The Design of a Neural-Network Machine Learning Model for Malware Detection

Quillen Flanian

Conjure-2

6/24/2023

-In accordance with the ACE core values, I have neither given nor received unauthorized aid on this paper.

Abstract

The rise of machine learning revolutionized the modern approach of solving complex problems by allowing us to harness a computer's superior pattern recognition ability. We utilized this modern technology to design a machine learning model able to detect malware within a given input file. We created a six-layer neural network and trained our model with several hundred benign and malicious files. Our model learns the patterns within malicious and benign files and is able to accurately distinguish between them. The neural network detects any dangerous files and predicts if the file is malicious or benign, enhancing our system security against malicious adversaries.

1. Introduction

Machine learning can be traced all the way back to 1952, when IBM's Arthur Samuel first coined the term in his model to predict the outcome of a game of checkers [1]. The recent explosion of advancements within the field of machine learning displays how useful this technology can be. Machine learning allows us to use a computer's superior pattern recognition to work around the internal biases of humans and detect details in information that we would not normally see. This opens up several interesting capabilities in the fields of software development and cyber security. Specifically within the field of cyber warfare, the ability to harness a computer's processing power to work with us on complex problems can give us a variety of advantages against an adversary. We can develop advanced models for penetration testing, computer forensics, or as we did, detecting malware within a file to bolster our computer security [2].

Cyber warfare becomes more volatile by the day, and with it demands stronger security protocols to account for the developing skills of our adversaries. There were a reported 5.5 billion malware attacks in 2022, illustrating the global dangers cyber operators face [3]. Our focus was designing a tool to help mitigate the dangers of these frequent malware attacks. We created a neural network model that is trained on various benign and malicious files, then uses its training to detect and alert any potential malware in future input. Our neural network model can be applied to help individuals and organizations enhance their security system-wide, as providing this security is of utmost importance in the field of cyber security.

1.1. Limitations and Assumptions

The design of this model relies on the Feed-Forward model, which comes with its own set of limitations. Known to be the most basic neural network, it can only transmit data in one direction (forward). This limits our ability to utilize deep learning within our model, as it requires back-propagation, which the feed-forward model does not support. This limits our scope of the model to simple classification of data rather than more complex, deep learning.

The creation of any neural network model requires a training set to build the foundation of the model's knowledge. The model then uses this foundation to make decisions about future input. The larger the training data, the better equipped the model is for future tests. For this problem, we were provided with a rather small set of training data which can limit the scope and possible real-world applications compared to a model with a much larger training set.

We also had to read in specific binary files and transform the input into a specific string format so the model could understand the patterns within each file. This specific format limits the versatility of our model as it requires a specific binary input type that may not always be available. The model is also limited to reading in individual files from a specific directory format. If a user wants to scan a single binary file, it needs to be in a folder named either "malware" or benignware", inside of another folder which the user can specify. Both benignware and malware files need to be present in this user folder, but the single executable file only needs to be in one of them.

Lastly, we are also limited in the type of operating system we can use, as the malware used in our training and test sets was made to corrupt the Windows operating system. We designed and tested our model on the Linux operating system, meaning the malware should not damage Linux systems, but if we interacted with this malware on a Windows machine it could cause harm to the system.

2. Background

Machine Learning (ML) is a subset of the overarching field of artificial intelligence. ML uses complex algorithms and large data sets as input to assemble predictive models that imitate natural learning. ML is a field built on complex mathematics, but we can simplify it into a more basic form of linear algebra and probability. From a simplistic perspective, ML is designed to take in a function ‘F’, a training data set of (x, y) points, a chosen algorithm and a hypothesis set of data, and then use this data to construct a close of approximation of the function ‘F’, which we deem ‘G’. The strength of our model relies on how accurate we can be in building our function ‘G’ to predict the outcomes of function ‘F’. There are multiple avenues we can take when designing a machine learning model, most often being Supervised, Reinforcement and Unsupervised learning.

2.1. Supervised vs. Unsupervised Learning

These various types of machine learning refer to the amount of formatting we need to use when organizing the data before the model processes it. Supervised learning requires that we organize the data into categories and place corresponding labels with our data so the model can interpret the patterns within the data and make future predictions. These data sets are often made up of (x, y) pairs, where ‘x’ is the sample of data and ‘y’ is the respective label for that sample. Unsupervised learning comes without any labels or organization, as the model takes in large amounts of data to detect more general patterns. The goals of these models are slightly different, as supervised learning often tries to predict the categorization of future unseen data, while unsupervised aims to gain insight and find general patterns on the input data as a whole. While it can be time consuming to organize the data with specific labels, we can maximize the accuracy of the model with supervised strategy. On the other hand, unsupervised learning can save formatting time, but we often sacrifice accuracy and efficiency within the model itself as it becomes more computationally complex [4].

2.2. Neural Networks

Just as machine learning is a subset of artificial intelligence, neural networks are a subset of machine learning, and one of many methods in creating a machine learning model. A neural network is an interconnected series of nodes that perform mathematical computations (often in the form of vertex/matrix multiplication) to produce a predictive output. This output then serves as the input of the next layer in the network until we reach the final layer, when the output becomes the final value of the model's prediction.

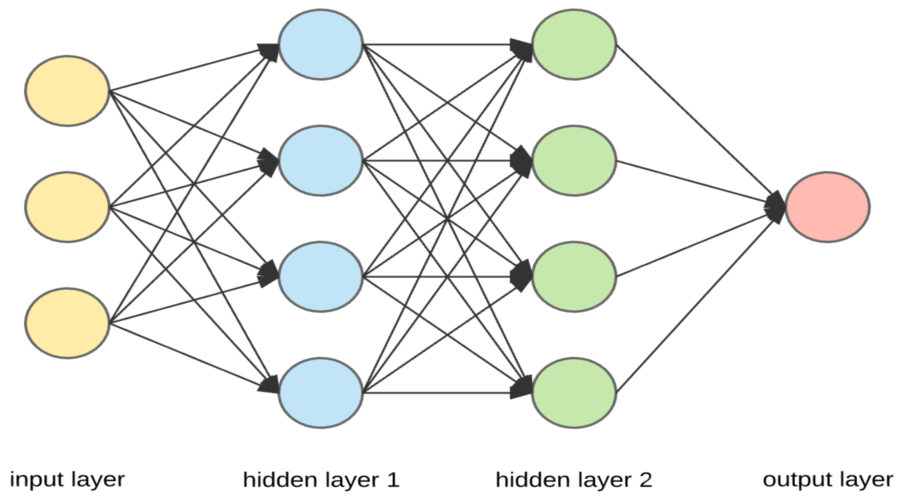


Figure 1. - Diagram of the layers of a neural network [5]

This process reflects the biological form of neural networks, as each node is interconnected with several others, forming one large network of nodes and layers. The number of layers is often customized. As we can see in Fig. 1, the basic layout consists of an original input layer, a final output layer, and the arbitrary number of hidden layers between the input and output.

2.2.1. The Feed-Forward model

Neural networks can be assembled into several different varieties, with the most basic being the Feed-Forward model. This model consists of an input layer, a customizable amount of hidden layers, and an output layer. The network transmits data in one direction, as each node takes in input, runs the input through an activation function with a certain weight and bias, then computes an output.

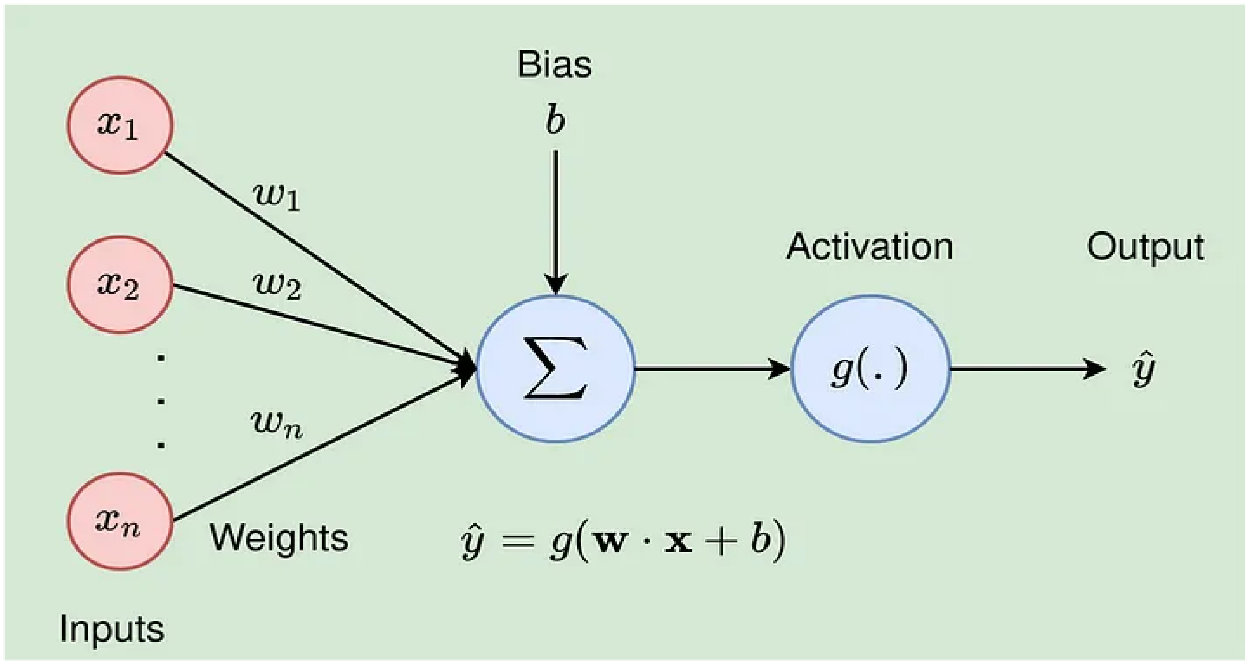


Figure 2. - Illustration of the mathematical operations within a neural network [6]

Fig. 2 displays the basic operations between each node in the feed forward model. The input nodes hold the sample data which get multiplied by their corresponding weights. These weights determine the strength of connection between any two nodes and affect the degree of importance any one node has on the output value of the model. The model continues to adjust these weights, increasing or decreasing the weights of each node as it learns to optimize the significance of any single node.

The nodes are then given a bias, representing the difference in how the model is predicting data versus what the correct output should be. These bias values serve to accelerate or delay the activation of a certain node, as it attempts to maximize the accuracy at each layer. This process is shown below in Fig. 3, using sigma notation to display the summation of each nodes' input, weight and bias.

$$x \mapsto \sum_{i=0}^{n} w_i x_i + b = w_0 x_0 + w_1 x_1 + \cdots + w_n b_n + b = \mathbf{w} \cdot \mathbf{x} + b$$

Figure 3. - Illustration of mathematical operations within each node [7].

After multiplying the input by the weights and adding the bias, the nodes form the input into the activation function of the respective layer [8]. Activation functions range from sigmoid to linear to trigonometric, but all serve similar purposes: allow the model to learn more complex patterns within a data set. If we do not apply any activation function, the number of hidden layers we include does not affect the learning capability of the model, so we prevent the model from being able to learn more complex patterns. With the activation functions applied, the neurons of each layer can begin to distinguish between patterns within the data as the number of layers and computations grow. The non-linearity of the activation functions means we can adjust for the differences in expected outcome and the model's prediction, allowing the model to learn as it progresses. The activation functions also control when each node should be triggered, comparing the weighted sum of the node to a certain threshold of activation. If the sum is above this threshold (often 0 as the neurons vary between -1 and 1), then the neuron will trigger and its output will be sent to the next node [9].

2.3. Loss and Cross-Entropy

The concept of loss guides the model as it continues to learn and make predictions, using the measurement of loss to adjust the weights across each node to determine how significant each node will be on the overall output. Loss serves as a more optimal form of adjustment compared to the binary 'right or wrong'. We can adjust the model based on how close it is to the expected output, rather than just the all-or-nothing approach of being right or wrong. The objective is to minimize the loss in a model, asserting we are as accurate compared to the expected outcome as possible. Entropy is often the most common function to measure loss, defined as the level of uncertainty in the possible outcomes of a scenario; the larger the measure of entropy (measured between zero and one), the more uncertain the outcome is [10].

We can go one step deeper in representing loss by using the cross-entropy function, extending this law of entropy to measure the distance between two probability distributions. Cross entropy uses logarithmic functions to adjust the weights applied to each node in the network, allowing us to optimize the significance of each node. Fig. 4 shows this cross-entropy function, taking the probability between two events and multiplying it by the logarithmic value of the model's prediction.

$$H(p,q) = -\sum_{x\in\text{classes}} \overset{\text{True probability distribution (one-hot)}}{p(x)} \log \underset{\text{Your model's predicted probability distribution}}{q(x)}$$

Figure 4. - sigma notation of the cross-entropy function [11]

2.4. Feature Extraction

Modern data sets in supervised learning models come with larger and larger amounts of features and labels accompanying the data. If the amount of features within the data set approaches the number of total data points, the model may suffer from overfitting, reduced efficiency and reduced accuracy on test data. To solve these issues, we use feature extraction, which aims to reduce the number of features in the data set by summarizing the existing features to create new ones, only keeping the important information needed to classify the data and getting rid of the redundant features. Feature extraction uses a hash function to compress the existing features into a smaller group, decreasing the amount of features we need to describe our data to the machine learning model. This new, smaller group of features should contain all the necessary information of the original features, but should limit overfitting, increase performance, and improve the accuracy of the model [12].

2.5. Overfitting

Overfitting describes the concept of excessive accuracy, to the point where we are not sure if the model is learning or just has become predictive of the training set. This phenomenon often occurs when we have too many features to represent our training data, and the model becomes ineffective in predicting future test sets of data. For example, if the model tests with 100% accuracy on the training data, we are not sure if the model knows what information it is processing or if it is just guessing. These hyper-accurate models often perform poorly when applied to test sets.

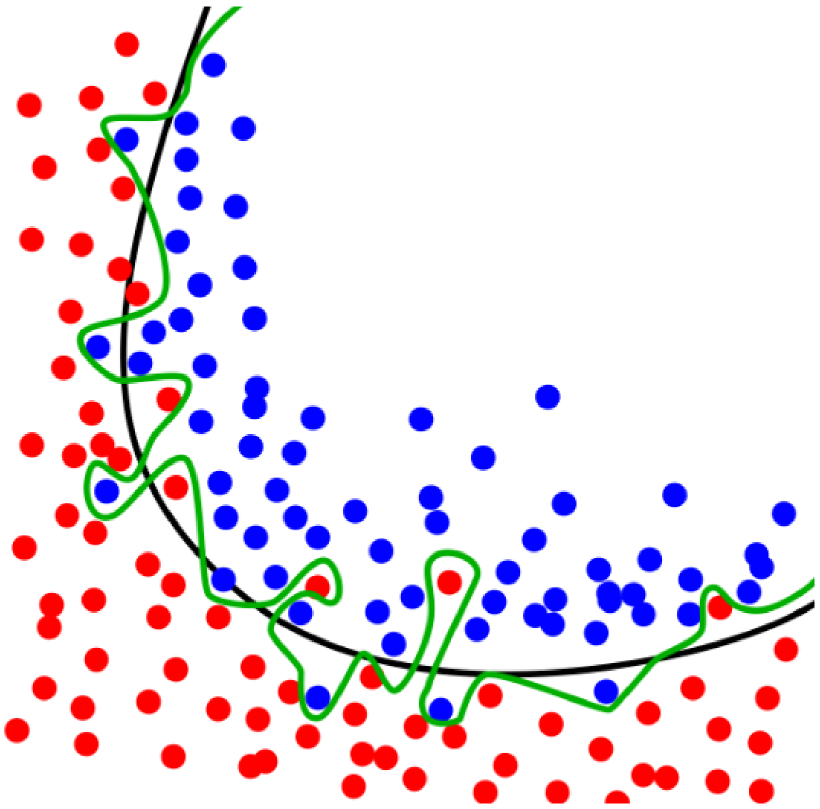


Figure 5. - Displaying the concept of overfitting to a data set [13]

As we can see in Fig. 5, the green line represents the model overfitting to the training data. At this point it is not forming a predictive model, but rather overfitting itself to the training data. The black line is more optimal, as it is learning and predicting the patterns in the data which is often much more successful on test sets.

2.6. Metrics - Accuracy, Recall, Precision

A machine learning model often has many ways to measure its effectiveness/correctness, as accuracy is not the only important metric. For example, if a model correctly identifies 100 files as malicious, that does not mean it is a perfect model. Rather, it could just be guessing the input is malware for every data point without any true learning taking place throughout the network. To account for this potential blindness, three main metrics are used to measure the correctness of a ML model: accuracy, recall and precision.

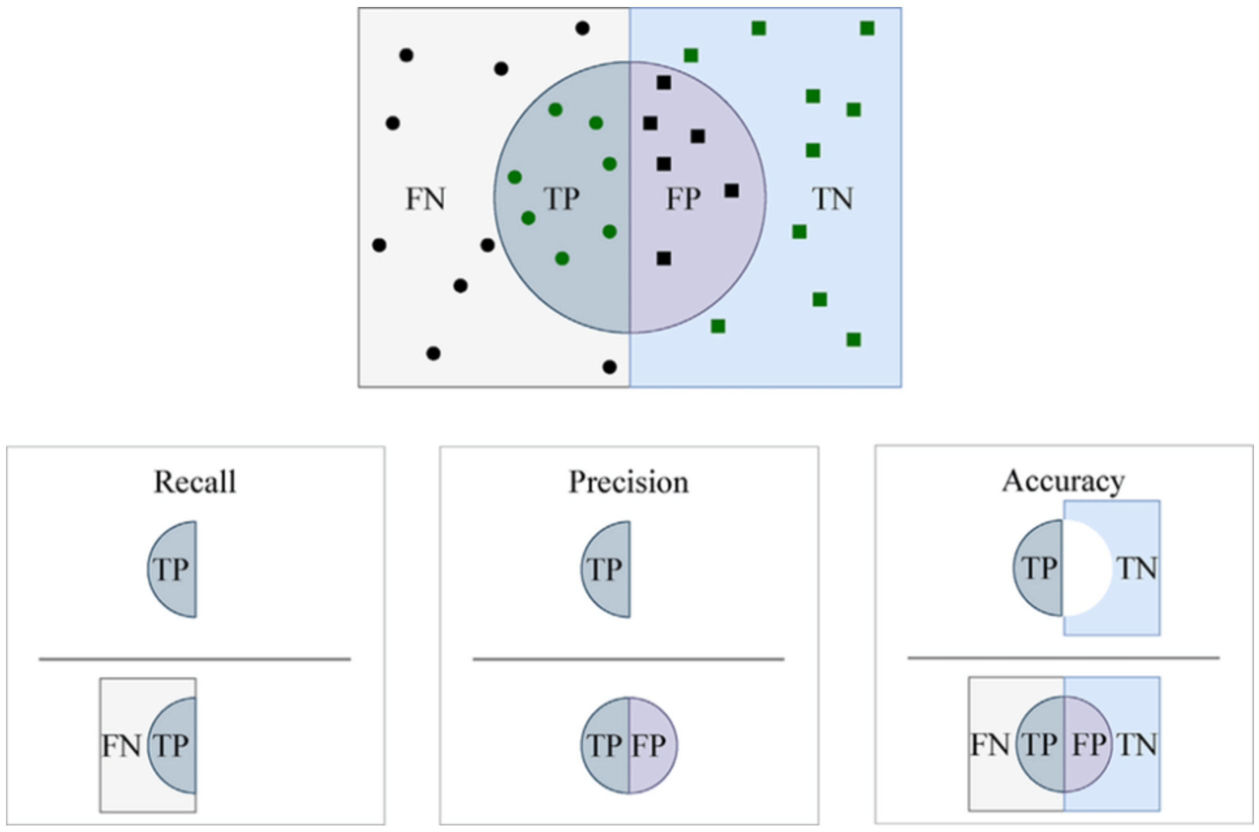


Figure 6. Representation of the three major metrics used during model compilation

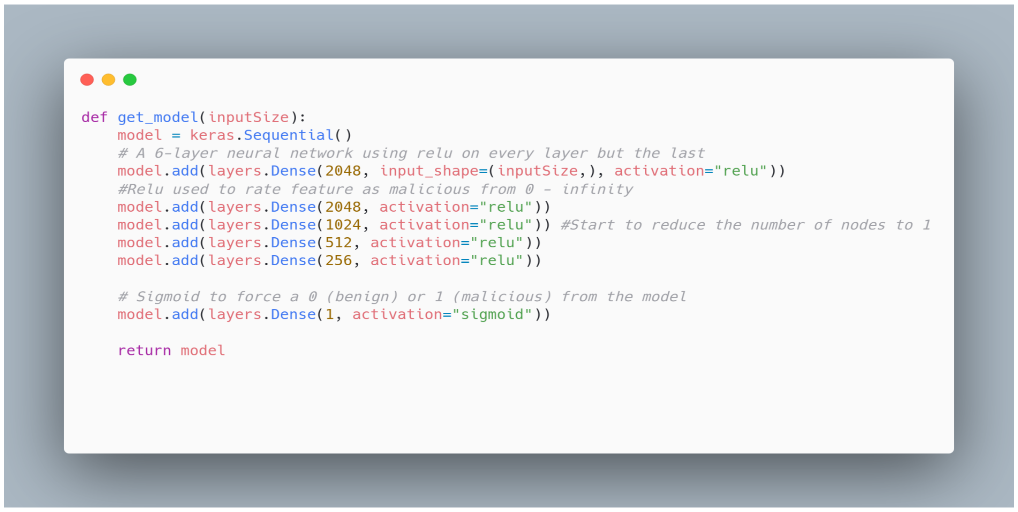
The four sections shown in Fig. 6 represent the four different results that can arise from a model making a prediction. For our model, false negative (FN) refers to the model asserting that a file was benign when it was actually malicious. True positive (TP) refers to the model correctly identifying a malware file as malicious. False positives (FP) represent that the model identified a file to be malicious when it was actually benign. Lastly, true negatives (TN) refer to the model correctly predicting a file is benign. Fig. 6 illustrates the three main metrics as ratios of these four possible outcomes. Recall is the ratio between all true positives compared to the combined number of false negatives plus true positives. Precision represents the ratio between all true positives and all combined positives. Lastly, accuracy refers to the ratio between all true predictions and total number of predictions. These three metrics form a complete measurement of a model's effectiveness.

3. Methods

The neural-network machine learning model we developed incorporates several different libraries and ML schemes to both maximize accuracy and avoid the aforementioned issues commonly seen in ML models. We constructed a feed-forward neural network with an input layer, four hidden layers and an output layer, using various activation functions, libraries and formatting styles. These six layers allow us to account for any advanced complexity, as the model has enough nodes and layers to adequately detect any patterns within the input, while also balancing the cost of computation of using too many layers. Our program consists of a main() function which then calls our other functions to format the data, create the model, train, test and evaluate the provided data sets. The main function prompts the user to either '--train' or '--test' the data, then calls the corresponding functions to satisfy the user request.

3.1. Construction of the Neural Network

Our first step in the program was creating the basic framework of the neural network. We set the number of unique features at 6400, meaning the model would evaluate 6400 features of each file to find the patterns that make the input malicious or benign. This connects back to the concepts of feature extraction and overfitting, as we wanted to eliminate any redundancy in the features the model would evaluate and take into consideration. 6400 allowed us to find the balance between redundancy and accuracy. We then called our get_model() function to create an instance of a neural network using the Keras library. This library is a common python importable that allows us to create an instance of a neural network and then customize the amount of the layers we need to account for the complexity of the input files. Keras provides us with simple implementation and built in weight vectors, meaning the design of our network becomes much easier. However, the ease of implementation comes with the cost of slower run times, as the model must perform more complex operations to abstract the computations away from the user [14].



```
def get_model(inputSize):
    model = keras.Sequential()
    # A 6-layer neural network using relu on every layer but the last
    model.add(layers.Dense(2048, input_shape=(inputSize,), activation="relu"))
    #Relu used to rate feature as malicious from 0 - infinity
    model.add(layers.Dense(2048, activation="relu"))
    model.add(layers.Dense(1024, activation="relu")) #Start to reduce the number of nodes to 1
    model.add(layers.Dense(512, activation="relu"))
    model.add(layers.Dense(256, activation="relu"))

    # Sigmoid to force a 0 (benign) or 1 (malicious) from the model
    model.add(layers.Dense(1, activation="sigmoid"))

    return model
```

Figure 7. - get_model() function, creating the basic foundation of the neural network

Fig. 7 shows the keras.Sequential() call, creating our instance of a model. We then add an input layer, four hidden layers and an output layer, constructing the groundwork of our feed-forward neural network. This feed-forward model gives us the benefits of simplicity, efficiency and speed, while also excelling in simple classification. The feed-forward model does not allow for back-propagation or deep learning, however those techniques are outside the scope of this model. This neural network should classify any input as either benign or malicious, meaning the feed-forward model is best suited for this problem given its strength of simple classification [15].

As we add each layer, we must specify the number of nodes per layer and the activation function each layer will use. We begin with 2048 nodes in the input layer and first hidden layer, meaning there are 2048 nodes in each of the first two layers of our neural network. We chose to include a large number of nodes to start with so we can ensure to include all of the necessary features of the data while training the model. As we continue to add layers, we see the number of nodes continue to be cut in half, until we reach the final output layer which uses one node. This allows the hidden layers to compress the data and outputs of each node to get a single value that represents the prediction of the model.

Along with the number of nodes for each layer, we see the activation function being specified. The first five layers use the Rectified Linear Unit (RELU) activation function while the output layer uses the sigmoid function. These activation functions allow us to achieve non-linearity from the model, meaning we can use increasing numbers of hidden layers to account for growing complexity of the input data. The functions allow the nodes to be activated once reaching a certain threshold, and can allow the model to learn patterns of more complex information as we continue to add layers to the network. We chose the RELU function for our first five layers as it is less computationally expensive than other more complex functions such as the sigmoid or tanh functions. The first five layers have much more nodes than the output layer so we wanted to choose a quicker, more simple function to balance the complexity of the input and hidden layers. The increased speed of the RELU function also allows us to make up for some of the slower computation times from using the Keras library. This function's output ranges from zero to infinity, meaning whichever positive output we get is directly proportional to the true value of the node and any negative value is represented as zero. This allows us to represent the severity of the malware as nodes with larger output values have a larger impact on the eventual output.

The last layer of our network now switches to the sigmoid function, serving as the final compression from our hidden layer nodes to one final value representing the model's prediction. Shown in Fig. 8, the sigmoid function has a value range from zero to one, meaning our output will now be compressed into a value ranging from zero to one. This sigmoid function suits the binary classification needs of our model as we can easily see if the model predicts the input to be benign (0) or malicious (1) [16].

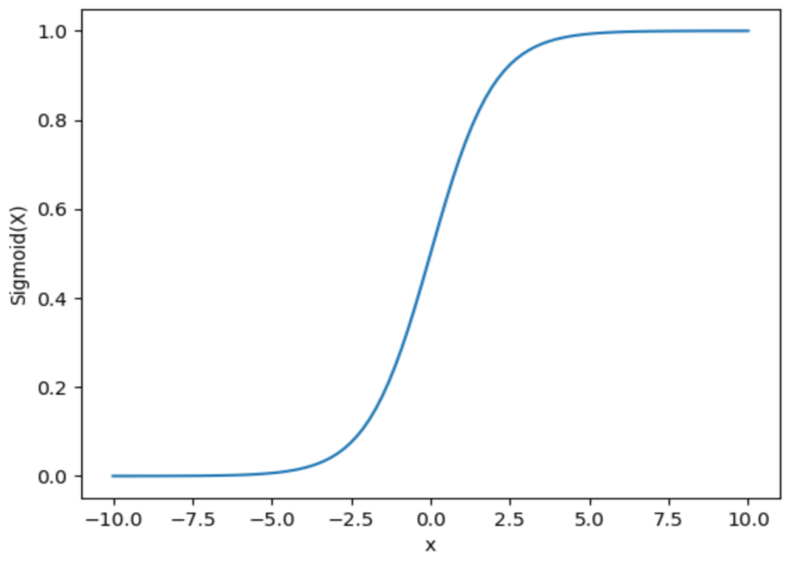
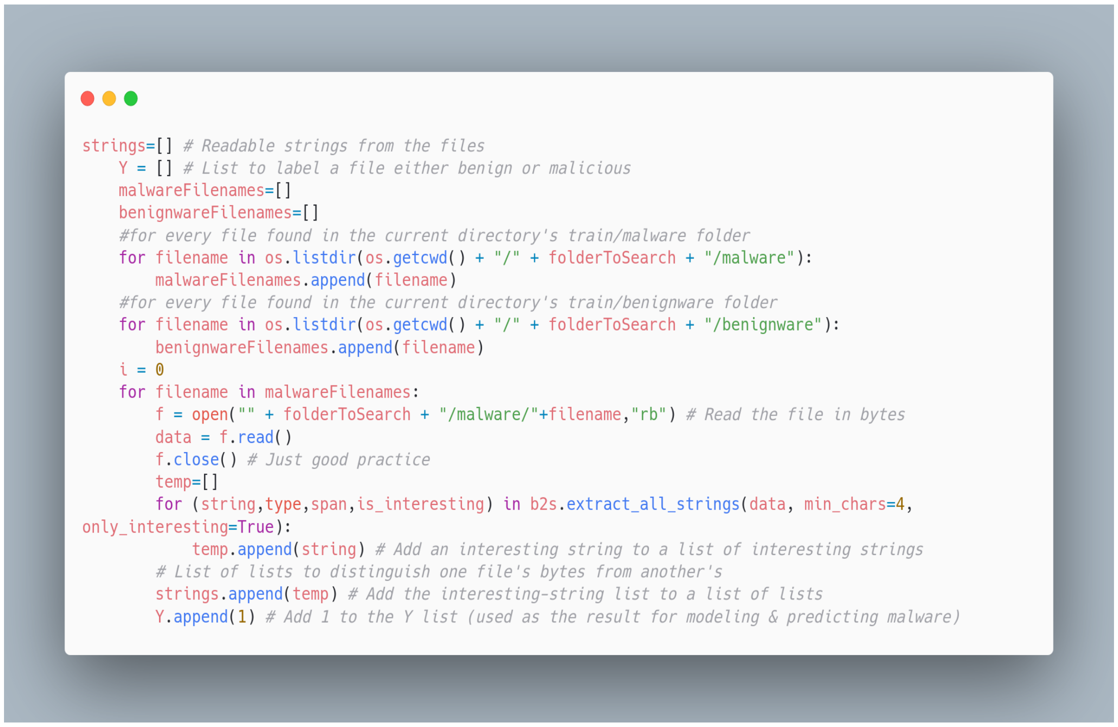


Figure 8. - Graph of the sigmoid function [17]

3.2. Training the Neural Network

Our next step in building the model was formatting the input data so the model can process it to find patterns between malicious and benign files. Our input was originally 900 binary files, 600 being benignware and 300 being malware. To allow the model to use these files, we used the 'binary2strings' (b2s) python library to translate the binary data of the input into strings. We then were able to use the b2s library to extract all of the 'interesting' strings from the files.

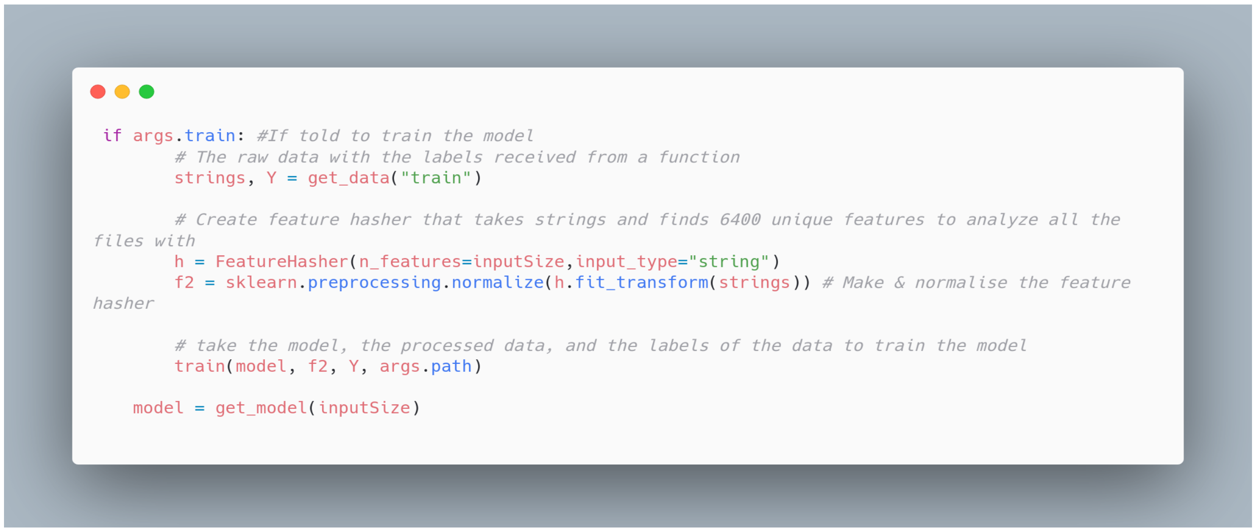


```
strings=[] # Readable strings from the files
    Y = [] # List to label a file either benign or malicious
    malwareFilenames=[]
    benignwareFilenames=[]
    #for every file found in the current directory's train/malware folder
    for filename in os.listdir(os.getcwd() + "/" + folderToSearch + "/malware"):
        malwareFilenames.append(filename)
    #for every file found in the current directory's train/benignware folder
    for filename in os.listdir(os.getcwd() + "/" + folderToSearch + "/benignware"):
        benignwareFilenames.append(filename)
    i = 0
    for filename in malwareFilenames:
        f = open("" + folderToSearch + "/malware/"+filename,"rb") # Read the file in bytes
        data = f.read()
        f.close() # Just good practice
        temp=[]
        for (string,type,span,is_interesting) in b2s.extract_all_strings(data, min_chars=4,
only_interesting=True):
            temp.append(string) # Add an interesting string to a list of interesting strings
        # List of lists to distinguish one file's bytes from another's
        strings.append(temp) # Add the interesting-string list to a list of lists
        Y.append(1) # Add 1 to the Y list (used as the result for modeling & predicting malware)
```

Figure 9. - Code displaying how to format the input files into readable strings

The code shown in Fig. 9 illustrates how we read in the input from the provided benignware and malware files respectively. We passed the variable ‘folderToSearch’ into the function from the function call in main() allowing the program to read in all of the necessary data to train and/or test the model. We then iterated over each file for both the benign and malicious lists to find the interesting strings in each file. The b2s library includes this functionality as a form of feature extraction, as it removes all unimportant pieces of data to avoid overfitting the data set. This increases our accuracy while decreasing the model’s training and testing time, ensuring we can apply our model to future test sets. We then append each interesting string to a temporary list, which is then also appended to our ‘strings’ list of lists. This process of appending lists of interesting strings is applied for both the benignware and malware files. The function then returns this ‘strings’ list along with our list ‘Y’, which holds the value one for every malicious file and a zero for every benign file we process. This model is a form of supervised machine learning, meaning the data is assigned a label so the model can categorize the data during both training and testing. This 'Y' list serves as the label for our data 'X', as we know the training data is already separated into malware and benignware. The values are then compared to the model's prediction during training and used to adjust the weights of the neural network.

After creating the foundations of our neural network and organizing our training data into the correct format, we then use the Feature Extraction and sklearn libraries to create a Feature Hasher we can use to help train our model.

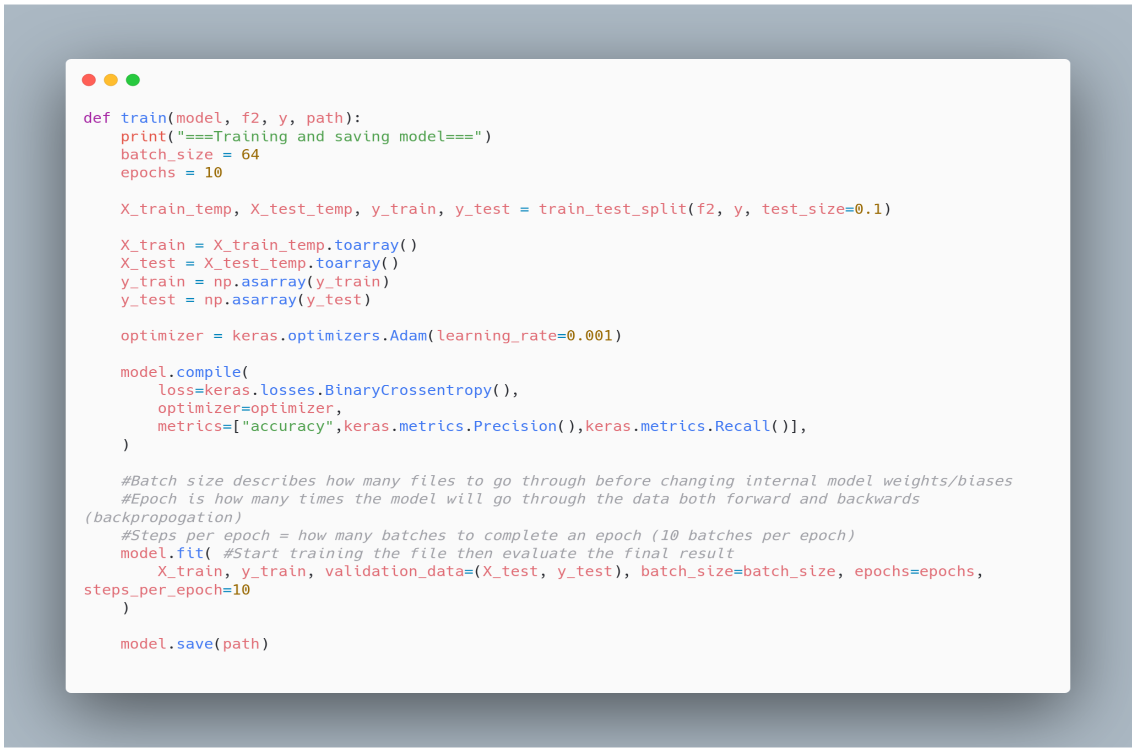


```
 if args.train: #If told to train the model
        # The raw data with the labels received from a function
        strings, Y = get_data("train")

        # Create feature hasher that takes strings and finds 6400 unique features to analyze all the
files with
        h = FeatureHasher(n_features=inputSize,input_type="string")
        f2 = sklearn.preprocessing.normalize(h.fit_transform(strings)) # Make & normalise the feature
hasher

        # take the model, the processed data, and the labels of the data to train the model
        train(model, f2, Y, args.path)

    model = get_model(inputSize)
```

Figure 10. - Code displaying the creation of a feature hasher and the call to train()

As shown in Fig. 10, we use the FeatureHasher library to create an instance of a hasher that can take in strings and creates a specified number of features. These libraries, similar to Keras, allow for simple implementation that can abstract the complex mathematics out of the program. This allows us to prioritize the organization and correct implementation of our model. We use the inputSize of 6400 to create as many essential features as possible to effectively classify the data while also balancing efficiency and avoiding redundancy. This hasher 'h' is then used as an argument when we create an instance of the sklearn library. This sklearn library has built in functions we can use for the classifications used to detect if a file is malicious or not. We use the transform() function on our ‘strings’ list to format our input files, allowing the sklearn library to normalize the data into a final feature hasher. This hasher ‘f2’ is then passed into our train() function along with our model, list ‘Y’ containing our comparative values, and the path of the model we use to load the most current model.

After calling this train() function from main, we now split our input data into training and testing sets, using the returned training points to compile and fit the model as it learns the patterns between malware and benignware.



```python
def train(model, f2, y, path):
    print("===Training and saving model===")
    batch_size = 64
    epochs = 10

    X_train_temp, X_test_temp, y_train, y_test = train_test_split(f2, y, test_size=0.1)

    X_train = X_train_temp.toarray()
    X_test = X_test_temp.toarray()
    y_train = np.asarray(y_train)
    y_test = np.asarray(y_test)

    optimizer = keras.optimizers.Adam(learning_rate=0.001)

    model.compile(
        loss=keras.losses.BinaryCrossentropy(),
        optimizer=optimizer,
        metrics=["accuracy",keras.metrics.Precision(),keras.metrics.Recall()],
    )

    #Batch size describes how many files to go through before changing internal model weights/biases
    #Epoch is how many times the model will go through the data both forward and backwards
(backpropogation)
    #Steps per epoch = how many batches to complete an epoch (10 batches per epoch)
    model.fit( #Start training the file then evaluate the final result
        X_train, y_train, validation_data=(X_test, y_test), batch_size=batch_size, epochs=epochs,
steps_per_epoch=10
    )

    model.save(path)
```

Figure 11. - Code displaying our function to train the neural network model

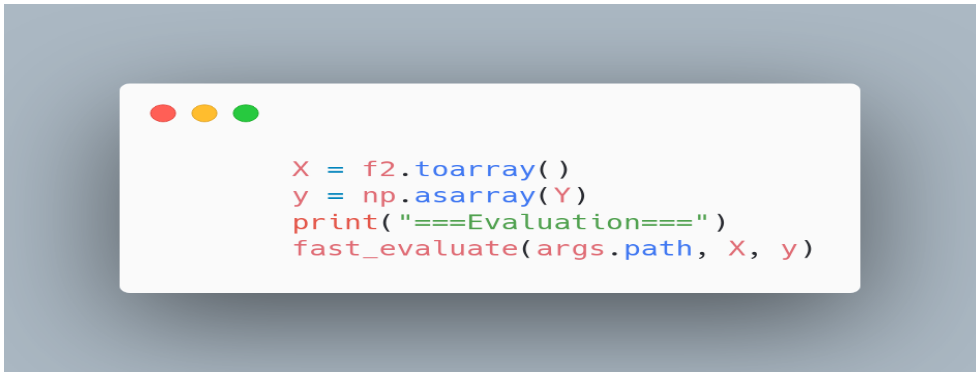
Fig. 11 demonstrates the call to the train_test_split function which we imported from the sklearn library. We pass our f2 hasher, the expected outcome list ‘y’ and the test size of 0.1 into the function, meaning 10% of our data will be reserved for testing the model as we wanted to maximize the amount of training for our model while preserving some provided data to test. This imported function returns four sets, a set of X and Y points for training, and a set of X and Y points for testing. We then use the toarray() functions to transform the returned X sets and the NumPy library function asarray() to transform the returned Y sets into usable arrays we can compile and input into the model. This NumPy (np) python library is imported to simplify the complex linear algebra functions we use and can improve the model's efficiency in terms of memory space and runtime speed compared to more standard Python data structures. After formatting our input arrays, we then used the inherent 'compile' and 'fit' functions in the Keras library to train the neural network. We first set an 'optimizer' variable using the Keras library with a learning rate of .001. This learning rate determines how fast the neural network changes and how severe the changes should be in response to any loss discovered during training. The value of .001 is recommended with the Keras library, as a value too small could lead to slower learning and one too large could lead to poor performance with respect to accuracy and the weights of each node [18].

After setting our optimizer variable, we then compile the model with the compile() function in Keras. Using this compile function allows us to prepare the model for training, as we can set the desired loss, optimization and metric settings for our model. We use the cross-entropy function to guide the loss of the model, which enables us to compare the output of the model's prediction with the expected output of the function given in our list Y. We also set our desired metrics by explicitly declaring the accuracy, precision and recall of our model. We set these three metrics to provide the most effective measurement of our model's correctness, as each of the metrics represent a different ratio of outcomes the model can make. Using all three allows us to consider all possible scenarios of our neural network to make a more applicable model for future test sets.

After the compilation of the model, the last step is to train the neural network using the model.fit() command. This fit() function again comes from the Keras library and allows us to pass our X and Y training sets, our batch size and our epochs to train the neural network. The batch size refers to how many files/strings of input data the model should process before updating the weights and biases of each node. Epochs refer to an iteration of the training data going through the neural network. As shown in Fig. 11 we set the batch size to 64, a moderate number of data sets to balance the tradeoff between performance and appropriate weights/biases, and the epochs to 10. We experimented with several different amounts of epoch iterations with 10 serving as the best balance between accuracy and performance. Fewer epochs led to a decrease in overall correctness and more epochs led to worsening performance and speed along with a plateau in the model's metrics.

3.3. Testing the Neural Network

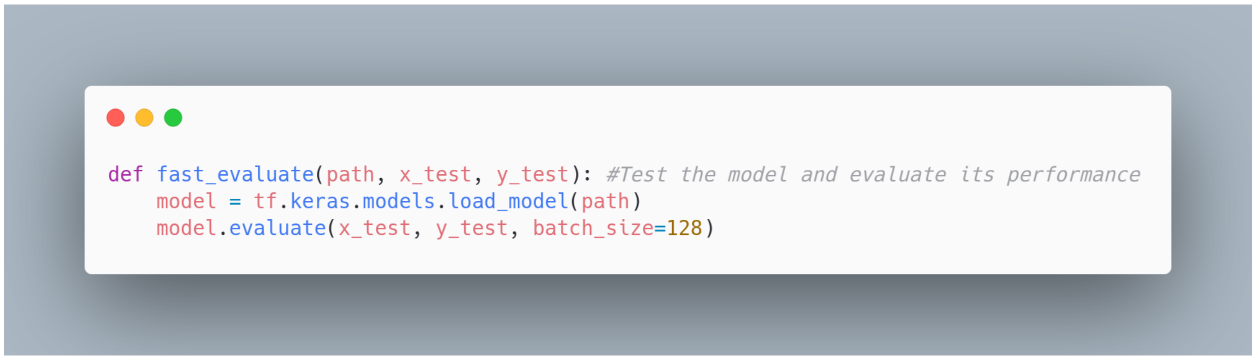
Now that our model is trained using the provided input files, we can now test the neural network with unseen data to verify its effectiveness. Similar to the process depicted in Fig. 10, our main function has the statement 'if args.test:', which executes if we add the '--test' extension on our command during program execution. The process of creating our model, formatting our data and creating our feature hashers are the same as the training sequence. We then replace the train_test_split() function with the commands below.



```
X = f2.toarray()
y = np.asarray(Y)
print("===Evaluation===")
fast_evaluate(args.path, X, y)
```

Figure 12. Transform of data for testing of the neural network.

We use the hasher and the np.asarray() functions exclusively to format the data set used for our testing. We then call the fast_evaluate() function to test the neural network.

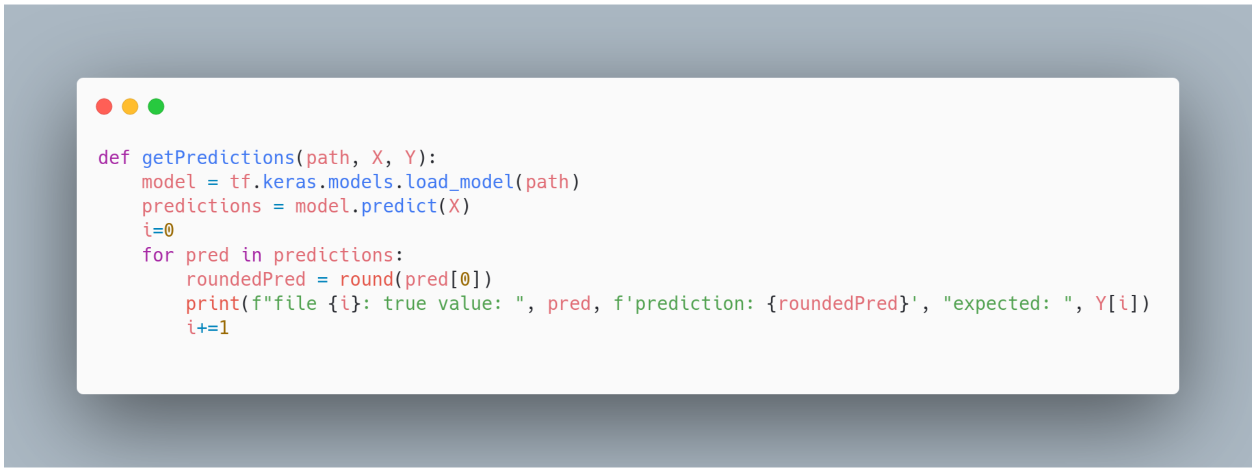


```
def fast_evaluate(path, x_test, y_test): #Test the model and evaluate its performance
    model = tf.keras.models.load_model(path)
    model.evaluate(x_test, y_test, batch_size=128)
```

Figure 13. - The function used to test the neural network

Fig. 13 shows the simple implementation of this fast_evaluate() function, as it is passed the path of the model a;ong with the test data and labels of each sample. The function first loads the most current model. We then call the evaluate() function from Keras, passing the test sets derived from splitting our input data, and our increased batch size of 128. We doubled the batch size for testing as the model has already been trained, so the need to adjust the weights and biases are less of a concern. This also allows for much faster testing while preserving the functionality of updating the weights as needed. We used 12 epochs for our testing parameters rather than 10. This small increase was able to increase performance while preserving the speed of the neural network. The program then displays the final metrics of the model after its test run to verify the results of its loss, recall, precision and accuracy.

The last step in testing our model was printing out the model's results for each input file to compare with the expected value from our original list Y. Using the predict() function from Keras with our model, we can return a list of predictions for each file ranging from zero (benign) to one (malicious). We then print this raw prediction, a rounded version of this prediction, and the expected outcome from the Y list to see how correct the model was for each file. After returning from our fast_evaluate() function shown in Fig. 13, we call our getPredictions() function.



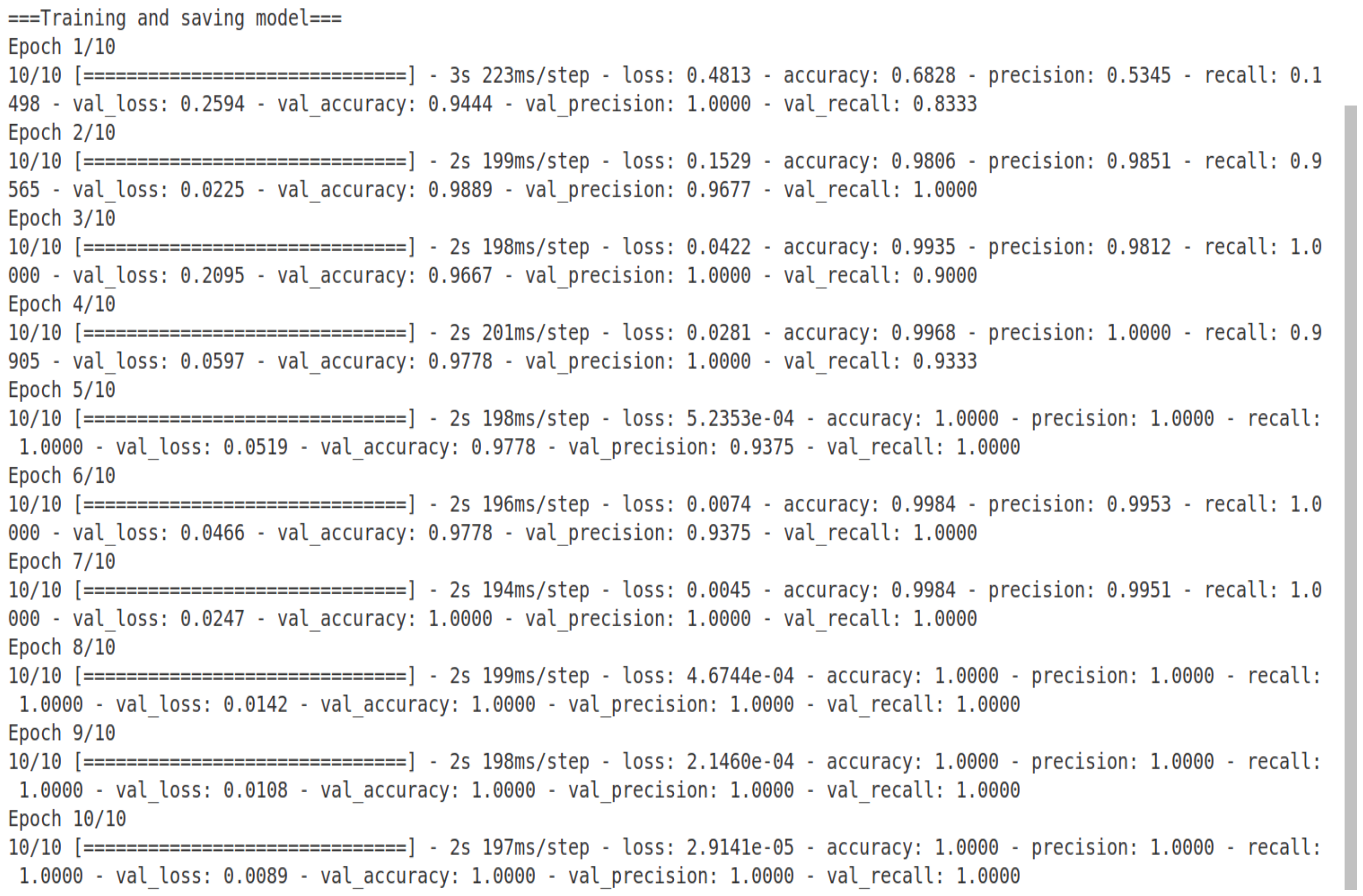
```python
def getPredictions(path, X, Y):
    model = tf.keras.models.load_model(path)
    predictions = model.predict(X)
    i=0
    for pred in predictions:
        roundedPred = round(pred[0])
        print(f"file {i}: true value: ", pred, f'prediction: {roundedPred}', "expected: ", Y[i])
        i+=1
```

Figure 14. - Function to print the model's predictions of each file

We again pass the path to load the instance of an existing model, then get the list of predictions with the .predict() call, passing the sample data X as a parameter. We then print out each prediction along with its rounded value and corresponding expected value. We then see the overall performance metrics of the model along with each individual prediction of each input file, verifying the effectiveness of the neural network.

4. Results

We designed a neural-network machine learning model to detect malware in a given input file with 98% accuracy, 97% precision and 98% recall, while keeping loss below .1%. We used functions from various common python libraries, such as Keras and NumPy to simplify our implementation and reduce the model's complexity while maximizing accuracy, recall and precision. Our neural network model is able to read in a single executable file or a folder of files, scan the files, and then predict if each file will be malicious or benign. We were able to use the various libraries and techniques to first train the model, then use it for testing on future data sets. The results from this training are shown below in Fig. 15

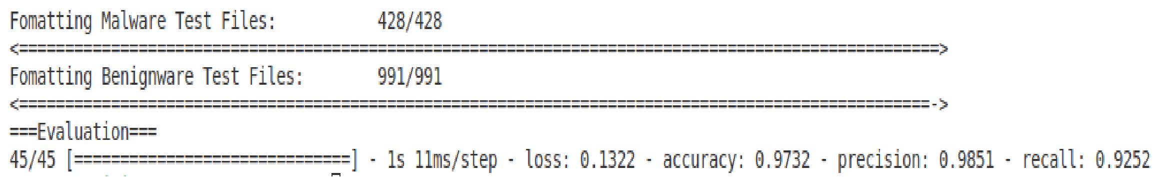


```
===Training and saving model===
Epoch 1/10
10/10 [==============================] - 3s 223ms/step - loss: 0.4813 - accuracy: 0.6828 - precision: 0.5345 - recall: 0.1
498 - val_loss: 0.2594 - val_accuracy: 0.9444 - val_precision: 1.0000 - val_recall: 0.8333
Epoch 2/10
10/10 [==============================] - 2s 199ms/step - loss: 0.1529 - accuracy: 0.9806 - precision: 0.9851 - recall: 0.9
565 - val_loss: 0.0225 - val_accuracy: 0.9889 - val_precision: 0.9677 - val_recall: 1.0000
Epoch 3/10
10/10 [==============================] - 2s 198ms/step - loss: 0.0422 - accuracy: 0.9935 - precision: 0.9812 - recall: 1.0
000 - val_loss: 0.2095 - val_accuracy: 0.9667 - val_precision: 1.0000 - val_recall: 0.9000
Epoch 4/10
10/10 [==============================] - 2s 201ms/step - loss: 0.0281 - accuracy: 0.9968 - precision: 1.0000 - recall: 0.9
905 - val_loss: 0.0597 - val_accuracy: 0.9778 - val_precision: 1.0000 - val_recall: 0.9333
Epoch 5/10
10/10 [==============================] - 2s 198ms/step - loss: 5.2353e-04 - accuracy: 1.0000 - precision: 1.0000 - recall:
 1.0000 - val_loss: 0.0519 - val_accuracy: 0.9778 - val_precision: 0.9375 - val_recall: 1.0000
Epoch 6/10
10/10 [==============================] - 2s 196ms/step - loss: 0.0074 - accuracy: 0.9984 - precision: 0.9953 - recall: 1.0
000 - val_loss: 0.0466 - val_accuracy: 0.9778 - val_precision: 0.9375 - val_recall: 1.0000
Epoch 7/10
10/10 [==============================] - 2s 194ms/step - loss: 0.0045 - accuracy: 0.9984 - precision: 0.9951 - recall: 1.0
000 - val_loss: 0.0247 - val_accuracy: 1.0000 - val_precision: 1.0000 - val_recall: 1.0000
Epoch 8/10
10/10 [==============================] - 2s 199ms/step - loss: 4.6744e-04 - accuracy: 1.0000 - precision: 1.0000 - recall:
 1.0000 - val_loss: 0.0142 - val_accuracy: 1.0000 - val_precision: 1.0000 - val_recall: 1.0000
Epoch 9/10
10/10 [==============================] - 2s 198ms/step - loss: 2.1460e-04 - accuracy: 1.0000 - precision: 1.0000 - recall:
 1.0000 - val_loss: 0.0108 - val_accuracy: 1.0000 - val_precision: 1.0000 - val_recall: 1.0000
Epoch 10/10
10/10 [==============================] - 2s 197ms/step - loss: 2.9141e-05 - accuracy: 1.0000 - precision: 1.0000 - recall:
 1.0000 - val_loss: 0.0089 - val_accuracy: 1.0000 - val_precision: 1.0000 - val_recall: 1.0000
```

Figure 15. Results of training the model on the provided training files

We see the first epoch produce sub-par results as the loss is near 50%, accuracy is around 68%, precision is 53% and recall is 15%. However, as we move through each iteration of the data and the model continues to learn, the four metrics improve to a much more appropriate standard. The next three epochs demonstrate increases in accuracy, precision and recall while also showing improvements in loss. On the 10th epoch, loss is about .002%, while accuracy, precision and recall are all 100%. While this does hint at overfitting, the gradual improvement over each epoch does demonstrate the model going through a learning process.

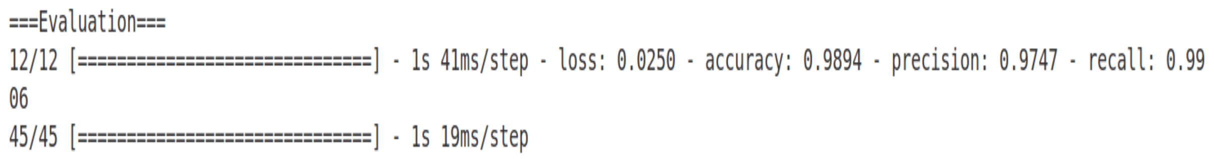
After training the model we then tested it with an unseen set of data. This new data contained 428 malware files and 991 benignware files. The first iteration of testing with this set produced the following results in Fig. 16.



```
Fomatting Malware Test Files:          428/428
<==========================================================================================>
Fomatting Benignware Test Files:       991/991
<=========================================================================================->
===Evaluation===
45/45 [==============================] - 1s 11ms/step - loss: 0.1322 - accuracy: 0.9732 - precision: 0.9851 - recall: 0.9252
```

Figure 16. Results from first test of the model on unseen data

These results depict adequate levels of all four metrics while also highlighting the lack of overfitting within the model as it was able to learn from the training set and apply this learning to predict an unseen set of files. As we continued to run this same set of data through the model, the metrics only improved as the model became more and more familiar with the patterns among the malware and benignware. The 5th iteration of running this test set through the model produced the following results shown in Fig. 17.

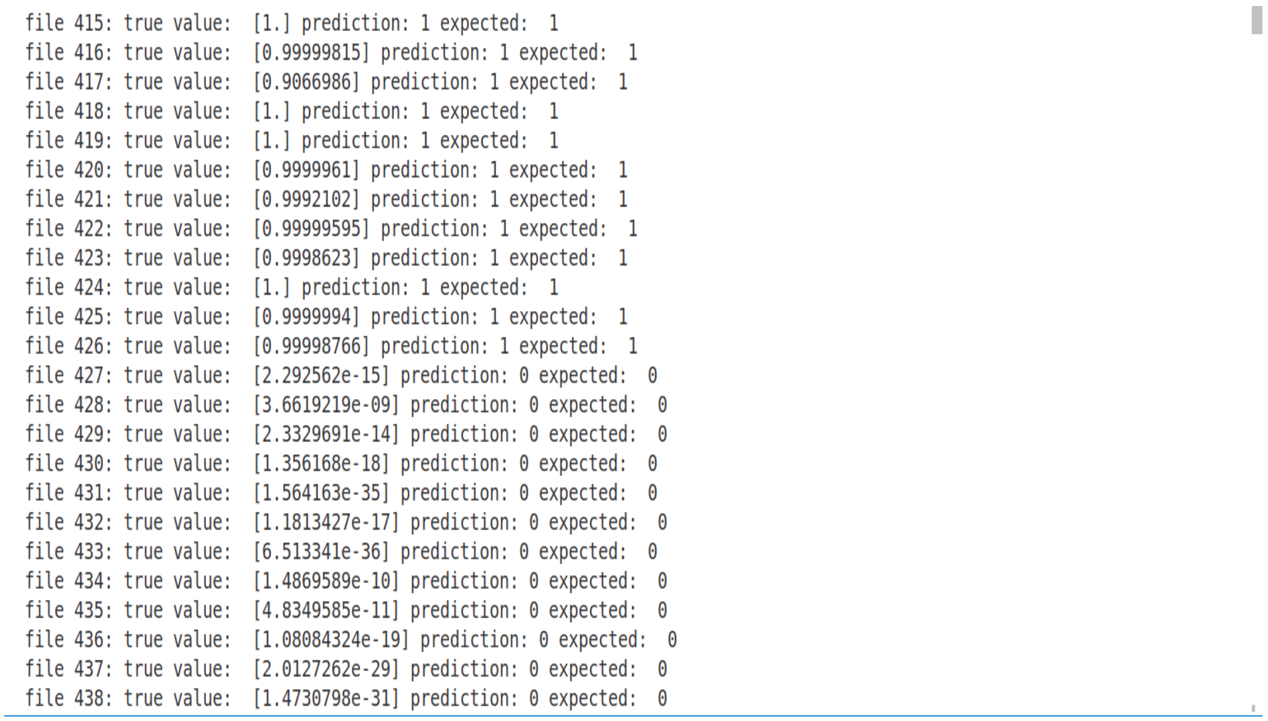


```
===Evaluation===
12/12 [==============================] - 1s 41ms/step - loss: 0.0250 - accuracy: 0.9894 - precision: 0.9747 - recall: 0.99
06
45/45 [==============================] - 1s 19ms/step
```

Figure 17. Results from 5th run of test data

As observed in Fig. 17, we can see improvements in three of the four major metrics, with the most significant improvements coming in the loss and recall metrics. This demonstrates our model continuing to learn the patterns within each file as it continues to process the same data, while also preserving its effectiveness on unseen data as shown in Fig. 16.

The model then displays the results of each individual prediction made for each file, along with its corresponding rounded value (rounded to a zero or one for simple analysis) and the expected value of each file. These results are shown below in Fig. 18.

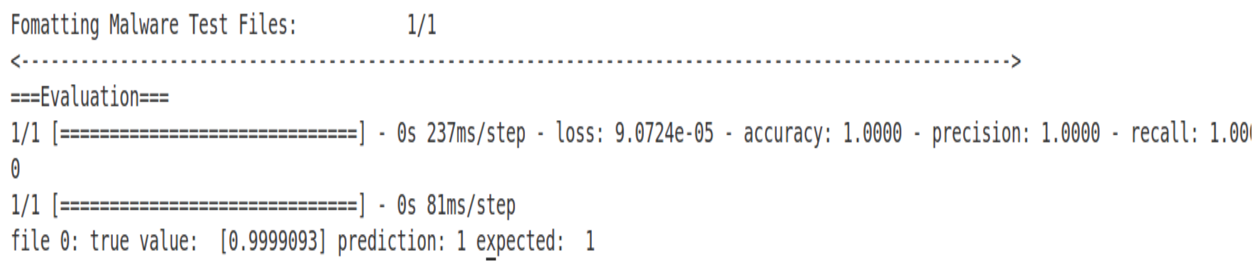


```
file 415: true value:  [1.] prediction: 1 expected:  1
file 416: true value:  [0.99999815] prediction: 1 expected:  1
file 417: true value:  [0.9066986] prediction: 1 expected:  1
file 418: true value:  [1.] prediction: 1 expected:  1
file 419: true value:  [1.] prediction: 1 expected:  1
file 420: true value:  [0.9999961] prediction: 1 expected:  1
file 421: true value:  [0.9992102] prediction: 1 expected:  1
file 422: true value:  [0.99999595] prediction: 1 expected:  1
file 423: true value:  [0.9998623] prediction: 1 expected:  1
file 424: true value:  [1.] prediction: 1 expected:  1
file 425: true value:  [0.9999994] prediction: 1 expected:  1
file 426: true value:  [0.99998766] prediction: 1 expected:  1
file 427: true value:  [2.292562e-15] prediction: 0 expected:  0
file 428: true value:  [3.6619219e-09] prediction: 0 expected:  0
file 429: true value:  [2.3329691e-14] prediction: 0 expected:  0
file 430: true value:  [1.356168e-18] prediction: 0 expected:  0
file 431: true value:  [1.564163e-35] prediction: 0 expected:  0
file 432: true value:  [1.1813427e-17] prediction: 0 expected:  0
file 433: true value:  [6.513341e-36] prediction: 0 expected:  0
file 434: true value:  [1.4869589e-10] prediction: 0 expected:  0
file 435: true value:  [4.8349585e-11] prediction: 0 expected:  0
file 436: true value:  [1.08084324e-19] prediction: 0 expected:  0
file 437: true value:  [2.0127262e-29] prediction: 0 expected:  0
file 438: true value:  [1.4730798e-31] prediction: 0 expected:  0
```

Figure 18. Results of the model's prediction of each individual file (files 415-438)

The model also allows for analysis of a single file as long as it is passed into the model in the correct directory format. The results of evaluating a single executable file are shown below in

Fig. 19



```
Fomatting Malware Test Files:         1/1
<------------------------------------------------------------------------------------------>
===Evaluation===
1/1 [==============================] - 0s 237ms/step - loss: 9.0724e-05 - accuracy: 1.0000 - precision: 1.0000 - recall: 1.00
0
1/1 [==============================] - 0s 81ms/step
file 0: true value:  [0.9999093] prediction: 1 expected:  1
```

Figure 19. Results of testing a single executable file

We can see the changes in metrics, as there is very little loss while having 100% accuracy, precision and recall because there is only one file to examine, reducing the chances of having an incorrect prediction.

While the neural network model performs well in the four major metrics, it does come with certain drawbacks. Our model is built on the foundation of several complex libraries and integrates them in a way that can further increase the complexity of the program. Each of the main libraries used, such as Keras and NumPy, come with their own individual drawbacks that then extend into the program. Keras can often cause decreases in speed and performance metrics while also having poor error handling, meaning it can become difficult to debug any arising errors in the model [19]. Furthermore, the NumPy library requires contiguous space in memory to operate and store its necessary arrays and matrices, causing further performance issues within the program in terms of speed. NumPy can also be difficult for the user as it produces variables and values that are often not compatible with the various python interpreters. Despite these drawbacks of using these more complex libraries, the benefits can provide us with higher levels of abstraction and much simpler program implementation. These libraries also excel in handling complex mathematical operations with respect to linear algebra, activation functions, and data analysis. NumPy specifically also has unique data structures that can improve our run time speed and memory space conservation [20].

One of the main concerns during implementation was balancing the classic tradeoffs seen in software engineering: cost versus performance, speed versus computational prowess, simplicity versus accuracy. Considering these tradeoffs at every decision allowed us to form a balanced machine learning model in terms of speed, performance, accuracy, simplicity and complexity. Through the use of the aforementioned libraries, we gain a degree of speed and simplicity that we sacrificed when adding several hidden layers and choosing a highly specific format of input files. We then gain a degree of computational complexity and performance back with our sigmoid activation function in our output layer, along with creating dense layers within our neural network. Together, the design choices of our program form a balanced and effective neural network that can detect malware with high performance in loss, accuracy, precision and recall.

5. Discussion

The neural network model serves as a functioning, accurate method in building a detection system for malware within any executable file. The neural network achieved roughly 98% in accuracy, precision and recall while limiting loss to below .1% on average. This model formed the foundations of a successful malware detection system, and despite having some various constrictions, the model has plenty of room for future growth.

5.1. Bounds and Limitations

The neural network model is bounded by the necessary format of the inputs. The model is based around supervised machine learning which requires labels to categorize the data so the model can understand and process the patterns within each category. The model also hinges on both the correct format of the input files themselves and the correct format of the directories these files are placed in. Due to the implementation of reading in the data from the files, the model requires that the input files are binary executables and they are in folders titled "benignware" and "malware", as these labels serve as the foundation of the supervised learning principle. Without this specific format, the model would not be able to read in the data and would not function properly.

This model is further bounded by the format as it limits the operating system (OS) we can work with. The training and testing data is considered to be malicious to the Windows OS. This bounds us to Linux and Mac systems as we cannot let the program open the malware on Windows machines, potentially corrupting the system. We can adjust the model to accept different types of malware, but the act of analyzing any malware through a neural network will always open up the possibility of harming a certain system.

5.2. Impact

The design of this model serves as a modern approach to malware detection, as the utilization of neural networks for this purpose has not been deeply studied yet. This gives us an interesting opportunity to delve into a newer field of neural network application. Given the modern nature of the problem, this solution can serve as the foundation for a new and evolving study. The fields of cyber and software security can use this model's framework to construct future models that can be applied to a wider variety of data sets. The preliminary success of this model represents the impact that neural networks can have on the study of cyber security and malware detection.

5.3. Future Work

The future of this model's development should be based on adapting the model to accept a wider variety of input, such as images and text files. As adversaries continue to evolve in their malicious techniques, we must respond in our techniques of detection and protection. Extending this malware detection to different types of input files would allow us to protect wider varieties of systems, files, and would form a more applicable model for real world use cases.

Secondly, the future of this project should be aimed at accepting a less strict file format. Currently, if a user wants to scan a single file for malware they must place two files named "benignware" and "malware" in another file, and then place their desired file in one of the two software files. This creates an obvious challenge for the user and can be a frustrating source of confusion. Taking the stress of directory organization away from the user and having a more adaptable model should be one of the more urgent priorities.

6. Works Cited

[1] K. D. Foote, "A Brief History of Machine Learning - DATAVERSITY," *DATAVERSITY*, Mar. 26, 2019. https://www.dataversity.net/a-brief-history-of-machine-learning/

[2] G. Gottsegen, "Machine Learning Cybersecurity: How It Works and Companies to Know," *Built in*, Jun. 30, 2019. https://builtin.com/artificial-intelligence/machine-learning-cybersecurity

[3] "Number of malware attacks per year 2018," *Statista*. https://www.statista.com/statistics/873097/malware-attacks-per-year-worldwide/

[4] J. Delua, "Supervised vs. unsupervised learning: What's the difference?," *IBM*, Mar. 12, 2021. https://www.ibm.com/cloud/blog/supervised-vs-unsupervised-learning

[5] D. Park, *Neural Network Diagram*. 2023.

[6] TowardsDataScience, 2017. Available: https://miro.medium.com/v2/resize:fit:720/format:webp/1*qQPpdtR0r1APiEfTqN74aA.png

[7] TowardsDataScience, 2017. Available: https://miro.medium.com/v2/resize:fit:720/format:webp/1*hIsDGGwg3dRx9ZlkmMAeDw.png

[8] DeepAI, "Weight (artificial neural network)," *DeepAI*, May 17, 2019. https://deepai.org/machine-learning-glossary-and-terms/weight-artificial-neural-network

[9] S. Tiwari, "Activation functions in Neural Networks - GeeksforGeeks," *GeeksforGeeks*, Feb. 06, 2018. https://www.geeksforgeeks.org/activation-functions-neural-networks/

[10] K. E. Koech, "Cross-Entropy Loss Function," *Medium*, Feb. 25, 2021. https://towardsdatascience.com/cross-entropy-loss-function-f38c4ec8643e

[11] Stack Overflow, Ed., *What is Cross-Entropy?* 2021. Accessed: Jul. 01, 2023. [Online]. Available: https://www.google.com/imgres?imgurl=https%3A%2F%2Fi.stack.imgur.com%2FgNip2.png&tbnid=EvY6xhP4_fjHMM&vet=12ahUKEwiv8Yjf8-3_AhUGF2IAHSGgDY8QMygFegUIARDbAQ..i&imgrefurl=https%3A%2F%2Fstackoverflow.com%2Fquestions%2F41990250%2Fwhat-is-cross-entropy&docid=mtdj-XG_g1QKxM&w=500&h=208&q=cross%20entropy%20in%20machine%20learning&client=ubuntu-sn&ved=2ahUKEwiv8Yjf8-3_AhUGF2IAHSGgDY8QMygFegUIARDbAQ%20https://towardsdatascience.com/feature-extraction-techniques-d619b56e31be

[12] P. P. Ippolito, "Feature Extraction Techniques," *Medium*, Oct. 11, 2019. https://towardsdatascience.com/feature-extraction-techniques-d619b56e31be

[13] D. Park, *Overfitting of Data*. 2023.

[14] "Python Keras Advantages and Limitations," *DataFlair*, Apr. 22, 2020. https://data-flair.training/blogs/python-keras-advantages-and-limitations/

[15] G. L. Team, "Types of Neural Networks and Definition of Neural Network," *GreatLearning*, Apr. 29, 2020. https://www.mygreatlearning.com/blog/types-of-neural-networks/

[16] S. Tiwari, "Activation functions in Neural Networks - GeeksforGeeks," *GeeksforGeeks*, Feb. 06, 2018. https://www.geeksforgeeks.org/activation-functions-neural-networks/

[17] GeeksForGeeks, *Sigmoid Function Graph*. Accessed: Jul. 01, 2023. [Online]. Available: https://media.geeksforgeeks.org/wp-content/uploads/20221013120722/1.png

[18] Jason Brownlee, “Understand the Impact of Learning Rate on Neural Network Performance,” *Machine Learning Mastery*, Jan. 24, 2019. https://machinelearningmastery.com/understand-the-dynamics-of-learning-rate-on-deep-learning-neural-networks/

[19] “Advantages and Drawbacks of Keras,” *TechVidvan*, Sep. 25, 2020. https://techvidvan.com/tutorials/advantages-drawbacks-of-keras/

[20] A. Saxena, “Introduction to NumPy,” *Medium*, Jun. 01, 2020. https://medium.com/analytics-vidhya/introduction-to-numpy-279bbc88c615