Semantic Segmentation

Computer Vision Project

By : Neirouz Bouchaira

Table of Contents

Introduction - Context and Objectives of the Project

In the field of natural resource exploration, salt deposits are a major target for oil and gas companies. However, interpreting seismic images remains a complex challenge.

This project is part of **semantic segmentation**, a Deep Learning technique that assigns a class to each pixel in an image. Applied to seismic images, semantic segmentation aims **to accurately identify areas containing salt deposits**.

In this project, we will explore the **UNet architecture**, a convolutional neural network particularly suited for image segmentation. We will examine the impact of various **architectural modifications**, such as replacing max pooling with strided convolutions and removing skip connections, on the model's performance. We will also study the importance of choosing the inference

threshold and propose strategies to optimize the precision and recall of the segmentation.

The ultimate goal is to develop a high-performing and robust semantic segmentation model capable of accurately identifying salt deposits in seismic images. **The final model has an accuracy of and a loss of**.

I. Construction of the model

I.1. Data Download and Exploratory Phase

Data Download

We downloaded the images and masks from the directories `competition_data/train/images` and `competition_data/train/masks`. The image and mask files were sorted and loaded using the `os` and `tensorflow.keras.preprocessing.image` libraries.

Exploratory Phase

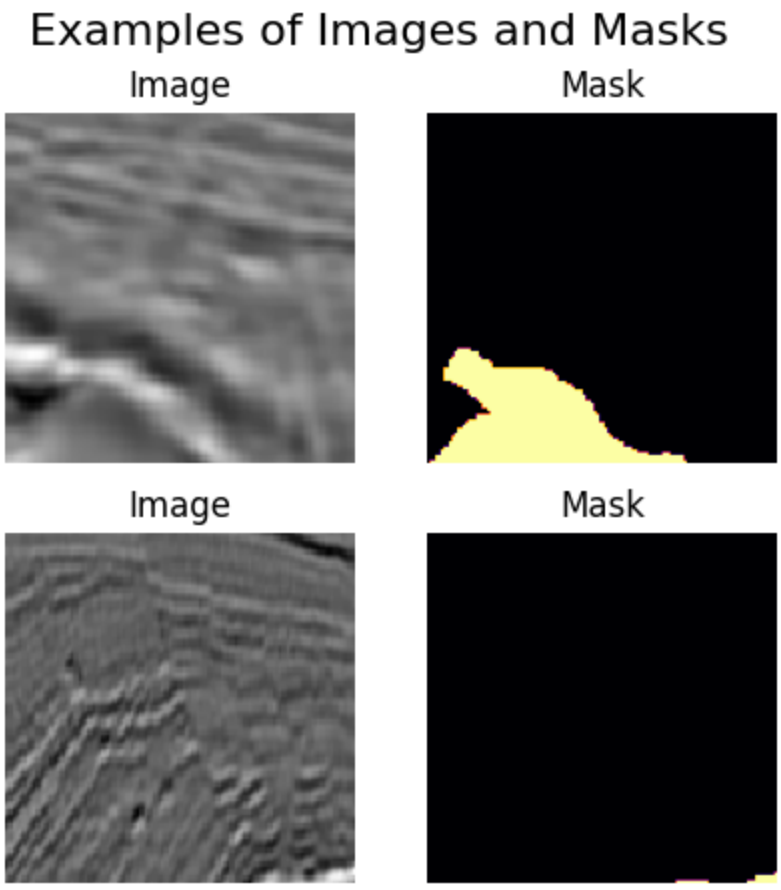
After loading the data, we split them into training, validation, and test sets with a distribution of 70%, 15%, and 15% respectively. The masks were normalized between 0 and 1 to facilitate model training.

Dataset Statistics

- **Training set size**: 2800
- **Validation set size**: 600
- **Test set size**: 600
- **Image size**: (128, 128, 3)
- **Mask size**: (128, 128, 1)

Data Visualization

We visualized some examples of images and masks to check the data quality and ensure that the masks correctly correspond to the images.



I.2. Baseline Model - UNet

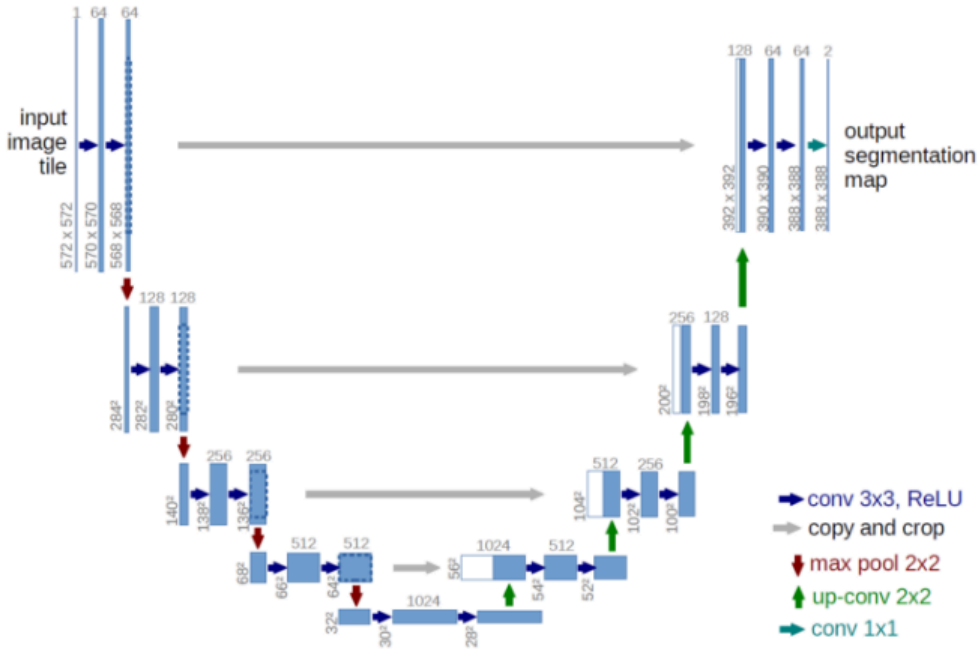
The U-Net architecture is a type of convolutional neural network designed for image segmentation tasks. It consists of two main parts: an encoder and a decoder.

The encoder, also known as the contracting path, is made up of several convolutional layers followed by pooling layers, which reduce the spatial dimension of the image while increasing the number of feature channels.

The decoder, or expansive path, uses transposed convolutional layers to increase the spatial dimension of the image and reduce the number of feature channels.

A key feature of U-Net is the use of skip connections between corresponding layers of the encoder and decoder, allowing low-level information to be transferred directly to the decoder layers, which helps preserve fine details in the image.

We will start by creating a U-Net model for 128x128x3 images that follows the given architecture:



I.3. Model Compilation and Training

To compile the model, we used the **Adam** optimizer with a `binary_crossentropy` loss function and the `accuracy` metric (defined with a threshold of 0.5). Then, we defined several callbacks to adjust the learning rate, stop training early, and save the best model based on validation loss. The model was trained for 100 epochs with a batch size of 32, using validation data to evaluate performance.

In [28]:
```python
# Define the loss function and optimizer
criterion = nn.BCELoss()
optimizer = optim.Adam(model.parameters())

# Learning rate scheduler
scheduler = ReduceLROnPlateau(optimizer, mode='min', factor=0.1, patience=5)

# Early stopping
class EarlyStopping:
    def __init__(self, patience=10, verbose=False):
        self.patience = patience
        self.verbose = verbose
        self.counter = 0
        self.best_loss = None
        self.early_stop = False

    def __call__(self, val_loss):
        if self.best_loss is None:
            self.best_loss = val_loss
        elif val_loss > self.best_loss:
            self.counter += 1
            if self.verbose:
```

```
                print(f'EarlyStopping counter: {self.counter} out of {self.p
            if self.counter >= self.patience:
                self.early_stop = True
        else:
            self.best_loss = val_loss
            self.counter = 0

early_stopping = EarlyStopping(patience=10, verbose=True)
```

Training

During model training, several metrics are tracked to evaluate its performance.

Accuracy measures the percentage of correct predictions on the training data.
Loss quantifies the model's error, with a lower value indicating better performance.
Validation accuracy and validation loss are the equivalents of these metrics but calculated on the validation data, allowing the evaluation of the model's generalization.
The learning rate controls the speed at which the model adjusts its weights; a rate that is too high can lead to fast but unstable convergence, while a rate that is too low can slow down the training.

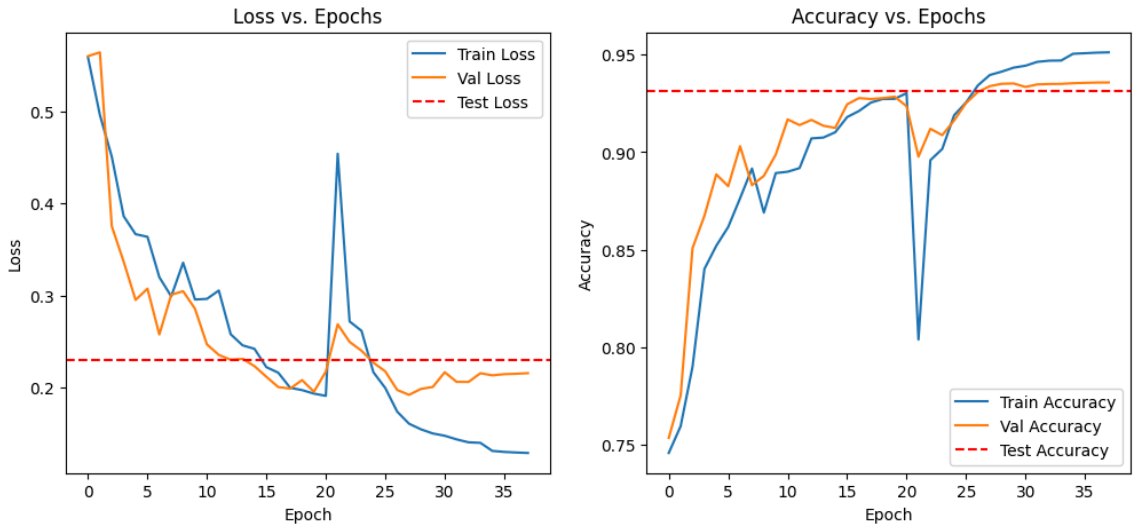
The training took approximately 20 minutes.

In [ ]:
```
for epoch in range(num_epochs=100):
    model.train()
    train_loss = 0.0
    train_correct = 0
    train_total = 0
    for images, masks in train_loader:
        optimizer.zero_grad()
        outputs = model(images.permute(0, 3, 1, 2))
        loss = criterion(outputs, masks.permute(0, 3, 1, 2))
        loss.backward()
        optimizer.step()
        train_loss += loss.item() * images.size(0)

        # Calculate accuracy
        predicted = (outputs > 0.5).float()
        train_correct += (predicted == masks.permute(0, 3, 1, 2)).sum().item
        train_total += masks.numel()

    train_loss /= len(train_loader.dataset)
    train_accuracy = train_correct / train_total
    train_losses.append(train_loss)
    train_accuracies.append(train_accuracy)

    model.eval()
    val_loss = 0.0
    val_correct = 0
    val_total = 0
```

```
    with torch.no_grad():
        for images, masks in val_loader:
            outputs = model(images.permute(0, 3, 1, 2))
            loss = criterion(outputs, masks.permute(0, 3, 1, 2))
            val_loss += loss.item() * images.size(0)

            # Calculate accuracy
            predicted = (outputs > 0.5).float()
            val_correct += (predicted == masks.permute(0, 3, 1, 2)).sum().it
            val_total += masks.numel()
            for i in range(outputs.size(0)):
                val_outputs.append(outputs[i].cpu().numpy())
            val_outputs_batch.append(outputs)

    val_loss /= len(val_loader.dataset)
    val_accuracy = val_correct / val_total
    val_losses.append(val_loss)
    val_accuracies.append(val_accuracy)

    print(f'Epoch {epoch+1}/{num_epochs}, Train Loss: {train_loss:.4f}, Trai

    scheduler.step(val_loss)
    print(f'Learning rate: {scheduler.optimizer.param_groups[0]["lr"]}')

    early_stopping(val_loss)
    if early_stopping.early_stop:
        print("Early stopping")
        break
val_outputs_original = val_outputs
```

I.4. Model Testing and Performance

We plotted the training and validation loss curves, and then the test loss curve to visualize the model's performance over epochs.



For this first model with a UNET architecture, we have a Test Loss of 0.2294 and a Test Accuracy of 93.16%.

- **Efficient Learning**: The UNET model learns efficiently, as shown by the decrease in loss and the increase in accuracy on the training and validation sets.
- **Slight Overfitting BUT Good Results**: There is a slight gap between training accuracy and validation accuracy, indicating slight overfitting. However, the gap is relatively small, suggesting that the model generalizes well.

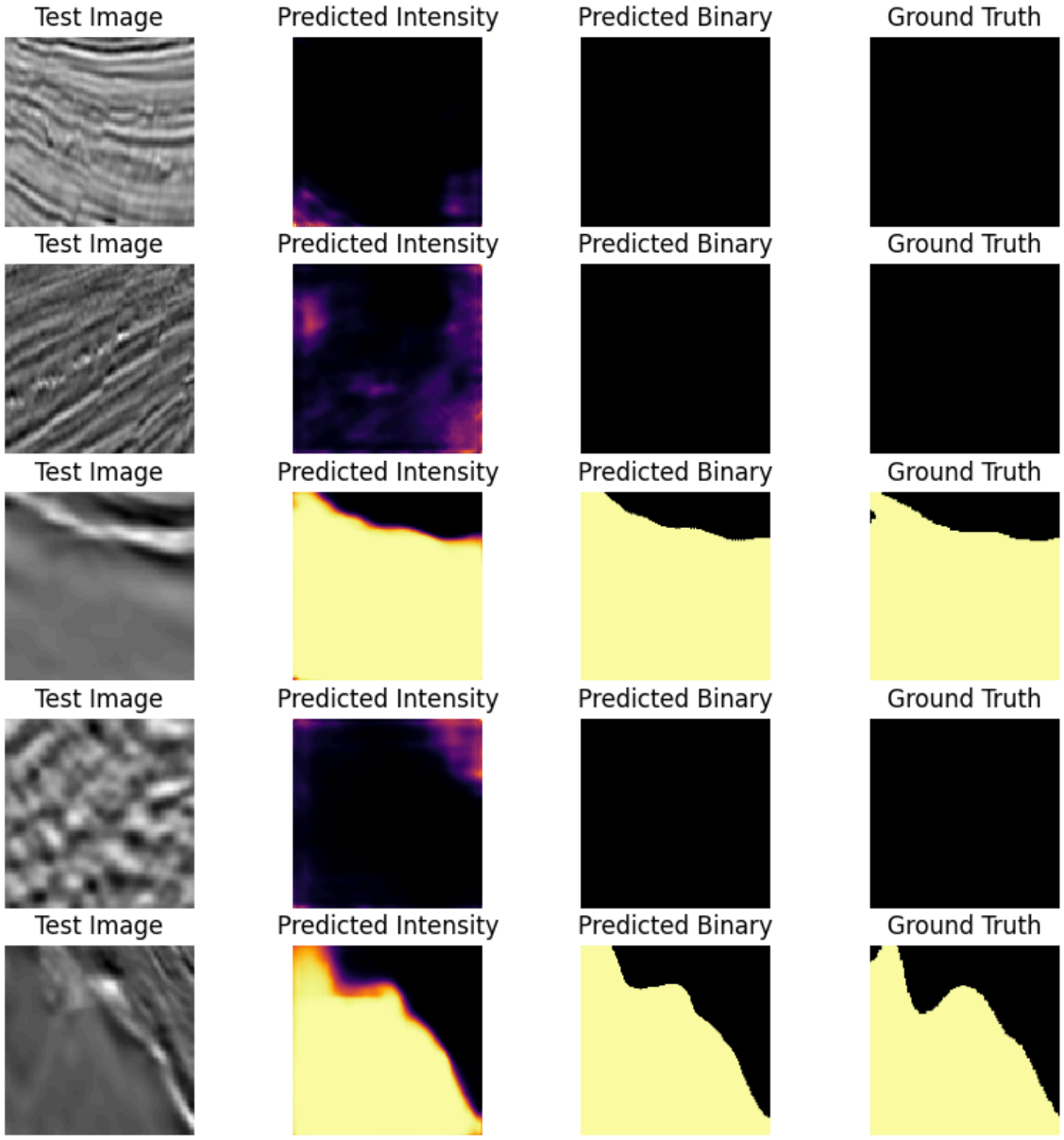
► Training was stopped early using an early stopping mechanism based on validation loss.

Points for Improvement to Consider for the Final Model:

Overfitting could be reduced by using regularization techniques such as dropout or L2 regularization.

I.5. Qualitative Analysis of Test Results

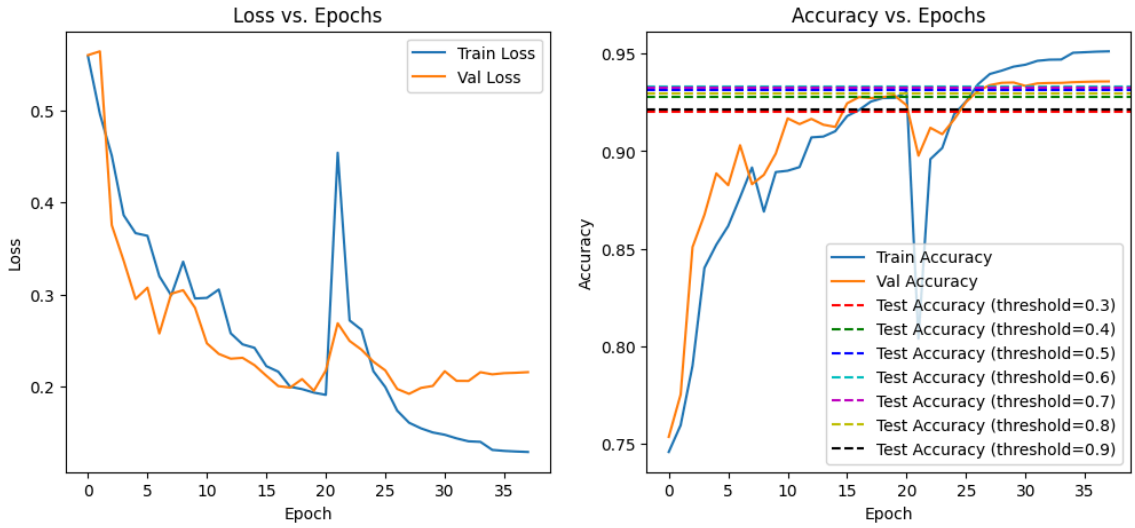
Qualitatively on the 5 examples, the model seems to predict salt deposits well.



I.6. Improvements: epoch=100, testing multiple thresholds

- Initially, when testing with 50 epochs, I reached the end without using Early stopping. Therefore, I decided to retest with a higher number of epochs to ensure the model reaches its best performance.
- I wanted to test my model with different accuracy definitions depending on the thresholds. So, I tried values other than 0.5: 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9.

Since the training took a lot of time, I considered retraining with these thresholds but did not do it. At the end of the notebook, I determine the optimal threshold for training and testing.



The threshold with the best results is 0.6, but since the difference is small, we keep 0.5 for the rest.

II. Architecture and training changes

II. 1. Replacing Max Pooling with Convolutions

Using convolutions with a different stride can indeed replace max pooling. By using a stride of 2, the size of the feature map is halved, similar to the effect of max pooling. However, there are notable differences:

1. **Information retention**: Convolutions with stride can retain more spatial information compared to max pooling, which only retains the maximum values.

2. **Additional parameters**: Convolutions with stride add extra parameters to the model, which can increase learning capacity but also the risk of overfitting.

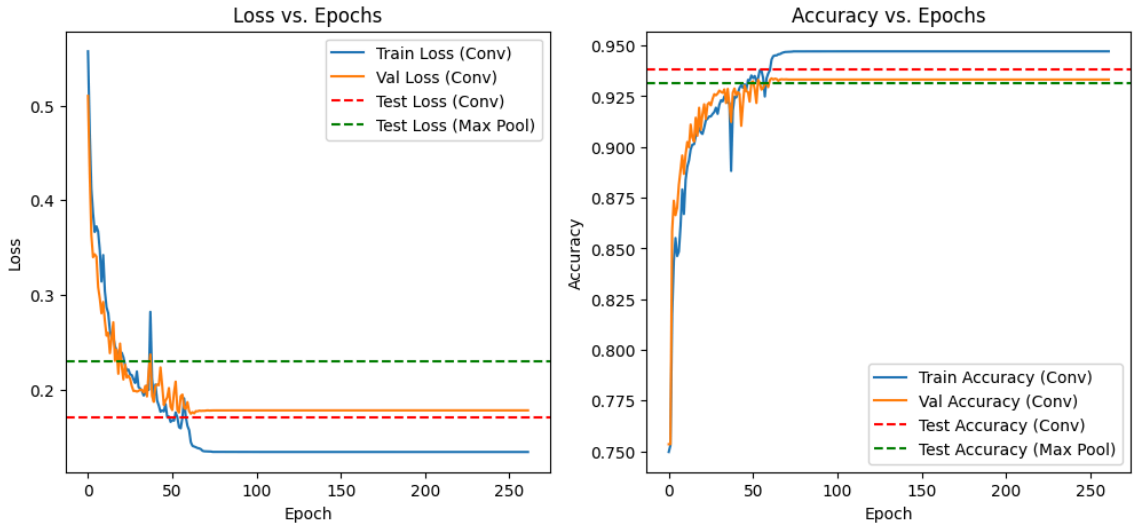
3. **Computational complexity**: Convolutions with stride can be more computationally expensive compared to max pooling.

> In some cases, convolutions with stride can improve segmentation accuracy by retaining more information, while in others, max pooling may suffice and be more efficient in terms of computation. As the computation time is expected to be more significant, I set 300 epochs.

In [ ]:
```
class UNet_maxpool_to_conv(nn.Module):
    def __init__(self):
        super(UNet_maxpool_to_conv, self).__init__()
```

```
        self.encoder = nn.Sequential(
            nn.Conv2d(3, 16, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(16, 16, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(16, 16, kernel_size=3, stride=2, padding=1),  # Replac

            nn.Conv2d(16, 32, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(32, 32, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(32, 32, kernel_size=3, stride=2, padding=1),  # Replac

            nn.Conv2d(32, 64, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(64, 64, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(64, 64, kernel_size=3, stride=2, padding=1),  # Replac

            nn.Conv2d(64, 128, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(128, 128, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(128, 128, kernel_size=3, stride=2, padding=1),  # Repl

            nn.Conv2d(128, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True)
        )
```

Results



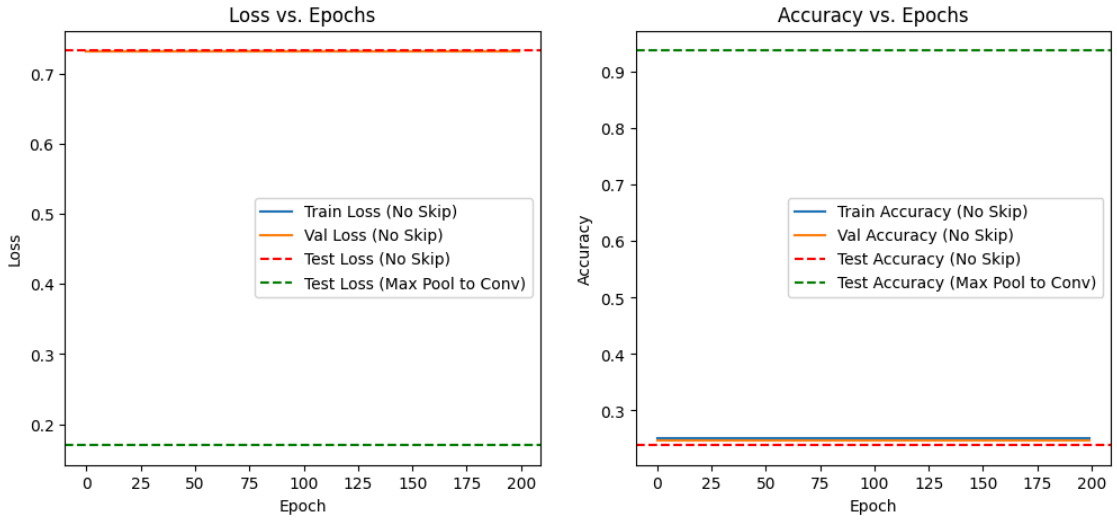
- **Similar Performance**: Both models (max pooling and convolutions) achieve comparable levels of accuracy and loss on the test set, indicating similar effectiveness for this task.

- **Training Time**: The model with max pooling trains much faster (20 minutes) than the one with convolutions (several hours), suggesting lower computational complexity.
- **Choice Based on Needs**: If training time is crucial, max pooling is preferable. If slightly higher performance is required and time is not a constraint, convolutions could be considered.

II. 2. Importance of Skip Connections

Skip connections are crucial in the U-Net architecture because they allow low-level information to be transferred from the encoding layers to the decoding layers. This helps preserve spatial details and fine features that might be lost during pooling and convolution operations. If we remove these skip connections, the model might struggle to reconstruct fine details in the decoding phase, leading to less accurate segmentation performance.

Results



The results without "skip connections" **are catastrophic**

The model fails to learn, with high loss and very low accuracy compared to the "Max Pool to Conv WITH SKIP CONNECTIONS" model which works very well: It achieves very low loss and very high accuracy, proving the importance of "skip connections" in this case.

II. 3. FCN and Auto-encoder

It is not strictly necessary to use an auto-encoder architecture for image segmentation. A fully convolutional network (FCN) architecture can also be used for this task. An FCN consists of a series of convolutions applied from input to output while maintaining the image size, i.e., width and height, throughout all convolutions.

However, there are significant differences between the two approaches:

1. **Detail Preservation**: Auto-encoder architectures, like U-Net, use skip connections to transfer low-level information from the encoding layers to the decoding layers. This helps preserve spatial details and fine features that might be lost during pooling and convolution operations. **An FCN without these skip connections might struggle to reconstruct fine details in the decoding phase.**

2. **Dimensionality Reduction**: Auto-encoders reduce the image dimension through pooling operations or convolutions with stride, allowing them to capture features at different scales. An FCN that maintains the image size throughout the convolutions might not effectively capture these multi-scale features.

3. **Computational Complexity**: FCNs that maintain the image size can be more computationally and memory-intensive, as they require many convolutions with large feature maps.

In [ ]:
```
class FCN_model(nn.Module):
    def __init__(self):
        super(FCN_model, self).__init__()

        self.conv1 = nn.Sequential(
            nn.Conv2d(3, 16, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(16, 16, kernel_size=3, padding=1),
            nn.ReLU(inplace=True)
        )

        self.conv2 = nn.Sequential(
            nn.Conv2d(16, 32, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(32, 32, kernel_size=3, padding=1),
            nn.ReLU(inplace=True)
        )

        self.conv3 = nn.Sequential(
            nn.Conv2d(32, 64, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(64, 64, kernel_size=3, padding=1),
            nn.ReLU(inplace=True)
        )
```

```
        self.conv4 = nn.Sequential(
            nn.Conv2d(64, 128, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(128, 128, kernel_size=3, padding=1),
            nn.ReLU(inplace=True)
        )

        self.conv5 = nn.Sequential(
            nn.Conv2d(128, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True)
        )

        self.final_conv = nn.Conv2d(256, 1, kernel_size=1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = self.conv3(x)
        x = self.conv4(x)
        x = self.conv5(x)
        x = self.final_conv(x)
        x = self.sigmoid(x)
        return x

# Create the model
FCN = FCN_model()
```

The model took 40 hours to run, so I had to stop it manually. **The training and validation accuracy are worse than the encoder-decoder models.**

II. 3. Threshold for inference

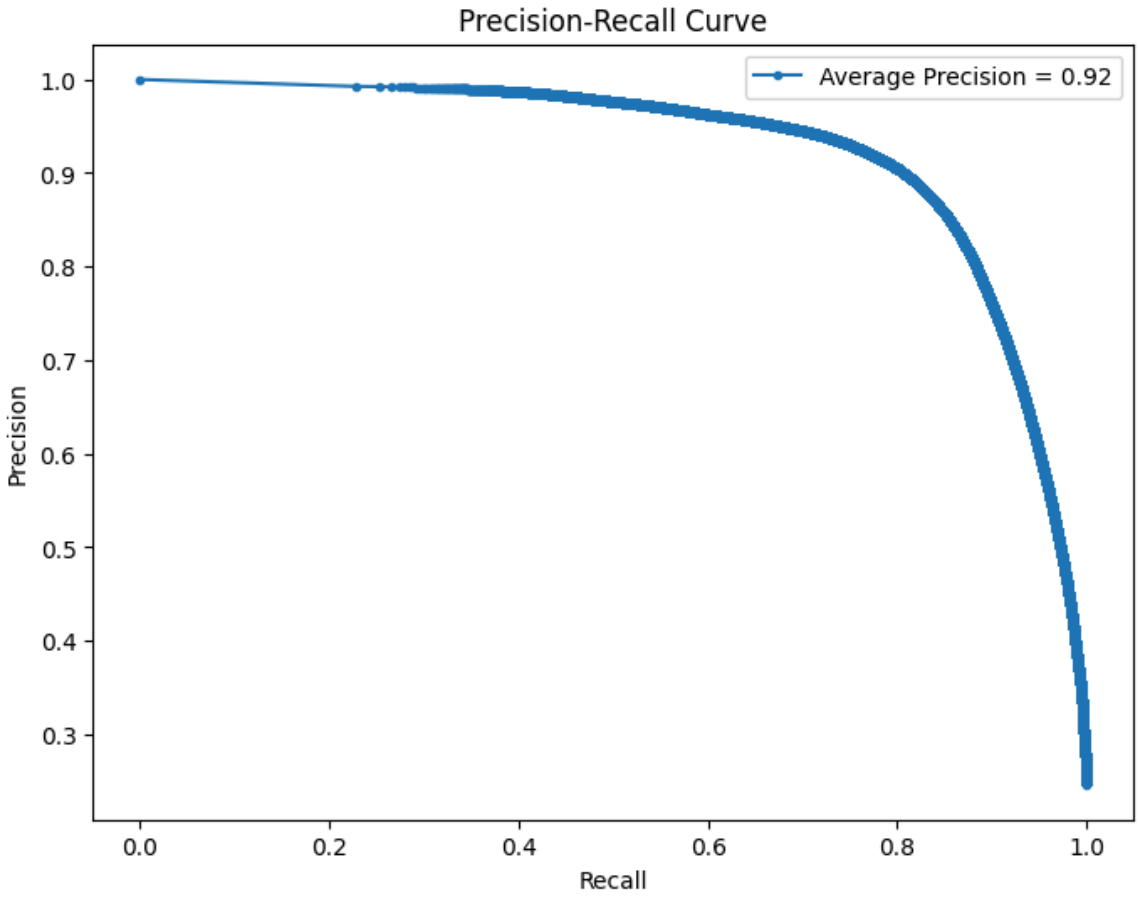
To define precision and recall in our use case, we need to understand what each metric represents:

- **Precision**: Precision is the ratio of the number of true positives (pixels correctly classified as salt) to the total number of pixels classified as salt (true positives + false positives). High precision means that few non-salt pixels are incorrectly classified as salt.
- **Recall**: Recall is the ratio of the number of true positives to the total number of pixels that are actually salt (true positives + false negatives). High recall means that most salt pixels are correctly detected.

- **Threshold Adjustment**: Instead of using a fixed threshold of 0.5, dynamically adjust the threshold based on the results of the precision-recall curve. The optimal threshold is the one that maximizes the F1 score, which is the harmonic mean of precision and recall.

In [10…

```python
from sklearn.metrics import precision_recall_curve, average_precision_sc

# Get the model predictions on the validation set
model.eval()
val_predictions = []
val_true_labels = []
with torch.no_grad():
    for images, masks in val_loader:
        outputs = model(images.permute(0, 3, 1, 2))
        val_predictions.append(outputs.cpu().numpy())
        val_true_labels.append(masks.cpu().numpy())

val_predictions = np.concatenate(val_predictions).ravel()
val_true_labels = np.concatenate(val_true_labels).ravel()

# Calculate precision-recall curve
precision, recall, thresholds = precision_recall_curve(val_true_labels,
average_precision = average_precision_score(val_true_labels, val_predict

# Plot precision-recall curve
plt.figure(figsize=(8, 6))
plt.plot(recall, precision, marker='.', label=f'Average Precision = {ave
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-Recall Curve')
plt.legend()
plt.show()

# Find the optimal threshold
f1_scores = 2 * (precision * recall) / (precision + recall)
optimal_threshold = thresholds[np.argmax(f1_scores)]
print(f'Optimal Threshold: {optimal_threshold:.2f}')
```




```
Optimal Threshold: 0.53
```

III. Final Model

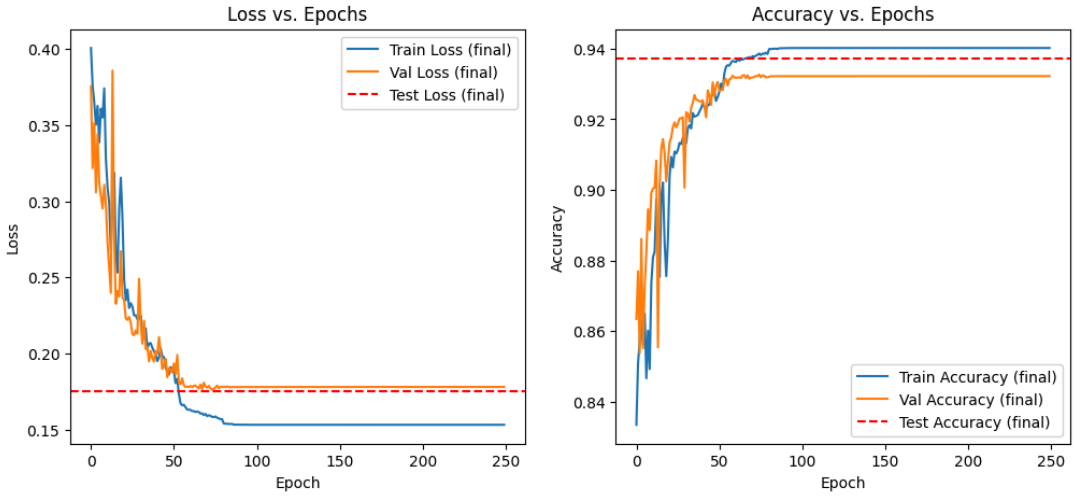
Our most optimized model to maximize accuracy and minimize loss is a UNET model with the following characteristics:

Architecture:

- Use of convolutions instead of max pooling.

Training and Testing:

- Threshold of 0.53 for accuracy definition and testing.
- Addition of weight decay (L2 regularization) to minimize overfitting.



IV. References

https://arxiv.org/abs/1505.04597

https://www.depends-on-the-definition.com/unet-keras-segmenting-images/

https://towardsdatascience.com/up-sampling-with-transposed-convolution-9ae4f2df52d0

Transposed convolution: Up sampling with Transposed Convolution