Detection of Acute Lymphoblastic Leukemia Using Convolutional Neural Networks

Andrew Benecchi

2022-05-02

Abstract

This paper explores automated classification of nuclei of immature white blood cells as either healthy or affected by leukemia; compared to previous published papers, the number of sample images in the training and testing sets is far larger due to augmentation through scalable, parallelizable methods. Linearly-connected convolutional neural networks are compared to branching multi-scale convolutional networks in order to determine the optimal classifier for cell-level classification, varying batch size, learning rate, learning rate policy (constant versus triangle-waveform). Predictions are taken on a separate set of cell data, and a threshold for positive patient classification is set using the separate cell data.

Introduction

Acute lymphoblastic leukemia (ALL) is a cancer of the blood and the leading cause of cancer and cancer-related death among children. Lymphoblasts are immature white blood cells that develop into B and T lymphocytes, and their production is moderated by the body's natural signaling mechanisms. When one has ALL, both their maturation process of their lymphoblasts and body's limiting signaling of lymphoblast production malfunctions, resulting in an increased blast count of malformed cells.

The French-American-British model described in Bain and Estcourt (2013) divides classification of cancerous blasts into 3 categories: L1, L2, and L3.

L1: Small to medium-sized, regular blast cells with a high nucleus-cytoplasm ratio

L2: Cells vary more in size, possibly with visible nucleoli, nuclear clefts, and more plentiful cytoplasm

L3: Blast cells with strongly basophilic (i.e. dyed deep blue) cytoplasm and prominent cytoplasmic vacuoles

Because the population is likely largely early-stage, affected cells appear to be more likely to have features of L1 and L2 blasts with nucleoli and irregular nuclei shapes. Visual inspection of some sample images appear to validate the frequency of cell abnormailities within the leukemic group, but the dataset still contains counterexamples—healthy-appearing cells in leukemic samples and abnormal cells in healthy samples.

The data come from the C-NMC Leukemia dataset from Gupta and Gupta (2019), which contains training data in three class-separated folds, with 3389 nuclei from healthy patients and 7272 nuclei from leukemic patients. Every image in the training data is labeled with their patient ID, cell number, and whether or not they are healthy. The validation data contains 1867 images, containing the suppressed cell ID (i.e. patient ID, cell number, ALL status), filename (1.bmp, … 1867.bmp), and class label. There is also a testing dataset of 2597 images with no labeling information intended for use in prediction in competitions, but this data is of no use in the paper. The original validation data will become the testing data, and a validation set will be created by subsetting the training data.

Table 1: Comparative table of literature's data sources.

Paper	Image Type	Source	Pre-Test Cells
Nasir et al. (2013)	Blood smear	Other	1009
Putzu et al. (2014)	Blood smear	IDB1	245
Mohapatra et al. (2014)	Blood smear	Other	270
Rawat et al. (2015)	Whole cell	IDB2	98
Mishra et al. (2017)	Blood smear	IDB1	770
Mishra, Majhi, Kumar Sa (2019)	Blood smear	IDB1	678
Tuba et al. (2019)	Whole cell	IDB2	100
This paper (2021)	Nucleus	C-NMC	5760

Table 2: Comparative table of literature's methodologies.

Segmentation	Features	Classifiers
HSI	Size, shape, color	Fuzzy ARTMAP NN (ternary classifier)
CIELAB	Shape, texture	Linear SVM, Polynomial SVM, RBF SVM, NB
Shadowed C-means	Size, shape, texture (GLCM), color	Ensemble: k-nearest, MLP, RBF NN, NB, SVM
Grayscale	Shape, texture (GLCM)	SVM
CIELAB	Discrete cosine transform	SVM
Shadowed C-means	DOST, PCA	Random Forest
HSV	Shape, uniform LBP	RBF SVM, k-nearest, NB
Premade	Pixel data	CNN

Literature Review

Segmentation

Much of the existing literature regarding the classification of lymphocytes uses blood smears as the initial data source, specifically the ALL-IDB datasets, which include whole smears in the dataset ALL-IDB1 and WBC-centered individual cell images in the dataset ALL-IDB2. As a result, the segmentation methods of the whole cell from the blood smear and the nucleus from the cytoplasm should be investigated in order to understand potential artifact sources. Putzu, Caocci, and Di Ruberto (2014) and Mishra et al. (2017) applied a feature extraction technique from Di Ruberto and Putzu (2013) where they separate the nuclei from the cytoplasm in RGB color space (having used CMYK in their segmentation, which is unnecessary for my data), then using RGB green color contrast and the $CIELAB$ ($L^*a^*b^*$) a^* ('reddishness') component together to split the cell into its two components; RGB operations alone accidentally sometimes include cytoplasm granulocytes as being part of the nucleus, hence the necessity of the color space conversion. Mohapatra, Patra, and Satpathy (2014) and Mishra, Majhi, and Sa (2019) also utilized $L^*a^*b^*$ conversion in their segmentation but used a fuzzy c-means (a.k.a. soft k-means) clustering method where datapoints can have multiple class labels in order to segment the nucleus and cytoplasm. Tuba et al. (2019) converted the image into HSI coordinates, then applied a hue threshold to separate the cell from the background, then a saturation threshold approach along with morphological opening to separate the nucleus from the cytoplasm.

Feature Extraction

There are four families of applicable image features that are of interest: size, shape, texture, and color. From the separated two portions of the image, Putzu, Caocci, and Di Ruberto (2014) extracted the following calculated shape features from the binary version of the image (i.e. if the pixel has color, then it is colored white, else black): area, perimeter, convex area, convex perimeter, major axis, minor axis and orientation features, and used them to calculate elongation $(1-\frac{a_{\text{min}}}{a_{\text{maj}}})$, eccentricity $\frac{\sqrt{a_{\text{maj}}^2-a_{\text{min}}^2}}{a_{\text{maj}}}$, rectangularity $\frac{A}{a_{\text{min}}a_{\text{maj}}}$, compactness $4\pi\frac{A}{\text{perimeter}^2}$, this shall be known as perimeter compactness), convexity $\frac{\text{convex perimeter}}{\text{perimeter}}$, roundness $4\pi\frac{A}{\text{convex perimeter}^2}$, this shall be called convex perimeter roundness), and solidity $\frac{A}{\text{convex area}}$. It is implied that measurements of cytoplasm area include the black pixels within the interior of the binary cytoplasm image, as the cytoplasm contains the nucleus. In addition to these 14 shape features, calculated on both the nucleus and cytoplasm (28 calculated features), they added two additional measures: cytoplasm-to-nucleus area ratio and the number of nuclear lobes (30 total shape features). The primary concern with shape features is issues with segmentation, but these issues are likely not as concerning if the input images are already cleaned. For texture features, conducted on grayscale images, they evaluated the following descriptors applied to the grey level co-occurrence matrix (GLCM) calculated starting from sub-images in grey level: autocorrelation, contrast, correlation, cluster prominence, cluster shade, dissimilarity, energy, entropy, homogeneity, maximum probability, sum of squares (variance), sum average, sum variance, sum entropy, difference variance, difference entropy, information measure of correlation, information measure of squared correlation, inverse difference normalized and inverse difference moment normalized (20 texture features); these features were calculated for angles of 0, 45, 90 and 135 degrees, resulting in 80 total texture features. In addition to the shape features, each of the RGB channels had the following color features extracted: background-censored mean, standard deviation, smoothness, skewness, kurtosis, uniformity and entropy, with to reduce the skew (21 total color features). All total, there are 131 features per cell, and a subset of 50 features was tested against the full model for comparison.

Mohapatra, Patra, and Satpathy (2014) used shape, texture, and color features, on binary, grayscale, and color images. For shape, they considered cytoplasm nucleus area, nucleus/cytoplasm ratio, total cell area, nucleus perimeter, nucleus form factor (identical to compactness from Putzu, Caocci, and Di Ruberto (2014)), nucleus roundness (not to be confused with the roundness in Putzu, Caocci, and Di Ruberto (2014), this measure is $\frac{4}{\pi}\frac{A}{a_{\text{maj}}^2}$ and shall be known as major-axis roundness), major axis-minor axis ratio, and compactness (not to be confused with compactness in Putzu, Caocci, and Di Ruberto (2014), this measure is $\sqrt{\frac{4}{\pi}\frac{A}{a_{\text{maj}}^2}}$ and should be called root-roundness compactness). They also calculated a measure of nucleus roughness using Hausdorff Dimension, and taking the variance, skewness, and kurtosis of Euclidean contour distances from the centroid. The previous four roughness values were also calculated for the cytoplasm. For texture, they used three approaches on only the image of the nucleus. One set of texture features came from Haar wavelet texture filtering, taking the mean and variance of the original image with low-pass filtering, and high-pass filtering in the horizontal and vertical directions. They also looked at high-pass filtering in the diagonal direction, but it provided little information useful for classification. They also used GLCM features, where they extracted contrast, correlation, homogeneity, energy, and entropy; this was carried out using offsets in the four cardinal directions. They also carried out a discrete Fourier Transform (DFT) on the nucleus image, they calculated mean, variance, skewness, and kurtosis values for the DFT. For color features, they calculated the background-censored mean of RGB color channels and HSV RGB color channels of the nucleus. The same calculations were performed on the cytoplasm. All features of all types were standardized to have zero mean and unit variance.

Rawat et al. (2015) used area, circular-approximation perimeter, area-approximated diameter, Euler's number e, convex-area solidity, major axis, minor axis, eccentricity, orientation, convex area, and extent for shape features; these values were calculated for both the nucleus and cytoplasm. They also used the following calculated values for GLCM: angular second moment (energy), contrast, correlation, sum of squares (variance), inverse difference moment (homogeneity), sum, average, sum of variance, sum entropy, entropy, difference of variance, difference of entropy, and information of correlation; these values were calculated for both the nucleus and the cytoplasm.

Tuba et al. (2019) implemented whole-cell area, cytoplasm-nucleus area ratio, whole cell-nucleus area ratio, nucleus perimeter, and nucleus circularity as shape features, and a reduced version of local binary pattern (LBP) called uniform local binary pattern (ULBP). ULBP reduces the number of patterns considered to be distinct, reducing a 256-feature set of LBPs to 59 ULBPs.

Mishra et al. (2017) used an approach where they applied the discrete cosine transform to images of the nucleus, and then evaluated restricting the number of coefficients used in classification tasks to set a threshold of 90 principal components.

Mishra, Majhi, and Sa (2019) applied the following technique: First apply a discrete orthonormal S-transform (DOST) to the grayscale cell images, which with the 256x256 image input, returned a 65536-dimensional feature vector. With so many features, small sample size was a concern. The team used principal component analysis (PCA) and linear discriminant analysis (LDA) to reduce the number of dimensions; for the first d principal components, LDA was carried out, and the earliest accuracy peak for LDA determined the threshold for number of principal components included; the experiment determined the first 35 principal components to be optimal.

Classifiers

Putzu, Caocci, and Di Ruberto (2014) evaluated four different SVM kernels: linear (L), quadratic (Q), polynomial (P) and Gaussian radial basis (R); their performances were compared to those of Euclidean (L2) k-nearest neighbors, and naive Bayes (NB) with a Gaussian (G) and kernel data distribution (K), and decision trees. Overall, using the full set only provided meaningful improvements for both naive Bayes classifiers and the radial-basis SVM. The radial-basis SVM provided the best classification accuracy overall.

Mohapatra, Patra, and Satpathy (2014) utilized ensemble containing a naive Bayes classifier, k-nearest neighbors (kNN), multilayer perceptron (MLP), radial basis function (RBF) neural network, and support vector machine (SVM). The five models use majority voting to decide their final classification.

Rawat et al. (2015) used a support vector machine in their analysis, and found that the ideal combination of shape and texture features for inclusion in the model were all texture features for both the cytoplasm and nucleus combined with shape features from just the nucleus.

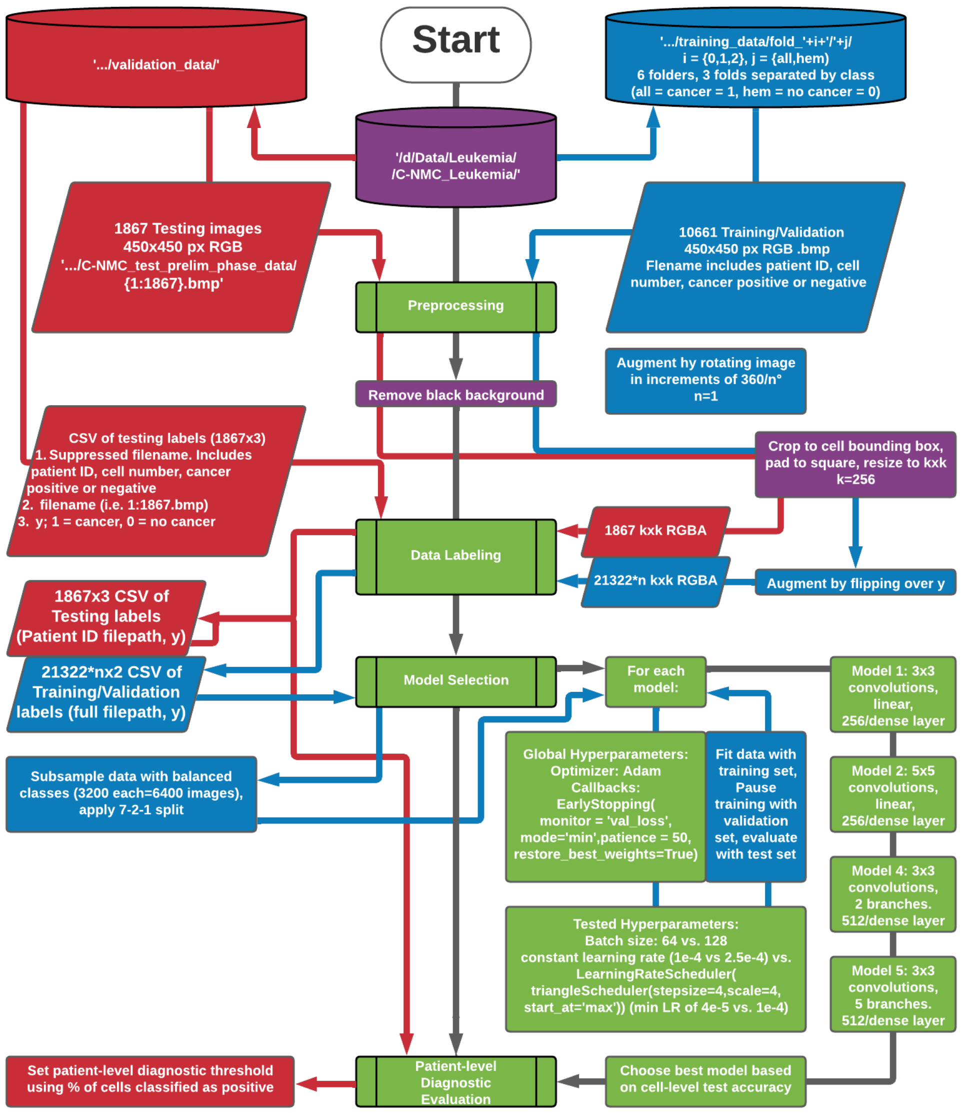
Mishra et al. (2017) compared their results to a neural network with backpropagation, a k-nearest neighbors, and a naive Bayes classifier; with many coefficients, the SVM performed best in terms of accuracy; its sensitivity $\left(\frac{TP}{TP+FN}\right)$ and specificity $\left(\frac{TN}{TN+FP}\right)$ were also measured, though the k-nearest neighbors had the best sensitivity. The high sensitivity of the k-NN is good at preventing false negatives, which are much more consequential to patient outcome than false positives, which can more easily be rescreened using a secondary method of diagnosis. The SVM performs best in specificity (a metric in which the k-NN performs quite poorly), which has an impact on patient well-being since false positives of a cancer diagnosis are significantly emotionally burdensome. With multiple images per cell and multiple cells per patient in my dataset, the overall likelihood of mis-diagnosing a patient should be quite low given that the prediction at the patient level is properly thresholded, as some cells in affected patients are still healthy despite the presence of leukocytes. As a result, a model that is more likely to predict false negatives as opposed to being averse to negative predictions at the cell-level may be transformed into a more accurate predictor with thresholding based on proportion of cells diagnosed as positive.

Tuba et al. (2019) used a radial-basis function SVM, which has two hyperparameters: the misclassification tolerance C and training point influence radius γ; they tuned their hyperparameters using a bare-bones fireworks algorithm (BBFWA). The advantage of BBFWA over grid search is that it generates novel points for each dimension instead of performing a tedious grid search; with only two hyperparameters to consider for the RBF SVM, this algorithm is quite an attractive alternative to grid search. Its sensitivity, specificity, and accuracy were measured against a neural network with backpropagation, a k-nearest neighbors, a naive Bayes classifier, and a DCT-based SVM from Mishra et al. (2017). Their SVM had the best accuracy overall, but the k-nearest neighbors performed best in sensitivity while the DCT performed best in specificity.

Cui, Chen, and Chen (2016) introduced the multi-scale convolutional neural network for time series classification problems. The model can be broken up into three components: the transformation stage, the local convolution stage, and the full convolution stage. The transformation stage can extract information in two ways: either through multi-scale transformation, where one downsamples the original time series (e.g. taking the 1st, 3rd, 5th, etc. observation), or through multi-frequency transformation, where one extracts frequency information (e.g. taking a moving average of the time series as a low-pass filter). Multi-frequency transformations such as the moving average, with different window sizes, can be managed in such a way that the time series are of the same length. The transformed time series are then fed into their respective local convolutional branches. In the case of the multi-scale branch, the time series are treated with filters of the same size in order to capture wider ranges of features within the data. The convolutional layers are separated by max pooling layers with a very large stride and size, which is kept as a factor of of $\frac{n}{p}$ such that the output is p units long; this allows the model's filters to only focus on local features while also providing the model invariance towards spatial shifts. These local convolutional layers are concatenated vertically, then further convolutions occur in the full convolutional stage, followed by fully-connected layers, then the final activation layer (softmax in classification problems). MCNNs appear to be best equipped to handle data with numerical input, as opposed to one-hot input used in text-related problems. While it finds utility in time series, it can be extended to higher-dimensional analogues, as in the methodology found in the work done by Schlegl et al. (2015) to predict semantic descriptions from medical images. Instead of feeding the network two different patch sizes from a larger image, this paper explores implementing average pooling to the initial data to return a downsampled version of the data, closer to the approach found in Cui, Chen, and Chen (2016), but does not contain an analogue to that approach's fully convolutional stage.

Gradient descent methods all must face the challenge of successfully traversing saddle points. By increasing the learning rate, the model becomes more likely escape the saddle, and by subsequently decreasing the learning rate, it is prevented from overcorrecting and returning back into the saddle. Smith (2015) presents a method of varying the learning rate with a triangular waveform; with a half-cycle length of four epochs and scale of 4, it achieved comparable results to keeping a constant learning rate in less than 50% of the time taken.

Methodology



Start
'.../validation_data/'
'/d/Data/Leukemia/ /C-NMC_Leukemia/'
'.../training_data/fold_'+i+'/'+j/ i = {0,1,2}, j = {all,hem} 6 folders, 3 folds separated by class (all = cancer = 1, hem = no cancer = 0)
1867 Testing images 450x450 px RGB '.../C-NMC_test_prelim_phase_data/ {1:1867}.bmp'
10661 Training/Validation 450x450 px RGB .bmp Flename includes patient ID, cell number, cancer positive or negative
Preprocessing
Augment hy rotating image in increments of 360/n° n=1
Remove black background
CSV of testing labels (1867x3) 1. Suppressed filename. Includes patient ID, cell number, cancer positive or negative 2. filename (i.e. 1:1867.bmp) 3. y; 1 = cancer, 0 = no cancer
Crop to cell bounding box, pad to square, resize to kxk k=256
1867 kxk RGBA
Data Labeling
21322*n kxk RGBA
Augment by flipping over y
1867x3 CSV of Testing labels (Patient ID filepath, y)
21322*nx2 CSV of Training/Validation labels (full filepath, y)
Model Selection
For each model:
Model 1: 3x3 convolutions, linear, 256/dense layer
Subsample data with balanced classes (3200 each=6400 images), apply 7-2-1 split
Global Hyperparameters: Optimizer: Adam Callbacks: EarlyStopping(monitor = 'val_loss', mode='min',patience = 50, restore_best_weights=True)
Fit data with training set, Pause training with validation set, evaluate with test set
Model 2: 5x5 convolutions, linear, 256/dense layer
Model 4: 3x3 convolutions, 2 branches. 512/dense layer
Tested Hyperparameters: Batch size: 64 vs. 128 constant learning rate (1e-4 vs 2.5e-4) vs. LearningRateScheduler(triangleScheduler(stepsize=4,scale=4, start_at='max')) (min LR of 4e-5 vs. 1e-4)
Model 5: 3x3 convolutions, 5 branches. 512/dense layer
Set patient-level diagnostic threshold using % of cells classified as positive
Patient-level Diagnostic Evaluation
Choose best model based on cell-level test accuracy

Figure 1: Flowchart of image augmentation, transformation, and classification processes

Computer Information

Experiments were performed on Ubuntu 20.04 running on Windows Subsystem for Linux on an 8-core, 16-thread AMD Ryzen 3700X and NVIDIA RTX 2070 SUPER, with 16GB maximum virtual memory usage. The memory limit was quickly reached with large operations, so future experimentation on a virtual machine or on upgraded hardware should be carried out to extend upon the methods in this paper.

It should be noted that one should take care in using such a setup; during training, the Blue Screen of Death repeatedly occurred due to a `DPC_WATCHDOG_VIOLATION` with a video display glitch only visible if viewing the workstation screen directly. It is yet to be determined whether the issue is due to video memory constraints, GPU driver-Windows-WSL communication issues, or an unforeseen transient or chronic hardware issue.

Software

Python 3.7

Linear Algebra and Data: `NumPy`, `pandas`

Image Processing: `PIL`, `OpenCV`

Parallelization and Looping: `multiprocessing`, `joblib`, `itertools`

Classification and Evaluation: `tensorflow`, `scikit-learn`

Other Packages: `re`, `time`

`scikit-learn`'s native parallelization (and others!') does not work on Windows Subsystem for Linux. As a result, it is necessary to use the aforementioned parallelization packages in order to construct alternatives to parallelizable search algorithms such as `GridSearch`. If possible it is much more preferable to use RAPIDS because instead of dividing up the work between the CPU's cores, it can take advantage of increased computing speed on the GPU and GPU-system memory sharing while also only taking up one working process (i.e. parallelization is moot).

Preprocessing and Data Augmentation

The first step in the preprocessing method of the source images [2] was to remove the background; this was achieved by taking a binary threshold where pixel values that are black (i.e. RGB (0,0,0)) were given a transparency value of 100% (i.e RGBA (0,0,0,1)), and all other pixels were given a value of 0%. One concern about this process is the potential inclusion of black pixels within the cell creating, The next step was to augment the data by rotating the image counterclockwise about its center; n copies can be made by $360/n°$, but for the experiments in the paper $n = 1$ due to computational memory limitations. The image is then cropped to the bounding box of the cell, padded, and resized to a $k \times k, k = 256$ square [3], and duplicated copies of the training and validation images are created by flipping the image across the y-axis. At the end of this preprocessing and augmentation process, there are $21322n$ $k \times k$ RGBA images between the cell-level training , validation, and test sets, and 1867 $k \times k$ RGBA images in the patient-level test set. In this case, with $n = 36$ there are 523584 positively-classed cells and 244008 negatively-classed cells for a total of 767592 potential samples.

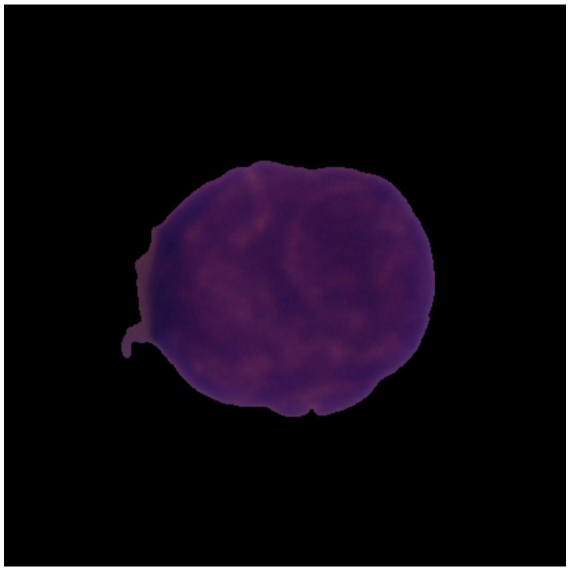


Figure 2: Original image from Patient 1 (leukemic) 450x450 RGB.

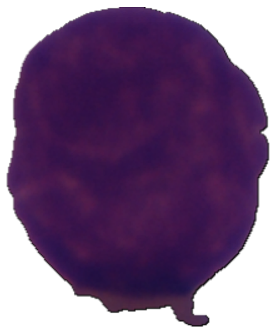


Figure 3: Leukemic cell with background removed, rotated 90 degrees, cropped to bounding box (206x252), then resized and padded to 256x256 RGBA.

Data Labeling

The folder structure of the original data was preserved and used to label the data in the new folder. The patient-level testing data's (found in `/validation_data/`) filenames, found in

`/validation_data/C-NMC_test_prelim_phase_data_labels.csv` were replaced with the `'Patient_ID'` cell-naming scheme instead of retaining the original names found in the `'new_names'` column in the aforementioned table.

Classification

From the data reservoir of 523584 positively-classed cells and 244008 negatively-classed cells, 3200 cells of each class are sampled without replacement to construct the model selection dataset. The data is split into a 7-2-1 split, with the training data setting the model parameters, the validation data's loss providing a training stopping mechanism, and the test data's accuracy providing a means of comparing model performance for all combinations of hyperparameters. A grid search of models, learning rates, learning policies, and batch sizes is carried out, though each learning rate policy has to be implemented in a separate grid search due to the increased difficulty of implementing model-specific and model-wide callbacks simultaneously.

Network Architectures

The first network architecture, Model 1, with 1 376 545 trainable parameters, consisted of successive combinations of two 3x3 convolutional layers followed by a 2x2 max pooling layer, with each pooling followed by a doubling of the number of convolutional filters. The 16x16x256 final convolutional layer receives a global average pooling, and then the pooled data is fed into two 256-neuron dense layers with 25% dropout, one final 256-neuron layer, then a sigmoid output layer. The second network architecture, Model 2, with 1 287 729 trainable parameters, replaces the two 3x3 convolutional layers with single 5x5 convolutional layers. The third network architecture, Model 4 (the original Model 3 was eliminated due to needing to construct further models to compare it), with 3 139 937 trainable parameters, consisted of two branches modeled after Model 1's convolutional portion, with one branch receiving an initial 2x2 average pooling step to replicate resizing the image to half its height and width instead of the dual-convolution-max-pooling combination. These two branches were concatenated after individually receiving global average pooling, and the dense stage of the network was identical to that of Model 1 and 2, but instead with 512 neurons per dense layer. The final tested network architecture, Model 5, with 2 963 169 trainable parameters, consisted of four branches (one keeping the original input, one receiving an initial 2x2 average pooling, another 4x4, and the last 8x8), using the same dual-convolution-max-pooling combination units, though the doubling of filter count did not occur for the final set of convolutional layers, resulting in the dense portion of the network having 512 neurons per layer. Full diagrams of the network architectures can be found in Appendix A, and the code for the network architectures can be found in Appendix B.

Global Hyperparameters

Global Hyperparameters:

- Loss Metric: `'binary_crossentropy'`
- Optimizer: `'Adam'`
- Callbacks: `EarlyStopping( monitor = 'val_loss', mode = 'min', patience = 50, restore_best_weights = True )`

The `EarlyStopping()` callback allows the model 50 epochs to improve on its best run according to its minimum validation loss, allowing the epoch count to be set to an arbitrarily large value (e.g. 4000); restoring the best weights ensures that it does not take the overfit solution that arises after failing to improve on validation loss. While this 50-epoch window is excessively generous, as most deviations between best solutions were fewer than 20 epochs apart, it allows the behavior of the model to be characterized as it attempts to improve its best fit.

Hyperparameter Selection

Tested Hyperparameters:

- Batch size: 64 vs. 128
- constant learning rate (1e-4 vs. 2.5e-4) vs. `LearningRateScheduler( triangleScheduler( stepsize = 4, scale = 4, start_at = 'max') )` (min LR of 4e-5 vs. 1e-4) corresponding to average of (1e-4 vs. 2.5e-4)

Table 3: Best hyperparameters by model and learning rate policy.

id	model	optimizer	min_lr	lr_policy	batch_size
model_1_A_r10n0_b2p1	model_1	Adam	1e-04	constant	128
model_2_A_r10n0_b2p0	model_2	Adam	1e-04	constant	64
model_4_A_r10n0_b2p0	model_4	Adam	1e-04	constant	64
model_5_A_r10n0_b2p0	model_5	Adam	1e-04	constant	64
model_1_tri_A_r10n0_b2p1	model_1_tri	Adam	4e-05	triangleScheduler(4,4,"max")	128
model_2_tri_A_r10n0_b2p1	model_2_tri	Adam	4e-05	triangleScheduler(4,4,"max")	128
model_4_tri_A_r10n1_b2p0	model_4_tri	Adam	1e-04	triangleScheduler(4,4,"max")	64
model_5_tri_A_r10n1_b2p0	model_5_tri	Adam	1e-04	triangleScheduler(4,4,"max")	64

Table 4: Best results by model and learning rate policy.

model	epochs	val_loss	time	tr_acc	tr_auc	vl_acc	vl_auc	ts_acc	ts_auc
model_1	131	0.3146	1778	0.9031	0.9610	0.8648	0.9387	0.8531	0.9275
model_2	100	0.2903	1059	0.9250	0.9770	0.8758	0.9485	0.8797	0.9468
model_4	94	0.2994	1889	0.9330	0.9778	0.8719	0.9456	0.8828	0.9398
model_5	84	0.2905	1975	0.9337	0.9782	0.8836	0.9471	0.8719	0.9367
model_1_tri	175	0.2965	2111	0.9214	0.9705	0.8766	0.9435	0.8625	0.9408
model_2_tri	155	0.2809	1273	0.9208	0.9761	0.8789	0.9547	0.8625	0.9404
model_4_tri	64	0.3186	1515	0.9051	0.9616	0.8609	0.9361	0.8562	0.9252
model_5_tri	42	0.3466	1411	0.9107	0.9646	0.8508	0.9232	0.8625	0.9250

Results

Model 4 with a constant learning rate policy performed best on the 640-cell sampled test set, though it only outperformed Model 2 by two correctly-classified cells; full results can be found in Appendix F. The effects of applying the cyclical learning rate policy appear to be rather inconclusive; while it occasionally reached convergence faster than the constant learning rate, sometimes the cyclical learning rate became a detriment, slowing convergence time. Epoch 94 achieved the minimum validation loss according to the chart in [4], at which point the training and validation loss had already separated, indicating that the model was drifting towards overfit parameters, further evidenced by the plateau in accuracy despite the increase in training loss. The model's bias was assessed to be minimal based on the comparable performance on negative and positive tasks; the confusion matrices show similar relative and absolute rates of occurrence of false positives and false negatives. The evolution of the loss metrics and accuracy values, and the training, validation, and test set confusion matrices can be found on the following pages.

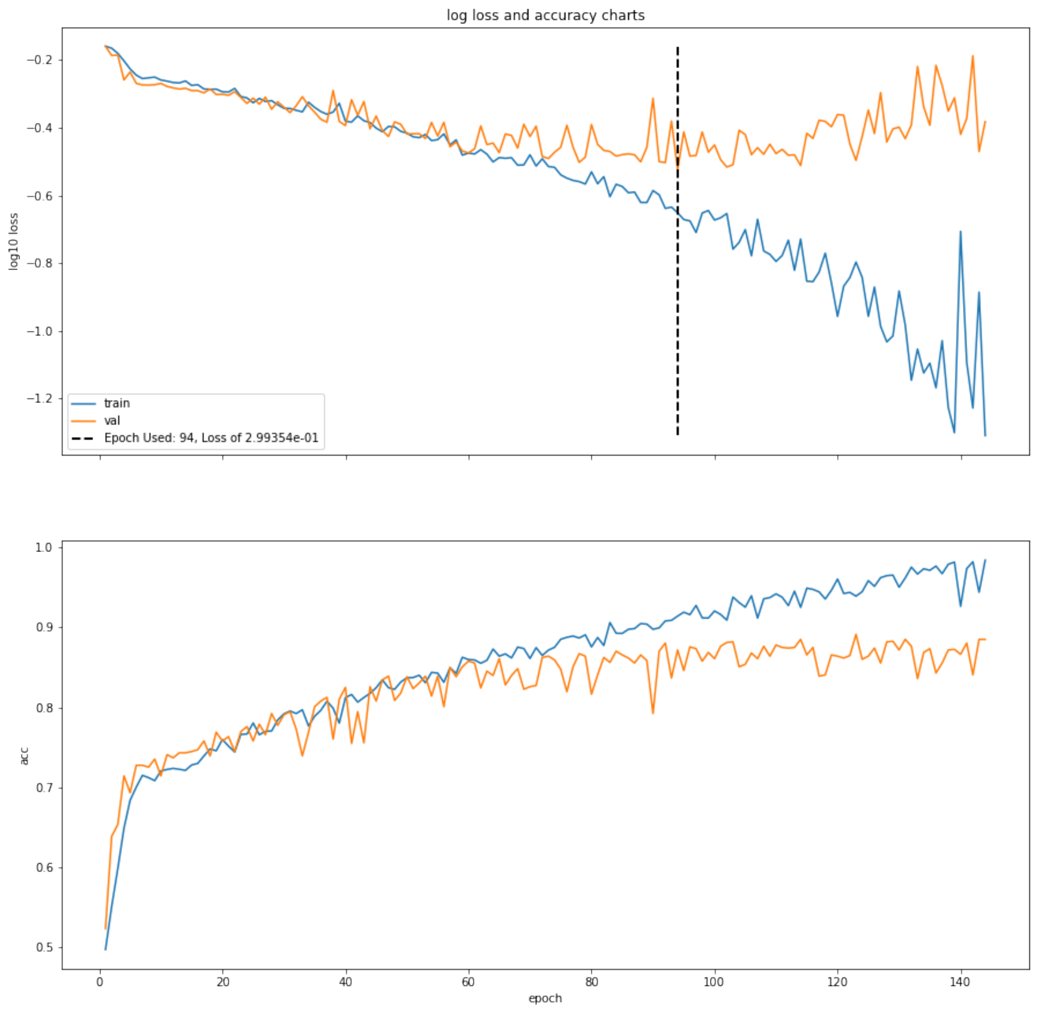


Figure 4: model_4 with Adam(1e-04), (min. learning rate) using a constant learning rate policy, and batch size of 64. Model saved in folder model_4_A_r10n0_b2p0

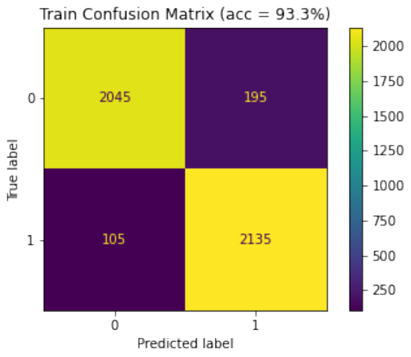
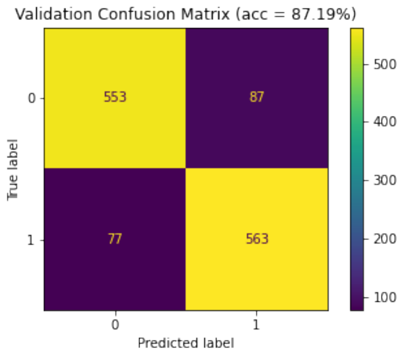


Figure 5: model_4 Training Accuracy: 93.3%



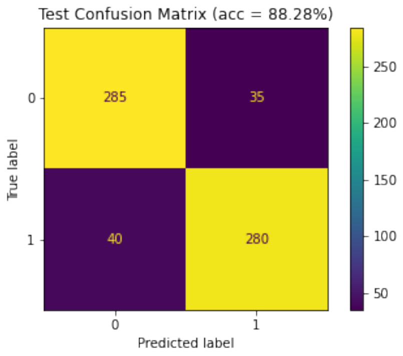


Figure 6: Validation Accuracy: 87.19%, Test Accuracy: 88.28%.

Application to Final Test Data

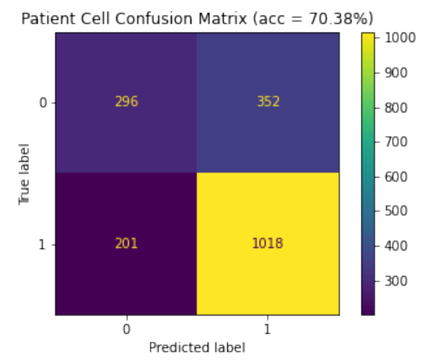


Figure 7: model_4 Patient-level Data Cell Accuracy: 70.38%

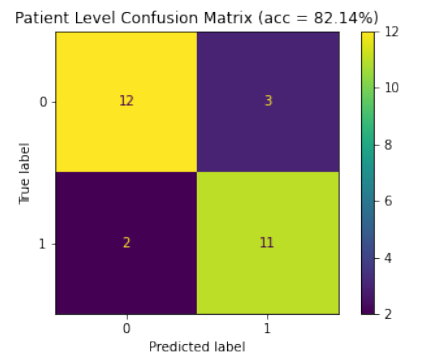


Figure 8: model_4 Patient-level Data Diagnosis Accuracy: 82.14%, (threshold of min. 69.28% of cells classed as positive)

The test data, like the original dataset, is imbalanced with more positive examples; the model performed well on the task of identifying the positive cells with a sensitivity of 83.51%, but poorly on the the task of identifying the negative cells with a specificity of 45.68%. Using the classifications of the test data, the proportion of positively-identified cells required to confer a positive diagnosis was determined to be optimal at $p_{\text{threshold}} = 69.28\%$, resulting in $23/28 = 82.14\%$ of patients being accurately classified.

Conclusions

There are plenty areas within my process that may have affected the accuracy of my classifier. There is a clear wide variation in cells within classes, due to the commonality of healthy-appearing cells within leukemic samples and the occasional appearance of cells with suspect characteristics within healthy samples. It is possible that the cyclical learning rate policy is interfering with the moment estimation process of the Adam optimizer, thus it is a logical next step to test a simpler optimizer such as vanilla stochastic gradient descent. Alternative learning policies such as those found in Smith (2018) or in one of the many simulated annealing approaches found in Nakamura et al. (2021) could resolve the convergence issues. Concatenating the layers and fully convolving them prior to final pooling as seen in the architecture of Cui, Chen, and Chen (2016) may provide another improvement to performance. Additionally, it may be worth emulating features of AlexNet such as using a small number of large kernels in the early stages of the model instead of keeping a consistent convolutional kernel size or using pooling layers with a stride value smaller than the pooling width. Alternatively, a transfer learning approach could be applied, where an autoencoder implementing global pooling forms the basis of the convolutional layers, then the encoded outputs are connected to a fully-connected classifier. Furthermore, there are necessary improvements to my methodologies that should fall into practices like those seen in Smith (2015) and Smith (2018).

References

Bain, B. J., and L. Estcourt. 2013. “FAB Classification of Leukemia.” In *Brenner’s Encyclopedia of Genetics (Second Edition)*, edited by Stanley Maloy and Kelly Hughes, Second Edition, 5–7. San Diego: Academic Press. https://doi.org/https://doi.org/10.1016/B978-0-12-374984-0.00515-5.

Cui, Zhicheng, Wenlin Chen, and Yixin Chen. 2016. “Multi-Scale Convolutional Neural Networks for Time Series Classification.” https://arxiv.org/abs/1603.06995.

Di Ruberto, Cecilia, and Lorenzo Putzu. 2013. “White Blood Cells Identification and Counting from Microscopic Blood Image.” *International Journal of Medical, Health, Biomedical and Pharmaceutical Engineering* 7 (January): 15–22.

Gupta, Anubha, and Ritu Gupta. 2019. “C_NMC_2019 Dataset ISBI 2019 ALL Challenge.” *The Cancer Imaging Archive*. https://doi.org/10.7937/tcia.2019.dc64i46r.

Islam, Md. Amirul, Matthew Kowal, Sen Jia, Konstantinos G. Derpanis, and Neil D. B. Bruce. 2021. “Global Pooling, More Than Meets the Eye: Position Information Is Encoded Channel-Wise in CNNs.” *CoRR* abs/2108.07884. https://arxiv.org/abs/2108.07884.

Mishra, Sonali, Banshidhar Majhi, and Pankaj Kumar Sa. 2019. “Texture Feature Based Classification on Microscopic Blood Smear for Acute Lymphoblastic Leukemia Detection.” *Biomedical Signal Processing and Control* 47: 303–11. https://doi.org/https://doi.org/10.1016/j.bspc.2018.08.012.

Mishra, Sonali, Lokesh Sharma, Bansidhar Majhi, and Pankaj Kumar Sa. 2017. “Microscopic Image Classification Using DCT for the Detection of Acute Lymphoblastic Leukemia (ALL).” In *Proceedings of International Conference on Computer Vision and Image Processing*, edited by Balasubramanian Raman, Sanjeev Kumar, Partha Pratim Roy, and Debashis Sen, 171–80. Singapore: Springer Singapore.

Mohapatra, Subrajeet, Dipti Patra, and Sanghamitra Satpathy. 2014. “An Ensemble Classifier System for Early Diagnosis of Acute Lymphoblastic Leukemia in Blood Microscopic Images.” *Neural Computing and Applications* 24 (7): 1887–1904.

Nakamura, Kensuke, Bilel Derbel, Kyoung-Jae Won, and Byung-Woo Hong. 2021. “Learning-Rate Annealing Methods for Deep Neural Networks.” *Electronics* 10 (16). https://doi.org/10.3390/electronics10162029.

Putzu, Lorenzo, Giovanni Caocci, and Cecilia Di Ruberto. 2014. “Leucocyte Classification for Leukaemia Detection Using Image Processing Techniques.” *Artificial Intelligence in Medicine* 62 (3): 179–91. https://doi.org/https://doi.org/10.1016/j.artmed.2014.09.002.

Rawat, Jyoti, Annapurna Singh, H. S. Bhadauria, and Jitendra Virmani. 2015. “Computer Aided Diagnostic System for Detection of Leukemia Using Microscopic Images.” *Procedia Computer Science* 70: 748–56. https://doi.org/https://doi.org/10.1016/j.procs.2015.10.113.

Schlegl, Thomas, Sebastian Waldstein, Wolf-Dieter Vogl, Ursula Schmidt-Erfurth, and Georg Langs. 2015. “Predicting Semantic Descriptions from Medical Images with Convolutional Neural Networks.” In *Information Processing in Medical Imaging : Proceedings of the ... Conference*, 24:437–48. https://doi.org/10.1007/978-3-319-19992-4_34.

Smith, Leslie N. 2015. “Cyclical Learning Rates for Training Neural Networks.” arXiv. https://doi.org/10.48550/ARXIV.1506.01186.

———. 2018. “A Disciplined Approach to Neural Network Hyper-Parameters: Part 1 – Learning Rate, Batch Size, Momentum, and Weight Decay.” arXiv. https://doi.org/10.48550/ARXIV.1803.09820.

Tuba, Eva, Ivana Strumberger, Nebojsa Bacanin, Dejan Zivkovic, and Milan Tuba. 2019. “Acute Lymphoblastic Leukemia Cell Detection in Microscopic Digital Images Based on Shape and Texture Features.” In *Advances in Swarm Intelligence*, edited by Ying Tan, Yuhui Shi, and Ben Niu, 142–51. Cham: Springer International Publishing.

Appendix A: Model Architecture Diagrams

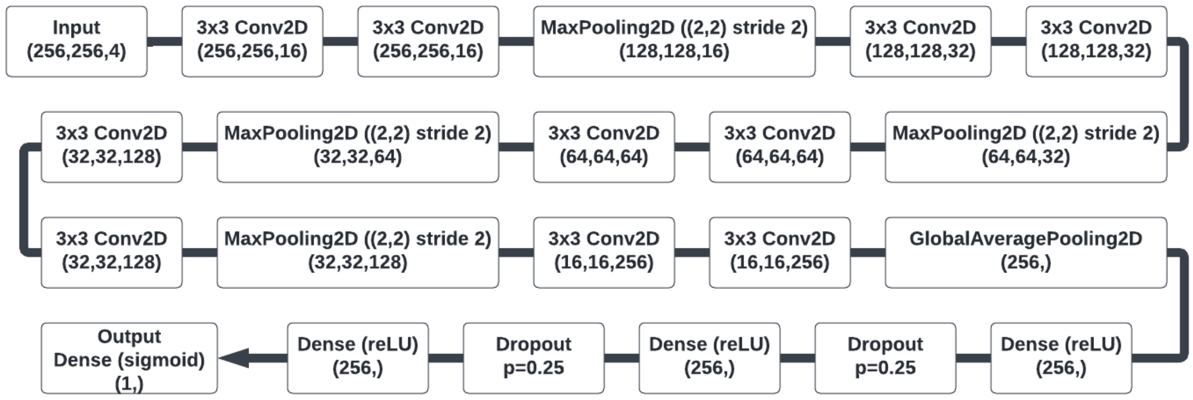


Figure 9: Flowchart of Model 1

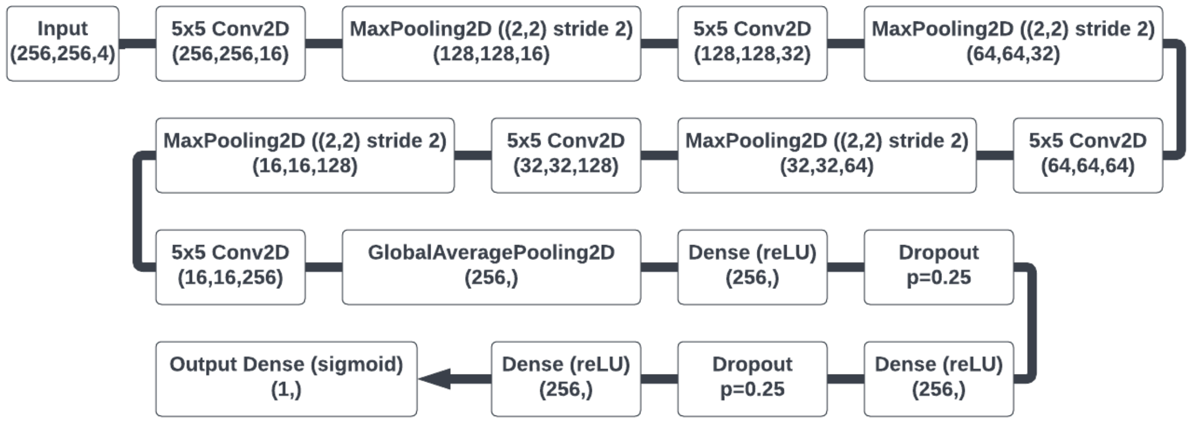

Input (256,256,4)
5x5 Conv2D (256,256,16)
MaxPooling2D ((2,2) stride 2) (128,128,16)
5x5 Conv2D (128,128,32)
MaxPooling2D ((2,2) stride 2) (64,64,32)
5x5 Conv2D (64,64,64)
MaxPooling2D ((2,2) stride 2) (32,32,64)
5x5 Conv2D (32,32,128)
MaxPooling2D ((2,2) stride 2) (16,16,128)
5x5 Conv2D (16,16,256)
GlobalAveragePooling2D (256,)
Dense (reLU) (256,)
Dropout p=0.25
Dense (reLU) (256,)
Dropout p=0.25
Dense (reLU) (256,)
Output Dense (sigmoid) (1,)

Figure 10: Flowchart of Model 2

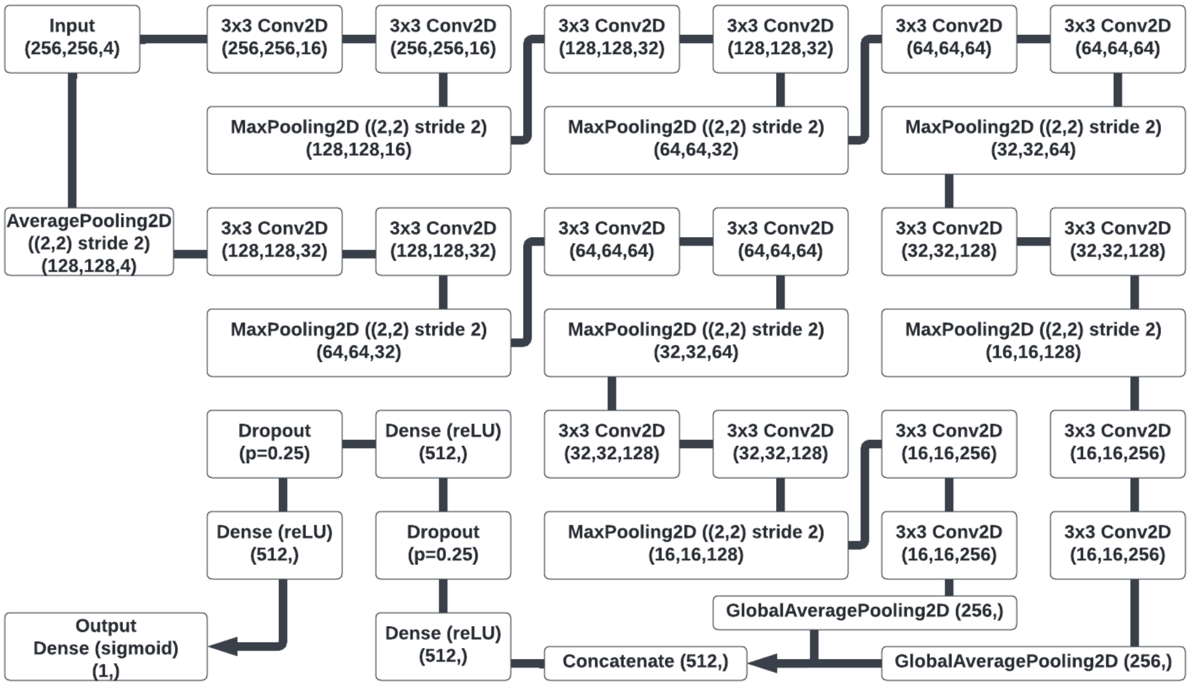


Figure 11: Flowchart of Model 4

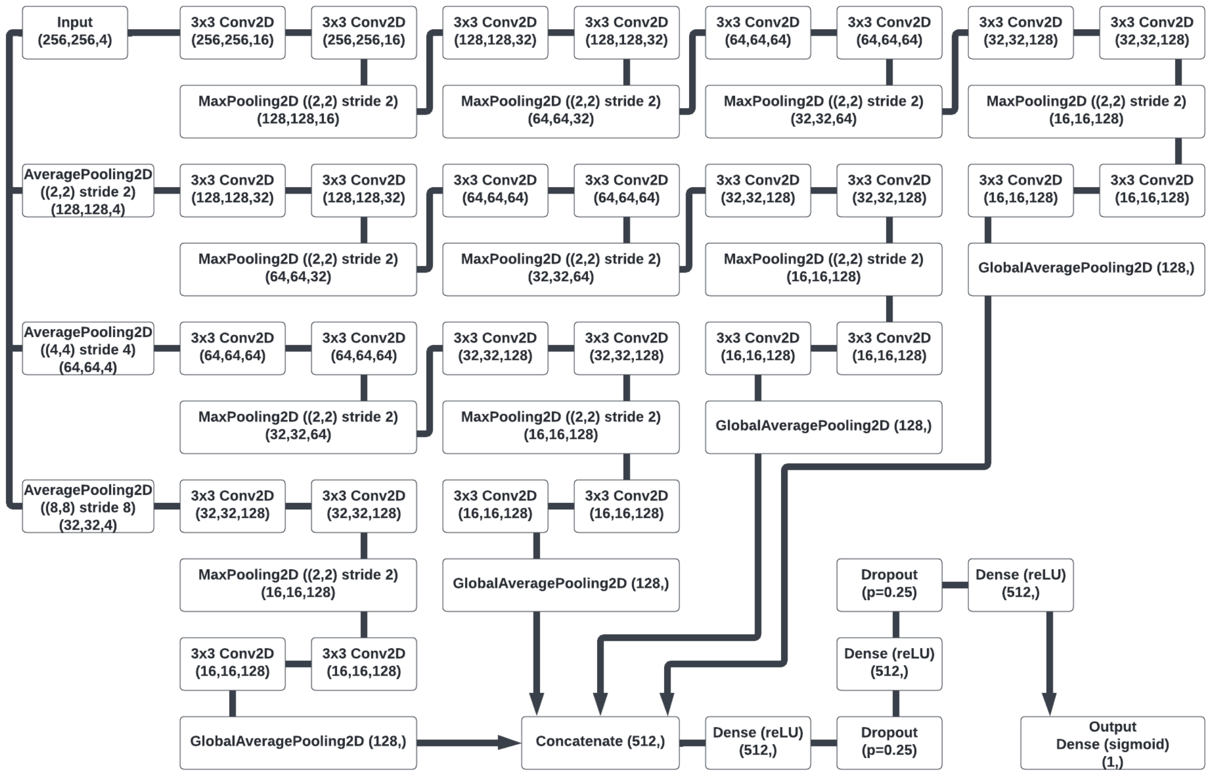


Figure 12: Flowchart of Model 5

Appendix B: Network Architecture Code

Required Packages

```
import tensorflow as tf
import tensorflow.keras as keras
import tensorflow.keras.backend as K
import sklearn.model_selection
from tensorflow.keras import layers, Model
from tensorflow.keras.utils import to_categorical, plot_model
from tensorflow.keras.callbacks import EarlyStopping, LearningRateScheduler, ReduceLROnPlateau
from tensorflow.keras.optimizers import Adam, SGD, RMSprop
from tensorflow.keras.layers import Dense, Flatten, Dropout, Reshape, Input
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Concatenate
from tensorflow.keras.layers import AveragePooling2D, GlobalAveragePooling2D
from tensorflow.keras.models import Sequential, Model, load_model
from tensorflow.keras.metrics import AUC
```

Model 1

```
inp = Input((256,256,4))
x0 = Conv2D(16, (3,3), activation='relu', padding='same', name="branch_0_c0a")(inp) #(256,256,16)
x0 = Conv2D(16, (3,3), activation='relu', padding='same', name="branch_0_c0b")(x0)  #(256,256,16)
x0 = MaxPooling2D((2,2), name="branch_0_mp0")(x0) #(128,128,16)
x0 = Conv2D(32, (3,3), activation='relu', padding='same', name="branch_0_c1a")(x0)  #(128,128,32)
x0 = Conv2D(32, (3,3), activation='relu', padding='same', name="branch_0_c1b")(x0)  #(128,128,32)
x0 = MaxPooling2D((2,2), name="branch_0_mp1")(x0) #(64,64,32)
x0 = Conv2D(64, (3,3), activation='relu', padding='same', name="branch_0_c2a")(x0)  #(64,64,64)
x0 = Conv2D(64, (3,3), activation='relu', padding='same', name="branch_0_c2b")(x0)  #(64,64,64)
x0 = MaxPooling2D((2,2), name="branch_0_mp2")(x0) #(32,32,64)
x0 = Conv2D(128, (3,3), activation='relu', padding='same', name="branch_0_c3a")(x0)  #(32,32,128)
x0 = Conv2D(128, (3,3), activation='relu', padding='same', name="branch_0_c3b")(x0)  #(32,32,128)
x0 = MaxPooling2D((2,2), name="branch_0_mp3")(x0) #(16,16,128)
x0 = Conv2D(256, (3,3), activation='relu', padding='same', name="branch_0_c4a")(x0)  #(16,16,256)
x0 = Conv2D(256, (3,3), activation='relu', padding='same', name="branch_0_c4b")(x0)  #(16,16,256)
m0 = GlobalAveragePooling2D(name="branch_0_gap")(x0) #(256,)
x = Dense(256,activation='relu',name='dense_0')(m0)
x = Dropout(0.25,name='drop_0')(x)
x = Dense(256,activation='relu',name='dense_1')(x)
x = Dropout(0.25,name='drop_1')(x)
x = Dense(256,activation='relu',name='dense_2')(x)
outputs_1 = Dense(1,activation='sigmoid',name='dense_final')(x)
model_1 = Model(inp, outputs_1, name="model_1")
model_1_tri = Model(inp, outputs_1, name="model_1_tri")
model_1.summary()
plot_model(model_1, to_file='model_1_plot.png', show_shapes=True, show_layer_names=True)
```

Model 2

```
inp = Input((256,256,4))
x0 = Conv2D(16, (5,5), activation='relu', padding='same', name="branch_0_c0a")(inp) #(256,256,16)
x0 = MaxPooling2D((2,2), name="branch_0_mp0")(x0) #(128,128,16)
```

```
x0 = Conv2D(32, (5,5), activation='relu', padding='same', name="branch_0_c1a")(x0)  #(128,128,32)
x0 = MaxPooling2D((2,2), name="branch_0_mp1")(x0) #(64,64,32)
x0 = Conv2D(64, (5,5), activation='relu', padding='same', name="branch_0_c2a")(x0)  #(64,64,64)
x0 = MaxPooling2D((2,2), name="branch_0_mp2")(x0) #(32,32,64)
x0 = Conv2D(128, (5,5), activation='relu', padding='same', name="branch_0_c3a")(x0)  #(32,32,128)
x0 = MaxPooling2D((2,2), name="branch_0_mp3")(x0) #(16,16,128)
x0 = Conv2D(256, (5,5), activation='relu', padding='same', name="branch_0_c4a")(x0)  #(16,16,256)
m0 = GlobalAveragePooling2D(name="branch_0_gap")(x0) #(256,)
#m0 = Flatten()(m0)
#m1 = Flatten()(m1)
#m2 = Flatten()(m2)
#m3 = Flatten()(m3)
#m4 = Flatten()(m4)
#x = Concatenate()([m0,m1,m2,m3,m4])
x = Dense(256,activation='relu',name='dense_0')(m0)
x = Dropout(0.25,name='drop_0')(x)
x = Dense(256,activation='relu',name='dense_1')(x)
x = Dropout(0.25,name='drop_1')(x)
x = Dense(256,activation='relu',name='dense_2')(x)
outputs_2 = Dense(1,activation='sigmoid',name='dense_final')(x)
model_2 = Model(inp, outputs_2, name="model_2")
model_2_tri = Model(inp, outputs_2, name="model_2_tri")
model_2.summary()
plot_model(model_2, to_file='model_2_plot.png', show_shapes=True, show_layer_names=True)
```

Model 4

```
inp = Input((256,256,4))

x0 = Conv2D(16, (3,3), activation='relu', padding='same', name="branch_0_c0a")(inp) #(256,256,16)
x0 = Conv2D(16, (3,3), activation='relu', padding='same', name="branch_0_c0b")(x0)  #(256,256,16)
x0 = MaxPooling2D((2,2), name="branch_0_mp0")(x0) #(128,128,16)
x0 = Conv2D(32, (3,3), activation='relu', padding='same', name="branch_0_c1a")(x0)  #(128,128,32)
x0 = Conv2D(32, (3,3), activation='relu', padding='same', name="branch_0_c1b")(x0)  #(128,128,32)
x0 = MaxPooling2D((2,2), name="branch_0_mp1")(x0) #(64,64,32)
x0 = Conv2D(64, (3,3), activation='relu', padding='same', name="branch_0_c2a")(x0)  #(64,64,64)
x0 = Conv2D(64, (3,3), activation='relu', padding='same', name="branch_0_c2b")(x0)  #(64,64,64)
x0 = MaxPooling2D((2,2), name="branch_0_mp2")(x0) #(32,32,64)
x0 = Conv2D(128, (3,3), activation='relu', padding='same', name="branch_0_c3a")(x0)  #(32,32,128)
x0 = Conv2D(128, (3,3), activation='relu', padding='same', name="branch_0_c3b")(x0)  #(32,32,128)
x0 = MaxPooling2D((2,2), name="branch_0_mp3")(x0) #(16,16,128)
x0 = Conv2D(256, (3,3), activation='relu', padding='same', name="branch_0_c4a")(x0)  #(16,16,256)
x0 = Conv2D(256, (3,3), activation='relu', padding='same', name="branch_0_c4b")(x0)  #(16,16,256)
m0 = GlobalAveragePooling2D(name="branch_0_gap")(x0) #(256,)

x1 = AveragePooling2D((2,2),name='branch_1')(inp) #(128,128,4)
x1 = Conv2D(32, (3,3), activation='relu', padding='same', name="branch_1_c1a")(x1)  #(128,128,32)
x1 = Conv2D(32, (3,3), activation='relu', padding='same', name="branch_1_c1b")(x1)  #(128,128,32)
x1 = MaxPooling2D((2,2), name="branch_1_mp1")(x1) #(64,64,32)
x1 = Conv2D(64, (3,3), activation='relu', padding='same', name="branch_1_c2a")(x1)  #(64,64,64)
x1 = Conv2D(64, (3,3), activation='relu', padding='same', name="branch_1_c2b")(x1)  #(64,64,64)
x1 = MaxPooling2D((2,2), name="branch_1_mp2")(x1) #(32,32,64)
x1 = Conv2D(128, (3,3), activation='relu', padding='same', name="branch_1_c3a")(x1)  #(32,32,128)
```

```
x1 = Conv2D(128, (3,3), activation='relu', padding='same', name="branch_1_c3b")(x1)  #(32,32,128)
x1 = MaxPooling2D((2,2), name="branch_1_mp3")(x1) #(16,16,128)
x1 = Conv2D(256, (3,3), activation='relu', padding='same', name="branch_1_c4a")(x1)  #(16,16,256)
x1 = Conv2D(256, (3,3), activation='relu', padding='same', name="branch_1_c4b")(x1)  #(16,16,256)
m1 = GlobalAveragePooling2D(name="branch_1_gap")(x1) #(256,)


x = Concatenate(name='cat')([m0,m1])
x = Dense(512,activation='relu',name='dense_0')(x)
x = Dropout(0.25,name='drop_0')(x)
x = Dense(512,activation='relu',name='dense_1')(x)
x = Dropout(0.25,name='drop_1')(x)
x = Dense(512,activation='relu',name='dense_2')(x)
outputs_4 = Dense(1,activation='sigmoid',name='dense_final')(x)
model_4 = Model(inp, outputs_4, name="model_4")
model_4_tri = Model(inp, outputs_4, name="model_4_tri")
model_4.summary()
plot_model(model_4, to_file='model_4_plot.png', show_shapes=True, show_layer_names=True)
```

Model 5

```
inp = Input((256,256,4))

x0 = Conv2D(16, (3,3), activation='relu', padding='same', name="branch_0_c0a")(inp) #(256,256,16)
x0 = Conv2D(16, (3,3), activation='relu', padding='same', name="branch_0_c0b")(x0)  #(256,256,16)
x0 = MaxPooling2D((2,2), name="branch_0_mp0")(x0) #(128,128,16)
x0 = Conv2D(32, (3,3), activation='relu', padding='same', name="branch_0_c1a")(x0)  #(128,128,32)
x0 = Conv2D(32, (3,3), activation='relu', padding='same', name="branch_0_c1b")(x0)  #(128,128,32)
x0 = MaxPooling2D((2,2), name="branch_0_mp1")(x0) #(64,64,32)
x0 = Conv2D(64, (3,3), activation='relu', padding='same', name="branch_0_c2a")(x0)  #(64,64,64)
x0 = Conv2D(64, (3,3), activation='relu', padding='same', name="branch_0_c2b")(x0)  #(64,64,64)
x0 = MaxPooling2D((2,2), name="branch_0_mp2")(x0) #(32,32,64)
x0 = Conv2D(128, (3,3), activation='relu', padding='same', name="branch_0_c3a")(x0)  #(32,32,128)
x0 = Conv2D(128, (3,3), activation='relu', padding='same', name="branch_0_c3b")(x0)  #(32,32,128)
x0 = MaxPooling2D((2,2), name="branch_0_mp3")(x0) #(16,16,128)
x0 = Conv2D(128, (3,3), activation='relu', padding='same', name="branch_0_c4a")(x0)  #(16,16,128)
x0 = Conv2D(128, (3,3), activation='relu', padding='same', name="branch_0_c4b")(x0)  #(16,16,128)
m0 = GlobalAveragePooling2D(name="branch_0_gap")(x0) #(128,)

x1 = AveragePooling2D((2,2),name='branch_1')(inp) #(128,128,4)
x1 = Conv2D(32, (3,3), activation='relu', padding='same', name="branch_1_c0a")(x1)  #(128,128,32)
x1 = Conv2D(32, (3,3), activation='relu', padding='same', name="branch_1_c0b")(x1)  #(128,128,32)
x1 = MaxPooling2D((2,2), name="branch_1_mp1")(x1) #(64,64,32)
x1 = Conv2D(64, (3,3), activation='relu', padding='same', name="branch_1_c1a")(x1)  #(64,64,64)
x1 = Conv2D(64, (3,3), activation='relu', padding='same', name="branch_1_c1b")(x1)  #(64,64,64)
x1 = MaxPooling2D((2,2), name="branch_1_mp2")(x1) #(32,32,64)
x1 = Conv2D(128, (3,3), activation='relu', padding='same', name="branch_1_c2a")(x1)  #(32,32,128)
x1 = Conv2D(128, (3,3), activation='relu', padding='same', name="branch_1_c2b")(x1)  #(32,32,128)
x1 = MaxPooling2D((2,2), name="branch_1_mp3")(x1) #(16,16,128)
x1 = Conv2D(128, (3,3), activation='relu', padding='same', name="branch_1_c3a")(x1)  #(16,16,128)
x1 = Conv2D(128, (3,3), activation='relu', padding='same', name="branch_1_c3b")(x1)  #(16,16,128)
m1 = GlobalAveragePooling2D(name="branch_1_gap")(x1) #(128,)
```

```
x2 = AveragePooling2D((4,4),name='branch_2')(inp) #(64,64,4)
x2 = Conv2D(64, (3,3), activation='relu', padding='same', name="branch_2_c0a")(x2)  #(64,64,64)
x2 = Conv2D(64, (3,3), activation='relu', padding='same', name="branch_2_c0b")(x2)  #(64,64,64)
x2 = MaxPooling2D((2,2), name="branch_2_mp2")(x2) #(32,32,64)
x2 = Conv2D(128, (3,3), activation='relu', padding='same', name="branch_2_c1a")(x2)  #(32,32,128)
x2 = Conv2D(128, (3,3), activation='relu', padding='same', name="branch_2_c1b")(x2)  #(32,32,128)
x2 = MaxPooling2D((2,2), name="branch_2_mp3")(x2) #(16,16,128)
x2 = Conv2D(128, (3,3), activation='relu', padding='same', name="branch_2_c2a")(x2)  #(16,16,128)
x2 = Conv2D(128, (3,3), activation='relu', padding='same', name="branch_2_c2b")(x2)  #(16,16,128)
m2 = GlobalAveragePooling2D(name="branch_2_gap")(x2) #(128,)

x3 = AveragePooling2D((8,8),name='branch_3')(inp) #(32,32,4)
x3 = Conv2D(128, (3,3), activation='relu', padding='same', name="branch_3_c0a")(x3)  #(32,32,128)
x3 = Conv2D(128, (3,3), activation='relu', padding='same', name="branch_3_c0b")(x3)  #(32,32,128)
x3 = MaxPooling2D((2,2), name="branch_3_mp3")(x3) #(16,16,128)
x3 = Conv2D(128, (3,3), activation='relu', padding='same', name="branch_3_c1a")(x3)  #(16,16,128)
x3 = Conv2D(128, (3,3), activation='relu', padding='same', name="branch_3_c1b")(x3)  #(16,16,128)
m3 = GlobalAveragePooling2D(name="branch_3_gap")(x3) #(128,)


x = Concatenate(name='cat')([m0,m1,m2,m3])
x = Dense(512,activation='relu',name='dense_0')(x)
x = Dropout(0.25,name='drop_0')(x)
x = Dense(512,activation='relu',name='dense_1')(x)
x = Dropout(0.25,name='drop_1')(x)
x = Dense(512,activation='relu',name='dense_2')(x)
outputs_5 = Dense(1,activation='sigmoid',name='dense_final')(x)
model_5 = Model(inp, outputs_5, name="model_5")
model_5_tri = Model(inp, outputs_5, name="model_5_tri")
model_5.summary()
plot_model(model_5, to_file='model_5_plot.png', show_shapes=True, show_layer_names=True)
```

Learning Rate Scheduler

```
def triangleScheduler(stepsize=4,scale=4,start_at='min'):
    def angler(epoch,lr):
        k = tf.cast(stepsize,'int32')
        s = tf.cast(scale,'float32')
        ep = tf.cast(epoch,'int32') # Epoch
        if ep == 0:
            lr0 = lr
            xq = pd.DataFrame({'lr0':lr0},index=[0])
            xq.to_csv('lr0.csv',index=False)
            if start_at !='max':
                lrf = lr0
            else:
                lrf = lr0*scale
        else:
            px = pd.read_csv('lr0.csv')
            lr0=px.values[0][0]
            if start_at !='max':
                lrf = lr0*(1+(scale-1)*-np.cumsum(
                np.concatenate([(1+np.zeros(stepsize)),(-1+np.zeros(stepsize))])/(stepsize))
                )[(ep-1) % (2*stepsize)]
        else:
            lrf = (lr0*scale-lr0*(scale-1)*np.cumsum(
            np.concatenate([(1+np.zeros(stepsize)),(-1+np.zeros(stepsize))])/(stepsize))
                )[(ep-1) % (2*stepsize)]
    def round_it(x, sig):
        return round(x, sig-int(np.floor(np.log10(abs(x))))-1)
    return round_it(lrf,5)
return angler
```

Appendix C: Data Generation Code (ran once)

Required Packages

```
import sklearn
import cv2
from IPython.display import display, Image
import numpy as np
import PIL
import multiprocessing
from joblib import Parallel, delayed
import os
from os import listdir
from os.path import isfile, join
import itertools as it
import pandas as pd
import re
import skimage
import skimage.color
import matplotlib
import matplotlib.pyplot as plt
import pandas as pd
import time
import matplotlib
import matplotlib.pyplot as plt
```

Preprocessing

Background Remover

Source

```
file_name = '/mnt/d/Data/Leukemia/C-NMC_Leukemia/training_data/fold_0/all/UID_1_1_1_all.bmp'
src = cv2.imread(file_name, 1)
cv2.imwrite('/mnt/d/Data/LeukemiaV2/rfiles/source.png',src)
def alpharm(img):
    tmp = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    _,alpha = cv2.threshold(tmp,0,255,cv2.THRESH_BINARY)
    b, g, r = cv2.split(img)
    rgba = [b,g,r, alpha]
    dst = cv2.merge(rgba,4)
    return dst
ds1 = alpharm(src)
cv2.imwrite('/mnt/d/Data/LeukemiaV2/rfiles/test0.png', ds1)
file_next1 = '/mnt/d/Data/LeukemiaV2/rfiles/test0.png'
src1 = cv2.imread(file_next1, 1)
```

Image Rotation

Source

```
def getTranslationMatrix2d(dx, dy):
    return np.matrix([[1, 0, dx], [0, 1, dy], [0, 0, 1]])
```

```
def rotateImage(image, angle):

    image_size = (image.shape[1], image.shape[0])
    image_center = tuple(np.array(image_size) / 2)
    rot_mat = np.vstack(
    [cv2.getRotationMatrix2D(image_center, angle, 1.0), [0, 0, 1]]
    )
    trans_mat = np.identity(3)
    w2 = image_size[0] * 0.5
    h2 = image_size[1] * 0.5
    rot_mat_notranslate = np.matrix(rot_mat[0:2, 0:2])
    tl = (np.array([-w2, h2]) * rot_mat_notranslate).A[0]
    tr = (np.array([w2, h2]) * rot_mat_notranslate).A[0]
    bl = (np.array([-w2, -h2]) * rot_mat_notranslate).A[0]
    br = (np.array([w2, -h2]) * rot_mat_notranslate).A[0]
    x_coords = [pt[0] for pt in [tl, tr, bl, br]]
    x_pos = [x for x in x_coords if x > 0]
    x_neg = [x for x in x_coords if x < 0]
    y_coords = [pt[1] for pt in [tl, tr, bl, br]]
    y_pos = [y for y in y_coords if y > 0]
    y_neg = [y for y in y_coords if y < 0]
    right_bound = max(x_pos)
    left_bound = min(x_neg)
    top_bound = max(y_pos)
    bot_bound = min(y_neg)
    new_w = int(abs(right_bound - left_bound))
    new_h = int(abs(top_bound - bot_bound))
    new_image_size = (new_w, new_h)
    new_midx = new_w * 0.5
    new_midy = new_h * 0.5
    dx = int(new_midx - w2)
    dy = int(new_midy - h2)
    trans_mat = getTranslationMatrix2d(dx, dy)
    affine_mat = (
    np.matrix(trans_mat) * np.matrix(rot_mat)
    )[0:2, :]
    result = cv2.warpAffine(image, affine_mat,
    new_image_size, flags=cv2.INTER_LINEAR)
    return result

img2 = alpharm(rotateImage(src1,90))
file_next2 = '/mnt/d/Data/LeukemiaV2/rfiles/test.png'
cv2.imwrite(file_next2, img2)
display(Image(filename=file_next2))
```

Autocropper

Autocrop Image

```
img3=PIL.Image.open(file_next2)
def cropDustA(img): # Autocrop
    imageBox = img.getbbox()
    img = img.crop(imageBox)
    return img
img3 = cropDustA(img3)
file_next3 = '/mnt/d/Data/LeukemiaV2/rfiles/test2.png'
img3.save(file_next3)
```

Make Square

Source

```
def make_square(im, resize=256,min_size=1, fill_color=(0, 0, 0, 0)): # Resize
    x, y = im.size
    size = max(min_size, x, y)
    new_im = PIL.Image.new('RGBA', (size, size), fill_color)
    new_im.paste(im, (int((size - x) / 2), int((size - y) / 2)))
    new_im = new_im.resize((resize,resize))
    return new_im
make_square(img3,resize=64).save(
'/mnt/d/Data/LeukemiaV2/rfiles/test3.png'
)
```

Master Recrop

```
def masterRecrop(img_path,out_path,deg,size,flip=False):
    file_name = img_path # import image
    src = cv2.imread(file_name, 1)
    ds1 = alpharm(src)
    #cv2.imwrite(
    '/mnt/d/Data/LeukemiaV2/rfiles/test.png', ds1
    )
    #src1 = cv2.imread(file_next1, 1)
    img2 = alpharm(rotateImage(src,deg))
    cv2.imwrite(out_path, img2)
    img3=PIL.Image.open(out_path)
    img3 = cropDustA(img3)
    img3 = make_square(img3,resize=size)
    if flip == True:
        img3 = img3.transpose(PIL.Image.FLIP_LEFT_RIGHT)
    img3.save(out_path)
```

Data Augmentation

```
myFolder = [x[0] for x in os.walk('/mnt/d/Data/Leukemia/C-NMC_Leukemia/')]
for j in [5,6,8,9,11,12]: #[5,6,8,9,11,12]
    inputs = 10*np.arange(36)
    onlyfiles = [f for f in listdir(myFolder[j]) if isfile(join(myFolder[j], f))]
    filelist = pd.Series(onlyfiles).apply(lambda x: myFolder[j]+'/'+str(x))
    print(filelist[0])
    for files in filelist:
        outpart = str(
        pd.Series(files).str.replace('C\-NMC_Leukemia','Cleaned',regex=True
        ).str.replace('\.bmp$','',regex=True)[0])
        #print((files,outpart))
        if __name__ == "__main__":
            processed_list = Parallel(n_jobs=-1, prefer='threads')(delayed(masterRecrop)(
                img_path = files,
                     out_path = outpart+'_r_'+str(k)+'.png',
                     deg = k,size = 256,flip = False)
                                                         for k in inputs)
        if __name__ == "__main__":
            processed_list = Parallel(n_jobs=-1, prefer='threads')(delayed(masterRecrop)(
                img_path = files,
                     out_path = outpart+'_l_'+str(k)+'.png',
                     deg = k,size = 256,flip = True)
                                                         for k in inputs)
```

Data Labeling

```
def lblMyTrain(path, index, oneStr):
  myCleanFolder = [x[0] for x in os.walk(path)]
  myFiles = pd.Series()
  myTrY = pd.Series()
  myLabel = pd.Series(int(bool(re.search(oneStr,myCleanFolder[index]))))
  onlyCF = [f for f in listdir(myCleanFolder[index]) if isfile(join(myCleanFolder[index], f))]
  fList = pd.Series(onlyCF).apply(lambda x: myCleanFolder[index]+'/'+str(x))
  myClass = myLabel.repeat(len(fList))
  myFiles = myFiles.append(fList).reset_index()
  myTrY = myTrY.append(myClass).reset_index()
  return pd.concat([myFiles, myTrY], axis=1)
folderInd = [5,6,8,9,11,12]
if __name__ == "__main__":
            processed_list = Parallel(n_jobs=-1, prefer='threads')(delayed(lblMyTrain)(
                path = '/mnt/d/Data/Leukemia/Cleaned/',
                     index = k, oneStr = 'all')
                                                         for k in folderInd)
myFileNamesLbls = pd.concat(processed_list,axis=0)
myFileNamesLbls.to_csv('/mnt/d/Data/Leukemia/Cleaned/fold_lbls.csv',index=False)
pd.read_csv('/mnt/d/Data/Leukemia/Cleaned/fold_lbls.csv').iloc[:,[1,3]].rename(
{'0':'file','0.1':'label'},axis=1).to_csv(
    '/mnt/d/Data/Leukemia/sample_these_lbls.csv',index=False)
```

Appendix D: Classifier Code

Required Packages

```
import matplotlib
import matplotlib.pyplot as plt
from skimage.viewer import ImageViewer
import psutil
import re
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay,
from sklearn.metrics import roc_auc_score, roc_curve, auc
import cupy
from skimage import color
from skimage import io
from skimage.transform import resize
import dask
import dask.array as da
from dask.utils import parse_bytes
psutil.cpu_count(logical = True)
os.getcwd()
tf.config.list_physical_devices('GPU')
```

Data Loading

```
class cell_class:
    def __init__(self,path_or_name = None):
        if path_or_name is not None: # load model if already exists
            self.model = keras.models.load_model(path_or_name)
            self.model_path = path_or_name
            self.model_name = re.sub('^.*/(?!.+/)','', path_or_name)
        pass
    def start(self, img_shape = (256,256), grayscale=False, per_class=1000,
              tr_val_split=(7,2,1), seed=1000): # initialize data
        #cluster = LocalCUDACluster()#rmm_managed_memory=True)
        #client = Client(cluster)
        lbsmp = pd.read_csv('/mnt/d/Data/Leukemia/sample_these_lbls.csv')
        self.img_shape = img_shape
        mnx = np.min(
        [len(lbsmp[lbsmp['label']==1].values), len(lbsmp[lbsmp['label']==0].values)]
        )
        if per_class > mnx:
            n = mnx
        else:
            n = int(per_class)
        self.seed = seed
        np.random.seed(self.seed)
        ox0 = np.random.choice(
        np.arange(len(lbsmp[lbsmp['label']==0].values)),size=n,replace=False)
        ox1 = np.random.choice(
        np.arange(len(lbsmp[lbsmp['label']==1].values)),size=n,replace=False)
        indices = np.arange(n)
        np.random.shuffle(indices)
        lbsmp = pd.concat(
            [lbsmp[lbsmp['label']==0].iloc[ox0,:],
                   lbsmp[lbsmp['label']==1].iloc[ox1,:]],
                   axis=0,ignore_index=True)
        self.tvt = tr_val_split
        ssm = sum(self.tvt)
        ini_tr = indices[:(n*sum(self.tvt[:1])//ssm)]
        ini_vl = indices[(n*sum(self.tvt[:1])//ssm):(n*sum(self.tvt[:2])//ssm)]
        ini_ts = indices[(n*sum(self.tvt[:2])//ssm):]
        ind_tr = np.concatenate([ini_tr,n+ini_tr])
        ind_vl = np.concatenate([ini_vl,n+ini_vl])
        ind_ts = np.concatenate([ini_ts,n+ini_ts])
        np.random.seed(self.seed)
        np.random.shuffle(ind_tr)
        np.random.seed(self.seed)
        np.random.shuffle(ind_vl)
        np.random.seed(self.seed)
        np.random.shuffle(ind_ts)
        def myimread(img,shape=(256,256)):
            pht = np.array(resize(io.imread(img), shape,
                       anti_aliasing=True)).astype('float32')
            return pht
        #self.batch_size = batch_size
        self.input_shape = list(self.img_shape)+[3]
        self.output_shape = 1
        if __name__ == '__main__':
                print('loading training data...')
                train_list = Parallel(n_jobs=-1, prefer='threads')(
                delayed(myimread)(
                    fn,self.img_shape) for fn in lbsmp.iloc[ind_tr,0])
                print('Done, loading validation data...')
                valid_list = Parallel(n_jobs=-1, prefer='threads')(
                delayed(myimread)(
                    fn,self.img_shape) for fn in lbsmp.iloc[ind_vl,0])
                print('Done, loading testing data...')
                test_list = Parallel(n_jobs=-1, prefer='threads')(
                delayed(myimread)(
                    fn,self.img_shape) for fn in lbsmp.iloc[ind_ts,0])
                print('Done!')
        self.train = (np.array(train_list), lbsmp.iloc[ind_tr,1].values)
        self.valid = (np.array(valid_list), lbsmp.iloc[ind_vl,1].values)
        self.test = (np.array(test_list), lbsmp.iloc[ind_ts,1].values)
        return (self.train,self.valid,self.test)
```

Model Maker

```
class model_maker:
    def __init__(self, data, path_or_name=None):
        self.train, self.valid, self.test = data[0], data[1], data[2]
        if path_or_name is not None: # load model if already exists
            self.model = keras.models.load_model(path_or_name)
            self.model_path = path_or_name
            self.model_name = re.sub('^.*/(?!.+/)','', path_or_name)
        pass
    def compile_model(self,model,path_or_name,
            optimizer='adam', loss='binary_crossentropy',
            metrics=['accuracy','AUC'], summary=False, **kwargs):
            self.model_path = path_or_name
            self.model_name = re.sub('^.*/(?!.+/)','', path_or_name)
            self.optimizer = optimizer
            self.loss = loss

            if str(type(model)) == "<class 'list'>":
            #if you want to specify a model as a list (Sequential only)
                self.model = Sequential(model)
            else:
                self.model = model
            self.metrics = metrics
            self.model.compile(optimizer=self.optimizer,
            loss=self.loss, metrics = self.metrics, **kwargs)
            self.model._name = self.model_name
            ### whether or not we want to display the summary
            if summary == True:
                self.model.summary()
    def get_fit(self, **kwargs):
        plt.clf()
        self.history = self.model.fit(self.train[0], self.train[1],
        validation_data=(self.valid[0],self.valid[1]), **kwargs)
        f, (ax1, ax2) = plt.subplots(2, 1, sharex=True,figsize=(15,15))
        self.model.save(self.model_path)
        ax1.set_title('log loss and accuracy charts')
        lshp = len(self.history.history['loss'])
        self.epochs_tried = lshp
        self.min_val_loss = np.min(self.history.history['val_loss'])
        self.epoch_used = 1+np.argmin(self.history.history['val_loss'])
        ax1.plot(np.arange(1,lshp+1),
        np.log10(self.history.history['loss']),label='train')
        ax1.plot(np.arange(1,lshp+1),
        np.log10(self.history.history['val_loss']),label='val')
        ql = np.ravel(
                    np.concatenate([np.log10(self.history.history['val_loss']),
                                   np.log10(self.history.history['loss'])]
                             )
                )
        ax1.plot([self.epoch_used,self.epoch_used],
                [np.nanmin(ql),np.nanmax(ql)], color="black",
            lw=2, linestyle='--',
            label="Epoch Used: "+str(
            self.epoch_used)+", Loss of "+"{:.5e}".format(self.min_val_loss)
                )
        ax1.legend(loc='lower left')
        ax1.set_ylabel('log10 loss')
        ax2.plot(np.arange(1,lshp+1), self.history.history['accuracy'])
        ax2.plot(np.arange(1,lshp+1), self.history.history['val_accuracy'])
        ax2.set_ylabel('acc')
        ax2.set_xlabel('epoch')
        #ax2.legend(['train', 'val'], loc='lower right')
        plt.savefig(self.model_path+'/loss_acc.png')
        plt.show()
    def get_test(self, data=None, name=None):
        plt.clf()
        if data is None:
            self.data = self.test
            self.name = 'test'
        else:
            self.data = data
            self.name = name
        if self.name == 'valid':
            namex = 'validation'
        else:
            namex = self.name
        self.predictions = self.model.predict(self.data[0])
        myCMAT = confusion_matrix(self.data[1], np.round(self.predictions,0))
        accx = (sum([myCMAT[x,x] for x in np.arange(myCMAT.shape[1])]))/self.data[1].shape[0]
        self.test_acc = accx
        self.test_auc = sklearn.metrics.roc_auc_score(self.data[1], self.predictions)
        ConfusionMatrixDisplay(myCMAT).plot()
        if self.name == 'valid':
            namex = 'validation'
        else:
            namex = self.name
        plt.title(namex.capitalize() + ' Confusion Matrix (acc = ' + str(
        np.round(100*accx,2)) + '%)')
        plt.savefig(self.model_path + '/' + self.name + '_cmat.png')
        plt.show()
        outs = pd.DataFrame({
            self.name+'_acc': self.test_acc,
            self.name+'_auc': self.test_auc
        },index=np.array([0]))
        self.error_metrics = outs
        plt.clf()
        # Compute micro-average ROC curve and ROC area
        fpr, tpr, _ = roc_curve(self.data[1], self.predictions)
        roc_auc = auc(fpr, tpr)
        plt.plot(
            fpr,
            tpr,
            color="darkorange",
            lw=2,
            label="ROC curve (area = %0.5f)" % roc_auc,
        )
        plt.plot([0, 1], [0, 1], color="navy", lw=2, linestyle="--")
        plt.xlim([0.0, 1.0])
        plt.ylim([0.0, 1.05])
        plt.xlabel("False Positive Rate")
        plt.ylabel("True Positive Rate")
        plt.title(namex.capitalize() + " ROC Curve")
        plt.legend(loc="lower right")
        plt.savefig(self.model_path + '/' + self.name + '_auc.png')
        plt.show()
```

Grid Searcher

```
class paramgs:
    def __init__(self,img_shape = (256,256), grayscale=False, per_class=1000,
             tr_val_split=(7,2,1), seed=1000):
        t0 = time.time()
        mdt = cell_class()
        cxx = mdt.start(img_shape = img_shape, grayscale=grayscale, per_class=per_class,
              tr_val_split=tr_val_split, seed=seed,
                       )
        self.attrs = model_maker(cxx)
        t1 = time.time()
        print(str(np.round(t1-t0,2))+ ' sec to initialize data')
    def all_names(self,models,optimizers,lrs,batch_sizes,prepend=''):
        t0 = time.time()
        self.models = models # list of models, Keras models
        self.optimizers = optimizers # list of optimizers, str
        self.lrs = lrs # learning rates
        self.batch_sizes = batch_sizes # list of batch sizes
        names = []
        modsx = []
        for nms in models:
            v0 = [name for name in globals() if globals()[name] is nms][0] # extract model names
            v0 =  prepend + v0
            modsx.append(v0)
            for v1 in optimizers:
                R = 0
                for v2 in lrs:
                    #exec("x.append(%s(%f))" % (v1,v2))
                    B = 0

                    for v3 in batch_sizes:
                        exec("names.append('%s_'+'%s'+'_r10n'+'%d'+'_b2p'+'%d')" % (v0,v1[0],R,B))
                        B += 1
                    R += 1
        self.names = names
        self.modsx = modsx
        t1 = time.time()
        print(str(np.round(t1-t0,2))+ ' sec to initialize model names')
        return self.names
    def ready_all(self,loss='binary_crossentropy', **kwargs):
        t0 = time.time()
        k = 0
        a, b, c, d = [], [], [], []
        m = []
        for v0 in self.models:
            for v1 in self.optimizers:
                for v2 in self.lrs:
                    def foo():
                        recipe = "%s(%f)" % (v1,v2)
                        ldict = {'recipe':recipe}
                        exec("qx = %s" % (recipe,),globals(),ldict)
                        qv = ldict['qx']
                        return qv
                    #print(foo())
                    for v3 in self.batch_sizes:
                        self.attrs.compile_model(v0,self.names[k],
                        optimizer = foo(), loss=loss,
                        summary=False, **kwargs)
                        self.attrs.model.save(self.names[k])
                        k += 1
                        a.append(v0)
                        b.append(v1)
                        c.append(v2)
                        d.append(v3)
        self.lmd = a # models
        self.lmo = b # optimizers
        self.lml = c # learning rates
        self.lmb = d # batch sizes
        t1 = time.time()
        print(str(np.round(t1-t0,2))+ ' sec to compile models')
    def fire_away(self,my_model, filename='results', debug=False, **kwargs):
        try:
            del self.attrs.model
            del self.attrs.model_path
        except AttributeError:
            pass
        t = []
        v = []
        s = []
        e = []
        m = []
        u = []
        h = []
        for j in np.arange(len(my_model)):
            t0 = time.time()
            if debug == False:
                try:
                    self.attrs.model_path = my_model[j]
                    self.attrs.model = keras.models.load_model(my_model[j]+'/')
                    self.attrs.get_fit(batch_size = self.lmb[j], **kwargs)
                    if type(t) == list:
                        self.attrs.get_test(data=self.attrs.train, name = 'train')
                        t = self.attrs.error_metrics
                        self.attrs.get_test(data=self.attrs.valid, name = 'valid')
                        v = self.attrs.error_metrics
                        self.attrs.get_test(data=self.attrs.test, name = 'test')
                        s = self.attrs.error_metrics
                    else:
                        self.attrs.get_test(data=self.attrs.train, name = 'train')
                        t = pd.concat([t,self.attrs.error_metrics],axis=0,ignore_index=True)
                        self.attrs.get_test(data=self.attrs.valid, name = 'valid')
                        v = pd.concat([v,self.attrs.error_metrics],axis=0,ignore_index=True)
                        self.attrs.get_test(data=self.attrs.test, name = 'test')
                        s = pd.concat([s,self.attrs.error_metrics],axis=0,ignore_index=True)
                    e.append(self.attrs.epochs_tried)
                    m.append(self.attrs.min_val_loss)
                    u.append(self.attrs.epoch_used)
                except:
                    t.append(np.nan)
                    v.append(np.nan)
                    s.append(np.nan)
                    e.append(np.nan)
                    m.append(np.nan)
                    u.append(np.nan)
            else:
                self.attrs.model_path = my_model[j]
                self.attrs.model = keras.models.load_model(my_model[j]+'/')
                self.attrs.get_fit(batch_size = self.lmb[j], **kwargs)
                if type(t) == list:
                    self.attrs.get_test(data=self.attrs.train, name = 'train')
                    t = self.attrs.error_metrics
                    self.attrs.get_test(data=self.attrs.valid, name = 'valid')
                    v = self.attrs.error_metrics
                    self.attrs.get_test(data=self.attrs.test, name = 'test')
                    s = self.attrs.error_metrics
                else:
                    self.attrs.get_test(data=self.attrs.train, name = 'train')
                    t = pd.concat([t,self.attrs.error_metrics],axis=0,ignore_index=True)
                    self.attrs.get_test(data=self.attrs.valid, name = 'valid')
                    v = pd.concat([v,self.attrs.error_metrics],axis=0,ignore_index=True)
                    self.attrs.get_test(data=self.attrs.test, name = 'test')
                    s = pd.concat([s,self.attrs.error_metrics],axis=0,ignore_index=True)
                e.append(self.attrs.epochs_tried)
                u.append(self.attrs.epoch_used)
                m.append(self.attrs.min_val_loss)
            t1 = time.time()
            h.append(t1-t0)
            print(str(np.round(t1-t0,2))+ ' sec to run model')
            self.epochs_tried = e
            self.epoch_used = u
            self.min_val_loss = m
            self.times = h
            tbx = pd.DataFrame(
                {
                    'model': self.names[:(j+1)],
                    'optimizer': self.lmo[:(j+1)],
                    'learn_rate': self.lml[:(j+1)],
                    'batch_size': self.lmb[:(j+1)],
                    'epochs_tried' : self.epochs_tried,
                    'epoch_used' : self.epoch_used,
                    'min_val_loss': self.min_val_loss,
                    'time': self.times
                }
                               )
            tbx = pd.concat([tbx,t,v,s],axis=1)
            self.results = tbx
            tbx.to_csv(filename+'.csv',index=False)
        return tbx
    def snooper(self,pattern,filename='results', debug=False, **kwargs):
        # Debug tool for the model
        ar = np.array([name for name in os.listdir(".") if os.path.isdir(name)])
```

```
sn = [re.match(pattern,name)!=None for name in os.listdir(".") if os.path.isdir(name)]
my_model = ar[sn]
t = []
v = []
s = []
e = []
m = []
u = []
h = []
for j in np.arange(len(my_model)):
    t0 = time.time()
    if debug == False:
        try:
            self.attrs.model_path = my_model[j]
            self.attrs.model = keras.models.load_model(my_model[j]+'/')
            if type(t) == list:
                self.attrs.get_test(data=self.attrs.train, name = 'train')
                t = self.attrs.error_metrics
                self.attrs.get_test(data=self.attrs.valid, name = 'valid')
                v = self.attrs.error_metrics
                self.attrs.get_test(data=self.attrs.test, name = 'test')
                s = self.attrs.error_metrics
            else:
                self.attrs.get_test(data=self.attrs.train, name = 'train')
                t = pd.concat([t,self.attrs.error_metrics],axis=0,ignore_index=True)
                self.attrs.get_test(data=self.attrs.valid, name = 'valid')
                v = pd.concat([v,self.attrs.error_metrics],axis=0,ignore_index=True)
                self.attrs.get_test(data=self.attrs.test, name = 'test')
                s = pd.concat([s,self.attrs.error_metrics],axis=0,ignore_index=True)
        except:
            t.append(np.nan)
            v.append(np.nan)
            s.append(np.nan)
            e.append(np.nan)
            m.append(np.nan)
            u.append(np.nan)
    else:
        self.attrs.model_path = my_model[j]
        self.attrs.model = keras.models.load_model(my_model[j]+'/')
        if type(t) == list:
            self.attrs.get_test(data=self.attrs.train, name = 'train')
            t = self.attrs.error_metrics
            self.attrs.get_test(data=self.attrs.valid, name = 'valid')
            v = self.attrs.error_metrics
            self.attrs.get_test(data=self.attrs.test, name = 'test')
            s = self.attrs.error_metrics
        else:
            self.attrs.get_test(data=self.attrs.train, name = 'train')
            t = pd.concat([t,self.attrs.error_metrics],axis=0,ignore_index=True)
            self.attrs.get_test(data=self.attrs.valid, name = 'valid')
            v = pd.concat([v,self.attrs.error_metrics],axis=0,ignore_index=True)
            self.attrs.get_test(data=self.attrs.test, name = 'test')
            s = pd.concat([s,self.attrs.error_metrics],axis=0,ignore_index=True)
    t1 = time.time()
```

```
            h.append(t1-t0)
            print(str(np.round(t1-t0,2))+ ' sec to load and test model')
            print(m)
            tbx = pd.DataFrame(
                {
                    'model': self.names,
                    #'optimizer': self.optimizers,
                    #'learn_rate': self.lrs,
                    #'batch_size': self.batch_sizes,
                }
                               )
            tbx = pd.concat([tbx,t,v,s],axis=1)
            self.results = tbx
            tbx.to_csv(filename+'.csv',index=False)
        return tbx
```

Clasification

```
mods_srt = [model_1,model_2,model_4,model_5]
mods_tri = [model_1_tri,model_2_tri,model_4_tri,model_5_tri]

ppg = paramgs(img_shape = (256,256), grayscale=False, per_class=3200,
              tr_val_split=(7,2,1), seed=1000)
ppg.all_names([mods_srt[x] for x in range(len(mods_srt))],
              ['Adam'],
              [10.**-4,2.5*(10.**-4)],
              [64,128])
ppg.ready_all(loss='binary_crossentropy')
ppg.fire_away(ppg.names,filename='results_srt_cnn_2',epochs=4000,
debug=True, verbose=1,
              callbacks = [
              EarlyStopping(
              monitor = 'val_loss', mode='min',patience = 50,
              restore_best_weights=True)
              ]
              )
del ppg
ppg = paramgs(img_shape = (256,256), grayscale=False, per_class=3200,
              tr_val_split=(7,2,1), seed=1000)
ppg.all_names([mods_tri[x] for x in range(len(mods_tri))],
              ['Adam'],
              [0.4*(10.**-4),10.**-4],
              [64,128])
ppg.ready_all(loss='binary_crossentropy')
ppg.fire_away(ppg.names,filename='results_tri_cnn_2',epochs=4000,
debug=True, verbose=1,
              callbacks = [LearningRateScheduler(
              triangleScheduler(stepsize=4,scale=4,start_at='max')
              ),
              EarlyStopping(
              monitor = 'val_loss', mode='min',patience = 50,
              restore_best_weights=True)])
```

Appendix E: Patient-level Diagnostic Code

```
class validationFit:
  def __init__(self, generate_data=False,img_size=(256,256)):
      if generate_data == True: #if true, this applies the preprocessing technique to the patient set.
          self.img_shape = img_size
          tph = '/mnt/d/Data/Leukemia/C-NMC_Leukemia/validation_data/C-NMC_test_prelim_phase_data/'
          testpath = tph
          pfx = pd.read_csv(
          '/mnt/d/Data/Leukemia/C-NMC_Leukemia/validation_data/C-NMC_test_prelim_phase_data_labels.csv'
          )
          pdx = pfx.copy()
          # Clean patient ID names so that they only include patient ID
          pdx['new_names'] = testpath + pdx['new_names'].astype(str)
          pdx['Patient_ID'] = pd.Series(testpath + pdx['Patient_ID'].astype(str)).str.replace(
              'C\-NMC_Leukemia','Cleaned', regex = True
          ).str.replace(
              '\.bmp$','.png',regex = True)
          testfiles = pdx['new_names']
          outfiles = pdx['Patient_ID']
          pfx['new_names'] = outfiles
          pfx['Patient_ID'] = pfx['Patient_ID'].str.replace(
          'UID_','',regex=True).str.replace('_.*','',regex=True)
          pfx.to_csv(
          '/mnt/d/Data/Leukemia/C-NMC_Leukemia/validation_data/patient_lbls.csv',index=False
          )
          t0 = time.time()
          print('Preprocessing...')
          if __name__ == "__main__":
              processed_list = Parallel(n_jobs=-1, prefer='threads')(delayed(masterRecrop)(
                  img_path = testfiles[k],
                  out_path = outfiles[k],
                  deg = 0,size = self.img_shape[0],flip = False) for k in range(len(testfiles)))
          t1 = time.time()
          str(np.round(t1-t0,2))+ ' sec to preprocess images'
          self.patients = pfx
      else:
          self.patients = pd.read_csv(
          '/mnt/d/Data/Leukemia/C-NMC_Leukemia/validation_data/patient_lbls.csv'
          )
          self.img_shape = img_size
      def myimread(img,shape=(256,256)):
          pht = np.array(resize(io.imread(img), shape, anti_aliasing = True)).astype('float32')
          return pht
      t0 = time.time()
      if __name__ == '__main__':
              print('loading training data...')
              train_list = Parallel(n_jobs=-1, prefer='threads')(delayed(myimread)(
                  fn,self.img_shape) for fn in self.patients['new_names'].values)
      t1 = time.time()
      str(np.round(t1-t0,2))+ ' sec to load data array'
      self.data = (
      np.array(train_list),
      self.patients['labels'].values, self.patients['Patient_ID'].values
```

```
    )
def load_fitted_model(self,path_or_name):
    if path_or_name is not None: # load model if already exists
        self.model = keras.models.load_model(path_or_name)
        self.model_path = path_or_name
        self.model_name = re.sub('^.*/(?!.+/)','', path_or_name)
    pass
def cell_test(self):
    self.predictions = self.model.predict(self.data[0])
    myCMAT = confusion_matrix(self.data[1], np.round(self.predictions,0))
    accx = (sum([myCMAT[x,x] for x in np.arange(myCMAT.shape[1])]))/self.data[1].shape[0]
    self.test_acc = accx
    self.test_auc = sklearn.metrics.roc_auc_score(self.data[1], self.predictions)
    ConfusionMatrixDisplay(myCMAT).plot()
    plt.title('Patient Cell Confusion Matrix (acc = ' + str(np.round(100*accx,2)) + '%)')
    plt.savefig(self.model_path + '/patient_cell_cmat.png')
    plt.show()
    outs = pd.DataFrame({
        'acc': self.test_acc,
        'auc': self.test_auc
    },index=np.array([0]))
    self.error_metrics = outs
    plt.clf()
    # Compute micro-average ROC curve and ROC area
    fpr, tpr, _ = roc_curve(self.data[1], self.predictions)
    roc_auc = auc(fpr, tpr)
    plt.plot(
        fpr,
        tpr,
        color="darkorange",
        lw=2,
        label="ROC curve (area = %0.5f)" % roc_auc,
    )
    plt.plot([0, 1], [0, 1], color="navy", lw=2, linestyle="--")
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel("False Positive Rate")
    plt.ylabel("True Positive Rate")
    plt.title( "Patient Cell ROC Curve")
    plt.legend(loc="lower right")
    plt.savefig(self.model_path + '/patient_cell_auc.png')
    plt.show()
def patient_test(self):
    mydf = pd.DataFrame({
        'id'       : self.data[2],
        'lbl'      : self.data[1],
        'bin_pred' : np.round(self.predictions.ravel(),0),
        'lin_pred' : self.predictions.ravel(),
        'pw2_pred' : self.predictions.ravel()**2,
    })
    pbms = mydf.groupby(['id'])['bin_pred'].mean()
    plms = mydf.groupby(['id'])['lin_pred'].mean()
    psps = (mydf.groupby(['id'])['pw2_pred']).mean()
    plbl = mydf.groupby(['id'])['lbl'].mean()
```

```
#pdfx = pd.concat([pbms,plms,plbl],axis=1)
bbq = np.concatenate([pbms.values,plms.values,psps.values,[0,1]],axis=0)
bbq.sort()
a00 = ([])
for x in np.arange(bbq.shape[0]):
    a00.append([bbq[x],
        np.mean(
            (
                (pbms.values >= bbq[x]) == (plbl.values)
            )
        )

    ])
a00 = np.array(a00)
adx = pd.DataFrame({
    'threshold' : a00[:,0],
    'binary_accuracy' : a00[:,1],
})
pqq = (pbms.values >= adx['threshold'][adx['binary_accuracy'].argmax()])
myCMAT = confusion_matrix(plbl.values, pqq)
accx = (sum([myCMAT[x,x] for x in np.arange(myCMAT.shape[1])]))/plbl.values.shape[0]
self.patient_acc = accx
self.patient_auc = sklearn.metrics.roc_auc_score(plbl.values, pbms.values)
ConfusionMatrixDisplay(myCMAT).plot()
plt.title('Patient Level Confusion Matrix (acc = ' + str(np.round(100*accx,2)) + '%)')
plt.savefig(self.model_path + '/patient_level_cmat.png')
plt.show()
plt.clf()
# Compute micro-average ROC curve and ROC area
fpr, tpr, _ = roc_curve(plbl.values, pbms.values)
roc_auc = auc(fpr, tpr)
plt.plot(
    fpr,
    tpr,
    color="darkorange",
    lw=2,
    label="ROC curve (area = %0.5f)" % roc_auc,
)
plt.plot([0, 1], [0, 1], color="navy", lw=2, linestyle="--")
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title( "Patient Level ROC Curve")
plt.legend(loc="lower right")
plt.savefig(self.model_path + '/patient_level_auc.png')
plt.show()
amx = pd.DataFrame({
    'model'       : self.model_name,
    'cell_acc'    : self.test_acc,
    'cell_auc'    : self.test_auc,
    'threshold'   : adx['threshold'][adx['binary_accuracy'].argmax()],
    'patient_acc' : self.patient_acc,
    'patient_auc' : self.patient_auc
},index=[0])
amx.to_csv(self.model_path + '/patient_level_results.csv',index=False)
return amx
```

Appendix F: Full Results

Table 5: All models' hyperparameters.

id	model	optimizer	min_lr	lr_policy	batch_size
model_1_A_r10n0_b2p0	model_1	Adam	0.00010	constant	64
model_1_A_r10n0_b2p1	model_1	Adam	0.00010	constant	128
model_1_A_r10n1_b2p0	model_1	Adam	0.00025	constant	64
model_1_A_r10n1_b2p1	model_1	Adam	0.00025	constant	128
model_2_A_r10n0_b2p0	model_2	Adam	0.00010	constant	64
model_2_A_r10n0_b2p1	model_2	Adam	0.00010	constant	128
model_2_A_r10n1_b2p0	model_2	Adam	0.00025	constant	64
model_2_A_r10n1_b2p1	model_2	Adam	0.00025	constant	128
model_4_A_r10n0_b2p0	model_4	Adam	0.00010	constant	64
model_4_A_r10n0_b2p1	model_4	Adam	0.00010	constant	128
model_4_A_r10n1_b2p0	model_4	Adam	0.00025	constant	64
model_4_A_r10n1_b2p1	model_4	Adam	0.00025	constant	128
model_5_A_r10n0_b2p0	model_5	Adam	0.00010	constant	64
model_5_A_r10n0_b2p1	model_5	Adam	0.00010	constant	128
model_5_A_r10n1_b2p0	model_5	Adam	0.00025	constant	64
model_5_A_r10n1_b2p1	model_5	Adam	0.00025	constant	128
model_1_tri_A_r10n0_b2p0	model_1_tri	Adam	0.00004	triangleScheduler(4,4,"max")	64
model_1_tri_A_r10n0_b2p1	model_1_tri	Adam	0.00004	triangleScheduler(4,4,"max")	128
model_1_tri_A_r10n1_b2p0	model_1_tri	Adam	0.00010	triangleScheduler(4,4,"max")	64
model_1_tri_A_r10n1_b2p1	model_1_tri	Adam	0.00010	triangleScheduler(4,4,"max")	128
model_2_tri_A_r10n0_b2p0	model_2_tri	Adam	0.00004	triangleScheduler(4,4,"max")	64
model_2_tri_A_r10n0_b2p1	model_2_tri	Adam	0.00004	triangleScheduler(4,4,"max")	128
model_2_tri_A_r10n1_b2p0	model_2_tri	Adam	0.00010	triangleScheduler(4,4,"max")	64
model_2_tri_A_r10n1_b2p1	model_2_tri	Adam	0.00010	triangleScheduler(4,4,"max")	128
model_4_tri_A_r10n0_b2p0	model_4_tri	Adam	0.00004	triangleScheduler(4,4,"max")	64
model_4_tri_A_r10n0_b2p1	model_4_tri	Adam	0.00004	triangleScheduler(4,4,"max")	128
model_4_tri_A_r10n1_b2p0	model_4_tri	Adam	0.00010	triangleScheduler(4,4,"max")	64
model_4_tri_A_r10n1_b2p1	model_4_tri	Adam	0.00010	triangleScheduler(4,4,"max")	128
model_5_tri_A_r10n0_b2p0	model_5_tri	Adam	0.00004	triangleScheduler(4,4,"max")	64
model_5_tri_A_r10n0_b2p1	model_5_tri	Adam	0.00004	triangleScheduler(4,4,"max")	128
model_5_tri_A_r10n1_b2p0	model_5_tri	Adam	0.00010	triangleScheduler(4,4,"max")	64
model_5_tri_A_r10n1_b2p1	model_5_tri	Adam	0.00010	triangleScheduler(4,4,"max")	128

Table 6: All models' results.

model	epochs	val_loss	time	tr_acc	tr_auc	vl_acc	vl_auc	ts_acc	ts_auc
model_1	85	0.3222	1347.0	0.8873	0.9515	0.8602	0.9324	0.8422	0.9219
model_1	131	0.3146	1778.0	0.9031	0.9610	0.8648	0.9387	0.8531	0.9275
model_1	113	0.4196	1650.0	0.8328	0.9118	0.8062	0.8856	0.8016	0.8840
model_1	62	0.4131	1064.0	0.8567	0.9422	0.8156	0.8938	0.7953	0.8899
model_2	100	0.2903	1059.0	0.9250	0.9770	0.8758	0.9485	0.8797	0.9468
model_2	150	0.2899	1286.0	0.9250	0.9726	0.8812	0.9492	0.8672	0.9340
model_2	46	0.3308	680.1	0.9328	0.9796	0.8641	0.9352	0.8656	0.9291
model_2	57	0.3243	725.3	0.8828	0.9485	0.8625	0.9311	0.8453	0.9192
model_4	94	0.2994	1889.0	0.9330	0.9778	0.8719	0.9456	0.8828	0.9398
model_4	109	0.2904	2029.0	0.9145	0.9710	0.8836	0.9469	0.8609	0.9348
model_4	53	0.3685	1341.0	0.8962	0.9581	0.8375	0.9205	0.8062	0.8913
model_4	32	0.3755	1044.0	0.8732	0.9327	0.8312	0.9152	0.8297	0.9047
model_5	84	0.2905	1975.0	0.9337	0.9782	0.8836	0.9471	0.8719	0.9367
model_5	130	0.2913	2531.0	0.9339	0.9831	0.8875	0.9486	0.8719	0.9386
model_5	52	0.3233	1463.0	0.9391	0.9831	0.8648	0.9333	0.8312	0.9057
model_5	56	0.3164	1490.0	0.8978	0.9567	0.8633	0.9340	0.8438	0.9133
model_1_tri	95	0.3084	1453.0	0.9011	0.9600	0.8672	0.9393	0.8609	0.9336
model_1_tri	175	0.2965	2111.0	0.9214	0.9705	0.8766	0.9435	0.8625	0.9408
model_1_tri	62	0.3450	1083.0	0.8750	0.9453	0.8508	0.9264	0.8250	0.9055
model_1_tri	113	0.4367	1522.0	0.8491	0.9252	0.8008	0.8794	0.7625	0.8475
model_2_tri	127	0.2841	1172.0	0.9703	0.9940	0.8945	0.9564	0.8500	0.9376
model_2_tri	155	0.2809	1273.0	0.9208	0.9761	0.8789	0.9547	0.8625	0.9404
model_2_tri	51	0.3408	677.6	0.9308	0.9763	0.8672	0.9349	0.8438	0.9169
model_2_tri	62	0.3012	690.6	0.9092	0.9719	0.8734	0.9446	0.8547	0.9300
model_4_tri	77	0.3327	1680.0	0.9045	0.9595	0.8602	0.9340	0.8453	0.9120
model_4_tri	159	0.3118	2624.0	0.9225	0.9763	0.8656	0.9446	0.8469	0.9254
model_4_tri	64	0.3186	1515.0	0.9051	0.9616	0.8609	0.9361	0.8562	0.9252
model_4_tri	68	0.3298	1511.0	0.8924	0.9553	0.8625	0.9319	0.8453	0.9122
model_5_tri	77	0.2974	1921.0	0.9087	0.9649	0.8805	0.9448	0.8531	0.9272
model_5_tri	124	0.3076	2578.0	0.9020	0.9612	0.8734	0.9407	0.8594	0.9249
model_5_tri	42	0.3466	1411.0	0.9107	0.9646	0.8508	0.9232	0.8625	0.9250
model_5_tri	54	0.3238	1527.0	0.9056	0.9594	0.8648	0.9365	0.8422	0.9169