Machine Learning for Molecular Engineering

Problem Set 4

Date: January 23, 2024
Due: 11:59 PM ET on Thursday, April 13, 2023

Instructions Please submit your solution on Canvas as an IPython (Jupyter) Notebook file, `*.ipynb`. We highly suggest using Google Colab for this course and using this template for this problem set. Your submission should contain *all* of the code that is needed to fully answer the questions in this problem set when executed from top to bottom and should run without errors (unless intentional). Please ensure that your plots are visible in the output file and that you are printing any additional comments or variables as needed.

To get started, open your Google Colab or Jupyter notebook starting from the problem set template file here. You will need to use the data files here.

Background

Computer vision uses computation to automatically process and analyze images to perform visual tasks that humans can do. In Python, a digital image can be loaded into an `numpy.array` or `torch.Tensor` with 3 dimensions (color, width, height) (Fig. 1).

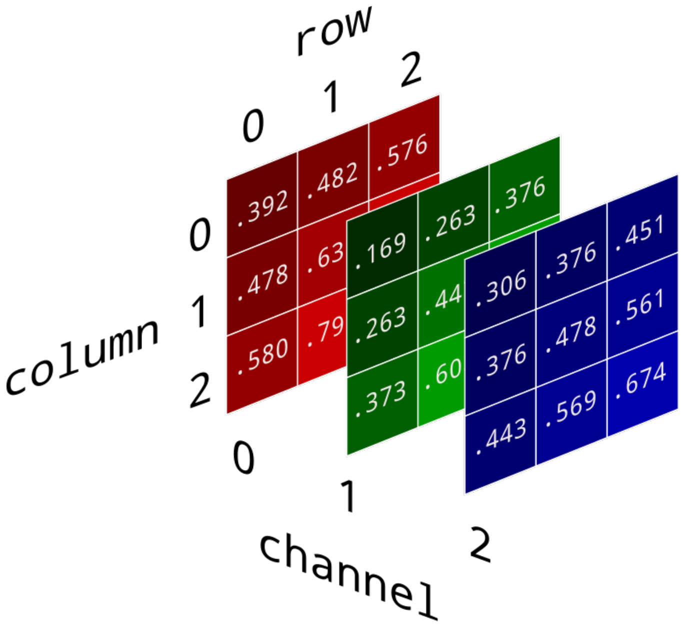

Figure 1: An image can be understood as a multi-dimensional array.

In materials and biological engineering, advanced imaging techniques using electron and optical microscopy can probe structures at multiple scales to better understand the underlying physical and chemical processes. When this information is processed in large batches, automated image processing program using machine learning can improve efficiency and accuracy. In this problem set, you will explore two tasks with computer vision techniques using PyTorch.

In the first task, you will use a Convolutional Neural Network (CNN) model to classify different types of defect on hot-rolled steel strips. Accurate classification of defects can help assess risks of potential materials failure. The machine learning technique you will apply is transfer learning. You will train a model that transfers the information stored in a model trained on ImageNet (around 14 million images!) to predict surface defects in materials engineering.

In the second task, you will use a U-net model to perform image segmentation on cell images to identify the locations and boundaries of nuclei in cells. For this task, you will be predicting an image filter. This will help scientists process images more efficiently to detect and understand different disease states of cells.

Part 1: Classify Steel Surface Defects with Transfer Learning

Strip steel is commonly used in home appliances and building constructions. The use of steel strips under different weather and mechanical conditions can cause various types of surface defects that will influence material properties like corrosion resistance, abrasion resistance and mechanical fatigue strength.

Efficient and automatic detection of these surface defects can predict component failure early to prevent an unwanted manufacturing incident. However, the traditional method based on human inspection is slow and has a high error rate [1]. You are going to develop a model that can classify 6 different types of surface defects: Crazing (Cr), Inclusion (In), Patches (Pa), Pitted Surface (PS), Rolled-in Scale (RS), and Scratch (Sc). The image dataset is obtained from the NEU Surface Defect Database [1]. Representative samples of these 6 types of defects are shown in Figure 3.

Figure 2: Industrial Steel Strip (source).

Part 1.1 (15 points) Make Image Datasets and DataLoaders

Download and unzip the NEU surface defect dataset (1800 gray-scaled images) using the code we provided. You will find image files with names like `Pa_26.jpg`. The first two letters indicate defect types.

First you need to define an `ImageDataset` object which takes the paths to the `.jpg` files. Please check the tutorial here to understand the example code. You will have to implement the `__getitem__()` method that loads and returns an image and its label given an index of the image file path. You can use `Image.open()` (from the `PIL` library) to load an image from the hard drive (not from memory) given the path to the image file and `img.convert('RGB')` to convert

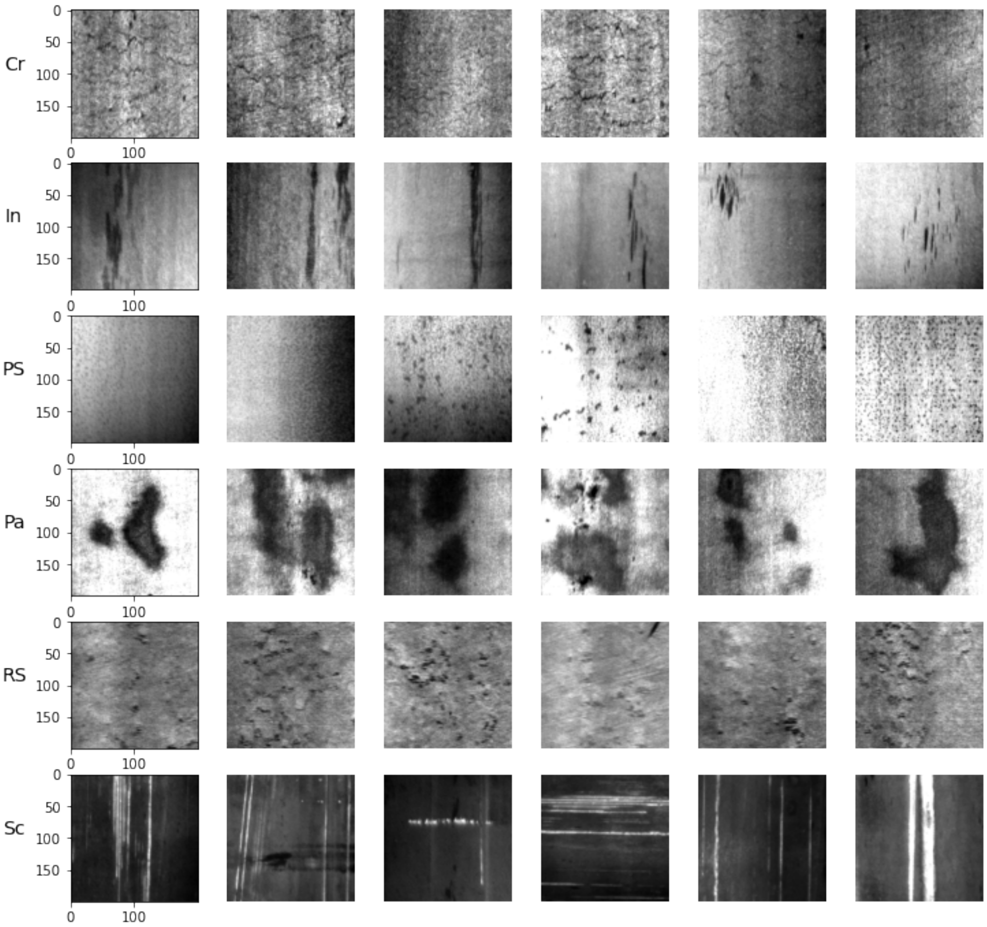

Figure 3: Six Classes of Steel Strip Surface Defects.

it to an RGB image with three color channels. You need to also retrieve the category information from file names. Encode your label into a categorical variable (`torch.LongTensor`). The `__getitem__()` method should return a tuple of an image (`torch.FloatTensor`) and a categorical label (`torch.LongTensor`).

The required input to your model is a `torch.FloatTensor`s with a shape of $(3 \times 224 \times 224)$ with each color value ranging from 0 to 1. You will need to perform data processing to get images into desired shape and format. The module you will use for this step is `torchvision.transforms` which should already be imported. Apply `transforms.Resize()` and `transforms.ToTensor()` to convert your image into a torch.Tensor with the desired shape and size. Please note that `PIL` uses the channel-last convention $(224 \times 224 \times 3)$ and `torch` uses the channel-first convention $((3 \times 224 \times 224))$. `transforms.ToTensor()` will convert the shape automatically for you. Lastly, normalize your image input with means of $(0.485, 0.456, 0.406)$ and standard deviations of $(0.229, 0.224, 0.225)$ for each color channel. You can apply `transforms.Normalize()` for this step. To combine all of these into one transform, you can use `transforms.Compose()` which will apply a list of transforms in sequence.

After you have defined your datasets, split your images randomly into train (70%), validation (10%) and test data (20%) and parse them into DataLoaders with a batch size of 4 to prepare for training a model.

Part 1.2 (10 points) Understand the Model Architecture

The model you will develop is based on the pretrained VGG16 model [2]. You can read this article for a quick overview of the model architecture. The model achieves 92.7% top-5 test accuracy in ImageNet, which is a dataset of over 14 million images belonging to 1000 classes! Load the pretrained model with the code we provided.

The model has 13 convolutional layers. Each convolution operation convolves over the image to obtain a filtered image which will be the input to the next convolution layer. Choose one image from your dataset and apply the pretrained VGG model to visualize the image. You can do this by looping over each sub-modules in `vgg16.features._modules.items()` and applying each sub-module to your selected image sequentially. Randomly visualize 4 channels of your image after 1, 5 and 10 sequential convolutions with `pyplot.imshow()`. Qualitatively describe what you observe in the image. It may help to also visualize the RGB channels of the original image.

The image is then flattened (reshaped) and passed into an MLP to obtain a feature vector of size of 4096. Mathematically, the VGG16 model converts an colored image ($\mathbb{R}^{3\times224\times224}$) into a feature vector ($\mathbb{R}^{4096}$).

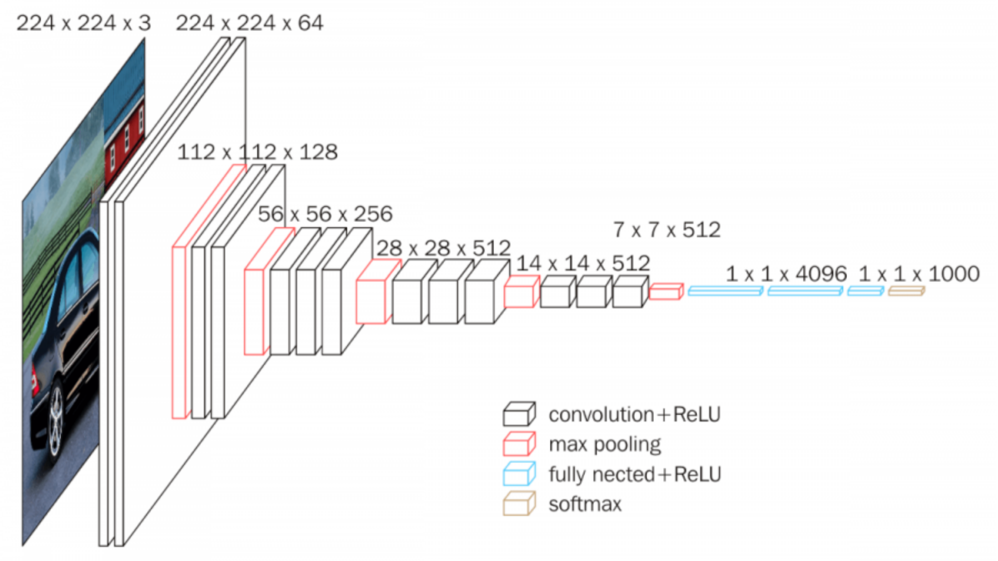

Figure 4: The Convolutional Architecture of VGG16.

Part 1.3 (20 points) Train a Classifier with Transfer Learning

In this section, we will fine-tune a classifier based on the pretrained VGG model. The VGG model operates on the input images with RGB channels and outputs a feature vector of size 4096. Since we want our classifier to output **logits** for six defect classes, we will add an MLP that produces a six-dimensional output on top of the VGG model. In the context of machine learning, a logit refers to an unnormalized output of the model. For instance, if the model's raw outputs (logits) are `[0.0, 0.0, 1.0, 1.0, 2.0, 4.0]`, the probabilities for each class would be `[0.0144, 0.0144, 0.0392, 0.0392, 0.1064, 0.7864]`, after normalization (`Softmax()`). You can see that the probabilities across the six classes sum up to 1 after `Softmax()`. For the loss function, we will use `nn.CrossEntropyLoss()`. `CrossEntropyLoss` is an operation equivalent to the combination of `LogSoftmax` and `NLLLoss`. Therefore, we will not apply an additional `Softmax` after the last MLP layer and instead have logits as the model outputs.

We will use the 4096-dimensional output from the VGG model as an input feature vector of an MLP. The MLP should have one hidden layer of size 2048 with ReLU activation. The last layer of MLP needs to output logits for six defect categories. Note that this is a transfer learning task: you are transferring the model trained on another dataset to be applied to a new problem domain, so you only need to optimize the MLP parameters. You do not need to optimize the parameters in the pretrained VGG16 model. As a result, unlike in previous problem sets, when defining your optimizer you want to feed it only the parameters you will optimize, which can be accessed via `.parameters` within any `torch.nn.Sequential` object. For the loss function, you can either use `CrossEntropyLoss` or `functional.cross_entropy` in `torch.nn` module.

Train your model for 5 epochs (yes, only 5 epochs) with a learning rate of `1e-3` and report the train and test loss. Visualize your model performance on the test dataset using a confusion matrix with `pyplot.imshow`. Annotate your confusion matrix with the number of samples in each true/predicted category pairs (there are 6 × 6 = 36 pairs); you can use `ax.text` to do this. You can compute the confusion matrix with `sklearn.metrics.confusion_matrix`.

In part 1.1, we ask you to resize and normalize RGB values of loaded images to specific values. Can you explain why you need to do that for this transfer learning task? Additionally, briefly answer the following question: what are the benefits of transfer learning versus training the entire stack (CNN + MLP). What are the potential limitations of this approach?

Part 1.4 (15 points) Obtain Saliency Maps

A saliency map is a way to interpret how your deep learning models reach a decision when classifying an image. The saliency map can be obtained by computing the gradient of your predicted logits for each class with respect to each pixel: $\nabla_x y_{\text{class}}$. The gradients of the input image pixels can be understood as how sensitive your pixel map will affect your predicted class in your model.

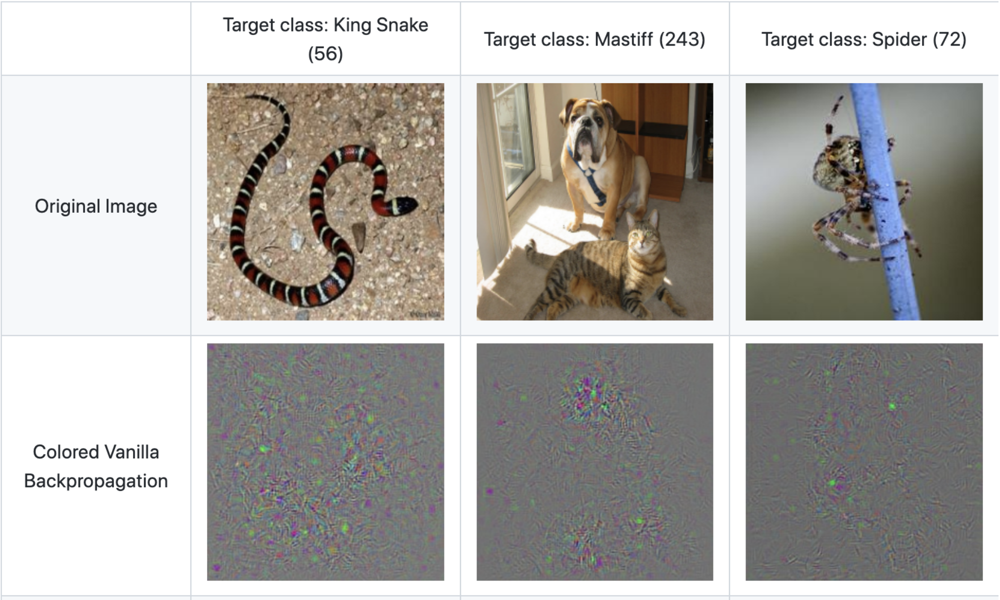
Target class: King Snake (56)
Target class: Mastiff (243)
Target class: Spider (72)
Original Image
Colored Vanilla Backpropagation

Figure 5: Visualizing saliency maps with automatic differentiation. (source)

In PyTorch, the gradient with respect to the input image can be efficiently computed with reverse-mode automatic differentiation (backpropagation) - the same way you've trained all your models. To do that, you need to first turn on the `requires_grad` flag for your input `x` with `x.requires_grad=True` and then obtain the predicted class logits with `y = model(x).max()`. Then you can obtain the gradient using `y.backward()` which performs backpropagation. You can retrieve the computed gradient with `x.grad.data`. This is the same as how you would compute the gradients on your model parameters with `loss.backward()`. For the computed gradient ($\mathbb{R}^{3\times224\times224}$), only take the absolute values and take the mean for the three color channels. You would obtain a sensitivity value for each pixel of shape (224×224).

Select two input images from each defect class, and obtain and visualize the saliency map for these images with `pyplot.imshow()`. Briefly comment on any pattern you observe in the saliency map.

Part 2: Image Segmentation

In computer vision, image segmentation is a computational technique that partitions images into different parts for easier analysis and processing. This technique has been very useful for photo editing and videoconferencing software. For example, Zoom uses image segmentation to add virtual backgrounds to your video stream.

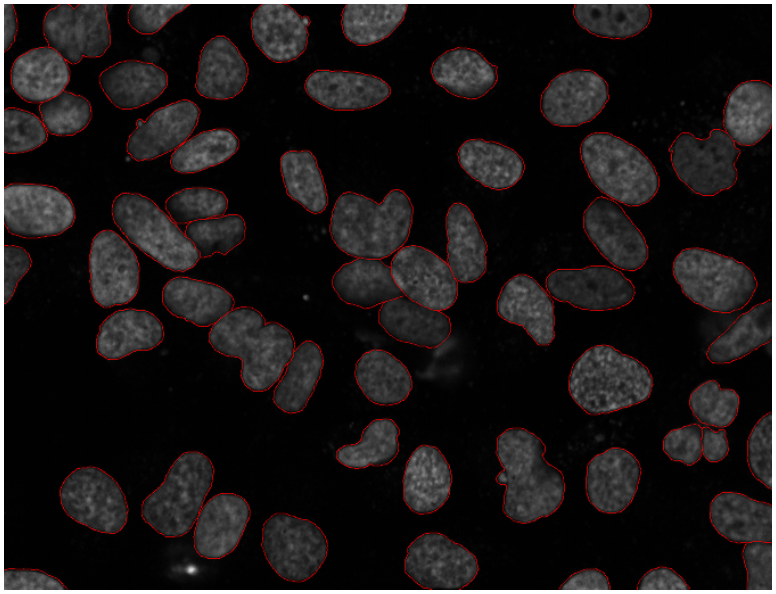

Figure 6: Applying image segmentation to cell images. The red lines marks the boundaries of cells.

Image segmentation is also useful for analyzing cell images. Accurate and efficient segmentation of the cell nuclei improves the quality of disease states assessment for cells. In this question, you are given a dataset of cell images, each labeled with a pixel-wise mask to indicate if it belongs to a nucleus or not. The dataset comes from the 2018 Kaggle data science bowl.

This part can be done independently of part 1 in a separate Colab session.

Part 2.1 (15 points) Build Datasets and DataLoaders

Download the image datasets with code we provided. Make your own image dataset with a `__getitem__()` method that loads and processes images (`X`) and segmentation masks (`y`). Similar to what you did in Part 1, make an `ImageDataset` object as a subclass of `torch.utils.data.dataset`. We provided a function `parse_data()` that reads the image and its mask given paths to the image files. The cell images you will load have 4 color channels: cyan, magenta, yellow, and key (black). `parse_data()` returns a tuple of images as `np.arrays` with dimensions of 256 × 256 × 4 (CMYK image) and 256 × 256 × 1 (mask). Like in part 1, you can convert them to channel-first `torch.Tensors` (4 × 256 × 256) using `transforms.ToTensor()`.

Build your Datasets (70% train, 10% validation, and 20% test) and DataLoaders with a batch size of 4. To split your data, apply the split to the `paths` list generated in our code, and feed in the resulting `paths` to your `ImageDataset` class.

Ideally, your model predictions should respect rotational and translational invariance. While CNNs are invariant to translations, rotational symmetry is not embedded in a CNN architecture. [1] To implicitly make your model adapt to different possible rotations, you can apply random rotation transformations to your training image samples. This technique is called **data augmentation**. Do you think random rotation data augmentation was necessary for our specific task of cell image segmentation? Also, why are CNNs invariant to translations? Briefly justify your answer.

Part 2.2 (20 points) Train a U-Net Model that Performs Image Segmentation

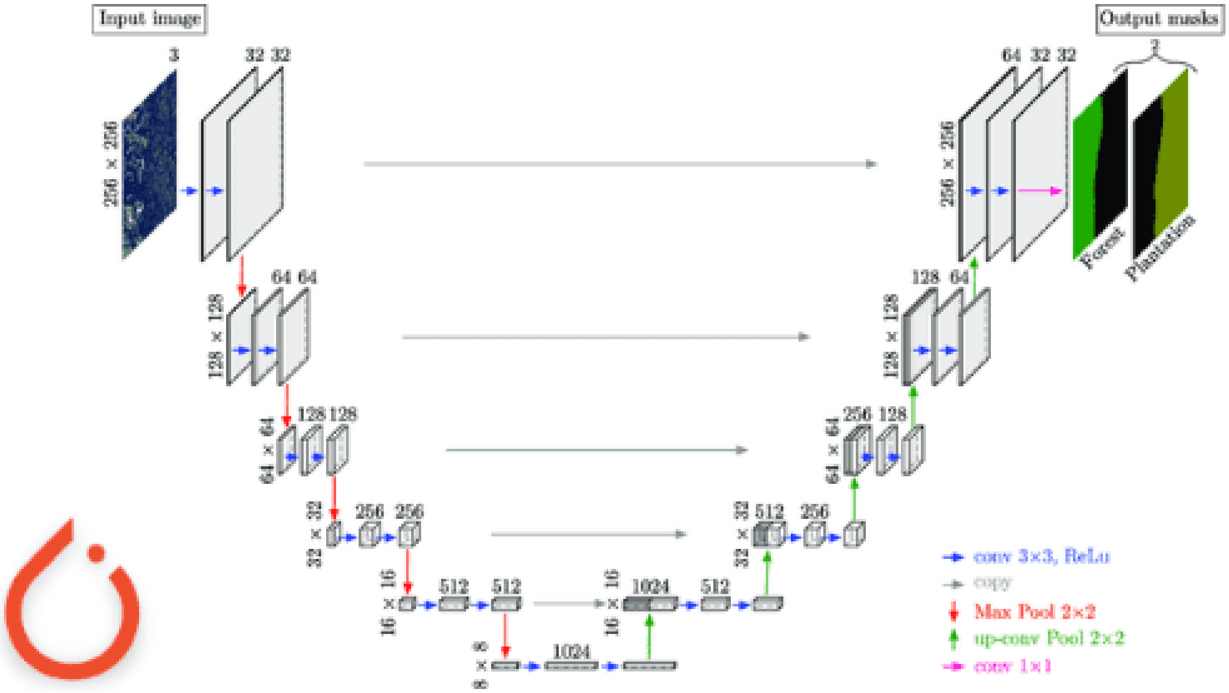

Figure 7: The U-Net architecture. The name stems from its U-shaped architecture. [4]

The model you will use is called U-Net which is a popular method for image segmentation[4]. Unlike a CNN model that successively contracting image information into a scalar output for

[1] It is possible to incorporate rotational equivariance in images, please see [3]

classification, a U-Net model uses an encoder and a decoder to parameterize pixel-wise output to predict pixel-wise labels (Fig. 7). A final pixel-wise sigmoid transformation is used to ensure that the pixel-wise outputs are from 0 to 1. Mathematically, the model takes an image and outputs an segmentation filter ($f : \mathbb{R}^{4\times256\times256} \to \mathbb{R}^{1\times256\times256}$).

A common loss function for training an image segmentation model is the Dice loss [5] (from Sørensen–Dice coefficient). Let p_i be your predicted mask and t_i be your target mask, the Dice loss is defined as :

$$\mathcal{L}_{\text{Dice}} = 1 - \frac{2\sum_i p_i t_i + 1}{\sum_i p_i + \sum_i t_i + 1} \tag{1}$$

Please implement the loss function `dice_loss()` which takes the predicted mask and the labeled mask as arguments. +1 is included in the numerator and the denominator to deal with edge cases when there are no predicted mask in an image.

We have implemented a untrained U-Net model for you to use. Train your model for 20 epochs with a learning rate of `1e-3` to obtain a model that predicts the segmentation mask for each image. You will need to convert your images and masks to float tensors using `.to(float)` for them to train properly. We have provided an function `plot_seg()` for you to visually compare the predicted segmentation and ground truth segmentation; check the dimensions requirements of `plot_seg()` carefully. When your training is finished, show the segmentation results for three images in the test data using `plot_seg()`.

Part 3: Feedback Survey

Part 3.1 (5 points) Please finish the feedback survey on our Canvas site

References

[1] Song, K. & Yan, Y. A noise robust method based on completed local binary patterns for hot-rolled steel strip surface defects. *Applied Surface Science* **285**, 858–864 (2013).

[2] Simonyan, K. & Zisserman, A. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[3] Weiler, M., Geiger, M., Welling, M., Boomsma, W. & Cohen, T. 3d steerable cnns: Learning rotationally equivariant features in volumetric data. *arXiv preprint arXiv:1807.02547* (2018).

[4] Ronneberger, O., Fischer, P. & Brox, T. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, 234–241 (Springer, 2015).

[5] Jadon, S. A survey of loss functions for semantic segmentation. In *2020 IEEE Conference on Computational Intelligence in Bioinformatics and Computational Biology (CIBCB)*, 1–7 (IEEE, 2020).