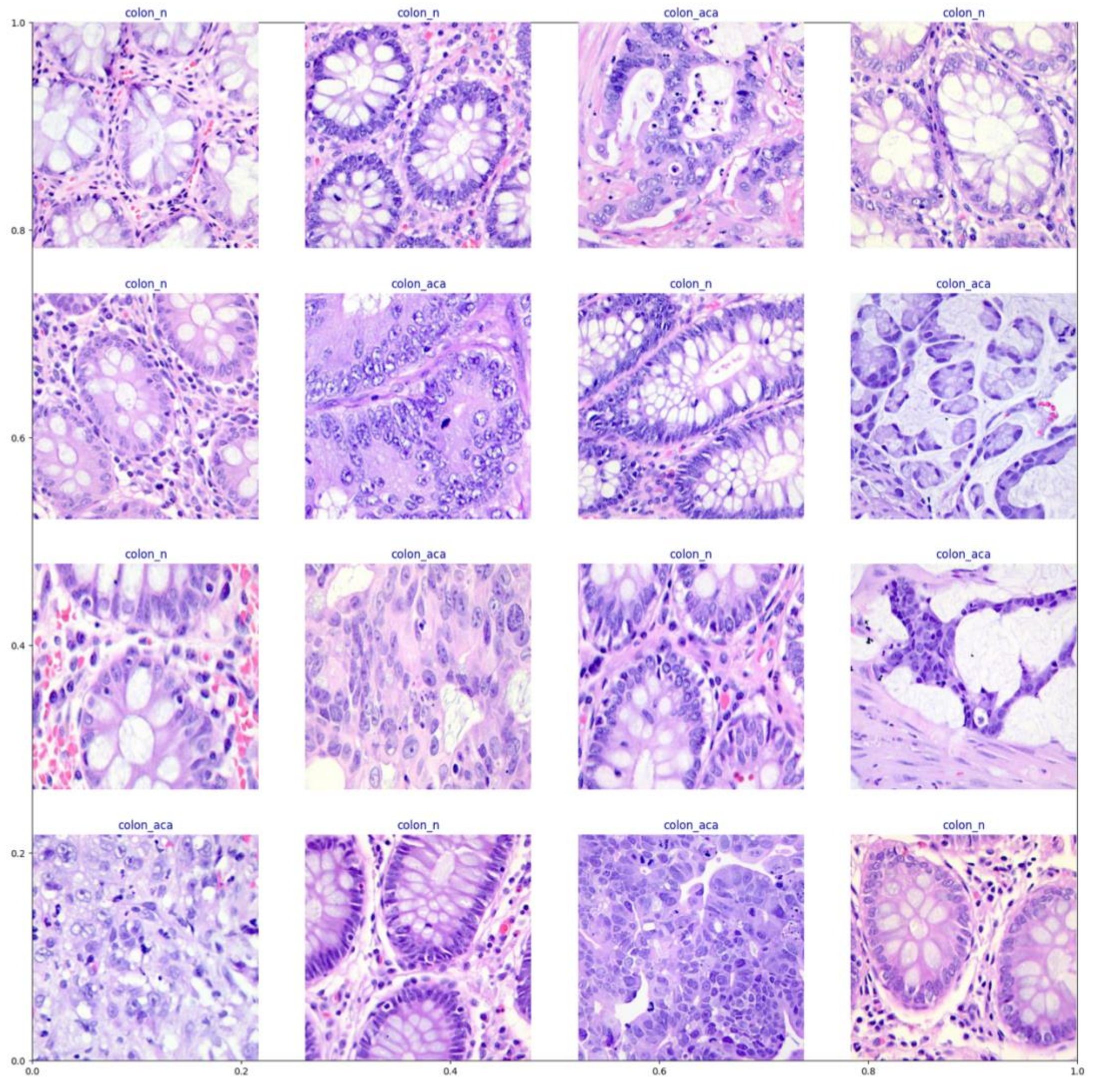
Colon Cancer Classification Using Neural Networks

Brandon Ling

December 23, 2024

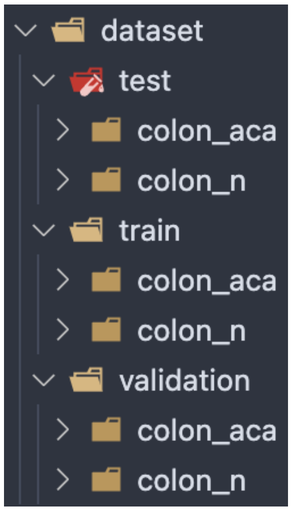
Overview

Currently, diagnostic measures diagnosing colon cancer involves a relatively simple procedure called a colonoscopy. While physicians are generally able to extract potentially cancerous polyps, they cannot tell simply by looking at them whether the polyps are cancerous without sending them to the lab for further analysis. It is in the lab where errors can be made when determining the cancerous nature of polyps, since certain polyps may be mistakenly overlooked. In this regard, machine learning holds immense promise for improving the screening efforts of physicians to prevent colorectal cancer cases and improve early-detection efforts. By training a neural network on existing lab samples of colon tissue, we can hope to accurately identify polyp samples (which are simply extensions of tissue) when they reach the lab. The below graphic contains 16 labelled image samples that were used in the training of the model. "colon-aca" and "colon-n" represent colon adenocarcinoma and benign colon tissue, respectively.



Data

When I began working on this project in September 2024, I was just beginning to deeply understand how neural networks worked. I spent my time learning about everything from feed forward and convolutional to long short-term memory neural networks to the modern transformer architectures that enable large language models to function, but I wanted to put that into action. To do that, I needed to find myself a dataset.


```
dataset
  test
    colon_aca
    colon_n
  train
    colon_aca
    colon_n
  validation
    colon_aca
    colon_n
```

Of course, the first place I looked in search of a meaningful dataset was Kaggle. Eventually, I stumbled upon the one used in this project, a collection of high-quality colon and lung samples. This dataset stuck out to me for a few main reasons. It was relatively large with 10 000 images of colon samples alone (examples shown above), it represented a potential solution of using AI to enable quicker diagnosis, and it was processed in a consistent manner. This enabled me to have to do very little data processing myself, aside from organizing my broader data directory into test, train, and validation sub-directories, as shown to the left. The approximate split that I used to separate the images into these distinct categories was 80/10/10 for training data, test data, and validation data, respectively.

Design

Choosing model architectures is often self-revealing, in that the optimal neural network to use is usually apparent before even testing them. In this case, I thought it was clear that I should train a convolutional neural network (CNN), given that this type of architecture is centered around image classification and computer vision applications. As such, that is where I began. I started with a simple design involving four CNN layers with ReLU activation layers, as well as 2D max pooling layers of kernel size two to down-sample the neuron outputs into increasingly smaller vector spaces. Essentially, this was the basis of the two main blocks of the CNN model. Finally, the model utilized a classification layer that first flattened the input into a one-dimensional tensor, then applied a fully connected layer followed by a ReLU activation, and then applied a fully connected layer followed by a SoftMax activation. To access these layers, I utilized the PyTorch library, which I chose over TensorFlow due to its broader applications in research.

I also needed to be able to feed the image data into the neural network. As such, it was necessary to apply relevant transformations to it. For the CNN model, I applied a resize to 180 x 180, a random horizontal flip, a random rotation by 10 degrees, a transformation to tensor, and finally a normalization. However, after setting up a basic training loop and running the model script, I found a couple things. First, the model had extremely long training times given it was working with large image data. Even after adjusting the number of hidden layers, the number of epochs, as well as the resize that I applied, the training was still quite computed intensive. Furthermore, I observed that as I tweaked hyperparameters to reduce the training time, the validation and test accuracy were at the point where it seemed that the model was not learning.

So, I quickly moved on to another model design option. I wanted to finetune the VGG16 and ResNet models, which are pretrained and very powerful image classification models. After setting up my Python scripts and loading the relevant layers of these models, I ran into the same issue with the CNN model, where the compute times were too long, and the accuracy was not to my expectations. When reading an article though, I learned about the idea of using pre-trained models as simple feature extractors, instead of as classifiers, and then applying a different classifier on them.

This led me to my final model design: using VGG16/ResNet models as feature extractors and then applying a random forest model to classify these features to be able to identify the image as either colon cancer or not. For both models, the only transformations that were necessary to the images were resizing to 224 x 224, transforming them to tensors, and normalizing the data. By far, this strategy led to the best diagnostic precision and accuracy.

Results

As previously mentioned, the convolutional neural network (CNN) based model that I aimed to train from scratch did not perform up to expectations, both in terms of training speed and model accuracy. While I was able to once achieve 95.7% test accuracy, it was impossible for me to recreate this in future attempts using any hyperparameters. Consistently, using the SGD optimizer (also trying the Adam optimizer) and by varying batch size, hidden layers, pooling kernel size, and more, I ended up getting around 55.6% accuracy. This abysmal accuracy is indicative of pure guessing, and the fact that it is greater than fifty can likely be attributed to imbalances in the data. Or perhaps it was an issue with the GPU cluster I was using to train these models. In any case, this was an extremely confusing phenomenon for me.

The VGG16 classifier with a random forest model achieved an accuracy of 99.0%, which was already quite impressive for me. However, because diagnosis requires the utmost precision, I was happy to see that the ResNet classifier with the random forest model achieved an accuracy of near 100%, at around 99.8%. In the training of the models using ResNet/VGG16 as classifiers, I utilized a batch size of 32, 100 estimators for the random forest, and removed the last layer of the respective feature extractor (which would usually classify the image).

Deployment

Once I completed my analysis of the various model architectures, I wanted to enable others to take advantage of the capabilities of my model. Up until this point, I had simply been training the models using a loop that I had created, printing the accuracy of the model on test and validation sets, and letting the weights disappear after the training script finished running. But, to deploy the model, I would need its weights, and thus I added a line to the ResNet model training script, which had the best results, to export the weights to the appropriate directory as a .pth file, which contains a serialized PyTorch state dictionary for the model.

Then, I setup an inference file using the Gradio package which enabled me to easily host my model online. However, I ran into an additional issue during this phase. While I had model weights in my possession, these were the random forest weights trained on the features extracted from the pretrained model, which in turn received transformed data themselves. As such, I created a feature extraction function that could take an image as input and apply the same transformations that were used to input them into the pretrained model, and then apply the pretrained model as well to get extract features. Only then was the data ready to be fed into the model.

Upon testing the Gradio web app that now could run inference, everything worked as expected for the test colon samples that I uploaded. However, when I uploaded other objects that were not cell samples, the model would still produce a determination of whether it believed the image represented colon cancer or not, even though it clearly did not fall into either of the two possible classes. To solve this issue, I setup a minimum threshold for determination at 0.75, making it so that the model would have to be 75% confident that an image was or was not colon cancer, and if not, it would output an indeterminate message. Since the model was usually 90% confident on actual colon samples, this seemed to fix the problem for the most part.

Finally, to make the model accessible to the public and have it run not just on localhost, I opened a space on HuggingFace, installed the necessary dependencies, and uploaded by repository. It can be found online here.

Conclusion

This has been a rewarding project that has allowed me to explore the fundamentals of finding a viable dataset, designing and training an AI model, and deploying it for real life usage.