```
#importing libraries
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)
```

```
<torch._C.Generator at 0x1e2589649d0>
```

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.

```
class ContractingBlock(nn.Module):
    def __init__(self, input_channels):
        """
        This class represents a contracting block of a U-Net model. It
doubles the number of
        channels in the first convolution layer and keeps the same
number of channels in the
        second convolution layer, followed by a max pooling operation.

        Args:
        - input_channels (int): number of input channels

        """
        super(ContractingBlock, self).__init__()

        self.conv1 = nn.Conv2d(input_channels, 2*input_channels,
kernel_size=3, padding=(1,1))
        self.conv2 = nn.Conv2d(2*input_channels, 2*input_channels,
kernel_size=3, padding=(1,1))
        self.activation = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(kernel_size=2, stride=2)


    def forward(self, x):
```

```
        """
        Defines the forward computation of the contracting block.

        Args:
        - x (tensor): input tensor of shape (batch_size,
input_channels, height, width)

        Returns:
        - tensor: output tensor of shape (batch_size,
2*input_channels, height/2, width/2)

        """
        x = self.conv1(x)
        x = self.activation(x)
        x = self.conv2(x)
        x = self.activation(x)
        x = self.maxpool(x)
        return x

# 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)

    print(test_out_conv1.shape)

test_contracting_block(128)
# 1, 512, 60, 60
```

```
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)

```
class ExpandingBlock(nn.Module):

    def __init__(self, input_channels):
        """
        This class represents an expanding block of a U-Net model. It
consists of an upsampling
        of the feature map, followed by a concatenation with the
corresponding feature map from
        the contracting path. Then, it performs two convolution
operations with a ReLU activation
        after each convolution.

        Args:
        - input_channels (int): number of input channels

        """
        super(ExpandingBlock, self).__init__()

        self.upsample = nn.Upsample(scale_factor=2, mode='bilinear',
align_corners=True)
        self.conv1 = nn.Conv2d(input_channels, input_channels//2,
kernel_size=3, padding=(1,1))
        self.conv2 = nn.Conv2d(input_channels, input_channels//2,
kernel_size=3, padding=(1,1))
        self.conv3 = nn.Conv2d(input_channels//2, input_channels//2,
kernel_size=3, padding=(1,1))

        self.activation = nn.ReLU()

    def forward(self, x, skip_con_x):
        """
        Defines the forward computation of the expanding block.

        Args:
        - x (tensor): input tensor of shape (batch_size,
input_channels, height, width)
        - skip_con_x (tensor): tensor from the corresponding
contracting block with the same shape
                               as x

        Returns:
        - tensor: output tensor of shape (batch_size,
input_channels/2, height*2, width*2)

        """
        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

#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_size * 2 + 6)
    x = torch.randn(test_samples, test_channels, test_size, test_size)
    x = test_block.upsample(x)

    print(x.shape)

test_expanding_block()
```

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

Feature Block 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

```
class FeatureMapBlock(nn.Module):

    def __init__(self, input_channels, output_channels):
        """
        This class represents the final feature map block of a U-Net
model. It consists of a
        1x1 convolutional layer that maps the input tensor to the
final output tensor with the
        desired number of channels.

        Args:
        - input_channels (int): number of input channels
        - output_channels (int): number of output channels

        """
        super(FeatureMapBlock, self).__init__()

        self.conv = nn.Conv2d(input_channels, output_channels,
kernel_size=1)


    def forward(self, x):
        """
        Defines the forward computation of the final feature map
```

```
block.

        Args:
        - x (tensor): input tensor of shape (batch_size,
input_channels, height, width)

        Returns:
        - tensor: output tensor of shape (batch_size, output_channels,
height, width)

        """
        x = self.conv(x)
        return x
```

U-Net

The UNet model is a popular architecture used for image segmentation tasks, which involves dividing an image into multiple segments, each representing a different object or feature. The architecture consists of an encoder and decoder network, which are connected through a bottleneck layer. The encoder network captures the low-level features of the input image and progressively reduces its spatial resolution, while the decoder network upsamples the feature maps and generates segmentation masks with the same resolution as the input image. The bottleneck layer connects the encoder and decoder networks, and preserves the spatial information needed for accurate segmentation.

In the context of fault prediction in 2D seismic images, the UNet model is trained to identify fault structures within the images. The input to the model is a 2D seismic image, and the output is a binary segmentation mask, where each pixel is classified as either a fault or non-fault. The model is trained using a loss function that measures the difference between the predicted segmentation mask and the ground truth mask, which is manually annotated by experts.

The implementation in PyTorch involves defining the UNet model architecture using PyTorch's nn.Module class, which allows for easy customization and modification of the model. The model is trained using PyTorch's built-in optimization and backpropagation functions, such as the Adam optimizer and the Cross-Entropy loss function. The training data is typically preprocessed using techniques such as data augmentation and normalization to improve the model's performance and generalization ability.

Overall, the UNet model implemented in PyTorch for fault prediction in 2D seismic images is a powerful tool for identifying and characterizing fault structures in subsurface geological formations.

```
class UNet(nn.Module):
    """
    This is the main UNet model class. It inherits from the nn.Module
PyTorch class.
    """
```

```
    def __init__(self, input_channels, output_channels,
hidden_channels=64):
        """
        The class constructor. Initializes all the layers that make up
the UNet model.

        Args:
            input_channels (int): The number of channels in the input
image.
            output_channels (int): The number of channels in the
output segmentation mask.
            hidden_channels (int): The number of channels in the
hidden layers. Default is 64.
        """
        super(UNet, self).__init__()

        # Define the layers of the contracting path
        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)

        # Define the layers of the expanding path
        self.expand1 = ExpandingBlock(hidden_channels * 16)
        self.expand2 = ExpandingBlock(hidden_channels * 8)
        self.expand3 = ExpandingBlock(hidden_channels * 4)
        self.expand4 = ExpandingBlock(hidden_channels * 2)

        # Define the layer that converts the final feature map to the
desired output format
        self.downfeature = FeatureMapBlock(hidden_channels,
output_channels)


    def forward(self, x):
        """
        The forward pass of the UNet model. It takes the input image
as input and returns the segmentation mask.

        Args:
            x (torch.Tensor): The input image tensor.

        Returns:
            torch.Tensor: The output segmentation mask tensor.
        """
        # Pass the input through the contracting path
        x0 = self.upfeature(x)
        x1 = self.contract1(x0)
```

```
        x2 = self.contract2(x1)
        x3 = self.contract3(x2)
        x4 = self.contract4(x3)

        # Pass the output of the contracting path through the
expanding path
        x5 = self.expand1(x4, x3)
        x6 = self.expand2(x5, x2)
        x7 = self.expand3(x6, x1)
        x8 = self.expand4(x7, x0)

        # Pass the final feature map through the output layer to
obtain the segmentation mask
        xn = self.downfeature(x8)
        return xn
```

The following code defines and instantiates a UNet model for image segmentation tasks and moves it to a specified device for computation, making it ready for training or inference on a dataset.

```
# Defining Model
input_dim = 1
label_dim = 1

# Create an instance of the UNet model with the specified input and
output dimensions
model = UNet(input_dim, label_dim)

# Move the model to the specified device (e.g. CPU or GPU) for
computation
model = model.to(device)

#Loading saved weights of pretrained model on synthtic seismic images
my_model = model.load_state_dict(torch.load('save',
map_location=torch.device('cuda')))

#Printing model parameters
print(model.parameters)
```

```
<bound method Module.parameters of UNet(
  (upfeature): FeatureMapBlock(
    (conv): Conv2d(1, 64, kernel_size=(1, 1), stride=(1, 1))
  )
  (contract1): ContractingBlock(
    (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
    (activation): ReLU(inplace=True)
    (maxpool): MaxPool2d(kernel_size=2, stride=2, padding=0,
dilation=1, ceil_mode=False)
```

```
  )
  (contract2): ContractingBlock(
    (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
    (activation): ReLU(inplace=True)
    (maxpool): MaxPool2d(kernel_size=2, stride=2, padding=0,
dilation=1, ceil_mode=False)
  )
  (contract3): ContractingBlock(
    (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
    (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
    (activation): ReLU(inplace=True)
    (maxpool): MaxPool2d(kernel_size=2, stride=2, padding=0,
dilation=1, ceil_mode=False)
  )
  (contract4): ContractingBlock(
    (conv1): Conv2d(512, 1024, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
    (conv2): Conv2d(1024, 1024, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
    (activation): ReLU(inplace=True)
    (maxpool): MaxPool2d(kernel_size=2, stride=2, padding=0,
dilation=1, ceil_mode=False)
  )
  (expand1): ExpandingBlock(
    (upsample): Upsample(scale_factor=2.0, mode=bilinear)
    (conv1): Conv2d(1024, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
    (conv2): Conv2d(1024, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
    (conv3): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
    (activation): ReLU()
  )
  (expand2): ExpandingBlock(
    (upsample): Upsample(scale_factor=2.0, mode=bilinear)
    (conv1): Conv2d(512, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
    (conv2): Conv2d(512, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
    (conv3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
    (activation): ReLU()
  )
  (expand3): ExpandingBlock(
    (upsample): Upsample(scale_factor=2.0, mode=bilinear)
```

```
    (conv1): Conv2d(256, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
    (conv2): Conv2d(256, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
    (conv3): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
    (activation): ReLU()
  )
  (expand4): ExpandingBlock(
    (upsample): Upsample(scale_factor=2.0, mode=bilinear)
    (conv1): Conv2d(128, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
    (conv2): Conv2d(128, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
    (conv3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
    (activation): ReLU()
  )
  (downfeature): FeatureMapBlock(
    (conv): Conv2d(64, 1, kernel_size=(1, 1), stride=(1, 1))
  )
)>
```

In PyTorch, model parameters are set to require gradients by default, which means that they are included in the computation of gradients during backpropagation and can be updated during optimization. However, it is possible to freeze some or all of the parameters in the model by setting their requires_grad attribute to False. This is often done when using pre-trained models, where the lower layers of the model are frozen to preserve the pre-trained weights and avoid overfitting.

```
#Unfreezing all parameters of model
for param in model.parameters():
    param.requires_grad = True

#Verifying all parameters are unfreezed and are capable to update
weights during learning
for child in model.children():
    print(child)
    for param in child.parameters():
        print(param.requires_grad)
```

```
FeatureMapBlock(
  (conv): Conv2d(1, 64, kernel_size=(1, 1), stride=(1, 1))
)
True
True
ContractingBlock(
  (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
  (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
```

```
padding=(1, 1))
  (activation): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
)
True
True
True
True
ContractingBlock(
  (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
  (activation): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
)
True
True
True
True
ContractingBlock(
  (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
  (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
  (activation): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
)
True
True
True
True
ContractingBlock(
  (conv1): Conv2d(512, 1024, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
  (conv2): Conv2d(1024, 1024, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
  (activation): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
)
True
True
True
True
ExpandingBlock(
  (upsample): Upsample(scale_factor=2.0, mode=bilinear)
```

```
  (conv1): Conv2d(1024, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
  (conv2): Conv2d(1024, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
  (conv3): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
  (activation): ReLU()
)
True
True
True
True
True
True
ExpandingBlock(
  (upsample): Upsample(scale_factor=2.0, mode=bilinear)
  (conv1): Conv2d(512, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
  (conv2): Conv2d(512, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
  (conv3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
  (activation): ReLU()
)
True
True
True
True
True
True
ExpandingBlock(
  (upsample): Upsample(scale_factor=2.0, mode=bilinear)
  (conv1): Conv2d(256, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
  (conv2): Conv2d(256, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
  (conv3): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
  (activation): ReLU()
)
True
True
True
True
True
True
ExpandingBlock(
  (upsample): Upsample(scale_factor=2.0, mode=bilinear)
  (conv1): Conv2d(128, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
```

```
  (conv2): Conv2d(128, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
  (conv3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
  (activation): ReLU()
)
True
True
True
True
True
True
FeatureMapBlock(
  (conv): Conv2d(64, 1, kernel_size=(1, 1), stride=(1, 1))
)
True
True
```

The following code snippet in Python using NumPy and os modules loads seismic images stored as .npy files from a directory and stacks them into a three-dimensional NumPy array for further processing.

First, the path to the folder containing the seismic .npy files is defined using the folder_path variable. Then, a list of all the .npy files in the folder is obtained using the os.listdir() function and filtered to only include files ending with .npy.

Next, a NumPy array seismic is created by loading each .npy file using the np.load() function and stacking them along the first dimension using the np.stack() function. This creates a three-dimensional array with the first dimension representing each seismic image, and the other two dimensions representing the height and width of each image.

Finally, the resulting seismic array is cast to a NumPy float32 type using np.float32() function and the shape of the resulting array is printed to confirm that it has the correct shape for further processing.

Overall, this code snippet provides a way to load and preprocess seismic images stored as .npy files in a specified directory, which can be useful for training or inference of deep learning models.

```
#Taking seismic images from directory and stacking them into an input
array
import numpy as np
import os

# Define the path to the folder containing the .npy files
folder_path =
"C:/Users/gpuuser2/Desktop/Export_5/fault_train2/new_seismic"

# Get a list of all the .npy files in the folder
file_list = os.listdir(folder_path)
```

```
file_list = [f for f in file_list if f.endswith('.npy')]

# Load each .npy file and stack them into a three-dimensional numpy
array
seismic = np.stack([np.load(os.path.join(folder_path, f)) for f in
file_list], axis=0)
seismic=np.float32(seismic)

# Print the shape of the resulting numpy array
print("Shape of seismic array:", seismic.shape)
```

```
Shape of seismic array: (85, 2001, 1501)
```

```
#Taking fault images from directory and stacking them into actual
response array

# Define the path to the folder containing the .npy files
folder_path =
"C:/Users/gpuuser2/Desktop/Export_5/fault_train2/new_fault"

# Get a list of all the .npy files in the folder
file_list = os.listdir(folder_path)
file_list = [f for f in file_list if f.endswith('.npy')]

# Load each .npy file and stack them into a three-dimensional numpy
array
fault = np.stack([np.load(os.path.join(folder_path, f)) for f in
file_list], axis=0)
fault=np.float32(fault)

# Print the shape of the resulting numpy array
print("Shape of fault array:", fault.shape)
```

```
Shape of fault array: (85, 2001, 1501)
```

In the following code, The first function crop_input takes an input image and crops it to a specified new shape. The function takes two arguments: image which is a NumPy array representing the input image, and new_shape which is a tuple representing the desired new shape of the image in the format (height, width).

The function first calculates the original height and width of the input image using its shape attribute. It then calculates the new height and width by extracting them from the new_shape argument. The function then calculates the starting indices for the crop by subtracting the new shape from the original shape and adding 1, and dividing by 2. This ensures that the cropped image is centered on the original image.

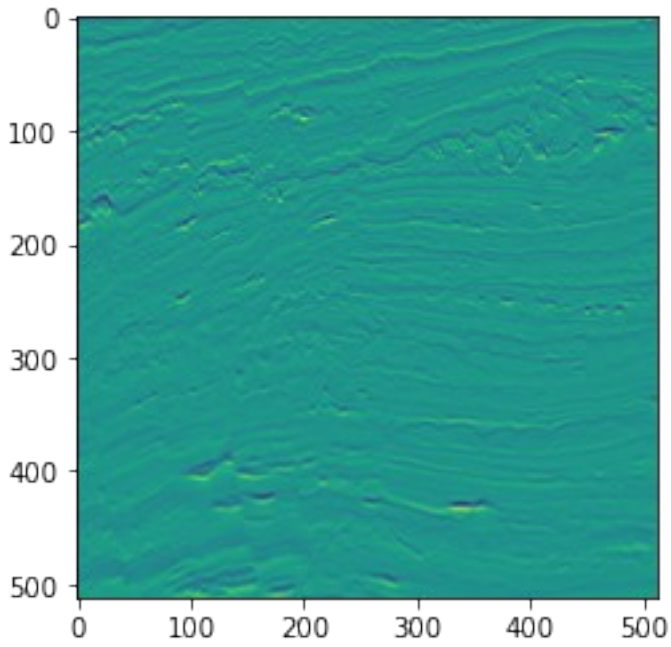
Finally, the function crops the input image to the new shape using NumPy indexing and returns the cropped image.

The second function pad_to takes an input image and pads it with zeros to a specified new shape. The function takes the same two arguments as crop_input. It calculates the amount of padding required on each side by subtracting the original height and width from the new shape. It then uses the F.pad() function from PyTorch to pad the input image with zeros on each side to achieve the desired new shape.

Overall, these two functions provide a way to preprocess input images for deep learning models by cropping or padding them to a specified shape.

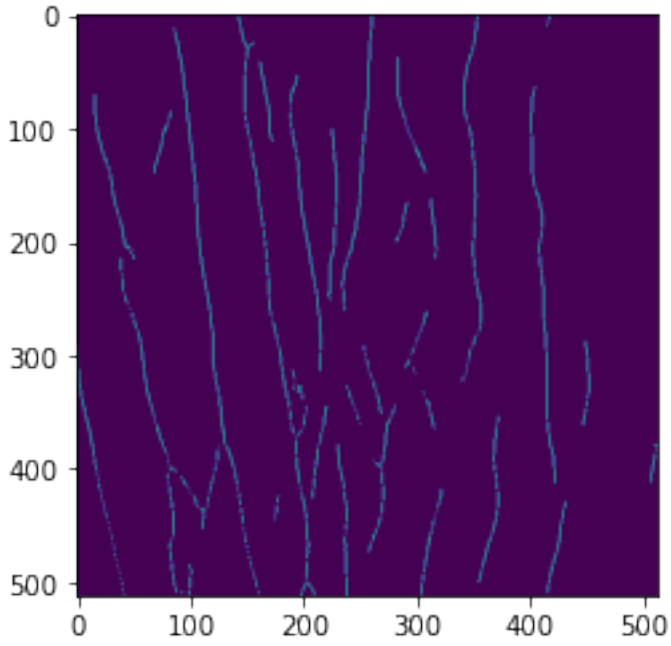
```
# Function to crop an input image to a specified new shape
def crop_input(image, new_shape):
    """
    Args:
        image (numpy array): Input image to be cropped
        new_shape (tuple): Desired shape of cropped image in format
(height, width)
    Returns:
        numpy array: Cropped image
    """
    # Get the original height and width of the image
    h, w = image.shape[0], image.shape[1]

    # Get the new height and width of the image
    new_h, new_w = new_shape[0], new_shape[1]

    # Calculate the starting indices for the crop
    start_h = int((h - new_h + 1)/2)
    start_w = int((w - new_w + 1)/2)

    # Crop the image to the new shape
    cropped_image = image[start_h:start_h + new_h, start_w:start_w +
new_w]

    return cropped_image

# Function to pad an input image to a specified new shape
import torch.nn.functional as F

def pad_to(image, new_shape):
    """
    Args:
        image (numpy array): Input image to be padded
        new_shape (tuple): Desired shape of padded image in format
(height, width)
    Returns:
        numpy array: Padded image
    """
    # Get the original height and width of the image
    h, w = image.shape[0], image.shape[1]
```

```
    # Get the new height and width of the image
    new_h, new_w = new_shape[0], new_shape[1]

    # Calculate the amount of padding required on each side
    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

    # Pad the image with zeros to the new shape
    padded_image = F.pad(image, pads, "constant", 0)

    return padded_image

new_shape = (512, 512)
image_list = []
fault_list = []
for i in range(85):
    images = torch.from_numpy(seismic[i])
    image_list.append(crop_input(images, new_shape).unsqueeze(0))

    # Define fault before running this code
    faults = torch.from_numpy(fault[i])
    fault_list.append(crop_input(faults, new_shape).unsqueeze(0))

# Stack the tensors
images_tensor = torch.cat(image_list, dim=0)
faults_tensor = torch.cat(fault_list, dim=0)

#converting inputs to tensors
new_shape = (512, 512)
image_list = []
fault_list = []
for i in range(85):
    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))

#Plotting sample seismic image
plt.imshow(np.squeeze(image_list[0]))

<matplotlib.image.AxesImage at 0x1e2824dda00>
```




```
#Plotting sample fault image
plt.imshow(np.squeeze(fault_list[0]))
```

```
<matplotlib.image.AxesImage at 0x1e27f618c70>
```




```
# create an empty list to store each cropped input image
image_list = []
# create an empty list to store each cropped label image
fault_list = []
# set the desired output shape for each image (height and width in
```

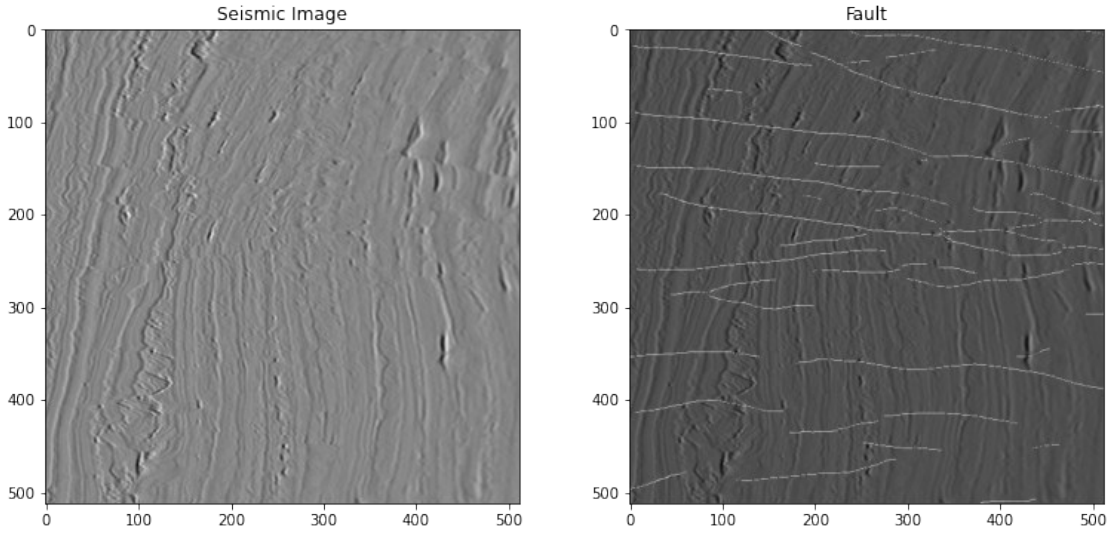
```
pixels)
new_shape = (512, 512)
# loop through each image in the seismic array
for i in range(85):
    # convert the seismic image to a PyTorch tensor
    images = torch.from_numpy(seismic[i])
    # crop the image to the desired output shape and add an extra
dimension
    # so that it has shape (1, new_shape[0], new_shape[1])
    image_list.append(crop_input(images, new_shape).unsqueeze(0))

    # convert the corresponding fault label image to a PyTorch tensor
    faults = torch.from_numpy(fault[i])
    # crop the label to the desired output shape and add an extra
dimension
    # so that it has shape (1, new_shape[0], new_shape[1])
    fault_list.append(crop_input(faults, new_shape).unsqueeze(0))

# stack all of the cropped input images into a single tensor
volumes = torch.stack(image_list)
# stack all of the cropped label images into a single tensor
labels = torch.stack(fault_list)

# create a PyTorch TensorDataset object that combines the volumes
tensor and labels tensor
dataset = torch.utils.data.TensorDataset(volumes, labels)
```

```
volumes.shape, labels.shape
```

```
(torch.Size([48, 1, 512, 512]), torch.Size([48, 1, 512, 512]))
```

```
# Plotting Fault over seismic image
fig = plt.figure(figsize=(20,20))  # creating a new figure with a size
of 20x20 inches

# Plot the first seismic image from the dataset
ax = fig.add_subplot(331)
plt.imshow(volumes[0].T, cmap="gray")  # show the seismic image,
transposed to match the typical image orientation
ax.set_title("Seismic Image")  # set a title for the subplot

# Plot the corresponding fault map for the first seismic image
ax = fig.add_subplot(332)
ax.imshow(volumes[0].T, cmap='gray')  # show the seismic image as the
background
ax.imshow(labels[0].T, cmap='gray', vmin=0, vmax=1, alpha=0.4)  #
overlay the fault map, setting the colormap, intensity range, and
transparency
ax.set_title("Fault")  # set a title for the subplot
```

```
plt.show()  # show the final figure
```




```
import matplotlib.pyplot as plt

def show_tensor_images(image, fault, pred, num_images=25, size=(1, 28,
28)):
    """
    Displays seismic, fault, and predicted fault images side by side
for visualization.

    Args:
    image (torch.Tensor): A tensor of seismic images.
    fault (torch.Tensor): A tensor of fault labels.
    pred (torch.Tensor): A tensor of predicted fault labels.
    num_images (int): The number of images to display (default is 25).
    size (tuple): The size of the images (default is 1x28x28).

    Returns:
    None
    """

    # Convert tensors to numpy arrays and remove extra dimensions
    image_unflat = image.detach().cpu().numpy().squeeze()
    fault_unflat = fault.detach().cpu().numpy().squeeze()
    pred_unflat = pred.detach().cpu().numpy().squeeze()

    # Create a figure with 3 subplots
    fig = plt.figure(figsize=(12,15))

    # Display seismic image in subplot 1
    ax = fig.add_subplot(331)
    ax.imshow(image_unflat, cmap = 'gray')
```

```
    ax.set_title("Seismic Image")

    # Display fault label in subplot 2
    ax = fig.add_subplot(332)
    ax.imshow(fault_unflat, cmap = 'gray', vmin=0, vmax=1, alpha=0.4)
    ax.set_title("Fault")

    # Display predicted fault label in subplot 3
    ax = fig.add_subplot(333)
    ax.imshow(pred_unflat, cmap = 'gray', vmin=0, vmax=1, alpha=0.4)
    ax.set_title("Predicted Fault")

    # Display the figure
    plt.show()

device = torch.device("cuda" if (torch.cuda.is_available()) else
"cpu")
print(device)
```

```
cuda
```

```
# Hyperparameters
criterion = nn.BCEWithLogitsLoss()
n_epochs = 85
input_dim = 1
label_dim = 1
display_step = 500
batch_size = 1
lr = 0.0002
initial_shape = 512
target_shape = 512
device = 'cuda'
```

The train() function in the code above is a function that performs training or fine-tuning of a pretrained model. It first loads data into a data loader with specified batch size and shuffling, initializes a UNet model with specified input and output dimensions and moves it to GPU if available. Then, it initializes an optimizer for UNet with specified learning rate and initializes variables for keeping track of training progress.

In the training loop, for each epoch, the function loops through each batch in the data loader and updates the UNet. For each batch, it calculates the loss between the predicted output and the ground truth labels, backpropagates the loss and updates the UNet parameters. It also calculates the accuracy and keeps track of the training losses and accuracies.

At the end of each epoch, it calculates the average training accuracy for the current epoch and prints it. Finally, it returns the final trained model, predicted outputs, and training loss and accuracy lists.

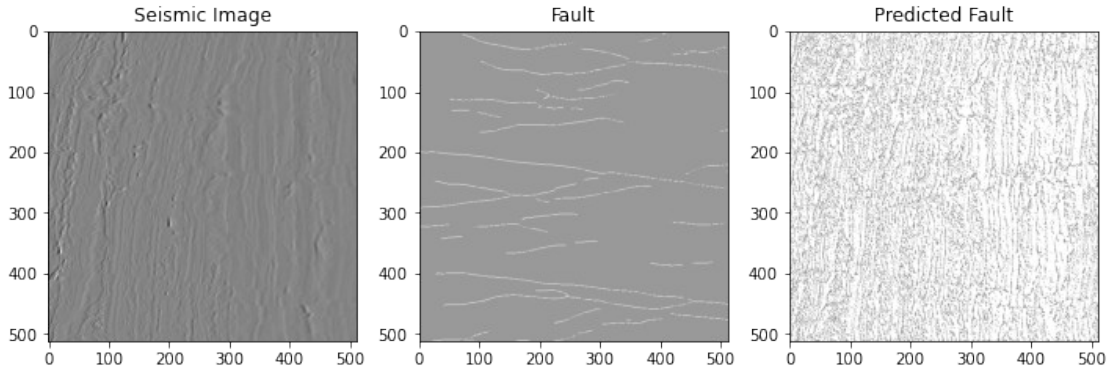
```
import statistics
#Function performing training or fine-tuning of Pretrained model
```

```
def train():
    # Load data into data loader with specified batch size and
shuffling
    dataloader = DataLoader(
        dataset,
        batch_size=batch_size,
        shuffle=True)

    # Initialize UNet model with specified input and output dimensions
and move to GPU if available
    unet = UNet(input_dim, label_dim).to(device)

    # Initialize optimizer for UNet with specified learning rate
    unet_opt = torch.optim.Adam(unet.parameters(), lr=lr)

    # Initialize variables for keeping track of training progress
    cur_step = 0
    s=0
    train_losses = []
    train_accs = []

    # Loop through each epoch
    for epoch in range(n_epochs):
        # Loop through each batch in the data loader
        for real, labels in tqdm(dataloader):
            # Get the current batch size
            cur_batch_size = len(real)

            # Move the real and labels tensors to the GPU if available
            real = real.to(device)
            labels = labels.to(device)

            ### Update U-Net ###

            # Reset gradients for the UNet optimizer
            unet_opt.zero_grad()

            # Get the predicted output of the UNet on the current
batch
            pred = unet(real)

            # Calculate the loss between the predicted output and the
ground truth labels
            unet_loss = criterion(pred, labels)
            train_losses.append(unet_loss.item())

            # Backpropagate the loss and update the UNet parameters
            unet_loss.backward()
            unet_opt.step()
```

```
            # Calculate accuracy
            binary_pred = (torch.sigmoid(pred) > 0.5).float()
            correct = (binary_pred == labels).sum().item()
            accuracy = (correct / (cur_batch_size * label_dim *
input_dim * input_dim))*(100)*(1/262144)
            train_accs.append(accuracy)

            # If the current step is a display step, print the loss
and accuracy and show the predicted and ground truth images
            if cur_step % display_step == 0:
                print(f"Epoch {epoch}: Step {cur_step}: U-Net loss:
{unet_loss.item()}, Train Accuracy: {accuracy}")

                show_tensor_images(real.T, labels.T, binary_pred.T,
size=(input_dim, target_shape, target_shape))
            cur_step += 1

        # Calculate the average training accuracy for the current
epoch
        avg_train_acc = statistics.mean(train_accs)
        print(f"Epoch {epoch}: Average Train Accuracy:
{avg_train_acc}")

    # Return the final trained model, predicted outputs, and training
loss and accuracy lists
    return unet, pred, train_losses, train_accs

# Call the train function and save the returned values to variables
model, pred, loss, acc = train()
```

{"model_id":"c9538781db644e58a716a7ac1a204a81","version_major":2,"version_minor":0}

```
Epoch 0: Step 0: U-Net loss: 54.097686767578125, Train Accuracy:
23.785400390625
```




```
Epoch 0: Average Train Accuracy: 96.2450812844669
```

{"model_id":"927f76618ab049238cbaa9b797e6bf1d","version_major":2,"version_minor":0}

Epoch 1: Average Train Accuracy: 96.88247905058019

{"model_id":"0d7b07a4e863489abbd717144739d58b","version_major":2,"version_minor":0}

Epoch 2: Average Train Accuracy: 97.36582138959099

{"model_id":"89444d7068a645bfabc0e4b208997d07","version_major":2,"version_minor":0}

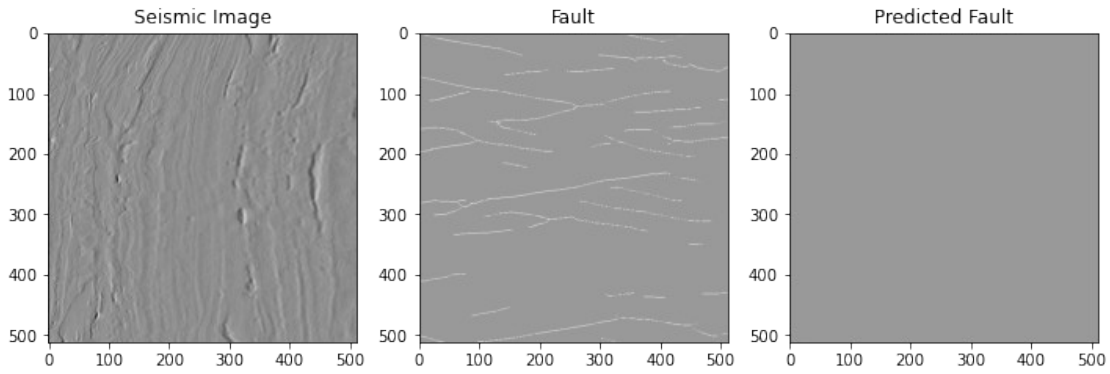
Epoch 3: Average Train Accuracy: 97.64395096722771

{"model_id":"cfbd0311821d455da11ea71ae1caa441","version_major":2,"version_minor":0}

Epoch 4: Average Train Accuracy: 97.81288506002987

{"model_id":"df91fcf34ec14448bf8a8abcde163c61","version_major":2,"version_minor":0}

Epoch 5: Step 500: U-Net loss: 0.07734248042106628, Train Accuracy: 98.69270324707031



Epoch 5: Average Train Accuracy: 97.92884901458142

{"model_id":"88b53b8cdb5f422ab6b32f6c3c27b460","version_major":2,"version_minor":0}

Epoch 6: Average Train Accuracy: 98.01223049644662

{"model_id":"95a56c2a39994b10bc578beaab89a265","version_major":2,"version_minor":0}

Epoch 7: Average Train Accuracy: 98.07485131656422

{"model_id":"eab46a6c38ea49b9be832b6fb0ea083c","version_major":2,"version_minor":0}

Epoch 8: Average Train Accuracy: 98.12379475512536

{"model_id":"0ab5e6e77a164ac98020c0919953896a","version_major":2,"version_minor":0}

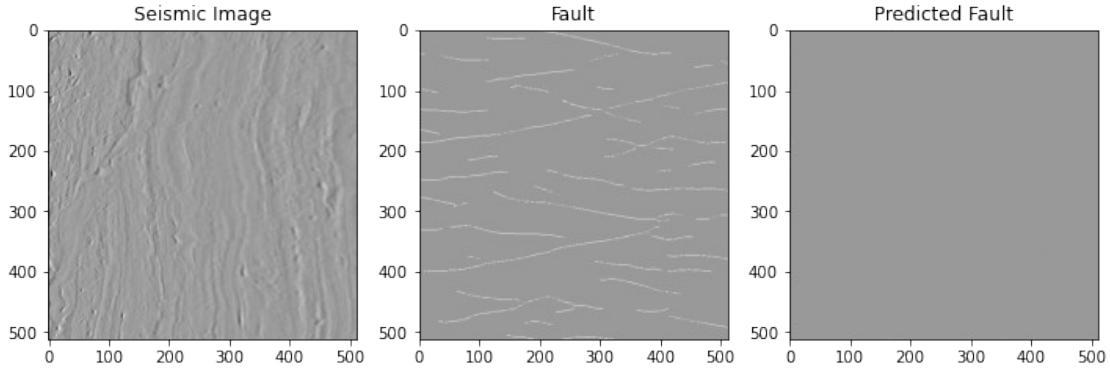
Epoch 9: Average Train Accuracy: 98.16300201416016

{"model_id":"517a90d478fb43a5853f2eaa4a649c5f","version_major":2,"version_minor":0}

Epoch 10: Average Train Accuracy: 98.19508272058823

{"model_id":"030686d615eb40d1b749159bcc7a13a1","version_major":2,"version_minor":0}

Epoch 11: Step 1000: U-Net loss: 0.061342526227235794, Train Accuracy: 98.53324890136719



Epoch 11: Average Train Accuracy: 98.22184618781594

{"model_id":"a2084bfbc11d40ed9c5f7edb3fd41699","version_major":2,"version_minor":0}

Epoch 12: Average Train Accuracy: 98.24447459225202

{"model_id":"9e7bf6c76da84d3c9abf1071cb597cfe","version_major":2,"version_minor":0}

Epoch 13: Average Train Accuracy: 98.26380657548664

{"model_id":"43ced715ccf64496a8dc57f83bda8880","version_major":2,"version_minor":0}

Epoch 14: Average Train Accuracy: 98.2806558048024

{"model_id":"b1cc7d6b75cf436d8132a572f7e63e15","version_major":2,"version_minor":0}

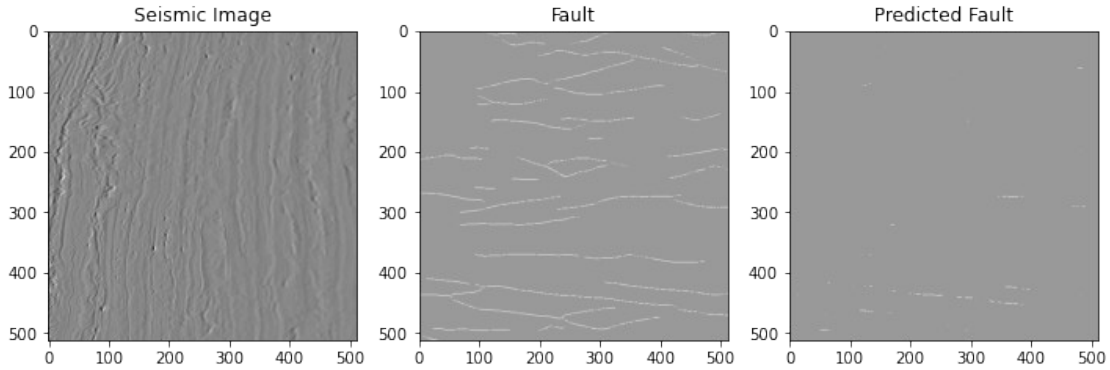
Epoch 15: Average Train Accuracy: 98.29526031718535

{"model_id":"9a723e5ac9b540fd837de8e856de982f","version_major":2,"version_minor":0}

Epoch 16: Average Train Accuracy: 98.30813477196082

{"model_id":"5e0410f6178241d5a2e691c0cc34a663","version_major":2,"version_minor":0}

Epoch 17: Step 1500: U-Net loss: 0.05252717062830925, Train Accuracy: 98.49205017089844



Epoch 17: Average Train Accuracy: 98.3195779837814

{"model_id":"402cce73423741edbf298d03d119bfd7","version_major":2,"version_minor":0}

Epoch 18: Average Train Accuracy: 98.32980719882268

{"model_id":"adf33cefad4e43aa85d248edd5cd8582","version_major":2,"version_minor":0}

Epoch 19: Average Train Accuracy: 98.33891745174633

{"model_id":"6765e97456ad48118f39be32569ebde6","version_major":2,"version_minor":0}

Epoch 20: Average Train Accuracy: 98.34713356167663

{"model_id":"e823037333324eb3aa8d79b774218ac9","version_major":2,"version_minor":0}

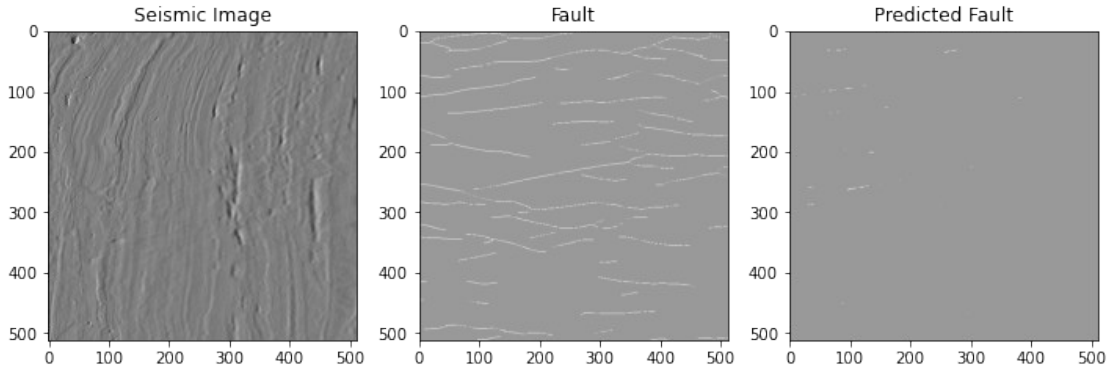
Epoch 21: Average Train Accuracy: 98.35491751604539

{"model_id":"880703e221444b489a2855a99849ed9e","version_major":2,"version_minor":0}

Epoch 22: Average Train Accuracy: 98.36219202222117

{"model_id":"38e1a22d728c4159905084ad039f653b","version_major":2,"version_minor":0}

Epoch 23: Step 2000: U-Net loss: 0.05265195295214653, Train Accuracy: 98.32191467285156



Epoch 23: Average Train Accuracy: 98.36916456035539

{"model_id":"06a4dca89c4246efaf58178d8a5da4d8","version_major":2,"version_minor":0}

Epoch 24: Average Train Accuracy: 98.37617761948529

{"model_id":"37c0634927254ba3a0da8c5c72f21cfd","version_major":2,"version_minor":0}

Epoch 25: Average Train Accuracy: 98.38238245761232

{"model_id":"9b36b32088204e53907ef4d747e76ffb","version_major":2,"version_minor":0}

Epoch 26: Average Train Accuracy: 98.3887877287688

{"model_id":"f7e706d8eed741c7ad41ba55b43bb2d9","version_major":2,"version_minor":0}

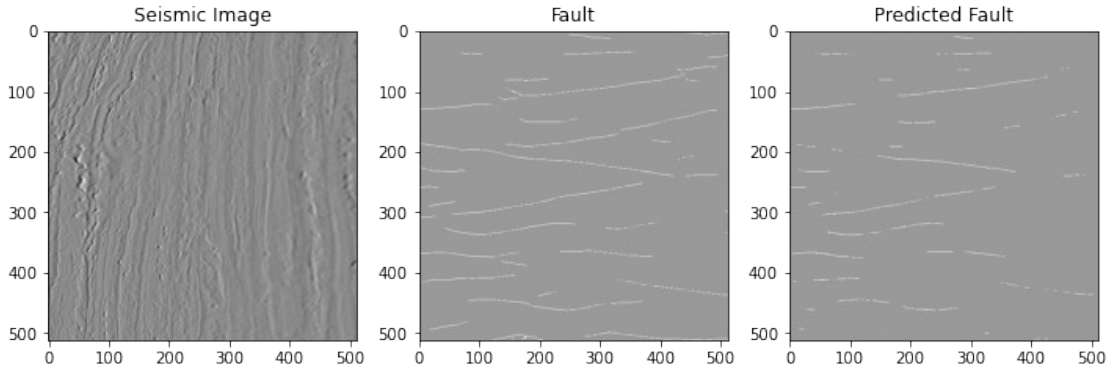
Epoch 27: Average Train Accuracy: 98.3930853835675

{"model_id":"3068ab1f376a485c9271bcc3773d9635","version_major":2,"version_minor":0}

Epoch 28: Average Train Accuracy: 98.39701252099708

{"model_id":"73da11962eff44748259bdc47fae11ef","version_major":2,"version_minor":0}

Epoch 29: Step 2500: U-Net loss: 0.03454490751028061, Train Accuracy: 98.60992431640625



Epoch 29: Average Train Accuracy: 98.40145051245595

{"model_id":"a4a386d6068c45bc9a00fb15f6249780","version_major":2,"version_minor":0}

Epoch 30: Average Train Accuracy: 98.40566298540901

{"model_id":"73d4ac4e90a34d51aaf95cc1e00a9bec","version_major":2,"version_minor":0}

Epoch 31: Average Train Accuracy: 98.40972395504222

{"model_id":"1fb5ff9114e14d4e9a2d70a715fbb3e5","version_major":2,"version_minor":0}

Epoch 32: Average Train Accuracy: 98.41474509281696

{"model_id":"7fc9ef8f61744c79aeef1a400ac8d11f","version_major":2,"version_minor":0}

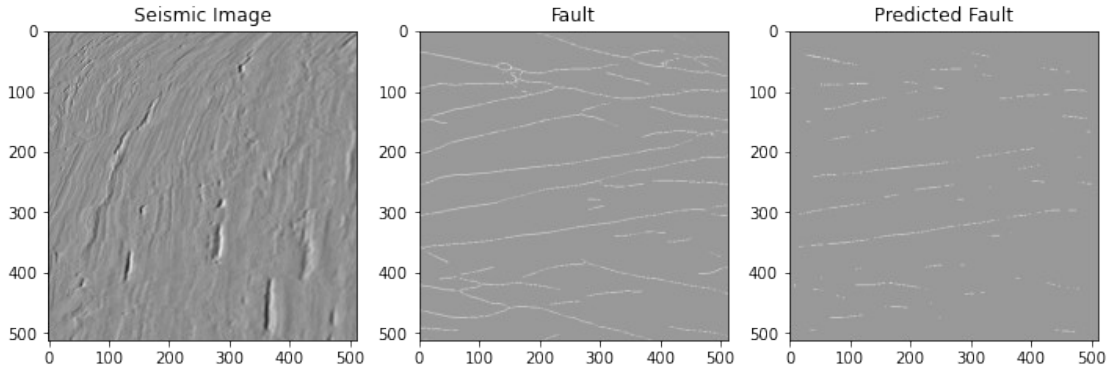
Epoch 33: Average Train Accuracy: 98.42061686268315

{"model_id":"bf047b33329e4f68b1dfabc70a189ad7","version_major":2,"version_minor":0}

Epoch 34: Average Train Accuracy: 98.42486277347854

{"model_id":"db7cef82beb047438df7e8bd64ce20ed","version_major":2,"version_minor":0}

Epoch 35: Step 3000: U-Net loss: 0.04111874848604202, Train Accuracy: 98.37532043457031



Epoch 35: Average Train Accuracy: 98.42913783453649

{"model_id":"ad7513a1bea3487da34cb74369de6b9b","version_major":2,"version_minor":0}

Epoch 36: Average Train Accuracy: 98.43360367185171

{"model_id":"f987836b76ee457198151a1a7eef6c12","version_major":2,"version_minor":0}

Epoch 37: Average Train Accuracy: 98.43816137166215

{"model_id":"6f145cd8c98d4aa396c94aa502a1fc23","version_major":2,"version_minor":0}

Epoch 38: Average Train Accuracy: 98.44266992169088

{"model_id":"26b8807d4c0d4b1d86a1f811350e7f33","version_major":2,"version_minor":0}

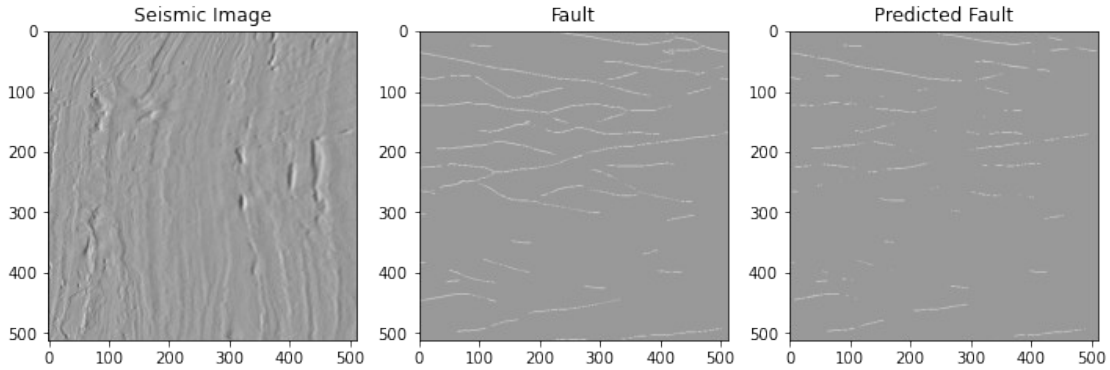
Epoch 39: Average Train Accuracy: 98.44743414486156

{"model_id":"97c3a9c4458b415ba91796d4de371542","version_major":2,"version_minor":0}

Epoch 40: Average Train Accuracy: 98.45184085358848

{"model_id":"f1a77329371540aca6152b92d7bd4317","version_major":2,"version_minor":0}

Epoch 41: Step 3500: U-Net loss: 0.02800845168530941, Train Accuracy: 98.80180358886719



Epoch 41: Average Train Accuracy: 98.45850594571324

{"model_id":"846c91dd51ed4ce1ab584713c6f397df","version_major":2,"version_minor":0}

Epoch 42: Average Train Accuracy: 98.46409057951169

{"model_id":"bf2ba08e4dee46af8ce3d194752d230d","version_major":2,"version_minor":0}

Epoch 43: Average Train Accuracy: 98.46882315242992

{"model_id":"dc116045ddea45309ea8b1c841eb478f","version_major":2,"version_minor":0}

Epoch 44: Average Train Accuracy: 98.47353816811555

{"model_id":"9dc096b7b63842aba58a430b5fbffbdf","version_major":2,"version_minor":0}

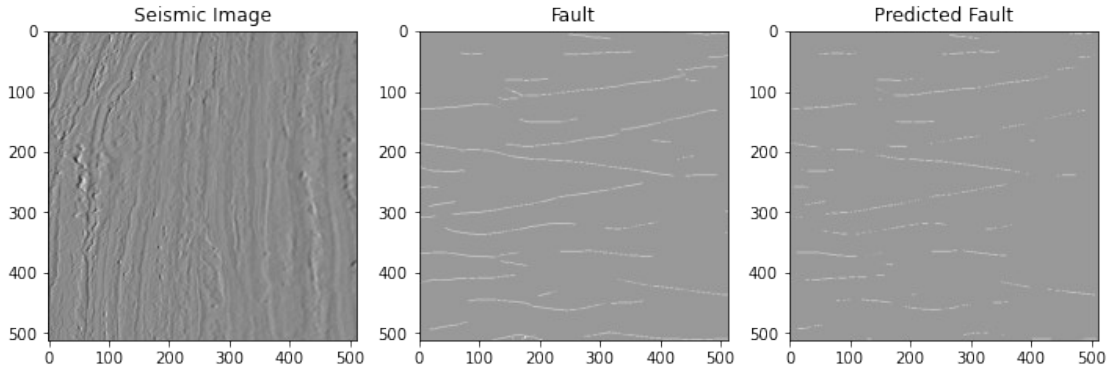
Epoch 45: Average Train Accuracy: 98.47904546791331

{"model_id":"8585065fb2494b88b328f1199e5243b9","version_major":2,"version_minor":0}

Epoch 46: Average Train Accuracy: 98.4837328895311

{"model_id":"312677a8df924191854c7db6889ddc0c","version_major":2,"version_minor":0}

Epoch 47: Step 4000: U-Net loss: 0.025001391768455505, Train Accuracy: 98.90594482421875



Epoch 47: Average Train Accuracy: 98.48995274188472

{"model_id":"15b42bf7b3b34b25ab8922c1ec031c62","version_major":2,"version_minor":0}

Epoch 48: Average Train Accuracy: 98.49418539388411

{"model_id":"2a2813250a5c45199134ef06bc18ee65","version_major":2,"version_minor":0}

Epoch 49: Average Train Accuracy: 98.49786044850069

{"model_id":"3421f0ad756f47f380b2ee87463182b9","version_major":2,"version_minor":0}

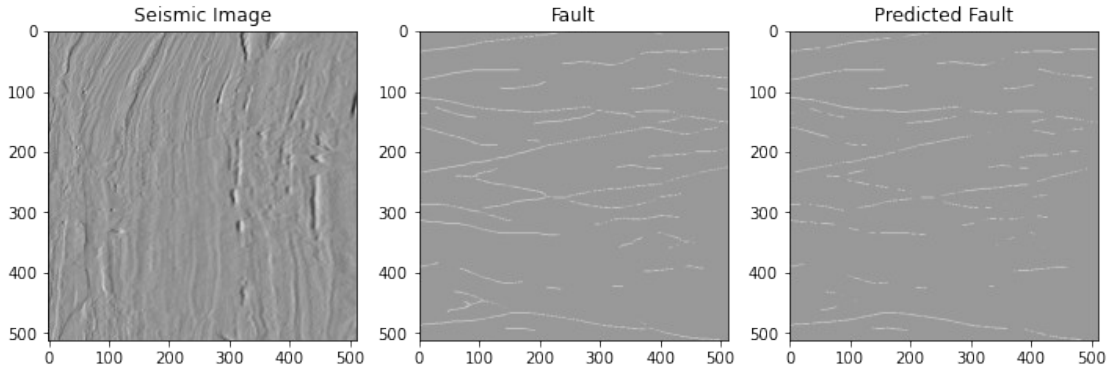
Epoch 50: Average Train Accuracy: 98.50243037134719

{"model_id":"88e16143cb854a15a5a28eaa47db87c7","version_major":2,"version_minor":0}

Epoch 51: Average Train Accuracy: 98.50772840404942

{"model_id":"993d39edeee24075b10d40687b96844e","version_major":2,"version_minor":0}

Epoch 52: Step 4500: U-Net loss: 0.02577022835612297, Train Accuracy: 98.84757995605469



Epoch 52: Average Train Accuracy: 98.51228965903228

{"model_id":"79584815e3874b3486fc343904b37b15","version_major":2,"version_minor":0}

Epoch 53: Average Train Accuracy: 98.51586668060236

{"model_id":"c174d0aea0d34bb8aadb1afa4bcac976","version_major":2,"version_minor":0}

Epoch 54: Average Train Accuracy: 98.52035506141377

{"model_id":"52a99c0f301449b59903a1863519fb41","version_major":2,"version_minor":0}

Epoch 55: Average Train Accuracy: 98.52589446957371

{"model_id":"2f63b33c29924faca93b4b7f2447dd18","version_major":2,"version_minor":0}

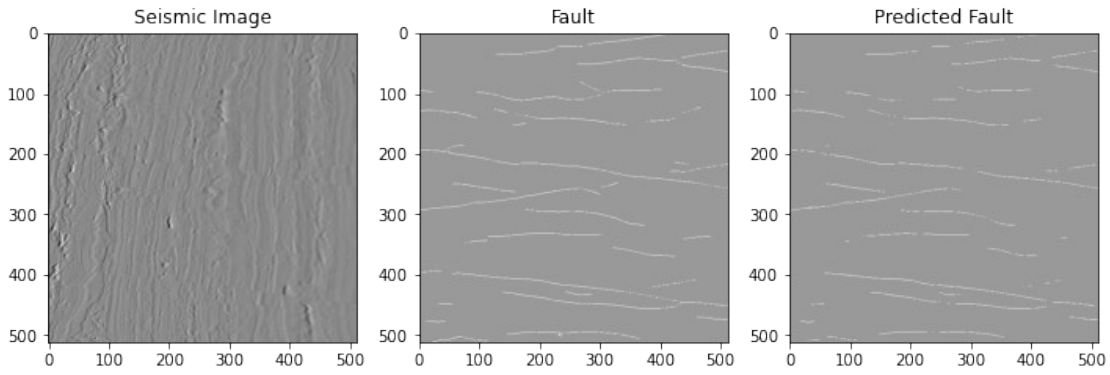
Epoch 56: Average Train Accuracy: 98.5310773190084

{"model_id":"ff48f77288994490b8c90f518a63249d","version_major":2,"version_minor":0}

Epoch 57: Average Train Accuracy: 98.53609205017709

{"model_id":"29566bff8a8141f4a5eee4e832f18727","version_major":2,"version_minor":0}

Epoch 58: Step 5000: U-Net loss: 0.020396478474140167, Train Accuracy: 99.05242919921875



Epoch 58: Average Train Accuracy: 98.54105317105325

{"model_id":"dcda9438418d4463825076a1dc32e0b0","version_major":2,"version_minor":0}

Epoch 59: Average Train Accuracy: 98.54569124707989

{"model_id":"ef4baee6563c4380829964850f83a52f","version_major":2,"version_minor":0}

Epoch 60: Average Train Accuracy: 98.55113486544721

{"model_id":"899325afc017487a8c4d8c2e8eb2dc6d","version_major":2,"version_minor":0}

Epoch 61: Average Train Accuracy: 98.55667280743651

{"model_id":"6d69383fb36841559fef247b72829c78","version_major":2,"version_minor":0}

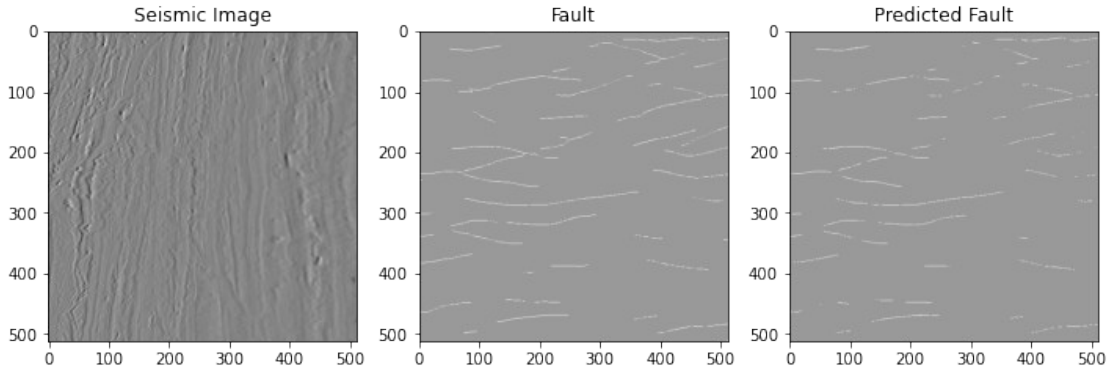
Epoch 62: Average Train Accuracy: 98.56161942112458

{"model_id":"a8d3efa6da1d46a2bcf9b4e0f582f1a6","version_major":2,"version_minor":0}

Epoch 63: Average Train Accuracy: 98.5668231459225

{"model_id":"911c9562f6cf459f9eda53066bbb459a","version_major":2,"version_minor":0}

Epoch 64: Step 5500: U-Net loss: 0.01943560317158699, Train Accuracy: 99.14207458496094



Epoch 64: Average Train Accuracy: 98.57138461980345

{"model_id":"a1b0852b47eb424a9290780cc7b7e849","version_major":2,"version_minor":0}

Epoch 65: Average Train Accuracy: 98.57484025326217

{"model_id":"df47717c09344fadaa37a759827ac464","version_major":2,"version_minor":0}

Epoch 66: Average Train Accuracy: 98.57939072927955

{"model_id":"99d2bd5ccede40c9abaa4d34460d5f41","version_major":2,"version_minor":0}

Epoch 67: Average Train Accuracy: 98.58314487760867

{"model_id":"2286d733b5c54c928bc63392b3938ab9","version_major":2,"version_minor":0}

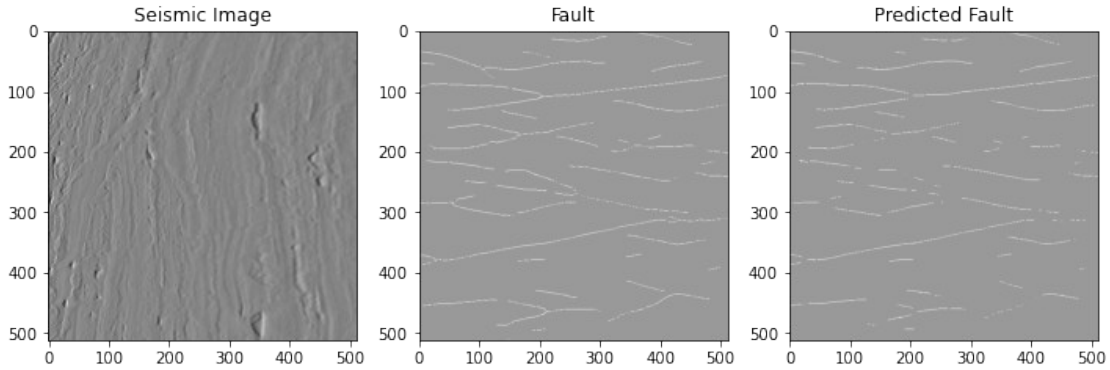
Epoch 68: Average Train Accuracy: 98.58623624965865

{"model_id":"17b6b40d2a4744d8aac09d72640fccd2","version_major":2,"version_minor":0}

Epoch 69: Average Train Accuracy: 98.59081864557346

{"model_id":"1355871d1683434c838b03cec3954b64","version_major":2,"version_minor":0}

Epoch 70: Step 6000: U-Net loss: 0.03413713723421097, Train Accuracy: 98.31161499023438



Epoch 70: Average Train Accuracy: 98.59508846254515

{"model_id":"f4a3b1f702aa4c7ba3896815f9cf26de","version_major":2,"version_minor":0}

Epoch 71: Average Train Accuracy: 98.59920938030567

{"model_id":"49df1aff54294db58df991f848edce68","version_major":2,"version_minor":0}

Epoch 72: Average Train Accuracy: 98.60243859548515

{"model_id":"0ddd8e1ab33c4116b883ca63e83b6c3b","version_major":2,"version_minor":0}

Epoch 73: Average Train Accuracy: 98.60571814265653

{"model_id":"19d8d9a03bed4657870bd02d4480f434","version_major":2,"version_minor":0}

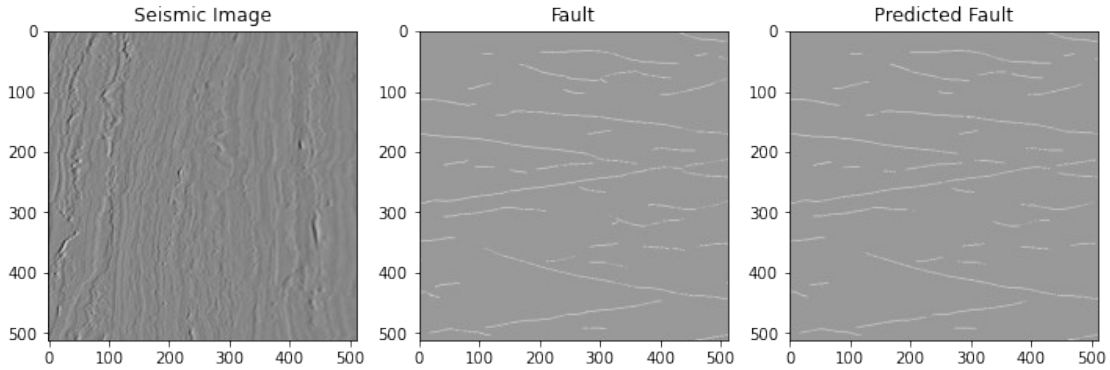
Epoch 74: Average Train Accuracy: 98.61108075310202

{"model_id":"ad7748e53791437d9b68ab0a58cbd4b5","version_major":2,"version_minor":0}

Epoch 75: Average Train Accuracy: 98.61726571900925

{"model_id":"0a17df684c534d6ea83631db6f0691b1","version_major":2,"version_minor":0}

Epoch 76: Step 6500: U-Net loss: 0.016912780702114105, Train Accuracy: 99.25956726074219



Epoch 76: Average Train Accuracy: 98.62329691398044

{"model_id":"42e06e5869ba40fb8e490c29071be7c8","version_major":2,"version_minor":0}

Epoch 77: Average Train Accuracy: 98.62902150794211

{"model_id":"bd9ed291fb7f498e82076f25d3d388be","version_major":2,"version_minor":0}

Epoch 78: Average Train Accuracy: 98.6332987815874

{"model_id":"0b73f7411b6b432a87e4573d63d00680","version_major":2,"version_minor":0}

Epoch 79: Average Train Accuracy: 98.63679706349092

{"model_id":"140a460040354b17a67d1b1137fa48b4","version_major":2,"version_minor":0}

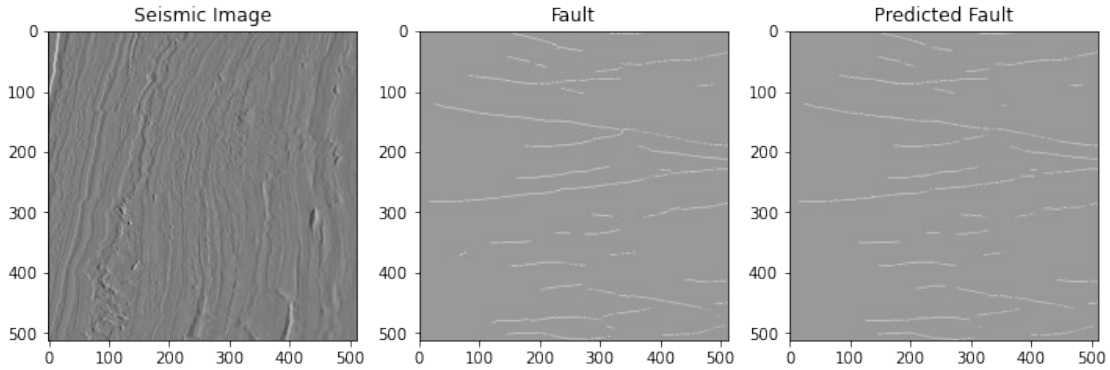
Epoch 80: Average Train Accuracy: 98.64198378574702

{"model_id":"638a1b118a1a44d798ee7813e2b341ee","version_major":2,"version_minor":0}

Epoch 81: Average Train Accuracy: 98.64651476122556

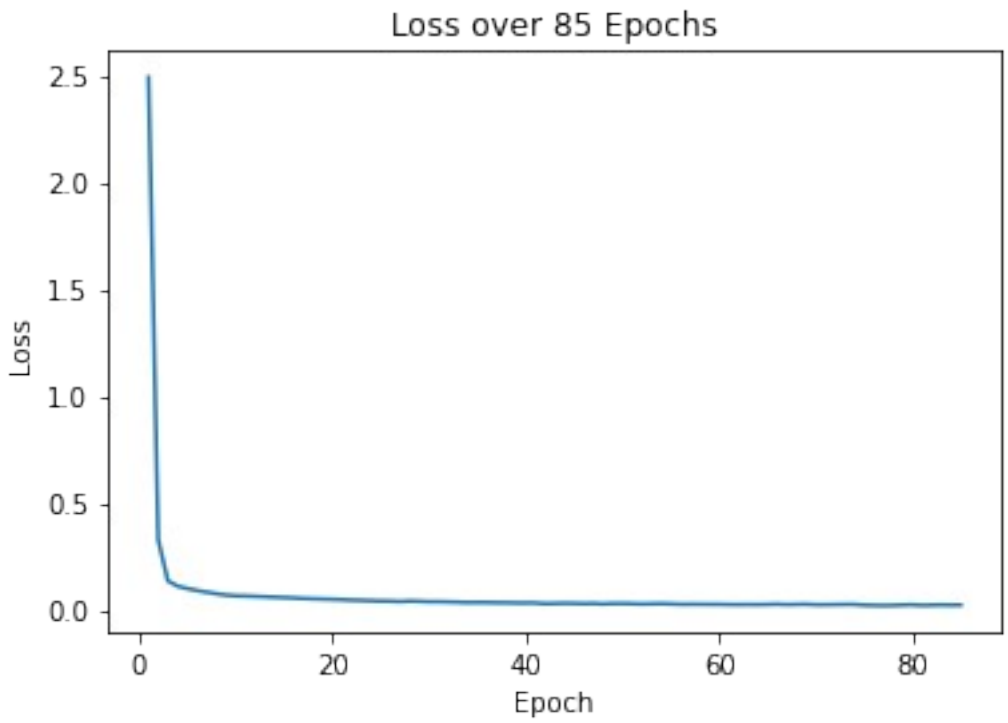
{"model_id":"c8eb404f980d4b0794b0805f28ebf33b","version_major":2,"version_minor":0}

Epoch 82: Step 7000: U-Net loss: 0.013877714052796364, Train Accuracy: 99.39002990722656

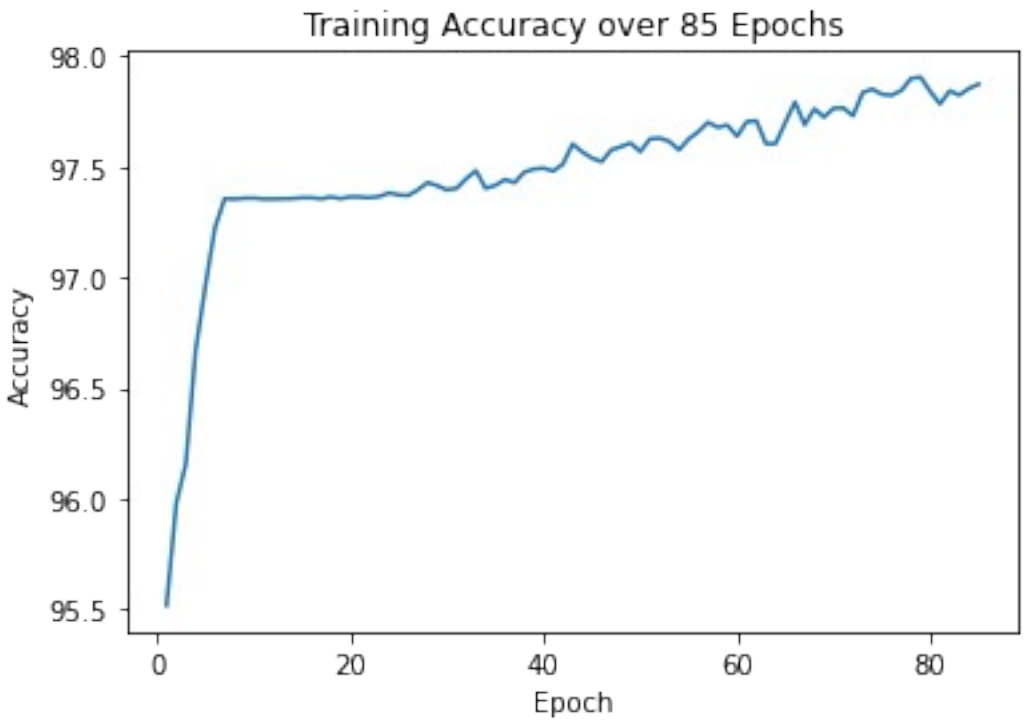



```
Epoch 82: Average Train Accuracy: 98.65017026472734

{"model_id":"4ec1fecfdd9147cb94845c7001bc2d9e","version_major":2,"vers
ion_minor":0}

Epoch 83: Average Train Accuracy: 98.65420242651504

{"model_id":"cc5c5c2a1f0649b7a9d59f24e955e907","version_major":2,"vers
ion_minor":0}

Epoch 84: Average Train Accuracy: 98.65806315596953
```

```python
# Convert the tensors to numpy array or Python float values
import statistics as st
loss1 = [st.mean(loss[i:i+85]) for i in range(0, len(loss), 85)]

loss_values = [val for val in loss1]

# Create the x-axis values for 85 epochs
epochs = range(1, 86)

# Plot the loss values against the epochs
plt.plot(epochs, loss_values)
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Loss over 85 Epochs')
plt.show()
```




```
import torch
import matplotlib.pyplot as plt



# Convert the tensors to numpy array or Python float values
loss_values = [val for val in s]

# Create the x-axis values for 85 epochs
epochs = range(1, 86)

# Plot the accuracy values against the epochs
plt.plot(epochs, loss_values)
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Training Accuracy over 85 Epochs')
plt.show()
```



Here, in above graph, its interesting to note that initial accuracy itself is 95.5%. However, this does not mean our model was able to predict fault with 95% accuracy. This merely means that, our predicted pixels were 95% similar to that of test. As true fault is on very few pixels, and rest all are zero pixels, a mere blank image full of 0 pixels will also show this 95% accuracy. Hence, evaluation should be on the basis of shape of training accuracy, rather than absolute values it shows

Testing the model

```
# Define the path to the folder containing the .npy files
folder_path = "C:/Users/gpuuser2/Desktop/unseen_fault"

# Get a list of all the .npy files in the folder
file_list = os.listdir(folder_path)
file_list = [f for f in file_list if f.endswith('.npy')]


# Load each .npy file and stack them into a three-dimensional numpy
array
fault_test = np.stack([np.load(os.path.join(folder_path, f)) for f in
file_list], axis=0)
fault_test=np.float32(fault_test)

# Print the shape of the resulting numpy array
print("Shape of fault array:", fault_test.shape)
```

```
Shape of fault array: (14, 2001, 1501)
```

```
# Define the path to the folder containing the .npy files
folder_path = "C:/Users/gpuuser2/Desktop/unseen seismic"

# Get a list of all the .npy files in the folder
file_list = os.listdir(folder_path)
file_list = [f for f in file_list if f.endswith('.npy')]


# Load each .npy file and stack them into a three-dimensional numpy
array
seismic_test = np.stack([np.load(os.path.join(folder_path, f)) for f
in file_list], axis=0)
seismic_test=np.float32(seismic_test)

# Print the shape of the resulting numpy array
print("Shape of fault array:", seismic_test.shape)
```

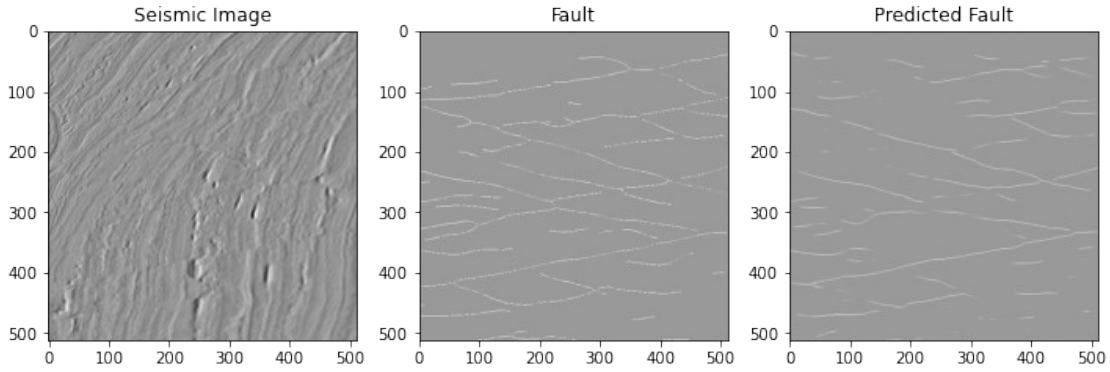
```
Shape of fault array: (14, 2001, 1501)
```

```
#converting inputs to tensors
new_shape = (512, 512)
image_list_t = []
fault_list_t = []
for i in range(seismic_test.shape[0]):
    images = torch.from_numpy(seismic_test[i])
    image_list_t.append(crop_input(images, new_shape ).unsqueeze(0))

    faults = torch.from_numpy(fault_test[i])
    fault_list_t.append(crop_input(faults, new_shape).unsqueeze(0))

# stack all of the cropped input images into a single tensor
volumes_t = torch.stack(image_list_t)
# stack all of the cropped label images into a single tensor
labels_t = torch.stack(fault_list_t)

# create a PyTorch TensorDataset object that combines the volumes
tensor and labels tensor
dataset_t = torch.utils.data.TensorDataset(volumes_t, labels_t)

# DataLoader to load the test dataset
dataloader = DataLoader(
    dataset_t,  # test dataset
    batch_size=batch_size,
    shuffle=True)  # shuffle the dataset

# iterate over each batch in the test dataset
for real, labels in tqdm(dataloader):

    # get the batch size
    cur_batch_size = len(real)
```

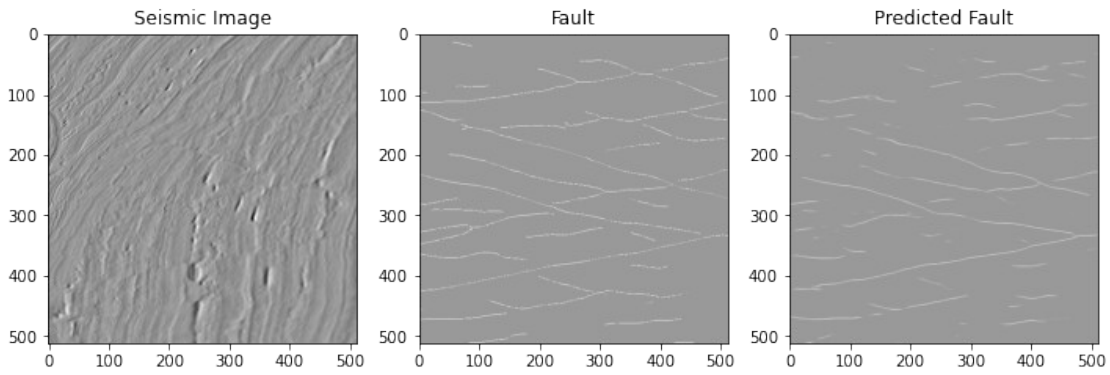
```
    # move the images and labels to the device
    real = real.to(device)
    labels = labels.to(device)
    print(real.shape)

    # perform forward pass on the U-Net model
    pred = model(real)

    # display the input image, true label image, and predicted label
image
    show_tensor_images(real.T, labels.T, torch.sigmoid(pred).T,
size=(input_dim, target_shape, target_shape))
```

{"model_id":"da5aacfb1e1f4b0c9acd67c6f42bf2ad","version_major":2,"vers
ion_minor":0}

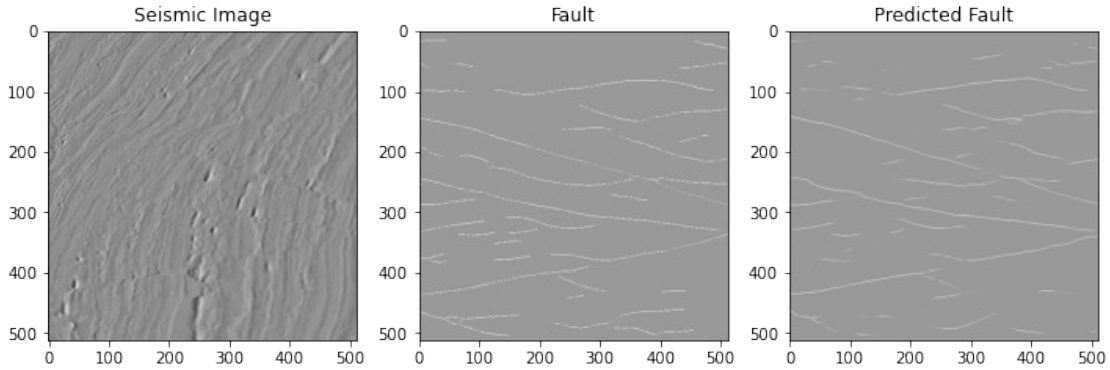
```
torch.Size([1, 1, 512, 512])
```



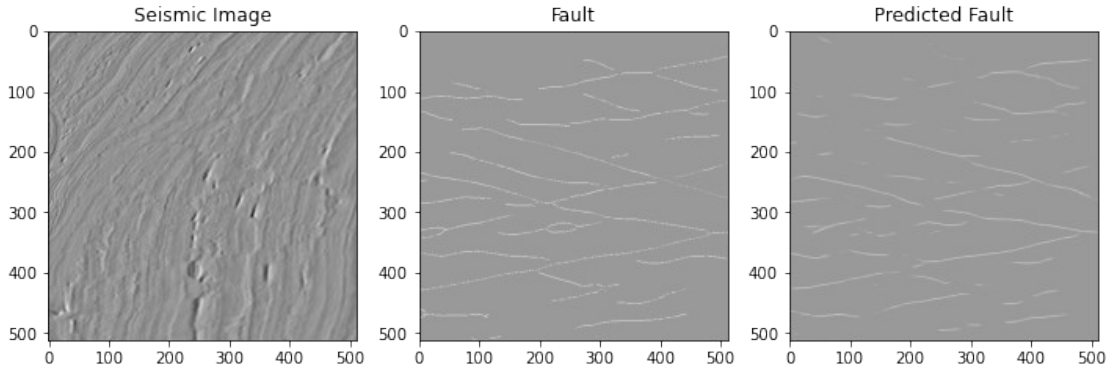

```
torch.Size([1, 1, 512, 512])
```



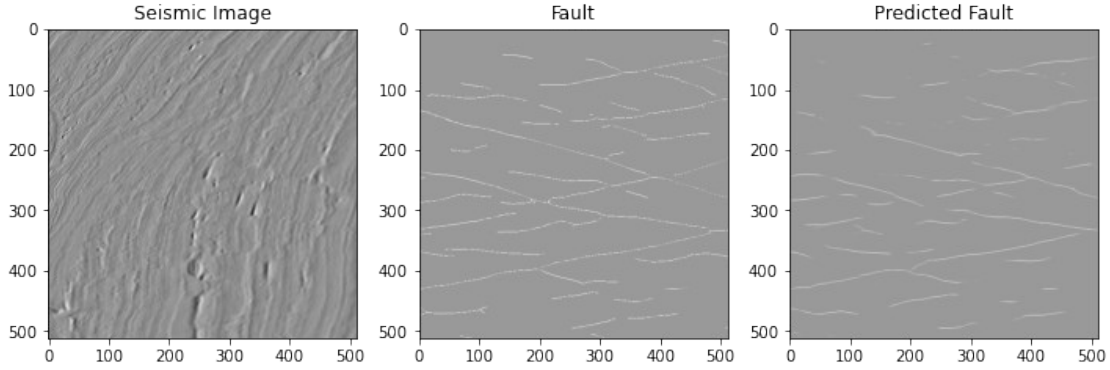

```
torch.Size([1, 1, 512, 512])
```



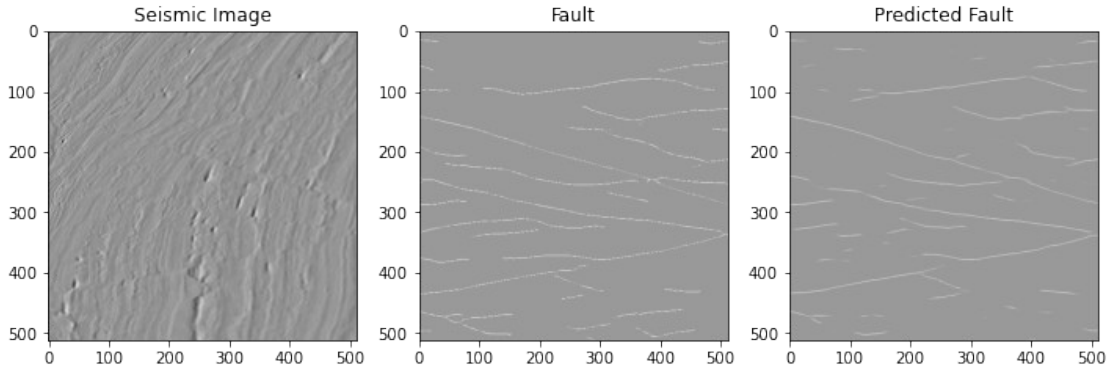

```
torch.Size([1, 1, 512, 512])
```



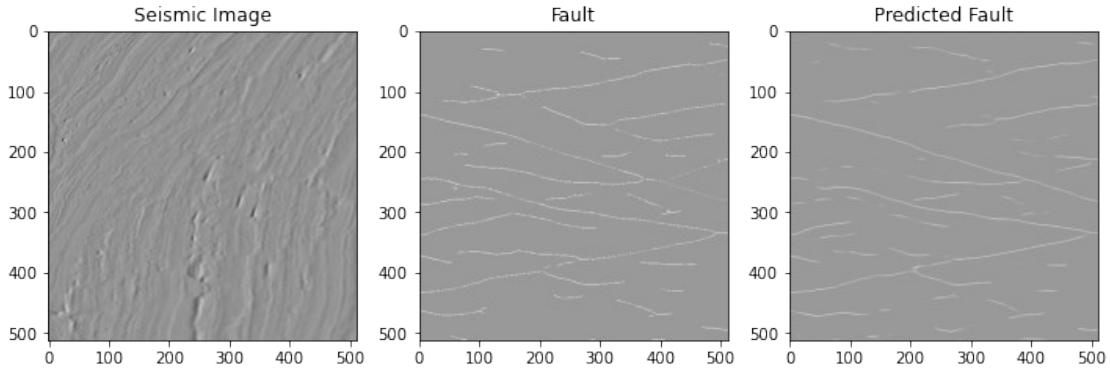

```
torch.Size([1, 1, 512, 512])
```



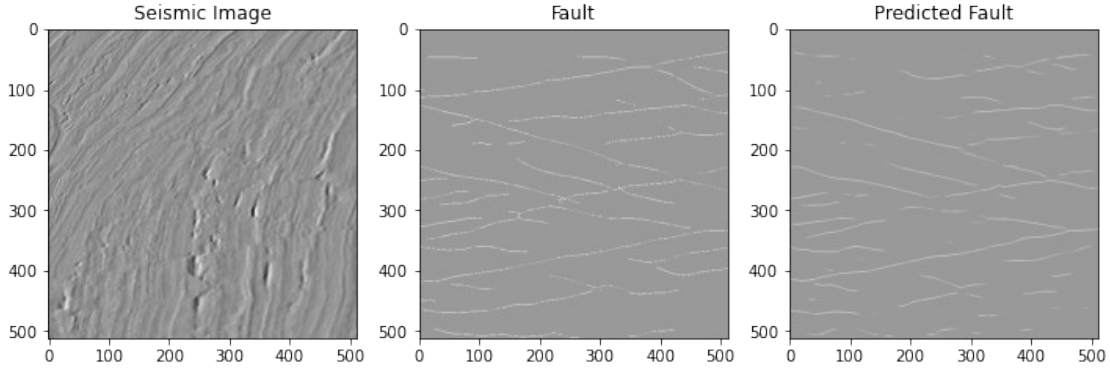

```
torch.Size([1, 1, 512, 512])
```



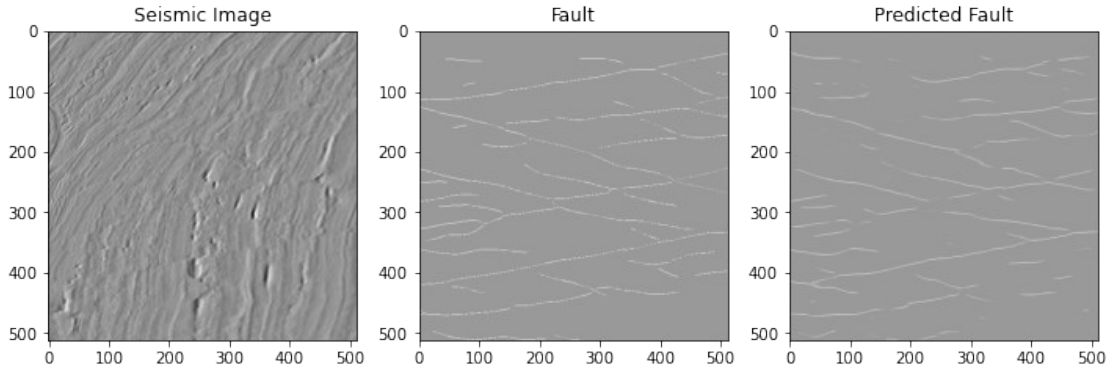

```
torch.Size([1, 1, 512, 512])
```



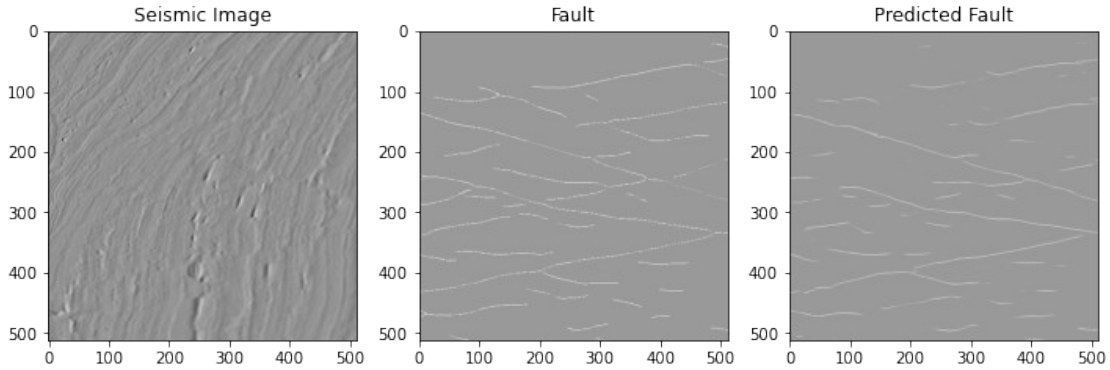

```
torch.Size([1, 1, 512, 512])
```



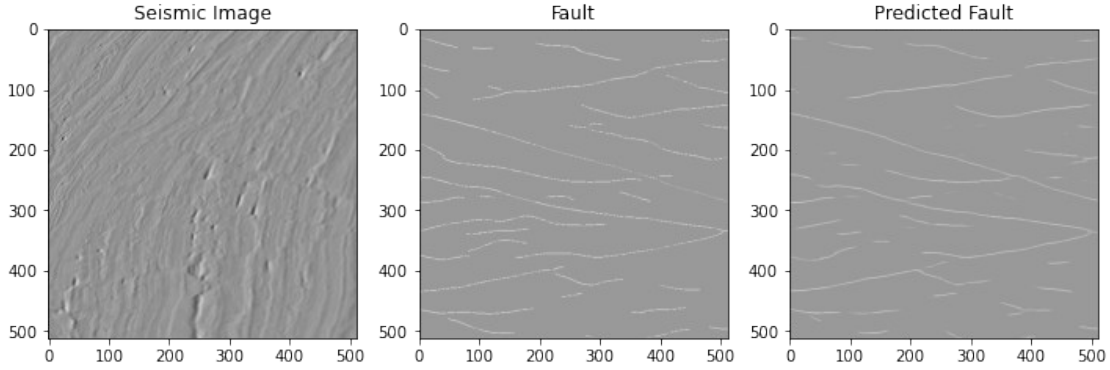

```
torch.Size([1, 1, 512, 512])
```



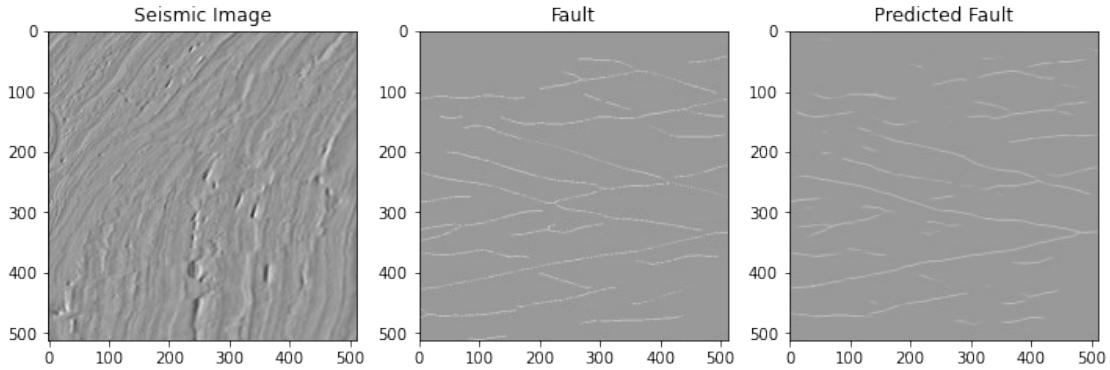

```
torch.Size([1, 1, 512, 512])
```



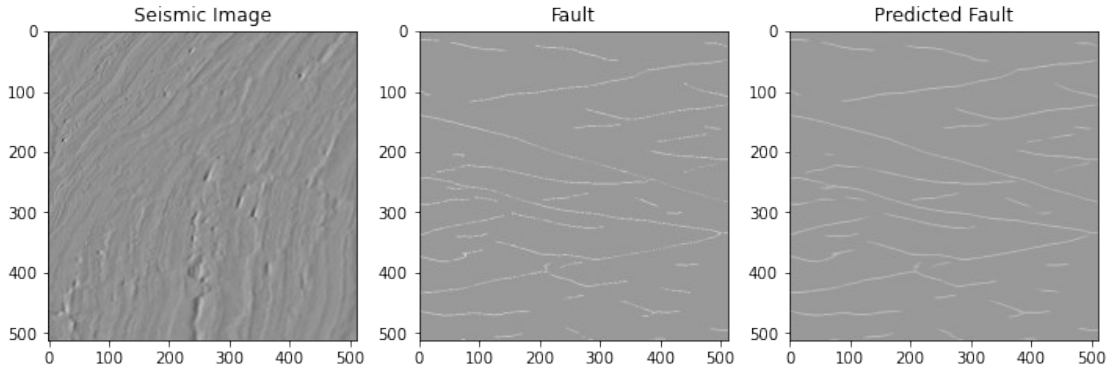

```
torch.Size([1, 1, 512, 512])
```



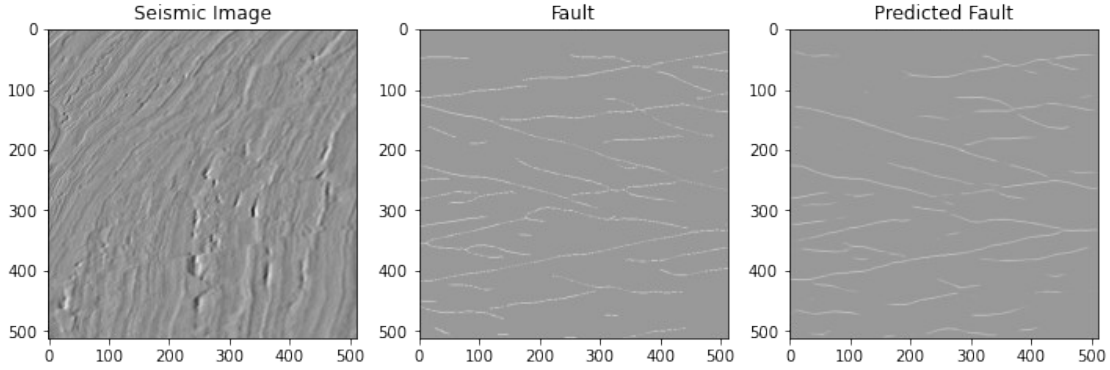

```
torch.Size([1, 1, 512, 512])
```




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




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



Saving Model

```
PATH='fault_train2'
model_path = os.path.join(PATH, "model_new"+".pt")

torch.save(model, model_path)

torch.save(model.state_dict(), model_path)
```