Capstone Project

Venkata Ramarao

Machine Learning Engineer Nanodegree

05/07/2017

Definition

Project Overview

White blood cells (also known as WBC or leukocytes) help our body fight infections by attacking bacteria, viruses and other germs that invade the body. A count of leukocytes can help reveal several hidden and undiagnosed diseases. In a manual microscopic review of blood samples, pathologists minutely examine the count and morphology (i.e. size and shape) of white blood cells.

Red blood cells (or RBC, or erythrocytes) are the most common type of blood cells, and they outnumber WBCs by about 600:1. So, in an image of a blood sample, you will see mostly RBCs, with a few WBCs thrown in here and there. See Figure 1.

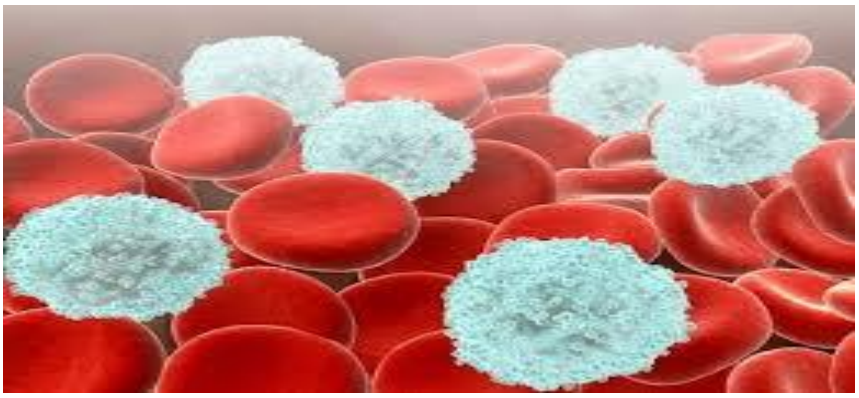


Figure 1. white blood cells in blood sample

In manual process pathologists analyze the blood sample and count the WBC (white blood cells), but this is not accurately defined the correct count and related disease, they use pre-defined approach to determine the health of the person. But if we use Image segmentation using deep learning supervised algorithm model, it accurately demarcates the boundary of WBC even when they are touching each other and identify correct count. This will improve the accuracy and speed of testing and yield better results.

I hope by using state of the art deep learning model for this image segmentation task will improve the accuracy of results and correctly identify the health of a person.

The dataset for this project originates from the SigTuple AI Challenge[1]. SigTuple released this data set to hire top AI talent for their engineering team. The data set is available at the hackathon[2] competition.

---

1. Hackathon: https://www.hackerearth.com/challenge/competitive/sigtuple-ai-challenge/
2. Data set: https://s3-ap-southeast-1.amazonaws.com/he-public-data/contests/SigTuple_data.tar

Problem Statement

The goal is to developing an efficient Deep Learning model using CNN (Convolutional Neural Networks) to accurately demarcate the boundary of white blood cells in microscopic images of blood

Metrics

We use Dice coefficient or F1 score for evaluation of the model on predicted masks with ground truth values.

The Dice score is often used to quantify the performance of image segmentation methods. There you annotate some ground truth region in your image and then make an automated algorithm to do it. You validate the algorithm by calculating the Dice score, which is a measure of how similar the objects are. So it is the size of the overlap of the two segmentations divided by the total size of the two objects. Using the same terms as describing accuracy, the Dice score is:

Dice score=number of true positives/number of positives + number of false positives

So the number of true positives, is the number that your method finds, the number of positives is the total number of positives that can be found and the number of false positives is the number of points that are negative that your method classifies as positive.

If the dice score is high, the model is correctly segmenting given images with right classes.

Analysis

Data Exploration

Structure of the data set:

```
├── Test_Data (61 files)
|    ├── 017532875DDF.jpg
|    ├── 029E137BB177.jpg
|    ├── 029E137BB179.jpg
|    ├── ...
└── Train_Data (328 files)
     ├── train-0.jpg
     ├── train-0-mask.jpg
```

├── train-100.jpg

├── train-100-mask.jpg

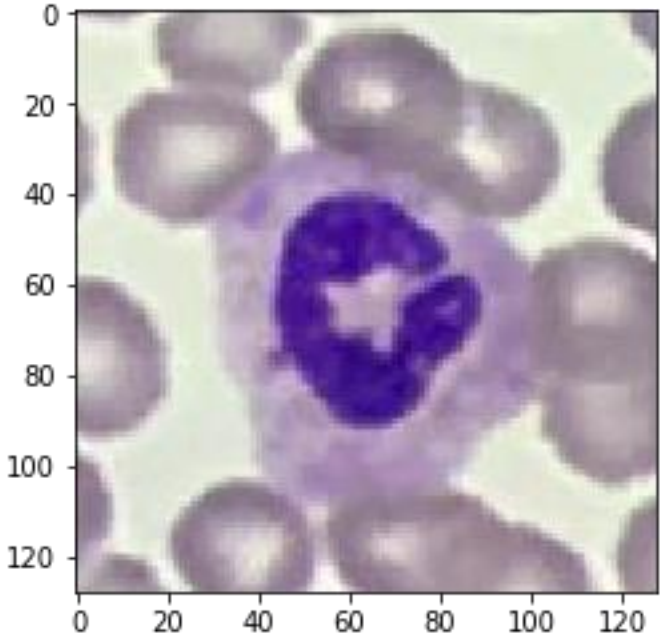
├── ...

The training set consists of 164 (128X128) patches showing WBCs, and the area has been demarcated in a mask file. The cell at the center of these patches are WBCs, while those surrounding the WBC are RBCs. The files are named like train-0.jpg, train-1.jpg, ... The corresponding mask files are named train-0-mask.jpg, train-1-mask.jpg, ..., respectively. There are also around 5 larger images (and corresponding masks) of blood, showing one or more WBCs in the image.

The test set consist of 61 larger images of blood smears of multiple dimensions, with three channels (R, G, B) from which we need to demarcate the WBC boundaries.

Both training and test data set have three colored channel images. The color information is not necessary because we only need the gray scale images that can discriminate the white blood cells from the remaining cells (red blood cells). There are far more pixels in our dataset that belong to red blood cells than there are pixels that belong to white blood cells. To compensate this class imbalance, we have used dice score as loss metric to predict the segmented masks.

Data Visualization



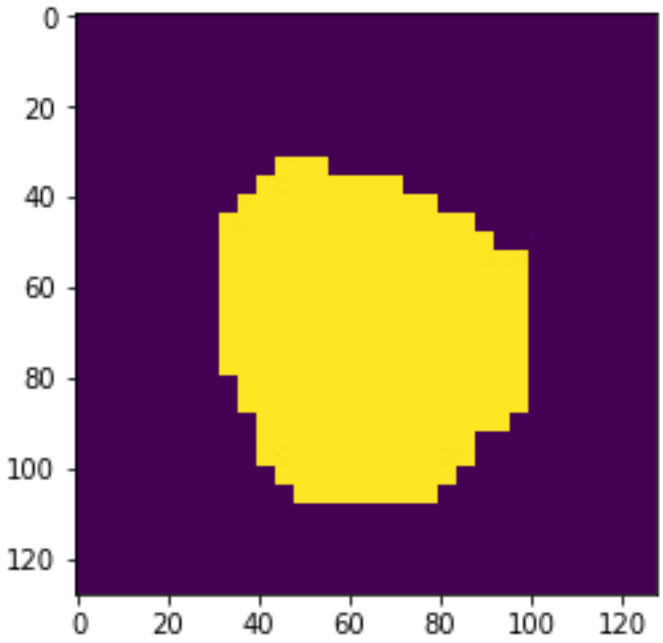


Figure 2. Blood cells with labelled white blood cell

An example of blood cells and associated white blood cell label in Figure 2.

The upper one is train image(train-0.jpg) with both WBCs (Blue region) and RBCs and the lower one is corresponding masked image(train-0-mask.jpg) with demarcated WBC boundary.

The training set have 128 * 128 dimensional images with 3 color channels. We can convert these colored images to gray scale to pass through our deep learning model for segmenting white blood cells. The model output image is a binary image with demarcated white blood cells and the remaining area is red blood cells.

Algorithms and Techniques

One of the best supervised classification method is Support Vector Machines (Generalization of support vector classifier). Support vector classifier is generalization version of max margin classifier.

If the data is not linearly separable and it is a general classification problem, then using SVM will give good results. SVM represent the data in high dimensional with kernel trick. The method will yield good results without overfitting the data set with less parameters and less memory constraint with support vectors. SVM will give state of the art results on general classification problems like classifying a given data point to any one of given binary or multi class labels. But it cannot extract the useful features from the given data like CNN (Convolutional Neural Networks). Even though by using k-fold cross validation with grid or randomized search, we still need to manually set the parameters (Kernel, C, Y values) to yield

good results. But in case of CNN the model will automatically learn and identify key features in the image using convolution maps with stochastic gradient descent weight update algorithm.

SVM will yield global optimum. But with CNN or neural networks, there is a possibility of local minimum and would not get a big picture on complete data. We can use any one of momentum techniques (Ex: Adam) to get over on local minima.

From the given data set of images, we have to demarcate the boundary of WBC from given image. We need to learn the features of image and understand what is mask and how mask is calculating when model is fitted with training data. CNN will automatically learn the different features and update their weights until gives a low error score.

So in conclusion based on the problem set and context of the problem we will go with any one of classification techniques like SVM, Random Forest, and Feed Forward Neural networks or CNN. In our case convolutional neural networks will give accurate results and correctly identify the boundary of white blood cells in images even though the cells are touching with each other for counting purpose.

Convolutional neural networks will yield state of the art results on image processing like object detection, classification and segmentation.

Convolutional neural network is a combination of convolutional map, filters, max pooling layers with activation and dropout functions.

For a given input image having dimensions (1,128,128,3) -> (number_of_samples, img_rows, img_cols, color_channels)

- First we need to define filters to run through entire image for capturing different features of image like edges, shapes etc.
- each filter slide through entire image with activation function like **ReLU** (rectified linear unit) produce a **convolutional map**. The activation function can act as a non-linear decision function to allow the input to pass through the neuron or not. If it passes it will return the value of the input otherwise return 0 in case of ReLU.
- Again we can apply same filters to better capture different features and produce an another convolutional map
- Although the role of the convolutional layer is to detect local conjunctions of features from the previous layer, the role of the pooling layer is to merge semantically similar features into one. Because the relative positions of the features forming a motif can vary somewhat, reliably detecting the motif can be done by coarse-graining the position of each feature. A typical **maximum pooling** (Max Pooling layer) unit computes the maximum of a local patch of units in one feature map (or in a few feature maps). Neighboring pooling units take input from patches that are shifted by more than one row or column, thereby reducing the dimension of the representation and creating an invariance to small shifts and distortions.

- Deep neural nets with a large number of parameters are very powerful machine learning systems. However, overfitting is a serious problem in such networks. Large networks are also slow to use, making it difficult to deal with overfitting by combining the predictions of many different large neural nets at test time. **Dropout** is a technique for addressing this problem. The key idea is to randomly drop units (along with their connections) from the neural network during training. This prevents units from co-adapting too much. During training, dropout samples from an exponential number of different "thinned" networks. At test time, it is easy to approximate the effect of averaging the predictions of all these thinned networks by simply using a single unthinned network that has smaller weights. This significantly reduces overfitting and gives major improvements over other regularization methods. Dropout improves the performance of neural networks on supervised learning tasks in vision, speech recognition, document classification and computational biology, obtaining state-of-the-art results on many benchmark data sets.
- Two or three stages of convolution, non-linearity and pooling with dropout are stacked, followed by more convolutional and fully-connected layers.
- We have to run images through this stack of layers multiple times to capture the useful features like object detection and segmentation. Each run you need to update the weights of the network to reduce the loss to accurately identify or segmentation. The process of updating weights with activation function and computing loss is called back propagation. The main goal is reach global minima (low error) with continuous updating of weights in each run. This is called optimization of the network for reaching low error rate. The rate at which each weight is updated is called learning rate. But there are possibilities you reach local minima instead of global minima. At that time, you need some momentum to cross local minima to reach global minima. For this we will use different type of momentum techniques like Adam etc.

If we apply convolutional and pooling operations sequentially on the given image, the image size will decrease deep down and it will not return correct masked area of WBCs. So in order to keep the image as equal to the input shape. We are going to apply de-convolution on convolutional feature maps to restore the image to original image resolution and correctly identify the demarcated area of white blood cells.

Benchmark

I am using modified U-net model and training strategy to get efficient results.

The original U-net model have achieved 77.5% score on DIC-HeLa data set from ISBI cell tracking challenge. I am taking this is as a benchmark and try to achieve more score with modified U-net architecture.

Methodology

Data Preprocessing

Before data can be used as input for machine learning algorithms, it often must be cleaned, formatted, and restructured — this is typically known as preprocessing. Fortunately, for this dataset, there are no invalid data we must deal with, however, there are 5 large images (corresponding masks) of blood. So we need to re-size these images to 128 * 128 without losing the pixel information from both train and mask data and store it in new folder called NEW_TRAIN_DATA. So that all the images (both train and mask) are in same dimension and pass to our deep learning algorithm for training.

From Figure 3 you can see the new images after preprocessing of large images (more than 128 * 128).

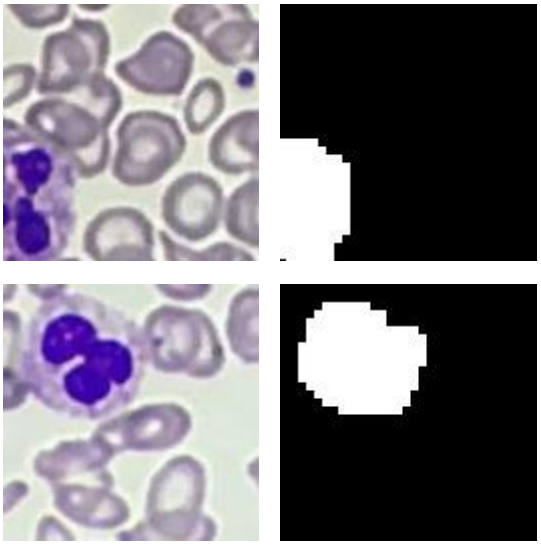


Figure 3. Images after preprocessing of large images in training set with associated white blood cells label.

Implementation

From the given training data set(SigTuple_data/Train_Data) the train images (train-0.jpg, train-1.jpg, train-2.jpg.. etc.) can act as a training set and mask images (train-0-mask.jpg, train-1-mask.jpg, train-2-mask.jpg.. etc.) as dependent to fit to our train images. We can create these two sets as X_train and Y_train and pass these for training to our model. Here we are using tensorflow as a back end for our Keras deep learning model.

When using TensorFlow as backend, Keras CNNs require a 4D array (which we'll also refer to as a 4D tensor) as input, with shape

(nbsamples,rows,columns,channels)

where nb_samples correspond to the total number of images (or samples), and rows, columns, and channels correspond to the number of rows, columns, and channels for each image, respectively.

For this problem who need to apply Instance based segmentation approach also called simultaneous detection and segmentation. If you use normal convolution approach we need more processing and training time for object detection and a separate method to segment each instance but if you use modified U-net model, you can run your input end to end at a time and there is no separate processing for identification and segmentation of instances.

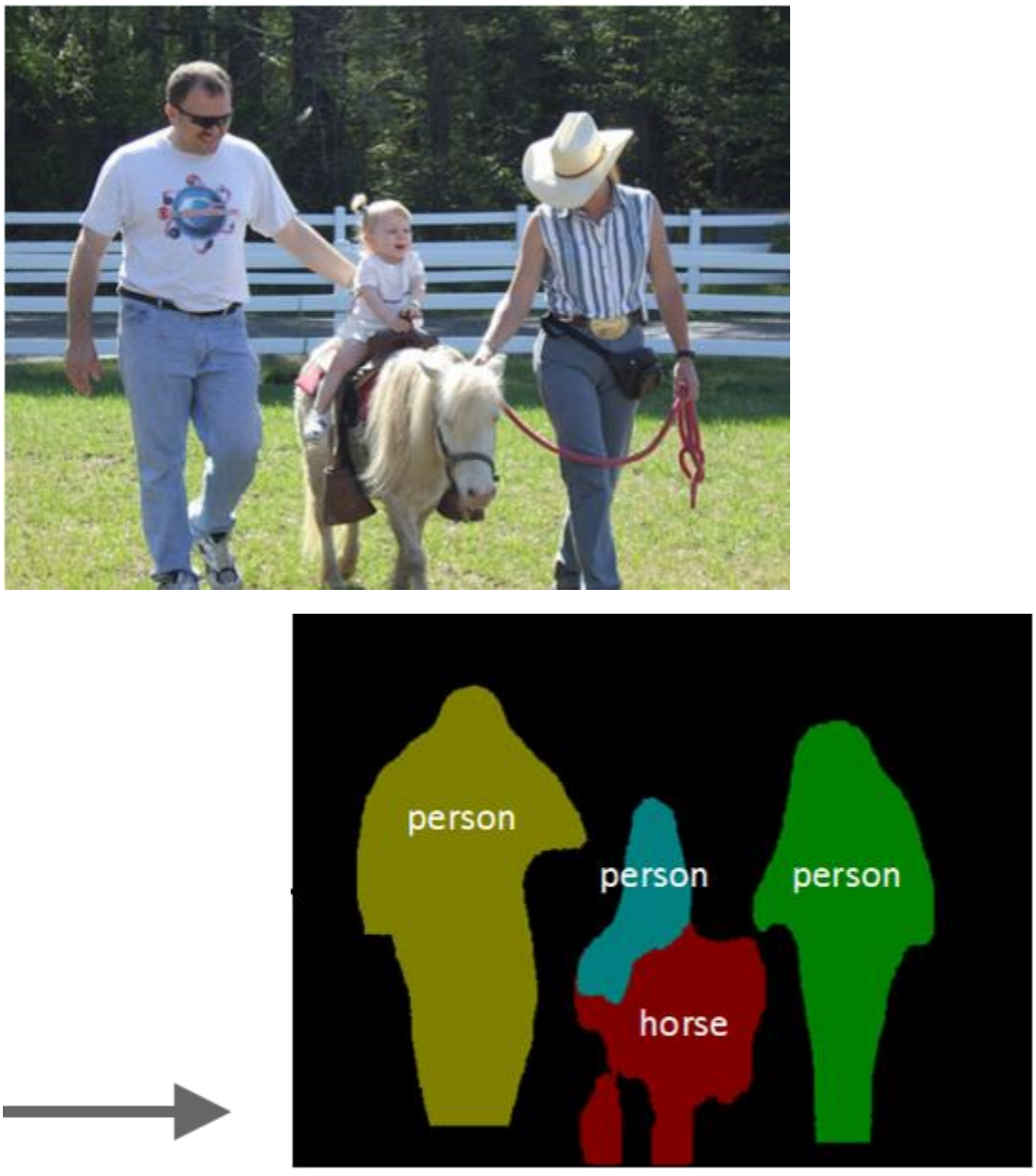


Figure 4. Instance based segmentation of persons and horse in image

From Figure 4, the model is correctly segmenting two classes (person, horse) correctly using the instance based segmentation with accurate results. If we apply the same model to our problem, it will give accurate results and correctly identify the boundary of white blood cells in images even though the cells are touching with each other for counting purposes.

We are using the same approach using modified U-net model (Figure 6) architecture for getting accurate results.

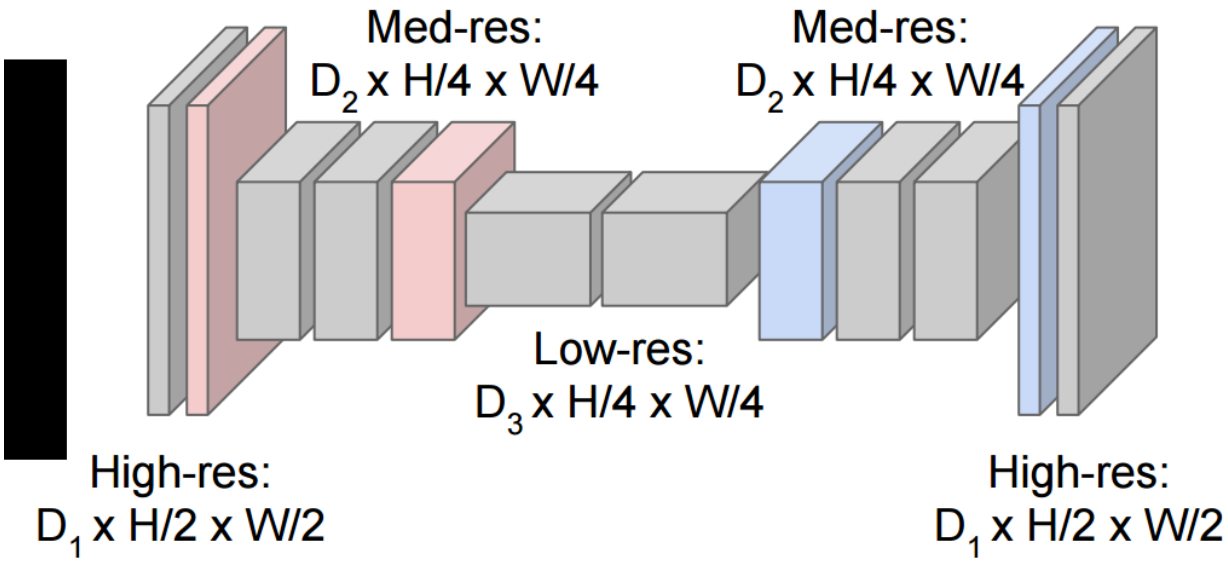


Figure 5. While this does not represent the exact architecture of our model, it demonstrates the principles of a down-sampling up-sampling convolutional network.

The contracting path follows the typical architecture of a convolutional network. It consists of the repeated application of two 3x3 convolutions (unpadded convolutions), each followed by a rectified linear unit (ReLU) and a 2x2 max pooling operation with stride 2 for down sampling. At each down sampling step, we double the number of feature channels. See Figure 5.

Every step in the expansive path consists of an up sampling of the feature map followed by a 2x2 convolution (\up-sampling") that halves the number of feature channels, a concatenation with the correspondingly cropped feature map from the contracting path, and two 3x3 convolutions, each followed by a ReLU. The cropping is necessary due to the loss of border pixels in every convolution.

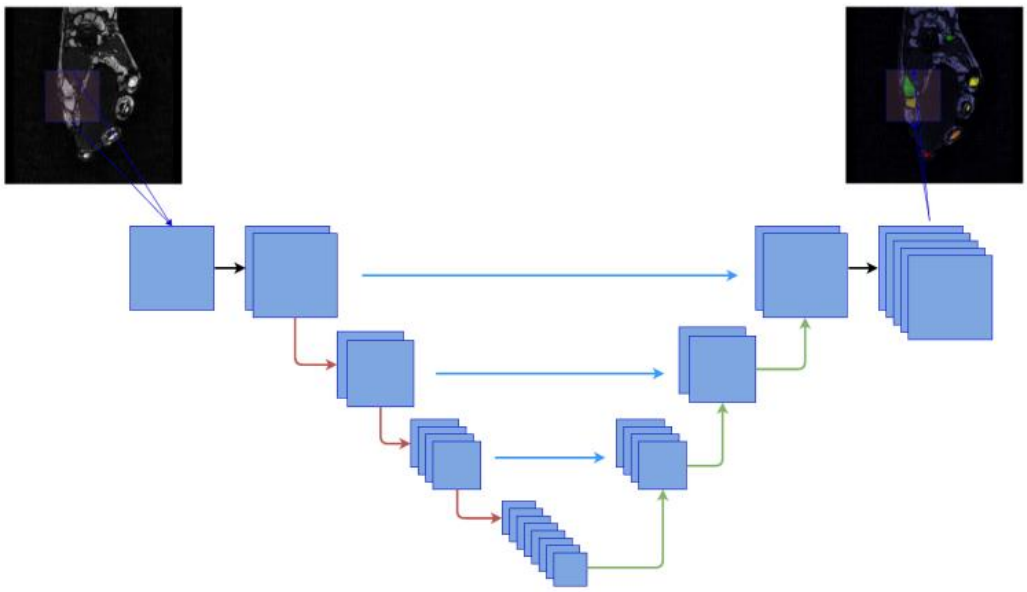


Figure 6. This figure does not represent the exact architecture of our model. However, it does illustrate the principles of a U-Net model for segmentation problem, in which the outputs of early down-sampling layers are concatenated to those of later up-sampling layers

I have trained this model with 50 epochs and have batch size of 3 and learning rate 1e-04(0.0001).

During training after each conv and pooling layer I have applied dropout layer to reduce overfitting due to size of the data.

Most of the time dropout is applied to fully connected layers compared to convolutional layers. since the convolutional layers don't have a lot of parameters, overfitting is not a problem and therefore dropout would not have much effect. However, the additional gain in performance obtained by adding dropout in the convolutional layers (3.02% to 2.55%) is worth noting. Dropout in the lower layers still helps because it provides noisy inputs for the higher fully connected layers which prevents them from overfitting.

Dropout is a float value between 0 and 1. Fraction of the input units to drop. I have used 0.25 dropout value for our model to train the model for efficient results

Keras Model Architecture:

```
Layer (type)                 Output Shape              Param #     Conne
cted to
==================================================================================
==========================
inputs (InputLayer)          (None, 128, 128, 1)   0
__________________________________________________________________________________
_________________________
```

```
conv1_1 (Conv2D)                (None, 128, 128, 32)  320         input
s[0][0]
_________________________________________________________________
_____________________________
conv1_2 (Conv2D)                (None, 128, 128, 32)  9248        conv1
_1[0][0]
_________________________________________________________________
_____________________________
pool_1 (MaxPooling2D)           (None, 64, 64, 32)    0           conv1
_2[0][0]
_________________________________________________________________
_____________________________
dropout_1 (Dropout)             (None, 64, 64, 32)    0           pool_
1[0][0]
_________________________________________________________________
_____________________________
conv2_1 (Conv2D)                (None, 64, 64, 64)    18496       dropo
ut_1[0][0]
_________________________________________________________________
_____________________________
conv2_2 (Conv2D)                (None, 64, 64, 64)    36928       conv2
_1[0][0]
_________________________________________________________________
_____________________________
pool_2 (MaxPooling2D)           (None, 32, 32, 64)    0           conv2
_2[0][0]
_________________________________________________________________
_____________________________
dropout_2 (Dropout)             (None, 32, 32, 64)    0           pool_
2[0][0]
_________________________________________________________________
_____________________________
conv3_1 (Conv2D)                (None, 32, 32, 128)   73856       dropo
ut_2[0][0]
_________________________________________________________________
_____________________________
conv3_2 (Conv2D)                (None, 32, 32, 128)   147584      conv3
_1[0][0]
_________________________________________________________________
_____________________________
pool_3 (MaxPooling2D)           (None, 16, 16, 128)   0           conv3
_2[0][0]
_________________________________________________________________
_____________________________
dropout_3 (Dropout)             (None, 16, 16, 128)   0           pool_
3[0][0]
_________________________________________________________________
_____________________________
conv4_1 (Conv2D)                (None, 16, 16, 256)   295168      dropo
ut_3[0][0]
_________________________________________________________________
_____________________________
conv4_2 (Conv2D)                (None, 16, 16, 256)   590080      conv4
_1[0][0]
```

```
__________________________________________________________________________________________________
pool_4 (MaxPooling2D)           (None, 8, 8, 256)    0           conv4_2[0][0]
__________________________________________________________________________________________________
dropout_4 (Dropout)             (None, 8, 8, 256)    0           pool_4[0][0]
__________________________________________________________________________________________________
conv5_1 (Conv2D)                (None, 8, 8, 512)    1180160     dropout_4[0][0]
__________________________________________________________________________________________________
conv5_2 (Conv2D)                (None, 8, 8, 512)    2359808     conv5_1[0][0]
__________________________________________________________________________________________________
upsample_1 (UpSampling2D)       (None, 16, 16, 512)  0           conv5_2[0][0]
__________________________________________________________________________________________________
concat_1 (Concatenate)          (None, 16, 16, 768)  0           upsample_1[0][0]
                                                                 conv4_2[0][0]
__________________________________________________________________________________________________
conv6_1 (Conv2D)                (None, 16, 16, 256)  1769728     concat_1[0][0]
__________________________________________________________________________________________________
conv6_2 (Conv2D)                (None, 16, 16, 256)  590080      conv6_1[0][0]
__________________________________________________________________________________________________
dropout_6 (Dropout)             (None, 16, 16, 256)  0           conv6_2[0][0]
__________________________________________________________________________________________________
upsample_2 (UpSampling2D)       (None, 32, 32, 256)  0           dropout_6[0][0]
__________________________________________________________________________________________________
concat_2 (Concatenate)          (None, 32, 32, 384)  0           upsample_2[0][0]
                                                                 conv3_2[0][0]
__________________________________________________________________________________________________
conv7_1 (Conv2D)                (None, 32, 32, 128)  442496      concat_2[0][0]
__________________________________________________________________________________________________
```

```
conv7_2 (Conv2D)                (None, 32, 32, 128)   147584      conv7
_1[0][0]
__________________________________________________________________________
_____
dropout_7 (Dropout)             (None, 32, 32, 128)   0           conv7
_2[0][0]
__________________________________________________________________________
_____
upsample_3 (UpSampling2D)       (None, 64, 64, 128)   0           dropo
ut_7[0][0]
__________________________________________________________________________
_____
concat_3 (Concatenate)          (None, 64, 64, 192)   0           upsam
ple_3[0][0]
                                                                    conv2
_2[0][0]
__________________________________________________________________________
_____
conv8_1 (Conv2D)                (None, 64, 64, 64)    110656      conca
t_3[0][0]
__________________________________________________________________________
_____
conv8_2 (Conv2D)                (None, 64, 64, 64)    36928       conv8
_1[0][0]
__________________________________________________________________________
_____
dropout_8 (Dropout)             (None, 64, 64, 64)    0           conv8
_2[0][0]
__________________________________________________________________________
_____
upsample_4 (UpSampling2D)       (None, 128, 128, 64)  0           dropo
ut_8[0][0]
__________________________________________________________________________
_____
concat_4 (Concatenate)          (None, 128, 128, 96)  0           upsam
ple_4[0][0]
                                                                    conv1
_2[0][0]
__________________________________________________________________________
_____
conv9_1 (Conv2D)                (None, 128, 128, 32)  27680       conca
t_4[0][0]
__________________________________________________________________________
_____
conv9_2 (Conv2D)                (None, 128, 128, 32)  9248        conv9
_1[0][0]
__________________________________________________________________________
_____
dropout_9 (Dropout)             (None, 128, 128, 32)  0           conv9
_2[0][0]
__________________________________________________________________________
_____
outputs (Conv2D)                (None, 128, 128, 1)   33          dropo
ut_9[0][0]
```

```
==================================================================
============================
Total params: 7,846,081
Trainable params: 7,846,081
Non-trainable params: 0
```

Refinement

As mentioned in the Benchmark section, the U-net architecture trained with Caffe achieved a good score, around 77%.

To get the initial result, this architecture was ported to Keras with tensor flow as back end; the result has score around 67%. This was improved upon by using the following techniques:

- Dynamic learning rate: whenever the loss function stopped decreasing, a learning rate drop was added. I have used Keras call back function **ReduceLROnPlateau** for reducing learning rate. This callback monitors loss as a quantity and if no improvement is seen for 10 epochs, the learning rate is reduced by a factor by which the learning rate will be reduced. new_lr = lr * factor
- Adding dropout to a layer: dropout randomly drops weights in the layer it's applied to during training and scales the weights so that the network keeps working during inference.

The final Keras model was derived by training in an iterative fashion, adjusting the parameters (e.g. learning rate) has score of 94%.

Results

Model Evaluation and Validation

We have used the given training set (both images and masks X_train, Y_train) to fit to our model and used dice coefficient as a metric to identify the accuracy of the model.

Finally, we applied trained model using dice score as metric to predict masks for X_train and compare those masks with Y_train. I mean with y_pred with y_true.

The model achieved state of the art results with dice score of 94% and identified correct pixels of white blood cells.

The model achieved better score compare to bench mark model I have chosen before.

Justification

The model correctly demarcates the boundary of white blood cells irrespective of its size and count.

As discussed above by adding dynamic learning rate and dropout the model improves a lot and achieved great results. I have used 0.25 as dropout rate, at any time ¾ neurons are active and contribute to learning the features. The higher the dropout, the less I would expect it to converge. So I pick this rate as good value and achieved good results.

All the images in test set are accurately demarcated with this model above 94%. And this dice score is far better compared to the bench mark model.

Conclusion

Free-Form Visualization

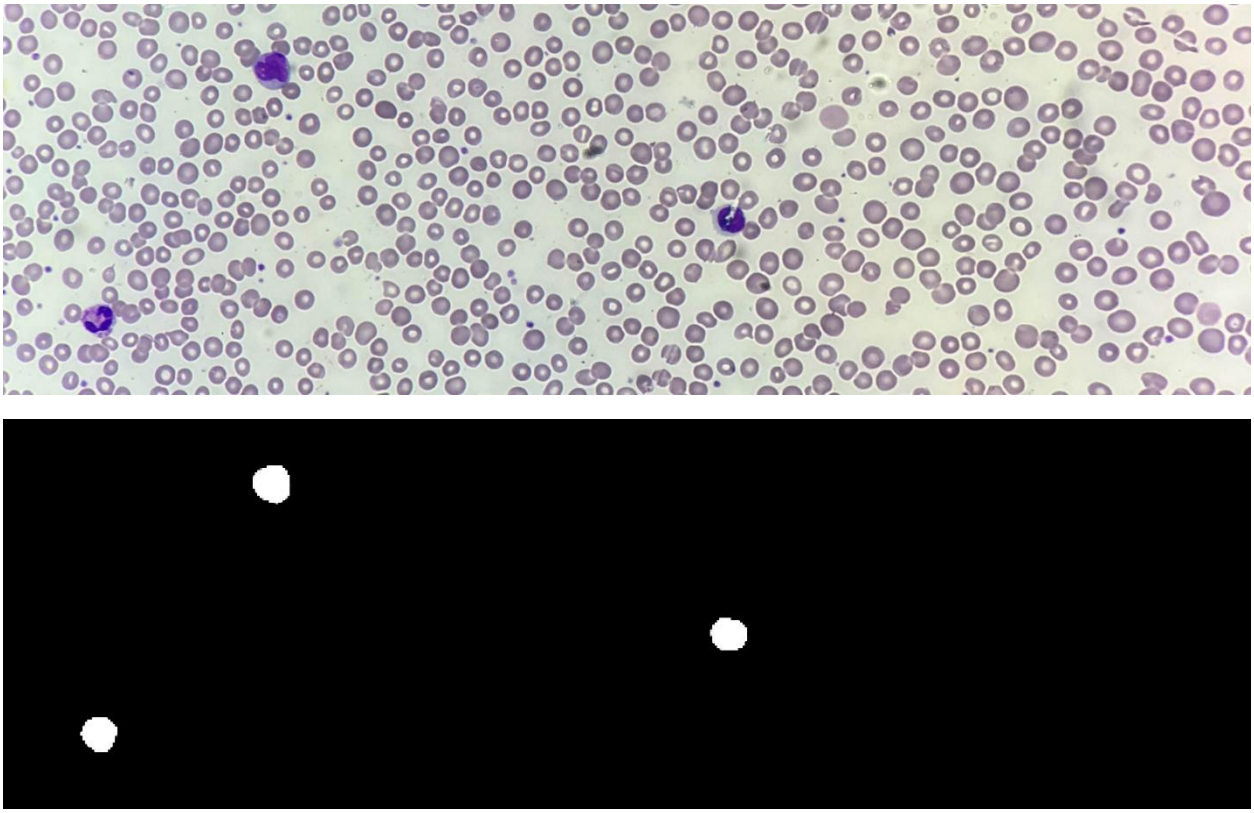


Figure 7. Test image with labelled white blood cells.

From Figure 7, the first image showing the test image with white (blue area) and red (white area) cells. The second image is predicted mask with white blood cells (white area).

Reflection

The process used for this project can be summarized using the following steps:

1. An initial problem and relevant, public datasets were found
2. The data was downloaded and preprocessed
3. A benchmark was created for the classifier

4. The classifier was trained using the data
5. The classified predicted masks for test set
6. Validated the classifier with y_true and y_pred

I found steps 4 is very interesting. The original model(U-net) is not returning best score compared to bench mark result. So I tweaked a bit by adding dynamic learning rate and dropout and resultant model given good dice score results.

I am so glad I found a good model(U-net) to solve this problem otherwise it would take a lot of time to come up with best model and needed a multiple gpu powered machines.

Improvement

From the validation results and predicted masks, the model outperforms the bench mark model and yield good results.

On aws gpu machine each epoch of training time takes less than 5 sec and completed training in less than 5 min.

If we use multiple gpus and develop this model entirely on tensorflow back end, we will serve thousands of requests at a time and serve practical applications like real time white blood cells segmentation.

I would like to port this model on android and use tensor flow as back end for production.