Machine Learning Engineer Nanodegree

Capstone Project

Ron Budnar

September 2019

I. Definition

Project Overview

Note: This capstone project was developed for the Recursion Cellular image Classification Kaggle competition (“Recursion Cellular Image Classification,” n.d.-a). More info on the dataset and background can be found at their website (“RXRX,” n.d.-a).

Drug and medical treatment research are both time consuming and costly. In addition to delayed treatment timelines, research and development costs can be passed on to the consumer in the form of high prescription drug prices and expensive tests, making therapy a cost-prohibitive option. The drug discovery and development process - in particular, understanding how drugs interact with cells - is one aspect of the drug and medical research pipeline that can benefit from improvements in cost and speed.

The process of drug research and experimentation often requires cellular image analysis. Even though great care is taken to ensure technical and environmental consistency throughout the experimentation process, biologically irrelevant variables (known as batch effects) can materialize in results and be a source of technical noise in cellular image analysis. Simply put, factors unrelated to the drug experimentation process can influence and confound experimental results, delaying research completion and increasing associated costs. As many fields of biology across the spectrum are affected by this phenomena, machine-aided “disentanglement” from experimental batch effects could have widespread implications and potentially help accelerate the drug discovery and development process.

Problem Statement

In a perfect world, if one group of cells of a particular type receive a specific treatment under specific environmental conditions, we expect to observe the same or very similar results when the same treatment is applied to the same type of cells in the same environmental conditions. Unfortunately, this is not always the case - due to environmental or technical imprecision (such as differences in temperatures, reagent concentration, technician skills, etc.), the resulting data may display differences not attributable to treatment alone. This phenomena is referred to as “batch effects” and is one of the main challenges this project must overcome.

With this in mind, the main goal of this project is to develop a machine learning model that can accurately classify experimental treatments applied to various cellular samples from a set of cellular images taken after treatment application. In the context of this specific dataset, one of 1,108 small interfering RNAs (siRNAs) were applied to each sample of a series of experiments, causing changes in cellular morphology (i.e., how the cells appear under a microscope). Two distinct sets of images were captured for each sample across 51 experiments (experiments were conducted at different times) and comprise the dataset for this problem.

To solve this problem, this project will develop and train a convolutional neural network to predict the treatment applied to each of the samples. The solution to this problem should be robust to batch effects that may be present between (or even within) experiments.

Metrics

The overall performance and success of this model will be measured by the classification accuracy of the test data. The dataset is provided by the previously mentioned kaggle competition and does not include labels on the test data; consequently, test set accuracy is determined by submitting a CSV file to the kaggle competition site for accuracy measurement. Since the test data is unlabeled, validation loss and accuracy will be monitored during the development process as a sign of improvement.

II. Analysis

Data Exploration and Exploratory Visualization

The dataset for this project, available from ("Recursion Cellular Image Classification," n.d.-b), is a set of cellular images taken from 51 separate experiments (i.e., each experiment was completed at a different time) provided by Recursion Pharmaceuticals for a Kaggle competition. Each experiment includes exactly one cell type and there were four cell types used across the experiments: HUVEC (24 experiments), HEPG2 (11 experiments), RPE (11 experiments), and U2OS (5 experiments).

Each experiment was conducted in batches of four plates containing 384 wells (16 rows x 24 columns of wells, shown in fig.1) in which a sample of cells and their treatments (or controls) were applied. The same 30 positive controls (different than the 1,108 experimental treatments) were used on every plate along with one untreated well (a negative control); in addition, the outer rows and columns of each plate were not used for the experiments. Thus, 277 wells on each of the four plates were used for experimental treatment resulting in 1,108 different treatments for each experiment (each treatment was applied to exactly one well in each experiment). The use of the same positive (treatment) and negative (no treatment) control wells across plates and experiments can provide the researcher with some insight into experimental result consistency differences due to deviation in technician skill or environmental regulation and is a standard technique in biological experimentation. However, the control wells are not used in generating test data predictions for submission to the competition.

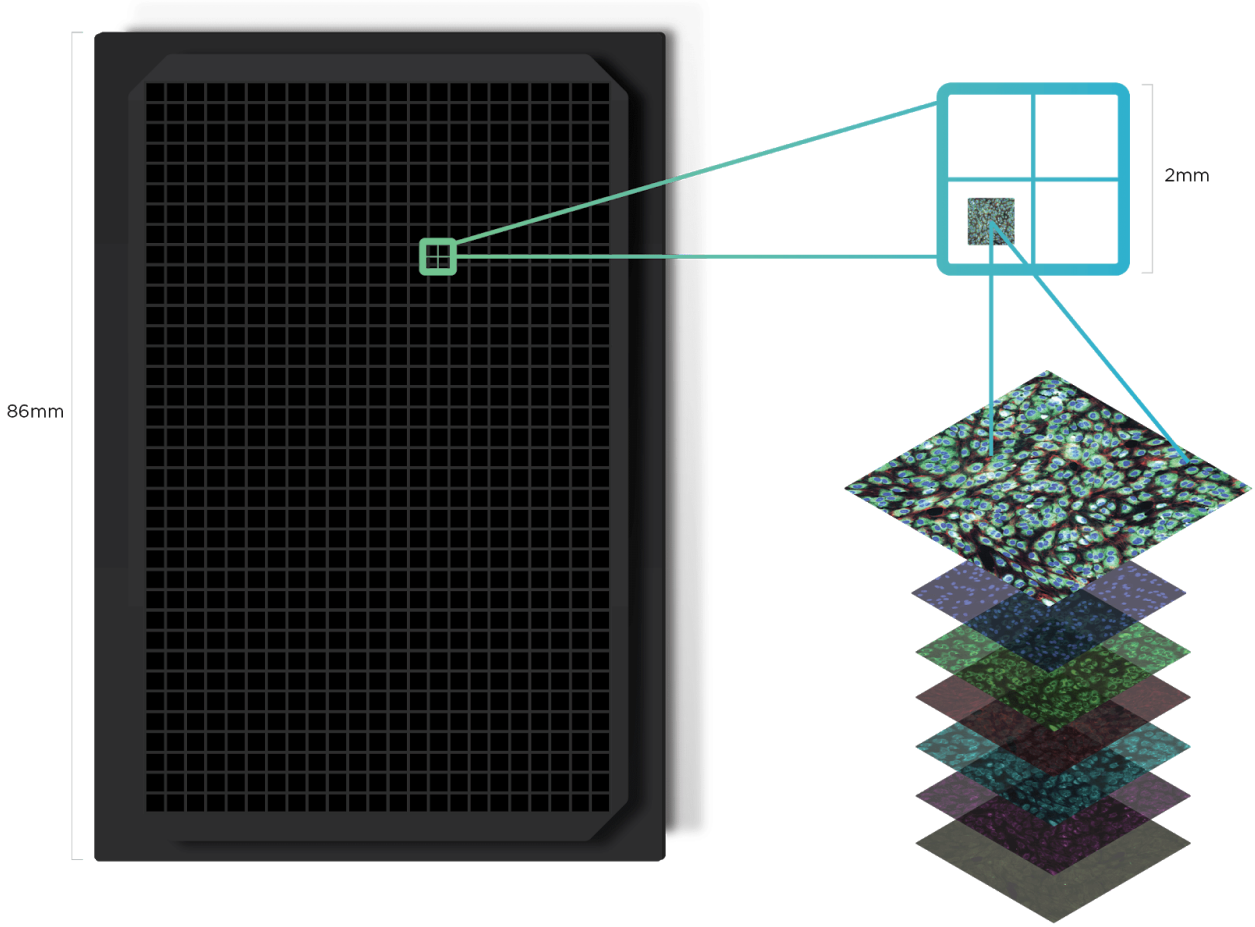

Figure 1. A 384-well plate used for cell samples. Samples (cells of a single type) are aliquoted into each well along with its corresponding treatment (siRNA). Image courtesy of https://www.rxrx.ai/#the-data

As a part of the experimentation process, the cells were stained using the “Cell Painting” protocol in which six different stains were applied to the cells. These stains adhere to different parts of the cell and allow special cell image capturing techniques to illuminate different structures of each cell; as such, six images were captured at each imaging site highlighting different structures on the same group of cells. Two non-overlapping sets of 512x512x3 images (in PNG format) were captured for each of the 1,108 samples in each experiment, resulting in twelve images captured for each sample. The experiments were pre-split into training and testing data for the competition, with 33 experiments designated as training data (16 HUVEC, 7 HEPG2, 7 RPE, and 3 U2OS) and the remaining 18 experiments designated as test data (8 HUVEC, 4 HEPG2, 4 RPE, and 2 U2OS). The split of test/train data is depicted in figure 2.

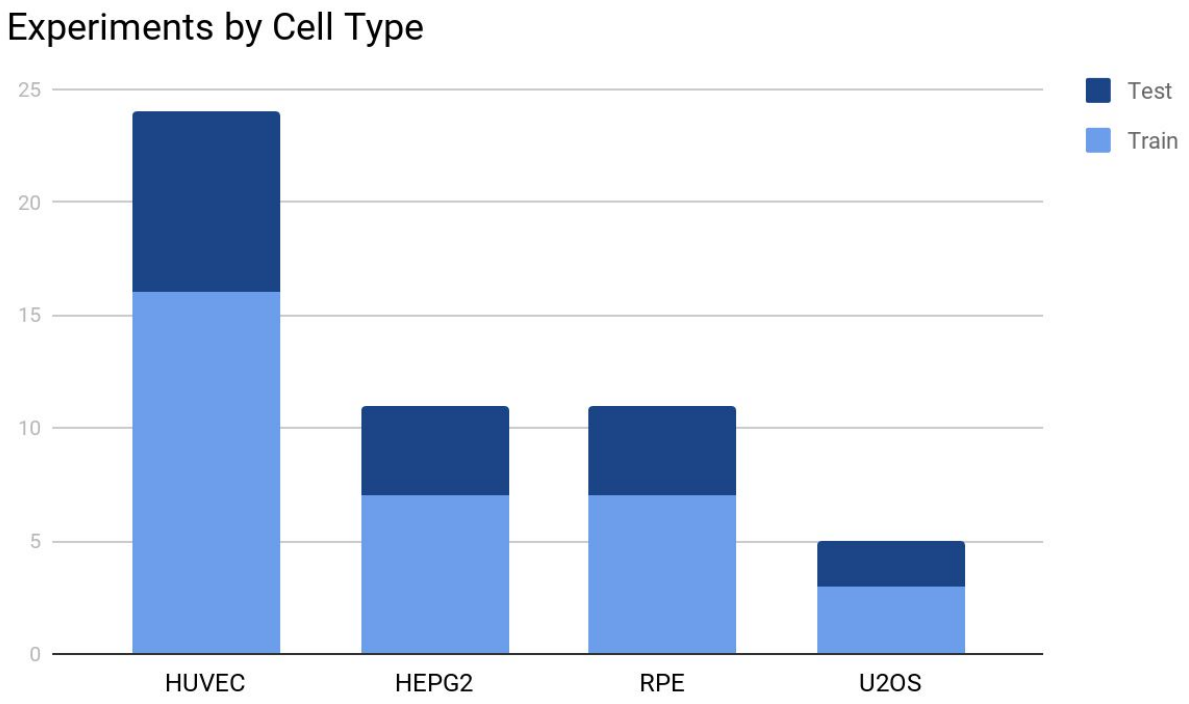

Figure 2. Number of experiments by their cell type. A total of 51 experiments exist in this dataset.

Overall, there were approximately 753,000 images between test, train, and control images. Although this may initially seem to be a large number of images, the high number of classes (1,108) resulted in a limited number of training samples for each classification. There were 33 training experiments with two sets of images for each classification for a result of 66 samples for each classification. However, as previously mentioned, the same treatment may not have the same morphological effects across the varying cell types and a model may have difficulty correctly identifying these variations. It is possible that separate models may need to be developed for each of the four individual cell types, further reducing the number of training samples per classification. In this case, the number of training samples would be double the number of experiments for each cell type - a total of 32 for HUVEC, 14 for HEPG2, 14 for RPE, and 6 for U2OS. Figure 3 depicts one sample (set of six images) of each of the four cell types with siRNA # 1 applied.

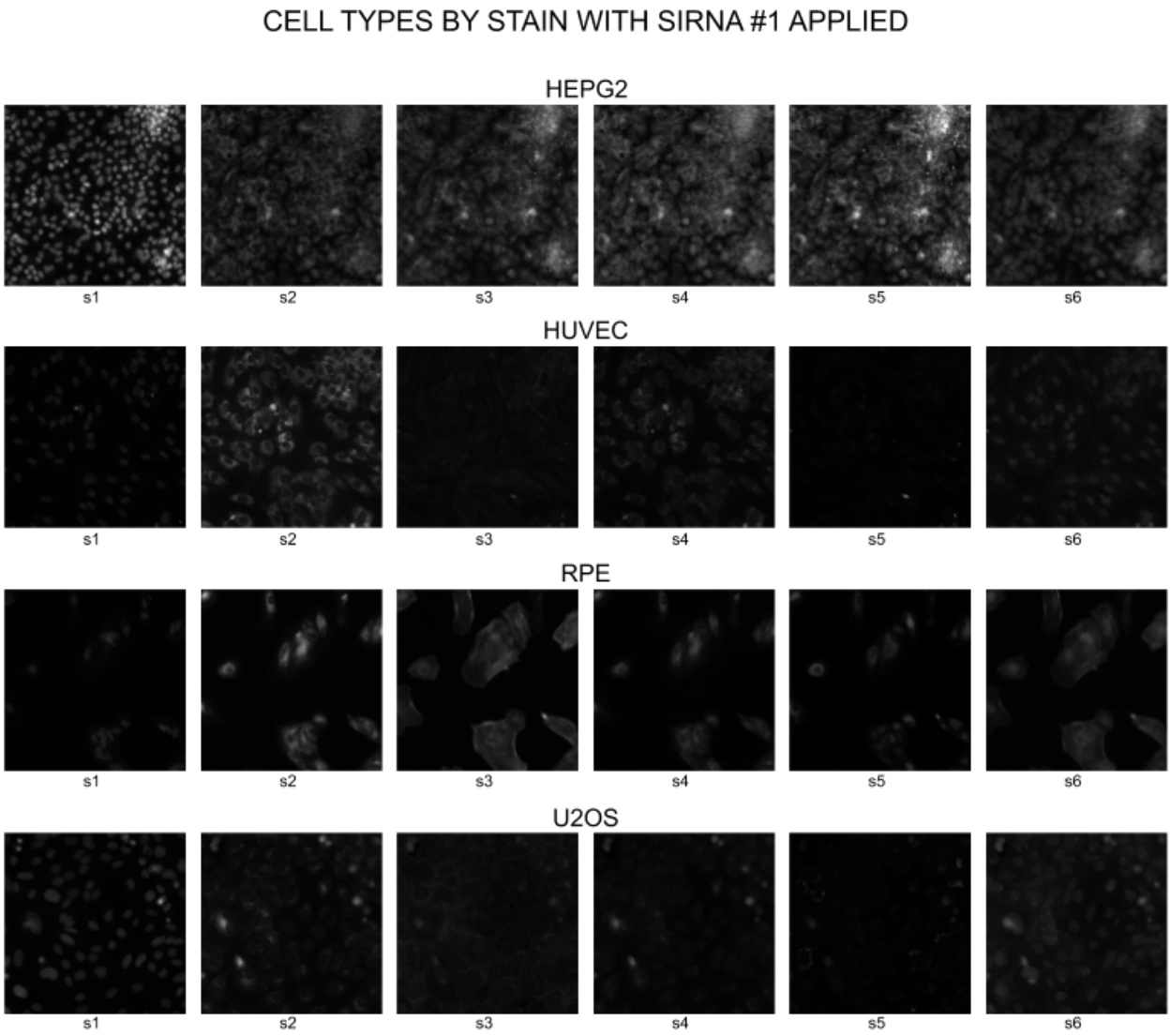

Figure 3. One sample set of six images from each of the four cell types with siRNA #1 applied. The above images were taken from the first experiment for each cell type in the training dataset.

In addition to an image set, a few CSV files were provided. The test and train CSVs included the ID code of the well, the experiment, plate, and specific well of each sample. The train CSV file also included the correctly labeled treatment (the siRNA, one of 1,108). For the purpose of the competition, the test CSV did not include the siRNA and test accuracy was evaluated from a competitor-uploaded csv file to the competition website containing the ID codes of the test data and their predictions. Similar CSV files were provided for the test and train controls; in both cases, the correct label was provided along with whether or not the control was positive (treatment) or negative (no treatment). Finally, a CSV file was provided that included the ID code, experiment,

plate, well, site, and channel of each image along with statistics computed from the image pixels (mean, median, min, max, and standard deviation).

While all data were present (i.e., no missing numbers, images, etc.), the dataset is highly imbalanced towards the HUVEC cell line, which is nearly half of the total data (24/51 experiments). A major challenge of this data set is that, in addition to the differing physical characteristics of each cell type, the same siRNA may not have the same morphological effect across the cell types. Figure 4 illustrates the morphological differences between the four cell types with the same siRNA applied. Due to this and the fact that there are relatively few sample sets for the non-HUVEC cell types, it may prove difficult to produce a model robust across all cell types and experimental treatments.

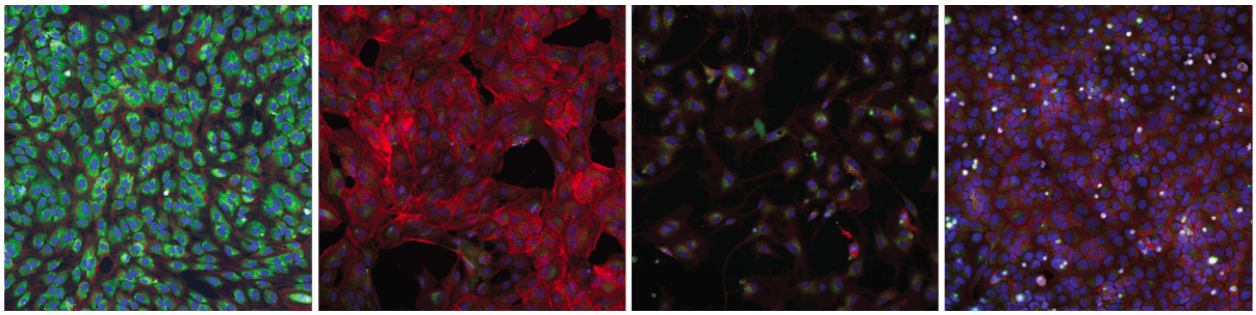

Figure 4. Composite images (i.e., all 6 channels combined) of the same treatment applied to the four different cell types. Images courtesy of ("RXRX," n.d.-b).

One of the main goals of this competition was to challenge participants to not only classify images correctly, but to devise a method of teasing out batch effects that inevitably occur across experiments. As shown in figure 5 (a set of composite images from two different treatments in the HUVEC cell type), it can be difficult even for a highly trained human to separate out treatment and batch effects from a series of images. This variance in experimental results due to batch effects adds an additional layer of complication the final model(s) needs to overcome, either implicitly as an effect of training or explicitly by using additional techniques to normalize images across experiments.

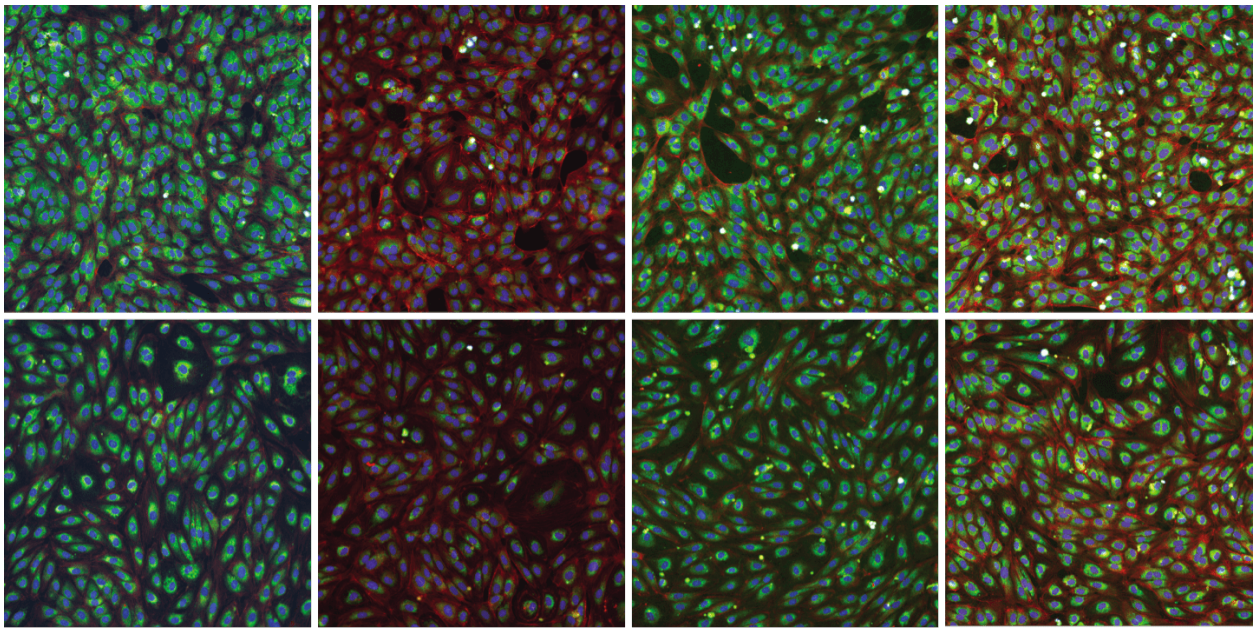

Figure 5. Images from two different treatments (rows) in HUVEC cells from four experimental batches (columns). Courtesy of ("RXRX," n.d.-b).

Algorithms and Techniques

The overarching approach I plan to use in this project is to develop a convolutional neural network for image classification using the Keras library. I will start by building a pipeline to classify one of the six images of each set at a time; I will then progress to either a multi-image model or a single image model where all six images are combined into one composite image for analysis. This decision will be predicated by the memory pressure and computational intensity of a six image model as compared to how well the composite image model can perform. A six image model will almost certainly be slower and more difficult to train than the single image model (especially with my available computational resources); however, I suspect that some information will be lost in the composite image model and will likely result in lower resulting accuracy.

Regardless of the choice to use either the 6-channel or the composite image for training, I plan to feed the network with a data generator that will resize and augment the images; resizing to reduce memory requirements (i.e., due to less trainable parameters) and augmentation to help avoid overfitting of the model. Once the images have been fed into the network, they will pass through a number of convolutional blocks each consisting of a convolutional layer, batch normalization layer, and a pooling layer.

The number of blocks and hyperparameters will be determined through experimentation, but I expect to start with a 5x5 or 3x3 filter and use 3x3 filters for the remaining layers; batch normalization will be left at the defaults. I will add additional blocks for dimensionality reduction as necessary. I also intend to employ at least some of the CNN optimization techniques discussed in a recent paper (He et al., 2019); specifically, I intend to use the warmup learning rate with cosine decay algorithm to automatically adjust learning rate as training progresses.

Especially with image classification, transfer learning has been well established as a potent technique to improve neural network classification accuracy (Huh, Agrawal, & Efros, 2016; Oquab, Bottou, Laptev, & Sivic, 2014). It is possible that an existing pre-trained neural network such as DenseNet, Resnet, etc., could improve the accuracy of the final model even though the input data may not be similar in nature. After testing publicly available models for transfer learning, I will also explore creating my own additional model for transfer learning using the control data (which are not used for model training or testing). Since the cell types are the same and there are more training samples available (the same controls are available on all four plates of each experiment instead of just one, resulting in four times as many control samples), transfer learning from such a network could also improve the final model accuracy.

If the preceding techniques do not provide reasonable accuracy, I will also experiment with creating a separate model for each of the four cell types. I expect this will result in a models that are easier to train since the physical and morphological characteristics of the four cell types differ with and without treatment.

Benchmark

As there are 1,108 possible classes for this model to choose from, a random choice model would result in approximately 0.09% accuracy. The resulting model should improve on this number. Training time and/or time to classification could also be profiled as a benchmark but will not be considered here.

It should be noted that prior literature has explored various CNN architectures for cellular image classification tasks, achieving as high as 97.2% test accuracy (Oei et al., 2019). Although this model only had to predict three classes, similar approaches could be explored in this project.

III. Methodology

Data Preprocessing

The first steps in preprocessing that I needed to take was to generate a convenient method for feeding images into my neural network. Since the number of training images was quite high (around 500,000), it would be infeasible to load them all in memory for processing. As such, the Keras library has several methods for feeding data to the model including a `fit_generator` method that allows data to be pulled into memory as needed instead of all at once. Thus, the main work in the data pre-processing step was to build a way to ensure a generator could consume image paths and return images to feed into the model.

As previously mentioned, the dataset provided 512x512x3 PNG format images for analysis along with CSV files containing the correct labels. The provided folder and file naming structure for the images was as follows (each bullet is a folder/file name in the hierarchy):

- `{test|train}`
- `{CELL_TYPE}-{experiment for that cell type #[1-24]}`
- `Plate{# [1-4]}`
- `{Row Letter[B-O]}{Column #[2-23]}_s{site #[1-2]}_w{channel #[1-6]}.png`

For example, an image in the second HEPG2 cell line experiment, first plate, well B02, first set of images, first channel has path and file name of `./train/HEPG2-01/Plate1/B02_s1_w1.png`. The file and folder names did not contain the label of the sample; this information was provided in the `train.csv` file. This file, however, did not contain the image names and file path (it contains `id_code, experiment, plate, well,` and `sirna`). Thus, to be able to feed my network quickly, I added functions to the `preprocessing.py` file to take an existing CSV, read it as a pandas dataframe and add the folder/file path along with the parsed cell type (from the experiment column). In addition, the original CSV did not split the samples into two site rows; only one row per sample was provided. For ease of processing, I added an extra row for the secondary site so that the data generator could simply read the image path directly from the dataframe and no further preprocessing needed to occur during training. The resulting dataframe was saved to a new CSV to be used as a parameter to the model.

The final preprocessing step I added in was image augmentation. In an attempt to keep the model and pipeline simple, I initially trained my models without image augmentation. However, my resulting models displayed overfitting (with training loss decreasing and validation loss increasing each epoch) fairly quickly into training. Thus, I added image

augmentation into my custom data generator and this issue was resolved. Images were randomly (p=.5) augmented as follows:

- Apply one of the following transformations:
    - Add the same value between -10 and +10 to each pixel ("Overview of Augmenters — imgaug 0.2.9 documentation," n.d.-a)
    - Multiply the same value between 0.9 and 1.1 to each pixel ("Overview of Augmenters — imgaug 0.2.9 documentation," n.d.-b)
    - Apply contrast normalization to the image by a factor of 0.9 to 1.1
- Horizontally flip the image (p=.5)
- Crop the image by a factor of between 0 and .1
- Vertically flip the image (p=.5)

Following augmentation, the images were resized to 224x224 and the pixel values were divided by 255 to be input into the model. I initially chose 224x224 so I could experiment with using the VGG16 imagenet model for transfer learning; I later attempted to increase the resolution to 350x350, but I experienced out of memory issues at this resolution and reverted to a size of 224x224.

The original dataset provided images from each of the six stains for each sample and it was left to the implementer to combine images into a composite if desired. A script publicly available on kaggle was used for this purpose (purplejester, 2019).

Implementation

As previously mentioned, the main library used for this project was Keras. Initial coding and testing was done via jupyter notebooks with logging in Tensorboard, but it quickly became cumbersome to keep track of experiments that had been run and how they had been configured. I slowly migrated the process to run from an anaconda prompt with logging sent to the website WandB in two project folders ("Weights & Biases," n.d.-a), ("Weights & Biases," n.d.-b). This eventually allowed me to save configuration files with parameters that were uploaded with the charts for each run automatically. Model checkpoints were saved for each run at the end of each epoch when the model's validation loss improved.

The first phase of implementation consisted of simply building the training pipeline and testing simple models. I initially used only one of the six images from each sample for training and testing. In this portion of the process, I was able to use the built in Keras `ImageDataGenerator` class and feed data to the model using a pandas dataframe. I did not initially implement image augmentation and the model contained as few as two

and as many as six convolutional blocks during this phase, ranging from 32 to 512 filters (doubling the filter size each block). The code worked and the model was able to train, but the results after training were very poor and the model quickly overtrained.

Next I added in a custom data generator to feed the model all six images from each sample. I modified a publicly available kernel to suit my use case (chandyalex, 2019); this was likely the most challenging portion of the implementation to get right. Using six images in parallel greatly increased memory pressure of the model and I experienced numerous out of memory issues and problems with GPU sync failures throughout the process since I was using my local machine. To decrease the number of trainable parameters and be able to train the model without interruption, I added several convolutional blocks and adjusted the size of my first dense layer coming out of the CNN block. The resulting model architecture is depicted in figure 5.

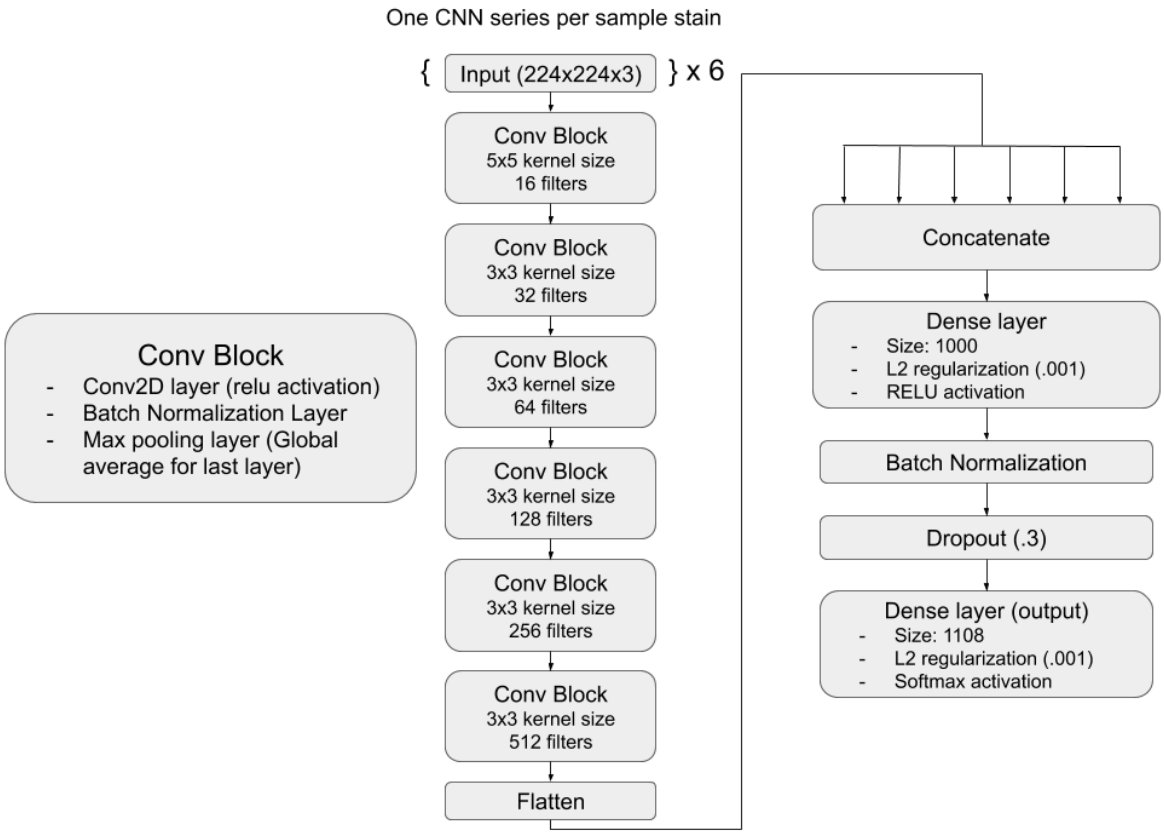

Figure 6. Final model architecture diagram.

As the experimentation process continued, I added additional parameters and methods to my training runner to be able to train different types of models. The result of model

changes were compared to the baseline of random chance and previous best models; I kept changes that resulted in improvements in model validation loss validation accuracy. The code and configuration parameters for these techniques are still available in the repository, but are not all referenced or in use by the training runner. This included the following types of training:

- Full dataset single-image training
- Full dataset Multi-image training
- Full dataset training using a publicly available pre-trained network (VGG16)
- Full dataset training for just learning the type of cell (one of four classes) rather than learning the treatment
    - This model was subsequently used for transfer learning on the full dataset
- Full dataset training to predict the treatment type of the controls only (one of 31 classes)
    - This model was subsequently used for transfer learning on the full dataset
- Training on the full set of treatments for each of the four cell types for both controls and for treatments
    - These models were subsequently used for transfer learning on the full dataset

The final piece of implementation was adding the ability to load a saved model, generate predictions for the test data, and save the results to a CSV file for upload to the Kaggle competition site. This necessitated a separate, stripped down version of the six image data generator due to the additional logic embedded in the training/validation generator.

Refinement

LEARNING RATE SELECTION

Initially I experimented with static, unchanging learning rates of .1, .01, .005, .001, .0005, and .0001. I then added in a Keras callback that decreased the learning rate by a factor of 0.5 after a plateau. After more research and reading through the “Bag of tricks” paper (He et al., 2019), I decided to give a more advanced learning rate a try - in that paper, they described two techniques for adjusting learning rate during training - one was to warm up the learning rate over a number of epochs to try and provide stability early in training. The second technique was to add a cosine decay after the warmup period. In contrast to a step decay where learning rate is dropped in a step-like fashion

and held for a number of epochs before further dropping the learning rate, cosine decay gradually drops the learning rate over time. Code for the warmup cosine decay scheduler was publicly available ("Bag of Tricks for Image Classification with Convolutional Neural Networks in Keras | DLology," n.d.) and imported for use in training. Finally, the authors of the "Bag of Tricks" paper noted that increasing training batch size can reduce noise in the gradients, and as such, learning rate could be increased. As they suggested, I eventually set the target learning rate to be `0.1 x batch_size / 256`, where batch size was typically 16 or 32.

CONVOLUTIONAL BLOCK REFINEMENT

I experimented with several variations of hyperparameters and convolutional block configuration. As suggested by this blog post (Hubens, 2019), I changed the Conv2D kernel initializer from its default to **`he_uniform.`** Following that change, I also tested out applying L2 regularization. Batch normalization was left in the model set to the default; after several test runs with L2 regularization set at 0.1, 0.01, and 0.001, I found that the model's validation accuracy was noticeably worse and I subsequently removed this parameter. Additionally, other online posts discussed changing the order of the activation layer and the batch normalization layers; I experimented with this and found the results to be inconclusive. I also experimented with adding in dropout layers to the convolutional block; this also resulted in much poorer performance (at least with batch normalization left in). Finally, I made numerous attempts at adjusting the number of convolutional blocks, filters, and kernel size. Overly deep series of CNN blocks as well as very small CNN blocks did not perform well on the validation set and either overtrained or failed to learn at all. In addition, with too few blocks to reduce the dimensionality of the dataset, I quickly ran into out of memory errors. Thus, I needed to have a minimum of 3 convolutional blocks for the model to run; more than six as presented in the final model (figure 6) was also counterproductive (in terms of validation accuracy and validation loss).

TRAINING WITH FULL DATASET AND TRANSFER LEARNING

All of my initial models were run against the full dataset and included all four cell types. However, the validation accuracy of the models did not improve much after the first few epochs (if at all). I experimented with different learning rates, L2 regularization, different CNN architectures (e.g., number of convolutional blocks, including modeling the network after other published CNNs such as VGG), and transfer learning. To my surprise, transfer learning using the VGG16 imagenet model as I implemented it did not

appear to improve my model's performance. The highest validation accuracy for the full model I was able to achieve during this phase of training was 2.5%.

TRANSFER LEARNING USING CELL TYPE MODEL

Following the lackluster results of training the full model to predict one of 1,108 treatments, I decided to scale back and attempt to solve a simpler problem as a sanity check. I modified the network to simply predict one of the four cell types, resulting in approximately 98% validation accuracy. I saved the resulting model and attempted to use it as a transfer learning model for the full network, but the new model's validation accuracy did not improve. Validation accuracy during this phase did not exceed 2%.

COMPOSITE IMAGE TRAINING

After converting the image sets into composite images, I briefly experimented with a single image network on the full dataset. This resulted in a simpler model with fewer trainable parameters, but also poorer accuracy. Validation accuracy during this phase did not exceed 1%.

CONTROL TRAINING WITH TRANSFER LEARNING

I decided to again scale the problem definition back temporarily for an additional sanity check to ensure that I could build at least a simple model that could learn, similar to the model trained to predict one of the four cell types. In this case, I experimented with a model that only used the control data - an extremely similar problem, but with more training samples and fewer classes. In this scenario, there were 31 total classes instead of 1,108; additionally, each of the 31 control samples were present on all four plates across all experiments, resulting in four times as many samples for the network to be able to train on. I was able to achieve approximately 60% validation accuracy training on the full dataset with this approach. I again tested transfer learning with this model; however, validation accuracy did not exceed 3%.

MODEL TRAINING WITH SPECIFIC CELL TYPE

Since the cell types differ in both morphology and in their response to treatment, I decided to try splitting my data by cell type. Although the results were still not impressive, this was the first substantial improvement to validation accuracy for the overall dataset that I achieved. Figure 7 in the free form visualization section shows the results from training the full dataset split by cell type using the model architecture shown

in figure 6. The final model presented here used this strategy and my Kaggle submission was based off models built from these runs.

IV. Results

Model Evaluation and Validation

The final model architecture for this project was shown in figure 6. Results from creating separate models split by cell type are shown in figure 7. As shown in figure 7, validation accuracy was approximately 51% for HUVEC, 37% for RPE, 32% for U2OS, and 19% for HEPG2. Once all four models had been run, I selected and loaded the model checkpoint that had the best validation loss for each of the cell types, generated predictions for the test data, and saved CSV prediction files to be combined and uploaded to the competition server. This resulted in a leaderboard score of 0.196, which was below what I expected for the validation scores shown in figure 7. However, this score is still far better than random chance, which would be 1/1,108; where this score is approximately 1/5.

The greatest improvement to overall model accuracy was driven by splitting the data and training models by cell type. Interestingly, there was a large disparity in validation accuracy when the model was split by cell type (a range of 30% between the cell types). The final model architecture was chosen over other models based on memory/computational requirements and changes in validation accuracy after running many training sessions. Although the resulting accuracy was not overly impressive, I believe the final model is reasonable given the challenging nature of this problem.

At roughly 20% accuracy on unseen data, I do not believe this particular model is robust enough for practical application; I do however, believe the results can be trusted as there were nearly 20,000 samples used in the test set. Indeed, the top ten competitors on the Kaggle leaderboard all had scores of 0.95 or greater at the time of this writing; with this in mind, there are certainly other techniques that could be employed to improve this model. It is possible that the data exploit listed in a competition discussion post greatly simplified the models of the top slice of competitors (https://www.kaggle.com/c/recursion-cellular-image-classification/discussion/102905#latest-624588). In this exploit, competitors discovered that 1,108 treatments were not randomly assigned across each of the four plates across the experiments; instead, the

same 277 treatments were assigned to each plate (but randomly assigned to wells). This information could be mined from the data and used during training to reduce the odds from one in 1,108 to one in 277. I chose not to use this exploit for this competition as this type of information may not be available in real world scenarios.

Justification

The final model's accuracy as reported by the competition website (19.6%) was considerably better than the benchmark of random chance (0.09%). However, this model did not reach my original goal of >= 70% and I would not consider this problem to be solved by this model. As previously discussed, this model used a set of six fairly straightforward CNNs in parallel and then concatenated into a fully connected network to attempt to solve this problem; based on the current results in the kaggle leaderboard, it appears there are alternative approaches than the one reported here that could provide better results for this problem.

V. Conclusion

Free-Form Visualization

Since the competition is still in progress and the test data is unlabeled, I am unable to present statistical visualizations on the test data. However, figure 7 below illustrates the validation accuracy and validation loss from the cell type training models used in my submission to the kaggle leaderboard. Almost half of all the experiments (24/51) were conducted using the HUVEC cell type; not surprisingly, the HUVEC cell type model also had the greatest validation accuracy at just over 51%. Interestingly, the HEPG2 cell type (11 experiments) had the worst validation accuracy even though U2OS had the least amount of data (5 experiments). All four cell type models used the same network configuration and hyperparameter setup and I was not able to determine the source of this discrepancy in accuracy.

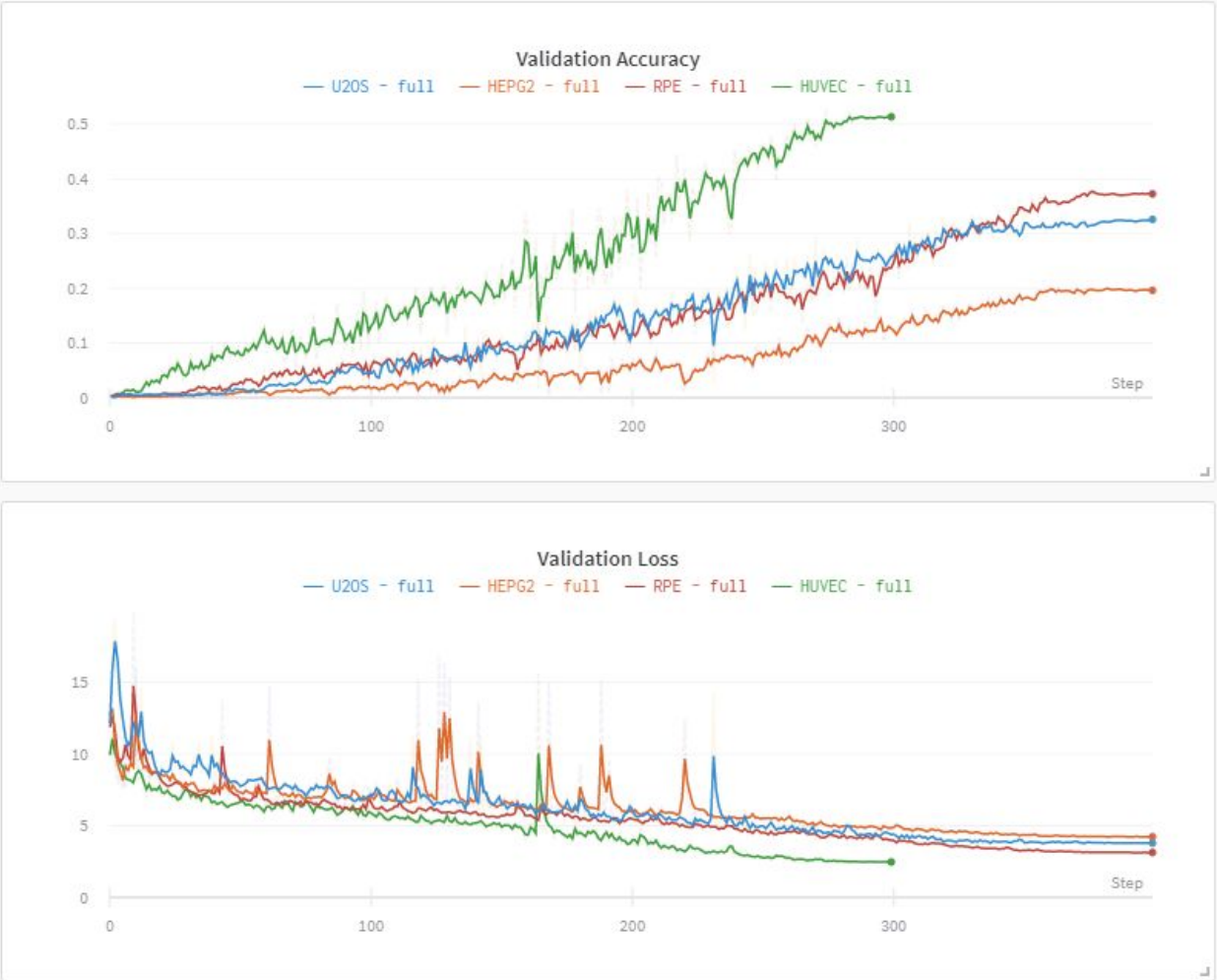

Figure 7. Validation accuracy and loss results from full dataset using network architecture displayed in figure 6. In both graphs, the x-axis is "step" or epoch. Note that HUVEC was run for 300 epochs instead of 400. The full report is available online here: https://app.wandb.ai/budnarr/Capstone-project/reports?view=budnarr%2FFull%20result%20set%20%231

Reflection

This project entailed building a preprocessing, CNN model training, and postprocessing pipeline that could consume a set of cellular images with one of 1,108 treatments applied, predict treatment probabilities, and finally transform the model's predictions into a CSV file to be uploaded to a Kaggle competition server for scoring. The overall process was iterative and as the project progressed, I added more features and automation to my process.

PREPROCESSING

Initially, my pipeline read in the originally provided CSV on each training run and preprocessed the data to generate the filenames needed for training. However, this proved to be somewhat slow and I eventually extracted this process out and generated a new file so this would never need to be run again.

MODEL BUILDING AND PERFORMANCE ANALYSIS

I tried many, many variations on my model throughout the project. I started with a very simple, single image model in order to verify that I could at least get a full training run going. The main strategies I used while developing my models were to first vary the model itself by changing number of layers, hyper parameters, etc.; and second, to use different slices of the data in the form of training on the composite image, controls, and on specific cell types. Training on specific cell types appeared to have the greatest improvement in my overall model accuracy.

The next main issue to tackle that, in retrospect, I wish I had known to do a better job of up front, was to be able to keep track of my training runs and their associated configuration and results. I started out using tensorboard to verify my results and at least keep track of some CNN configuration, but I did not find it to be a robust tool for keeping track of everything I needed, such as hyper parameters, notes on a training run, etc. I eventually found the site https://www.wandb.com/ which I was able to use for free to not only visualize my training results, but also keep track of the parameters I configured for each of my training runs and add notes as needed. While I discovered this site fairly early in the process, I did not take full advantage of the capabilities of this product until much later during the training process. The bulk of my results can be viewed in these two project folders: https://app.wandb.ai/budnarr/Capstone%20project?workspace=user-budnarr, and https://app.wandb.ai/budnarr/Capstone-project?workspace=user-budnarr. Note that a number of failed runs were removed; in addition, I did not know about being able to save my training parameters until late in the project.

POSTPROCESSING

Since I ended up training on specific cell types rather than on the full dataset, the final step in this process was to generate predictions from each of the cell type models and then transform them into a file for upload and scoring on the Kaggle competition server. This process was fairly straight forward.

DIFFICULTIES

The biggest frustration I had in this project was that I ran into very frequent GPU crashes along with out of memory errors early on in the process. This interrupted the

training process and, since I had not yet learned how to resume training from a saved checkpoint, I wasted a lot of time starting and restarting the training process. My solution for the GPU crashes was to essentially start a training session and let it run without me using my computer; for the out of memory issues, I had to decrease the number of trainable parameters for my model.

The other big challenge was developing a multi-input model for training. The built-in Keras ImageDataGenerator is simple and easy to use, but I was unable to find a way to use that generator to feed my model with six images in parallel. Instead, I extended a community provided generator in order to suit this purpose.

Finally, I found it more difficult than I thought it would have been to develop a model that could result in a very accurate treatment predictions. There were also many times when I noticed that I had set a parameter incorrectly, but the model still trained and produced results that seemed in line with other results - this “silent failure” (so to speak) was perhaps the most nerve wracking part of the whole process.

The last main difficulty - or at least surprise - was that I did not seem to get much of any boost from the use of transfer learning. It is entirely possible I implemented the transfer learning incorrectly or that the treatments applied to the 31 controls were different enough to not be able to glean much information from them. However, I suspect an error in implementation that I was unable to detect.

Improvement

One of the main areas of improvement that I was unsure of how to implement was improving my data throughput and using cloud based training instead of local training. I joined the competition late and did not notice that starter code was provided along with a set of tfrecords (apart from the downloadable dataset) until I was far into the project. From what I found on tfrecords, it appears that they can help improve the speed of getting data into the model substantially (one blog post reported ~20% boost in overall speed). Another aspect that I missed out on was cloud based training. I discovered later into development that I might have been able to use some free credits to train my models in the cloud, which also could have improved my training speed.

One of the main challenges that this competition presented was to be able to tease out batch effects from treatment effects. I believe that the use of the controls are likely the

key to this since the same controls are available across all plates and all experiments. However, I was unable to devise a reasonable method of normalizing the treatment images to the control images.

Finally (and obviously), the resulting model accuracy could stand to be improved quite a bit. I do not yet know what solutions other competitors have provided, but I do see that the top 10 in the Kaggle leaderboard for this competition were all over 95% accuracy at the time of this writing. Though this was my first on-my-own project (and first competition as well), I would like to have been able to climb much higher on the leaderboard than I did.

REFERENCES

Bag of Tricks for Image Classification with Convolutional Neural Networks in Keras | DLology. (n.d.). Retrieved September 14, 2019, from https://www.dlology.com/blog/bag-of-tricks-for-image-classification-with-convolutional-neural-networks-in-keras/

chandyalex. (2019, July 24). Recursion Cellular Keras Densenet. Retrieved September 11, 2019, from Kaggle website: https://kaggle.com/chandyalex/recursion-cellular-keras-densenet

He, T., Zhang, Z., Zhang, H., Zhang, Z., Xie, J., & Li, M. (2019). Bag of Tricks for Image Classification with Convolutional Neural Networks. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. Retrieved from https://arxiv.org/abs/1812.01187

Hubens, N. (2019, July 6). Why default CNN are broken in Keras and how to fix them. Retrieved September 15, 2019, from Medium website: https://towardsdatascience.com/why-default-cnn-are-broken-in-keras-and-how-to-fix-them-ce295e5e5f2

Huh, M., Agrawal, P., & Efros, A. A. (2016). What makes ImageNet good for transfer learning? *arXiv*. Retrieved from https://arxiv.org/abs/1608.08614

Oei, R. W., Hou, G., Liu, F., Zhong, J., Zhang, J., An, Z., … Yang, Y. (2019). Convolutional neural network for cell classification using microscope images of intracellular actin networks. *PloS One*, *14*(3), e0213626.

Oquab, M., Bottou, L., Laptev, I., & Sivic, J. (2014). Learning and Transferring Mid-level Image Representations Using Convolutional Neural Networks. *2014 IEEE Conference on Computer Vision and Pattern Recognition*. https://doi.org/10.1109/cvpr.2014.222

Overview of Augmenters — imgaug 0.2.9 documentation. (n.d.-a). Retrieved September 11,

2019, from https://imgaug.readthedocs.io/en/latest/source/augmenters.html?highlight=add#add

Overview of Augmenters — imgaug 0.2.9 documentation. (n.d.-b). Retrieved September 11, 2019, from https://imgaug.readthedocs.io/en/latest/source/augmenters.html?highlight=add#multiply

purplejester. (2019, August 28). Joining channels into RGB images (parallel). Retrieved September 12, 2019, from Kaggle website: https://kaggle.com/purplejester/joining-channels-into-rgb-images-parallel

Recursion Cellular Image Classification. (n.d.-a). Retrieved September 8, 2019, from https://kaggle.com/c/recursion-cellular-image-classification

Recursion Cellular Image Classification. (n.d.-b). Retrieved September 8, 2019, from https://kaggle.com/c/recursion-cellular-image-classification

RXRX. (n.d.-a). Retrieved September 8, 2019, from https://www.rxrx.ai/#about

RXRX. (n.d.-b). Retrieved September 8, 2019, from https://www.rxrx.ai/#the-data

Weights & Biases. (n.d.-a). Retrieved September 11, 2019, from https://app.wandb.ai/budnarr/Capstone-project

Weights & Biases. (n.d.-b). Retrieved September 11, 2019, from https://app.wandb.ai/budnarr/Capstone%20project