Final

1. A Feed-Forward Neural Network

(FFNN) is a type of artificial neural network where connections between the nodes do not form cycles. It is the simplest type of artificial neural network and is the basis for many more complex network architectures.

Input layer receives the initial data. The number of neurons in this layer corresponds to the number of features in the input data.

Hidden Layers: One or more layers where the computation happens. Each neuron in a hidden layer takes input from every neuron in the previous layer, processes it, and passes the output to every neuron in the next layer.

Output Layer: The final layer produces the output of the network. The number of neurons here depends on the problem (e.g., one neuron for binary classification, multiple neurons for multi-class classification).

Each neuron processes input through a weighted sum of inputs, adds a bias, and then applies an activation function to introduce non-linearity.

Forward Propagation-

Weighted Sum: each neuron in a layer receives input from all neurons in the previous layer, which are multiplied by respective weights and summed up.

A bias term is added to the weighted sum to shift the activation function.

Activation Function: The result of the weighted sum and bias is passed through an activation function (like sigmoid, tanh, ReLU) to introduce non-linearity into the model.

Training process include: Initialization - weights and biases are initialized, often randomly. Forward Pass that input data is passed through the network to generate an output. Loss Calculation- the difference between the network's output and the actual target value is calculated using a loss function (e.g., mean squared error for regression, cross-entropy loss for classification).

Backward Pass (Backpropagation): The loss is propagated back through the network to update the weights and biases. This involves: calculating gradients of the loss with respect to each weight using the chain rule. Updating the weights and biases using an optimization algorithm (e.g., gradient descent).

Iteration - here the steps 2-4 are repeated for many iterations or epochs until the model converges (i.e., the loss is minimized).

Main characteristics- Non-cyclic- connections in FFNN are directed and acyclic, meaning the data flows in one direction—from input to output. Fully Connected- every neuron in one layer is connected to every neuron in the next layer. Feed-forward Nature- there are no feedback loops; outputs of one layer are inputs to the next, with no layer influencing a previous layer.

Mainc application

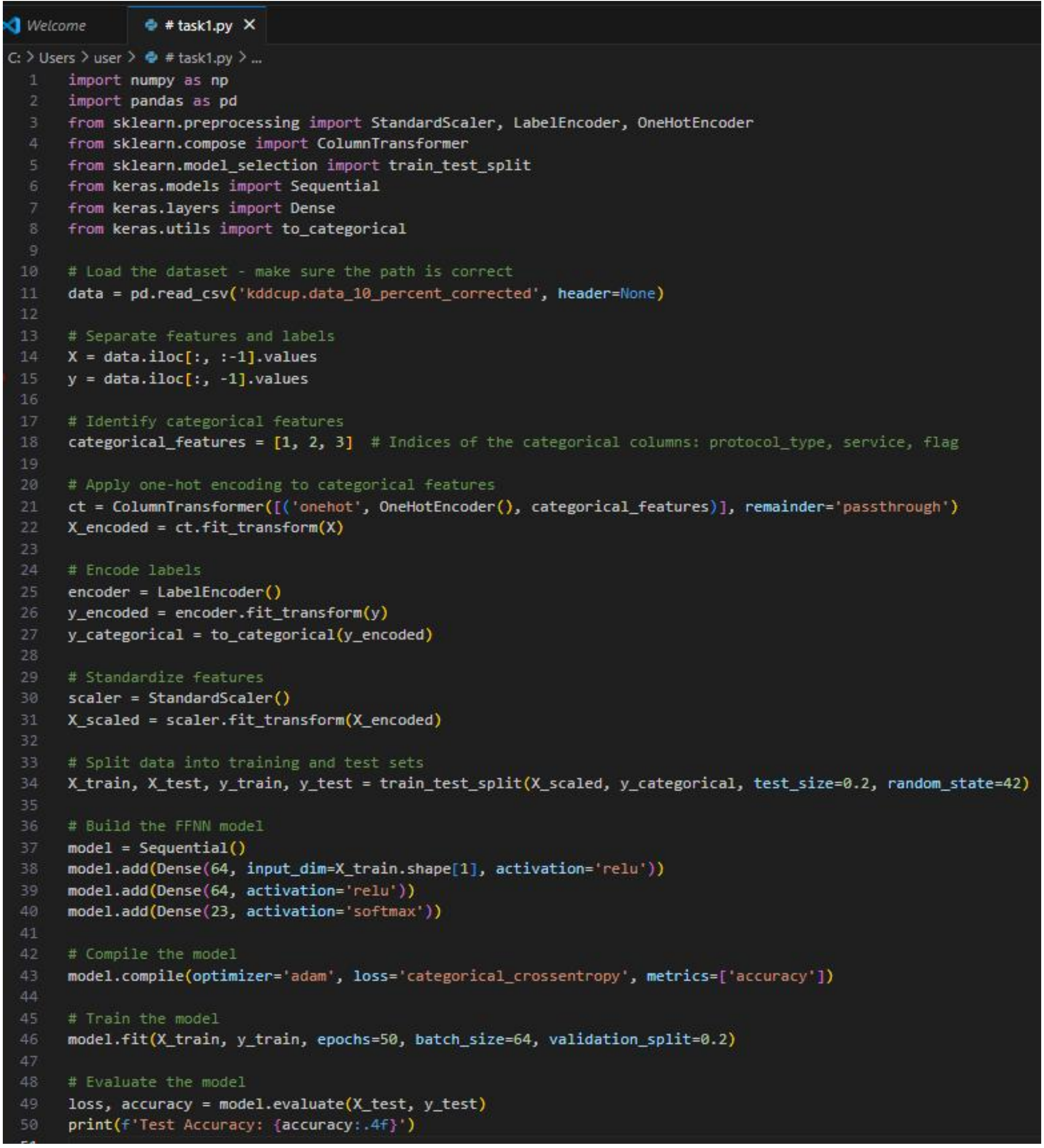
Classification: FFNNs can be used for binary and multi-class classification tasks. Regression- they can predict continuous values. Function Approximation- FFNNs can approximate complex functions given sufficient data and training.

About some limitations

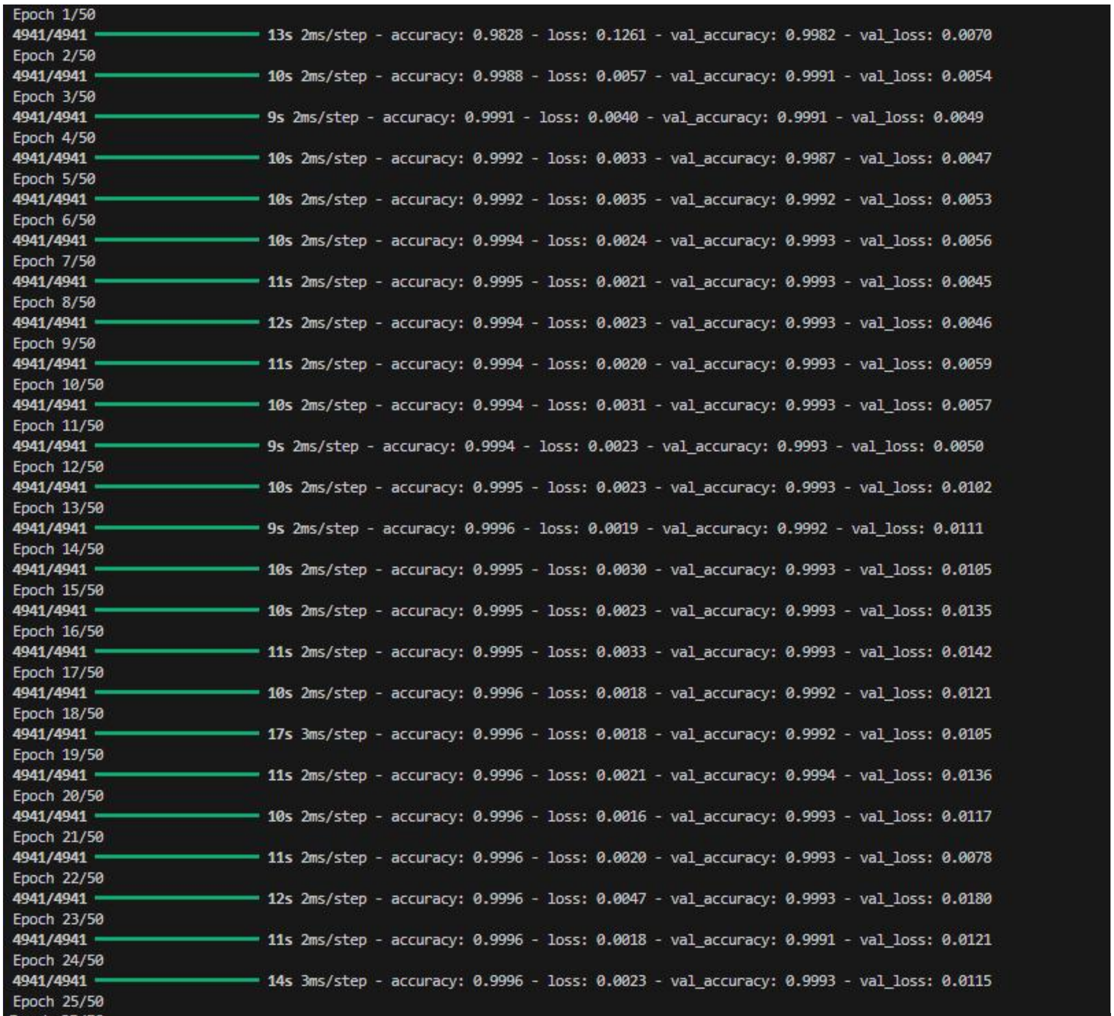
Overfitting: Without sufficient data or regularization techniques, FFNNs can overfit, performing well on training data but poorly on unseen data. Computationally Intensive: Training deep networks can be computationally expensive. Feature Engineering: FFNNs often require extensive preprocessing and feature engineering to perform well.

In summary, Feed-Forward Neural Networks are a foundational model in machine learning, effective for a wide range of tasks but with limitations that can be mitigated with techniques such as regularization, deeper architectures, and more advanced training algorithms.

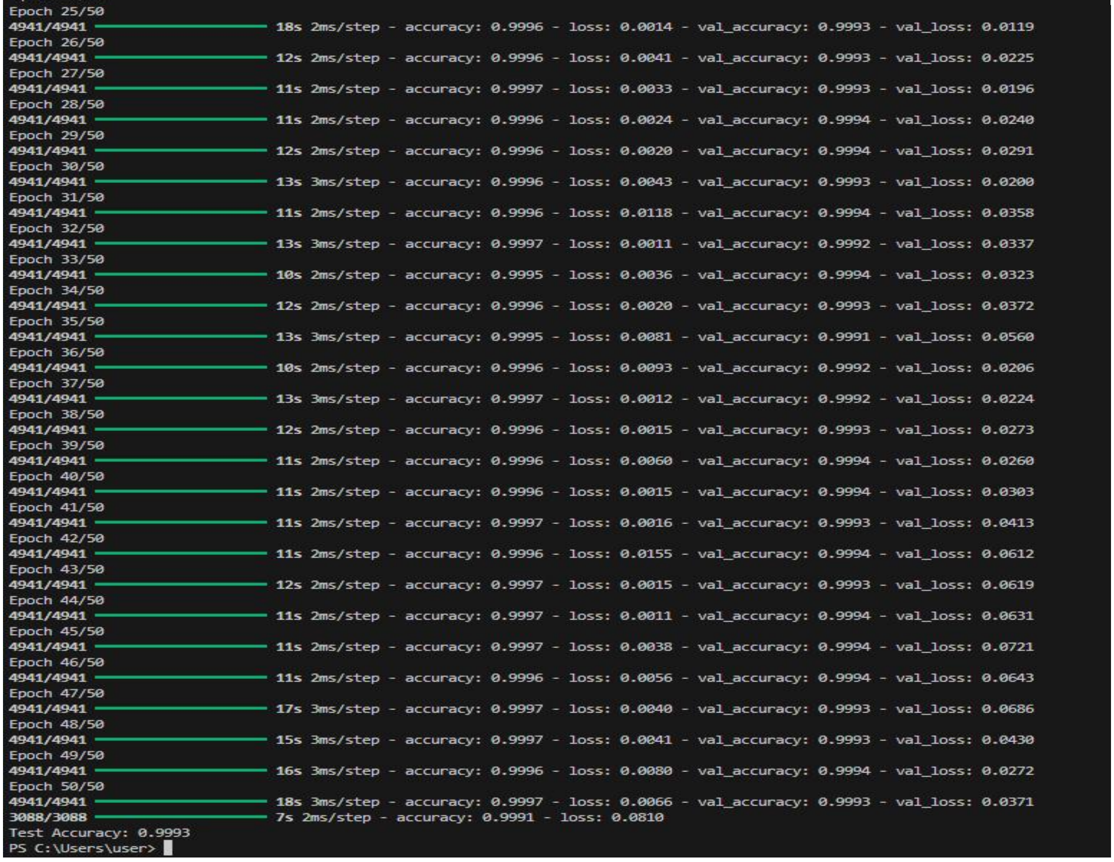
Practicale example:



```python
import numpy as np
import pandas as pd
from sklearn.preprocessing import StandardScaler, LabelEncoder, OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.model_selection import train_test_split
from keras.models import Sequential
from keras.layers import Dense
from keras.utils import to_categorical

# Load the dataset - make sure the path is correct
data = pd.read_csv('kddcup.data_10_percent_corrected', header=None)

# Separate features and labels
X = data.iloc[:, :-1].values
y = data.iloc[:, -1].values

# Identify categorical features
categorical_features = [1, 2, 3]  # Indices of the categorical columns: protocol_type, service, flag

# Apply one-hot encoding to categorical features
ct = ColumnTransformer([('onehot', OneHotEncoder(), categorical_features)], remainder='passthrough')
X_encoded = ct.fit_transform(X)

# Encode labels
encoder = LabelEncoder()
y_encoded = encoder.fit_transform(y)
y_categorical = to_categorical(y_encoded)

# Standardize features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X_encoded)

# Split data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y_categorical, test_size=0.2, random_state=42)

# Build the FFNN model
model = Sequential()
model.add(Dense(64, input_dim=X_train.shape[1], activation='relu'))
model.add(Dense(64, activation='relu'))
model.add(Dense(23, activation='softmax'))

# Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# Train the model
model.fit(X_train, y_train, epochs=50, batch_size=64, validation_split=0.2)

# Evaluate the model
loss, accuracy = model.evaluate(X_test, y_test)
print(f'Test Accuracy: {accuracy:.4f}')
```



```
Epoch 1/50
4941/4941 ━━━━━━━━━━━━━━━━━━━━ 13s 2ms/step - accuracy: 0.9828 - loss: 0.1261 - val_accuracy: 0.9982 - val_loss: 0.0070
Epoch 2/50
4941/4941 ━━━━━━━━━━━━━━━━━━━━ 10s 2ms/step - accuracy: 0.9988 - loss: 0.0057 - val_accuracy: 0.9991 - val_loss: 0.0054
Epoch 3/50
4941/4941 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.9991 - loss: 0.0040 - val_accuracy: 0.9991 - val_loss: 0.0049
Epoch 4/50
4941/4941 ━━━━━━━━━━━━━━━━━━━━ 10s 2ms/step - accuracy: 0.9992 - loss: 0.0033 - val_accuracy: 0.9987 - val_loss: 0.0047
Epoch 5/50
4941/4941 ━━━━━━━━━━━━━━━━━━━━ 10s 2ms/step - accuracy: 0.9992 - loss: 0.0035 - val_accuracy: 0.9992 - val_loss: 0.0053
Epoch 6/50
4941/4941 ━━━━━━━━━━━━━━━━━━━━ 10s 2ms/step - accuracy: 0.9994 - loss: 0.0024 - val_accuracy: 0.9993 - val_loss: 0.0056
Epoch 7/50
4941/4941 ━━━━━━━━━━━━━━━━━━━━ 11s 2ms/step - accuracy: 0.9995 - loss: 0.0021 - val_accuracy: 0.9993 - val_loss: 0.0045
Epoch 8/50
4941/4941 ━━━━━━━━━━━━━━━━━━━━ 12s 2ms/step - accuracy: 0.9994 - loss: 0.0023 - val_accuracy: 0.9993 - val_loss: 0.0046
Epoch 9/50
4941/4941 ━━━━━━━━━━━━━━━━━━━━ 11s 2ms/step - accuracy: 0.9994 - loss: 0.0020 - val_accuracy: 0.9993 - val_loss: 0.0059
Epoch 10/50
4941/4941 ━━━━━━━━━━━━━━━━━━━━ 10s 2ms/step - accuracy: 0.9994 - loss: 0.0031 - val_accuracy: 0.9993 - val_loss: 0.0057
Epoch 11/50
4941/4941 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.9994 - loss: 0.0023 - val_accuracy: 0.9993 - val_loss: 0.0050
Epoch 12/50
4941/4941 ━━━━━━━━━━━━━━━━━━━━ 10s 2ms/step - accuracy: 0.9995 - loss: 0.0023 - val_accuracy: 0.9993 - val_loss: 0.0102
Epoch 13/50
4941/4941 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.9996 - loss: 0.0019 - val_accuracy: 0.9992 - val_loss: 0.0111
Epoch 14/50
4941/4941 ━━━━━━━━━━━━━━━━━━━━ 10s 2ms/step - accuracy: 0.9995 - loss: 0.0030 - val_accuracy: 0.9993 - val_loss: 0.0105
Epoch 15/50
4941/4941 ━━━━━━━━━━━━━━━━━━━━ 10s 2ms/step - accuracy: 0.9995 - loss: 0.0023 - val_accuracy: 0.9993 - val_loss: 0.0135
Epoch 16/50
4941/4941 ━━━━━━━━━━━━━━━━━━━━ 11s 2ms/step - accuracy: 0.9995 - loss: 0.0033 - val_accuracy: 0.9993 - val_loss: 0.0142
Epoch 17/50
4941/4941 ━━━━━━━━━━━━━━━━━━━━ 10s 2ms/step - accuracy: 0.9996 - loss: 0.0018 - val_accuracy: 0.9992 - val_loss: 0.0121
Epoch 18/50
4941/4941 ━━━━━━━━━━━━━━━━━━━━ 17s 3ms/step - accuracy: 0.9996 - loss: 0.0018 - val_accuracy: 0.9992 - val_loss: 0.0105
Epoch 19/50
4941/4941 ━━━━━━━━━━━━━━━━━━━━ 11s 2ms/step - accuracy: 0.9996 - loss: 0.0021 - val_accuracy: 0.9994 - val_loss: 0.0136
Epoch 20/50
4941/4941 ━━━━━━━━━━━━━━━━━━━━ 10s 2ms/step - accuracy: 0.9996 - loss: 0.0016 - val_accuracy: 0.9993 - val_loss: 0.0117
Epoch 21/50
4941/4941 ━━━━━━━━━━━━━━━━━━━━ 11s 2ms/step - accuracy: 0.9996 - loss: 0.0020 - val_accuracy: 0.9993 - val_loss: 0.0078
Epoch 22/50
4941/4941 ━━━━━━━━━━━━━━━━━━━━ 12s 2ms/step - accuracy: 0.9996 - loss: 0.0047 - val_accuracy: 0.9993 - val_loss: 0.0180
Epoch 23/50
4941/4941 ━━━━━━━━━━━━━━━━━━━━ 11s 2ms/step - accuracy: 0.9996 - loss: 0.0018 - val_accuracy: 0.9991 - val_loss: 0.0121
Epoch 24/50
4941/4941 ━━━━━━━━━━━━━━━━━━━━ 14s 3ms/step - accuracy: 0.9996 - loss: 0.0023 - val_accuracy: 0.9993 - val_loss: 0.0115
Epoch 25/50
```



```
Epoch 25/50
4941/4941 ━━━━━━━━━━━━━━━━━━━━ 18s 2ms/step - accuracy: 0.9996 - loss: 0.0014 - val_accuracy: 0.9993 - val_loss: 0.0119
Epoch 26/50
4941/4941 ━━━━━━━━━━━━━━━━━━━━ 12s 2ms/step - accuracy: 0.9996 - loss: 0.0041 - val_accuracy: 0.9993 - val_loss: 0.0225
Epoch 27/50
4941/4941 ━━━━━━━━━━━━━━━━━━━━ 11s 2ms/step - accuracy: 0.9997 - loss: 0.0033 - val_accuracy: 0.9993 - val_loss: 0.0196
Epoch 28/50
4941/4941 ━━━━━━━━━━━━━━━━━━━━ 11s 2ms/step - accuracy: 0.9996 - loss: 0.0024 - val_accuracy: 0.9994 - val_loss: 0.0240
Epoch 29/50
4941/4941 ━━━━━━━━━━━━━━━━━━━━ 12s 2ms/step - accuracy: 0.9996 - loss: 0.0020 - val_accuracy: 0.9994 - val_loss: 0.0291
Epoch 30/50
4941/4941 ━━━━━━━━━━━━━━━━━━━━ 13s 3ms/step - accuracy: 0.9996 - loss: 0.0043 - val_accuracy: 0.9993 - val_loss: 0.0200
Epoch 31/50
4941/4941 ━━━━━━━━━━━━━━━━━━━━ 11s 2ms/step - accuracy: 0.9996 - loss: 0.0118 - val_accuracy: 0.9994 - val_loss: 0.0358
Epoch 32/50
4941/4941 ━━━━━━━━━━━━━━━━━━━━ 13s 3ms/step - accuracy: 0.9997 - loss: 0.0011 - val_accuracy: 0.9992 - val_loss: 0.0337
Epoch 33/50
4941/4941 ━━━━━━━━━━━━━━━━━━━━ 10s 2ms/step - accuracy: 0.9995 - loss: 0.0036 - val_accuracy: 0.9994 - val_loss: 0.0323
Epoch 34/50
4941/4941 ━━━━━━━━━━━━━━━━━━━━ 12s 2ms/step - accuracy: 0.9996 - loss: 0.0020 - val_accuracy: 0.9993 - val_loss: 0.0372
Epoch 35/50
4941/4941 ━━━━━━━━━━━━━━━━━━━━ 13s 3ms/step - accuracy: 0.9995 - loss: 0.0081 - val_accuracy: 0.9991 - val_loss: 0.0560
Epoch 36/50
4941/4941 ━━━━━━━━━━━━━━━━━━━━ 10s 2ms/step - accuracy: 0.9996 - loss: 0.0093 - val_accuracy: 0.9992 - val_loss: 0.0206
Epoch 37/50
4941/4941 ━━━━━━━━━━━━━━━━━━━━ 13s 3ms/step - accuracy: 0.9997 - loss: 0.0012 - val_accuracy: 0.9992 - val_loss: 0.0224
Epoch 38/50
4941/4941 ━━━━━━━━━━━━━━━━━━━━ 12s 2ms/step - accuracy: 0.9996 - loss: 0.0015 - val_accuracy: 0.9993 - val_loss: 0.0273
Epoch 39/50
4941/4941 ━━━━━━━━━━━━━━━━━━━━ 11s 2ms/step - accuracy: 0.9996 - loss: 0.0060 - val_accuracy: 0.9994 - val_loss: 0.0260
Epoch 40/50
4941/4941 ━━━━━━━━━━━━━━━━━━━━ 11s 2ms/step - accuracy: 0.9996 - loss: 0.0015 - val_accuracy: 0.9994 - val_loss: 0.0303
Epoch 41/50
4941/4941 ━━━━━━━━━━━━━━━━━━━━ 11s 2ms/step - accuracy: 0.9997 - loss: 0.0016 - val_accuracy: 0.9993 - val_loss: 0.0413
Epoch 42/50
4941/4941 ━━━━━━━━━━━━━━━━━━━━ 11s 2ms/step - accuracy: 0.9996 - loss: 0.0155 - val_accuracy: 0.9994 - val_loss: 0.0612
Epoch 43/50
4941/4941 ━━━━━━━━━━━━━━━━━━━━ 12s 2ms/step - accuracy: 0.9997 - loss: 0.0015 - val_accuracy: 0.9993 - val_loss: 0.0619
Epoch 44/50
4941/4941 ━━━━━━━━━━━━━━━━━━━━ 11s 2ms/step - accuracy: 0.9997 - loss: 0.0011 - val_accuracy: 0.9994 - val_loss: 0.0631
Epoch 45/50
4941/4941 ━━━━━━━━━━━━━━━━━━━━ 11s 2ms/step - accuracy: 0.9997 - loss: 0.0038 - val_accuracy: 0.9994 - val_loss: 0.0721
Epoch 46/50
4941/4941 ━━━━━━━━━━━━━━━━━━━━ 11s 2ms/step - accuracy: 0.9996 - loss: 0.0056 - val_accuracy: 0.9994 - val_loss: 0.0643
Epoch 47/50
4941/4941 ━━━━━━━━━━━━━━━━━━━━ 17s 3ms/step - accuracy: 0.9997 - loss: 0.0040 - val_accuracy: 0.9993 - val_loss: 0.0686
Epoch 48/50
4941/4941 ━━━━━━━━━━━━━━━━━━━━ 15s 3ms/step - accuracy: 0.9997 - loss: 0.0041 - val_accuracy: 0.9993 - val_loss: 0.0430
Epoch 49/50
4941/4941 ━━━━━━━━━━━━━━━━━━━━ 16s 3ms/step - accuracy: 0.9996 - loss: 0.0080 - val_accuracy: 0.9994 - val_loss: 0.0272
Epoch 50/50
4941/4941 ━━━━━━━━━━━━━━━━━━━━ 18s 3ms/step - accuracy: 0.9997 - loss: 0.0066 - val_accuracy: 0.9993 - val_loss: 0.0371
3088/3088 ━━━━━━━━━━━━━━━━━━━━ 7s 2ms/step - accuracy: 0.9991 - loss: 0.0810
Test Accuracy: 0.9993
PS C:\Users\user>
```

Test Accuracy: 0,9993

I have downloaded the KDD Cup 1999 dataset (it is a widely used dataset for evaluating machine learning models, particularly for intrusion detection systems (IDS). It was created for the Third International Knowledge Discovery and Data Mining Tools Competition and has become a standard benchmark in the field of cybersecurity. (Source code of the dataset: https://github.com/IndexFziQ/ML-ATIC/blob/master/kddcup.data_10_percent.gz)

By training an FFNN on the KDD Cup 1999 dataset, we can build an effective intrusion detection system. This practical example demonstrates the steps from data preparation to model evaluation, providing a foundation for deploying FFNNs in cybersecurity applications.