SMU Data Science Review

Volume 3 | Number 2 Article 4

2020

Acute Lymphoblastic Leukemia Detection Using Depthwise Separable Convolutional Neural Networks

Laurence P. Clinton Jr.
Southern Methodist University, lpclinton@smu.edu

Karen M. Somes
Southern Methodist University, ksomes@smu.edu

Yongjun Chu
Southern Methodist University, ychu@smu.edu

Faizan Javed
Southern Methodist University, fjaved@smu.edu

Acute Lymphoblastic Leukemia Detection Using Depthwise Separable Convolutional Neural Networks

Yongjun Chu[1], Laurence Clinton[1], Karen Somes[1], and Faizan Javed[1]

Master of Science in Data Science, Southern Methodist University, Dallas TX 75275
ychu,lpclinton,ksomes,fjaved@smu.edu

Abstract. In this paper, we present a neural network with depthwise separable convolutions (Xception) for the identification of leukemic B-lymphoblast cells, commonly known as Acute Lymphocytic Leukemia (ALL). Earliest possible detection of these cancerous cells is required to minimize the physical toll on the patient and the treatment challenges presented by the disease. Through a transfer learning approach, we tested various convolutional neural network algorithms on our augmented microscopic blood smear image dataset to assess the best performing architecture for classifying leukemic cells, resulting in the Xception architecture. We obtained 99% and 91% accuracy on the training and testing sets, respectively. Furthermore, we achieved a recall rate as high as 98% showing good discrimination power against false negatives.

Keywords: Acute Lymphocytic Leukemia · ALL · Artificial Neural Networks · Convolutional Neural Networks.

1 Introduction

Leukemia is a common cancer worldwide with more than 250,000-300,000 new cases each year. In 2019, it was expected that 61,780 people were expected to be diagnosed with leukemia in the US alone. There are an estimated 399,967 people currently living with or in remission from leukemia in the US [2]. Leukemia is a cancer of blood or bone marrow where blood cells are produced and is characterized by the proliferation of abnormal white blood cells (WBCs) in the bone marrow, resulting in an increase of immature WBCs in the bone marrow. One of the most significant symptoms of leukemia is the presence of an excess number of blast cells in peripheral blood. Therefore, blood smears are routinely examined under a microscope for proper identification and classification of blast cells by hematologists[9]. Leukemia can be pathologically classified into two categories on a broader sense: (1) Acute leukemia (progresses quickly); and (2) chronic leukemia (progresses slowly). In this paper, we study the presence of Acute Lymphoblastic Leukemia (ALL) only.

Although ALL is not as commonly occurring as other types of cancers, the death rate for ALL is quite high. The American Cancer Society's estimates for ALL in the United States for 2020 (including both children and adults) are about 6,150 new cases (3,470 in males and 2,680 in females) and about 1,520 deaths (860 in males and 660 in females). The risk for developing ALL is highest in children younger than 5 years of age. The risk then declines slowly until the mid-20s, and begins to rise again slowly after age 50. Overall, about 4 of every 10 cases of ALL are in adults[4].

Early diagnosis of the ALL is essential. Due to its rapid spread into the bloodstream and other vital organs, ALL is fatal if left untreated. The current most important diagnostic methodology for initial ALL screening is microscopic examination of a blood smear. However, the task of identifying immature leukemic blasts from normal cells under the microscope is challenging because morphologically the images of the two cells appear similar. Thus, manual examination of the slides is often accompanied by inconsistent and subjective reports. In addition, diagnostic confusion may occur due to the appearance of similar signs by other disorders. Thus, we need a cost-effective and robust automated image processing system for ALL screening which can greatly help pathologists to have more accurate diagnosis by improving the clarity of features in images.

In this study, we attempt to develop new approaches for accurate and precise ALL diagnosis from microscopic blood images. We have obtained a large image data set from Cancer Imaging Archive, which were collected from 118 individuals (ALL and healthy) with a total of 12,528 images. We will also apply different image augmentation techniques to further increase the number of samples in our study in order to alleviate the outfitting effect which is often associated with deep learning. Additionally, we will compare our approach to reported ML algorithms and evaluate their performance side by side.

A common model for image classification is a Support Vector Machine (SVM). SVM requires feature extractions from images to feed as input variables into the model, where these features summarize characteristics of the image. For example, this model includes color features, geometric features, texture features, and statistical features, which are high-level mathematical summaries of the pixel locations/channels. However, the performance of the model depends highly on the feature extraction and selection techniques to calculate the decision boundary between classes.

A Convolutional Neural Network (CNN) model uses convolutions of the image as "features" instead of summary features like SVM, as input. It contains multiple convolution and max pooling layers, as well as several dense artificial neural network layers, to learn patterns and representations characteristic to the classes. Over the last 7-8 years, neural network models from the early days, like AlexNet, to the latest ones, such as NASNet, have dominated the classification space over traditional methods, such as SVM, as techniques around deep

learning have matured and more researchers are finding success[11]. Due to this, we expect our CNN model to outperform SVM for our single-cell classification objective.

Herein, we first performed classification using SVM on our data set, following the steps outlined in research by Patel et al. for feature extraction. We also carried out an AlexNet based CNN classification. AlexNet has been used by others on ALL classification and higher than 90 percent accuracy was reported [17]. To our disappointment, however, the classification accuracy for both SVM and AlexNet was sub par, just over 67 percent in accuracy, dramatically lower than what has been reported so far. Further optimizations on parameters did not improve the accuracy. This demonstrates that our single-cell image data is highly challenging for classification.

We then turned our attention to latest deep CNN models and applied a transfer learning approach in classification. In this approach, parameters of a deep CNN model are directly imported, only replacing the top neural network (NN) layer with another one that fit our data structure, a binary classification problem. We further fine tuned the imported parameters to achieve higher accuracy. We explored three of some of the latest models, InceptionV3, Xception and NASNet. All three models performed significantly better than AlexNet and SVM, with each one achieving an accuracy of over 90 percent within just 10 epochs. The great increase in accuracy suggests that finding and tuning a data-fit neural network architecture is the key to achieving a premier classification accuracy on problems as challenging as this.

2 Related Works

Previous research supports that a machine learning (ML) algorithm will help to identify the blood cells with ALL from healthy cells. Several ML algorithms have been developed in the past to classify and recognize leukemia disease from microscopic images. For instance, Paswan et al. used support vector machine (SVM) and k-nearest neighbor (k-NN) to classify AML leukemia subtype, obtaining an accuracy of 83 percent. Patel et al. applied SVM for classifying ALL leukemia subtype and achieved 93 percent accuracy[15]. Recently, several research groups have tested the CNN approach (with architectures such as AlexNet) and even achieved over 90 percent accuracy on ALL detection in a few cases[13, 14, 20, 21]. CNN has also been successfully used in other type of cancer detection, including skin, prostate and gastric cancer predicting[12, 18, 22, 24]. Although the prediction accuracy on ALL for some of reported models appears to be impressive, performance metrics highly depend on the nature of the testing and training sets, particularly if the classes are unbalanced. Therefore it is important that we assess these methods on our data.

3 Data

The data set is downloadable from the cancerimagingarchive.net and is available for commercial, scientific and educational purposes and licensed under the Creative Commons Attribution 3.0 Unported License[8].

The data is comprised of images of both normal and acute lymphoblastic leukemia cells and is segmented into training and test data sets. The training data set contains 10,661 cell images, 7,272 cancerous, and 3,389 normal. Training cell images were taken from 73 subjects, 47 of which had cancer, and 26 normal subjects. The test set is comprised of 1,867 total cell images, 1,219 cancerous, and 648 normal. Test cell images were taken from 28 subjects, 13 of which had cancer, and 15 normal subjects.

The data was processed into our CNN model containing convolution layers, pooling layers, Artificial Neural Network hidden layers, and an output layer which classifies the image as a binary output of 1 or 0, cancerous or benign. The model also uses image augmentation which generates more images by rotating them at different angles and zoom levels.

4 Explanation of ANNs and CNNs

Convolutional Neural Networks (CNN) are machine learning algorithms commonly used to classify imagery. CNNs are composed of multiple layers such as the convolution layer, pooling layer, and fully connected Artificial Neural Network (ANN) layer. The architecture of a classic CNN model, AlexNet is presented in Figure 1. The first layer is the convolution layer, where important features in the images are feature mapped. One of the distinct differences between the convolution layer and the fully connected artificial neural network layer is that the ANN learns global patterns, such as patterns of all pixels of the input images. The convolution layer, on the other hand, learns local patterns[7]. A key mechanism in this process is through the use of a 2D sliding window of inputs, such as a 3x3 window grid that slides over the input imagery at a certain stride, mapping distinct features. The images are broken into local patterns that can be discerned, such as edges and textures (see Figure 2). This allows for important features to be mapped as well as dimensionality to be reduced on the input images. Images are transformed or operated on in the convolution layer via tensors such as 3D tensors. Tensors have dimensions of height, width, and depth which represent the value of color channel. For gray scale images the depth value is 1, whereas with RGB images (Red Green Blue) the typical channel depth is three. The red, green, blue channels represent frequency ranges in the electromagnetic spectrum in the form of pixel values.

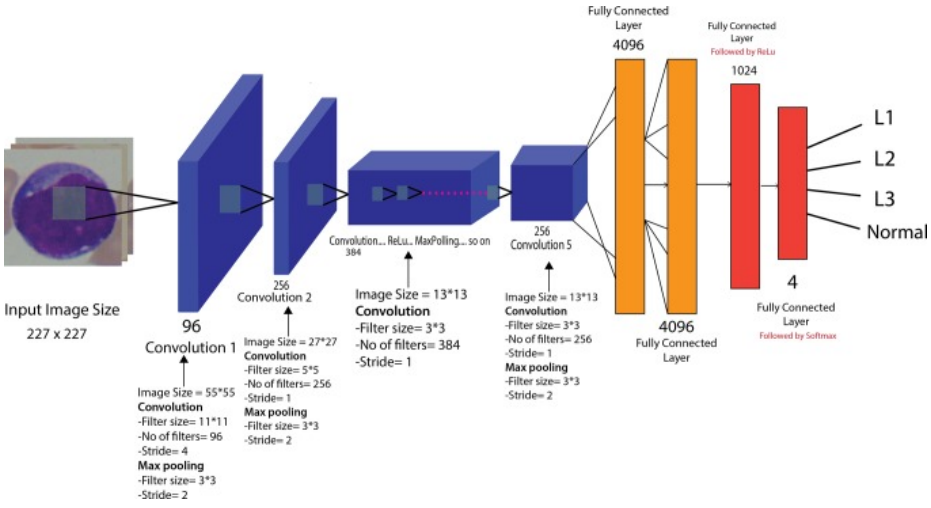

Fig. 1. AlexNet, a Convolutional Neural Network composed of convolution layers,pooling layers, and a fully connected layer[18]

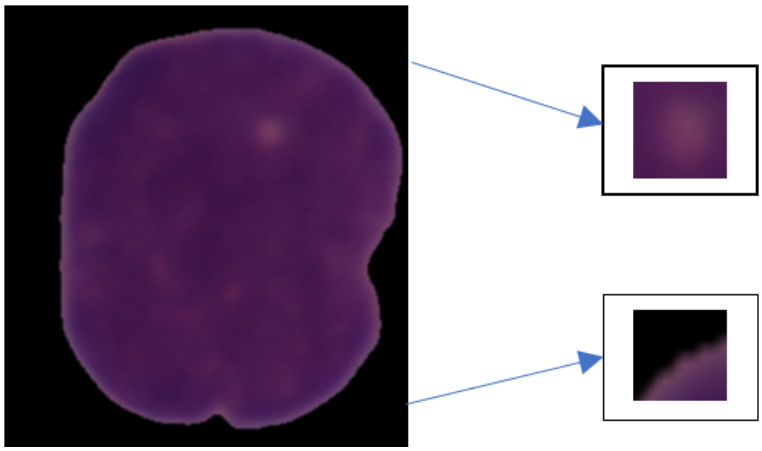

Fig. 2. Convolution layer mapping local patterns

The convolution operation extracts grids of pixels from the input image via a sliding window and applies a learned linear transformation to all extracted grids, resulting in an output feature map[7]. The resulting 3D tensor contains the encoding of feature rich aspects of the input data.

A key advantage of CNNs are that they are considered to be translation invariant, meaning patterns learned from one local area do not have to be relearned on other areas of the image[7]. For example, a pattern learned on the upper right corner of a picture can be recognized in other areas such as another corner. The fully connected layer, on the other hand, would have to relearn the pattern since the locale changed. This advantage allows CNNs to be efficient when processing images, and thus less training is needed to learn features. CNNs can be designed and constructed to follow a certain learning pattern. The first convolution layer learns patterns such as edges. This is followed by a second convolution layer that will focus on larger and more complex patterns made of the features of the first layer. This learning pattern continues with each additional convolution layer, thus allowing CNNs to learn increasingly complex patterns[7].

After the convolution layer maps important features, data is transferred to the next layer in the CNN, the pooling layer. Different types of pooling layers exist such as average pooling and max pooling, however, the end goal is the same, in that the pooling layer reduces image dimensions while preserving important features. In the case of max pooling, the maximum value of each color channel is extracted by using a sliding window of some size, such as a 2x2 grid of pixels, that maps the maximum pixel value found in the sliding window grid[7]. This culminates with a reduced feature map by a factor of two. Average pooling uses the same mechanism of the sliding window to reduce dimensionality, but instead takes the average value of the color channels as opposed to the maximum value.

From the max pooling layer, the data is flattened to a vector for input into the fully connected Artificial Neural Network layer that could potentially be comprised of several thousands of neurons. The goal of the fully connected layer is to classify the input images to binary or multiple classes.

The basic component of the fully connected layer is the perceptron, or the composition of a single neuron within one layer of the ANN (see Figure 3). The perceptron consists of input layers, an activation function, and an output that gets passed to another layer within the ANN or the final output layer. The flattened layer serves as inputs into perceptrons of the fully connected layer within the overall CNN architecture. An arbitrary set of weights is initially assigned to inputs denoting importance. The activation function within each neuron or perceptron transforms the weighted sum of inputs from a node into the activation of the node, serving as the input to other nodes of deeper layers within the ANN. Several types of activation functions exist such as Sigmoid, Tanh, Rectified Linear Unit (ReLU), and Leaky ReLU that perform different calculations on the input depending upon intended purposes. The purpose of the

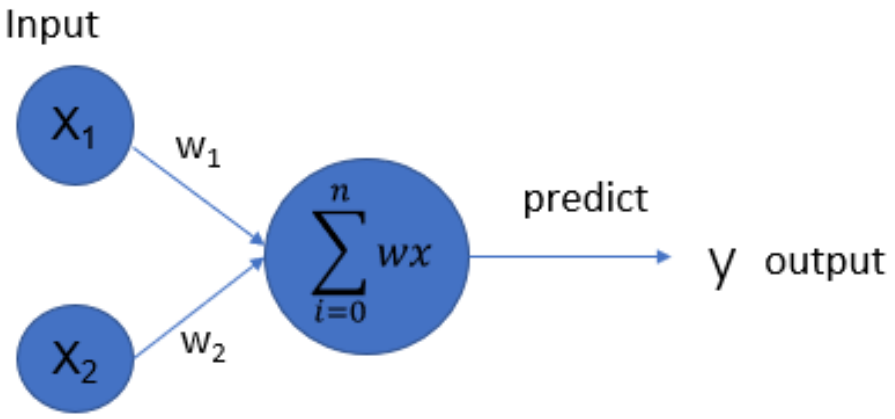

Fig. 3. Single layer perceptron consisting of input layers, activation function, and output value[3]

activation function is to map values of the input into resulting values ranging between 0 to 1, or -1 to 1, depending upon the function used. In our case, we used ReLU as our activation function for the hidden layers within the fully connected layer of the ANN.

The ReLU activation function is a linear activation function that outputs positive values and sets negative values to zero (see Figure 4). Of the various activation functions, ReLu is commonly selected for ANNs due to its ease of use in training and its overall good performance. The pattern of the sum of weighted values passed through activation functions, calculated from one perceptron to another within each layer of the ANN, continues until it reaches the output layer. This final neuron uses the Sigmoid activation function for binary classification (see Figure 5).

$$\sigma(x) = \frac{1}{1 + e{-x}} \tag{1}$$

This final output layer in our model classifies images into a binary outcome of 1 or 0, or in this case, cancerous or benign.

Once the output layer makes the prediction $\hat{y}$, the estimate is evaluated against the true value. A cost function is then computed. The end goal of the cost function is to take the error between the predicted value and actual value and calculate a loss. The cost has the form:

$$C = \frac{1}{2n} \Sigma y(x) - a^L(x)^2 \tag{2}$$

Where n corresponds to the number of training samples; x is the individual training sample; the desired output is denoted by y=y(x); L indicates the number of layers in the network; and aL = aL(x) is the vector of activations[3].

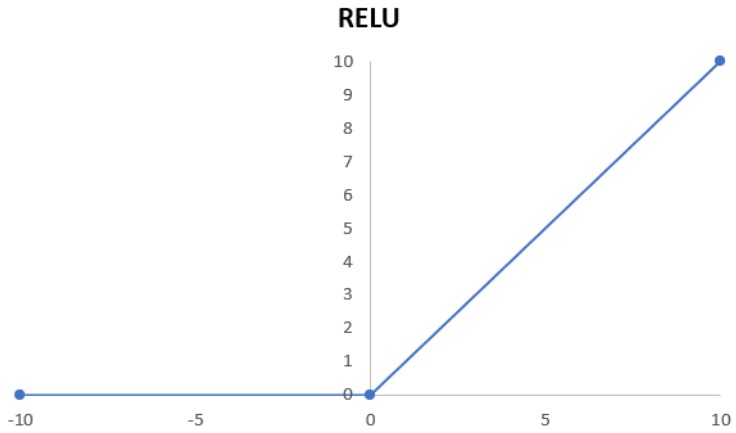

Fig. 4. Rectified linear activation function outputs positive values and sets negative values to zero (max(0,x))[3]

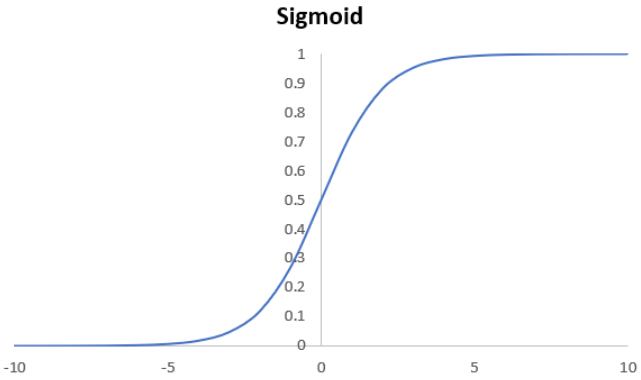

Fig. 5. Sigmoid function for binary classification where the sigmoid function is calculated as 1 divided by 1 plus Euler's number to the negative power of x (weighted value for that node)[3]

Once the cost function is calculated, the neurons' weights are adjusted through a process called back-propagation as the ANN trains epoch to epoch. The target of back-propagation is the calculation of the partial derivative of the cost function with respect to any weight w and bias b. Back-propagation utilizes a stochastic process whereby values are randomly chosen and then the partial derivatives of the cost function are computed, thus allowing the loss function to achieve a global minimum[16].

The rationale behind back-propagation is to re-weight neurons, bolstering those that performed well and penalizing those that under-performed. This is the essence of learning for ANNs. New weights for the neurons are calculated by way of taking the old weight minus the calculated derivative times the learning rate given from the model. A positive derivative rate signifies the new weight should be reduced. A negative derivative translates into an increase in weight, leading to an increase in error, and thus a larger new weight. However, if the derivative is zero, then a stable minimum has been reached and there should be no revision on the weight.

The process of computing the loss function, back-propagation, and re-weighting continues based on the number of epochs specified for the CNN. With each iteration the ultimate goal is to minimize the cost function and get $\hat{y}$, the predicted value, equal to the true y value, thus achieving 100 percent accuracy. Recent improvements have been made to the already performant CNN. Google created an Inception family of networks where the input is processed by multiple parallel convolutional branches, culminating in outputs merging back into a tensor[19].

In recent years, more advances in the CNN architecture have been made. The team at Microsoft discovered that as deeper networks start converging, accuracy becomes saturated and then starts degrading. This was found to be not due to over-training, which is common to many learning models. However, when the back-propagation algorithm propagates a loss signal from the output to earlier layers in the network, the feedback signal must pass through a deep stack of layers, and as a result may be lost. The team then added residual connections to the CNN model to address the vanishing gradient problem. Residual connections create a skip connection that adds an output tensor to a deeper layer in the network, thus avoiding the problem of the vanishing gradient (see Figure 6)[10]. Skip functions work by skipping one or more layers via performing an identity mapping, where the outputs are added to the outputs of the stacked layers[10]. The theory under evaluation by Microsoft is that if the layers are constructed as identity mappings, models with deeper layers should have training errors equivalent to shallower networks.

Another approach to improve performance and accuracy of the CNN model was taken by the team at Google, called Inception[19]. The team noted that the typical method of improving performance of deep neural networks was to increase the depth and width of layers. However, this approach tends towards a

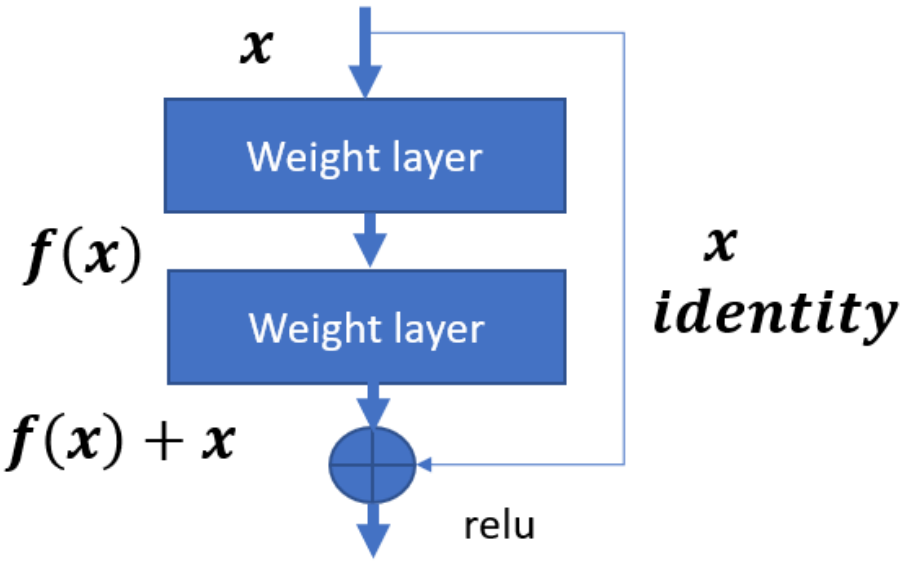

Fig. 6. Skip function in residual block

large number of parameters which can lead to a decrease in model generalization (over-fitting). Moreover, the computation resource requirements also increase. Layers chained within the convolutional neural network have a uniform increase in filter results, resulting in a quadratic increase in computation. This can be considered inefficient and wasteful, especially if the weights are updated to nearly zero. To resolve this, the team at Google proposed moving from a fully connected to sparsely connected architecture, even within the convolutions.

Inception seeks to approximate a sparse architecture and aggregates using 1x1 convolutions before the more computationally expensive 3x3 and 5x5 convolutions. Furthermore, max-pooling layers of stride two are used to reduce the grid resolution (see Figure 7). The Inception module proved itself to be accurate by winning the ImageNet Large-Scale Visual Recognition Challenge 2014 (ILSVRC 2014)[19]. More recently, a lighter CNN model that contains fewer weight parameters and more optimal floating-point operations resulted in overall higher accuracy than Inception. This new model developed by Francois Chollet, founder of the Keras deep learning library, is called Depthwise separable convolution network[1]. The depthwise separable convolution model is invoked by the SeparableConv2D object from the Keras library. This layer performs a spatial convolution on each channel of its input separately and then mixes the output channels via a point-wise convolution. The advantage of depthwise convolutions is that it is much less computationally intensive.

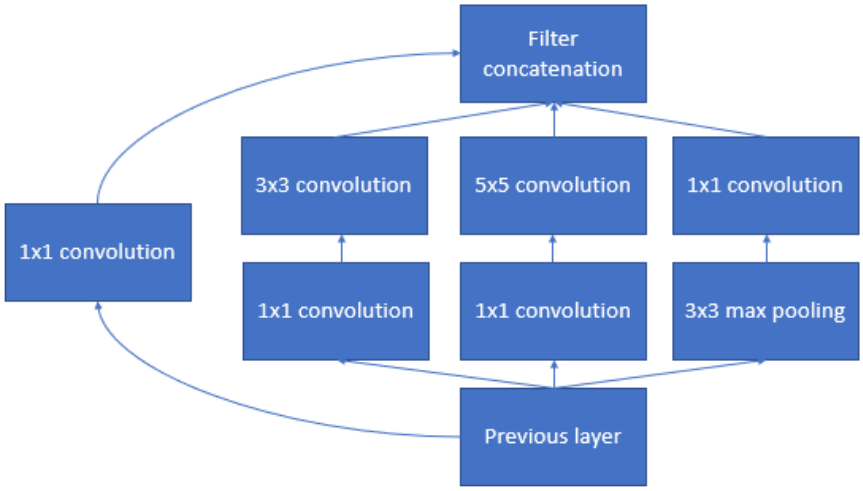

Fig. 7. Inception module shown with dimension reduction[19]

With CNNs, typically RGB images have 3 channels. When a convolution is done such as a 5x5 convolution, a 5x5x3 multiplication occurs when the kernel moves. Depthwise convolutions only do one channel at at time, resulting in 5x5x1. Furthermore, a separate operation is performed using 1x1x3 kernel. The depthwise separable convolution architecture forms the basis of the Xception architecture, which was shown to be more accurate than ResNet, while still incorporating the skip function introduced by ResNet. The Xception architecture furthermore builds upon the Inception model but replaces the Inception modules with depthwise separable convolutions. The basic theory of the Xception module is that cross-channel correlations and spatial correlations in the feature maps of convolutional neural networks can be decoupled, through the use of depthwise separable convolutions[6]. Newer yet is a Neural Architecture Search (NAS) framework developed by Google brain which combines reinforcement learning with a Recurrent Neural Network. This latest neural network, called NASNet, achieved the top accuracy score on ImageNET over previous top performers Inception and Xception (see Table 1).

NASNet is constructed such that a controller recurrent neural network (RNN) samples a series of its child neural networks that contain different architectures. The child networks are trained and tested against a validation test set. The accuracy measurements obtained from the child networks then update the controller RNN such that the controller generates new and better performing architectures over a number iterations, given a set of two initial hidden states (see Figure 9). Rather than relying solely on architecture engineering, this structure allows for model learning directly on blocks of the dataset that can then be transferred to the overall set[23].

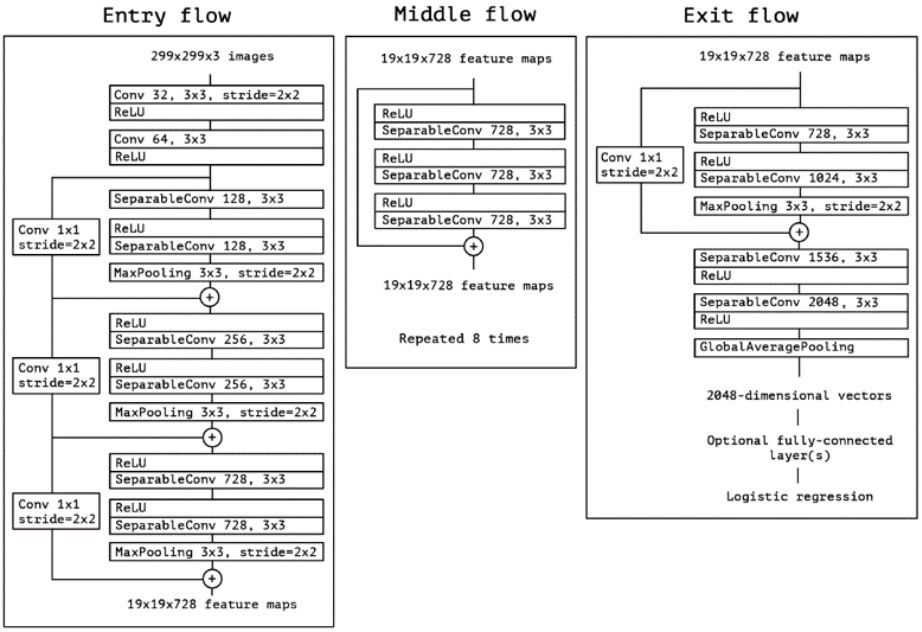

Fig. 8. Xception architechure

Table 1. NASNet accuracy compared to other models such as Xception and Inception.

Model	Image Size	Parameters	Mult-Adds	Top 1 Acc.	Top 5 Acc.
Inception V2	224x224	11.2 M	1.95 B	74.8%	92.2%
NASNet-A (5 @ 1538)	**299x299**	**10.9 M**	**2.35 B**	**78.6%**	**94.2%**
Inception V3	299x299	23.8 M	5.72 B	78.8%	94.4%
Xception	299x299	22.8 M	8.38 B	79.0%	94.5%
Inception ResNet V2	299x299	55.8 M	13.2 B	80.1%	95.1%
NASNet-A (7 @ 1920)	**299x299**	**22.6 M**	**4.93 B**	**80.8%**	**95.3%**
ResNeXt-101 (64 x 4d)	320x320	83.6 M	31.5 B	80.9%	95.6%
PolyNet	331x331	92 M	34.7 B	81.3%	95.8%
DPN-131	320x320	79.5 M	32.0 B	81.5%	95.8%
SENet	**320x320**	**145.8 M**	**42.3 B**	**82.7%**	**96.2%**
NASNet-A (6 @ 4032)	**331x331**	**88.9 M**	**23.8 B**	**82.7%**	**96.2%**

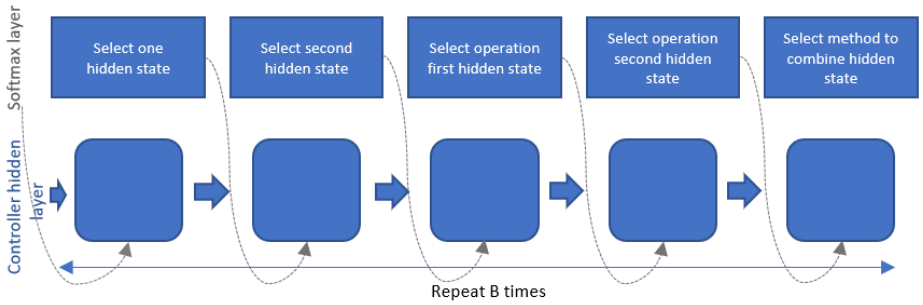

Fig. 9. NASNet controller RNN constructing new convolution block

5 Methodology

Several models with varying key characteristics were tested to identify the best performing method for classifying leukemic cells. The best performing method was then further developed to optimize efficiency and performance.

5.1 Feature extraction in SVM

SVM is a traditional classification method that requires pre-determined input features for modeling. Additionally, successful prediction hinges on comprehensive feature extraction from the images in the dataset. To ensure thoroughness in this process, we extracted features in accordance to the research completed by Patel et. al, including: color features (mean color values); geometric features (perimeter, radius, area, rectangularity, compactness, convexity, concavity, symmetry, elongation, eccentricity, solidity); texture features (entropy, energy, homogeneity, correlation); statistical features (skewness, mean, variance)[15]. A multi-dimension hyperplane decision boundary, the result of the SVM, then determined the categorization of the cell image into a class: cancerous or benign. The performance of this methodology is used as a point of reference to understand the gains from our tuned neural network. See Appendix E for implementation details.

5.2 Transfer learning from CNN models

Training convolutional neural networks can potentially take several days or weeks on large datasets to effectively learn. Transfer learning leverages previously solved problems for related tasks to reduce the time needed to build a model from the ground up. In practice, most deep learning models are built for

specific datasets and domains, but transfer learning seeks to utilize knowledge gained on previous tasks to refine new models[17]. We explored four neural network architectures, AlexNet, InceptionV3, Xception, and NASNetLarge, that had been primed for solving other problems and assessed their success on our data. We then selected the model with the best performance (Xception) for further optimization. Xception was trained on ImageNet with over 14 million images and 20,000 categories, as well as on Google's internal JFT image dataset, which contains over 350 million images and 17,000 classes[6]. We leveraged the weights from the pre-trained Xception architecture and utilized it as the starting point to train on our ALL dataset, thus allowing us to reduce training time and harness the knowledge gained from having previously been trained on image repositories such as ImageNet. Additionally, we added to the top of the architecture a global average pooling layer and Dense layer for classification for our dataset. See Appendix A-C for implementation details.

5.3 Light weight network

The transferred Xception model performs well on this dataset without adjustments to the architecture. Having determined the best framework for classifying ALL, we also evaluated if a lighter weight model could be developed based on the Xception architecture. Through experimentation, a lighter weight model was generated that contained only 1,248,060 parameters and 13 layers (including pooling layers) and was trained on our ALL dataset (see Appendix D). The model incorporated data augmentation and a skip function, that was first introduced with ResNET. Depthwise Separable Convolutional layers were used in conjunction with Batch Normalization layers, which normalize and scale the layers. In our model optimizer, Stochastic Gradient Descent with Nestorov achieved higher accuracy scores compared to Adam, AdamMax, RMSPro. Furthermore, a learning rate of 0.02 and momentum 0.4 was utilized. See Appendix D for implementation details.

6 Results

6.1 SVM results

The SVM resulted in an accuracy of 76%, precision of 69%, recall of 50%, and AUC of 69% with corresponding ROC curve as shown in Figure 10. This confirms our initial hypothesis that a well-tuned neural network will outperform SVM on this complex image set.

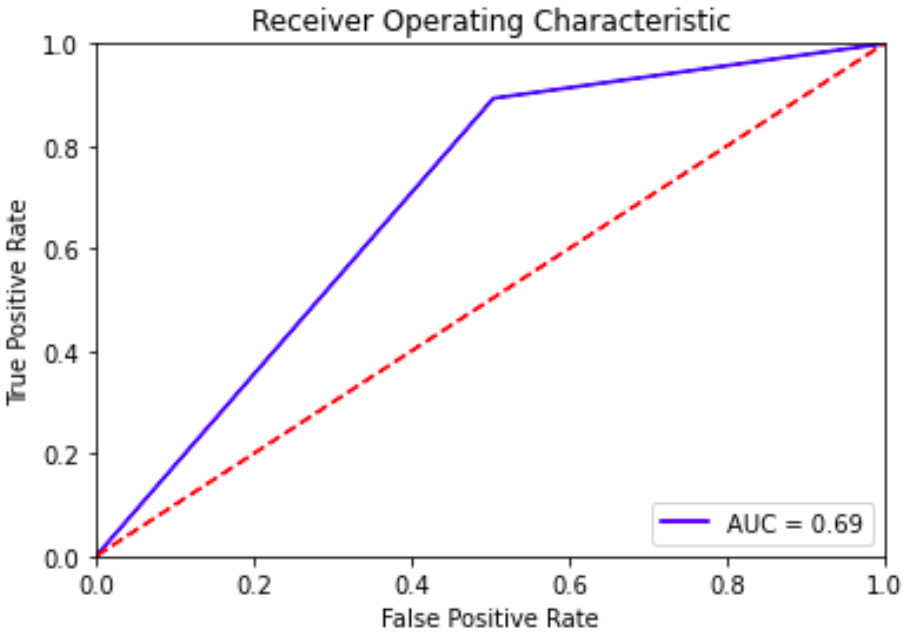

Fig. 10. ROC curve of SVM predictions

6.2 Deep NN results

We used the training dataset for model training and validation dataset for prediction. Figure 11 shows the prediction accuracy for each of four models we tested: AlexNet, InceptionV3, Xception and NASNetLarge. AlexNet gave us the lowest prediction accuracy at 68% among the 4 models. For the other three models, the accuracy on both the training and validation datasets were significantly higher, reaching 99% and 90%, respectively.

When examining the trends of accuracy changing with the increase of epoch numbers, we noticed that these three models display different patterns. The accuracy on validation dataset using Xception increased quickly to 85-90% and stabilized in that range throughout the 30 epochs tested. Although InceptionV3 model yields a similar accuracy on the validation dataset, the variation of accuracy within 30 epochs is clearly greater than that from Xception. Interestingly, the NASNetLarge model yielded a significantly lower accuracy at the early stage of training and then increased to 90% accuracy after 20 epochs. This may suggests that we had overfitting with the NASNetLarge model since it has the largest numbers parameters and most complicated architecture among all models tested. NASNetLarge also displayed noticeable variations even after it reaches a 90% accuracy on validation dataset. Additionally, the lightweight model based on Xception framework was run for 10 iterations. It achieved training accuracy of

91.07% and validation accuracy of 82.53%. The lighter weight model was not as accurate as the other robust models, but shows potential. Overall, these results suggest that Xception NN fits our data the best.

In the medical domain, false negatives are considered a severe result. In our study, we attempt to address early detection with ALL. A false negative in this domain is equivalent to a patient having been tested and told they were negative, but in fact ALL was present. An important measure for False Negatives is the recall rate. To this end, we plotted out the recall rates on the validation dataset for all the models along with precision and AUC values in Figure 12. Similar to what has been observed from accuracy plotting in Figure 11, the trend with recall, precision and AUC progression demonstrates again that InceptionV3, Xception and NASNetLarge perform significantly better than AlexNet. Xception model was the best among all four model as it quickly reaches to the best values and is able to maintain its high performance. Critically, Xception model was able to achieve an impressive a 98% recall rate, suggesting an exciting prospect of it being used in clinical applications.

To investigate why some models performed better than others, we obtained the numbers of trainable parameters and layers for each model (Table 2). The parameter numbers partially suggest the model complexity. More parameter numbers will generally require longer computation time for training. The number of layers in a model is an indicator of model depth and complexity as well. As shown in Figure 13, AlexNet has more parameters than both InceptionV3 and Xception, but with significantly smaller number of layers. The fact that AlexNet is the least efficient model among all suggests that the model depth is a much more valuable factor to consider than merely the number of parameters when we construct a model. On the other hand, NASNetLarge has the most parameters and layers, yet it is not the best model in our testing. Considering its outstanding performance on training data, but not as high performance on the validation data as shown in Figure 11 and Figure 12, we suggest that a model with too many parameters and layers is more likely to over-fit the training data and thus deteriorate its performance on testing or real data. As was discussed in "Explanation of ANNs and CNNs", the likely reasons that Xception outperformed InceptionV3 are that the former uses a depthwise scalable convolution and also incorporates some key features (residual connecting) from ResNet. In short, based on our testing results, to efficiently classify cell images as challenging as ours, we need a model that not only has considerable model depth, but also addresses issues that arise with more complex models (i.e., vanishing gradient) through the implementation of depthwise scalable convolutions and long range residual connection functionality.

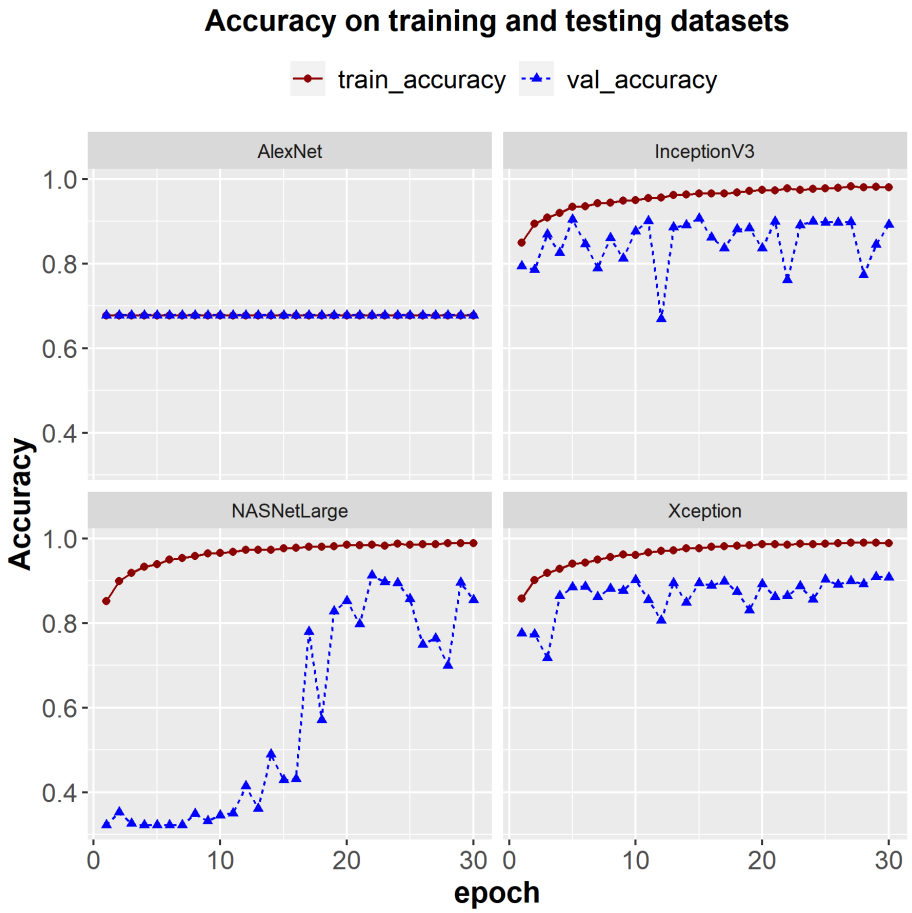

Fig. 11. Prediction accuracy from different deep Neural Networks

Table 2. The numbers of trainable parameters and layers for five tested models.

Model	Trainable parameters (millions)	Layers
AlexNet	62	16
InceptionV3	22	312
Xception	21	133
NASNetLarge	86	1040
LightWeightXception	1.2	9

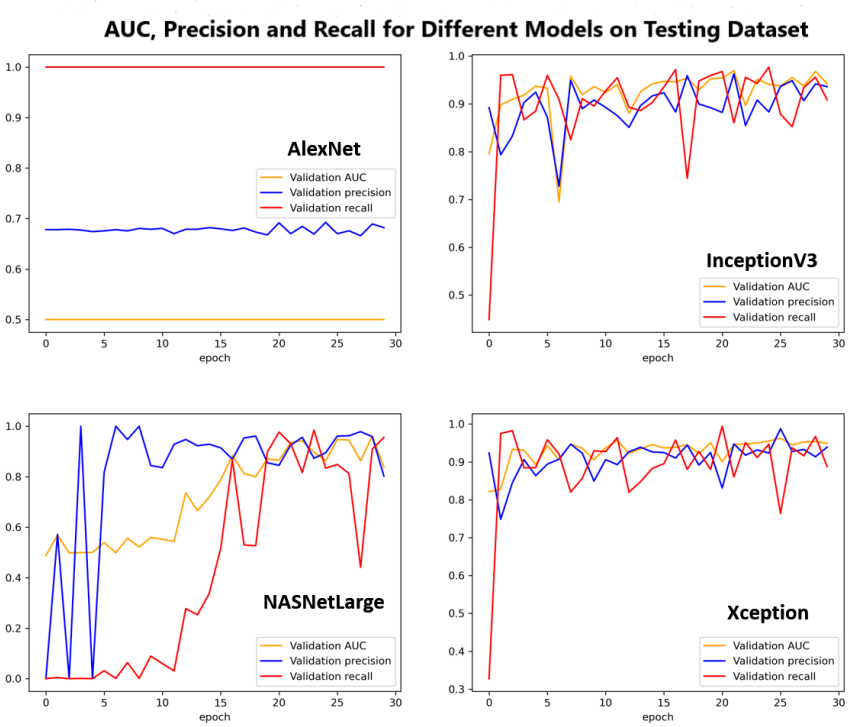

Fig. 12. The AUC, precision and recall values from different deep Neural Networks on testing dataset.

7 Ethics

The implementation of our model in practice hinges on the primary ethical question, whether society can trust an algorithm enough in order to relinquish any portion of decision making from a human when human life is in consideration. As presented by Carter et al., even if the tested model performance appears completely accurate and works in concert with a pathologist, inevitably a fraction of diagnosis responsibility would shift to a machine, blurring the explicable with the inexplicable. Also, the medical community requires rigorous testing and validation prior to implementation, and a deep learning model is no exception to these processes. If the model is accepted by the medical community and corresponding regulators, then ethical questions must be considered as well, including internal and external privacy controls on personal identifiable information (PII) when data is collected, analyzed, and stored[5].

The issue of confidence in an automated diagnosis method can be assuaged through thorough testing, both computationally and in line with medical practices, and direct comparison to current day practices[5]. If a machine can make equal or better determinations, the question of ethics changes. The reasons for not adopting these models must be concerning enough ethically to trump improved performance.

As mentioned previously, one cannot assess a neural network's decision making; we can only assess the results and the logic behind implementation and function. Carter et al. also presents that this is unlike most medical practices today, where doctors and practitioners can explain diagnostics to their patients. The doctor, who is ultimately issuing the diagnosis, is now forced to be accountable for automated results without the ability to explain how the results are generated. In an extreme worst case, a model could return a false negative, a professional could then quickly inspect and accept the results as accurate, and the patient could unknowingly progress with life threatening Leukemia. If the patient seeks legal action, the doctor would be legally accountable for not only their decision but technology he or she cannot explain. To control for this, there must be some regulatory policy imposed on machine learning models to release medical professionals from accountability for the output.

In further regards to policy, introducing AI as common practice would require expansion of current regulatory bodies[5]. After the model is developed and adequately tested, it must be maintained and routinely assessed. Without proper governance, an external authority, and established timelines, such maintenance could be ignored or deprioritized since it requires time and resources aside from day to day medical operations. Not only would new policies be developed, but also these regulators would have to expand their scope to include technical oversight.

Model compliance is one issue, and data compliance is another. Carter et. al also opines on data storage, retention, and confidentiality as some common issues in the field of data science when human data is involved. Our Neural Network depends on human blood sample specimens, which must be collected and stored according to existing guidelines. Additionally, consent must be given by the patient for future use in retraining the model, in consonance with most modeling and privacy agreements today. The image data should also not be traceable to a patient when stored for modeling purposes; it should be treated to the same level of confidentiality as the patient's medical record. The model developer or maintainer does not need to know the identity of the person from which the image came[5]. Access and identification should be strictly controlled on a need to know basis.

8 Conclusion

We have shown that identifying ALL at the cellular level can be achieved with the proper engineering and adequate data and computing resources. Our experimentation has revealed that high performing models tend to rely on architectures with greater depth rather than a greater number of parameters. We also exhibited that utilizing models trained for solving different tasks through transfer learning proved to be an effective method for cancer detection. The computational and time cost is less relative to building a complex model from scratch with large volumes of domain data. In particular, the Xception model was shown to be a strong performing ALL classifier when applied through transfer learning, achieving accuracy of 99% on training and 91% on the validation set. Additionally, the light weight model trained solely on the cell images reflected the performance efficiencies obtained through the use of depthwise separable convolutional layers and other characteristics of the Xception model.

The potential is promising for a less complex and unique architecture to meet Xception's ability to classify leukemic cells. Our developed model requires more rounds of training and a larger collection of cell images to be more competitive. Additionally, the incorporation of reinforcement learning could boost performance, particularly in this setting, where the image set is relatively small. Cell images can be collected over time to incrementally build up the dataset. Reinforcement learning then provides the framework to train networks on these smaller batches of data to later combine to the main model.

Overall, the ability of the Xception model to generalize well is the most encouraging. It was trained on millions of images that are not related to blood cells, yet it is still able to achieve performance comparable to other models in the ALL domain. This has further implications for applying transfer learning to other cancer detection problems. Xception can be used as a starting point for other diagnostic neural networks, shrinking the time and development cost

hurdles and establishing a high performing benchmark for further optimization. Convolutional Neural Networks can be a powerful medical tool not only for Acute Lymphoblastic Leukemia, but other cancers as well.

References

1. Applications, https://keras.io/applications
2. Key statistics for acute lymphocytic leukemia (all), https://www.cancer.org/cancer/acute-lymphocytic-leukemia/about/key-statistics.html
3. Nielsen, m.a.: Neural networks and deep learning, http://neuralnetworksanddeeplearning.com/
4. Ahmed, N., et al.: Identification of leukemia subtypes from microscopic images using convolutional neural network. Diagnostics **9.3**(12) (2019)
5. Carter, S., Rogers, W., Than Win, K., Frazer, H., Richards, B., Houssami, N.: The ethical, legal and social implications of using artificial intelligence systems in breast cancer care. The Breast **49**, 25–32 (2020)
6. Chollet, F.: Xception: Deep learning with depthwise separable convolutions. Web (2017)
7. Chollet, F.: Deep Learning with Python. No. 1, Manning Publishing, Shelter Island, NY (2018)
8. Gupta, A., et al.: All challenge dataset of isbi 2019 (2019), https://doi.org/10.7937/tcia.2019.dc64i46r"
9. Haworth, C., Heppleston, A.D., Jones, P.H.M., Campbell, R.H., Evans, D.I., Palmer, M.K.: Routine bone marrow examination in the management of acute lymphoblastic leukemia of childhood. Royal Manchester Children's Hospital **34**, 483–485 (1981)
10. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. Open Access **34**, 483–485 (2015)
11. Jeeva, M.: The scuffle between two algorithms - neural network vs. support vector machine (September 2018), https://medium.com/analytics-vidhya/the-scuffle-between-two-algorithms-neural-network-vs-support-vector-machine-16abe0eb4181
12. Jha, K.K., Dutta, H.S.: Mutual information based hybrid model and deep learning for acute lymphocytic leukemia detection in single cell blood smear images. Computer Methods and Programs in Biomedicine **179** (2019)
13. Maron, R., et al.: Systematic outperformance of 112 dermatologists in multiclass skin cancer image classification by convolutional neural networks. European Journal of Cancer **119**, 57–65 (2019)
14. Paswan, S., Rathore, Y.K.: Detection and classification of blood cancer from microscopic cell images using svm knn and nn classifier. International Journal of Advanced Research, Ideas and Innovations in Technology **3** (2017)
15. Patel, N., Misha, A.: Automated leukemia detection using microscopic images. Procedia Computer Science **58**, 635–642 (2015)
16. Raiko, T., Valpola, H., LeCunn, Y.: Deep learning made easier by linear transformations in perceptrons. vol. 22, pp. 924–932. Proceedings of the Fifteenth International Conference on Artificial Intelligence and Statistics (2012)
17. Sarkar, D.: A comprehensive hands-on guide to transfer learning with real-world applications in deep learning (November 2018)

18. Shafique, S., Tehsin, S.: Acute lymphoblastic leukemia detection and classification of its subtypes using pretrained deep convolutional neural networks. Technology in Cancer Research and Treatment **17** (2018)
19. Szegedy, C., et al.: Going deeper with convolutions. vol. 10, pp. 1–9. 2015 IEEE Conference on Computer Vision and Pattern Recognition (2015)
20. Thanh, T., et al.: Leukemia blood cell image classification using convolutional neural network. International Journal of Computer Theory and Engineering **10(2)** (2018)
21. Yoo, S., Gujrathi, I., Haider, M., Khalvati, F.: Prostate cancer detection using deep convolutional neural networks. Scientific Reports **9** (2019)
22. Zhang, N., et al.: Skin cancer diagnosis based on optimized convolutional neural network. Artificial Intelligence in Medicine **102** (2020)
23. Zhoph, B., et al.: Learning transferable architectures for scalable image recognition. IEEE/CVF Conference on Computer Vision and Pattern Recognition (2018)
24. Zhu, Y., et al.: Application of convolutional neural network in the diagnosis of the invasion depth of gastric cancer based on conventional endoscopy. Gastrointestinal Endoscopy **89(4)** (2019)

9 Appendix A

```
# InceptionV3 transfer code
import matplotlib.pyplot as plt
from numpy.random import seed
seed(1)

import os
import tensorflow
from tensorflow.keras import applications
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras import optimizers
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dropout, Flatten, Dense,
    GlobalAveragePooling2D
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam

class DetectCancer:

  def _init_(self,image_dir,model_name,train_set,epoch,image_size,batch):

    self.image_dir = image_dir
    self.model_name = model_name
    self.train_set = train_set
    self.epochs = epoch
    self.image_size = image_size
    self.batch = batch

  def createModel(image_dir,model_name,train_set,epoch,image_size,batch):
```

```
  base_model = applications.InceptionV3(weights='imagenet', include_top
      =False, input_shape=(image_size, image_size, 3))
  model_top = Sequential()
  model_top.add(GlobalAveragePooling2D(input_shape=base_model.
      output_shape[1:],data_format=None)), model_top.add(Dense(256,
      activation='relu'))
  model_top.add(Dropout(0.5))
  model_top.add(Dense(1, activation='sigmoid'))

  # We now put the new top onto the network
  model = Model(inputs=base_model.input, outputs=model_top(base_model.
      output))

  model.summary()

  model.compile(optimizer=Adam(lr=0.0001, beta_1=0.9, beta_2=0.999,
      epsilon=1e-08,decay=0.0), loss='binary_crossentropy', metrics=['
      accuracy'])

  DetectCancer.augmentImages(image_dir,model_name, train_set, model,
      epoch,image_size,batch)

def augmentImages(image_dir,model_name,train_set,model,epoch,image_size
    ,batch):


  train_datagen = ImageDataGenerator(rescale = 1./255, #scaled to zero
      and one values
                                shear_range = 0.2,
                                zoom_range = 0.2,
                                rotation_range = 40,
                                height_shift_range = 0.2,
                                width_shift_range = 0.2,
                                horizontal_flip = True,
                                fill_mode='nearest',
                                validation_split=0.10)

  training_set = train_datagen.flow_from_directory(train_set,
                                            target_size = (image_size
                                                , image_size),
                                            subset = 'training',
                                            batch_size = batch,
                                            class_mode = 'binary')

  testing_set = train_datagen.flow_from_directory(train_set,target_size
       = (image_size, image_size),
                                        subset='validation',
                                        batch_size = batch,
                                        class_mode = 'binary')
```

```
    history = model.fit_generator(training_set,steps_per_epoch =
        training_set.samples/10,validation_data = testing_set,
        validation_steps=testing_set.samples/10,epochs=epoch)

    print(history.history.keys())

    DetectCancer.saveModel(image_dir,model_name,model)

  def saveModel(image_dir,model_name,lk_model):
    lk_model.save(image_dir + os.sep + model_name)

if __name__ == "__main__":

  image_dir = '/work/greencenter/ychu/Capstone_project/CNN_models/
      InceptionV3'
  os.chdir
  model_name = 'InceptionV3.h5'
  train_set = '/work/greencenter/ychu/Capston_project/C-NMC_combined'
  epoch = 30
  image_size = 299
  batch = 10

  DetectCancer.createModel(image_dir,model_name,train_set,epoch,
      image_size,batch)
```

10 Appendix B

```
# Xception transfer code
from numpy.random import seed
seed(1)

#import libraries
import matplotlib.pyplot as plt
import os
import tensorflow

from tensorflow.keras import applications
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras import optimizers
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dropout, Flatten, Dense,
    GlobalAveragePooling2D
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam

class DetectCancer:
```

```
def _init_(self,image_dir,model_name,train_set,epoch,image_size,batch):

  self.image_dir = image_dir
  self.model_name = model_name
  self.train_set = train_set
  #self.train_size = train_size
  self.epochs = epoch
  self.image_size = image_size
  self.batch = batch

def createModel(image_dir,model_name,train_set,epoch,image_size,batch):

  base_model = applications.Xception(weights='imagenet', include_top=
      False, input_shape=(image_size, image_size, 3))

  model_top = Sequential()
  model_top.add(GlobalAveragePooling2D(input_shape=base_model.
      output_shape[1:], data_format=None)),
  model_top.add(Dense(256, activation='relu'))
  model_top.add(Dropout(0.5))
  model_top.add(Dense(1, activation='sigmoid'))

  # We now put the new top onto the network
  model = Model(inputs=base_model.input, outputs=model_top(base_model.
      output))

  model.summary()

  model.compile(optimizer=Adam(lr=0.0001, beta_1=0.9, beta_2=0.999,
      epsilon=1e-08,decay=0.0), loss='binary_crossentropy', metrics=['
      accuracy'])

  DetectCancer.augmentImages(image_dir,model_name, train_set, model,
      epoch,image_size,batch)

def augmentImages(image_dir,model_name,train_set,model,epoch,image_size
    ,batch):

  train_datagen = ImageDataGenerator(rescale = 1./255, #scaled to zero
      and one values
                                     shear_range = 0.2,
                                     zoom_range = 0.2,
                                     rotation_range = 40,
                                     height_shift_range = 0.2,
                                     width_shift_range = 0.2,
                                     horizontal_flip = True,
                                     fill_mode='nearest',
                                     validation_split=0.10)
```

```
    training_set = train_datagen.flow_from_directory(train_set,
                                                     target_size = (image_size
                                                         , image_size),
                                                     subset = 'training',
                                                     batch_size = batch,
                                                     class_mode = 'binary')



    testing_set = train_datagen.flow_from_directory(train_set,target_size
         = (image_size, image_size),
                                                 subset='validation',
                                                 batch_size = batch,
                                                 class_mode = 'binary')


    history = model.fit_generator(training_set,steps_per_epoch =
        training_set.samples/10,validation_data = testing_set,
        validation_steps=testing_set.samples/10,epochs=epoch)

    print(history.history.keys())

    DetectCancer.saveModel(image_dir,model_name,model)

  def saveModel(image_dir,model_name,lk_model):
    lk_model.save(image_dir + os.sep + model_name)

if __name__ == "__main__":

  image_dir = '/work/greencenter/ychu/Capstone_project/CNN_models/
      Xception'
  os.chdir(image_dir)
  model_name = 'Xception.h5'
  train_set = '/work/greencenter/ychu/Capstone_project/C-NMC_combined'
  epoch = 30
  image_size = 299
  batch = 10

  DetectCancer.createModel(image_dir,model_name,train_set,epoch,
      image_size,batch)
```

11 Appendix C

```
# NASNetLarge transfer code
from numpy.random import seed
seed(1)

#import libraries
```

```
import matplotlib.pyplot as plt
import os
import tensorflow

from tensorflow.keras import applications
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras import optimizers
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dropout, Flatten, Dense,
    GlobalAveragePooling2D
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam

class DetectCancer:

  def _init_(self,image_dir,model_name,train_set,epoch,image_size,batch):

    self.image_dir = image_dir
    self.model_name = model_name
    self.train_set = train_set
    self.epochs = epoch
    self.image_size = image_size
    self.batch = batch

  def createModel(image_dir,model_name,train_set,epoch,image_size,batch):

    base_model = applications.NASNetLarge(weights='imagenet', include_top
        =False, input_shape=(image_size, image_size, 3))

    model_top.add(GlobalAveragePooling2D(input_shape=base_model.
        output_shape[1:], data_format=None)),
    model_top.add(Dense(256, activation='relu'))
    model_top.add(Dropout(0.5))
    model_top.add(Dense(1, activation='sigmoid'))

    # We now put the new top onto the network
    model = Model(inputs=base_model.input, outputs=model_top(base_model.
        output))

    model.summary()

    model.compile(optimizer=Adam(lr=0.0001, beta_1=0.9, beta_2=0.999,
        epsilon=1e-08,decay=0.0), loss='binary_crossentropy', metrics=['
        accuracy'])

    DetectCancer.augmentImages(image_dir,model_name, train_set, model,
        epoch,image_size,batch)

  def augmentImages(image_dir,model_name,train_set,model,epoch,image_size
      ,batch):
```

```
    train_datagen = ImageDataGenerator(rescale = 1./255, #scaled to zero
        and one values
                                    shear_range = 0.2,
                                    zoom_range = 0.2,
                                    rotation_range = 40,
                                    height_shift_range = 0.2,
                                    width_shift_range = 0.2,
                                    horizontal_flip = True,
                                    fill_mode='nearest',
                                    validation_split=0.10)

    training_set = train_datagen.flow_from_directory(train_set,
                                                target_size = (image_size
                                                    , image_size),
                                                subset = 'training',
                                                batch_size = batch,
                                                class_mode = 'binary')



    testing_set = train_datagen.flow_from_directory(train_set,target_size
         = (image_size, image_size),
                                                subset='validation',
                                                batch_size = batch,
                                                class_mode = 'binary')


    history = model.fit_generator(training_set,steps_per_epoch =
        training_set.samples,10,validation_data = testing_set,
        validation_steps=testing_set.samples/10,epochs=epoch)

    print(history.history.keys())

    DetectCancer.saveModel(image_dir,model_name,model)

  def saveModel(image_dir,model_name,lk_model):
    lk_model.save(image_dir + os.sep + model_name)

if __name__ == "__main__":

  image_dir = '/work/greencenter/ychu/Capstone_project/CNN_models/
      NASNetLarge'
  os.chdir(image_dir)
  model_name = 'NASNetLarge.h5'
  train_set = '/work/greencenter/ychu/Capstone_project/C-NMC_combined'
  epoch = 30
  image_size = 331
  batch = 10
```

```
  DetectCancer.createModel(image_dir,model_name,train_set,epoch,
      image_size,batch)
```

12 Appendix D

```
#https://keras.io/examples/vision

import tensorflow as tf

from tensorflow import keras

from tensorflow.keras import preprocessing

from tensorflow.keras import layers

from tensorflow.keras.preprocessing.image import ImageDataGenerator

import numpy as np

import matplotlib.pyplot as plt

import os

from google.colab import drive

drive.mount('/gdrive')
image_dir = '/gdrive/My␣Drive/Colab␣Notebooks/data'
os.chdir(image_dir)

class LightWeightXception:

  def _init_(self,image_size,size,batch_size,train_set,train_datagen,
      train_ds,val_ds):
}
    self.image_size = image_size
    self.size = size
    self.batch_size = batch_size
    self.train_set = train_set
    self.train_datagen = train_datagen
    self.train_ds = train_ds
    self.val_ds = val_ds
}

  def genmodel(input_shape, num_classes):

      inputs = keras.Input(shape=input_shape)
      x = inputs
      x = layers.SeparableConv2D(32, 3, strides=2, padding="same")(x)
      x = layers.BatchNormalization()(x)
```

```
    x = layers.Activation("relu")(x)
    x = layers.SeparableConv2D(64, 3, padding="same")(x)
    x = layers.BatchNormalization()(x)
    x = layers.Activation("relu")(x)

    initialize residual
    resid_block = x
}
    for size in [128, 256, 512]:

        x = layers.Activation("relu")(x)
        x = layers.SeparableConv2D(size, 3, padding="same")(x)
        x = layers.BatchNormalization()(x)

        x = layers.Activation("relu")(x)
        x = layers.SeparableConv2D(size, 3, padding="same")(x)
        x = layers.BatchNormalization()(x)

        x = layers.MaxPooling2D(3, strides=2, padding="same")(x)

        # Project residual
        residual = layers.Conv2D(size, 1, strides=2, padding="same")(
            resid_block)
        x = layers.add([x, residual])
        resid_block = x
        }

    x = layers.SeparableConv2D(1024, 3, padding="same")(x)
    x = layers.BatchNormalization()(x)
    x = layers.Activation("relu")(x)

    x = layers.GlobalAveragePooling2D()(x)
    x = layers.Dropout(0.2)(x)

    outputs = layers.Dense(1, activation='sigmoid')(x)
    return keras.Model(inputs, outputs)

def set_params(image_size,size,batch_size,train_set,train_datagen,
    train_ds,val_ds,epoch):

  image_size =
  model = LightWeightXception.gen_model(input_shape=image_size + (3,),
      num_classes=2)
  keras.utils.plot_model(model, show_shapes=True)
  model.summary()

  epochs = epoch
  callbacks = [keras.callbacks.ModelCheckpoint("save_at_{epoch}.h5")]
```

```
    opt = keras.optimizers.SGD(learning_rate=0.02, momentum=0.4, nesterov
        =True, name='SGD')

    model.compile(optimizer=opt,loss="binary_crossentropy", metrics=["
        accuracy"])

    keras.utils.plot_model(model, show_shapes=True)

    history = model.fit(train_ds, epochs=epochs, callbacks=callbacks,
        validation_data=val_ds,)

    LightWeightXception.plotAcc(history)

  def plotAcc(history):

      acc = history.history['accuracy']
      val_acc = history.history['val_accuracy']
      loss = history.history['loss']
      val_loss = history.history['val_loss']
      epochs = range(1, len(acc) + 1)
      plt.plot(epochs, acc, 'bo', label='Training␣acc')
      plt.plot(epochs, val_acc, 'b', label='Validation␣acc')
      plt.title('Training␣and␣validation␣accuracy')
      plt.legend()
      plt.figure()
      plt.plot(epochs, loss, 'bo', label='Training␣loss')
      plt.plot(epochs, val_loss, 'b', label='Validation␣loss')
      plt.title('Training␣and␣validation␣loss')
      plt.legend()
      plt.show()

 }

image_size = (450, 450)
size = 450
batch_size = 32
epoch = 10

train_set = '/gdrive/My␣Drive/Colab␣Notebooks/data/comb_training'

train_datagen = ImageDataGenerator(rescale = 1./size,
                                   shear_range = 0.2,
                                   zoom_range = 0.2,
                                   rotation_range = 30,
                                   height_shift_range = 0.2,
                                   width_shift_range = 0.2,
                                   horizontal_flip = False,
                                   fill_mode='nearest',
                                   validation_split=0.15)
```

```
train_ds = train_datagen.flow_from_directory(train_set,
                                             target_size = image_size,
                                             subset = 'training',
                                             batch_size = batch_size,
                                             seed = 49,
                                             class_mode = 'binary')


val_ds = train_datagen.flow_from_directory(train_set,target_size =
    image_size,
                                           subset='validation',
                                           batch_size = batch_size,
                                           seed = 49,
                                           class_mode = 'binary')

LightWeightXception.set_params(image_size,size,batch_size,train_set,
    train_datagen,train_ds,val_ds,epoch)
```

13 Appendix E

```
test_datagen = ImageDataGenerator(rescale = 1./128)
image_size = 128
batch = 10661
# Tree 2
train_datagen = ImageDataGenerator(rescale = 1./128, #scaled to zero and
    one values
                                   shear_range = 0.2,
                                   zoom_range = 0.2,
                                   rotation_range = 40,
                                   height_shift_range = 0.2,
                                   width_shift_range = 0.2,
                                   horizontal_flip = True,
                                   fill_mode='nearest')
                                  # validation_split=0.15)


training_set = train_datagen.flow_from_directory(train_set,
                                             target_size = (image_size
                                                 , image_size),
                                             class_mode = 'binary',
                                             batch_size=batch)

testing_set = test_datagen.flow_from_directory(test_set,target_size = (
    image_size, image_size), class_mode = 'binary', batch_size=1867)

X_train, y_train = training_set.next()
X_test, y_test= testing_set.next()
X_both=np.concatenate([X_test,X_train],axis=0)
```

```
y_both=np.concatenate([y_test,y_train],axis=0)

from sklearn.model_selection import train_test_split
f_train,f_test,l_train,l_test=train_test_split(X_both, y_both, test_size
    =0.15, random_state=121)
#color features i.e. mean color values of image
avg_pic=np.zeros(10648)
apt=np.zeros(1880)
for i in range(0, 10648):
  im=f_train[i]
  # get shape
  w,h,d=im.shape
  # change shape
  im.shape = (w*h, d)
  # get average
  avgrgb=tuple(im.mean(axis=0))
  avg_pic[i]=((avgrgb[0]+avgrgb[1]+avgrgb[2])/3)
for i in range(0, 1880):
  im=f_test[i]
  # get shape
  w,h,d=im.shape
  # change shape
  im.shape = (w*h, d)
  # get average
  avgrgb=tuple(im.mean(axis=0))
  apt[i]=((avgrgb[0]+avgrgb[1]+avgrgb[2])/3)
#perimeter, radius, area, rectangularity, compactness, convexity,
    concavity, symmetry
from scipy import ndimage
from skimage import measure
rad=np.zeros(10648)
perims=np.zeros(10648)
area=np.zeros(10648)
rect=np.zeros(10648)
comp=np.zeros(10648)
convex=np.zeros(10648)
concav=np.zeros(10648)
elong=np.zeros(10648)
eccen=np.zeros(10648)
solidity=np.zeros(10648)
radt=np.zeros(1880)
perimst=np.zeros(1880)
areat=np.zeros(1880)
rectt=np.zeros(1880)
compt=np.zeros(1880)
convext=np.zeros(1880)
concavt=np.zeros(1880)
elongt=np.zeros(1880)
eccent=np.zeros(1880)
solidityt=np.zeros(1880)
```

```
for i in range(0, 10648):
  img=f_train[i]
  gray = img[:,:,0]
  blobs=gray>0
  perims[i]=measure.perimeter(gray)
  labels, nlabels = ndimage.label(blobs)
  properties = measure.regionprops(labels)
  for p in properties:
      min_row, min_col, max_row, max_col = p.bbox
      area[i]=p.area
      rad[i]=max(max_row - min_row, max_col - min_col)
      rect[i]=p.area/p.bbox_area
      comp[i]=p.area/(np.power(perims[i],2))
      convex[i]=measure.perimeter(p.convex_image)/perims[i]
      elong[i]=1-((max_row-min_row)/(max_col-min_col))
      eccen[i]=(max_col-min_col)/(max_row-min_row)
      solidity[i]=p.area/p.convex_area
for i in range(0, 1880):
  img=f_test[i]
  gray = img[:,:,0]
  blobs=gray>0
  perimst[i]=measure.perimeter(gray)
  labels, nlabels = ndimage.label(blobs)
  properties = measure.regionprops(labels)
  for p in properties:
      min_row, min_col, max_row, max_col = p.bbox
      areat[i]=p.area
      radt[i]=max(max_row - min_row, max_col - min_col)
      rectt[i]=p.area/p.bbox_area
      compt[i]=p.area/(np.power(perimst[i],2))
      convext[i]=measure.perimeter(p.convex_image)/perimst[i]
      elongt[i]=1-((max_row-min_row)/(max_col-min_col))
      eccent[i]=(max_col-min_col)/(max_row-min_row)
      solidityt[i]=p.area/p.convex_area

#Texture features
from skimage.feature import greycomatrix, greycoprops
from skimage.filters.rank import entropy
from skimage.morphology import disk
entrop=np.zeros(10648)
contr=np.zeros(10648)
homogen=np.zeros(10648)
energy=np.zeros(10648)
corr=np.zeros(10648)
for i in range(0, 10648):
  img=f_train[i]
  gray = img[:,:,0]
  gray=gray.astype(int)
  entrop[i]=entropy(gray, disk(5)).mean()
  g = greycomatrix(gray, [1], [2], levels=4,normed=True, symmetric=True)
```

```
  contr[i] = greycoprops(g, 'contrast')[0][0]
  homogen[i] = greycoprops(g, 'homogeneity')[0][0]
  energy[i] = greycoprops(g, 'energy')[0][0]
  corr[i]= greycoprops(g, 'correlation')[0][0]
entropt=np.zeros(1880)
contrt=np.zeros(1880)
homogent=np.zeros(1880)
energyt=np.zeros(1880)
corrt=np.zeros(1880)
for i in range(0, 1880):
  img=f_test[i]
  gray = img[:,:,0]
  gray=gray.astype(int)
  entropt[i]=entropy(gray, disk(5)).mean()
  g = greycomatrix(gray, [1], [2], levels=4,normed=True, symmetric=True)
  contrt[i] = greycoprops(g, 'contrast')[0][0]
  homogent[i] = greycoprops(g, 'homogeneity')[0][0]
  energyt[i] = greycoprops(g, 'energy')[0][0]
  corrt[i]= greycoprops(g, 'correlation')[0][0]

import cv2
from scipy import ndimage
# statistical features: skewness, mean, variance, gradient matrix
vx=np.zeros(10648)
vy=np.zeros(10648)
skx=np.zeros(10648)
sky=np.zeros(10648)
med=np.zeros(10648)
for i in range(0, 10648):
  img=f_train[i]
  gray = img[:,:,0]
  h,w = np.shape(gray)
  x = range(w)
  y = range(h)
  #calculate projections along the x and y axes
  yp = np.sum(gray,axis=1)
  xp = np.sum(gray,axis=0)
  #centroid
  cx = np.sum(x*xp)/np.sum(xp)
  cy = np.sum(y*yp)/np.sum(yp)
  x2 = (range(w) - cx)**2
  y2 = (range(h) - cy)**2
  sx = np.sqrt( np.sum(x2*xp)/np.sum(xp) )
  sy = np.sqrt( np.sum(y2*yp)/np.sum(yp) )
  X2,Y2 = np.meshgrid(x2,y2)
  #Find the variance
  vx[i] = np.sum(gray*X2)/np.sum(gray)
  vy[i] = np.sum(gray*Y2)/np.sum(gray)
  #skewness
  x3 = (x-cx)**3
```

```
  y3 = (y-cy)**3
  skx[i] = np.sum(xp*x3)/(np.sum(xp) * sx**3)
  sky[i] = np.sum(yp*y3)/(np.sum(yp) * sy**3)
  med_filt=ndimage.median_filter(gray, size=(16,16), output=np.float64,
      mode="reflect")
  med[i]=np.mean(med_filt[:][:])
vxt=np.zeros(1880)
vyt=np.zeros(1880)
skxt=np.zeros(1880)
skyt=np.zeros(1880)
medt=np.zeros(1880)

for i in range(0, 1880):
  img=f_test[i]
  gray = img[:,:,0]
  h,w = np.shape(gray)
  x = range(w)
  y = range(h)
  #calculate projections along the x and y axes
  yp = np.sum(gray,axis=1)
  xp = np.sum(gray,axis=0)
  #centroid
  cx = np.sum(x*xp)/np.sum(xp)
  cy = np.sum(y*yp)/np.sum(yp)
  x2 = (range(w) - cx)**2
  y2 = (range(h) - cy)**2
  sx = np.sqrt( np.sum(x2*xp)/np.sum(xp) )
  sy = np.sqrt( np.sum(y2*yp)/np.sum(yp) )
  X2,Y2 = np.meshgrid(x2,y2)
  Find the variance
  vxt[i] = np.sum(gray*X2)/np.sum(gray)
  vyt[i] = np.sum(gray*Y2)/np.sum(gray)
  skewness
  x3 = (x-cx)**3
  y3 = (y-cy)**3
  skxt[i] = np.sum(xp*x3)/(np.sum(xp) * sx**3)
  skyt[i] = np.sum(yp*y3)/(np.sum(yp) * sy**3)
  med_filt=ndimage.median_filter(gray, size=(16,16), output=np.float64,
      mode="reflect")
  medt[i]=np.mean(med_filt[:][:])

from sklearn.preprocessing import MinMaxScaler
from sklearn import svm, metrics

x_train = np.column_stack((rad, area,rect,comp,convex,concav,elong,eccen,
    solidity,entrop,contr,homogen,energy,
                       corr,vx,vy,skx,sky,med,avg_pic))
x_test = np.column_stack((radt, areat,rectt,compt,convext,concavt,elongt,
    eccent,solidityt,entropt,contrt,homogent,energyt,
                       corrt,vxt,vyt,skxt,skyt,medt,apt))
```

```
scaler = MinMaxScaler(feature_range=(0, 1))
#Normalize The feature vectors
x_train=np.where
x_train=np.where(np.isnan(x_train),0,x_train)
x_test=np.where(np.isinf(x_test),0,x_test)
x_test=np.where(np.isnan(x_test),0,x_test)
scaled_train= scaler.fit_transform(x_train)
scaled_test=scaler.fit_transform(x_test)
l_test=l_test.astype(int)

classifier = svm.SVC()
classifier.fit(scaled_train, l_train)
Now predict the value of the digit on the second half:
predicted = classifier.predict(scaled_test)
print(metrics.classification_report(l_test,predicted))
print(metrics.confusion_matrix(l_test,predicted))
```