Machine Learning for Molecular Engineering

Problem Set 5

Date: April 27, 2021
Due: 10 pm ET on Thursday, April 8, 2021

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 the first problem set. If you have not used Google Colab, you might find this example notebook helpful. After opening this template link, you should select "Save a copy in Drive" from the File menu. When you've completed the assignment, you can download the `*.ipynb` file from the File menu to upload to Canvas. 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. Commenting your code in-line and adding any additional comments in markdown (as "Text cells") will help the grader understand your approach and award partial credit. You are encouraged to ask for clarification on Piazza if you find any of the question statements unclear or ambiguous; chances are, another student feels the same way!

To get started, open your Google Colab or Jupyter notebook starting from the problem set template files for part 1 and part 2. 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).

In materials and biological engineering, advanced imaging techniques using Electron Microscopy and optical microscopy can probe the structures at multiple scales to better understand the underlying physical and chemical processes. When these information are processed in large batches, automated image processing program using machine learning can provide the benefit of efficiency and accuracy to extract information. 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 (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. Different from a classification or regression 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.

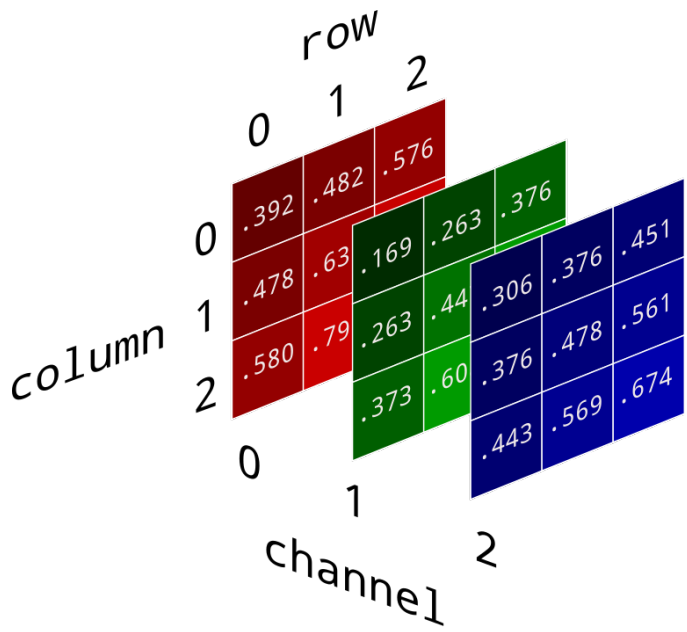

Figure 1: An image can be understood as a multi-dimensional array

Part 1: (15 points) 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 materials 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 unwanted manufacturing incident due to materials failure. However, traditional method based on human inspection is slow and has shown high error rate [1]. You are going to develop a model that 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 4.

Part 1.1 (15 points) Make Image datasets and dataloader

Download and unzip the NEU surface defect dataset (1800 gray-scaled images) using the code we provided. You will find images files with a name like `Pa_26.jpg`. The first two letters indicate defect types. Split your images randomly into train(70%), validation(10%) and test data(20%).

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 again implement the `__getitem__()` method that loads and returns an image and its label given an index of the image file path. You can use `PIL.Image.open()` to load an image given the path to the image file. The function should directly load images for the hard drive instead of loading all the image data into memory.

Figure 2: Industrial Steel Strip source

The `__getitem__()` method returns a tuple of image(`torch.FloatTensor`) and a categorical label (`torch.LongTensor`). The required input to your model `torch.FloatTensor`s with a shape of $(3 \times 224 \times 224)$ with each color value ranging from 0 to 1. You need to perform data processing to get images into desired shape and format. The two useful libraries are `PIL` (Python Imaging Library) and `torchvision.transforms` modules which are pre-installed on Colab. You can visualize each image with `matplotlib.pyplot.imshow()`.

First load your image with `Image.open()` in `PIL`. This will give you a gray-scaled image object with only one color channel. You can use `img.convert('RGB')` to convert it to a RGB image with three color channels (if `img` is your loaded image object). Then, you can apply `transforms.ToTensor()` and `transforms.Resize()` 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. (Think about why you need to normalize your image data to these specific values. You need to answer a relevant question in part 1.3).

You need to also retrieve the category information from file names. Encode your label into a categorical variable (`torch.LongTensor`). After you have defined your datasets, parse them into dataloaders 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 the article here 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 filtered images. You can do this by looping over each sub-modules in `vgg16.features` and apply the module to images sequentially. Randomly visualize 4 channels of filtered image after the 1, 5 and 10 sequential convolutions layer with `pyplot.imshow()`. Qualitatively describe what you observed in the filtered images. The filtered 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 ($\mathbb{R}^{4096}$).

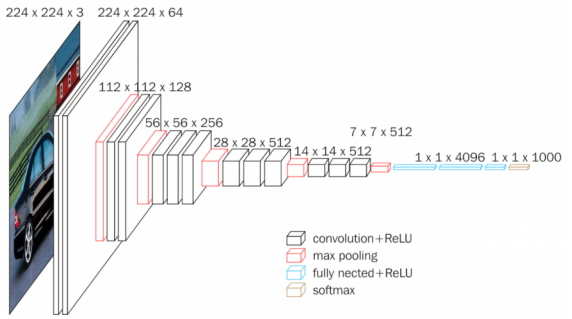

Figure 3: Convolutional architectures of VGG16

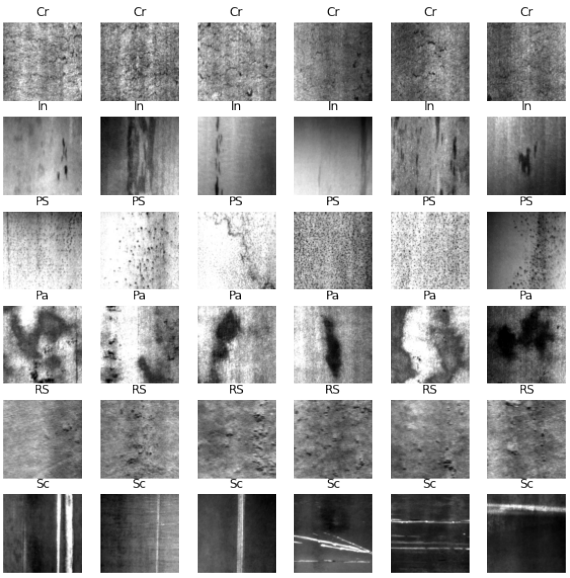

Figure 4: six classes of steel strip surface defects

Part 1.3 (20 points) Define and train a classifier with transfer learning

The VGG model operates on the input images with RGB channels and outputs a feature vector of size 4096. Use this vector as features to train an MLP with one hidden layer of size 2048. The model need to output logits for different 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 optmize the MLP parameters. You do not need to optimize the parameters in the pretrained VGG16 model. 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 batch size of 4 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 $6 \times 6 = 36$ pairs). 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 benefit of transfer learning versus training the entire stack (CNN + MLP). What are the potential limitations of this approach?

Part 1.4 (15 points) Obtain salience maps

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: $\nabla_x y_{class}$. The gradients of the input image pixels can be understood as the sensitivities of how your pixel map will affect your predicted class in your model.

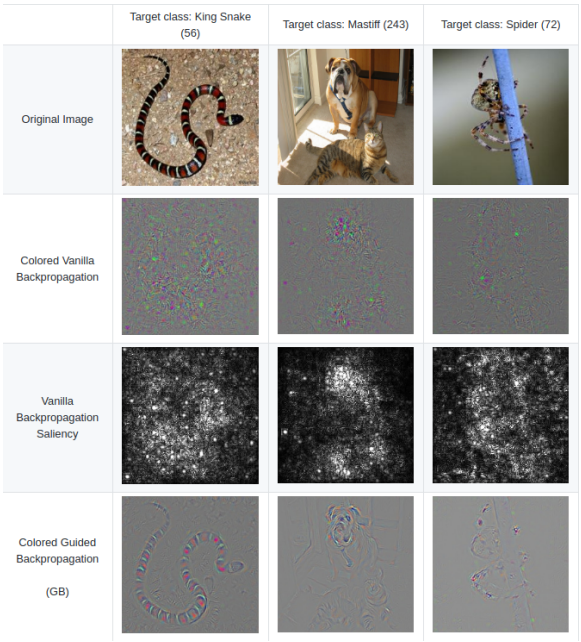

Figure 5: Visualizing saliency maps with automatic differentiation.(source)

In pytorch, the gradient with respect to the input image can be efficiently computed with reverse-mode automatic differentiation. 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 reverse mode AD (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 your 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 and conferencing softwares. For example, Zoom uses image segmentation to add virtual backgrounds to your video stream.

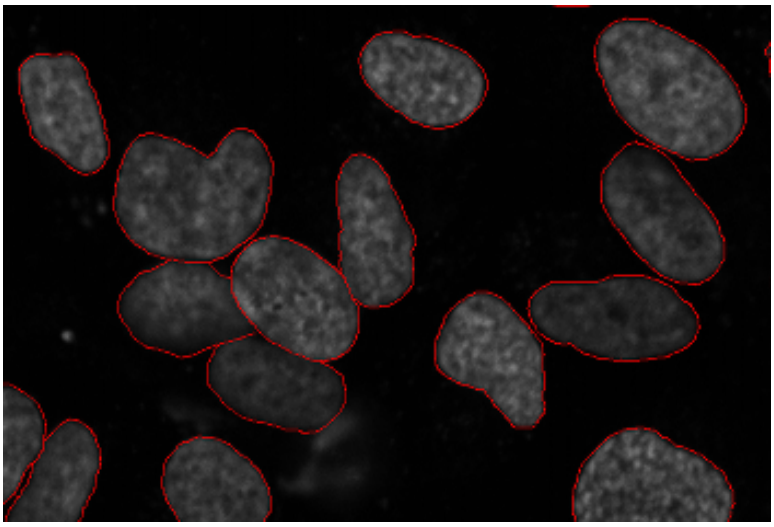

Figure 6: Applying image segmentation to cell images. The red lines marks the boundaries of cells

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 (link).

Part 2.1 (15 points) Build data-sets and data-loaders

Download the image data-sets 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 image dataset object as a sublcass of verb+torch.utils.data.dataset+. We provided a function `parse_image()` that read 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_image()` returns a tuple of images as `np.arrays` with dimensions of $256 \times 256 \times 4$ (CMYK image) and $256 \times 256 \times 1$ (mask). Like in part 1, you can convert them to channel-first `torch.Tensors` ($4 \times 256 \times 256$) using `transforms.ToTensor()`.

Ideally, your model should respect rotational invariance. However, this 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 both images and the corresponding segmentation mask. The function you need is `transforms.functional.rotate`. To do this, randomly sample a rotation degree between -60 and 60 degrees and apply the same rotation to the image pair in the `__getitem__()` method after the image is loaded. Build your datasets (70 % train, 10% validation, and 20% test) and dataloaders with a batch size of 4.

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

Rotating images is a common technique to make your classifier adapt to images with different rotations. Moreover, your model prediction should also be invariant to translations. Is it necessary to also apply translations to your image? (Hint: think about the symmetry preserved by the convolution operation) Briefly justify your answer.

Part 2.2 (20 points) Train a U-net model that performs image segmentation

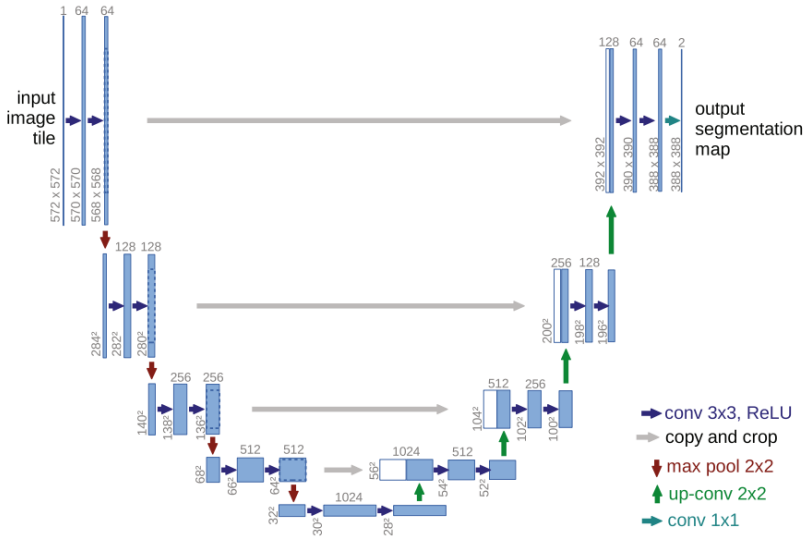

Figure 7: The U-Net architecture. The name stems from its U-shaped architecture. [4]

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 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 ($f : \mathbb{R}^{4\times256\times256} \rightarrow \mathbb{R}^{1\times256\times256}$).

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 :

$$L_{dice} = 1 - \frac{2\sum_i p_i t_i + 1}{\sum_i p_i + \sum_i t_i + 1} \tag{1}$$

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 yo use. Like before, split your data into train, validation and test datasets and feed them into three loaders. Train your model for 20 epochs to obtain a model that predicts the segmentation mask for each image. We provided an function `plot_seg()` for you to visually compare the predicted segmentation and ground truth segmentation. 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).