# Pytorch based U-Net for Fault Prediction

In [44]:
```
import segyio
import matplotlib.pyplot as plt
import numpy as np

import torch
from torch import nn
from tqdm.auto import tqdm
from torchvision import transforms
from torchvision.utils import make_grid
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
torch.manual_seed(0)
```

Out[44]:
```
<torch._C.Generator at 0x20db7448e50>
```

In [45]:
```
import pipreqs
! pipreqs --force
```

```
INFO: Successfully saved requirements file in C:\Users\repii\Music\Project\CNN-Based-Sei
smic-Fault-Prediction-main\requirements.txt
```

## Contracting Path

The contracting path is the encoder section of the U-Net which involves several downsampling steps. It consists of the repeated application of two 3x3 convolutions (unpadded), each followed by a ReLU and 2x2 max pooling operation with stride of 2 for downsampling. At each downsampling step we double the number of feature channels. Note: in the original U-Net framework, the resulting output has smaller size than the input. I am using a padding of (1,1) to make sure we get the same shape as input in Expanding block.

In [2]:
```
class ContractingBlock(nn.Module):
    '''
    ContractingBlock Class
    Performs two convolutions followed by a max pool operation.
    Values:
        input_channels: the number of channels to expect from a given input
    '''
    def __init__(self, input_channels):
        super(ContractingBlock, self).__init__()

        # Wou want to double the number of channels in the first convolution
        # and keep the same number of channels in the second.

        self.conv1 = nn.Conv2d(input_channels, 2*input_channels, kernel_size=3, padding
        self.conv2 = nn.Conv2d(2*input_channels, 2*input_channels, kernel_size=3, paddi
        self.activation = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(kernel_size=2, stride=2)


    def forward(self, x):
```

```
        '''
        Function for completing a forward pass of ContractingBlock:
        Given an image tensor, completes a contracting block and returns the transforme
        Parameters:
            x: image tensor of shape (batch size, channels, height, width)
        '''
        x = self.conv1(x)
        x = self.activation(x)
        x = self.conv2(x)
        x = self.activation(x)
        x = self.maxpool(x)
        return x
```

In [3]:
```
# unit test
def test_contracting_block(test_samples = 1, test_channels=1, test_size=254):
    test_block = ContractingBlock(test_channels)
    test_in = torch.randn(test_samples, test_channels, test_size, test_size)
    test_out_conv1 = test_block.conv1(test_in)

    # Make sure that the first convolution has the right shape
    print(test_out_conv1.shape)
```

In [4]:
```
test_contracting_block(128)
```

```
torch.Size([128, 2, 254, 254])
```

## Expanding Path

This is the decodign section of U-Net which has several upsampling steps. Original UNET needs this crop function in order to crop the image from contracting path and concatenate it to the current image on the expanding path - this is to form a skip connection. For our purpose, we want the input and output to be of same shape so we won't be applying these function in this experiment. However, I am leaving it here, if in case any of these is useful in the future.

Every step in expanding path consists of an upsampling of the feature map followed by a 2x2 convolution("up-convolution") that halves the number of feature channels, a concatenation with the correspondingly cropped feature map from contracting path and two 3x3 convolutions, each followed by a ReLU. The cropping is necessary due to the loss of border pixels in every convolution. (Later models basded on U-Net often use padding in the convolutions to prevent the size of the image from changing outside of the upsampling /downsampling steps)

In [5]:
```
class ExpandingBlock(nn.Module):
    '''
    ExpandingBlock Class
    Performs an upsampling, a convolution, a concatenation of its two inputs,
    followed by two more convolutions.
    Values:
        input_channels: the number of channels to expect from a given input
    '''
    def __init__(self, input_channels):
        super(ExpandingBlock, self).__init__()
```

```
        # "Every step in the expanding path consists of an upsampling of the feature ma
        self.upsample = nn.Upsample(scale_factor=2, mode='bilinear', align_corners=True
        self.conv1 = nn.Conv2d(input_channels, input_channels//2, kernel_size=3, paddin
        self.conv2 = nn.Conv2d(input_channels, input_channels//2, kernel_size=3, paddin
        self.conv3 = nn.Conv2d(input_channels//2, input_channels//2, kernel_size=3, pad

        self.activation = nn.ReLU() # "each followed by a ReLU"

    def forward(self, x, skip_con_x):
        '''
        Function for completing a forward pass of ExpandingBlock:
        Given an image tensor, completes an expanding block and returns the transformed
        Parameters:
            x: image tensor of shape (batch size, channels, height, width)
            skip_con_x: the image tensor from the contracting path (from the opposing b
                    for the skip connection

        Note: In the original Unet implementation, the output shape is smaller than the
        requires a skip connection layer size to be matched with current layer.
        In this application, since our input and output are to be same size, we will no
        skip connection layer. However, there is a placehold commented, if needed in fu

        '''
        x = self.upsample(x)
        x = self.conv1(x)
        x = torch.cat([x, skip_con_x], axis=1)
        x = self.conv2(x)
        x = self.activation(x)
        x = self.conv3(x)
        x = self.activation(x)
        return x
```

In [6]:

```
#UNIT TEST
def test_expanding_block(test_samples=1, test_channels=64*16, test_size=32):
    test_block = ExpandingBlock(test_channels)
    skip_con_x = torch.randn(test_samples, test_channels // 2, test_size * 2 + 6, test_
    x = torch.randn(test_samples, test_channels, test_size, test_size)
    x = test_block.upsample(x)

    # Make sure that the first convolution produces the right shape
    print(x.shape)

test_expanding_block()
# print("Success!")
```

```
torch.Size([1, 1024, 64, 64])
```

## Final Layer

This layer takes in a tensor with arbitrarily many tensors and produces a tensor with the same number of pixels but with the correct number of the output channels. At the final layer, a 1x1 convolution is used to map each 64-component feature vectore to the desired number of classes. In total the network has 23 convolutional layers

In [7]:

```
class FeatureMapBlock(nn.Module):
    '''
```

```
    FeatureMapBlock Class
    The final layer of a UNet -
    maps each pixel to a pixel with the correct number of output dimensions
    using a 1x1 convolution.
    Values:
        input_channels: the number of channels to expect from a given input
    '''
    def __init__(self, input_channels, output_channels):
        super(FeatureMapBlock, self).__init__()

        # "Every step in the expanding path consists of an upsampling of the feature ma
        self.conv = nn.Conv2d(input_channels, output_channels, kernel_size=1)


    def forward(self, x):
        '''
        Function for completing a forward pass of FeatureMapBlock:
        Given an image tensor, returns it mapped to the desired number of channels.
        Parameters:
            x: image tensor of shape (batch size, channels, height, width)
        '''
        x = self.conv(x)
        return x
```

# U-Net

In [9]:

```
class UNet(nn.Module):
    '''
    UNet Class
    A series of 4 contracting blocks followed by 4 expanding blocks to
    transform an input image into the corresponding paired image, with an upfeature
    layer at the start and a downfeature layer at the end
    Values:
        input_channels: the number of channels to expect from a given input
        output_channels: the number of channels to expect for a given output
    '''
    def __init__(self, input_channels, output_channels, hidden_channels=64):
        super(UNet, self).__init__()
        # "Every step in the expanding path consists of an upsampling of the feature ma
        self.upfeature = FeatureMapBlock(input_channels, hidden_channels)
        self.contract1 = ContractingBlock(hidden_channels)
        self.contract2 = ContractingBlock(hidden_channels * 2)
        self.contract3 = ContractingBlock(hidden_channels * 4)
        self.contract4 = ContractingBlock(hidden_channels * 8)
        self.expand1 = ExpandingBlock(hidden_channels * 16)
        self.expand2 = ExpandingBlock(hidden_channels * 8)
        self.expand3 = ExpandingBlock(hidden_channels * 4)
        self.expand4 = ExpandingBlock(hidden_channels * 2)
        self.downfeature = FeatureMapBlock(hidden_channels, output_channels)

    def forward(self, x):
        '''
        Function for completing a forward pass of UNet:
        Given an image tensor, passes it through U-Net and returns the output.
        Parameters:
            x: image tensor of shape (batch size, channels, height, width)
        '''
```

```
        x0 = self.upfeature(x)
        x1 = self.contract1(x0)
        x2 = self.contract2(x1)
        x3 = self.contract3(x2)
        x4 = self.contract4(x3)

        x5 = self.expand1(x4, x3)
        x6 = self.expand2(x5, x2)
        x7 = self.expand3(x6, x1)
        x8 = self.expand4(x7, x0)
        xn = self.downfeature(x8)
        return xn
```

In [10]:
```
# unit test
test_unet = UNet(1,1)
# print(test_unet(torch.randn(1, 1, 256, 256)).shape)
print(test_unet(torch.randn(1, 1, 512, 512)).shape)
```

```
torch.Size([1, 1, 512, 512])
```

# Prepare for modelling

In [11]:
```
# First load datasets

filename_pp = "D:\Fault identification\Equinor Synthetic model\issap20_Pp.sgy"
filename_ai = "D:\Fault identification\Equinor Synthetic model\issap20_AI.sgy"
filename_fault = "D:\Fault identification\Equinor Synthetic model\issap20_Fault.sgy"
```

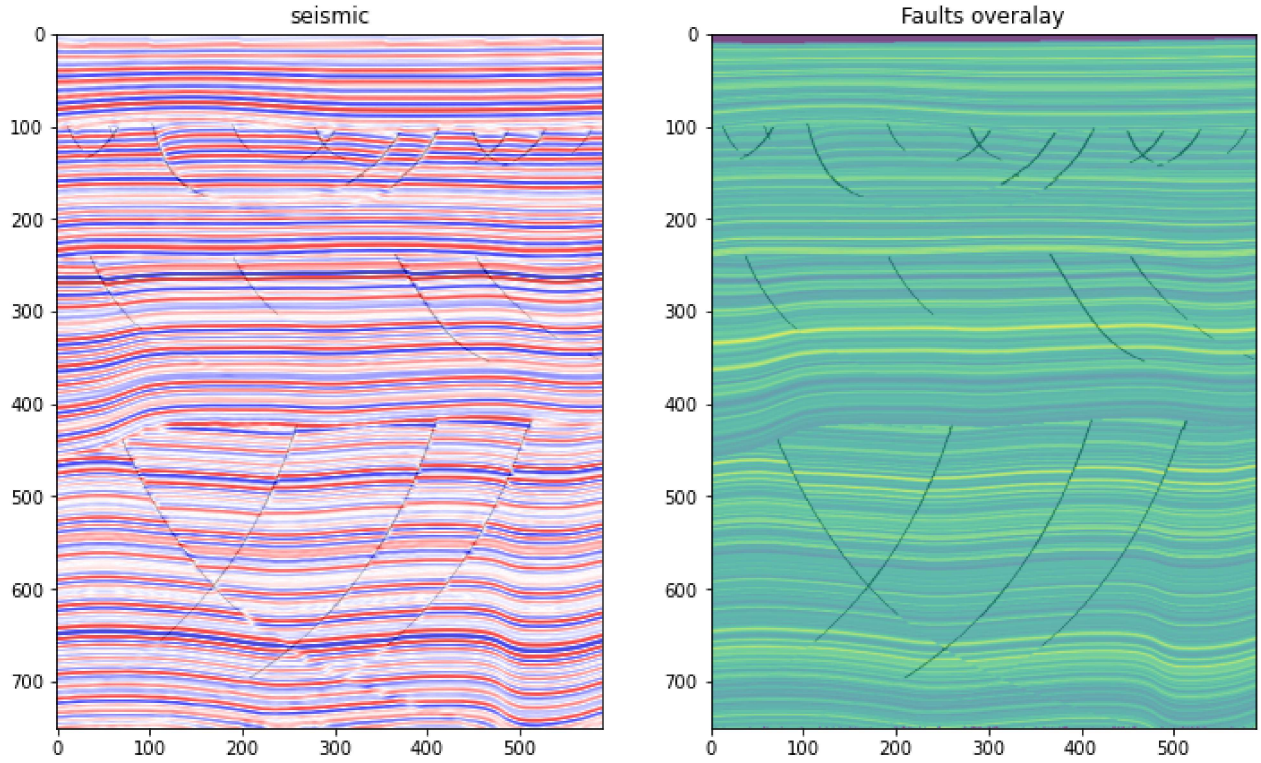
In [12]:
```
# Note: the the xline header info location is at segyio.su.cdpx [181]
def segy2numpy(filename: str) -> np.array:
    with segyio.open(filename, xline=181) as segyfile:
        return segyio.tools.cube(segyfile)

seismic = segy2numpy(filename_pp)
ai = segy2numpy(filename_ai)
fault = segy2numpy(filename_fault)
f"Number of inlines: {seismic.shape[0]}, crosslines: {seismic.shape[1]}, samples: {seis
```

Out[12]:
```
'Number of inlines: 101, crosslines: 589, samples: 751'
```

In [13]:
```
# plot inline
ILINE=10
plt.figure(figsize=(12, 10))
title = ['seismic', 'Faults overalay']
cmaps = ["seismic", None]

for i, cube in enumerate([seismic, ai]):
    plt.subplot(1, 2, i+1)
    plt.title(title[i])
    plt.imshow(cube[ILINE,:, :].T, cmap=cmaps[i])
    plt.imshow(fault[ILINE,:, :].T, cmap='Greys',  vmin=0, vmax=1, alpha=0.3)
    plt.axis()
plt.show()
```




## Cropping Input to the shape

The function below is to crop or pad the input data to make into the shape that is multiple of 32x32 or any combination. The real shape of the synthetic data (589x751) was causing problem. This is a quick workaround to get some experiment going on. Ideally, we want to use the actual shape of the data, until I find the solution.

In [14]:

```
def crop_input(image, new_shape):
    '''
    Function for cropping an image tensor: Given an image tensor and the new shape,
    crops to the center pixels.
    Parameters:
        image: image tensor of shape (batch size, channels, height, width)
        new_shape: a torch.Size object with the shape you want x to have
    '''

    h, w = image.shape[0], image.shape[1]   # values inside the brackets depends on the
    new_h, new_w = new_shape[0], new_shape[1]

    start_h = int((h - new_h + 1)/2)
    start_w = int((w - new_w + 1)/2)

    cropped_image = image[start_h:start_h + new_h, start_w:start_w + new_w]

    return cropped_image

# Function to pad
import torch.nn.functional as F

def pad_to(image, new_shape):
    '''
    Function for padding an image tensor.
```

```
    If somehow the expanding layer output and the skip connection doesn't match,
    these might be helpful.
    '''
    h, w = image.shape[0], image.shape[1]
    new_h, new_w = new_shape[0], new_shape[1]

    inc_h, inc_w = new_h -h, new_w - w
    left, right = 0, inc_w
    top, bottom = 0, inc_h
    pads = left, right, top, bottom

    # zero-padding by default.
    # See others at https://pytorch.org/docs/stable/nn.functional.html#torch.nn.functio
    padded_image = F.pad(image, pads, "constant", 0)

    return padded_image
```

In [16]:
```
new_shape = (512, 512)
image_list = []
fault_list = []
for i in range(101):
    images = torch.from_numpy(seismic[i])
    image_list.append(crop_input(images, new_shape ).unsqueeze(0))

    faults = torch.from_numpy(fault[i])
    fault_list.append(crop_input(faults, new_shape).unsqueeze(0))
```

In [17]:
```
image_list[0].shape
```

Out[17]:
```
torch.Size([1, 512, 512])
```

In [18]:
```
# Loading data into Pytorch Dataset Utitlity
volumes = torch.stack(image_list)
labels = torch.stack(fault_list)
dataset = torch.utils.data.TensorDataset(volumes, labels)
```

In [19]:
```
volumes.shape, labels.shape
```

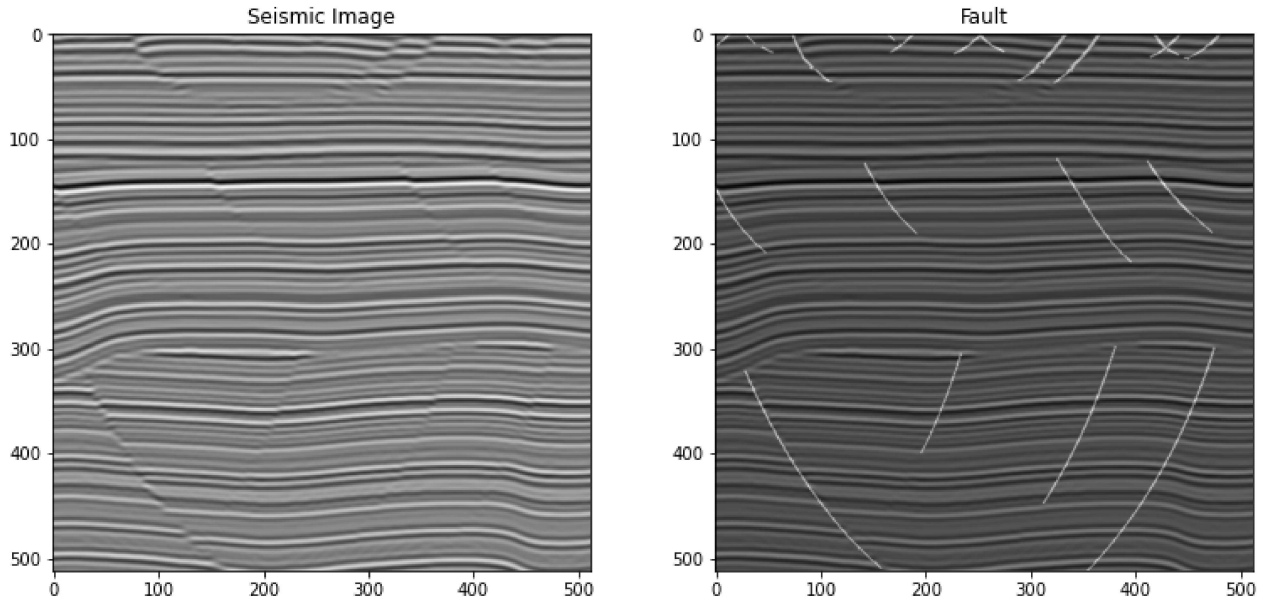
Out[19]:
```
(torch.Size([101, 1, 512, 512]), torch.Size([101, 1, 512, 512]))
```

In [20]:
```
# One last sanity check
fig = plt.figure(figsize=(20,20))

ax = fig.add_subplot(331)
plt.imshow(volumes[0].T, cmap="gray");
ax.set_title("Seismic Image")

ax = fig.add_subplot(332)
ax.imshow(volumes[0].T, cmap = 'gray')
ax.imshow(labels[0].T, cmap = 'gray', vmin=0, vmax=1, alpha=0.4)
ax.set_title("Fault")
plt.show();
```

```
C:\Users\repii\AppData\Local\Temp/ipykernel_39700/1976699852.py:5: UserWarning: The use
of `x.T` on tensors of dimension other than 2 to reverse their shape is deprecated and i
t will throw an error in a future release. Consider `x.mT` to transpose batches of matri
cesor `x.permute(*torch.arange(x.ndim - 1, -1, -1))` to reverse the dimensions of a tens
or. (Triggered internally at  C:\actions-runner\_work\pytorch\pytorch\builder\windows\py
torch\aten\src\ATen\native\TensorShape.cpp:2318.)
  plt.imshow(volumes[0].T, cmap="gray");
```




In [21]:

```
# Special function to display images side by side after training

def show_tensor_images(image, fault, pred, num_images=25, size=(1, 28, 28)):
    '''
    Function for visualizing images: Given a tensor of images, number of images, and
    size per image, plots and prints the images in an uniform grid.
    '''
    # image_shifted = (image_tensor + 1) / 2
    #image_shifted = image_tensor

    image_unflat = image.detach().cpu()
    fault_unflat = fault.detach().cpu()
    pred_unflat = pred.detach().cpu()


    fig = plt.figure(figsize=(12,15))

    ax = fig.add_subplot(331)
    ax.imshow(image_unflat.squeeze(), cmap = 'gray')
    ax.set_title("Seismic Image")

    ax = fig.add_subplot(332)
    ax.imshow(image_unflat.squeeze(), cmap = 'gray')
    ax.imshow(fault_unflat.squeeze(), cmap = 'gray', vmin=0, vmax=1, alpha=0.4)
    ax.set_title("Fault")

    ax = fig.add_subplot(333)
    ax.imshow(image_unflat.squeeze(), cmap = 'gray')
    ax.imshow(pred_unflat.squeeze(), cmap = 'gray', vmin=0, vmax=1, alpha=0.4)
    ax.set_title("Predicted Fault")

    plt.show()
```

## Training

- criterion: the loss function
- n_epochs: the number of times you iterate through the entire dataset when training
- input_dim: the number of channels of the input image
- label_dim: the number of channels of the output image
- display_step: how often to display/visualize the images
- batch_size: the number of images per forward/backward pass
- lr: the learning rate
- initial_shape: the size of the input image (in pixels)
- target_shape: the size of the output image (in pixels)
- device: the device type

In [22]:
```
device = torch.device("cuda:0" if (torch.cuda.is_available()) else "cpu")
print(device)
```

```
cpu
```

In [46]:
```
# Hyperparameters
criterion = nn.BCEWithLogitsLoss()
n_epochs =20
input_dim = 1
label_dim = 1
display_step = 500
batch_size = 1
lr = 0.0002
initial_shape = 512
target_shape = 512
device = 'cpu'
```

In [47]:
```
def train():
    dataloader = DataLoader(
        dataset,
        batch_size=1,
        shuffle=True)
    unet = UNet(input_dim, label_dim).to(device)
    unet_opt = torch.optim.Adam(unet.parameters(), lr=lr)
    cur_step = 0

    train_losses = []

    for epoch in range(n_epochs):

        for real, labels in tqdm(dataloader):

            cur_batch_size = len(real)
            # Flatten the image
            real = real.to(device)
            labels = labels.to(device)


            ### Update U-Net ###
```

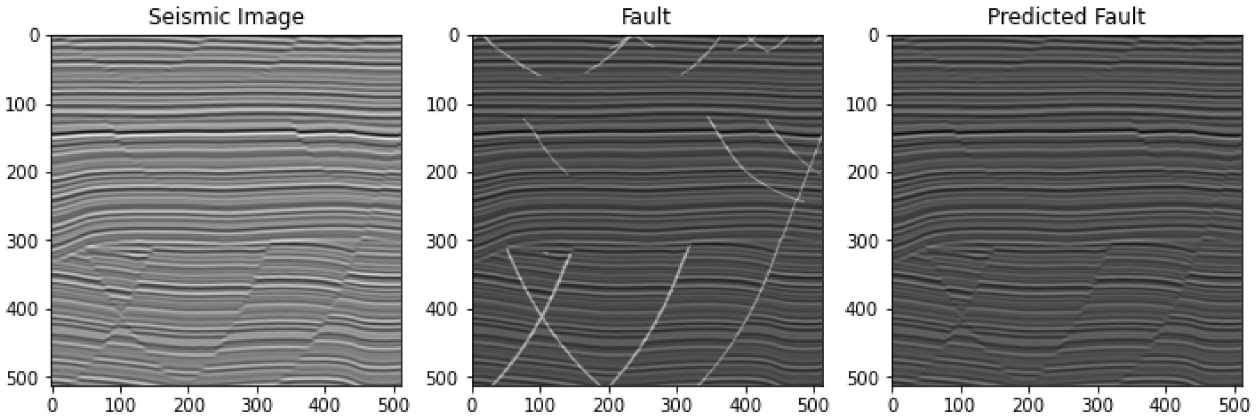
```
            unet_opt.zero_grad()
            pred = unet(real)
            #print(pred.shape)
            unet_loss = criterion(pred, labels)
            unet_loss.backward()
            unet_opt.step()
            cur_step += 1
        print(f"Epoch {epoch}: Step {cur_step}: U-Net loss: {unet_loss.item()}")
        show_tensor_images(real.T, labels.T, torch.sigmoid(pred).T, size=(input_dim, ta
        train_losses.append(unet_loss)


    return unet, pred, train_losses
```

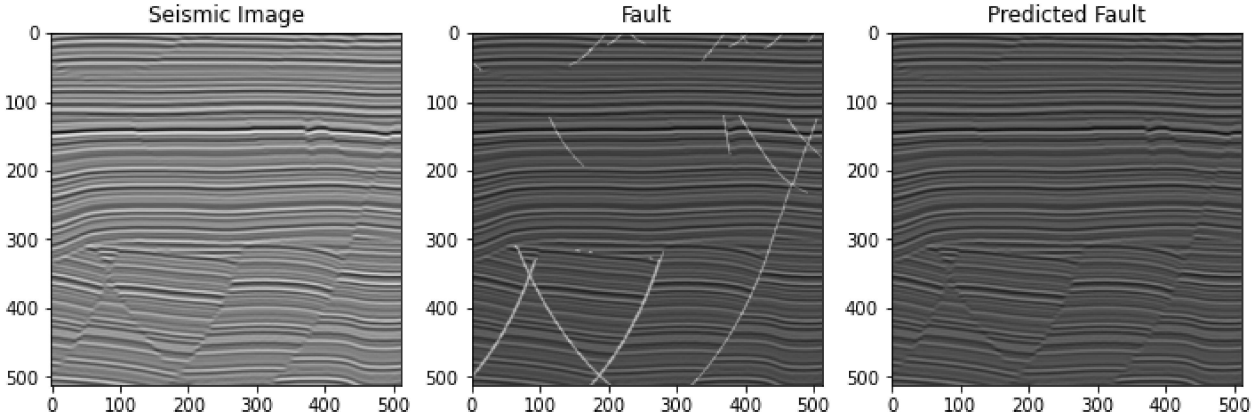
In [48]:

```
model, pred, loss = train()
```

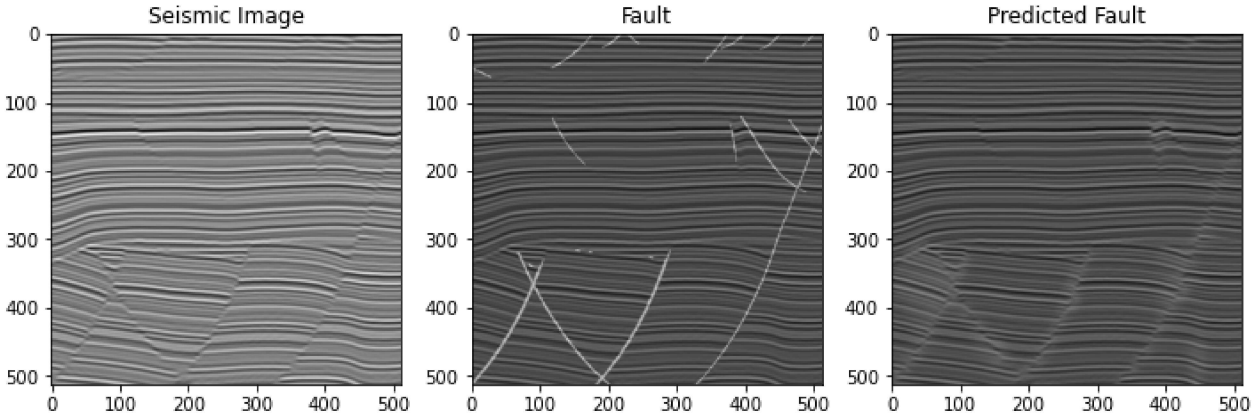
```
Epoch 0: Step 101: U-Net loss: 0.0961155965924263
```



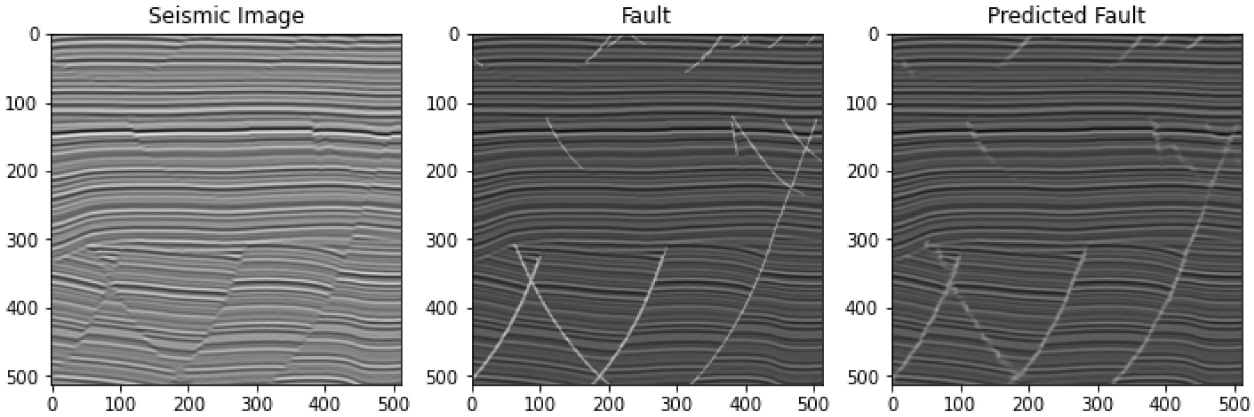

```
Epoch 1: Step 202: U-Net loss: 0.08524934202432632
```



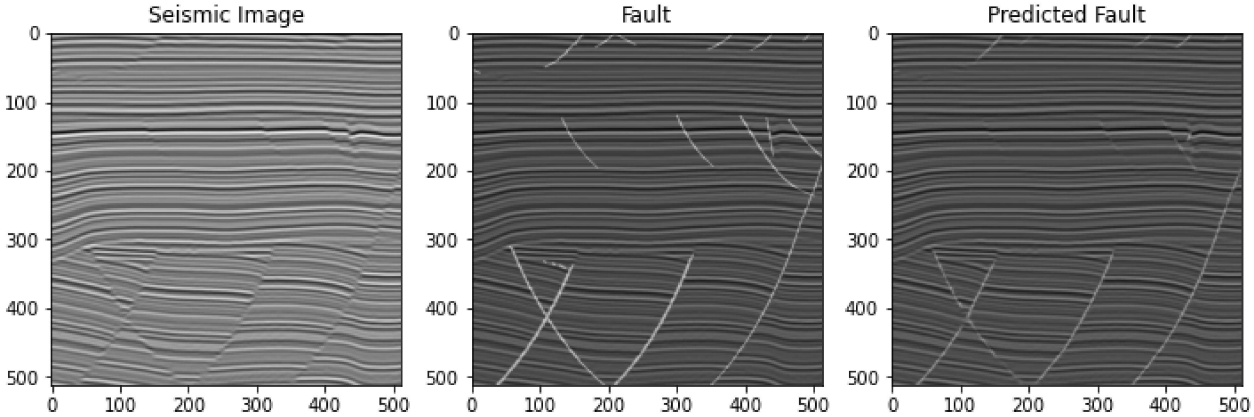

```
Epoch 2: Step 303: U-Net loss: 0.06665940582752228
```



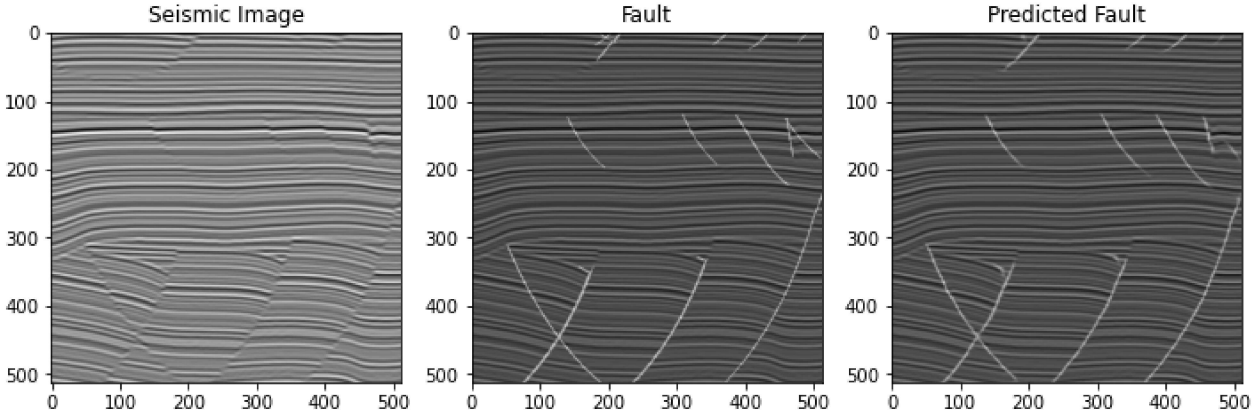

```
Epoch 3: Step 404: U-Net loss: 0.03519545868039131
```



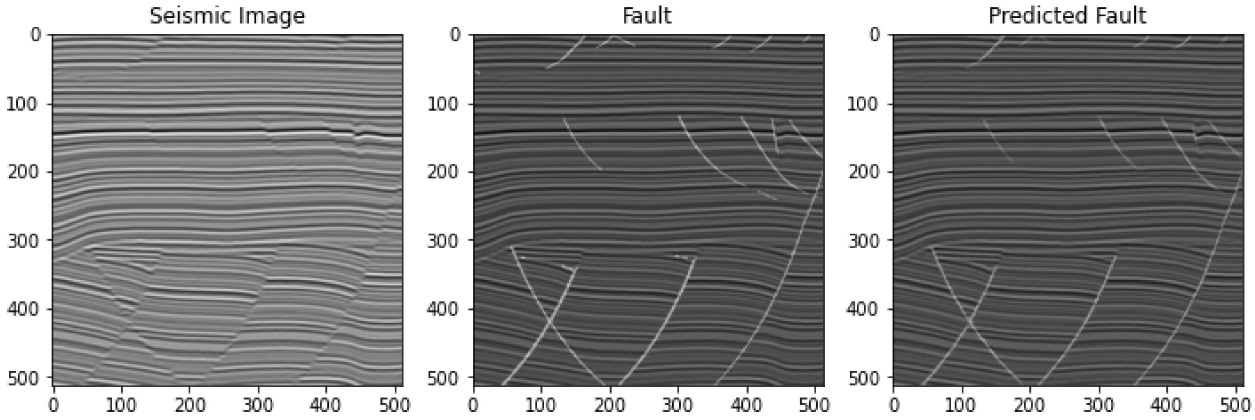

```
Epoch 4: Step 505: U-Net loss: 0.03571798652410507
```



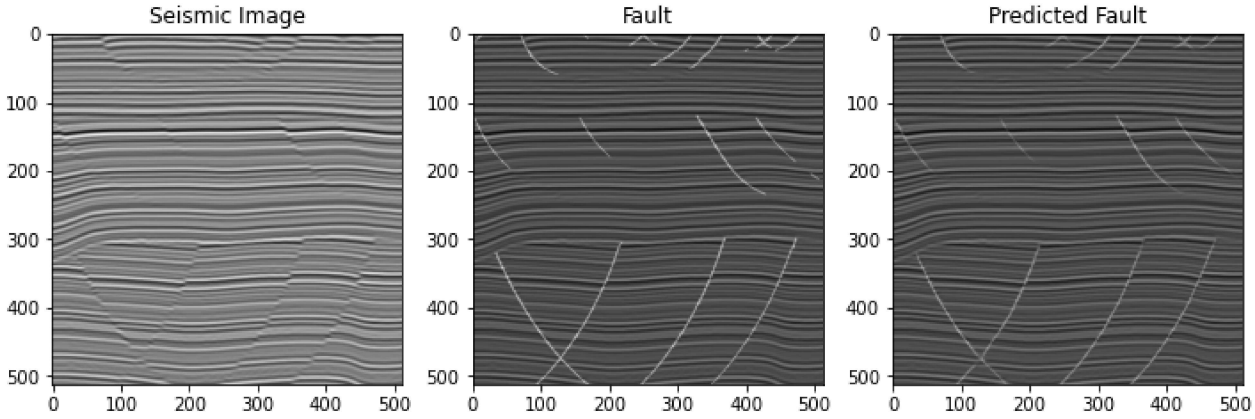

```
Epoch 5: Step 606: U-Net loss: 0.018819835036993027
```



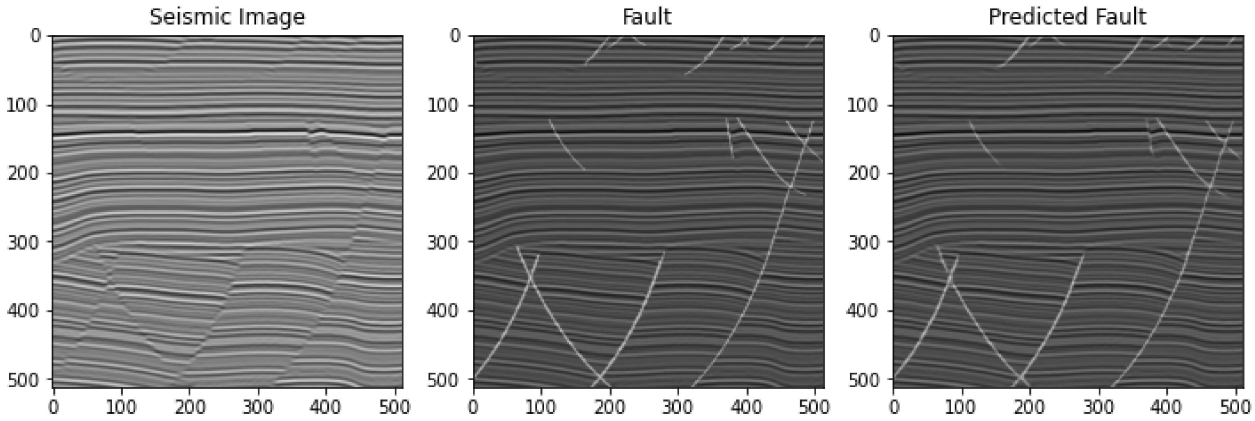
```
Epoch 6: Step 707: U-Net loss: 0.022425450384616852
```



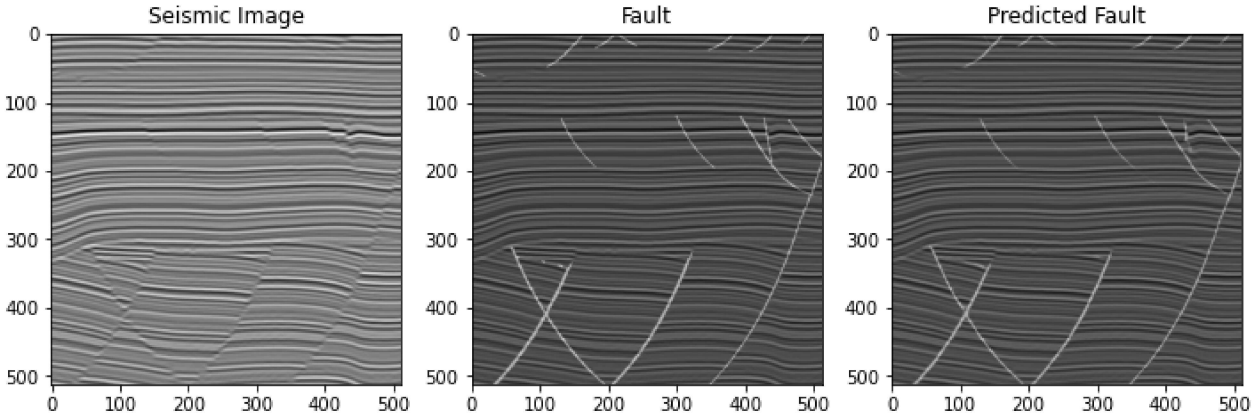

```
Epoch 7: Step 808: U-Net loss: 0.017194250598549843
```



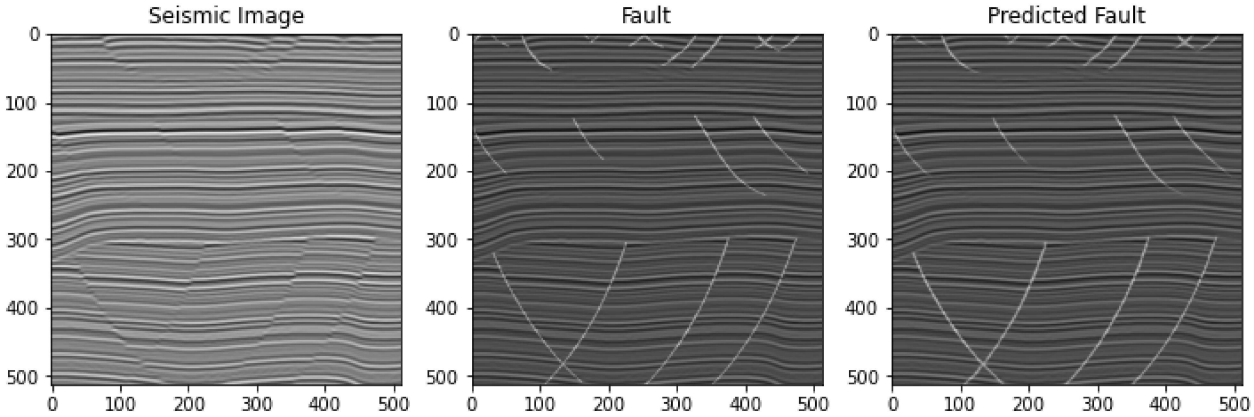

```
Epoch 8: Step 909: U-Net loss: 0.01608785055577755
```



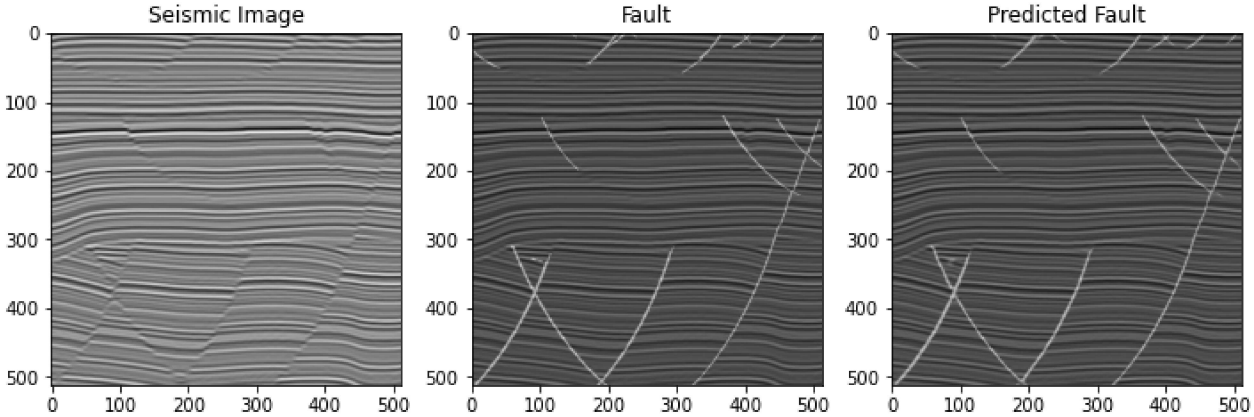

```
Epoch 9: Step 1010: U-Net loss: 0.011765312403440475
```



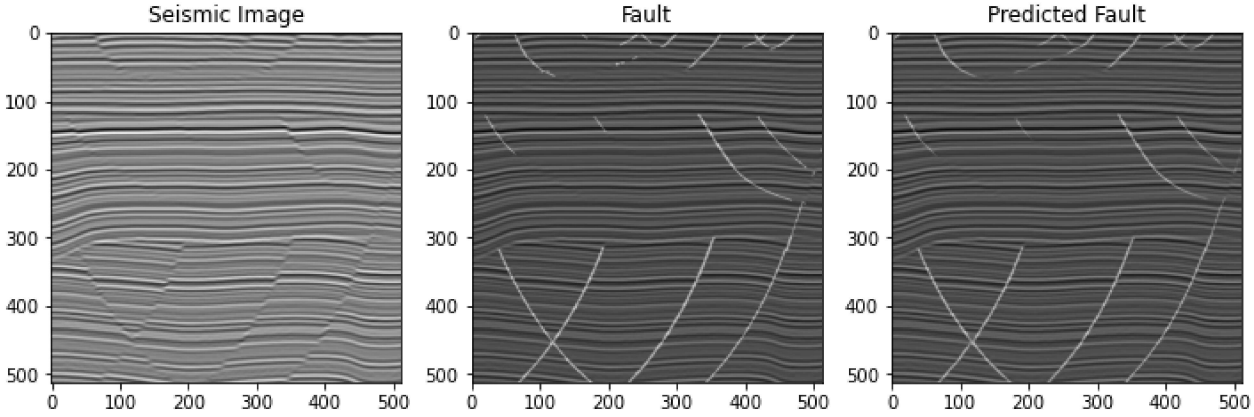

```
Epoch 10: Step 1111: U-Net loss: 0.01531960628926754
```



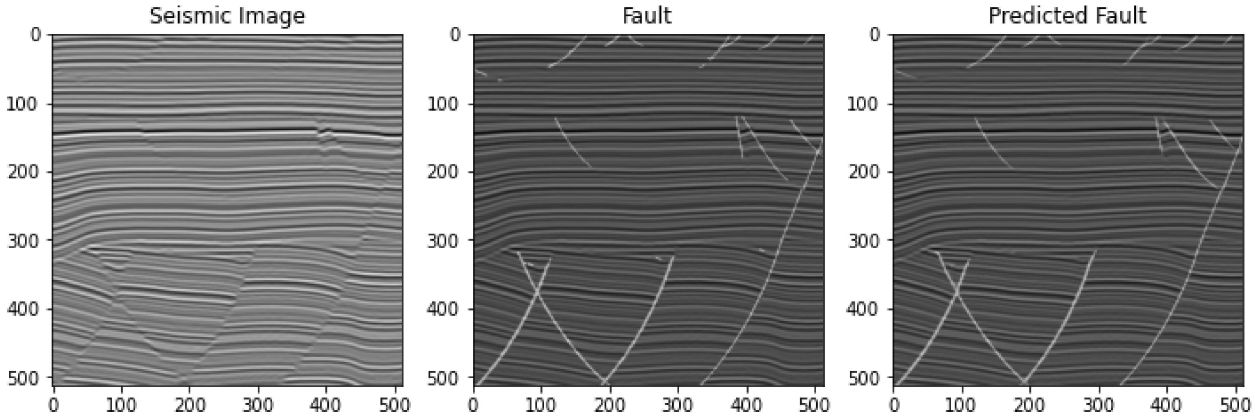

```
Epoch 11: Step 1212: U-Net loss: 0.011064683087170124
```



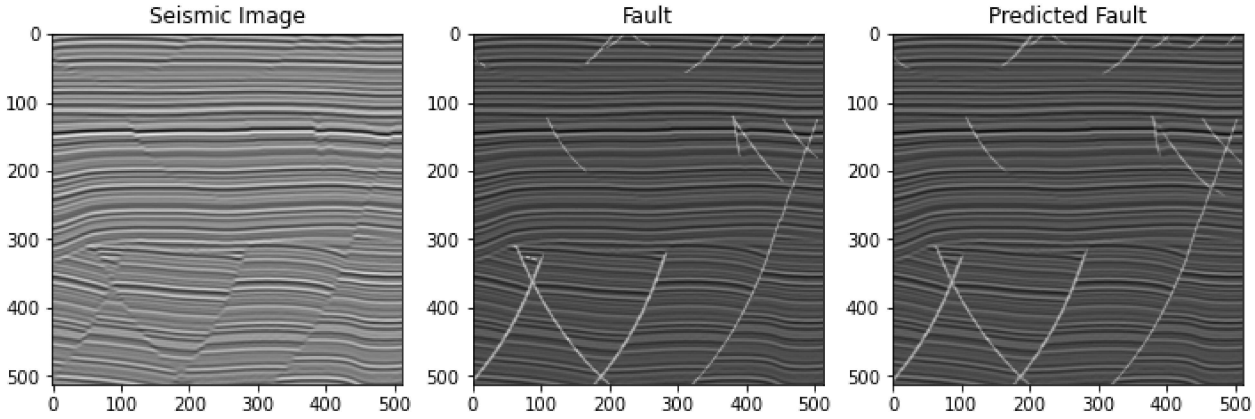

```
Epoch 12: Step 1313: U-Net loss: 0.011931095272302628
```



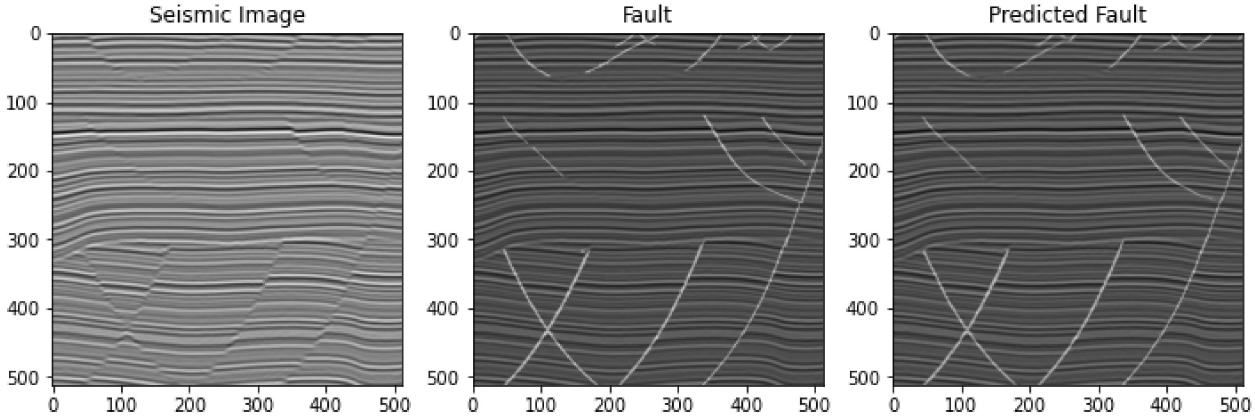
```
Epoch 13: Step 1414: U-Net loss: 0.010801272466778755
```



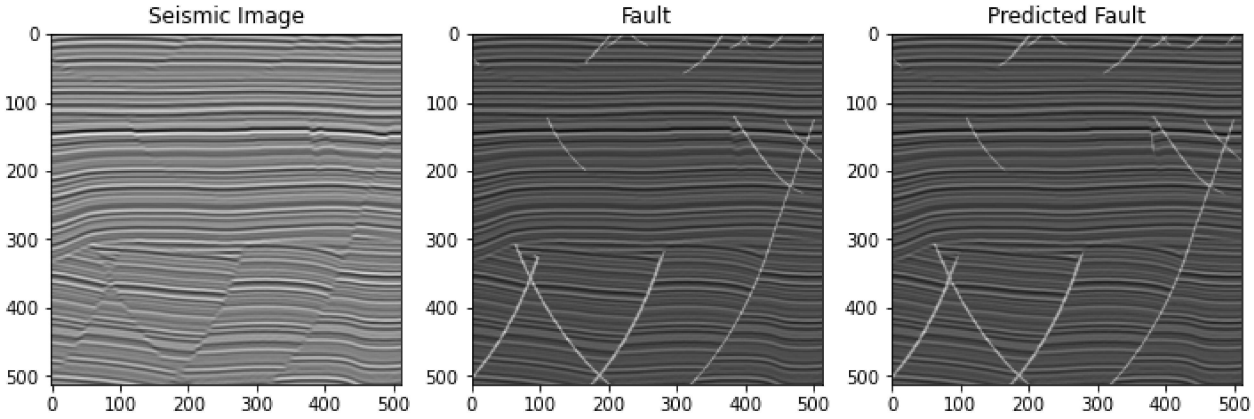

```
Epoch 14: Step 1515: U-Net loss: 0.009889635257422924
```



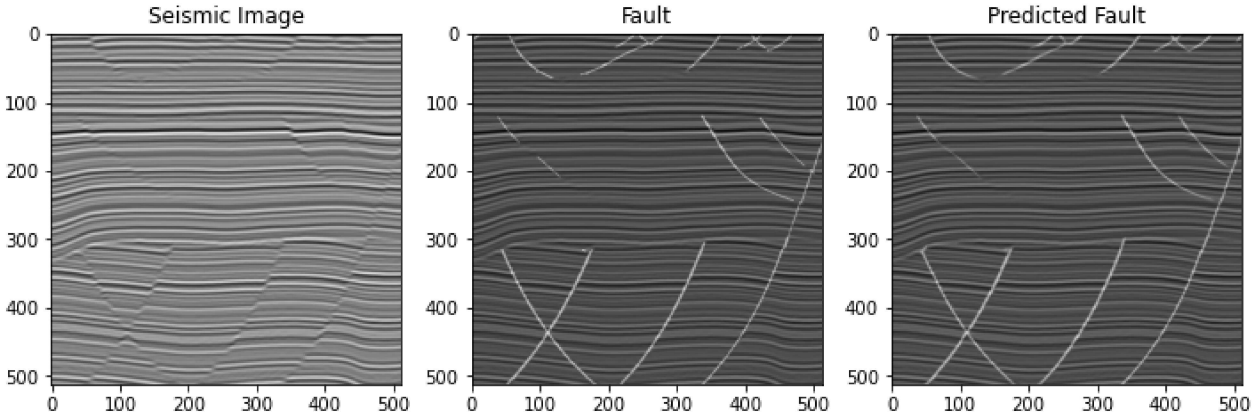

```
Epoch 15: Step 1616: U-Net loss: 0.011190870776772499
```



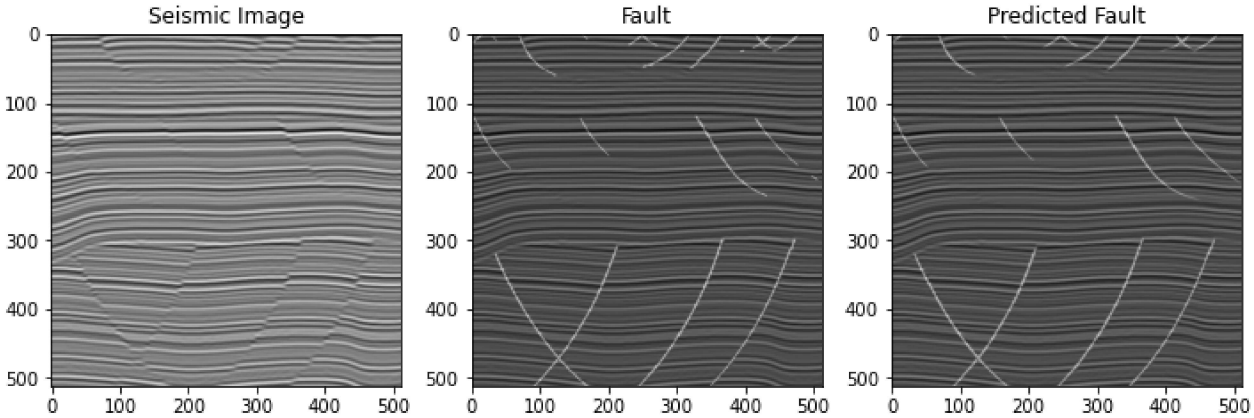

```
Epoch 16: Step 1717: U-Net loss: 0.008254233747720718
```



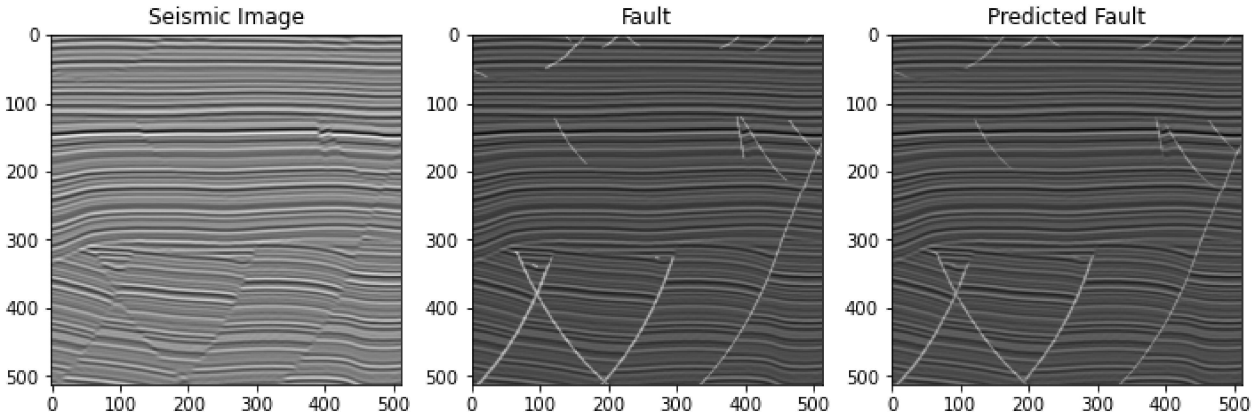

```
Epoch 17: Step 1818: U-Net loss: 0.013269666582345963
```




```
Epoch 18: Step 1919: U-Net loss: 0.009330361150205135
```

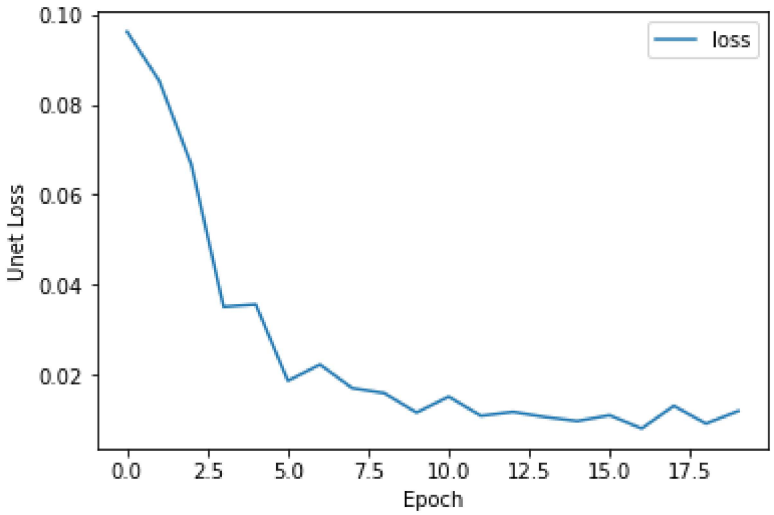



```
Epoch 19: Step 2020: U-Net loss: 0.012113308534026146
```



In [49]:
```
loss_list_numpy=[]
for i in loss:
    loss_list_numpy.append(i.detach().numpy())
```

In [50]:
```
plt.plot(loss_list_numpy, label = 'loss')
plt.xlabel('Epoch')
plt.ylabel('Unet Loss')
plt.legend()
plt.show()
```




In [ ]:

In [ ]: