2020

Software Engineering around Machine Learning

MASTER OF SOFTWARE DEVELOPMENT CAPSTONE REPORT

ALEX NELSON

UNIVERSITY OF UTAH STUDENT 2020

Alex Nelson Master of Software Development Capstone Report

Software Engineering around Machine Learning

Table of Contents

Alex Nelson Master of Software Development Capstone Report

Software Engineering around Machine Learning

Introduction:

Purpose: Provide a functioning API connected to a classifier with the purpose of classifying images of cells to the correct cell type, even with varied RNA strands found within the cells.

Problem this project solves: This project provides a classifier that can be accessed via a "get" API request. This will allow users to classify future images of cells whether they are taken by the user, or obtained from the Salt Lake City company Recursion. Currently Recursion has provided a public dataset [RxRx1] of cell images with varied RNA strands within the cells. This dataset release was a one-time release in June 2019 and it is not maintained by Recursion. It also has not been updated since the initial release. Although Recursion has released a RxRx2 dataset in August 2020, the RxRx2 dataset differs as it only has cells of type HUVEC contained in RxRx2, whereas RxRx1 has four cell types contained. Recursion does not currently provide a classifier along with their datasets, making any type of add on research tedious as images must be identified individually with no help from Recursion.

Background:

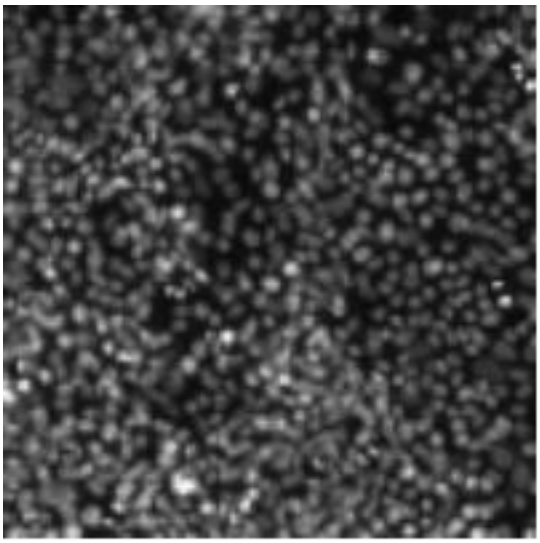
The RxRx1 dataset provided by Recursion contains 750,060 images of 125,010 different cells. There are four types of cells contained in these images:

1. HEPG2 – A cancerous liver cell
2. HUVEC – A cell from the umbilical cord
3. RPE – A cell found in the eye near the retina with the purpose of nourishing the retina
4. U2OS – A cell found in bone tissue

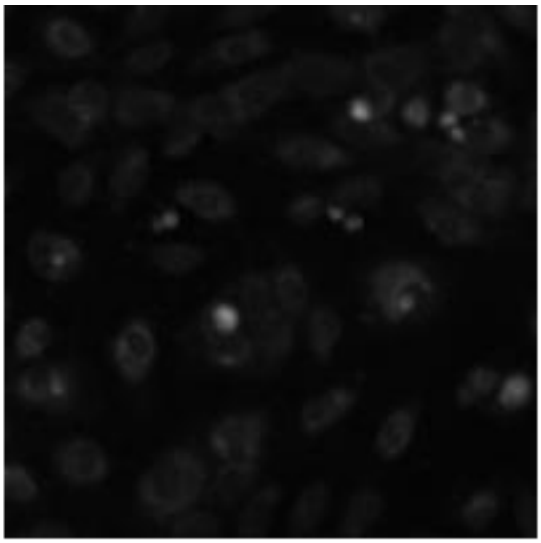
In addition to the four types of cells, the images contained in this dataset have different perturbations of RNA injected into the cells. There are 1,138 different types of RNA strands found within these images. These RNA strands are labeled by siRNA numbers that were assigned by the company that provided Recursion with these RNA strands, ThermoFisher. There were two distinct sites (s1 or s2) where these cell images were captured, and six individual wells (labeled as w1 through w6). These individual wells were referenced by Recursion as Stains or Channels. The six stains were:

1. Hoechst
2. ConA
3. Phalloidin
4. Syto14
5. MitoTracker
6. WGA

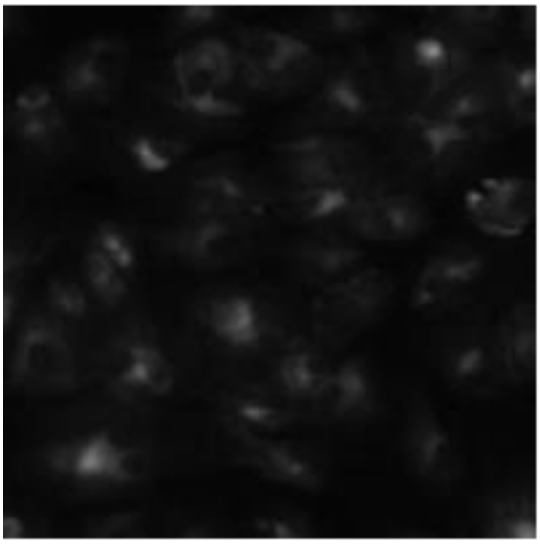
Because of this there are 6 unique images for each of the cells in the dataset. Here is one example of each of the different types of cells:



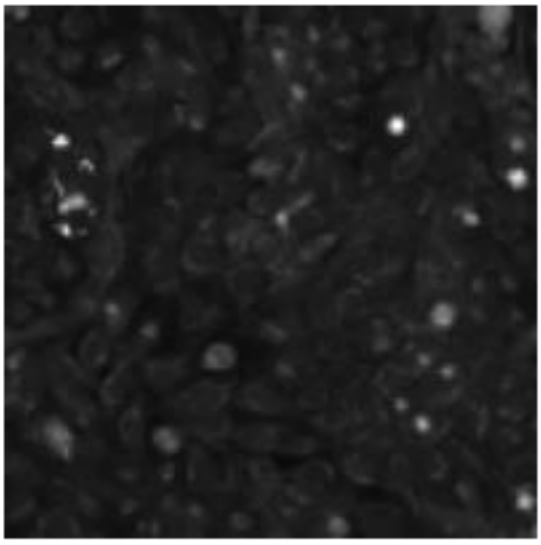
HEPG2 Cell



HUVEC Cell



RPE Cell



U2OS Cell

Recursion initially made this dataset available for a Kagle competition in 2019, and as such the images are already split into a training and testing set of images. The breakdown is as follows:

Training Set –

103,464 images of 17,244 different HEPG2 cells
236,052 images of 39,342 different HUVEC cells
103,476 images of 17,246 different RPE cells
44,352 images of 7,392 different U2OS cells
For a total of 487,848 images of 81,308 different cells

Testing Set –

56,100 images of 9,350 different HEPG2 cells
118,248 images of 19,708 different HUVEC cells
58,944 images of 9,824 different RPE cells
29,424 images of 4,904 different U2OS cells
For a total of 262,716 images of 43,786 different cells

Currently there is not a public classifier for cell images to be able to take in future images and classify them to the cell types found above. It was an exciting challenge to design a classifier. A description of the model can be found in the Solution Description. Once a model was created, an API was made to obtain predictions from the model. Unfortunately, I was unable to complete all my goals as I was unable to successfully host the API publicly. Even though I was unable to host the API publicly, I have included a small section on attempts to make the API available to the general public.

Solution Description:

The solution can be broken down into 3 main portions:

1. Building a model
2. Training a model
3. Building an API

Building a model-

Two different types of models were considered as they have been found to be very common for image classification, Convolution Neural Networks and Generative Adversarial Networks.

In a **Convolution Neural Network** classifier, where an input (specifically in this case an image), is passed through multiple sets of virtual neurons. These neurons have different mathematical equations that manipulate the input data and pass it on to the next neuron. Rather than relying on the user to decide the numbers of the equations inside the neurons, the neurons numbers inside the functions or “weights” will be decided based off accuracy on a set

of training images. This is the beauty of machine learning. It was not necessary for me to have a prior deep understanding of the biology behind the different cell types and RNA strands to allow a computer to find mathematical "features" within an image that can help predict the cell type of images correctly. These features may not be easily understood by the human mind as they are discovered by what appear to be random numbers. A user can give the CNN model instructions on how many neurons and what type of neurons to create, but is not required to manually adjust the weights of every individual neuron. This allows the user to add many neurons as the cost for the user is relatively low. During the training process, images with known labels, are passed into the CNN model, and the weights are adjusted so that the model would correctly predict the label of these images.

In a **Generative Adversarial Network**, there are two main components. Although GAN's are not exclusive to image classification, the examples following use images as an image classifier was the desire of this project. The first of the main components is a generator. A generator "generates" or creates images of the right size. The second component is a discriminator. The discriminator takes in an image and outputs whether this image was a "fake" image created by the generator, or a "real" image of the desired cell. As a GAN is trained, the generator takes the results of the discriminator to improve the quality of the image output with the desire of tricking the discriminator to think the created image is real. At first, the generator will produce random noisy images, that look nothing like a desired image. But quickly, the generator will start to produce more believable images. The accuracy of the discriminator to correctly judge whether the image was "fake" or "real" fluctuates related to the accuracy of the generator being able to trick the discriminator. As the generator improves the accuracy of the discriminator decreases, but the discriminator "weights" are not fixed and they change as more and more images both real and fake are fed into it. As the discriminator improves, the generator now has new input that will help improve the quality of the images. This leads to a competing accuracy between the generator and discriminator.

My model: I knew the images coming into my model would not just need a binary prediction of cell or no cell. My model needed to be able to predict which one of the four types of cells an image was. I decided a Convolution Neural Network classifier was an excellent model to use as I knew it was a great model for a multi-class labeling, as they can produce multiple predictions at once. Whereas, to do multi-class predicting with a Generative Adversarial Network, I would need to build four different generators and discriminators. I recognize that with this said GAN's are still a fantastic option for multi-class prediction, and in the future, I may repeat the process with a GAN. However, due to the goals of the project to be to build an API on top of a machine learning model, I decided to focus on the Convolution Neural Network only for the sake of this project.

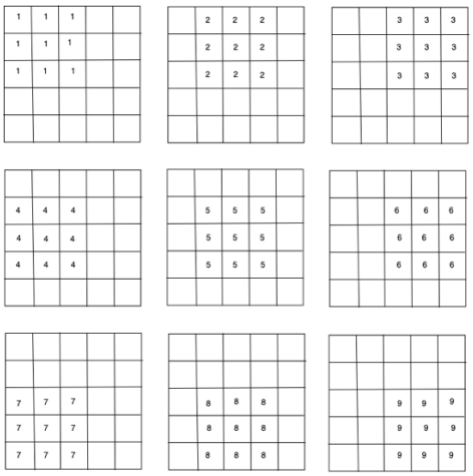
Two of the most common types of a CNN are functional and sequential, especially when dealing with the common keras model building library. When a **functional model** is used, data can travel through a neuron multiple times. This can lead to a model where it is hard to define "layers", as a neuron that is used at the beginning of the process could also be used at the end of the machine learning process. When a **sequential model** is used, data can only travel

through a neuron once. This makes it easy to define “layers” in a model, as data can only travel in one direction. Data received by the first layer of neurons can only be received from the initial input. Data received by the second layer of neurons can only be received from the output of the first layer of neurons and so on for as many layers as the user sets up.

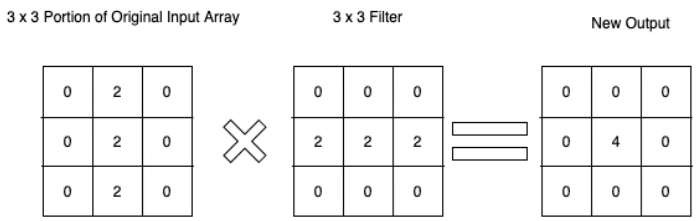
When building a multi-class labeling CNN, the final layer inside the model needs to match up with the possible outcomes. For instance, because I was predicting between four different types of cells, my last layer of the model produces four numbers. Each of these numbers is a percentage, the likelihood that a specific image could be categorized to the related cell. The sum of these percentages equals 100%, and the highest individual percentage is the prediction. To successfully produce a multi-class prediction, a “final” layer of neurons must be achieved. Because of this “final” layer, a sequential CNN was chosen.

The next step to understanding a sequential Convolution Neural Net is to understand the different types of layers that can be applied inside the net to create the Neurons.

While trying to understand Convolution Neural Nets, it is important to understand what a **convolution layer** does. A convolution is application of a “filter” to some type input that results in a new output. If you had a 5 x 5 array (in my case an image) of data, and wanted to apply a 3 x 3 filter, you would be able to apply the 3 x 3 array in 9 different ways.

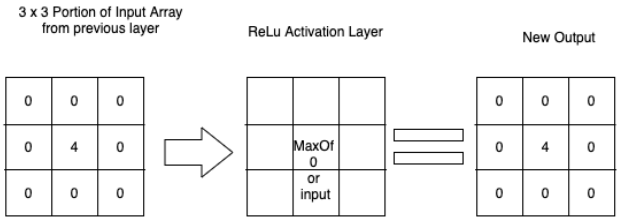


When you apply a filter, you are multiplying the input by the numbers inside the filter to get a new array that is the size of the original input, with changed numbers.



The goal of these changed numbers is to identify “features” of an input that are indicative of what matters within the array as far as the final classification. When a model is trained, different filters are applied and tested until the model filter numbers or “weights” are optimized to have found the best accuracy on the training inputs.

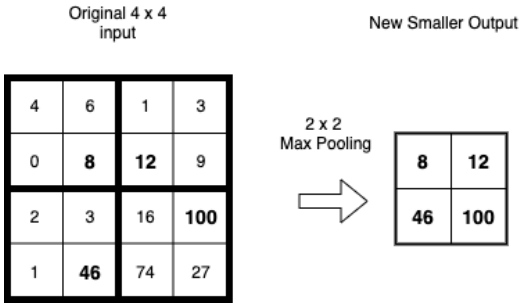
Typically, after a convolution layer, an **activation layer** is applied. This activation layer is a simple mathematic function that is applied to the input to simplify the outputs from the convolution layer. The most common of these is a **ReLu activation layer** or rectified linear unit. This ReLu layer guarantees there will be no negative numbers in the output. This is achieved with a simple application of find the max between 0 or the input. Common alternatives not used are a sigmoid activation layer or a tanh activation layer.



The only other type of activation layer used with my model, was a **softmax activation layer**. A softmax is an exponential function that when applied to an input between 1 and 0 will produce an output that is closer to 1 or 0 than the original output. A softmax application layer is applied at the end of a model to produce probabilities of a specific result. This is essential for a multi-class labeling project as it is necessary to make all the probabilities of the final layer to equal 1.0 or 100% thus making it likely that there is a clear front runner prediction.

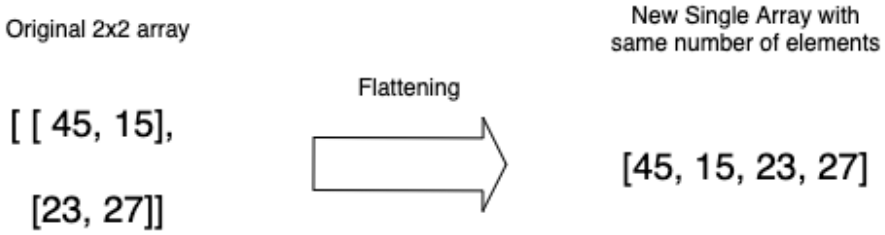
When dealing with convolution layers it is important to understand how much work applying filters can be for the machines tasked with building a model. For a simple example, as stated above, let us assume a single neuron has a 3x3 filter that will be applied to a 5x5 input 9 different ways. This leads to 81 different numbers being output (9 numbers inside the filter multiplied by 9 times of application). Even though the final output of this single neuron is not 81 numbers big, a computer must hold all 81 different numbers while doing its calculations to produce a new 5 x 5 output. In contrast, a 3x3 filter that is applied to a 4x4 input can only be done 4 different ways. This leads to 36 different numbers being output (9 numbers inside the filter multiplied by 4 times of application). Although 36 data points compared to 81 datapoints is minimal in modern computing, you can imagine how much data a model has to deal with as the size of the input grows. All of the Recursion images used to build and test the model are 512x512 pixels. This leads to a massive amount of numbers tied to a single output from a single neuron. In addition to this, Convolution Neural Nets work best when multiple neurons are on each level of a sequential model. To help reduce the amount of storage needed within a model and increase the speed of building a model, max pooling was created.

max pooling is the process of taking a large input and reducing the size to a smaller output by saving the max number from a specified region. Here is a diagram of how the process looks:



The last type of layer used was a density layer. A **density layer** is an inner layer or final layer that takes in the input from all the neurons in one layer above it, and reduces it to a simple specified set of numbers. For example, on this project I wanted a layer at the end of my model that produced 4 numbers, the probability for each of Recursion's four types of cells. By applying a density layer at the end of my model, I was able to take in multiple inputs from multiple neurons in the layer above and produce four individual numbers based on those inputs. A density layer could also be used in the middle of a model, but the purpose is to take multiple inputs and produce one output.

The final concept used in my model was **flattening**. Because the goal of my project was to produce a single array of four numbers as a predicting classifier, somewhere in my model I needed to transition from a 2d array to a single array. **Flattening** is the process of turning a multi-dimensional array into a single array with the same number of elements. For example, flattening a 2x2 input would produce a single array of 4 numbers as an output.



Here is how the final model was constructed:

1. A convolution layer of 32 3x3 filters with “relu” activation and max pooling

2. An additional convolution layer of 64 3x3 filters with “relu” activation and max pooling

   (outputs are flattened between levels 2 and 3)

3. A density layer that produces an array of 128 numbers (2 for each of the above 64) with “relu” activation applied

4. A final density layer producing an array of 4 numbers with “softmax” activation so the 4 numbers summed together equal 1 and relate to the 4 different types of cells.

A visual representation of the model can be found on the very last page of this report.

The model was built using the TensorFlow library specifically with the sub-library keras in python.

Training the model-

Training a CNN model involves using a computer to set the weights or numbers of each of the different virtual neurons on each level of the model. Specific to my model, here are the weights that needed to be set:

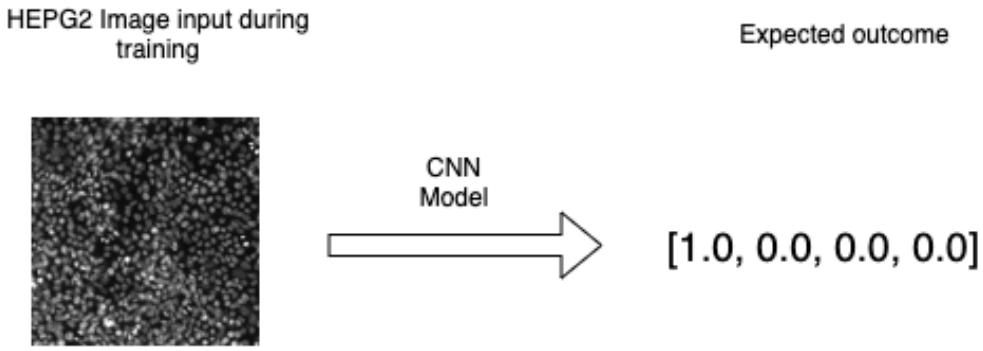
1. 32 different filters (3x3 2d arrays) on the **first** layer

2. 64 different filters (3x3 2d arrays) on the **second** layer

3. One function for the density section inside the **third** layer that takes in 64 arrays from the second layer and outputs a single array of 128 numbers

4. One function for the density section inside the **fourth** layer that takes in a single array of 128 numbers and outputs an array of 4 numbers between 0 and 1.

For example, below is a known image of an HEPG2 cell. As the model is set up to produce a final array of four numbers, the expected outcome for this image is also an array of four numbers.



Multiple training images are needed to train the model, and every image needs an associated array of four numbers as the expected outcome.

To do this on a TensorFlow sequential CNN model, I was able to call the method “predict” directly on my python model object with the inputs of the training images and the known expected outcomes associated with each image. My machine changed the weights inside each neuron of the model, measured the actual outcome against the expected outcome, and adjusted the weights accordingly.

The optimizer:

Having a machine learning model find the best possible weights can seem like magic. There are three parts to finding the correct weights within a model:

1. What numbers to start the weights at.

2. When and how often to evaluate the weights and then start the process again.

3. When to stop adjusting the weights

To determine how to handle these three parts, a model relies on an optimizer. An **optimizer** is a pre-programmed method of finding the best weights inside a neural net based on training data.

To understand optimizers, it is important to understand what happens during training and what a learning rate is. During training, a weight is changed, the outcome is recalculated, evaluated against the expected outcome, and logged for future use. This process happens numerous times during training. The amount the weights are changed by is directly related to the learning rate. **A learning rate** is a number between 0 and 1, and is the proportion that weights are updated during the training process. A larger learning rate leads to a bigger change,

and a smaller rate leads to smaller change and thus more time spent training a model. At a set time, the logged differences between actual and expected outcomes are used to obtain an average or mean to determine a new learning rate.

There are many different optimizers, and researching them to discover the best one can be daunting. Luckily, most optimizers have been created with specific types of machine learning in mind, and an optimizer called "Adam" was created specifically for image classification. The **Adam optimizer** derived its name from the term adaptive moment estimation. Previous common optimizers would only change the learning rates after going through an entire predetermined amount of data and evaluating each datapoint in the set equally, or change the learning rate based on the most recent data. The Adam optimizer attempted to speed this up by taking into account two averages. The first average was created by a predetermined amount of data where all datapoints are evaluated equally. The second average is heavily based on the most recent data. This came about from an attempt to combine two previous optimizers, the Adaptive Gradient Algorithm, and the Root Mean Square Propagation. The Adam optimizer is commonly regarded as the most efficient optimizer for multi-label image classification.

Back to the model:

Due to the large number of images, I found it best to reshape the images into 2d arrays and assign expected outcomes outside of the model. This saved the model conversion time of converting each image into a 2d array. Using the python library Pillow, I was able to reshape the images. Pillow was a very effective choice as it was easy to transition from a reshaped array to an image that was easy to display within a Jupyter Notebook, making spot testing easy and accessible. Because this was done outside of the model, it was also best to store the reshaped data and expected outputs so the reshape only needed to happen once. After researching available libraries and tools, the python library Pickle was discovered and used for storage. This made it very quick to upload already reshaped arrays of the images to be fed into the CNN model. Training the model has been easy, although time consuming as TensorFlow allows for weights to be trained multiple times and saved after each time.

Although the initial plan was to train the model using the free resources at AWS, I quickly found that to do so I would have to use access ec2instances of AWS that are charged based on hours run and number of GB's used. For this reason, I decided to continue training my model on my local machine chunks at a time. I trained on the stain well w2 first, as this was the most vibrant of the images, and seemed to have the most contrast. I made the decision that for my model, I would train on an equal amount of each cell type. That means that my model has been currently trained on 29,568 of the training images up to this point. It was the intention of the project to train more, but with time constraints I felt 86% accuracy was a strong enough to continue on to the next stage of the project.

Building an API-

An API is a set of code that enables data transmission between one software product and another. In this case, the two points of software that I was looking to connect was a User with my CNN model. There were essentially three tasks that the API had to handle:

1. Receiving an image and a request for a prediction based on that image from a user

2. Run the image through the model and produce a prediction

3. Send the prediction back to the user

Receiving an image from a user: Building an API that could receive an image was not an intuitive process for me. After finding the Flask python library, (more information found below), I knew I wanted any inputs from a user to work within a python script. The opencv library made this simple as it enables a user to encode either a .png or .jpeg image file and produce a string version of this data. It is very simple for a user to send this string information within a "get" request and have it properly received on the API end. To actually receive these "get" requests, the request sub-library from the Flask python library allowed me to create a route for the API, and a method inside a specific tag on the api address "/api/predict" that would run the desired python script. Inside this API method, the image is decoded also using the opencv library and then turned into a 2d array that is ready to be fed to the trained model to produce a prediction.

Producing a prediction: Although it would have been nice to leave the prediction as simple as the predict method within the TensorFlow library, there were more things I needed to consider. As the desire for this API was to make it available to more users than just myself, I knew it was not good practice to leave my trained model on my personal machine. Having a central non-local location for storage of the model also allows me to update the model at any time to ensure users will have the most recent version of the model. This allows for improvement of the model through future training. I utilized the storage made available by AWS s3 to store my model remotely on a machine located in Oregon (AWS us-west-region-2). Within the API script, using the boto3 method, the machine running the python script will pull the necessary files that make up the model and the trained weights, and then use the TensorFlow library to load the model so it is ready to make predictions. This process of accessing the s3 files and loading the model is done only once at the very start of the python script, rather than every time a request is made. This allows the user to make as many predictions as the like consecutively with only having to load the model once.

Security note: Allowing users to access my personal s3 is dangerous and was not taken lightly. To avoid unwanted changes or corruption to my CNN model or access to unrelated files, I have set up an IAM role with my AWS account that has "read-only" access limited to only the s3 portion of a single s3 bucket that contains the TensorFlow model.

Once a get request is received and decoded inside the API, the opencv image object is reshaped into 2d arrays that fit the format needed by the TensorFlow model. Once that is complete, the reshaped image is passed to the model, and a prediction is produced using the predict method provided by TensorFlow and the sub-library keras.

Sending the prediction back to the user: When the prediction is produced it comes as an array of four numbers. The python script will parse the array and format it into a string with labels so it is easy to read for the user. This string format is then sent back to the user with a 200 request.

Results:

The final results of this project include a working model that correctly predicts 512x512 pixels of cells between HEPG2, HUVEC, RPE, and U2OS cells at an 86% accuracy. The project also includes a successful python script that loads the model from remote AWS s3 storage and creates a local API for the user.

Notes on library management: Many libraries that have been mentioned in this report were necessary for building and testing the model, such as Pillow, pickle, and numerous TensorFlow sub-libraries such as keras. These libraries, although necessary for the creation of the model, are not necessary for a user's machine to run the python script to locally create and run the API. Here is a list of the needed python libraries needed on the local machine to run the python script:

1. sys and os
2. boto3
3. Flask and request from flask
4. TensorFlow
5. numpy
6. cv2 from opencv-python

If a user does not want to deal with library management, a virtual environment with these six libraries is provided within the s3 location that is available to users with the restricted role mentioned in the Solution Description–Building the API–Producing the Prediction section of this report.

Python version 3.0 or greater is an absolute requirement to run this python script. The python script and request instructions can be found on the public GitHub repository:

https://github.com/alexnel24/RecursionPredictingAPI

Looking back and future changes:

Although I am pleased with the outcome of this project, there are a few things I would try differently if I did this project again. Here are three things I would like to have done differently:

1. Similar process with a GAN model instead of a CNN model.

2. Hosting the API remotely in a public manner so a user does not have to host the API and use it locally

3. Changes to allow images that are not 512x512 pixels to allow for a wider range of images.

Resources:

I would like to give a big thank you to Dr. Varun Shankar who was my lead mentor on this project, and Dr. Ben Jones who introduced the concept of CNN models in his Data Analytics & Visualization course. Below is a list of the libraries used on this project and the documentations for all. Also included are a few websites that were helpful in learning about machine learning and CNN models:

A. Recursion rxrx1 dataset - https://www.rxrx.ai
B. TensorFlow - https://www.tensorflow.org/community/contribute/docs_style
C. opencv-python - https://docs.opencv.org/master/d6/d00/tutorial_py_root.html
D. Flask - https://flask.palletsprojects.com/en/1.1.x/
E. Pickle - https://docs.python.org/3/library/pickle.html
F. Pillow - https://pillow.readthedocs.io/en/stable/
G. Boto3 - https://boto3.amazonaws.com/v1/documentation/api/latest/index.html
H. NumPy - https://numpy.org
I. Python - https://www.python.org
J. AWS - https://docs.aws.amazon.com/index.html
K. Researching CNN models - https://towardsdatascience.com
L. Additional Research on ml and optimizers - https://machinelearningmastery.com

On the final page is a visual representation of the latest CNN model

Visual Representation of the CNN Model: