Applications of Linear Algebra to Malaria Diagnosis:

The Convolutional Neural Network

Pages: 18

INTRODUCTION AND RATIONALE

I am currently taking a course in Linear Algebra at the Georgia Institute of Technology. When I started this project, I knew I wanted to be able to apply that experience to an independent investigation in the real world. Additionally, my first experience with research involved the application of computer science to human health problems. These two experiences together provided the perfect aim for my project: an exploration of the mathematical underpinnings of machine learning in order to implement from scratch an image classification network to help diagnose malaria. Machine learning and image classification remain active fields of research today as they have far-reaching implications for things like autonomous vehicles and social media content moderation. Additionally, the entire field of Bioinformatics derives from the application of new computational technologies to biological processes. This project is no different.

MODELING THE PROBLEM

Before I could begin dedicated research, I needed to find a dataset and then represent that data to test the final model. I used the website Kaggle[1] which has a number of pre-sorted datasets specifically for use in machine learning experimentation. The data I found, "Malaria parasite in blood smears" uploaded by Parikshit Sanyal, consists of 200 images divided based on the presence of a *Plasmodium vivax* trophozoite, the microorganism which causes Malaria. By applying a machine learning solution to identify the presence of the trophozoite, we can greatly simplify the work of pathologists who would otherwise need to manually examine blood samples to definitively diagnose malaria.

---

[1] Sanyal, Parikshit. "Malaria Parasite in Blood Smears." Kaggle, 1 Nov. 2021, www.kaggle.com/cmacus/malaria-parasite-in-blood-smears.

While this part of the project is more in the realms of computer science or biology, it is not without mathematics. As every computer image is composed of pixels (512 x 384, in my case), the image can actually be represented as a matrix. But as the images are colored, each pixel has an RGB value to denote the color. Therefore, I chose to represent each image in my dataset as a tensor (multi-dimensional matrix) with dimensions 512 x 384 x 3. Figure 1 below illustrates this concept with three separate but overlaid square matrices in 4-space.

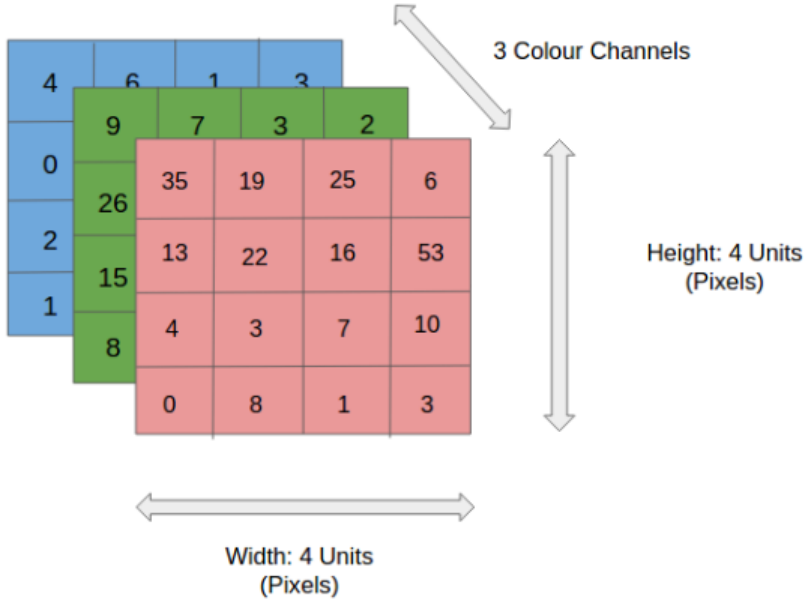

Fig 1: Illustration of the tensor image representation for a 4-pixel by 4-pixel image[2]

Using this representation, it becomes quite easy to represent the entire image as nested arrays which can be operated on according to the standard rules of linear algebra.

At this point, I normalized each color value by multiplying each tensor by the scalar $\frac{1}{255}$ as 255 is the maximum value any RGB cell could take. Thus, every value was a decimal on the closed interval [0, 1] which maps very well to the eventual neural network activation functions and probability analysis.

[2] Saha, Sumit. "A Comprehensive Guide to Convolutional Neural Networks - the ELI5 Way." Medium, Towards Data Science, 17 Dec. 2018, towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53.

INTRODUCING THE MODEL

In my research of different neural network architectures to use I identified three primary criteria in order of importance:

1) The network must be effective at image classification from a relatively small dataset. This point is by far the most important as it determines the possibility of the project accomplishing its aim.
2) The network must be relatively sparse in its connections between layers and/or have another method of minimizing computation. The program needs to be able to run on my personal laptop without a GPU and with only 16 Gb of RAM. These specifications are far below the norm for machine learning applications so the network must be able to compensate for the deficit.
3) The network must be relatively simple so as to not overextend the boundaries of this project. Already, the topic assumes knowledge of linear algebra and some multivariate calculus for the basic training so it is imperative that the architecture not introduce much more complexity.

The perfect solution seemed to be the Convolutional Neural Network (CNN), introduced by Yann LeCun et al. in 1988.[3] This model (with some modifications) has seen great popularity in recent years at the task of image classification as it deftly minimizes the number of computations required at a single time and is composed of relatively few layers. Moreover, the CNN is especially good at identifying spatial-dependencies in small areas -- a characteristic that makes it especially well-suited to my intended real-world application of identifying single trophozoites within a larger image.

[3] Lecun, Yann, et al. “Object Recognition with Gradient-Based Learning.” Shape, Contour and Grouping in Computer Vision Lecture Notes in Computer Science, 1999, pp. 319–345., doi:10.1007/3-540-46805-6_19.

STRUCTURE

The Dense Layer

Like most neural networks, the CNN is a composition of functions that transforms data into a usable output state. These functions can be imagined as "layers" which take in data, operate on it, and provide an output which the next layer takes in as output, and so forth. The essential predictive layer of the CNN, like many other neural networks, is called a "dense" layer. In the CNN, this is the last layer (save for the evaluation function). This layer corresponds to the functions

$$F = XW + \vec{b} \quad (1)$$

$$\sigma(F) = \frac{1}{1+e^{-F}} \quad (2)$$

Where X is the input matrix ($m \times n$), $\vec{b}$ is a "bias" vector ($m \times o$) corresponding to a random offset, W is a matrix ($n \times o$) of weights, and $\sigma(x)$ is the Sigmoid activation function which scales the output according to a logistic curve with horizontal asymptotes at -1 and 1. Here, m is the number of data points to be evaluated, n is the number of features of each data point (similar to the number of coordinates of a particular point in geometric space), and o is the number of possible output states for the expected prediction. In my model, for example, m will be 200 and o will be 1 because I had 200 images that corresponded to two mutually exclusive states -- a positive or negative diagnosis, represented in binary. The value of n will be calculated by other layers of the model and passed on to this dense layer.

This function is called a "dense" layer because the matrix multiplication causes every entry of the input to be scaled by every entry of the weight matrix as shown by the green and purple layers here:

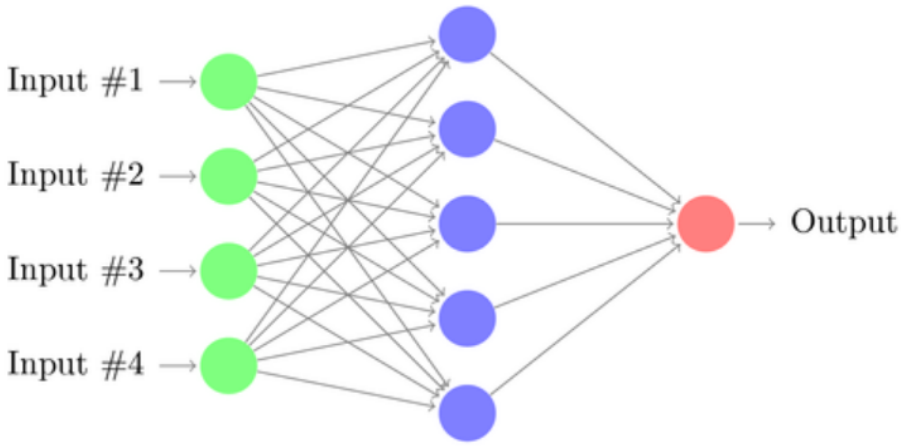

Fig 2: The "dense" neural network layer modeled as a multilayered perceptron[4]

The output of this layer will be a 200×1 orthonormal matrix which will characterize each of the inputs as positive or negative.

W and $\vec{b}$ will be initialized with completely random elements on the interval [0, 1]. In order to accurately predict anything, these must be tuned. The most common way to do this is with the Backpropagation Algorithm which implements a form of gradient descent to minimize error.[5] I introduce the loss function

$$E(y,\ a)\ =\ \frac{1}{2}\sum_{j}(\ y_{j}-a_{j})^{2} \tag{3}$$

Where y_j is the expected output of the model and a_j is the actual output of the model for a particular sample. By changing the weights and biases of the dense layer above to generate an output which minimizes this curve, we have a model for which the predictions match the expected value -- i.e. we have a model that correctly identifies an image as containing a *Plasmodium vivax* protozoa and thus a model which can diagnose malaria from the blood

[4] Agrawal, Aayush. "Building a Neural Network from Scratch." Medium, Towards Data Science, 18 Feb. 2019, towardsdatascience.com/building-neural-network-from-scratch-9c88535bf8e9.

[5] Nielsen, Michael A. "Neural Networks and Deep Learning." Neural Networks and Deep Learning, Determination Press, 1 Jan. 1970, neuralnetworksanddeeplearning.com/chap2.html.

sample. In the derivation following, I choose to update only the weights because the error from the bias term is generally small and the equations are very similar.

Intuitively, to create a new weight matrix, *W*,* we want to adjust the original weight according to the amount to which that particular weight affects the total error. In other words, the updated weights will be a matrix where each entry w_j* is of the form

$$w_j^* = w_j + \frac{\partial E}{\partial w_j} \tag{4}$$

for some entry w_j of the original weight matrix. We can then apply the chain rule to equations 1-3 above to find

$$\frac{\partial E}{\partial w_j} = \frac{\partial E}{\partial a_j}\frac{\partial a_j}{\partial f_j}\frac{\partial f_j}{\partial w_j} \tag{5}$$

Where a_j is the output of the activation function $\sigma(x)$ and f_j is a particular entry of the weighted input matrix $F = XW$ (assuming $\vec{b} = \vec{0}$ which will simplify the model without introducing too much error). We can then evaluate each product separately which is done easily because of our choice of functions:

$$\frac{\partial E}{\partial a_j} = a_j - y_j$$

$$\frac{\partial a_j}{\partial f_j} = \sigma'(f_j)$$

$$\frac{\partial f_j}{\partial w_j} = x_j$$

Simplifying:

$$\frac{\partial E_j}{\partial w_j} = (\sigma(f_j) - y_j) \circ \sigma(f_j)(1 - \sigma(f_j)) \circ x_j \tag{6}$$

Where the operator ◦ denotes element-wise matrix multiplication. Substituting equation (6) into equation (4) above, we have an updated weight matrix. Iteration of this process "trains" the

dense layer of the model, eventually producing a function that can be used to classify new images *not* in the training set with good accuracy.

The Convolutional Layer

Unfortunately, the dense layer described above only works for 2-dimensional matrices, not 3-dimensional tensors as explained in the "Modelling" section. To solve this problem and allow the dense layer -- the core of the neural network -- to accurately process colored images, we introduce additional data processing layers which also implement the algorithm's space and computation-saving features.

The first and most important layer of the CNN is the convolutional layer for which the architecture is named. This function is similar to the dense layer above but is more localized. The same way that the human eye is composed of various parallel sensory receptors which each activate unique neurons which are processed together in the brain, the convolutional layer is a linear combination of local tensor operations which are combined into one aggregate feature map. In practice, this looks like a small matrix ("the filter") sliding across each dimension of the tensor representation of the image, multiplying that matrix element-wise with each of the entries of the tensor across all three dimensions, and then summing those values to produce a single output value which describes the entire region of the image. As successive filter applications overlap each other, the resulting output is a 2-dimensional "feature map" which encodes the most important information from the 3-dimensional representation.

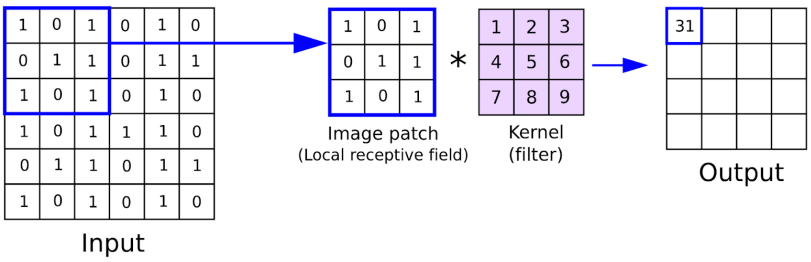

Fig 3: Convolution with an arbitrary input and filter[6]

This process is described mathematically by the formula

$$X(i,\ j)\ *\ F(k,\ l)\ =\ \sum_{k=1}^{m-1}\sum_{l=1}^{n-1} X(i\ +\ k,\ j\ +\ l)\,F(k,\ l) \qquad (7)^{7}$$

Which multiplies, sums, and shifts the $m \times n$ filter matrix F with indices k, l across each layer of the input tensor X whose entries are described with the i, j indices in the same process illustrated above.

In order to optimize the space and computation requirements of this algorithm, I take advantage of the fact that this operation is equivalent to a linear combination of the elements of each matrix by actually rearranging each partition of the matrix X into column vectors (appending the result of each layer in the tensor) and the associated kernel F into row vectors in order to take the dot product between the two, giving the same output. This method is functionally identical to the earlier double-sum representation but allows me to take advantage of the way matrices are represented in the programming language Python to more easily compute their convolution.

$$\begin{bmatrix}1&2&3\\4&5&6\\7&8&9\end{bmatrix} * \begin{bmatrix}1&1\\-1&-1\end{bmatrix} \rightarrow \begin{bmatrix}1&2&4&5\\2&3&5&6\\4&5&7&8\\5&6&8&9\end{bmatrix} \cdot \begin{bmatrix}1\\1\\-1\\-1\end{bmatrix} = \begin{bmatrix}-6\\-6\\-6\\-6\end{bmatrix} \rightarrow \begin{bmatrix}-6&-6\\-6&-6\end{bmatrix}$$

The equation above shows the process described above for an arbitrary $3 \times 3 \times 1$image and a 2×2 filter with step size 1. The partitions of the image matrix X are organized into a

[6] Bhattacharyya, Sreejani, et al. "What Is A Convolutional Layer?" Analytics India Magazine, 18 June 2021, analyticsindiamag.com/what-is-a-convolutional-layer.

[7] "Convolution." Glossary - Convolution, homepages.inf.ed.ac.uk/rbf/HIPR2/convolve.htm.

symmetric matrix composed of the vectors which are dotted with the rearranged filter. The product is then reshaped once more to create the feature map.

After the feature map is generated, the dimensionality of the input space is reduced to 2, at which point it can be passed to successive layers. Empirically,[8] using the Rectified Linear Unit activation function on each element of the feature map at this point improves training accuracy by introducing nonlinearities into an otherwise completely linear process.

$$R(x) = max(0, x)$$

Effectively, this sets all negative values to zero, amplifying the effects of the positive contour features. This function marks the first time the algorithm develops beyond the capacity of a simple linear regression.

The Pooling Layer

After the ReLU function, a pooling layer is used. This other nonlinear function helps to reduce the input space to successive layers by further amplifying only the most important features of a local area: intuitively, the absolute location of a certain feature in an image is not as important as the local features which make it possible to identify at all. One of the most common functions for this process is Max Pooling, which partitions the input image matrix and then outputs the maximum entry from each sub-matrix, creating a smaller matrix that contains only the most important elements of the feature map.[9] The two most important parameters for a max pooling function are the filter size and the stride.

---

[8] Brownlee, Jason. "A Gentle Introduction to the Rectified Linear Unit (ReLU)." Machine Learning Mastery, 20 Aug. 2020, machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/.

[9] Scherer, Dominik, et al. "Evaluation of Pooling Operations in Convolutional Architectures for Object Recognition." Artificial Neural Networks – ICANN 2010 Lecture Notes in Computer Science, 2010, pp. 92–101., doi:10.1007/978-3-642-15825-4_10.

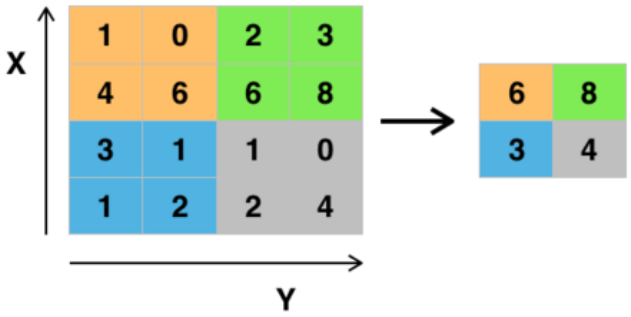

Fig 4: Max Pooling[10]

In this figure, the filter size is 2×2 and the stride size is 2 because each block matrix has dimensions 2×2 and none of the block matrices overlap (the first entry of each new partition corresponds to position $A_{i+2\,j}$ relative to the first position $A_{i\,j}$ of the previous partition). From each partition, the maximum value is returned. The choice of filter and stride size depends on the input data. My images, for example, are 512×384 pixels so the standard[11] 2×2 filter and 2-step stride will work because

$$512 \bmod 4 \equiv 0\,, \qquad 384 \bmod 4 \equiv 0$$

so the matrix will tile evenly. If this were not true, however, I would need to choose a different filter or stride size and/or zero-pad one dimension of my input.

Another pooling function such as Average pooling (where the arithmetic mean of the entries is propagated forward) could also be chosen at this step. I chose to focus on Max Pooling because it is simpler to explain and implement because it generally makes faster and more accurate models than other forms of pooling.[12]

---

[10] Riznyk, Olexa. "Max Pooling Image." Wikimedia Commons, Creative Commons Attribution-Share Alike 4.0 International license. https://en.wikipedia.org/wiki/File:Max_pooling.png

[11] Géron, Aurélien. Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow: Unsupervised learning techniques. United States, O'Reilly Media, Incorporated, 2019.

[12] Milosevic, Nemanja. "Convolutions and Convolutional Neural Networks." Introduction to Convolutional Neural Networks, 2020, doi:10.1007/978-1-4842-5648-0_12.

After the Pooling layer, a CNN can have multiple further iterations of convolutional and pooling layers, often with other forms of down-sampling in between. The input size for my model is sufficiently small, however, that I only need to use one convolution and one pooling filter to achieve satisfactory results. Finally, the dense layer derived above is applied and the model produces its output.

Training

Of course, because the convolutional kernel described above began as completely random numbers, it too must be trained. This is accomplished with exactly the same algorithm as the training of the Dense layer: gradient descent via backpropagation. Using the same chain rule process as in the dense layer section:

$$\frac{\partial E}{\partial F_i} = \frac{\partial E}{\partial a_i}\frac{\partial a_i}{\partial f_i}\frac{\partial f_i}{\partial x_i}\frac{\partial x_i}{\partial P_i}\frac{\partial P_i}{\partial R_i}\frac{\partial R_i}{\partial C_i}\frac{\partial C_i}{\partial F_i} \tag{8}$$

The first three factors are exactly the same as calculated in the dense layer:

$$\frac{\partial E}{\partial a_i} = a_i - y_i$$

$$\frac{\partial a_i}{\partial f_i} = \sigma'(f_i)$$

$$\frac{\partial f_i}{\partial x_i} = w_i$$

The P_i factors are related to the MaxPooling layer. As this layer only identified the maximum among the set of inputs, no values were changed -- this layer affected the dimensionality of the gradients but not the values. For the computer implementation, the training in this step involves rescaling the dimensions of the error which is propagated backwards by populating an empty

array of the same dimensions as the convolutional output with the gradient values from the dene layer. In calculating $\frac{\partial E}{\partial F_i}$, this factor can be ignored.

The R_i factor refers to the ReLu function introduced above, a piecewise function where $\frac{dR}{dx} = 0,\ x < 0$ and $\frac{dR}{dx} = 1,\ x > 0$. Like in the pooling layer, all 0 values of the derivative can be ignored because the backpropagation algorithm is only concerned with those values which are propagated forward in the first place. Further, because the function's partial derivative with respect to the output of the convolution will be 1 at all meaningful points, the entire factor can be ignored as it will have no effect on the product.

Finally, using equation (7):

$$C = \sum_{k=1}^{m-1} \sum_{l=1}^{n-1} X(i\ +\ k,\ j\ +\ l)\ F(k,\ l)$$

$$\frac{\partial C}{\partial F} = X$$

which makes perfect sense considering that the convolution is simply the dot product of the two. At last simplifying equation (8):

$$\frac{\partial E}{\partial F_i} = (a_i - y_i) \circ \sigma'(f_i) \circ w_i \circ x_i$$

Which yields the final formula to update the convolutional kernel:

$$K_i^* = K_i + [(a_i - y_i) \circ \sigma'(f_i) \circ w_i \circ x_i] \qquad (9)$$

IMPLEMENTATION AND INTERPRETATION

Though the math itself yielded a simple matrix equation, the actual process of implementing the algorithm in Python proved quite difficult. I made use of the standard library NumPy[13] which introduces a way to turn normal Python arrays into n-dimensional matrix representations and which gives generalizations of functions like the dot product, cross product, and matrix multiplication.

First, I implemented the easy parts: the activation functions, the pooling layer, and the final model evaluator. For that final function, I used a function called Binary Cross Entropy which is very good for models whose final output comes in the forms of two mutually exclusive events -- as my positive and negative diagnosis predictions do. The formula for that loss function is

$$L(q) = -\frac{1}{N}\sum_{i=1}^{N} y_i \cdot log(p(y_i)) + (1 - y_i) \cdot log(1 - p(y_i))$$

where y is the "label" (1 for positive diagnosis and 0 for negative) and *p(y)* is the predicted probability of positive diagnosis over the *N* samples. While I could have implemented this function myself, I chose to just use the version already implemented in the popular machine learning framework Tensorflow to avoid needing to reshape the final matrix again.

The convolution layer was also relatively easy to implement using the dot product method I explained above. Actually turning the matrices into vectors is an operation that would have been difficult to do by hand or to implement myself but which was built into the NumPy library already.

The dense layer, I implemented as a doubly-nested loop which cycled the entire training dataset multiple times ("epochs"), using equations (4) and (6) to update the weights after each

[13] NumPy, numpy.org/.

image. Using multiple epochs allowed me to refine the model and improve its accuracy on the training set by having more "points" to which to "fit the curve." My first model used a dense layer with 128 outputs, followed by the final dense layer with 1 output. This dual structure greatly strengthened the predictive power of the model.

All that accomplished, the structure of the model was complete. All that remained was to actually process my data and to train the model. It was in this processing phase that I had the most trouble. Unlike my earlier efforts, here I had no mathematical formula to reference. All the next functions simply implement predefined matrix operations on a given input; here I had to *create* a matrix myself. My final solution runs in $O(n^4)$ time, iterating over each pixel-tuple that forms the input image and individually placing those values into my input matrix for each picture to create a 4-dimensional tensor. This is not simple, elegant, or efficient. As it was, loading in the dataset took almost 2 hours of computation time plus more for image processing and for me to align all the dimensions properly. For any other project, it would be *far* better to simply use already-available functions to do this automatically.

Having finally loaded the images into a tensor and built the model, I began the training process. Here, I ran the model multiple times, recording the accuracy (evaluated by the Binary Cross Entropy function described above) as a percentage of images classified correctly for the training database and the testing database (20% of the original dataset which was not included in the training to measure the ability of the model to extrapolate past its learned heuristics) against the number of epochs. For several hours of failed trials, my accuracy for both trials was stuck at 50% -- no learning was occurring. After spending a good deal of time debugging each layer of my model, I realized the problem was caused by an incorrect balance of model features: I was

using too many convolutional filters and not enough dense layer neurons. In other words, the program was able to process the images very well but could not actually interpret them.

To fix this, I began adjusting the model hyperparameters. The first trial that showed any success whatsoever achieved roughly 65% testing accuracy.

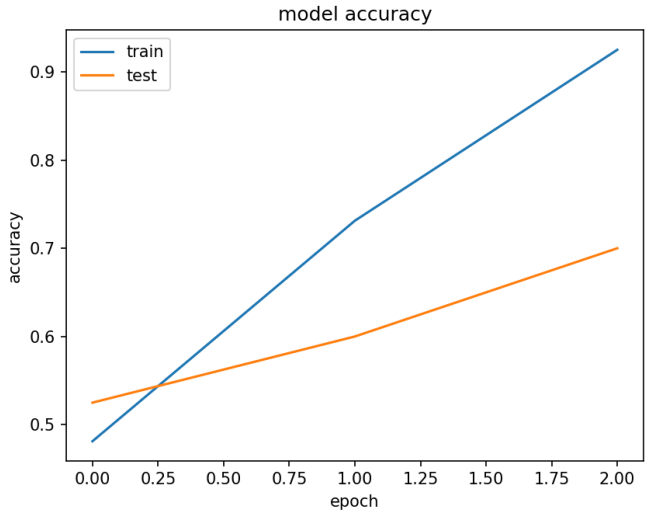

Fig 5: The first successful model

This trial used a batch size of 2 (meaning 2 images were processed at a time), 16 filters of size 3×3in the single convolutional layer, 1 max pooling layer, a ReLu-activated dense layer with 128 separate weight matrices to train in parallel, a final 1-dimensional dense layer using the sigmoid activation function, and 3 epochs. Additionally, I introduced a new constant coefficient, η, to equation (4):

$$w_j^* = w_j + \eta \frac{\partial E}{\partial w_j}$$

This "learning rate," which I set to 0.0001 for this trial, helped moderate the speed of gradient descent, reducing the influence of confounding random errors. This is a hyperparameter I manually adjusted across trials to get the best results.

Fig 4 above shows a large gap between the training accuracy (which approached 95%) and the testing accuracy (~65%). In machine learning, this gap is called "overfitting" and is

caused by insufficient data. It means that the predictive algorithm has become too specific to the training values and generalizes poorly to new data.

In later trials, I adjusted several of the hyperparameters above. In my most successful attempt, I used 25 convolutional filters instead of 16, 5 epochs instead of 3, 512 dense neurons instead of 128, and a learning rate of 0.001 instead of 0.0001. Additionally, I introduced a "dropout" layer after the MaxPooling function which set a random 10% of values in the input to 0 to introduce further nonlinearities and reduce the computational space. By increasing the number of filters and dense neurons by so much, this network had a total of 797,984,641 trainable parameters.

This architecture allowed me to achieve 75% test accuracy as shown in the figure below, though overfitting still occurred. In running this model, however, my personal laptop utilized 100% of available RAM and 96% of available CPU for 3 hours. If the algorithm were even slightly more complicated or if another process tried to start, the entire computation would have crashed. Clearly, any further improvements achieved by simply increasing the number of trainable parameters via convolutional filters or dense predictive matrices would need a more powerful device or a much more efficient algorithm.

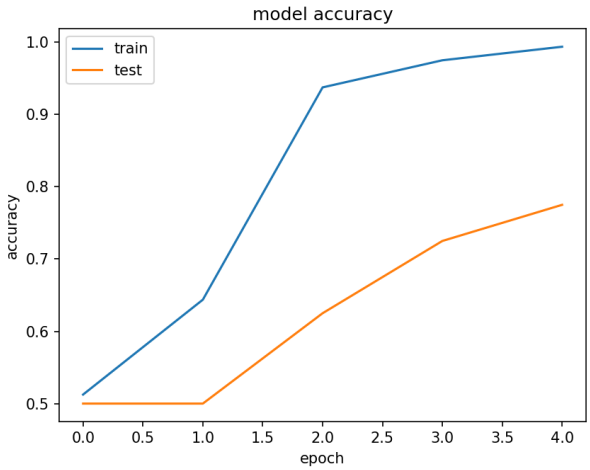

Fig 6: The most successful model

To further decrease the overfitting, a larger dataset would be required. Two-hundred images is considered very small compared to most machine learning projects which regularly use hundreds of thousands of images.

CONCLUSION

In beginning this project I set out to accomplish three things: apply the linear algebra I learned in my courses at Georgia Tech to a real world problem, delve more deeply into the mathematical bases for the field of computer science and artificial intelligence specifically, and produce a useful, working model that could help diagnose malaria. My first and foremost goal was to be challenged and to learn throughout the process. This I accomplished.

I knew from the beginning that my chosen topic would not be easy but throughout the process, I ended up facing more difficulty than originally expected. For example, I knew that linear algebra is used in computer science to represent and transform images. I did not expect, however, to need multivariable calculus -- especially to such a large degree. Further, I did not anticipate the problems I would face in implementing the model; I expected the mathematical derivations to be the most difficult part of the project.

Ultimately, I am proud of the work I put in and the product I created. Though a model of 75% accuracy is not yet fit to be used in actual medical settings, I was at least able to prove that machine learning has an application in malaria diagnosis. Further, my attempts at further training the model were not limited by the algorithm itself but by the computational power of my personal laptop -- with a more powerful computer it seems entirely likely that a much more accurate model could be attained with only small variations to the architecture I described here. If I were to develop the project further, I would also use a much larger dataset of less

well-curated images. Currently, the model learned only a very specific subset of images of blood smears that do not necessarily reflect the conditions, quality, or level of preparation of real-world imaging in the places where malaria diagnoses remain common. Finally, I recognize that my particular implementation of these functions is not efficient nor scalable. Some layers contained triple or quadruple-nested loops and I have no doubt that further optimization is possible.

WORKS CITED

CS231n Convolutional Neural Networks for Visual Recognition, cs231n.github.io/convolutional-networks/.

"5 Techniques to Prevent Overfitting in Neural Networks." *KDnuggets*, www.kdnuggets.com/2019/12/5-techniques-prevent-overfitting-neural-networks.html.

Agrawal, Aayush. "Building a Neural Network from Scratch." *Medium*, Towards Data Science, 18 Feb. 2019, towardsdatascience.com/building-neural-network-from-scratch-9c88535bf8e9.

Bhattacharyya, Sreejani, et al. "What Is A Convolutional Layer?" *Analytics India Magazine*, 18 June 2021, analyticsindiamag.com/what-is-a-convolutional-layer.

Brownlee, Jason. "A Gentle Introduction to the Rectified Linear Unit (ReLU)." *Machine Learning Mastery*, 20 Aug. 2020, machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/.

"Convolution." *Glossary - Convolution*, homepages.inf.ed.ac.uk/rbf/HIPR2/convolve.htm.

"Convolution." *Convolution - an Overview | ScienceDirect Topics*, www.sciencedirect.com/topics/mathematics/convolution.

Géron, Aurélien. *Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow: Unsupervised learning techniques.* United States, O'Reilly Media, Incorporated, 2019.

Lecun, Yann. "Deep Learning & Convolutional Networks." *2015 IEEE Hot Chips 27 Symposium (HCS)*, 2015, doi:10.1109/hotchips.2015.7477328.

Lecun, Yann, et al. "Object Recognition with Gradient-Based Learning." *Shape, Contour and Grouping in Computer Vision Lecture Notes in Computer Science*, 1999, pp. 319–345.,

doi:10.1007/3-540-46805-6_19.

"Load and Preprocess Images : TensorFlow Core." *TensorFlow*, www.tensorflow.org/tutorials/load_data/images.

Milosevic, Nemanja. "Convolutions and Convolutional Neural Networks." *Introduction to Convolutional Neural Networks*, 2020, doi:10.1007/978-1-4842-5648-0_12.

Nielsen, Michael A. "Neural Networks and Deep Learning." *Neural Networks and Deep Learning*, Determination Press, 1 Jan. 1970, neuralnetworksanddeeplearning.com/chap2.html.

NumPy, numpy.org/.

Riznyk, Olexa. "Max Pooling Image." Wikimedia Commons, Creative Commons Attribution-Share Alike 4.0 International license. https://en.wikipedia.org/wiki/File:Max_pooling.png

Saha, Sumit. "A Comprehensive Guide to Convolutional Neural Networks - the ELI5 Way." *Medium*, Towards Data Science, 17 Dec. 2018, towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53.

Sanyal, Parikshit. "Malaria Parasite in Blood Smears." *Kaggle*, 1 Nov. 2021, www.kaggle.com/cmacus/malaria-parasite-in-blood-smears.

Scherer, Dominik, et al. "Evaluation of Pooling Operations in Convolutional Architectures for Object Recognition." *Artificial Neural Networks – ICANN 2010 Lecture Notes in Computer Science*, 2010, pp. 92–101., doi:10.1007/978-3-642-15825-4_10.

Shenoy, Anirudh. "How Are Convolutions Actually Performed Under the Hood?" *Medium*, Towards Data Science, 13 Dec. 2019,

towardsdatascience.com/how-are-convolutions-actually-performed-under-the-hood-226523ce7fbf.