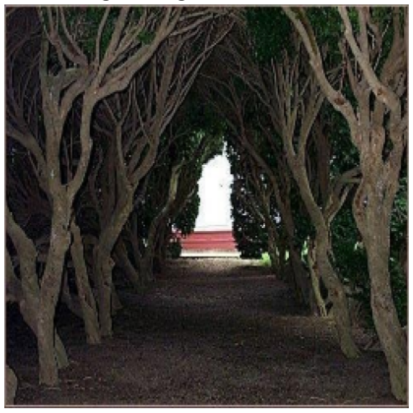

Ground Truth Mask

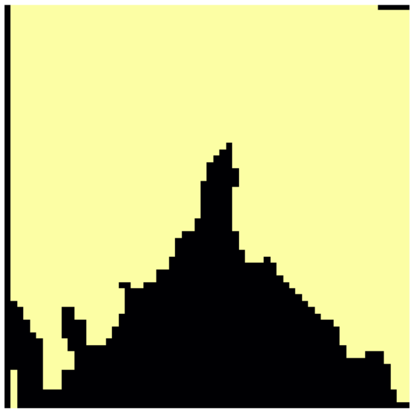

Predicted Mask

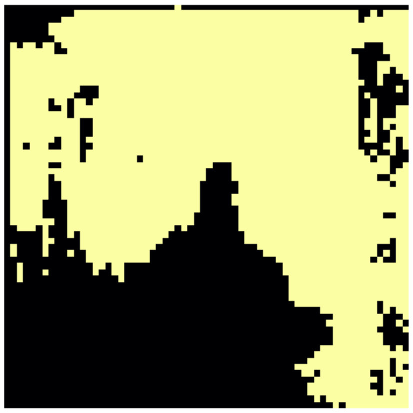
Original Image (De-normalized)


Ground Truth Mask


Predicted Mask


Original Image (De-normalized)


Ground Truth Mask


Predicted Mask

Original Image (De-normalized)

Ground Truth Mask

Predicted Mask

Discussion Question

1. Why do we overlap and merge patches when reducing dimensions instead of e.g. max pool?
2. How does efficient self-attention affect the performance of the model?
3. Why do we only have one activation function in MixFFN?
4. If we removed padding='same' in MixFFN, would the performance of our model be affected? Why or why not? Suppose that there are no dimensional issues caused by this.
5. What is the difference between cross-entropy loss and miou? Why do we not use the miou to train?
6. Why are transformers especially suitable for this architecture? Namely, passing through different feature map stages and using a very small decoder. What problems might arise if we used convolutions instead?
7. Do you think that adding an FPN makes sense for this architecture? Justify your answer.

8. Your achieved performance is likely not very high, considering the simplicity of the data set. This assignment makes several simplications to the training process that contribute to this. Choose two such simplifications and explain why they degrade the achieved performance.

1) We overlap and merge patches for dimensionality reduction as opposed to max. pooling as it allows to retain more information from the input data. Since max pooling only keeps the max value within the video, it is prone to loosing important information. While overlap and merge for patches allows for better representation especially when we have overlapping patches.

2) Efficient Self Attention basically reduces the computational complexity and memory usage by reducing the number of operation needed for attention calculations. We did dimensionality reduction of keys and queries before applying attention mechanism.

3) We only use one activation function (GELU here) to introduce non-linearity to the model, while still keeping the model efficient.

4) Yes, removing `padding=same` will change the performance slightly as the model given no dimensional issues are caused by it. As this allows the model to still process the spatial information without changing the size of the output. But change in padding will slightly change how boundary information is handled.

5) Cross Entropy Loss measures the difference between the predicted probabilities and true labels, generally used for gradient based optimization. Whereas, mIoU is an evaluation metric that measures the overlap between predicted and actual class. Since it is not differentiable, we can't use it for loss function.

6) Transformers are especially suitable for this architecture due to their ability to capture long range dependencies and their global receptive field. Their self attention mechanism allows to process all parts of the input in relation to each other irrespective of their distance. If convolutions were used intead,we would face issues with the local receptive field and limitation to capturing long range dependencies without a complex model. CNNs can help gather local features but dto aggregate features globally, we would need much more complex model. Transformers can work with global aggregation more efficiently.

7) Yes, adding FPN to this architecture makes sense as it provides complimentary strengths to Transformers. FPN can effectively aggregate multi-scale spatial information. Combining the ehances the model bu allowing it handle variations in object sizes and different scales. Further, FPN can also provide enhanced local-global feature integration as it integrates low level details with the high level semantic information.

8) We didn't perform data augmentation. Data augmentation is beneficial as it perfors transformations, scaling and various other augmentations to enhance the diversity of the training set. Since we didn't perform it, the model might not learn those variations that can happen in the real world.

Additionally, we didn't try different optimization techniques when talking specifically about the training process. Using advanced optimization techniques could be effective in convergence to minimizing loss.