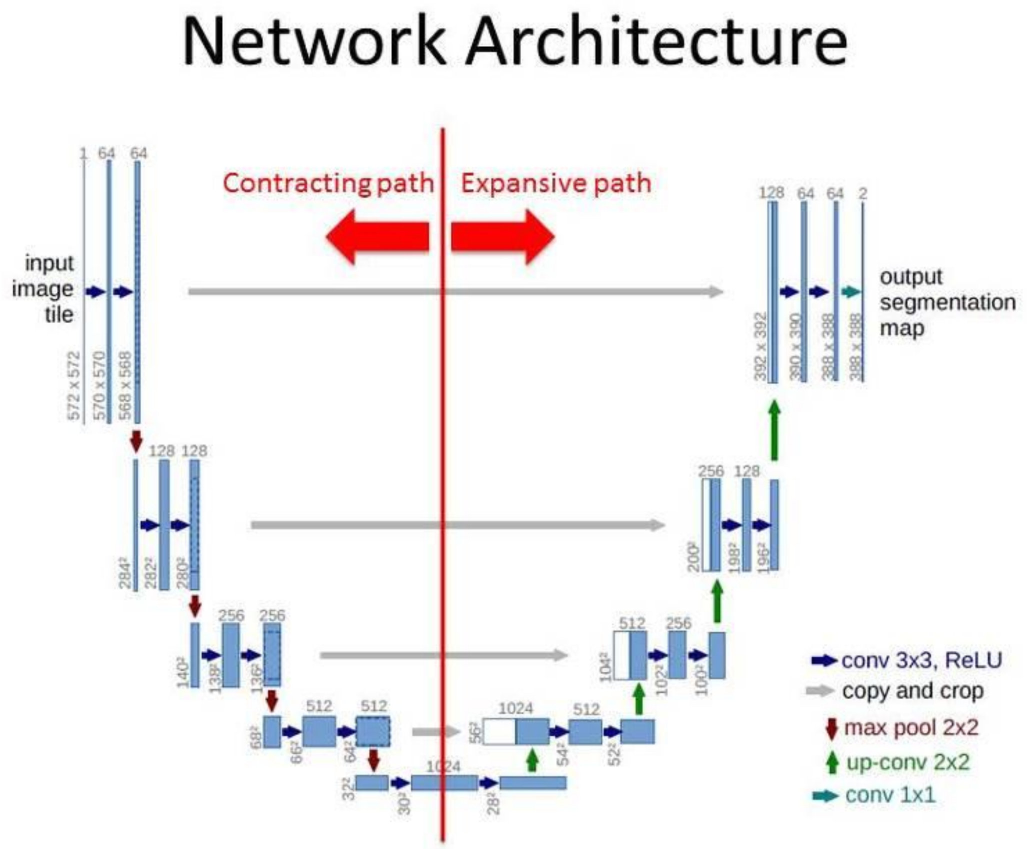
DEEP LEARNING UPDATE 3 REPORT



Contracting Path: The contracting path of a U-Net architecture consists of a series of convolutional layers followed by max-pooling layers. This path is responsible for reducing the spatial resolution of the input image while increasing the number of feature maps.

Bottleneck: The bottleneck of a U-Net architecture is the middle part of the "U" shape. It consists of a series of convolutional layers without any max-pooling layers, which allows the network to capture high-level features while retaining spatial information.

Expanding Path: The expanding path of a U-Net architecture consists of a series of transposed convolutional layers followed by concatenation with feature maps from the contracting path at the same level. This path is responsible for increasing the spatial resolution of the feature maps while reducing the number of feature maps

Skip Connections: The skip connections are the connections between the contracting and expanding paths of the U-Net architecture. These connections allow the network to propagate spatial information from the contracting path to the expanding path, which helps to preserve fine-grained details in the output segmentation maps.

This is our Architecture:

```
SegmentationModel(
  (relu): ReLU()
  (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv3): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (upconv1): ConvTranspose2d(128, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), output_padding=(1, 1))
  (conv4): Conv2d(128, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (upconv2): ConvTranspose2d(64, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), output_padding=(1, 1))
  (conv5): Conv2d(64, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (upconv3): ConvTranspose2d(32, 16, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), output_padding=(1, 1))
  (outconv): Conv2d(16, 6, kernel_size=(1, 1), stride=(2, 2))
)
```

The architecture consists of two main parts: the encoder and the decoder. The encoder is a series of convolutional layers that reduce the spatial resolution of the input image while increasing the number of feature maps. The decoder is a series of transposed convolutional layers that upsample the feature maps to the original resolution while reducing the number of feature maps. The skip connections between the encoder and decoder layers are used to combine the high-level and low-level features, allowing the network to make more precise segmentation predictions.

In this specific implementation, the encoder consists of three convolutional layers and the decoder consists of three transposed convolutional layers. The network takes a 3-channel RGB image as input and outputs a segmentation mask with 6 channels, one for each class label. The activation function used throughout the network is the Rectified Linear Unit (ReLU), and the weights are initialized using the Kaiming initialization method. The loss function used is the Cross Entropy Loss.

<u>Objective function</u>

<u>Loss function :</u>

CrossEntropyLoss is a commonly used loss function for multi-class classification problems. In fact, it is one of the most widely used loss functions for multi-class image segmentation tasks, where each pixel in the image is assigned a label corresponding to one of several classes. The CrossEntropyLoss function combines a softmax activation function with a negative log-likelihood loss, which makes it well-suited for multi-class problems.

Optimizer:

Stochastic Gradient Descent (SGD), which is a popular optimization algorithm used in deep learning. It is used to update the parameters of the model during the training process to minimize the loss function.

Our Model layer architecture
Our SegmentationModel uses a total of 9 layers, including six convolutional layers and 3 transposed convolutional layers.

Our Input image dimensions are 3*320*320, where 3 is **RGB Dimension**, 320 is **height** and 320 is **width**, and Batch size is 8. So our Model takes [8,3,320,320] as input.

So here our output shape is out1 shape: torch.Size([8, 32, 320, 320]), after applying the Relu Activation function we get the out1 shape after ReLU: torch.Size([8, 32, 320, 320]).

So similarly for next outshape's after applying the as the Relu Activation functions we get:

- ➔ out2 shape: torch.Size([8, 64, 320, 320])

  out2 shape after ReLU: torch.Size([8, 64, 320, 320])
- ➔ out3 shape: torch.Size([8, 128, 160, 160])

  out3 shape after ReLU: torch.Size([8, 128, 160, 160])
- ➔ out4 shape: torch.Size([8, 64, 320, 320])

  out4 shape after concatenation: torch.Size([8, 128, 320, 320])
- ➔ out4 shape after conv4: torch.Size([8, 64, 160, 160])

  out4 shape after ReLU: torch.Size([8, 64, 160, 160])
- ➔ out5 shape: torch.Size([8, 32, 320, 320])

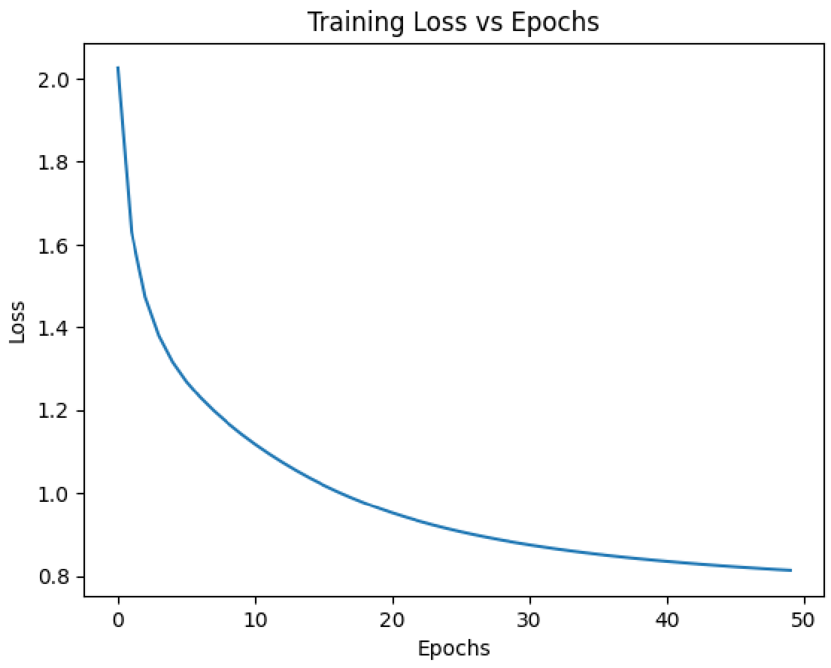
  out5 shape after concatenation: torch.Size([8, 64, 320, 320])
- ➔ out5 shape after conv5: torch.Size([8, 32, 320, 320])

  out shape after ReLU: torch.Size([8, 32, 320, 320])
- ➔ so for the final output shape, out6 shape: torch.Size([8, 16, 640, 640]), after applying the Relu Activation function we get the Final output shape: torch.Size([8, 6, 320, 320]).

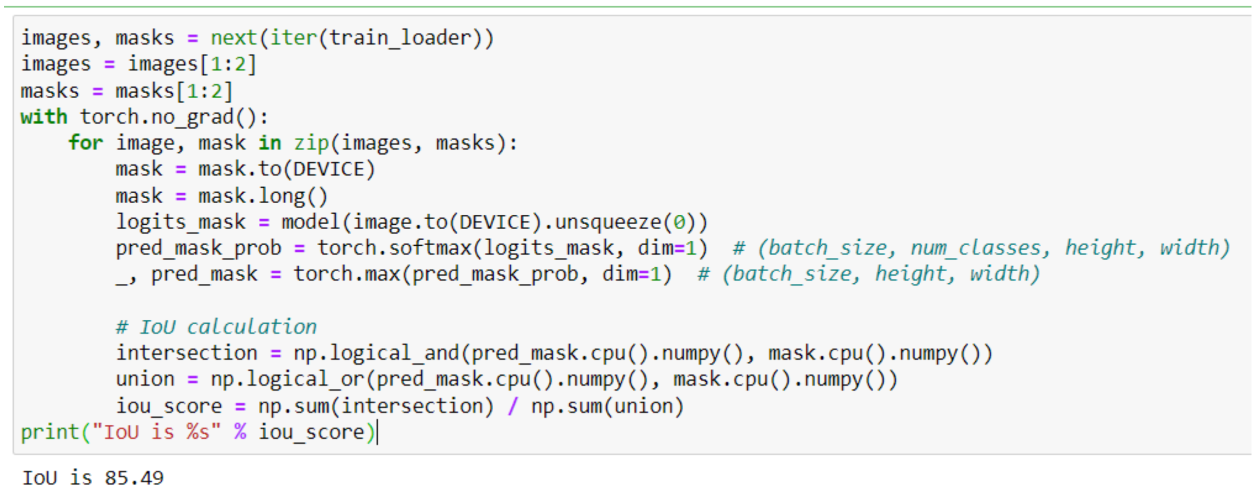
Experiments and Results:

```
# training only on 2 samples!
images, masks = next(iter(train_loader))
images = images[0:2]
masks = masks[0:2]

losses = []
for epoch in range(50):
    images = images.to(DEVICE)
    masks = masks.to(DEVICE)
    masks = masks.long()  # Convert to integer tensor
    optimizer.zero_grad()
    logits, loss = model(images, masks)
    loss.mean().backward()
    optimizer.step()
    losses.append(loss.mean().item())

plt.plot(losses)
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training Loss vs Epochs')
plt.show()
```

Initially we have tired to over fit the model just by providing only 2 samples from training dateset and we have taken 50 epochs and plotted the training loss vs epochs.



```
Intersection over union on two samples.
```



```
images, masks = next(iter(train_loader))
images = images[1:2]
masks = masks[1:2]
with torch.no_grad():
    for image, mask in zip(images, masks):
        mask = mask.to(DEVICE)
        mask = mask.long()
        logits_mask = model(image.to(DEVICE).unsqueeze(0))
        pred_mask_prob = torch.softmax(logits_mask, dim=1)  # (batch_size, num_classes, height, width)
        _, pred_mask = torch.max(pred_mask_prob, dim=1)  # (batch_size, height, width)

        # IoU calculation
        intersection = np.logical_and(pred_mask.cpu().numpy(), mask.cpu().numpy())
        union = np.logical_or(pred_mask.cpu().numpy(), mask.cpu().numpy())
        iou_score = np.sum(intersection) / np.sum(union)
print("IoU is %s" % iou_score)
```

```
IoU is 85.49
```

The code first selects a single image and its mask from the training data loader. Then, using the selected image and mask, the model predicts a mask probability tensor using the model instance. The probabilities are converted into class predictions by taking the maximum class probability along the channel dimension using torch.max.

The IoU score is then computed by first converting both the predicted and ground truth masks to NumPy arrays using .cpu().numpy(). Then, the intersection and union between the predicted and ground truth masks are calculated element-wise using NumPy's logical_and and logical_or functions. Finally, the IoU score is computed as the ratio of the sum of the intersection over the sum of the union.

The IoU score is printed at the end of the loop using Python's print() function.

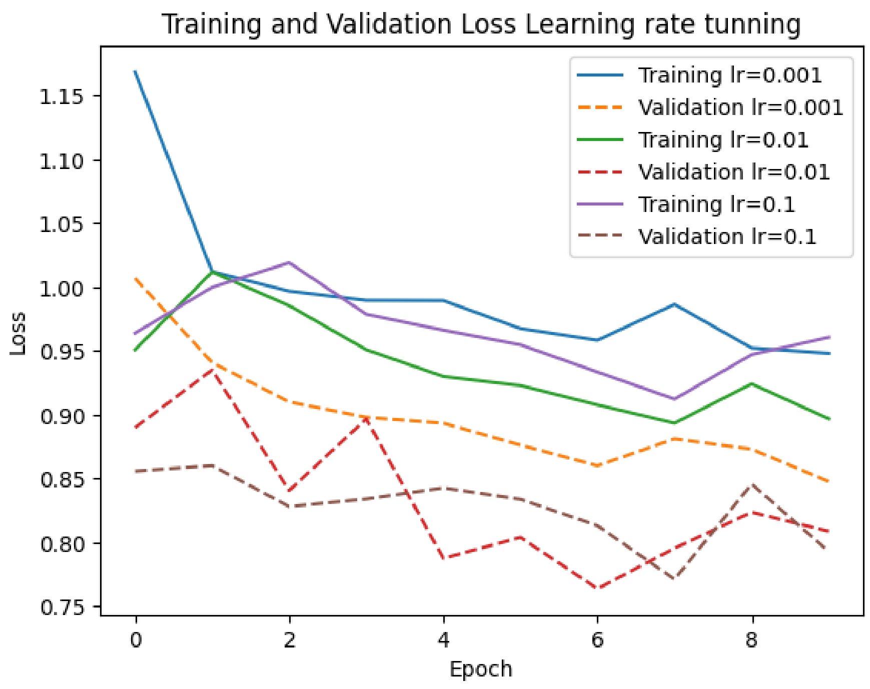
We got over 85 % for the samples. It is clear that the model is working fine.

Hyper parameter tunning

```
# tuning the learning rate
```

```
learning_rates = [0.001, 0.01, 0.1]
training_loss = dict()
validation_loss = dict()
best_valid_losses = dict()
for lr in learning_rates:
  optimizer = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.9)
  best_valid_loss = 1000
  train_losses = list()
  valid_losses = list()
  for i in range(10):
    train_loss = train_fn(train_loader , model , optimizer)
    valid_loss = eval_fn(val_loader , model )
    train_losses.append(train_loss)
    valid_losses.append(valid_loss)
    if valid_loss< best_valid_loss :
        torch.save(model.state_dict(), '/content/drive/MyDrive/Deep learning project final/Learning-rate{}.pt'.format(lr))
        print ("SAVED")
        best_valid_loss = valid_loss
    print(f"Epoch : {i+1} train_loss :{train_loss} valid_loss :{valid_loss}")
  training_loss.update({lr:train_losses})
  validation_loss.update({lr:valid_losses})
  best_valid_losses.update({lr:best_valid_loss})
```

We have selected three different learning rate for hyper parameter tunning. And created 3 models for each learning rate and selected the learning rate with lowest loss among them. By inspecting the graph plotted.

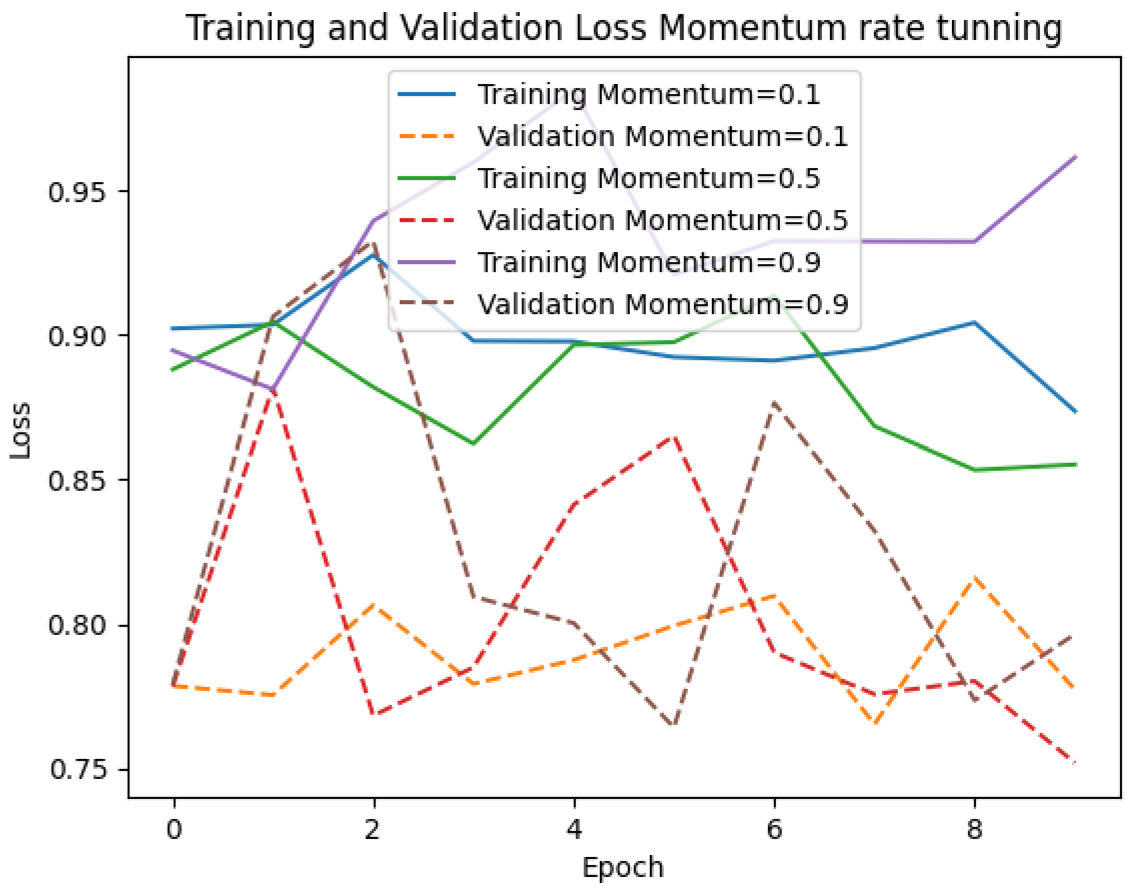


Looking at this its clear that validation error for leanring rate 0.1 is low so we have decided to consider that for further tunning.

Tunning momentum now.

In the similar way we have taken three different values for momentum and also we took the best learning rate from above tunning for further tunning.

And created 3 models to pick one of the best model with lowest loss again.



There is so much of loss shooting but when compared with other models it is clear that momentum with 0.1 has lowest loss. So we can consider this has the best momentum so far.

Next we have considered best learning rate and momentum so far and created a new model with those parameters. And used that model for testing on test dataset.

```python
def evaluate_test_data(index):
    image, mask = test_data_set[index]
    logits_mask = model(image.to(DEVICE).unsqueeze (0)) #(C, H, W) -> (1, C, H, W)
    print(logits_mask.shape)
    pred_mask_prob = torch.softmax(logits_mask, dim=1)  # (batch_size, num_classes, height, width)
    _, pred_mask = torch.max(pred_mask_prob, dim=1)  # (batch_size, height, width)
    show_image(image, mask, pred_mask)
    rgb_mask = mask_to_rgb(pred_mask)
    # Create an alpha mask where alpha=0 for background pixels and alpha=0.5 for all other pixels
    alpha_mask = np.where(np.all(rgb_mask == (0, 0, 0), axis=-1), 0, 0.9)
    # Create a new RGBA mask combining the RGB mask and alpha mask
    rgba_mask = np.concatenate([rgb_mask, alpha_mask[..., np.newaxis]], axis=-1)
    # Overlay the mask on the image
    plt.imshow(image.permute(1, 2, 0))
    plt.imshow(rgba_mask, alpha=0.5)
    plt.title("Picture with Predicted Mask Applied")
    plt.show()
```

This is a function evaluate_test_data that takes an index as input and does the following:

Get the image and mask from the test dataset using the input index.

Pass the image through the trained model to get the predicted mask probabilities using softmax activation.

Take the argmax of the predicted mask probabilities to get the predicted mask.

Show the original image, ground truth mask, and predicted mask using a show_image function.

Convert the predicted mask to an RGB format using a mask_to_rgb function.

Create an alpha mask that is transparent for background pixels and semi-transparent for all other pixels.

Combine the RGB mask and alpha mask to create an RGBA mask.

Overlay the RGBA mask on the original image using plt.imshow.

Show the resulting image with the predicted mask applied.

```
def IOU_cal(pred_masks, masks):
  lst = list()
  for pred_mask, mask in zip(pred_masks, masks):
    intersection = np.logical_and(pred_mask.cpu().numpy(), mask.cpu().numpy())
    union = np.logical_or(pred_mask.cpu().numpy(), mask.cpu().numpy())
    iou_score = np.sum(intersection) / np.sum(union)
    lst.append(iou_score)
    return sum(lst)/len(lst)
```

```
pred_masks = []
true_masks = []
for i in range(len(test_data_set)):
 image, mask = test_data_set[i]
 logits_mask = model(image.to(DEVICE).unsqueeze(0)) #(C, H, W) -> (1, C, H, W)
 pred_mask_prob = torch.softmax(logits_mask, dim=1) # (batch_size, num_classes, height, width)
 _, pred_mask = torch.max(pred_mask_prob, dim=1) # (batch_size, height, width)
 pred_masks.append(pred_mask)
 true_masks.append(mask)
mean_iou = IOU_cal(pred_masks, true_masks)
print("IoU is %s" % mean_iou)
```

```
IoU is 0.498
```

Finally, we have used intersection over union to find out the iou score, we got over 0.498 which is average rating.