MICHAEL J. VAN SLYKE

A COMPUTER VISION MODEL WHICH DETECTS BONE FRACTURES IN THE UPPER EXTREMITIES NAMELY: WRISTS, FOREARMS, UPPER ARM, & SHOULDER FRACTURES

PROBLEM STATEMENT:

The Human body is a deeply complex organism, so much so that people have dedicated their lives to understanding it. X-ray machines are a wonderful device which has greatly increased our insight into the human body, particularly that of our bones. With that being said, interpreting the images can be difficult for novice and trainee medical staff, especially when fractures in the bones are very small. The aim of this project is to aid in fracture classification so as to expedite treatment of fractures in the wrist, arm and shoulder.

Thus this project endeavored to Build an AI over the course of a week that detects bone fractures in X-rays images at at least 85% accuracy to aid medical personnel in diagnosing fractures in the upper extremities and thus expediting treatment.

Annotated X-ray images were pulled from kaggle.com's 'bone fracture detection computer vision project' dataset which were analyzed by leveraging the Deep Learning models found in PyTorch.

DATA WRANGLING:

Due to the data originating from Kaggle.com, It was really quite clean. Too clean as a matter of fact as will be described later in the EDA section. There was a large body of approximately 8000 X-ray images for training, an extra 2000 for Validating (an important part of Computer Vision modeling as will be described later) and another 169 images for testing the model. Along with the images was a matching annotation file with integer and decimal values providing true classification data and demarking bounding box locations respectively. In the process of initial training of the model it became clear that hidden in this mass of data there were in fact errors and anomalies which needed to be addressed in order to improve modeling and performance.

The first error being: there was no classification for 'no fracture detected' These annotations were simply empty .txt files for which I provided an acceptable null value for the bounding boxes and for classification.

The other common errors found were extra boundaries for bounding box data. This was caused due to the annotations trying to cover too much ground per se. There are three ways to annotate a bounding box and these annotations endeavored to cover all of them with the only delimiter between the data points being a space. Although I knew that a .json file would have served this functionality better, which at its heart is a good idea, it was well outside of the scope and time limitations set for this project and thus in importing data with my __getitem__ method of my dataset class, I had to instantiate a series of logic gates to address the discrepancies in annotation data.

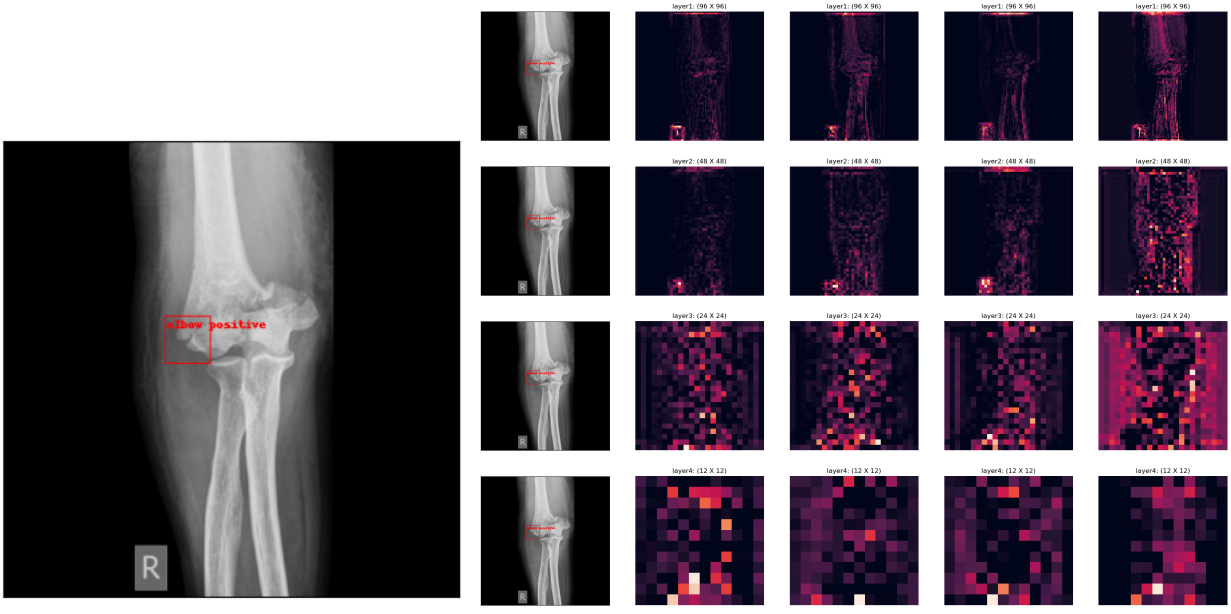
MICHAEL J. VAN SLYKE

EXPLORATORY DATA ANALYSIS/FEATURE ENGINEERING

Image data is very different from other kinds of data sources. With images, which are essentially 3-dimensional arrays of pixel data, it is very difficult to establish a pattern with a computer that is intelligible; Made doubly frustrating that we as human beings can clearly see with our eyes and experience that say a humorous bone has been snapped clean in two for example. However, since the computer can only see and replicate the data in that aforementioned 3-dimensional array, How do we teach it to see what is there?

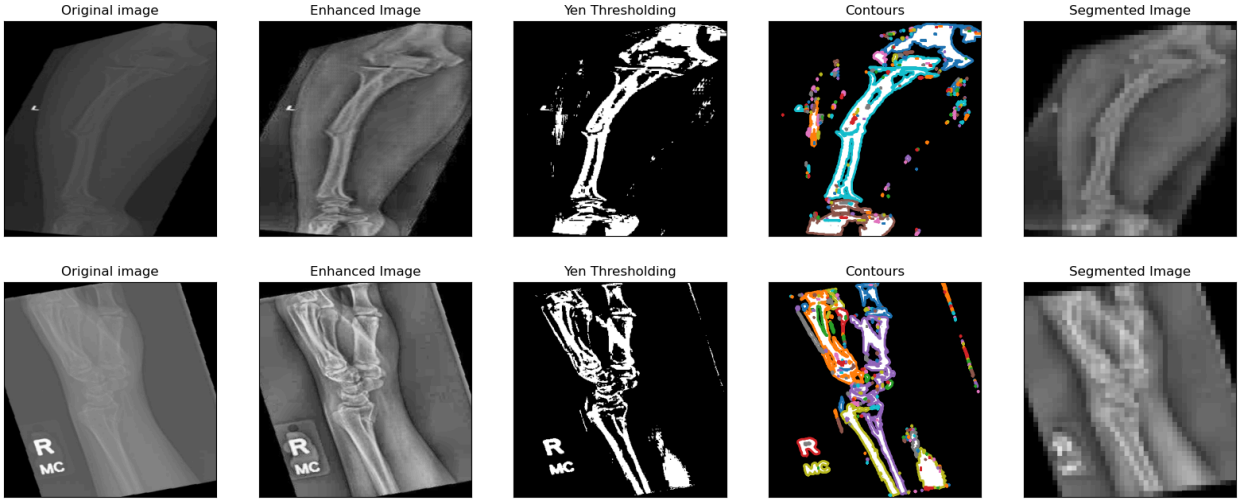
Thus feature engineering becomes our primary resource. With PyTorch models we need our images to all be the exact same dimension in both Height and Width to standardize the models input, We also muddy the waters, (no such thing as a perfect picture) so we intentionally introduce 'noise' into the data set to make it more general use. Examples would be horizontal/vertical flips, blacking/whiting out portions of the image matrix, introducing gaussian blurs, or random color noise for example. Also if we wish to see what features our trial models are examining when 'looking' at images in our dataset, we can see very interesting patterns emerge. Computers 'see' by moving a mask, a 3X3 block of pixels for example, which it then takes the average of that sector of pixels and moves over 1 or 2 pixels (we can configure how much we want it to move) over the image and continues to take averages and it proceeds to make patterns, whether these patterns are relevant or not is determined by the supervision of the bounding box and classification annotations.

That's a mouthful of explanation to say that the computer continues to simplify the image pixel data and focuses on line patterns it finds, and when we extract those features, we get images like these:

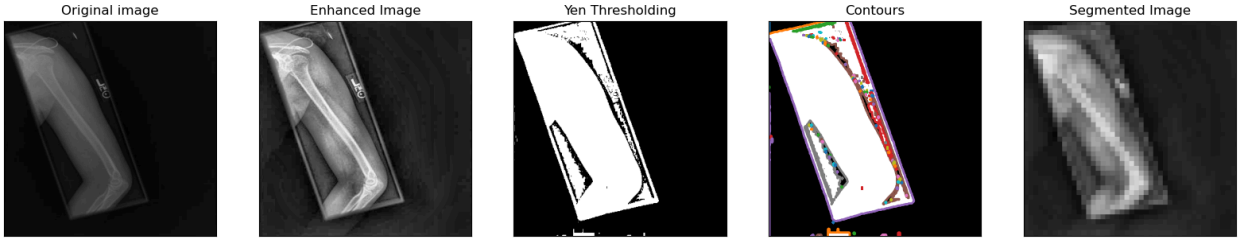


PyTorch deep learning is heavy work, and I had to learn it on the fly for this project. It was very satisfying but it did make it worth it to pursue other avenues which use conventional Machine

Learning techniques from SKLearn and SKImage. However I decided that for practical and precision purposes, it would be best to go the Deep Learning route While considering a conventional machine learning classifier model, I developed these images as potential feature set candidates.



I changed my mind to deep learning however from inconsistency in the thresholding step. Sometimes images would be masked to the degree of absolute uselessness as seen here:



MODEL PREPROCESSING:

With the heavy lifting now done It was time to get these images ready for analysis. Image data is complex and difficult to calculate through. So PyTorch created a new kind of numpy array called a Tensor. What is special about a Tensor is 2 fold, Firstly and most importantly, Tensors can be processed non-sequentially on GPU accelerators called CUDA cores, this vastly speeds up image processing and consequently model training time. Secondly their shape is backwards. So where a nd.array() image would be marked (H, W, C) where H=Height, W=Width, and C=Color, a Tensor image would have Color in front (C, W, H). Since the images are set to be 224 X 224, a Tensor would look like (3, 224, 244) for its shape. This is important as during this process images need to be permuted back and forth in order to be processed, trained on and then visualized.

Fortunately PyTorch provides a method to stage all transformation processes prior to feeding images in batches to the model. I landed on a selection of 7 transformations for the image data: first to change their HXW to 224X224 as mentioned before and then to convert them to a Tensor. The remaining 5 transformations I had randomly select %10 of the image data for each

transform in order to better variate the data sets and make training more difficult and consequently, more robust. These transforms were as follows:

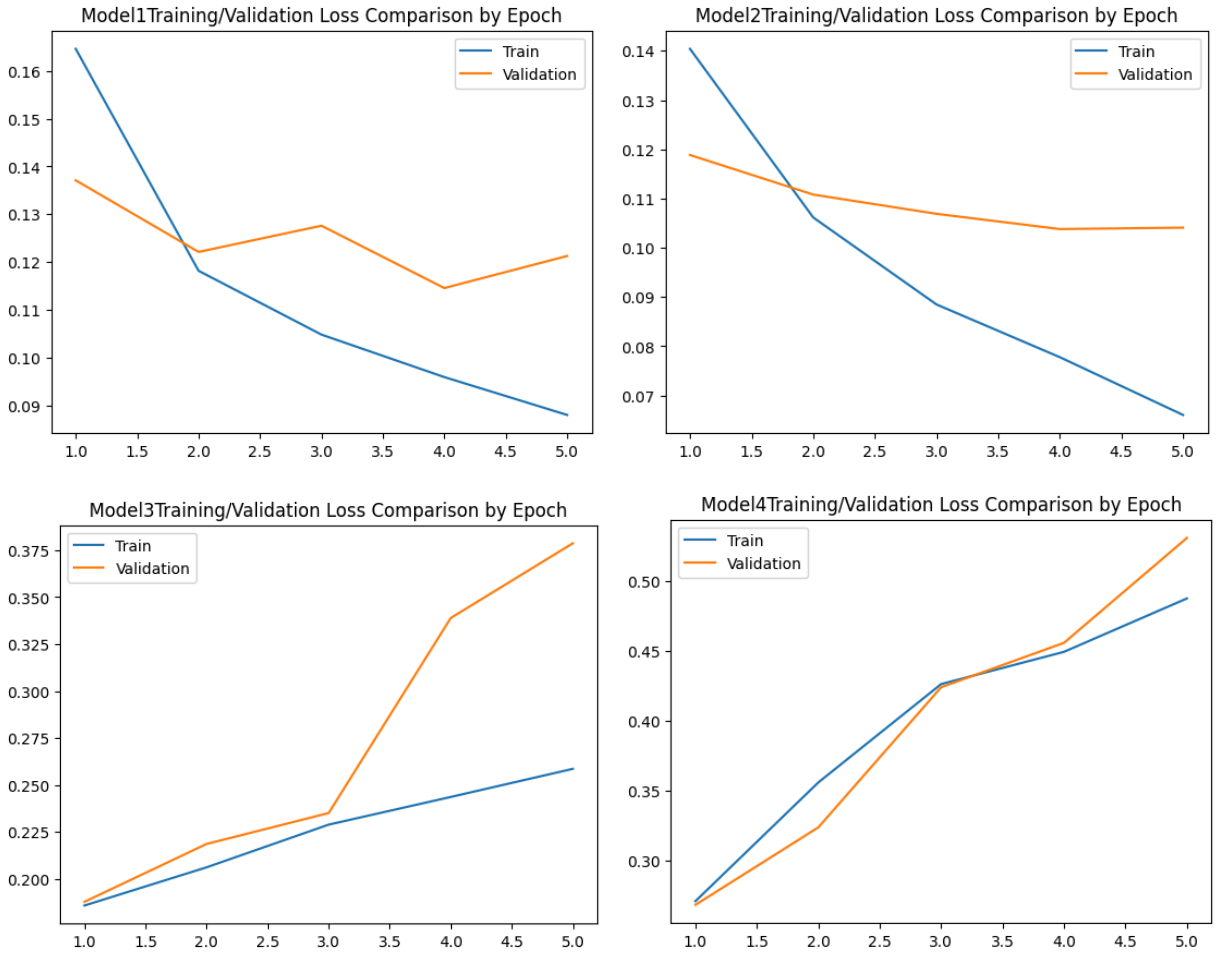
- transforms.RandomAutocontrast(0.1)
- transforms.RandomAdjustSharpness(0.1)
- transforms.RandomErasing(0.1)
- transforms.RandomHorizontalFlip(0.1)
- transforms.RandomInvert(0.1)

MODELING:

With this I was now prepared to begin training RCNN Models of which I select 4 from PyTorch's library those models being:

- Fasterrcnn_resnet50_fpn = Model1
- Fasterrcnn_resnet50_fpn_v2 = Model2
- Fasterrcnn_mobilenet_v3_large_fpn = Model3
- Fasterrcnn_mobilenet_v3_large_320_fpn = Model4

I selected these models because their Neural Network architectures came pre-built for my intentions which was to classify fractures and detect bounding boxes for said classifications. Therefore I selected all 4 of them to try. This continues to hold validity as well considering that they were pre trained for unique purposes, EX: the mobilenet faster RCNN networks were designed to be deployed on lighter hardware, like a mobile device, where as the first 2 are definitely heavier and meant for doing deeper image analysis comparatively and this is definitely marked in their scores between the train and validation sets. As shown here:



My ultimate decision, since this was meant to be deployed in a hospital-like circumstance, I selected Model2 due to its slightly greater precision at %92 accuracy +/- %1. Compared to model1. With that said, Model 1 was more reactive to the variance in the data as can be seen by the volatility in the Validation's performance. Which could make it a safer candidate as this could be indicative of not over-fitting.

INSIGHTS:

This was a very enjoyable model, really only made difficult by a lack of computing resources. Neural Networks, or deep learning models have a fascinating process in how they compute data and turn that into a usable solution, in this case a model with %92 efficacy and classifying arm breaks. This however is not sufficient. We would not accept a doctor operating on %92 and neither should we give a computer a free pass.

The next steps in improving this model further would be examining the layers of the model further. There were already ample images of x-rays of every arm fracture category I could fathom. The only way to improve this model now would be a deep dive into the nodes, and figure out what happens in each layer of the model, decide which ones could stand to be

skipped to speed up the process, which ones need to be modified to improve model performance. These would be the steps that could take this model from its current ability to one of a truly professional standard ready for deployment.