Detecting Network Based Intrusions using Neural Networks

The following paper aims to give an overview of some basic machine learning techniques that can be used to identify network based intrusions. This will include prepossessing steps used to format the data correctly and feature analysis. Neural network based models will then be applied to perform binary classification of network intrusion data into either normal patterns or attack patterns. The effectiveness of these models will be evaluated and improvements, including tuning their hyperparameters, will be considered.

Contents

1. Preprocessing and Data Analysis

The data source for this paper is the RT-IoT2022 network intrusion detection dataset, which can be downloaded from the UC Irvine Machine Learning repository (https://archive.ics.uci.edu/dataset/942/rt-iot2022). This is a previously proprietary dataset which was released in 2024. The data is gathered from a set of Internet-of-Things (IoT) devices, including the ThingSpeak-LED, Wipro-Bulb, and MQTT-Temp devices. Simulated

network attacks were performed against these devices, including brute-force, denial-of-service and network scanning patterns. The dataset is designed to help researches improve the capacity of network based intrusion detection systems.

The quality of the input data is a key factor that can determine how well a machine learning algorithm can learn. The main objectives of data preprocessing are as follows:

- Dealing with missing values in the dataset.
- Appropriately dealing with categorical data.
- Identifying relevant features that can be used in model construction.

The network intrusion detection dataset under evaluation does not have any missing values, meaning that every row in the dataset has assigned values for all input features. Consequently, there is no need to address missing values in the dataset, such as employing interpolation techniques to estimate values based on other training examples in the dataset.

Categorical data refers to variables that have specific categories or groups. Within categorical data, there are two additional subcategories: ordinal and nominal features.

Ordinal features represent a subtype of categorical variables where the values exhibit a meaningful order or ranking, although the distinctions between these values lacks a precise or meaningful definition. Take, for instance, t-shirt sizes, which constitute ordinal data due to the inherent ordering of sizes (small, medium, large) without a precisely defined difference between them. Ordinal features are commonly encoded into numerical values while retaining their original ordering. Using the example of t-shirt sizes, this encoding might assign values like 1 for small, 2 for medium, and 3 for large, providing a numerical representation that respects the established order of the data.

In nominal features, values represent distinct categories or groups without any inherent ordering or ranking. Consider the colour of a t-shirt, which can be categorised into groups such as blue, black, and green. However, these colour categories lack any inherent order. If we were to apply the same encoding as used for ordinal data, this would introduce an ordering to the data that does not exist. For instance, encoding blue as 1, black as 2, and green as 3 would erroneously imply an ordering; that is, blue is less than black which is less than green. The absence of a specific ordering in the colour values means that providing this information to a machine learning algorithm could lead the model to learn non-existent relationships between data points due to our chosen representation of data values.

To address this challenge, the recommended solution is to employ one-hot encoding. This technique involves generating new features for every distinct value in the nominal feature column. Using the example of t-shirt colours, this would result in three new features labelled *blue*, *black*, and *green*. Each colour is represented by a binary value. For example, the colour blue would be encoded as (blue=1, black=0, green=0), effectively capturing the presence or absence of each colour in a binary format.

Our dataset contained a number of features which contain categorical data, for example, *proto* and *service*. These two features correspond to the protocol used to make the connection (for example, TCP) and the service which was connected to (for example, http). Both features are nominal features as there is no inherent ordering to the values. To deal with these features we perform one-hot encoding to obtain new features, one for each unique value observed in these features.

Furthermore, we scale all numerical features to fall within the range of 0 to 1. This normalisation guarantees that no single feature will dominate the learning process and ensures equal contributions from all features. Without this normalisation, features with larger scales would be disproportionately emphasised by the model.

Finally, the class labels in this dataset correspond to the type of network attack that is being performed against the IoT devices. For example, the *ARP_poisioning* classification was encoded as 1 as this represents a network intrusion attack. However, the *MQTT* classification, which did not represent a network attack and was instead a normal pattern, was encoded as 0. The same encoding was performed across all categories of attack patterns and normal patterns.

Data processing increased the datasets total feature size from 85 to 94 features. All numerical features were suitably normalised in a range of $[0, 1]$. We appropriately dealt with categorical data by suitably encoding the data depending on whether the features were ordinal or nominal.

2. Neural Networks

A neural network (NN) is a computational model inspired by the structure and functioning of the human brain. NNs are designed to perform tasks related to pattern recognition and information processing. It consists of interconnected nodes, or artificial neurons, organised into layers. The three main types of layers in a neural network are the input layer, hidden layers, and the output layer. Each node in one layer is connected to each node in the next layer. Like synapses in the biological brain, each connection can transmit signals to other nodes. The signal received at each node is a real number and the output of each node is computed as a function if its inputs.

In a NN, information is passed through the network by adjusting the weights assigned to connections between neurons. Each connection has a weight that represents the strength of the connection. During training, the network learns to adjust these weights based on the input data and desired output, optimising its ability to make accurate predictions or classifications.

Neural networks are widely used in machine learning for tasks such as image and speech recognition, natural language processing, and solving complex problems where traditional algorithmic approaches may be less effective. The architecture and complexity of neural networks can vary, with deep neural networks having multiple hidden layers, giving rise to the term ‘deep learning.’

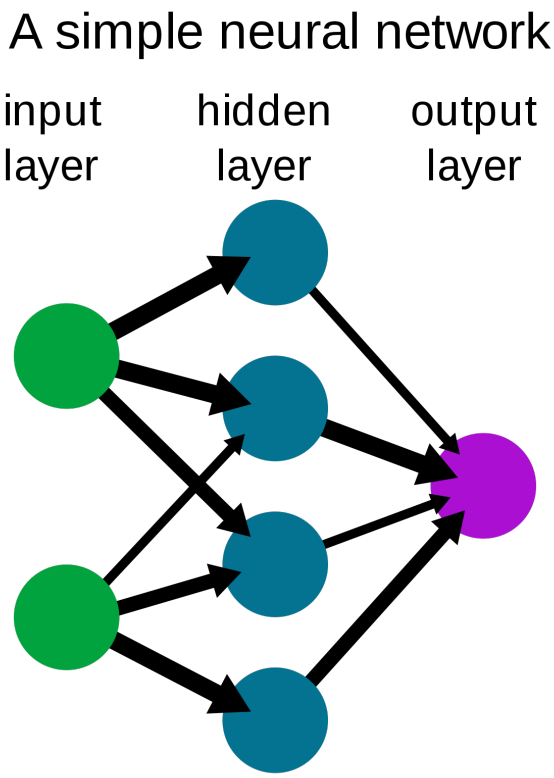

FIG. 1: A simplified example of a NN. There is an input layer, which takes in data fed to the algorithm, and consists of a set of nodes. The input layer nodes are each connected to every node in the hidden layer by a series of weighted connections. Larger weights imply a stronger connection between the nodes. The hidden layer nodes are then connected to the output layer nodes. The results in the output layer can be considered as the output of the neural network. NN can have more than one hidden layer of nodes, which is known as a deep neural network.

2.1. Artificial Neurons

NNs are composed of artificial neurons which are based on the structure of biological neurons. Each neuron has inputs and produces a single output. The output of each neuron is calculated by performing the weighted sum of its inputs (weighted by the weights of the connections to the neuron). We also add a bias term to this sum, which acts as an offset, allowing the model to fit better to the data. This sum is then passed through an activation function (which is usually non-linear) to produce an output.

More formally, we have a set of input values from the previous neurons in the network. Each one of these connections has a weight value which is used to rank the significance of these connections. These can be represented as vectors x and w for the input values and weights respectively.

$$w = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}, x = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$$

We then define an activation function σ, which is used to compute the output of the neuron, also called the neuron's activation. The activation can be computed as:

$$z = w_1 x_1 + \cdots + w_m x_m = w^T x + b,$$

where b is the bias vector, used to introduce an offset to shift the activation function. The activation of the neuron is then computed as $\sigma(z)$.

Neurons are organised into multiple layers, with this becoming particularly prominent in deep learning, where multiple layers of neurons are constructed. Neurons of one layer are connected to all neurons in the preceding and following layers. The input layer receives external data as input to the network. The output of the network is received from the output layer (or final layer). In between these layers are a set of hidden layers.

2.2. Activation Functions

An activation function is designed to introduce non-linearity to the model. As previously seen, it operates on the weighted sum of input values and a bias term in each neuron. The activation function decides whether a neuron should be activated or not, influencing the information flow through the network. By introducing non-linearity, activation functions enable neural networks to model complex relationships in data, capture patterns, and learn hierarchical representations.

Some popular activation functions include the sigmoid, hyperbolic tangent (tanh), and rectified linear unit (ReLU), each with its own characteristics and suitability for different scenarios:

Sigmoid The sigmoid function, also known as the logistic function, is a widely used activation function in NNs. It transforms input values into a range between 0 and 1, effectively squashing them to be interpreted as probabilities. The sigmoid function is characterized by its S-shaped curve, smoothly transitioning from near 0 for large negative inputs to near 1 for large positive inputs. This property makes it particularly useful in binary classification problems, where the output can be interpreted as the likelihood of belonging to a certain class.

Hyperbolic Tangent (tanh) Similar to the sigmoid function, tanh squashes input values, but it maps them to a range between -1 and 1, offering a symmetric S-shaped curve. This property allows tanh to mitigate the so called vanishing gradient problem to some extent compared to the sigmoid function.

ReLU The Rectified Linear Unit (ReLU) replaces all negative input values with zero and leaves positive values unchanged. This simple thresholding at zero introduces non-linearity to the model, allowing it to learn and represent complex patterns in the data.

2.3. Learning

A neural network learns through a process called backpropagation, which involves forward and backward passes through the network. During the forward pass, input data is fed through the network, and the weighted sum, activation function, and output are computed for each neuron. The predicted output is then compared to the actual target values, and the difference (error) is calculated using a loss function.

In order to perform the backward pass, we must define a function which the network should aim to minimise. For supervised learning, we defined the loss function which measures the difference between the predicted output and the actual target values for a given set of input data. It quantifies how well or poorly the model is performing on the task at hand. The goal of training the network is to minimise the value of the loss; lower values of the loss means the network is performing its task better as the difference between the network's predicted output and the actual values are lower.

In the backward pass, the gradient of the loss with respect to the network's weights are computed using the chain rule. This gradient represents the direction and magnitude of the steepest increase in the loss. The weights are then adjusted in the opposite direction of the gradient to minimise the loss. This process is typically performed using optimisation algorithms like stochastic gradient descent (SGD) or its variants.

During each iteration (epoch) of training, the weights are updated iteratively to reduce the error. This continues until the model reaches a state where the error is minimised, and the network has learned to make accurate predictions on the training data. Regularisation techniques, learning rate adjustments, and other optimisation strategies may be employed to enhance the training process and prevent overfitting.

2.4. Types of Learning

Machine learning is commonly categorised into three types of learning: supervised learning, unsupervised learning and reinforcement learning. We will discuss supervised and unsupervised learning in this section.

Supervised learning is a paradigm in machine learning where a model is trained on a labelled dataset, consisting of input-output pairs. The model is considered supervised as the correct answer for each input is given, allowing the model to learn the mapping from inputs to the correct outputs. In the context of machine learning, we measure the difference between the networks predicted output and the actual value using a loss function. As seen before, the network learns by minimising this loss function and updating the weights appropriately.

Unsupervised learning is a machine learning paradigm where the algorithm is given unlabeled data and tasked with discovering inherent patterns, structures, or relationships without being provided specific labelled outcomes. The model is provided a cost function, which is some function of the data and the network's output. The cost function is dependant on the task at hand and assumptions made about the problem. The network then learns by minimising this cost function.

3. Neural Networks Applied to Network Data

In the following section a standard artificial neural network will be applied to the RT-IoT2022 dataset. We will begin by implementing a relatively simple NN to begin with and then increase the complexity of the model in an effort to improve the performance of the model.

The NN will be constructed using PyTorch. PyTorch is an open-source deep learning framework that provides a flexible and dynamic computational graph, enabling researchers and developers to efficiently build and train neural networks.

3.1. Architecture and PyTorch Code

Our original model will be a fairly simplest model with only one hidden layer. The input layer will have 94 neurons which matches the 94 features in our input data. The hidden layer will have 16 neurons and the output layer will have two neurons corresponding to the two classes (either a attack pattern or normal network traffic pattern).

Initialise the model by instructing PyTorch to use a GPU if one is available.

```
import torch
import torch.nn as nn

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

The algorithm begins by importing and processing the dataset from a local CSV file which contains the data to analyse. Note that the data has already been appropriately normalised and preprocessed, as discussed in section 1. We also split the training data into training and testing datasets. The testing dataset will be used to train the model and the testing dataset will be used to evaluate the models performance.

```
import pandas as pd

# Load dataset
df = pd.read_csv('.\data\RT_IOT2022_sanitised.csv')

# Split into training and testing dataframes
# Shuffle the dataset
```

```
df = df.sample(frac=1, random_state=42).reset_index(drop=True)

train_size = int(0.8 * len(df))
train_df = df.iloc[:train_size, :]
train_df = train_df.reset_index(drop=True)
test_df = df.iloc[train_size:, :]
test_df = test_df.reset_index(drop=True)
```

A custom dataset class is then defined. The class is designed to take a pandas DataFrame as input, which is stored as a class attribute. The *__len__* method is implemented to return the total number of samples in the dataset, which is equivalent to the number of rows in the provided DataFrame. The *__getitem__* method is implemented to retrieve a specific sample from the dataset given an index (*idx*). This method returns the features and the labels for a particular index in the DataFrame.

```
from torch.utils.data import Dataset, DataLoader

class CustomDataset(Dataset):
    def __init__(self, dataframe):
        self.dataframe = dataframe

    def __len__(self):
        return len(self.dataframe)

    def __getitem__(self, idx):
        features = torch.tensor(self.dataframe.drop('class', axis=1).iloc[
    idx, :].values, dtype=torch.float32)
        target = torch.tensor(self.dataframe.loc[idx, 'class'], dtype=torch
    .float32)
        return features, target

# Load data into a custom dataset
train_dataset = CustomDataset(train_df)
test_dataset = CustomDataset(test_df)
```

These custom datasets (one for testing and one for training) are then loaded into a PyTorch DataLoader object. The DataLoader object is a utility that provides an efficient way to load and iterate over datasets. We've specified a batch size of 100, meaning the model will trained on a subset of data (in this case 100 samples) before updating the weights. This means that the model's weights will be updated multiple times during each epoch of training.

```
batch_size = 100
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=
    True)
```

```
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False
    )
```

The model is then defined in a class which is inherited from the *nn.Module* class. The class defines the layers in the neural network and the activation functions used at each layer, in this case the ReLU and Linear activation functions. The class also defines how the forward pass will be performed in the *forward* function; namely, how the activation of each neuron in the network will be calculated.

The loss and optimiser are also defined in section. We are using Cross Entropy Loss for the loss function and Stochastic Gradient Decent for the optimiser function.

```
class NeuralNet(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(NeuralNet, self).__init__()
        self.l1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.l2 = nn.Linear(hidden_size, 16)
        self.relu2 = nn.ReLU()
        self.l3 = nn.Linear(16, num_classes)

    def forward(self, x):
        out = self.l1(x)
        out = self.relu(out)
        out = self.l2(out)
        out = self.relu2(out)
        out = self.l3(out)
        return out

# Define hyperparamters
input_size = train_df.shape[1] - 1
hidden_size = 16
num_classes = 2
num_epochs = 10
learning_rate = 0.01

# Define model
model = NeuralNet(input_size, hidden_size, num_classes).to(device)

# loss and optimiser
criterion = nn.CrossEntropyLoss()
optimiser = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

A training loop is then performed. The training loop iterates over the number of training epochs

with an inner loop iterating over the batches in the training dataloader. The features and labels are also moved the GPU device if one is available. The NN is then used to make predictions based on the input features, as part of the forward pass. The loss is also calculated at this stage.

The optimiser's gradients are zeroed using *optimizer.zero_grad()* to prevent accumulation from previous iterations. The backward pass is then completed where the gradients of the loss with respect to the model parameters are computed. The optimiser then performs a parameter update based on the computed gradients.

```
n_total_steps = len(train_loader)
for epoch in range(num_epochs):
    for i, (features, labels) in enumerate(train_loader):
        features = features.to(device)
        labels = labels.to(device)
        labels = labels.long()

        # forward
        outputs = model(features)
        loss = criterion(outputs, labels)

        # backward
        optimiser.zero_grad()
        loss.backward()
        optimiser.step()

        if (i+1) % 5 == 0:
            print(f'Epoch {epoch+1} / {num_epochs} | Step {i+1} / {
   n_total_steps} | Loss = {loss.item():.4f}')
```

The algorithm then performs the testing validation over the testing dataset. The model is used to make predictions based on the input features. The *torch.max* function is used to obtain the predicted class indices. After processing all the batches in the testing dataset the accuracy of the model is then calculated.

```
with torch.no_grad():
    n_correct = 0
    n_samples = 0
    for features, labels in test_loader:
        features = features.to(device)
        labels = labels.to(device)
        outputs = model(features)


        # value, index
```

```
        _, predictions = torch.max(outputs, dim=1)
        n_samples += labels.shape[0]
        n_correct += (predictions == labels).sum().item()

    acc = 100.0 * n_correct / n_samples
    print(f'Accuracy = {acc}')
```

3.2. Accuracy and Hyperparameter Tuning

Our model, with 94 input features, a hidden layer with 16 neurons and 2 output neurons achieved an accuracy of 88.5% after training for 10 epochs. This suggests that our relatively simplistic model fails the capture patterns in this dataset appropriately. In particular, the model is unlikely to capture the highly non-linear relationships between input features and the output labels. In this section we will experiment with hyperparamter tuning to improve the models performance.

Increasing the number of parameters in the model increases the model's capacity. More parameters provide the neural network with greater flexibility and capacity to capture intricate patterns, dependencies, and non-linearities in the input data. One simple way to increase the number of parameters in the model is to increase the number of nodes in our hidden layer. For example lets increase the number of hidden neurons to 64.

```
class NeuralNet(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(NeuralNet, self).__init__()
        self.l1 = nn.Linear(input_size, 64)
        self.relu = nn.ReLU()
        self.l2 = nn.Linear(64, 64)
        self.relu2 = nn.ReLU()
        self.l3 = nn.Linear(64, num_classes)

    def forward(self, x):
        out = self.l1(x)
        out = self.relu(out)
        out = self.l2(out)
        out = self.relu2(out)
        out = self.l3(out)
        return out
```

In this case the model achieves an accuracy of 90.3% which is a minor improvement on our previous accuracy. This suggests the increasing the number of neurons in the hidden layer allows the model to capture non-linear relationships in the data. However, increasing the number of hidden neurons past this value had a negligible effect on the accuracy of the model. The next step is to add a second hidden layer and fine-tune the number of neurons in each hidden state.

We add a second hidden layer with 64 neurons as an initial test.

```
class NeuralNet(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(NeuralNet, self).__init__()
        self.l1 = nn.Linear(input_size, 64)
        self.relu = nn.ReLU()
        self.l2 = nn.Linear(64, 64)
        self.l3 = nn.Linear(64, 64)
        self.l4 = nn.Linear(64, num_classes)

    def forward(self, x):
        out = self.l1(x)
        out = self.relu(out)
        out = self.l2(out)
        out = self.relu(out)
        out = self.l3(out)
        out = self.relu(out)
        out = self.l4(out)
        return out
```

The model's accuracy remains fixed at 90.3% for this modified model. Additionally, increasing the number of hidden layers does not significantly alter the performance of the model. Training the model for more epochs also does not increase the model's performance. This could suggest that the optimised of loss function used by the model is not optimal.

Instead lets use the popular Adam optimiser for our training algorithm.

```
# loss and optimiser
criterion = nn.CrossEntropyLoss()
optimiser = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

This increases the accuracy of the model to 95% when considering a model with one hidden layer. However, increasing the number of hidden layers in the model does not improve the accuracy.

After tuning various hyperparmeters of the model the best accuracy that could be achieved on the dataset was 96.5%. This used the updated Adam optimiser and a learning rate of 0.02.

3.3. Learning Curve

A learning curve is a graphical representation that depicts the relationship between a model's performance and the amount of training data or the number of training iterations (epochs). Learning curves consist of a training performance and a validation performance; typically this

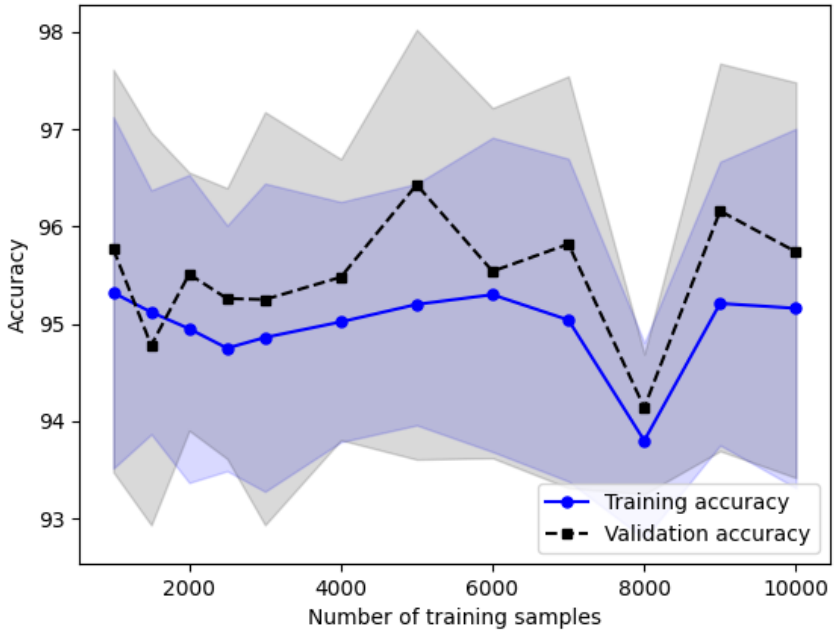

FIG. 2: The diagram shows the accuracy of the training dataset and the validation dataset by the number of training samples given to the model. The training accuracy is shown in blue and the validation accuracy is shown in black. Each dataset has the standard deviation show in the respective shaded colour. The training and validation performance reach a rough plateau over the training dataset suggesting that adding further training samples into the model will not improve its performance. The optimal number of training samples is anywhere in the region from 2000 to 4000 samples from the original dataset.

is the accuracy of the model. This is plotted against the number of training epochs or the total number of training samples supplied to the model.

If both the training and validation performance are poor and do not improve with additional data or iterations, it may indicate underfitting. The model is not capturing the underlying patterns in the data.

If the training performance is significantly better than the validation performance, and the gap increases with more data or iterations, it suggests overfitting. The model may be memorising the training data but failing to generalise to new data.

If both the training and validation performance stabilise and reach a plateau, it indicates that the model has likely converged and further training may not significantly improve performance.

Figure 2 shows the learning curve for the model by the total number of training samples supplied to the model. As can be seen the model has reached a rough plateau with the training accuracy and the validation accuracy reaching a rough convergence. Therefore, it is unlikely that adding more training data into the model will improve performance.

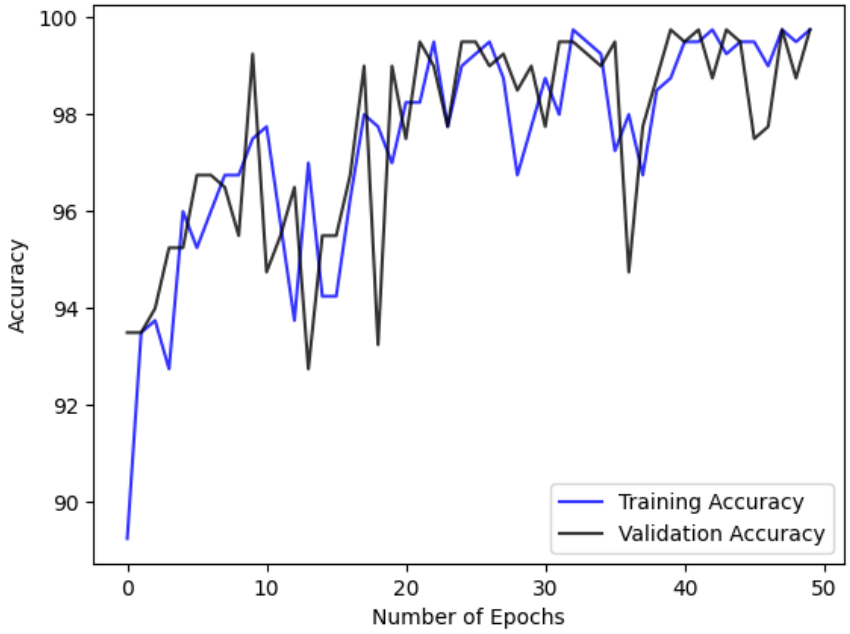

FIG. 3: The diagram shows the accuracy of the training dataset and the validation dataset by the number of epochs that the model is trained for. The training accuracy is shown in blue and the validation accuracy is shown in black. The training and validation performance reach a rough plateau over the training dataset suggesting that training epochs for the model will not improve its performance. We can see that the optimal number of epochs for training is roughly 20.

Figure 3 shows the learning curve for the model by the total number of epochs that the model is trained for. Again the model reaches a rough plateau after roughly 20 training epochs, with the training and validation accuracy converging. This suggests that training the model for more than 20 epochs will not improve its performance significantly.

3.4. Final Model

After performing our model hypertuning to identify the best optimiser algorithm, model architecture and learning rates, the model can be further improved using the results of our learning curves to further enhance the model.

The final model uses the Adam optimiser with a learning rate of 0.01. The model has a single hidden layer with 128 nodes. The number of epochs that the model is trained on is 20 and the total number of samples taken from the original dataset for training, validation and testing is 4000. The final model has an accuracy of 97.5%.

4. Conclusions

In conclusion, this paper has analysed the requirements for data preprocessing and the importance of appropriately normalising and transforming data before it is provided to a NN. We also outlined the basic building blocks and functionality of a NN and gave a brief overview on how the NN model is trained. We then looked at a practical application of NN to the RT-IoT2022 network data, and trained the model to detect attack patterns in the network. The model's hyperparameters were manually tuned to improve the performance of the model. We also used learning curves to identify the optimum number of training samples and training epochs for our model.

The NN was successfully able to identify attack patterns and normal traffic patterns with a final accuracy of 97.5%. The final accuracy of the model is reasonably good and certainly provides a useful baseline model for an intrusion detection systems (IDSs). However, such a detection system would likely need to employ other detection mechanisms alongside this model to fully account for all possible attack patterns. Furthermore, our model can only detect attack patterns, but can give no formalisation for what kind of attack pattern was taking place. An extension of this model could be to formulate a multi-class classification problem which classifies inputs into particular attack patterns rather than just a binary normal network traffic and attack pattern, as we have done in this research.