Journal of Machine Learning Research 1 (2020) 1-6 Submitted 5/20; Published 5/20

Deep Learning Classification of Blood Cell Images

Michael Chen MC8000@NYU.EDU
MS in Biomedical Informatics
New York University
New York, NY 10003, USA

Allen Zhu ASZ241@NYU.EDU
MS in Biomedical Informatics
New York University
New York, NY 10003, USA

Editor: Dr. Deniz and Dr. Razavian

Abstract

This paper explores the utilization of various deep learning models for differentiating between four classes of white blood cell images: eosinophils, lymphocyte, monocytes, and neutrophils. We explored autoencoders and convolutional neural network (CNN) architectures including residual network (ResNet), asymmetric convolutional network (ACNet), and Inception. We examined the effects that changes to the model have on model performance. We discuss our speculations of why some models might have performed better than others. A high accuracy model would benefit the field of pathology by conserving resources and reducing human error.

Keywords: Deep Learning, Convolutional Neural Networks, ResNet, Inception, Autoencoders, Pathology, White Blood Cells

1. Introduction

Deep learning took hold around 2006 and has since surpassed its predecessor, machine learning, in a variety of domains such as image processing, computer vision, speech recognition, natural language translation, art, medical imaging, medical information processing, robotics, bioinformatics, natural language processing, cybersecurity, and more (Alom et al., 2018). Hematology is another area in which deep learning has great potential. Blood cells including eosinophils, lymphocytes, monocytes, and neutrophils are all significant in indicating infection by a foreign agent (Doan, 1954). However, manual differentiation by microscope is neither time- nor energy-efficient, which drives motivation to develop automated classification with methods such as deep learning (Sabino et al., 2004).

Previously, researchers achieved an accuracy of 72.3% when classifying blood cells using a decision tree, a machine learning method (Tantikitti et al., 2015). In a more recent 2018 study, researchers achieved an accuracy of 78% for the same classification task using a CNN-based model (Tiwari et al., 2018). We chose to try autoencoder, ResNet, and Inception variants, all architectures of which have shown promising results for image classification. Autoencoders are characterized by an encoder and a decoder. ResNet is characterized by skip connections. ACNet is characterized by asymmetrical convolution blocks (ACBs). Inception is characterized by wider networks rather than deeper networks. Given the vast

©2020 Michael Chen and Allen Zhu.

success of these architectures in image recognition, we hypothesize that we can achieve equal or better results with our deep learning models.

2. Materials and Methods

2.1 Data

The MIT-licensed Blood Cell Count and Detection (BCCD) Dataset was provided as part of a Kaggle challenge. The dataset consists of 12515 320x240px .jpeg images belonging to one of four classes: eosinophil, lymphocyte, monocyte, and neutrophil. Images in the train and test sets were augmented and evenly distributed among the four classes. There was a test_simple set with unaugmented images, which also had more variation among the four classes. View Table 1 for a more detailed breakdown of the data.

Table 1: Detailed Breakdown of Data

	train	test	test_simple	**Total**
Eosinophil	2497	623	13	3133
Lymphocyte	2483	620	6	3109
Monocyte	2478	620	4	3102
Neutrophil	2499	624	48	3171
Total	9957	2487	71	**12515**

2.2 Methods

2.2.1 Autoencoder

An autoencoder is an unsupervised (or rather self-supervised) neural network that learns important features from unlabeled data (Ng et al., 2011). The autoencoder takes an input, creates a compressed representation (code) with an encoder, then attempts to reproduce the original input with a decoder. Traditionally, we utilize a regression loss function to minimize the difference between the reproduced input and the original input. For our purposes, we also implemented a multi-class classification loss function to also acquire loss between the predicted class of the code and the actual class of the input. Together, in our case, MSE loss and cross entropy loss are combined and minimized as one coherent loss function. For our optimizer, we used Adaptive Moment Estimation Algorithm (Adam). Adam has been hailed as an easy-to-implement optimizer that is computationally efficient and works well with non-stationary objects and noisy gradients (Kingma and Ba, 2014). Since the test set was already separated from the train set, the train set was split such that the validation set had the same number of images as the test set, representing a 60:20:20 train:validation:test split. Each image was preprocessed using their Z-score (input minus the mean, divided by the standard deviation). Training and validation was done in minibatches of four images over twenty epochs. Four variations of autoencoders were trained, validated, and tested. Two of these variations contained a hidden layer within the encoder and decoder. The other two variations removed this hidden layer. For precise details regarding the architecture, modifications, and performance, please refer to Appendix A..

2.2.2 ResNet, Custom CNN, and ACNet

For models in this section, we used cross entropy loss as our loss function and Adam as our optimizer. Again, all models were trained for twenty epochs. The built-in ResNet18 was imported as-is and only the final layer was changed to reflect the four classes in our problem. ResNet18 was trained, validated, and evaluated with minibatches of four images only once, primarily to ensure functioning support code. A custom CNN with ResBlocks (skip connections) were then trained, validated, and evaluated with 31 variations in a grid-search fashion. Modifications ranged from simple hyperparameter tuning (input/output size, minibatch size, etc.) to more substantial structural changes (making the network shallower/deeper, adding convolution layers to ResBlocks, etc.). Among these structural changes, we experimented with the use of asymmetrical convolution blocks (ACBs) in place of square kernels. ACBs perform dxd, 1xd, and dx1 convolutions in parallel, then combine them to strengthen the "skeleton" (middle horizontal and vertical regions) of the kernel (Ding et al., 2019). For precise details regarding the architecture, modifications, and performance, please refer to Appendix A.

2.2.3 Inception

Models in this section were inspired by Google's Inception neural network. These models prioritized bottlenecking, factorization, and width rather than the traditional depth that current state of the art models strive for. Like previous models, the ones mentioned here utilizes cross entropy loss as its loss metric, but switches to adadelta for its optimizer function. Adadelta is a learning rate that dynamically adapts over time and thus requires minimal manual tuning (Zeiler, 2012). One of the top performing models in the Kaggle challenge was a 24-layer deep inception model, reaching 99% testing accuracy, and serves as the benchmark for the rest of the model mentioned here. Image preparation steps were done by using the Kaggle challenge's host, Paul Money's code, and were thus done slightly differently compared to the encoder and residual networks. Images were first resized from a size of [320, 240, 3] to [80, 60, 3] before the pixel data was converted into a numpy array. The array was then normalized by dividing everything by 255.0. The y-labels were one-hot encoded into categories. Models performances were evaluated based on their precision and recall metrics. Several additional pre-processing steps were also included prior to training the image, where rotations of up to 10°, slight horizontal/vertical shifts, and flips around the horizontal axis would be randomly introduced. As there was a testing set already provided, a validation set of data was generated using 20% of the training set. Several models were first built featuring only bottlenecking and factorization features. A 1x1 conv2d layer was introduced before a varying amount of 3x3 conv2d layers. The equivalent of stacking two 3x3 conv2d layers on top of each other gives the equivalent kernel size of 5x5 and stacking three 3x3 conv2d layers on top of each other gives and equivalent kernel size of 7x7. Parallelization was introduced in later models where different kernel sizes (1x1, 3x3, 5x5, and 7x7) were run in parallel before being concatenated and subject to a maxpooling operation. Note that all the larger kernel sizes (5x5 and 7x7) were still subject to factorization. Several model iterations took factorization further and turned a 3x3 conv2d layers into a 1x3 layer followed by a 3x1 layer, with the final 3x3 factorization done in parallel before concatenation (refer to appendix A, figure 4 for model visualization). During the training process models were

initially evaluated for 15 epochs. Validation accuracy was monitored, where the weights of the best performing model was saved each time the model achieved a high validation accuracy. For several of the later models that saw an increase in complexity, models were trained for 50 epochs as 15 epochs did not seem sufficient enough to fully train the model.

3. Results

3.0.1 Evaluation Discrepancy

One major finding was a discrepancy that would arise between the test and test_simple sets. Recall that the differences between these sets are 1) augmentation vs. no augmentation and 2) balanced vs. imbalanced data. ResNet18 yielded a decent accuracy of 85.7% over the test set, but a mere 25.4% accuracy over the test_simple set. The test set would consistently have a higher accuracy than the test_simple set, with the greatest difference in accuracies being 64.1% in MyCNN_v16.

3.0.2 Models That Worked

Of the 31 custom CNN variations, versions v4, v9, and v11 stood out the most. These layers featured two, three, and four convolution layers, respectively, to each of their ResBlocks. In summary, while accuracy over the test set increased gradually, accuracy over the test_simple set increased, then decreased again. Overall, v9 performed best out of these 31 models. View Table 2 for v4, v9, and v11 accuracies.

Of the dozen inception-based models trained, most of them reached at least 80% accuracy on the validation set over the course of 15 epochs. Models that failed to reach this benchmark includes one that featured a 1x1 conv2d layer followed by a single 3x3 conv2d layer, and another that feature 5 parallel 'towers' running in parallel (4 towers containing bottle neck + a variety of factorized kernel sizes, the last tower containing a max pooling before a 1x1 conv2d layer) before concatenating all the outputs together. Several additional "tower-based' models were evaluated. The performance of the two best models featuring parallelization are listed in Table 4 below. ParaA is a model that features a two 3x3 conv2D 'stem' that feeds into 3 parallel towers. One tower features an effective kernel size of 5x5, the second tower features an effective kernel size of 3x3, and the third tower features a max-pooling into a 1x1 conv2D layer. The outputs of the three towers are then concatenated and flattened into a fully connected layer before being put through a softmax layer. ParaB is a model that is the same as ParaA but feeds the inital output into another three towers that is the same as as in ParaA.

3.0.3 Models That Require More Work

For the autoencoder, both train and validation loss/accuracy stabilized by the fifth epoch, but neither improved beyond that. Evaluation over the test and test_simple sets revealed poor performance; 25.1% and 18.3%, respectively. The confusion matrix shown in Figure 1 shows that every input is being predicted as a neutrophil. For the ACNet, the test accuracy was 85.9% while the test_simple accuracy was 66.2

Table 2: Accuracies of MyCNN Versions v4, v9, and v11

	v4	v9	v11
test	85.4%	86.5%	88.2%
test_simple	74.6%	84.5%	78.9%

Table 3: Accuracies of Models with Bottleneck/Factorized Layers Only

	3x3 Effective Size	5x5 Effective Size	7x7 Effective Size
test	73%	87%	88%
test_simple	63.4%	78.9%	85.9%

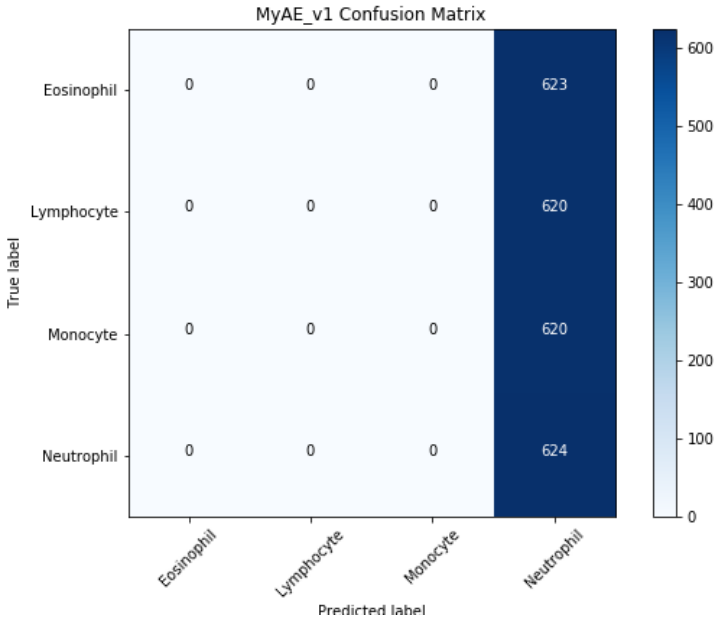

Figure 1: Autoencoder Confusion Matrix

4. Discussion

Overall, we can accept our hypothesis that some of our models performed better than those from previous years. Loss graphs followed an expected trend and multiple models confirmed the accuracy we were seeing in both test and test_simple sets.

The discrepancy between test and test_simple accuracies comes down to either the difference in augmentation or the difference in balance. We can rule out the difference in balance because based on the test accuracy, we know that our model is not being a lazy predictor (as our autoencoder was) and that the model is not predicting correctly by chance. We can thus conclude that balance is not as important in the test set as it is in the train set.

Furthermore, there was a consistent discrepancy between the validation accuracy compared to the testing accuracy in general. This raises questions on potential issues of overfit-

ting. The subsetted validation set may be too similar to the training set and insufficiently different from either test sets to generalize properly.

The ACBs yielded decent results, but definitely not at a level that the original authors witnessed. More work in this area would be valuable. Perhaps ACBs are more effective for deeper networks. Perhaps the configuration of ACBs should be applied to different layers.

An additional finding that arose when examining AUROC plots of the custom CNN was that lymphocytes were consistently the easiest blood cell class to predict. We can speculate that this is attributed to the denser nucleus of lymphocytes compared to the other blood cell classes, which can easily be identified, even by the untrained eye.

The autoencoder, on the other hand, exhibited very poor performance. In fact, the results show that the autoencoder is not able to learn at all. Future works include the addition of several hidden layers in the encoder/decoder, and trying a convolutional autoencoder.

Acknowledgments

We would like to acknowledge support for this project from the professors and TAs of the Deep Learning Course (BMIN-GA 3007).

Appendix A.

In this appendix, we provide the ResNet18 performance as well as the model architecture, changelog, and performance for the custom CNN and autoencoder.

Figure 2: ResNet18 Performance

Filename	Details/Changes
ResNet18.ipynb	Directly imported and trained successfully. Used to establish support code. Accuracy: 85.7%. Simple Accuracy: 25.4%.

Figure 3: MyCNN Architecture

```
MyCNN(
  (conv1): Sequential(
    (0): Conv2d(3, 16, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (res1a): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (res1b): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv2): Sequential(
    (0): Conv2d(16, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (res2a): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (res2b): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv3): Sequential(
    (0): Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (res3a): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (res3b): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (fc1): Linear(in_features=1280, out_features=128, bias=True)
  (fc2): Linear(in_features=128, out_features=4, bias=True)
)
```

Figure 4: MyCNN Changelog and Performance

Filename	Details/Changes
MyCNN.ipynb	Original. Model from HW2. Accuracy: 86.2%. Simple Accuracy: 76.1%. Added third 3x3 convolution in each residual block. Accuracy: 81.0%. Simple Accuracy: 77.5%. Reverted to original residual blocks.
MyCNN_v1.ipynb	Changed layers from 16>32>64>1280 to 16>64>256>5210. Accuracy: 73.4%. Simple Accuracy: 30.1%.
MyCNN_v2.ipynb	Changed layers from 16>32>64>1280 to 8>16>32>640. Accuracy: 81.7%. Simple Accuracy: 36.6%.
MyCNN_v3.ipynb	Added layer to go from 16>32>64>1280 to 16>32>64>128>128. Accuracy: 86.6%. Simple Accuracy: 70.4%.
MyCNN_v4.ipynb	From v3, changed FC layers from 128>128, 128>4 to 128>512, 512>4. Accuracy: 85.4%. Simple Accuracy: 74.7%. Changed batch size to 16. Accuracy: 84.7%. Simple Accuracy: 67.6%. Changed batch size to 1. Accuracy: 59.1%. Simple Accuracy: 32.4%. Reverted to original batch size.
MyCNN_v5.ipynb	From v4, changed FC layers from 128>512, 512>4 to 128>256, 256>4. Accuracy: 80.7%. Simple Accuracy: 31.0%.
MyCNN_v6.ipynb	From v4, changed FC layers from 128>512, 512>4 to 128>64, 64>4. Accuracy: 80.2%. Simple Accuracy: 78.9%.
MyCNN_v7.ipynb	From v3, changed FC layers from 128>128, 128>4 to 128>1024, 1024>4. Accuracy: 82.1%. Simple Accuracy: 31.0%.
MyCNN_v8.ipynb	From v3, changed layers from 16>32>64>128 to 8>16>64>512. From v3, changed FC layers from 128>128, 128>4 to 512>128, 128>4. Accuracy: 80.5%. Simple Accuracy: 60.6%.
MyCNN_v9.ipynb	From v4, added a third 3x3 convolution in each residual block. This is typically reserved for deeper networks, but just curious. Accuracy: 86.5%. Simple Accuracy: 84.5%.
MyCNN_v10.ipynb	From v4, changed self.res1 (a, b) kernel size from 3 to (1x3) & (3x1). Accuracy: 82.5%. Simple Accuracy: 64.8%.
MyCNN_v11.ipynb	From v9, added a fourth 3x3 convolution in each residual block. Accuracy: 88.2%. Simple Accuracy: 78.9%.
MyCNN_v12.ipynb	From v10, corrected the ACB (I was missing a 3x3 between 1x3 and 3x1). Accuracy: 85.9%. Simple Accuracy: 66.2%.
MyCNN_v13.ipynb	From v12, corrected the ACB (added in parallel instead of in series). Accuracy: 80.7%. Simple Accuracy: 50.7%.
MyCNN_v14.ipynb	From v4, added batch normalization between each conv and relu. ^ In conv layers only, not the residual blocks. Accuracy: 84.6%. Simple Accuracy: 75.8%.
MyCNN_v15.ipynb	From v4, added batch normalization between each conv and relu. ^ In all layers (conv and residual blocks). Accuracy: 81.5%. Simple Accuracy: 73.2%.
MyCNN_v16.ipynb	From v4, added batch normalization between each conv and relu. ^ In residual blocks only, not the conv layers. Accuracy: 82.4%. Simple Accuracy: 18.3%.
MyCNN_v17.ipynb	From v4, introduced dropout before each FC layer. 50%: Accuracy: 70.6%. Simple Accuracy: 50.7%. 20%: Accuracy: 84.5%. Simple Accuracy: 36.6%. Changed batch size to 16. Accuracy: 75.0%. Simple Accuracy: 62.0%. Changed batch size to 1. Accuracy: 73.2%. Simple Accuracy: 67.6%.
MyCNN_v18.ipynb	From v9, introduced dropout before each FC layer. 50%: Accuracy: 86.5%. Simple Accuracy: 84.5%. 20%: Accuracy: 86.5%. Simple Accuracy: 84.5%. Changed batch size to 16. Accuracy: 85.4%. Simple Accuracy: 80.3%. Changed batch size to 1. Accuracy: 82.5%. Simple Accuracy: 69.0%.
MyCNN_v19.ipynb	From v13, introduced dropout before each FC layer. 50%: Accuracy: 44%. Simple Accuracy: 28%. 20%: Accuracy: 83.2%. Simple Accuracy: 62.0%.
MyCNN_v20.ipynb	Removed layer from original. Made channels 3>16>64. FC1 input is not 19200. FC1 output / FC2 input is now 1024 instead of 128. Accuracy: 67.7%. Simple Accuracy: 40.9%.

Figure 5: Autoencoder Architecture

```
MyAE(
  (encoder): Sequential(
    (0): Linear(in_features=230400, out_features=128, bias=True)
    (1): ReLU()
    (2): Linear(in_features=128, out_features=1024, bias=True)
    (3): ReLU()
  )
  (decoder): Sequential(
    (0): Linear(in_features=1024, out_features=128, bias=True)
    (1): ReLU()
    (2): Linear(in_features=128, out_features=230400, bias=True)
    (3): ReLU()
  )
  (dropout): Dropout(p=0.2, inplace=False)
  (fc): Linear(in_features=1024, out_features=4, bias=True)
)
```

Figure 6: Autoencoder Changelog and Performance

Filename	Details/Changes
Autoencoder.ipynb	Original. Model from lab7. Failed training. Evaluation metrics (starting from evaluate_model) don't work. Fixed in v1.
Autoencoder_v1.ipynb	Switched hidden_dim and code_dim (now 1024 and 128, respectively) Accuracy: 25.1%. Simple Accuracy: 18.3%. As you can see from the confusion matrix, the model is a lazy predictor. Everything is predicted as "Neutrophil".
Autoencoder_v2.ipynb	From v1, removed hidden layer. Set code_dim to 1024. From v1, increased batch size to 16. From v1, increased learning rate to 0.001.
Autoencoder_v3.ipynb	From v1, removed hidden layer. Set code_dim to 128. From v1, increased batch size to 32. From v1, increased learning rate to 0.001.

Appendix B.

In this appendix, we describe the delegation of responsibilities throughout this project.

Michael Chen:

- Proposal Writeup: 40%
- Proposal Presentation: 50%
- Models: 50%
- Final Presentation: 50%
- Final Writeup: 60%

Allen Zhu:

- Proposal Writeup: 60%
- Proposal Presentation: 50%
- Models: 50%
- Final Presentation: 50%
- Final Writeup: 40%

References

Md. Zahangir Alom, Tarek M. Taha, Christopher Yakopcic, Stefan Westberg, Mahmudul Hasan, Brian C. Van Esesn, Abdul A. S. Awwal, and Vijayan K. Asari. The history began from alexnet: A comprehensive survey on deep learning approaches. *CoRR*, abs/1803.01164, 2018. URL http://arxiv.org/abs/1803.01164.

Xiaohan Ding, Yuchen Guo, Guiguang Ding, and Jungong Han. Acnet: Strengthening the kernel skeletons for powerful cnn via asymmetric convolution blocks. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1911–1920, 2019.

Charles A Doan. The white blood cells in health and disease. *Bulletin of the New York Academy of Medicine*, 30(6):415, 1954.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Andrew Ng et al. Sparse autoencoder. *CS294A Lecture notes*, 72(2011):1–19, 2011.

Daniela Mayumi Ushizima Sabino, Luciano da Fontoura Costa, Edgar Gil Rizzatti, and Marco Antonio Zago. A texture approach to leukocyte recognition. *Real-Time Imaging*, 10(4):205–216, 2004.

Sarach Tantikitti, Sompong Tumswadi, and Wichian Premchaiswadi. Image processing for detection of dengue virus based on wbc classification and decision tree. In *2015 13th International Conference on ICT and Knowledge Engineering (ICT & Knowledge Engineering 2015)*, pages 84–89. IEEE, 2015.

Prayag Tiwari, Jia Qian, Qiuchi Li, Benyou Wang, Deepak Gupta, Ashish Khanna, Joel JPC Rodrigues, and Victor Hugo C de Albuquerque. Detection of subtype blood cells using deep learning. *Cognitive Systems Research*, 52:1036–1044, 2018.