Murdoch University

ICT607: Artificial Intelligence for Cybersecurity

Experiment 6

Lab 6: Intrusion detection with artificial neural network

Intrusion detection is the process of monitoring computer networks or systems for unauthorized access or malicious activity. The goal of intrusion detection is to identify any abnormal behavior or security violations in a system or network and alert security personnel or automated response systems to take appropriate action.

In this laboratory, we will use the KDD Cup 1999 dataset, which is a widely used dataset for evaluating intrusion detection systems.

1 Load the preprocessed dataset from Lab 5

First, mount your Google Drive appropriately. If you do not have the preprocessed data in your Google Drive, you can download the data from LMS and upload into your Google Drive.

Alternatively, you can upload the data directly into your Colab session storage.

[1]:
```
import numpy as np
```

[2]:
```
# initilising variables so that Colab doesn't show warnings
X_train = np.load("drive/MyDrive/ICT607/KDD_Cup_Preprocessed/kddcup_X_train.
 ↪npy")
X_test = np.load("drive/MyDrive/ICT607/KDD_Cup_Preprocessed/kddcup_X_test.npy")
y_train = np.load("drive/MyDrive/ICT607/KDD_Cup_Preprocessed/kddcup_y_train.
 ↪npy")
y_test = np.load("drive/MyDrive/ICT607/KDD_Cup_Preprocessed/kddcup_y_test.npy")
```

[3]:
```
print(X_train.shape, X_test.shape)
print(y_train.shape, y_test.shape)
```

```
(330994, 29) (163027, 29)
(330994, 1) (163027, 1)
```

2 Building an artificial neural network classifier

Artificial neural network consists of a network of interconnected nodes, or artificial neurons, that can receive input data, process that data through a series of mathematical computations, and produce output data.

The neurons in a neural network are organized into layers, with each layer processing information and passing it on to the next layer until a final output is produced. Neural networks can be trained on a set of labeled data to learn patterns and relationships within that data, and then can be used to make predictions or classify new, unlabeled data.

In a neural network, weights and biases are the parameters that determine the behavior and output of the network.

Weights are the numerical values that are assigned to the connections between neurons in the network. These weights are learned during the training process, where the network is shown a set of labeled input-output pairs and adjusts its weights to minimize the difference between the predicted and actual outputs. The weights essentially control the strength and direction of the connections between neurons, and can greatly affect the accuracy and performance of the network.

Biases, on the other hand, are the values that are added to the output of each neuron in the network. These values are also learned during training, and help to adjust the overall output of the network. Biases can help to account for differences in the data and make the network more flexible and robust.

[4]:
```
import tensorflow as tf
import matplotlib.pyplot as plt
```

[5]:
```
model = tf.keras.Sequential([
    tf.keras.layers.Dense(
        units=8,
        activation="relu",
        input_dim=X_train.shape[1]
    ), # 1st hidden layer (or, 2nd layer because 1st layer is the input layer)

    tf.keras.layers.Dense(
        units=5,
        activation="softmax"
    ) # output layer
])
```

- 8 is the number of neurons (or units) in the layer. This is a hyperparameter that can be tuned to control the capacity and complexity of the neural network.
- activation='relu' specifies the activation function to be used for the layer. The 'relu' function is a common choice for hidden layers in neural networks, as it is efficient to compute and has been shown to work well in practice.
- input_dim=X_train.shape[1] specifies the input dimension of the layer. In this case, it indicates that the layer expects an input with X_train.shape[1] features. The input dimension is only required for the first layer of the neural network, as the subsequent layers will automatically infer the input dimension from the output of the previous layer.

[6]:
```
model.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"]
```

```
)
```

- optimizer="tf.keras.optimizers.Adam(learning_rate=0.001)" specifies the optimizer algorithm to be used during training. In this case, the Adam optimization algorithm will be used, with learning rate of 0.001. Adam is a popular gradient descent optimization algorithm that is computationally efficient and works well in practice for a wide range of neural network architectures and problems.
- loss="sparse_categorical_crossentropy" specifies the loss function to be used during training with **integer** labels. The sparse_categorical_crossentropy loss function is commonly used for multi-class classification problems, where each example belongs to exactly one class. This function calculates the cross-entropy loss between the predicted probabilities and the true class labels, and it is used to measure how well the model is performing during training.
- metrics=["accuracy"] specifies the evaluation metric(s) to be used during training and testing. In this case, the metric being used is "accuracy". This metric calculates the proportion of correctly classified examples out of all the examples in the dataset, and it is a commonly used metric for classification problems.

[7]:
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 8)                 240

 dense_1 (Dense)             (None, 5)                 45

=================================================================
Total params: 285 (1.11 KB)
Trainable params: 285 (1.11 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

[8]:
```
classify_history = model.fit(
    X_train,
    y_train,
    epochs=5,
    batch_size=32,
    validation_data=(X_test,y_test)
)
```

```
Epoch 1/5
10344/10344 [==============================] - 45s 4ms/step - loss: 0.0721 -
accuracy: 0.9785 - val_loss: 0.0300 - val_accuracy: 0.9915
Epoch 2/5
10344/10344 [==============================] - 31s 3ms/step - loss: 0.0253 -
accuracy: 0.9920 - val_loss: 0.0206 - val_accuracy: 0.9927
Epoch 3/5
```

```
10344/10344 [==============================] - 32s 3ms/step - loss: 0.0152 -
accuracy: 0.9950 - val_loss: 0.0114 - val_accuracy: 0.9977
Epoch 4/5
10344/10344 [==============================] - 33s 3ms/step - loss: 0.0090 -
accuracy: 0.9980 - val_loss: 0.0081 - val_accuracy: 0.9981
Epoch 5/5
10344/10344 [==============================] - 28s 3ms/step - loss: 0.0071 -
accuracy: 0.9983 - val_loss: 0.0072 - val_accuracy: 0.9982
```

- X_train: The input features of the training dataset. This is a NumPy array or Pandas DataFrame that contains the independent variables that will be used to predict the dependent variable.
- y_train: The target variable of the training dataset. This is a NumPy array or Pandas Series that contains the dependent variable that the model is trying to predict.
- epochs=5: The number of times the entire training dataset will be used to update the weights of the neural network model. One epoch is defined as one iteration over the entire training dataset.
- batch_size=32: The number of samples that will be used in each update of the model weights. The training dataset is divided into batches of this size, and the weights are updated after each batch. A smaller batch size can lead to more frequent updates and faster convergence, but it can also increase training time and memory usage.

[9]:
```
history_dict = classify_history.history
```

[10]:
```
history_dict
```

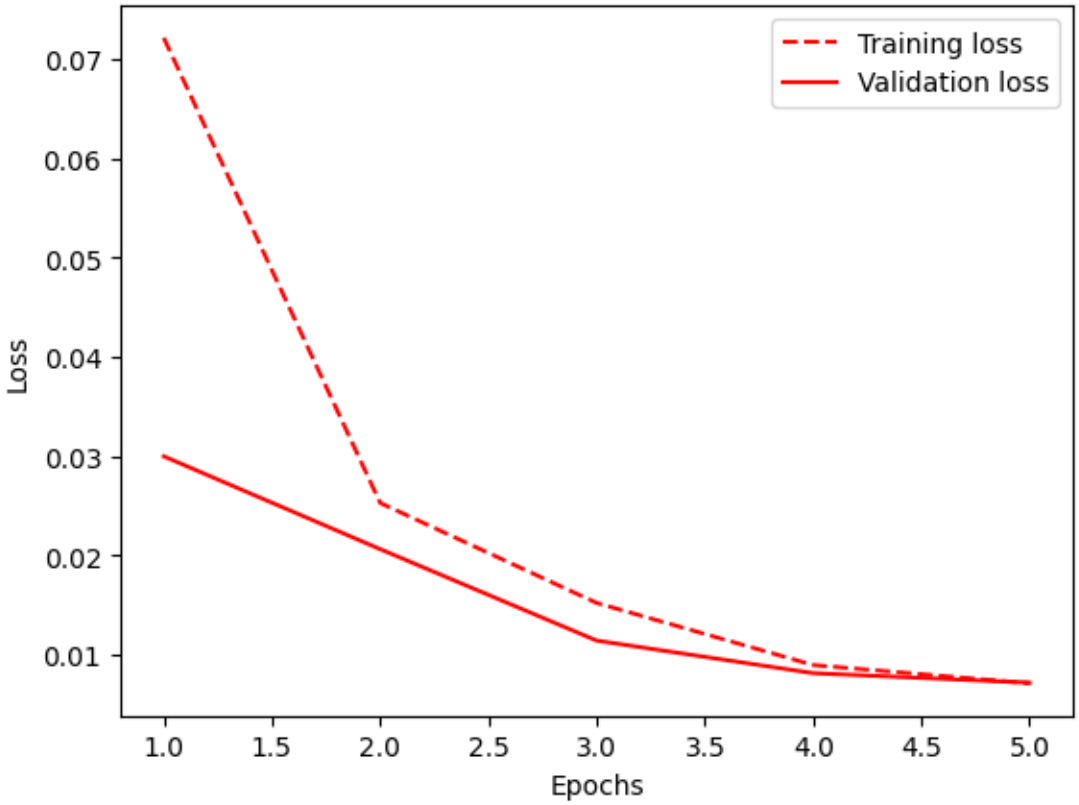
[10]:
```
{'loss': [0.07213364541530609,
  0.02528698928654194,
  0.015213788487017155,
  0.00896252691745758,
  0.007116894703358412],
 'accuracy': [0.9784950613975525,
  0.9919877648353577,
  0.9949848055839539,
  0.9979999661445618,
  0.9982749223709106],
 'val_loss': [0.02998279593884945,
  0.020633555948734283,
  0.011422812007367611,
  0.008144832216203213,
  0.007188682444393635],
 'val_accuracy': [0.9914737939834595,
  0.9926515221595764,
  0.9977181553840637,
  0.9980984926223755,
  0.998178243637085]}
```

[11]:
```
train_loss = history_dict["loss"]
val_loss = history_dict["val_loss"]
train_acc = history_dict["accuracy"]
val_acc = history_dict["val_accuracy"]

epochs = range(1,len(train_loss)+1)
```

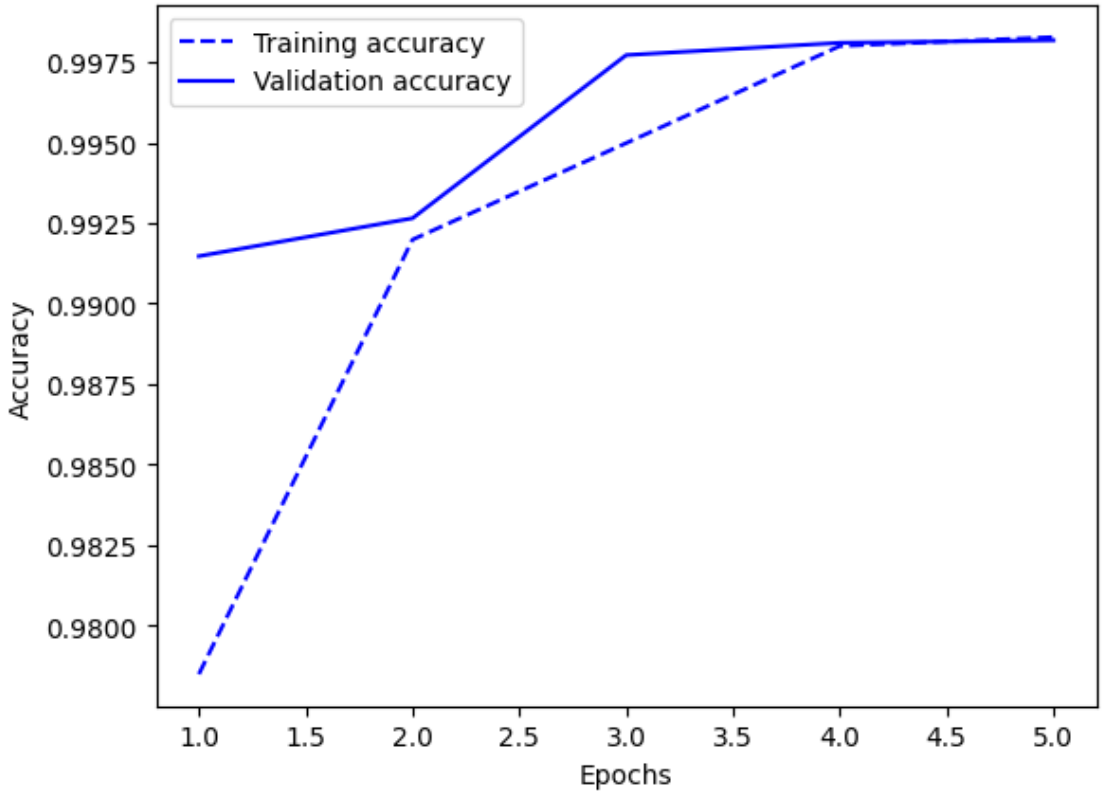
[12]:
```
plt.plot(epochs,train_loss,'r--',label="Training loss")
plt.plot(epochs,val_loss,'r',label="Validation loss")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
```

[12]: <matplotlib.legend.Legend at 0x7ce883864250>



[13]:
```
plt.plot(epochs,train_acc,'b--',label="Training accuracy")
plt.plot(epochs,val_acc,'b',label="Validation accuracy")
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.legend()
```

[13]: <matplotlib.legend.Legend at 0x7ce894c5b070>



3 Evaluatation by other matrics

3.1 Test accuracy

[14]:
```
from sklearn.metrics import accuracy_score
```

[15]:
```
test_pred = model.predict(X_test)
```

```
5095/5095 [==============================] - 8s 2ms/step
```

- model.predict(X_test) applies the trained neural network model to the input features X_test to generate predicted values for the target variable. The output of this function is a NumPy array containing the predicted values for each sample in the test dataset.

[16]:
```
test_pred
```

[16]:
```
array([[9.99999464e-01, 5.24002360e-07, 8.38908054e-09, 1.13246364e-10,
        1.65660513e-10],
       [9.99999464e-01, 5.24002360e-07, 8.38908054e-09, 1.13246364e-10,
        1.65660513e-10],
```

```
        [9.99999464e-01, 5.24002360e-07, 8.38908054e-09, 1.13246364e-10,
         1.65660513e-10],
        …,
        [1.00796507e-03, 9.98987257e-01, 2.17983455e-07, 1.99868987e-06,
         2.49041932e-06],
        [9.99951124e-01, 6.06711765e-08, 4.88055339e-05, 2.51041659e-08,
         3.30997718e-09],
        [9.99999523e-01, 5.24002417e-07, 8.38907965e-09, 1.13246371e-10,
         1.65660513e-10]], dtype=float32)
```

[17]:
```
y_test
```

[17]:
```
array([[0],
       [0],
       [0],
       …,
       [1],
       [0],
       [0]])
```

[18]:
```
y_pred = np.argmax(test_pred, axis=1)
```

[19]:
```
y_pred
```

[19]:
```
array([0, 0, 0, …, 1, 0, 0])
```

[20]:
```
test_acc = accuracy_score(y_test, y_pred)
```

- y_test is the actual target variable values for the test dataset.
- test_pred is the predicted target variable values for the test dataset.
- The np.argmax(test_pred, axis=1) function returns the index of the maximum value in each row of test_pred, which corresponds to the predicted class label.
- accuracy_score(y_test, np.argmax(test_pred, axis=1)) calculates the accuracy of the predicted labels by comparing them to the actual labels in y_test.

[21]:
```
print(f"Test accuracy: {test_acc}")
```

```
Test accuracy: 0.9981782158783514
```

Acknowledgement: Parts of this program is taken and improved from https://www.kaggle.com/code/iamyajat/intrusion-detection-system-using-neural-networks, which has been released under the Apache 2.0 open source license

3.2 Confusion matrix

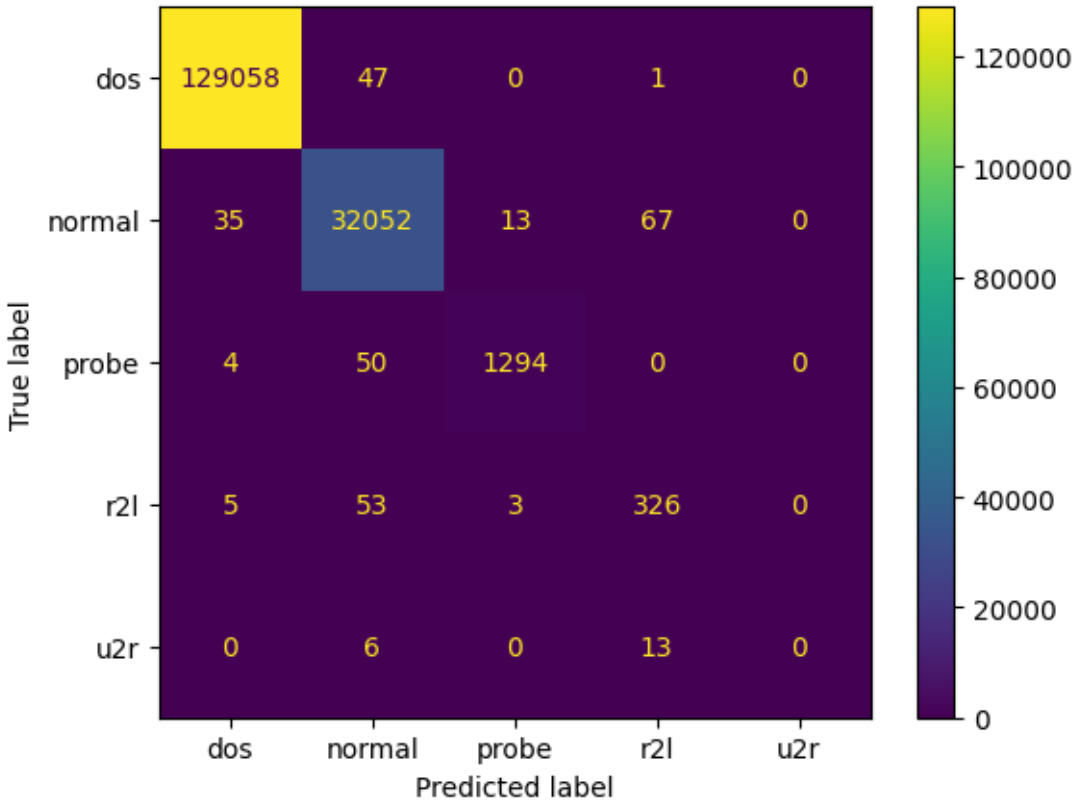
[22]:
```
import numpy as np
import itertools
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
```

[23]:
```
class_names = ['dos', 'normal', 'probe', 'r2l', 'u2r'] # same sequence as in␣
↪amap variable in load_KDD1999.ipynb
```

[24]:
```
conf_matrix = confusion_matrix(y_test, y_pred)
```

[25]:
```
conf_matrix = confusion_matrix(y_test, y_pred, normalize=None)
disp = ConfusionMatrixDisplay(confusion_matrix=conf_matrix,␣
↪display_labels=class_names)
disp.plot()
```

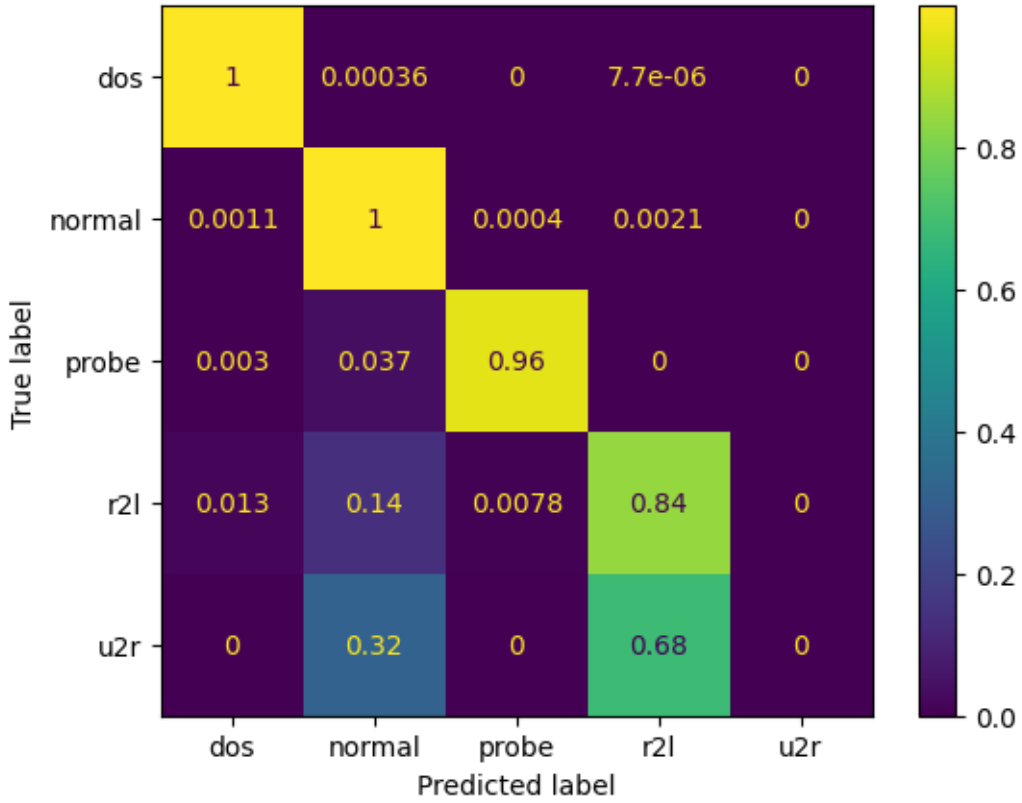
[25]:
```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at
0x7ce88130f850>
```



[26]:
```
conf_matrix = confusion_matrix(y_test, y_pred, normalize="true")
```

```
disp = ConfusionMatrixDisplay(confusion_matrix=conf_matrix,␣
 ↪display_labels=class_names)
disp.plot()
```

[26]: <sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x7ce881391a20>



3.3 F1 score

[27]:
```
from sklearn.metrics import f1_score
```

[28]:
```
f1_score(y_test, y_pred, average='macro')
```

[28]: 0.7580514177745272

4 Practice task

On the same KDD Cup dataset, develop a different neural network model and report performances (test accuracy, confusion matrix and F1 score) with different hyperparameters (such as, optimizer, activation function, learning rate, etc.).