Detecting Malaria With Convolutional Neural Networks

Aditya Gupta

Adlai E. Stevenson High School | Grade 11

Computer Science | Design

Research Advisor | Dr. Susan Brontman

Contents

Acknowledgements

Thank you to all the people that helped me in pursuing this research, including my older brother for inspiring me to enter the field of deep learning, my parents for making it possible for me to research this topic, and my sponsor, Dr. Brontman, for her invaluable advice, guidance, and feedback. I could not have completed this project without you!

1 Problem

Malaria is a disease caused by a *Plasmodium* parasite, commonly transferred by mosquitoes. Over two billion people are at risk of malaria every year, and the World Health Organization estimates that there were over 247 million cases of malaria in 2021, with over 619,000 deaths. Individuals in third-world countries have particularly high risks of contracting malaria, with over 95% of malaria cases occurring in underdeveloped areas and children under 5 accounting for approximately 80% of deaths (World Health Organization, 2023). Furthermore, in these less developed countries, inadequate medical staffing and resources makes it difficult to handle malaria on a widespread scale with current technology. Currently, malaria is primarily diagnosed in these third-world countries with cheap microscopes but a time consuming and potentially error prone procedure of counting blood smears, or rapid diagnostic tests (RDTs) which have frequent false positives and cannot detect all instances of *Plasmodium* parasites (Buck & Finnigan, 2023). High staff turnover and a frequent lack of expertise among medical personnel on malaria create additional challenges, leading to the delay of malaria diagnosis (Mosquera-Romero et al., 2018).

However, quick diagnosis of malaria is essential to effective treatment of the disease and the prevention of further complications (World Health Organization, 2023). Currently, the major issue with microscopy malaria testing is the time consuming process of manually identifying and counting potentially infected blood smears. Convolutional neural networks can autonomously perform such a task in a far more efficient and quick manner than humans can, and modern neural networks have consistently outperformed humans in terms of accuracy at such tasks. Therefore, by creating an autonomous neural network to diagnose malaria from relatively inexpensive and low-quality cell imaging, this design aims to help malaria be detected in a cost-effective, scalable, sustainable, and efficient manner, so that malaria can be more easily treated by medical staff.

2 Design Criteria

In order to create a useful neural network that outperforms current methods of malaria detection, several design criteria must be met:

- The model is able to correctly classify instances of malaria with an accuracy above 95%, which is above the current estimated level that microscopy tests typically provide. Accuracy is calculated by recording the total number of examples the model has been tested on and the number of correct classifications it has made during the model evaluation phase. Achieving a greater-than 95% accuracy makes the neural network more accurate than most microscopy and rapid diagnosis tests.

- The model is lightweight, able to be locally run and produce a diagnosis in less than 0.1s after training (measured with the tqdm library). A simple neural network is significantly cheaper and faster than current microscopy and rapid diagnosis testing.

- The recall of model is above 95%. Recall measures the proportion of false negatives out of all true positives, which helps prevent the model from missing instances of malaria.

- The model should be applicable to real-world data and avoid overfitting to training data. Several regularization methods will be applied to the model to prevent overfitting, and learning will be assessed with a test set of data to confirm the model has not overfit.

- The model can work successfully with suboptimal imaging. By using low-resolution images captured from cheaper cell imaging technology and applying data augmentation, the model will be trained on images of varying quality so that it can work on a variety of data, including cheaper microscopes commonly used in third-world countries.

Parameters to be adjusted include: model hyperparameters, such as learning rate and batch size; model structure, such as the number of layers, size of each layer, and amount of residual; functions, such as the loss function, activation function, and optimizer; and regularization parameters, such as the dropout values, type of weight-decay, and weight-decay rate.

3 Review of Literature

Deep learning has burst into the lives of many people in the last year, with the rise of generative AI assistants pushed by companies like OpenAI and Google. However, for more than a decade, it has been silently powering many essential applications, such as fraud detection and speech recognition. Crucially, deep learning has recently been gaining recognition in the medical industry, especially in areas like cancer research (IBM Cloud Education, n.d.). However, other areas of medical research—such as the detection of diseases, like malaria, on the cellular level have largely remained unchanged for a significant period of time (Mosquera-Romero et al., 2018). The scalability, accuracy, and cost-effectiveness of deep learning with neural networks makes them ideal to diagnose malaria more effectively(IBM Cloud Education, n.d.).

3.1 Deep Learning

Deep learning is a subset of machine learning, a subset of artificial intelligence that analyzes trends in data to predict or classify new input data based on patterns learned from previous examples. Typically, machine learning models need specially structured data that requires human effort—deep learning models can learn features of the data directly from the data, without human intervention (IBM Cloud Education, n.d.). This limits human bias and error from affecting the model, while producing better results with less manpower. In fact, deep learning models have consistently outperformed humans in the areas of bioinformatics and medical image processing (Alzubaidi et al., 2021). However, deep learning requires more data and computing power than traditional machine learning (IBM Cloud Education, n.d.).

3.2 Neural Networks

Deep learning is implemented with neural networks, loosely inspired by the biology of the brain. Neural networks are organized into layers of neurons, with each neural network having an input layer, a series of hidden layers, and an output layer. Each neuron in each hidden layer connects to each neuron in the previous and following layer. Each neuron outputs a single value, which affects the outputs of neurons in the following layer, which in turn affects the output of layers following that layer, repeatedly throughout the network (Nielsen, 2015).

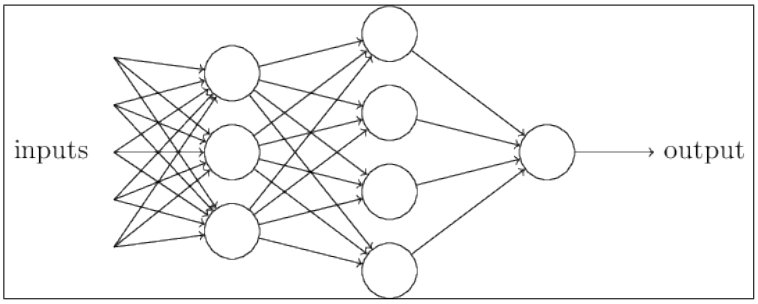


Figure 1: Structure of a neural network (Nielsen, 2015)

In order to determine the value to output, a neuron assigns unique weights to each neuron in the previous layer. Each weight represents how important information from the corresponding neuron in the previous layer is and if that information makes it more likely for the "current" neuron to fire (positive weight) or less likely (negative weight). Each weight (w) is multiplied with the output of the corresponding neuron. The products are summed together, and a bias (b) is subtracted. The bias is an inverse measure of how active a neuron is (Nielsen, 2015). This sum is then composed with an activation function (A), which determines the output (x)—a higher sum corresponds to a higher output value (Sharma et al., 2020). The first artificial neurons, perceptrons, used a binary step function:

$$A(w,b) = \begin{cases} 0 & \sum_{j=1}^{n} w_j x_j - b_j \leq 0 \\ 1 & \sum_{j=1}^{n} w_j x_j - b_j \geq 0 \end{cases} \tag{1}$$

Figure 2: The binary step function, used by perceptrons (Nielsen, 2015)

Initially, these weights and biases are randomized. However, for the neural network to truly learn, it needs to optimize these weights and biases to get the desired activations in the output layer for certain activations in the input layer. To do this, a small change in the weights and biases should reflect a proportional and predictable change in the activation of various neurons. Perceptrons do not accomplish this, since a small change in a weight or bias could potentially completely flip the output of a neuron (Nielsen, 2015).

For this reason, the binary step function is obsolete. The activation function needs to be continuous for the neural network to learn, and it also needs to be nonlinear to model the complexity of the data being fed to the neural network. Commonly, the ReLU function, or Rectified Linear Unit, is used instead.

Figure 3: The binary step function (left) and ReLU function (right) (Baheti, 2021)

ReLU can be defined as $R(x) = \max(0, x)$ where x is the weighted sum of the activations with the bias subtracted. One issue with ReLU is that large negative values lead to a zeroed output, known as the dying ReLU problem. Leaky ReLU aims to solve this by using a small negative gradient when $x < 0$ rather than the zero-gradient seen in Figure 3 (right), but still has limitations (Sharma et al., 2020).

3.3 Gradient Descent and Backpropagation

Initially, all the weights and biases in a model are randomized, resulting in poor performance. To become more accurate, the model must adjust its weights and biases appropriately over the course of several iterations of all of the data, or epochs (Nielsen, 2015). To do this, the model must know how incorrect it was compared to the expected or labeled value of the data. A loss function calculates the error, or loss, between the actual activations in the output layer and the expected outputs, with softmax cross-entropy typically being used for classification tasks (Wang et al., 2022). Therefore, for the model to learn and maximize its accuracy, it must minimize the loss function (Ruder, 2016). Since the loss function relates the weights and biases of a model, as input, to the level of error, or output, the model must find the weights and biases that correspond to a minimum (Nielsen, 2015).

Gradient descent is used to find a local minimum in the loss function. The intuition behind gradient descent is similar to a ball rolling into a valley (local minimum) (Nielsen, 2015). The model computes the partial derivatives of the loss function, which reveal the direction of steepest increase in the loss, known as a gradient. By adjusting the parameters in the opposite direction of the gradient, the model can take steps to reach the local minimum (Stanford Education, n.d.). However, this also means gradient descent can get stuck at saddle-points, where the gradient of the function is very small in all directions (Ruder, 2016). In the ball analogy, saddle-points would be plateaus of flat land slowing down the ball's momentum resulting in it getting stuck (Nielsen, 2015).

The learning rate defines the size of the steps that the model takes towards the local minimum. A small learning rate will result in slow, but consistent progress towards the local minimum, while a larger learning rate can approach the minimum faster but also runs the risk of overshooting the local minimum, and is thus less predictable. Learning rate schedulers adjust the learning rate of a neural network over time. Typically, these schedulers start with a larger learning rate, which decreases over time as the model approaches the local minimum (Ruder, 2016).

Although finding the global minimum in the loss function is ideal, it is computationally infeasible to do so, which is why local minimums are found with gradient descent (Stanford Education, n.d.). Interestingly, the local minima that gradient descent finds off randomly initialized weights and biases are roughly equivalent, suggesting that the networks find common features in data despite different weights and biases (Arpit et al., 2017).

In the ball analogy, there were only two parameters to consider when finding the gradient: the $\langle x, y \rangle$ vector for the ball. Modern neural networks can have billions of weights and biases and need to compute a massive vector that describes how to adjust each one. Computing partial derivatives of the loss function to find the gradient with traditional calculus is computationally infeasible, so an algorithm called backpropagation is used (Nielsen, 2015).

Backpropagation begins at the back of the model, where the outputs are. In order to adjust the output of a neuron to the expected value, which will minimize loss, the model can adjust the weights of the neuron and the bias of the neuron. The model will also adjust the activation of the neurons in the previous layer; to do that, it must adjust the weights and biases of those neurons connecting them to the layer before them. Applying this process recursively, the error and necessary changes to the model are propagated from the back to the front. The model computes the ideal changes to every neuron in the network for each neuron in the output layer individually, and then sums together these changes. This vectorized sum is the gradient (Nielsen, 2015).

When a new gradient is calculated for every training example, it is known as stochastic gradient descent. Stochastic gradient descent is fast to calculate, but also takes a suboptimal path to the local minimum. To remedy this, batch gradient descent is sometimes used: the vectors for each training example are summed together, creating a gradient for all the training data. Although this results in a more direct path to the local minimum, the gradient takes a long time to compute for large datasets. Mini-batch gradient descent captures the benefits of both stochastic and batch gradient descent by subdividing the training data into mini-batches and computing the gradient for each mini-batch (Patrikar, 2019).

3.4 Regularization

The number of epochs required for learning optimizer to effectively adjust the weights and biases of the model through backpropagation varies with each neural network, so generally a large number of epochs are chosen. However, if there are too many epochs, the model can overfit to the training data, essentially memorizing the training data and losing the ability to generalize to unseen testing data, as it becomes too sensitive to random noise (Arpit et al., 2017). Interestingly, Arpit et al. found that while deep neural networks with sufficient capacity would memorize datasets of random noise, they would typically minimize the cost of structured data faster, suggesting deep neural networks make important observations about the data before they begin to memorize.

Various methods of regularization exist to prevent a model from overfitting. One is using a validation set of data after each epoch. The model is tested on the validation set, but does not backpropagate, so it does not learn from the validation set. Rather, the model is being assessed on whether over the past epoch loss on the validation set decreased from in prior epochs. If the model does not improve over a few epochs, it is likely overfitting to the training data, and training is stopped early (Koehrsen, 2018).

L2 regularization decreases the weights of the model each time it is updated, preventing the model from building up too much complexity and overfitting. Dropout is another method of regularization, which disables a few nodes each time backpropagation is performed. The idea is to constantly create a new smaller set of nodes passing information, which not only simplifies the model, but forces the model to compress patterns into fewer nodes instead of overfitting (Jain, 2023). However, if the weight decay or dropout value is too high, the model can become too simple and underfit the data (Alzubaidi et al., 2021).

3.5 Convolutional Neural Networks and Transfer Learning

Images hold a massive amount of data. A single 244x244 pixel color image requires at least 178,608 weights for just a single layer in a traditional neural network. Traditional neural networks do not scale well with image data due to the massive number of required parameters making them computationally inefficient. Thus, convolutional neural networks (CNNs) are primarily used for image data instead (Gavrilova, 2021).

Unlike traditional neural networks, CNNs share weights between neurons and have sparse interactions—layers where every neuron does not connect to every neuron in the previous layer—in order to simplify and speed up training. Structurally, a CNN also differs from traditional neural networks, using convolutional and pooling layers in addition to traditional fully connected layers (Alzubaidi et al., 2021).

Images first go through preprocessing, which scales images to the same resolution for data consistency. Preprocessing also applies data augmentation, a process where images are slightly cropped and rotated to artificially increase the training sample, since the content of an image does not change even if its position or orientation is slightly altered. Thus, augmentation helps the model learn features of the data instead of overfitting (Koehrsen, 2018). Preprocessing also converts the image into a tensor of values from 1 to 255, representing the brightness of each color channel (RGB) for each pixel (Gavrilova, 2021).

In a convolutional layer, a filter of $N * N$ pixels slide over the image tensor, multiplying the value of the filter by the values of a pixel. The values over the $N * N$ area are then summed together to form a new value, as seen in Figure 4. As the filter slides over the image, these values form a new tensor, called a feature map. Feature maps help extract relevant information from the image. Various feature maps are created by different layers, which are then compressed and combined together in pooling layers, which take the average of a region in the feature map and form a smaller feature map. Pooling layers help reduce sensitivity to noise and speed up training. These feature maps are then flattened and fed to a fully-connected layer which determines output (Gavrilova, 2021).

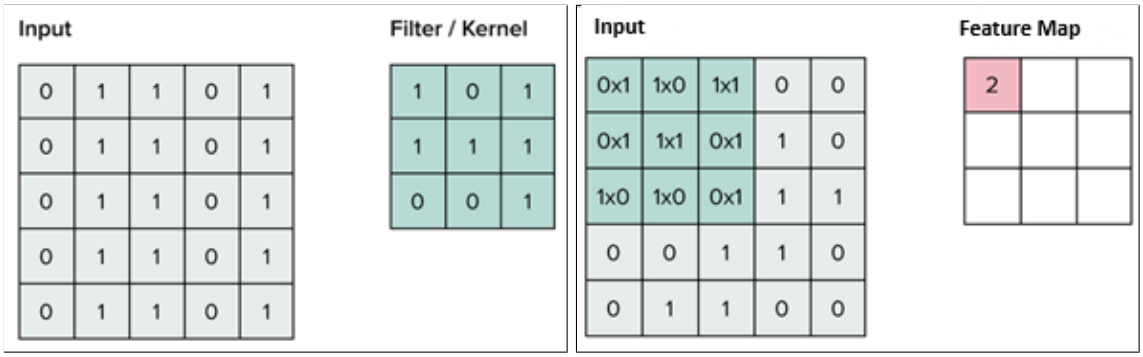


Figure 4: The convolution operation (Gavrilova, 2021)

Even with these optimizations, convolutional neural networks can still take a long time to train. The idea of transfer learning is to take a pretrained, general convolutional neural network and freeze the early convolutional layers in the model, which extract general information about the model. Then, the later layers of the model—which extract more specific features of the model—are retrained, specializing the model to one unique use case. Gradually, the earlier layers can be unfrozen and slowly retrained as needed. This approach transfers some of the general knowledge already learned by the pretrained neural network, drastically reducing training time (Koehrsen, 2018). One common model for transfer learning is the ResNet-50, a residual neural network (He et al., 2016).

3.6 Residual Neural Networks

As mentioned earlier, certain activation functions can lead to vanishing gradients. This problem is exacerbated in dense neural networks, such as convolutional neural networks. Although activation functions like Leaky-ReLU attempt to solve the issue, they have had limited success and their own issues (Alzubaidi et al., 2021).

Residual neural networks attempt to solve the issue by introducing skip-connections, which allow layers deeper in the neural network to skip some of the intermediate layers, so that if any intermediate layer hurts the performance of the neural network, it gets skipped. Additionally, the neural network adds residual over layers, so that gradients do not vanish but instead propagate over layers. Essentially, if $f(x_L)$ is the input to a layer L and $g(x_{L-1})$ is

the output of the previous layer, then with residual neural networks, $f(x) = g(x_{L-1}) + x_{L-1}$, rather than $f(x_L) = g(x_{L-1})$ in traditional neural networks (He et al., 2016).

Residual neural networks also have bottleneck layers, which reduce the number of neurons in a neural network layer from the previous layer, forcing the model to compress the most information into fewer neurons. This helps the model generalize and avoid overfitting. Residual neural networks allow for extremely deep and optimized neural networks with limited complexity and relatively short training times, making them ideal for transfer learning.

3.7 Malaria

Malaria is a life-threatening disease caused by a *Plasmodium* parasite transmitted to humans by the *Anopheles* mosquito, with over two billion individuals at risk of contracting malaria each year (Buck & Finnigan, 2023). The World Health Organization estimates that in 2021, there were over 247 million cases of malaria, among which there were approximately 619,000 deaths. Malaria disproportionately impacts third-world countries, with over 95% of malaria cases occurring in underdeveloped areas. Furthermore, individuals under the age of 5 accounted for more than 80% of deaths (World Health Organization, 2023).

Current methods of malaria detection include microscopy testing and rapid diagnostic testing (RDTs). Although RDTs can provide quicker results at a low cost, they are limited to detecting just one of five infectious parasitic species, and often produce false-positives (Buck & Finnigan, 2023). Although microscopy testing remains the "gold-standard" for malaria detection, it requires expensive equipment, medical expertise, and is very time consuming for medical professionals, as it requires them to count hundreds of stained blood smears and examine them for the malaria parasite (Center for Disease Control and Prevention, 2018). In third world countries, outbreaks of malaria with high volumes of patients can occur, making microscopy testing impossible (Mosquera-Romero et al., 2018).

Mosquera-Romero et al. conducted a detailed study of malaria detection and treatment methods used in Ecuador, a third-world country in South America. They found that so-

cioeconomic factors, a lack of resources, and a weak pre-existing medical network made it difficult to diagnose malaria in a timely fashion. Additionally, a high turnover and lack of medical staff and a general lack of expertise on the disease further slowed any diagnosis of the infection (Mosquera-Romero et al., 2018). This is problematic, since quick diagnosis and treatment of malaria is essential to prevent further complications; while early treatment often leads to a rapid recession of symptoms, delayed treatment allows for intensified fevers and for the infection to spread to the brain. Known as cerebral malaria, this can cause the patient to fall into a coma and even die (Buck & Finnigan, 2023).

The repeated task of examining hundreds of blood smears for the malaria parasite can be potentially automated by a convolutional neural network. Neural networks have consistently outperformed humans for classification tasks, including in the medical field (IBM Cloud Education, n.d.). Thus, a neural network model could provide a higher accuracy than microscopy testing for malaria at a fraction of the cost, as well as providing results far more quickly than traditional microscopy testing since a neural network can examine images far faster than a human can. Furthermore, a neural network would be scalable so that it could handle malaria outbreaks in a way traditional testing methods cannot. A well-trained neural network could also help medical experts unfamiliar with the disease. Additionally, they can be trained on low-quality imaging of cells, allowing for the use of cheaper and more accessible equipment.

Thus, for a neural network to be an effective solution to diagnosing malaria, the model would need to be lightweight, so that it is cheap to run and does not require significant computing power, enabling the model to be run locally in remote locations, where many malaria outbreaks occur. Additionally, such a model would need to be trained on lower-quality images to ensure cheaper microscopes could be used to identify malaria. The model would also need to produce consistent, accurate results quickly.

4 Materials

- 64-bit computer with a CUDA-capable GPU running Windows, macOS, or Linux
- Anaconda 3 Distribution with PyTorch
- Internet access
- Jetbrains Dataspell IDE
- National Institutes of Health (NIH) Malaria Cell Images Dataset (Retrieved 12/23/23)

5 Design Plan

5.1 Setup

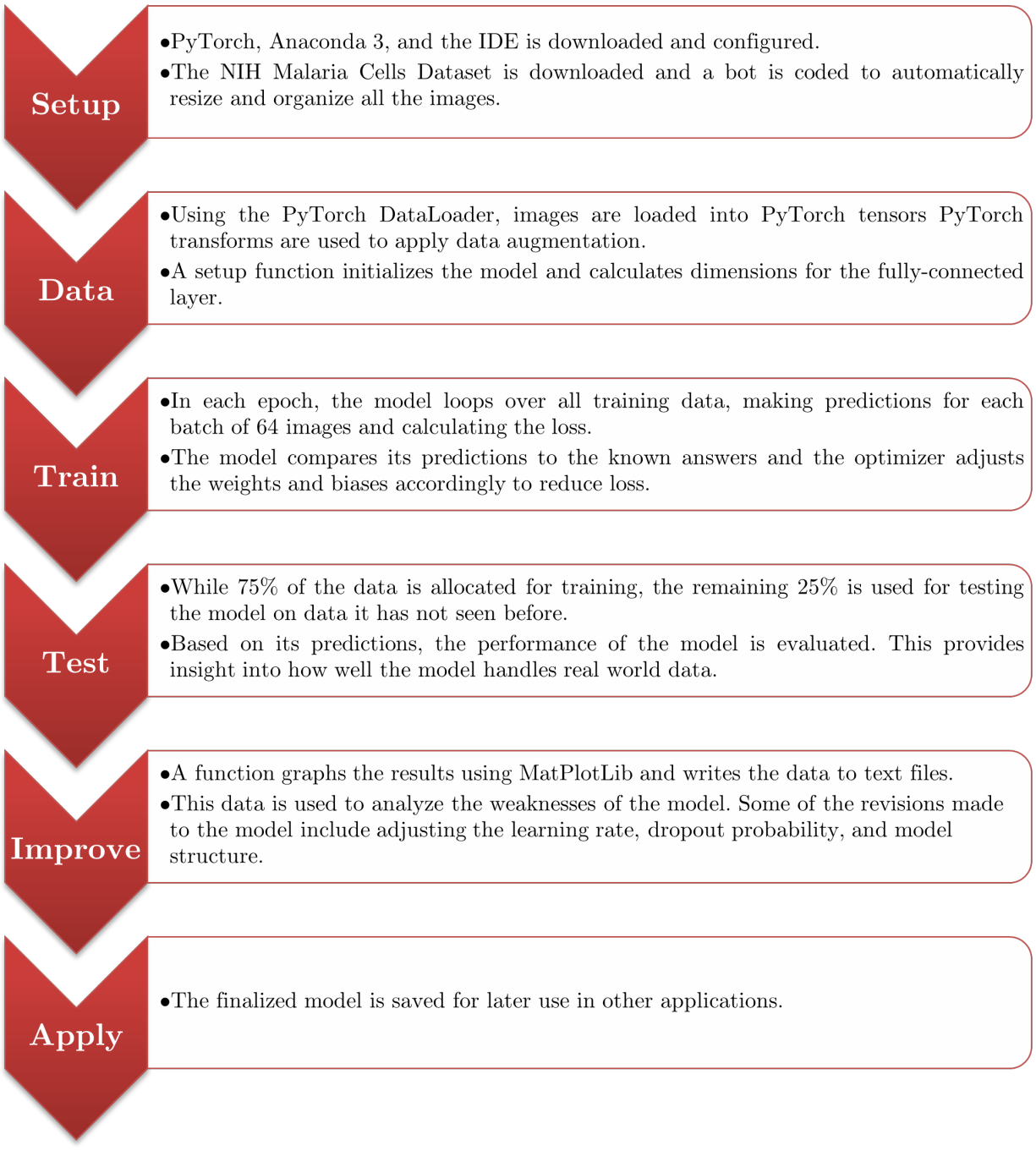
1. Downloaded and configured Anaconda 3 Distribution.
2. Used the conda installer to download and configure PyTorch.
3. Downloaded and configured DataSpell IDE.
4. Exported the Malaria Cell Images Dataset by the National Institutes of Health (NIH).
5. Developed a Python program to reorganize the data into a format ideal for PyTorch's Dataloader class using Python's os library and rescale the images to a fixed resolution of 100x100 pixels with the Image class.
6. Inspected and cleaned the image data with PyTorch functions and DataLoader.

5.2 Developing the Source Code

7. Extended the neural network module of PyTorch by creating a child class to implement the convolutional neural network classifier.
8. Wrote the initialization function for the class, which implements each layer of the model, the optimizer (AdamW) and loss function (Cross Entropy Loss).
9. Implemented the CNN layers by creating a modular CNNCell Class before using inheritance to implement the CNN Classifier.

10. Used the PyTorch Dataloader class to import the image data and convert it to PyTorch tensors. Split the data into training and testing data splits with a 75:25 ratio using a seeded generator for reproducibility.
11. Used PyTorch transforms to implement data augmentation.
12. Wrote a training function for the CNN class, which calculated accuracy and loss with each batch and backpropagated the loss using the Adam optimizer.
13. Wrote a function to maintain and update performance statistics such as loss and accuracy arrays, which were analyzed to inform the revisions and improvements made to the prototype while also recording its performance during testing.
14. Wrote a testing function for the CNN class which used PyTorch's evaluation mode to prevent unexpected behavior and data leakage that could occur if batch normalization statistics were used.
15. Wrote a function that generated and saved graphs of training and testing data and wrote that data to text files.
16. Wrote driver code that created an instance of the CNN class and specified its parameters.
17. Used PyTorch to train and test the designed model. The designed model automatically calculated its accuracy and loss through the statistics function written in step 13.
18. Exported the training and testing results of the neural network.
19. Analyzed the accuracy, loss, and recall data gathered by the training and testing functions across epochs, which informed the revisions and improvements made to the model.
20. Steps 17–19 were repeated several times to optimize the model. Changes included adjusting the learning rate, padding, and stride, reworking the dropout and regularization parameters and introducing a weight-decay optimizer (AdamW), and applying transfer learning on the ResNet-50, which substantionally altered the structure of the model. These changes are further explained in the Results and Discussion section.

5.3 Overall Flowchart



6 Results and Discussion

The initial prototype of the convolutional neural network had 6 convolutional layers, each with batch normalization and a ReLU activation function. The filter (kernel) had a size of 5x5 pixels with a padding of 4 pixels and stride of 1 pixel. There were also 3 pooling layers, one after each pair of convolutional layers. No regularization techniques were applied for the initial prototype. The model had one fully-connected linear layer, and used the Adam optimizer with a learning rate of $\alpha = 0.05$ with a batch size of 64. A fixed seed was used for generating training and testing splits. In total, 30,034 images were used to train and test the model, with 16,255 being malaria-infected cells and 13,799 being uninfected. Images were scaled to 100x100 pixels and data augmentation was applied through PyTorch transforms.

Figure 5: Cross-Entropy Loss Per Epoch for Version 1	
Epoch	**Cross-Entropy Loss**
1	1828.8645
2	220.6017
3	215.3022
4	210.2717
5	210.3488
Test	72.0428
Best	210.2717

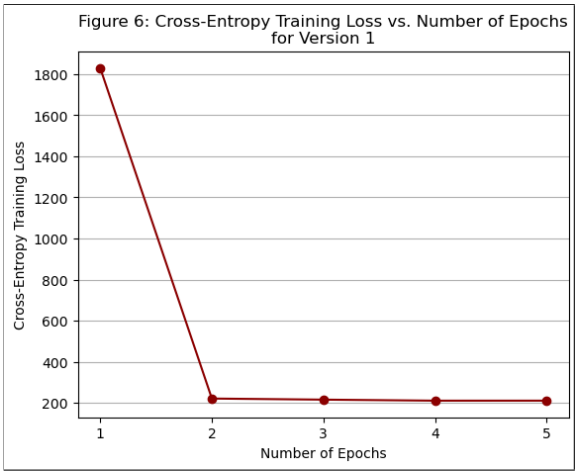
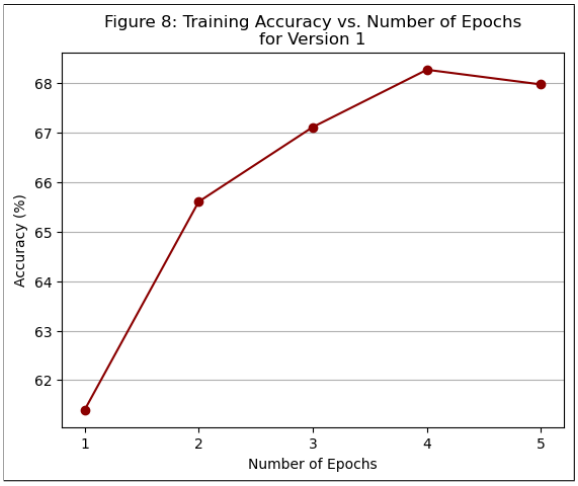


Figure 7: Accuracy Per Epoch for Version 1	
Epoch	**Accuracy (%)**
1	61.3947
2	65.6072
3	67.1165
4	68.2750
5	67.9776
Test	67.5450
Best	68.2750



Figures 5–6 show the cumulative loss, or total error, of the model across each epoch or iteration over all the data. A lower loss generally indicates a more accurate overall model, and the optimizer or the model's learning function attempts to minimize loss in the model. Therefore, loss considers not only the precision of the model, like accuracy does, but also the confidence of the model in its predictions—something that accuracy does not measure. Cross-entropy loss does not have any unit. Additionally, note that testing loss is expected to be several times lower than training loss as three times as many images are used for training compared to testing, so the cumulative loss should be higher. Furthermore, the complexity of a model can influence the total loss, as the cumulative loss sums together the loss of all neurons, so with more neurons, the model may have a higher loss, even though the loss per neuron is lower.

As seen in Figures 5–6, the loss of the model plateaued after epoch 2, failing to decrease significantly. This is reflected by Figures 7–8, where the accuracy fluctuated after epoch 3, indicative of overfitting, which occurs when the model begins to memorize training data instead of learning features of the data. The model did not perform particularly admirably, either, achieving a testing accuracy of just 67.545%. Testing accuracy (labeled "Test" in Figure 7) is the accuracy of the model in evaluation mode on a unique seperate set of data that it was not trained on.

To address the issue of quickly failing to learn new features, while also improving the overall accuracy, a number of changes were made to the second version of the model. The learning rate was decreased to $\alpha = 0.01$, and the kernel size was decreased to 2 pixels (with a padding of 1 pixel) to increase resolution of feature maps. Dropout regularization was introduced to reduce overfitting, with three dropout layers added. Also, an additional convolutional layer was added before the third pooling layer. Each of these changes was individually tested and found to be beneficial before being implemented.

Figure 9: Cross-Entropy Loss Per Epoch for Version 2	
Epoch	**Cross-Entropy Loss**
1	2128.2155
2	193.3296
3	145.6502
4	68.2861
5	64.1012
Test	31.0001
Best	64.1012

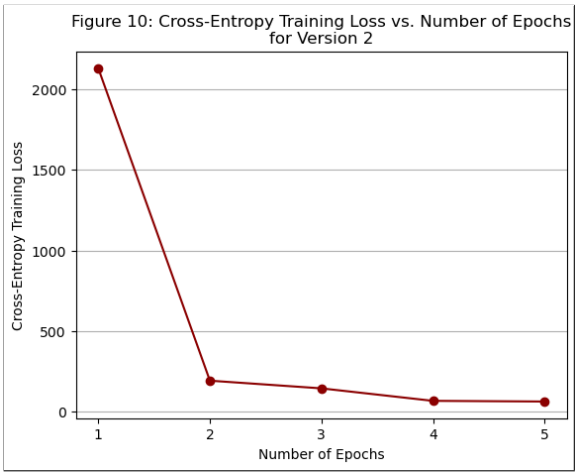
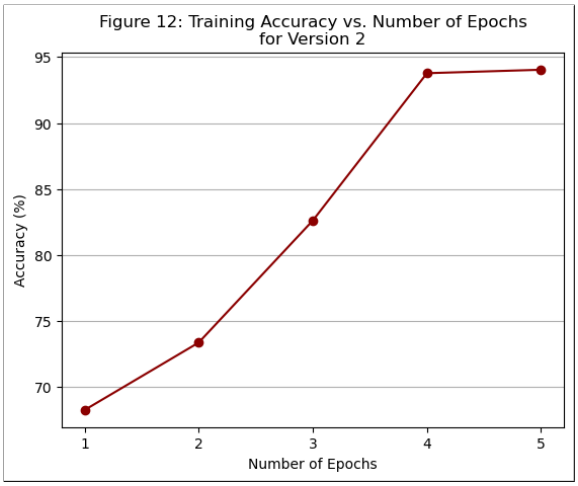


Figure 11: Accuracy Per Epoch for Version 2	
Epoch	**Accuracy (%)**
1	68.2573
2	73.3665
3	82.6127
4	93.7722
5	94.0341
Test	89.2479
Best	94.0341



As seen in Figure 9, the model had a higher initial loss, as there were more parameters due to the additional convolutional layer, and the loss per epoch is the sum of the loss across all parameters. The model actually performed better initially, as seen in Figure 11 as it had a higher initial accuracy than the first version did. As seen in Figure 10, the second iteration of the model decreased its loss rapidly and continued to decrease in loss well beyond when the first iteration plateaued. As a result, accuracy continued to substantially increase in the epochs 3–4 due to the introduction of dropout and smaller learning rate, and the second model displayed improvement over the first model, with a testing accuracy of 89.248%.

The model did display some overfitting towards the end of training, as seen in Figures 12 and 14. Additionally, the model still fell short of achieving the goal of 95% accuracy as specified in the design criteria. Therefore, the dropout probability was increased and the initialization parameters of the model were iteratively adjusted. In addition, the Adam opti-

mizer was changed for the AdamW optimizer, which implements weight-decay regularization with added momentum in proportion to step size.

Figure 13: Cross-Entropy Loss Per Epoch for Version 3	
Epoch	**Cross-Entropy Loss**
1	532.5393
2	80.4902
3	58.8891
4	53.6350
5	48.7871
Test	17.5850
Best	48.7871

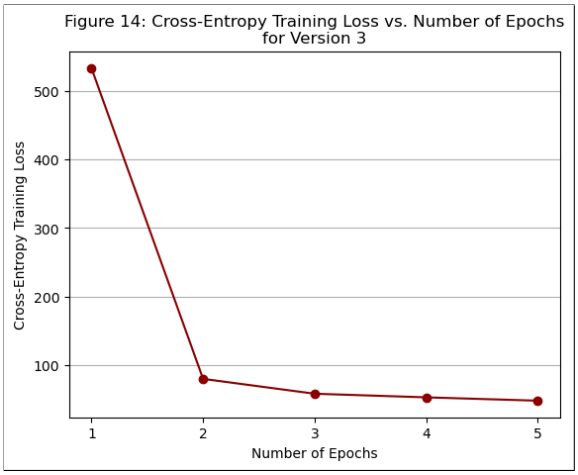
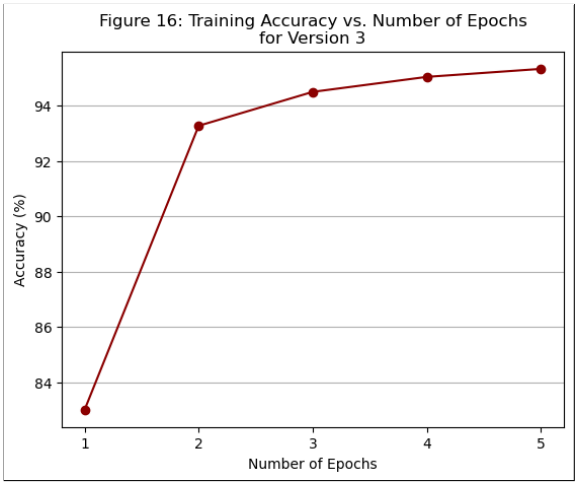


Figure 15: Accuracy Per Epoch for Version 3	
Epoch	**Accuracy (%)**
1	83.0078
2	93.2706
3	94.4957
4	95.0373
5	95.3258
Test	95.5244
Best	95.5244



Adjustment to the initialization parameters significantly increased the initial learning of the model, which reached 93.271% by the end of epoch 2. Additionally, loss was also substantially decreased, as seen in Figures 13–14. The model continued to slightly improve in accuracy for the remaining epochs, achieving a final testing accuracy of 95.524%, as seen in Figures 15–16. Although this accomplished the initial design criteria of achieving at least 95% accuracy, further changes were made to improve the performance of the model. The model structure was changed from 7 convolutional layers, 3 pooling layers, 3 dropout layers, and a fully connected layer to the 50 layer structure of ResNet-50, using residual and increasing dropout regulation to limit impact to the efficiency of the model. Most parameter optimizations implemented across testing in the previous versions were preserved.

Figure 17: Cross-Entropy Loss Per Epoch for Version 4	
Epoch	**Cross-Entropy Loss**
1	117.5687
2	66.0245
3	62.4362
4	57.1389
5	53.3281
Test	17.3407
Best	53.3281

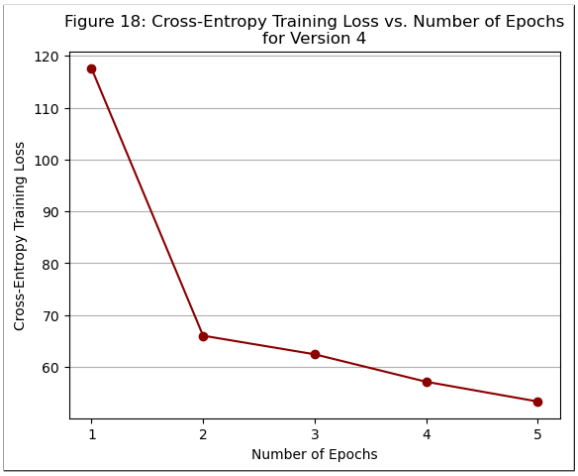
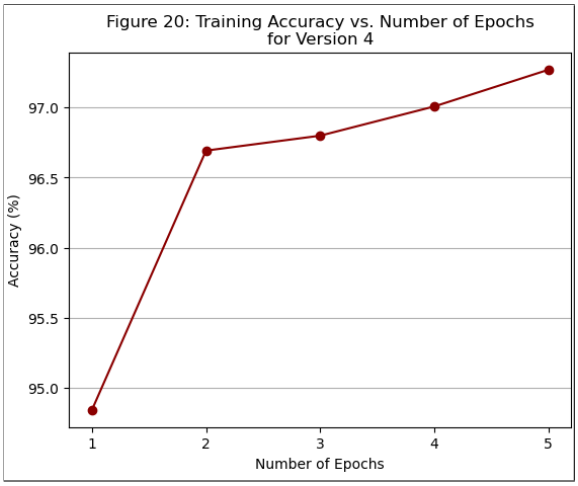


Figure 19: Accuracy Per Epoch for Version 4	
Epoch	**Accuracy (%)**
1	94.8420
2	96.6930
3	96.7995
4	97.0082
5	97.2701
Test	97.4867
Best	97.4867
Recall	98.2114



The fourth model produced a final testing accuracy of 97.487%, well above the design criteria. Additionally, the model achieved a recall of 98.211%, which also exceeded the design criteria of 95%. Recall is essentially a measure of the false negative rate of a model, which can be particularly problematic in medical situations, as a patient who has malaria and is misdiagnosed may face extremely harmful consequences. Use of the ResNet-50 architecture and the implementation of transfer learning helped the model continue learning and improving in accuracy across all five epochs. Cumulative loss did not decrease substantially from version 3, as there was an increase in the number of neurons—however, the loss per neuron did decrease.

Figure 21: Training Accuracy Per Epoch for Each Version of the Model				
Accuracy (%)	**Version**			
Epoch	**1**	**2**	**3**	**4**
1	61.3947	68.2573	83.0078	94.8420
2	65.6072	73.3665	93.2706	96.6930
3	67.1165	82.6127	94.4957	96.7995
4	68.2750	93.7722	95.0373	97.0082
5	67.9776	94.0341	95.3258	97.2701
Test	67.5450	89.2479	95.5244	97.4867
Best	68.2750	94.0341	95.5244	97.4867

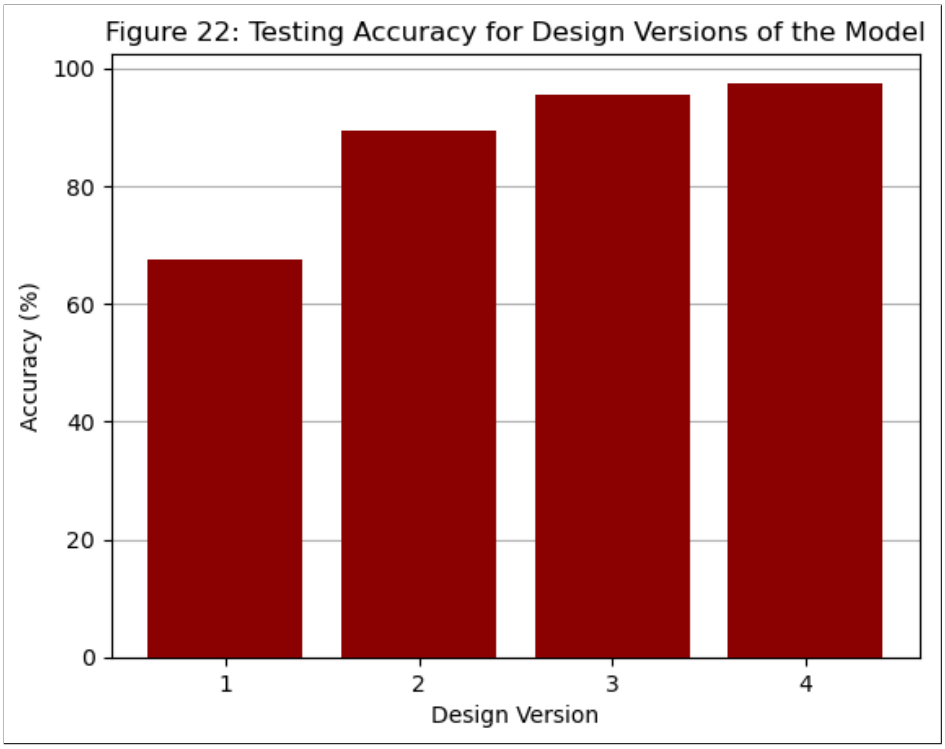


Figure 21 shows the training accuracy per epoch and final testing accuracy of the model across all four versions. Figure 22 shows a graph of the final testing accuracy for all four versions, allowing for an easy comparison of the accuracy of the models, which increased from version 1 to version 4.

7 Conclusion

This design project developed a convolutional neural network to help improve the speed, accuracy, and consistency of identifying malaria in third-world countries while reducing the cost of malaria testing. An efficient convolutional neural network with relatively few parameters was developed to automate the repetitive and time-consuming task of identifying cells infected by malaria under a microscope. Malaria still remains a major issue in many third-world countries, especially those in Africa, as the World Health Organization estimates there were more than 247 million cases of malaria in 2021 with over 600,000 deaths, necesitating improvements to the detection and treatment process of malaria (World Health Organization, 2023).

The initial prototype of the model did not achieve the accuracy or recall constraints on the model, which were both set at 95%, as it only achieved an accuracy of 67.545%. In order to achieve these constraints, a number of changes, including lowering the learning rate, implementing dropout and adjusting dropout probability, shrinking the kernel size, implementing transfer learning on the ResNet-50 architecture, and implementing weight decay in the optimizer were made to improve the model performance over three different redesigns. These changes resulted in the model achieving a final testing accuracy of 97.487% and recall of 98.211%, achieving the target thesholds. Because the model was trained on a dataset developed with suboptimal microscope imaging technology and data augmentation was applied, the model is applicable to real-world situations. The model minimized overfitting through the use of dropout and weight decay regularization. By limiting the number of parameters and using regularization, the model was computationally lightweight, classifying roughly 181 images per second.

Many potential errors were avoided by the controlled nature of computational projects. However, there was still randomness in the weight initialization of the model, which can lead to minor variations in the model's performance. A small bias may have arisen from a slight inequality in the ratio of infected and uninfected cell images that the model trained on, which

was roughly 16:13. Ideally, the ratio would be 1:1, but the slightly imperfect distribution will not hinder model performance in the real world.

In practice, this model can be implemented for a relatively low-cost in most third-world countries, including in remote regions, as after training the model does not require internet or high-computational power to function. The model can accurately identify instances of malaria more consistently than current rapid-tests, while doing so with significantly cheaper equipment and more quickly than microscopy testing, making it an ideal solution to identifying infected patients, especially in high-volume situations.

One limitation of the model is that it requires a separate system to digitize the data of microscope slides before it can be utilized by the neural network. Such a process is known as whole slide imaging and requires a separate digital camera if the microscope is not digital. Thus, additional complexity is introduced for the model to run with inexpensive microscopes in the field, which introduces additional delays and costs.

Additional optimizations, such as the use of learning rate schedulers, further parameter tuning, and revising dropout mechanisms are currently being explored to improve the effectiveness of the neural network. Furthermore, the possibility of testing the model in third-world countries, specifically Kenya, to help identify malaria more rapidly in the real world is currently being explored. Such an endeavor could not only serve as additional training data to help the model perform better, but could also help diagnose infections in a faster, more accurate, and more cost-effective manner.

Reference List

Alzubaidi, L., Zhang, J., Humaidi, A. J., Al-Dujaili, A., Duan, Y., Al-Shamma, O., Santamaria, J., Fadhel, M. A., Al-Amidie, M., & Farhan, L. (2021). Review of deep learning: Concepts, CNN architectures, challenges, applications, future directions. *Journal of Big Data*, (8). https://doi.org/10.1186/s40537-021-00444-8

Arpit, D., Jastrzebski, S., Ballas, N., Krueger, D., Bengio, E., Kanwal, M. S., Maharaj, T., Fischer, A., Courville, A., Bengio, Y., & Lacoste-Julien, S. (2017, August). A closer look at memorization in deep networks. In D. Precup & Y. W. Teh (Eds.), *Proceedings of the 34th international conference on machine learning* (pp. 233–242). PMLR. https://proceedings.mlr.press/v70/arpit17a.html

Baheti, P. (2021, May). *Activation functions in neural networks [12 types & use cases]*. Retrieved October 14, 2023, from https://www.v7labs.com/blog/neural-networks-activation-functions

Buck, E., & Finnigan, N. A. (2023, July). *Malaria*. National Library for Medicine. https://www.ncbi.nlm.nih.gov/books/NBK551711/

Center for Disease Control and Prevention. (2018, July). *Malaria diagnosis*. Retrieved October 15, 2023, from https://www.cdc.gov/malaria/diagnosis_treatment/diagnosis.html

Gavrilova, Y. (2021, August). *Convolutional neural networks for beginners*. Retrieved October 4, 2023, from https://serokell.io/blog/introduction-to-convolutional-neural-networks

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

IBM Cloud Education. (n.d.). *What is deep learning?* Retrieved October 3, 2023, from https://www.ibm.com/topics/deep-learning

Jain, S. (2023, September). *An overview of regularization techniques in deep learning (with Python code)*. Retrieved October 10, 2023, from https://www.analyticsvidhya.com/blog/2018/04/fundamentals-deep-learning-regularization-techniques/

Koehrsen, W. (2018, November). *Transfer learning with convolutional neural networks in PyTorch*. Retrieved October 4, 2023, from https://towardsdatascience.com/transfer-learning-with-convolutional-neural-networks-in-pytorch-dd09190245ce

Mosquera-Romero, M., Zuluaga-Idarraga, L., & Tobon-Castano, A. (2018). Challenges for the diagnosis and treatment of malaria in low transmission settings in San Lorenzo, Esmeraldas, Ecuador. *Malaria Journal*, (17). https://doi.org/10.1186/s12936-018-2591-z

Nielsen, M. A. (2015). *Neural networks and deep learning*. Determination Press. http://neuralnetworksanddeeplearning.com/

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., ... Chintala, S. (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library. *Advances in Neural Information Processing Systems*, *32*, 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

Patrikar, S. (2019, September). *Batch, mini batch & stochastic gradient descent*. Retrieved October 13, 2023, from https://towardsdatascience.com/accuracy-precision-recall-or-f1-331fb37c5cb9

Ruder, S. (2016). An overview of gradient descent optimization algorithms. *Insight Center for Data Analytics*. https://doi.org/10.48550/arXiv.1609.04747

Sharma, S., Sharma, S., & Athaiya, A. (2020). Activation functions in neural networks. *International Journal of Engineering Applied Sciences and Technology*, *4*(12), 310–316.

Stanford Education. (n.d.). *Optimization: Stochastic gradient descent*. Retrieved October 12, 2023, from https://cs231n.github.io/optimization-1/

Wang, Q., Ma, Y., Zhao, K., & Tian, Y. (2022). A comprehensive survey of loss functions in machine learning. *Annals of Data Science*, (9), 187–212. https://doi.org/10.1007/s40745-020-00253-5

World Health Organization. (2023, March). *Malaria*. Retrieved October 8, 2023, from https://www.who.int/news-room/fact-sheets/detail/malaria

Appendix A: Full Code

The full code of the final model is shown below. The code can also be viewed at https://github.com/aaggupta07/detecting-malaria.

```
# Base Libraries
import numpy as np
import matplotlib.pyplot as plt

# Organizing Image Data
import os
from PIL import Image

# Timing
from tqdm.auto import tqdm
from time import time

# Torch imports
from torch import device
from torch import cuda
from torch import zeros
from torch import Generator
from torch import sum
from torch import where
from torch import argmax
from torch import tensor

from torch.utils.data import random_split
from torch.utils.data import DataLoader

from torch.nn import Module
from torch.nn import Conv2d
from torch.nn import BatchNorm2d
from torch.nn import MaxPool2d
from torch.nn import Sequential
from torch.nn import Linear
from torch.nn import CrossEntropyLoss
from torch.nn import Dropout

from torch import optim
import torch.nn.functional as F

from torchvision import transforms
from torchvision import datasets
from torchvision import models
```

```
# Image setup
path = "OBSCURED FOR PRIVACY"
folders = os.listdir(path)
rescale = 100

for folder in folders:
    files = os.listdir(path + '/' + folder)
    for idx, file in enumerate(tqdm(files, desc="Files Renamed/Resized: ")):
        image = Image.open(os.path.join(path, folder, file))
        image = image.resize((rescale, rescale))
        new_path = os.path.join(path, folder, str(idx))
        image.save(str(new_path) + ".png")


class ResNetClassifier(Module):
    def __init__(self, lr, tol, batch_size,
                 epochs, num_classes, scale, data_path, seed):
        super(ResNetClassifier, self).__init__()
        self.lr = lr
        self.tol = tol
        self.batch_size = batch_size
        self.epochs = epochs
        self.num_classes = num_classes
        self.scale = scale
        self.data_path = data_path
        self.seed = seed
        self.device = device("cuda" if cuda.is_available() else "cpu")

        self.loss_history = []
        self.acc_history = []
        self.epoch_acc_history = []
        self.test_loss = -1
        self.test_acc = -1

        self.model = models.resnet50(weights=models.ResNet50_Weights.DEFAULT)
        .to(self.device)

        self.in_dims = self.model.fc.in_features
        self.fc = Linear(in_features=self.in_dims, out_features=num_classes)
        self.loss = CrossEntropyLoss()
        self.optimizer = optim.AdamW(self.parameters(), lr = self.lr)

        self.to(self.device)

        self.train_data_loader = None
        self.test_data_loader = None
        self.get_data()
```

```
def forward(self, batch_data):
    batch_data = batch_data.clone().detach().to(self.device)
    output = self.model(batch_data)
    return output

def get_data(self):
    tf = transforms.Compose(
        [transforms.RandomHorizontalFlip(), transforms.ToTensor()])

    all_data = datasets.ImageFolder(self.data_path, transform=tf)
    train_data, test_data = random_split(all_data, [0.75, 0.25],
        generator=Generator().manual_seed(self.seed))
    self.train_data_loader = DataLoader(train_data,
        batch_size=self.batch_size, shuffle=True, num_workers=4)
    self.test_data_loader = DataLoader(test_data, batch_size=self.batch_size,
        shuffle=True, num_workers=4)

def train_(self):
    time_started = time()

    for i in range(self.epochs):
        header = f"Epoch {i+1}/{self.epochs}"
        print(header)
        print("-" * len(header))

        loader = self.train_data_loader
        self.train()

        # For plotting and statistics
        epoch_loss = 0
        epoch_acc = []

        # Main training loop
        for in_data, label in loader:
            self.optimizer.zero_grad()

            # Get model prediction and determine how well it did
            label = label.to(self.device)
            prediction = self.forward(in_data)
            classes = argmax(prediction, dim=1)

            wrong = where(tensor(classes != label).to(self.device),
                          tensor([1.]).to(self.device),
                          tensor([0.]).to(self.device))

            # Calculate loss and accuracy
```

```
            acc = 1 - sum(wrong) / self.batch_size
            batch_loss = self.loss(prediction, label)
            epoch_loss += batch_loss.item()

            # Train through backpropagation
            batch_loss.backward()
            self.optimizer.step()

            # Statistics
            self.acc_history.append(acc.item())
            epoch_acc.append(acc.item())

        elapsed = time() - time_started
        print(f"Finished epoch {i+1} with loss {epoch_loss} and accuracy of
            {np.mean(epoch_acc)} in time "
            f"{int((elapsed//60)//60)}:{int(elapsed//60)}:{int(elapsed%60)}")

        self.loss_history.append(epoch_loss)
        self.epoch_acc_history.append(np.mean(epoch_acc))
        print()

    print("Training complete...")

def test_(self):
    self.eval()

    time_started = time()
    total_loss = 0
    total_acc = []

    for in_data, label in self.test_data_loader:

        # Compare correct answers to model
        label = label.to(self.device)
        prediction = self.forward(in_data)
        classes = argmax(prediction, dim=1)

        wrong = where(tensor(classes != label).to(self.device),
                      tensor([1.]).to(self.device),
                      tensor([0.]).to(self.device))

        acc = 1 - sum(wrong) / self.batch_size
        batch_loss = self.loss(prediction, label)

        total_acc.append(acc.item())
        total_loss += batch_loss.item()
```

```
        elapsed = int(time() - time_started)
        self.test_acc = np.mean(total_acc)
        self.test_loss = total_loss

        print(f"Finished with loss {self.test_loss} and accuracy of
        {self.test_acc} in {elapsed // 60}:{elapsed % 60}")


model = ResNetClassifier(lr=0.01, tol=2, batch_size=64,
        epochs=5, num_classes=2, scale=100,
        data_path="OBSCURED FOR PRIVACY", seed=251)
model.train_()
model.test_()

# Graphs
v = input("Enter version #: ")
fig = int(input("Enter figure #: "))
epoch_dims = np.arange(1, len(model.loss_history)+1, 1)
path = 'data-graphs/'
v_num = int(v[0])

# Graph 1
plt.plot(model.loss_history, marker='o', color='darkred')
plt.title(f"Figure {fig}: Cross-Entropy Training Loss vs.
Number of Epochs\nfor Version {v_num}")
plt.xlabel("Number of Epochs")
plt.ylabel("Cross-Entropy Training Loss")
plt.xticks(np.arange(len(model.loss_history)), epoch_dims)
plt.grid(axis='y')
plt.savefig(path + 'v' + v + "-train-loss-graph.png", bbox_inches='tight')
plt.show()

# Graph 2
plt.plot(model.epoch_acc_history, marker='o', color='darkred')
plt.title(f"Figure {fig+2}: Training Accuracy vs.
Number of Epochs\nfor Version {v_num}")
plt.xlabel("Number of Epochs")
plt.ylabel("Accuracy (%)")
plt.xticks(np.arange(len(model.epoch_acc_history)), epoch_dims)
plt.grid(axis='y')
plt.savefig(path + 'v' + v + "-epoch-acc-graph.png", bbox_inches='tight')
plt.show()


# Datafiles
def write_to(fw, arr, header):
    fw.write(header + '\n[' + str(arr[0]))
```

```
    for num in range(1, len(arr)):
        fw.write(', ' + str(arr[num]))
    fw.write(']\n\n')


try: v
except NameError: v = input("Enter version: ")

with open(f'data-graphs/v{v}-data.txt', 'w') as f_out:
    write_to(f_out, model.loss_history, "Training Loss")
    write_to(f_out, model.epoch_acc_history, "Training Accuracy (Per Epoch)")

    f_out.write(f"Testing Loss, Accuracy\n{model.test_loss},
    {model.test_acc}\n\n")

    write_to(f_out, model.acc_history, "Training Accuracy (Per Batch)")
```