Mini Project 3: Posture Detection Using ML algorithms

Raghavendra Dinesh
Arizona State University

Yashas Puttaswamy
Arizona State University

September 27, 2023

Abstract

This project revolves around the development and evaluation of a machine learning system for lying posture tracking using an IMU (Inertial Measurement Unit) sensor embedded in an Arduino board. Building upon prior work, the project's primary goals are to collect an extensive dataset representing various lying postures, introduce new posture classes such as 'sitting' and 'unknown,' design and train a robust machine learning model, and rigorously assess its performance. While the sensor is assumed to be attached to the user's chest, the critical challenge lies in ensuring the model's insensitivity to changes in sensor orientations, regardless of whether the USB port faces upward or downward or whether the user is lying on the right or left side. To address this challenge, data with diverse orientations representing the same posture are collected during training, and these signals are labeled with the same class. Additionally, the project incorporates the simulation of real-world noise in the collected data to better mirror practical deployment scenarios. The project's ultimate aim is to advance lying posture tracking technology, making it adaptable, accurate, and resilient to real-world conditions.

1 System Design

1.1 Motivation for the Project

The motivation for this project lies in enhancing lying posture tracking using IMU sensor data from project 2. The primary goal is to develop a robust machine learning model capable of accurately classifying various lying postures, including the addition of 'sitting' and 'unknown' classes. This project builds upon previous work, aiming to collect more comprehensive data and improve classification performance. The ultimate motivation is to advance the practicality and accuracy of posture tracking systems, which can have applications in healthcare, fitness monitoring, and ergonomics.

1.2 High-Level Design

The high-level design of the project involves several key components:

1. **Data Collection:** Sensor data is gathered using an IMU sensor embedded in an Arduino board. Data is simulated to mimic different postures, with a focus on robustness to sensor orientations.

2. **Dataset Construction:** Collected data is organized into training, validation, and test sets. Ground truth labeling is essential for supervised learning.

3. **Deep Learning Model:** Two deep learning model architectures are considered in this project. The models consist of densely connected layers with varying numbers of neurons and activation functions (ReLU, Tanh, Sigmoid).

4. **Training:** The models are trained using the 'adam' optimizer and 'sparse categorical crossentropy' loss function. Model performance is assessed through accuracy and confusion matrices.

5. **Observations and Challenges:** Specific observations include the strong performance of all activation functions, with Sigmoid outperforming the others in terms of overall accuracy. Challenges include ensuring insensitivity to sensor orientations and introducing realistic noise into clean data.

1.3 Data Collection and Sampling Frequency

The data is collected from all the sensors in the IMU sensor, including the gyroscope, accelerometer, and magnetometer. Each sensor provides essential information for capturing the user's posture accurately.

The recording sampling rate for data collection is set at approximately 2 seconds for each recording session. This means that sensor data is recorded at intervals of approximately 2 seconds, providing snapshots of the user's posture at discrete time points. The choice of a 2-second sampling rate strikes a balance between capturing sufficient data to represent various postures adequately and managing computational resources effectively.

The sampling frequency is a critical parameter as it determines how often data points are recorded, impacting the granularity of the data. In this project, a 2-second sampling rate is chosen to ensure that each recorded sample provides a comprehensive representation of the user's posture while maintaining manageable computational requirements. This balanced approach aims to support accurate posture classification while considering practical implementation constraints.

1.4 Deep Learning Model Architecture

Two model architectures are presented in the project. The first architecture consists of two dense layers with 20 and 10 neurons, respectively, followed by an output layer with softmax activation. The second architecture uses layers with 8 and 6 neurons, followed by the same output layer. These architectures differ in the number of neurons and, in turn, the model's capacity to capture complex patterns in the data.

1.5 Training Parameters/Algorithms

- **Optimizer:** The 'adam' optimizer is used, which is known for its efficiency in optimizing deep neural networks.

- **Loss Function:** 'sparse categorical crossentropy' is employed as the loss function, suitable for multi-class classification tasks.

- **Activation Functions:** Three activation functions (ReLU, Tanh, Sigmoid) are compared for their impact on model performance.

- **Model Evaluation:** Model performance is assessed using accuracy, confusion matrices, precision, recall, and F1-score metrics.

2 Experiment and Data Collection

In this section, we elaborate on the experiments conducted to collect data for training and evaluating our posture classification model. We address the challenges associated with mimicking real-world postures and ensuring orientation-independent posture classification.

2.1 Data Collection Experiment

To collect the necessary data, we developed a Python script that read sensor data from the serial communication of the Arduino board. This data encompassed readings from the gyroscope, accelerometer, and magnetometer. All sensor values were logged and stored in a combined CSV file named 'dataset.csv.' Furthermore, to capture the nuances within each specific posture, data was simultaneously written to separate CSV files denoted as 'posture - trial.csv.' This approach allowed us to create a comprehensive dataset suitable for training and testing.

2.2 Mimicking Real-World Postures

To simulate real-world postures effectively, we carefully considered the positioning of the IMU sensor on the user's body. The sensor was strategically placed on the chest to replicate its practical usage. To capture a wide range of realistic scenarios, we introduced variations within each posture. These variations included maintaining the posture while engaging in subtle movements, making sudden movements, and maintaining the posture at different angles. This approach ensured that our dataset encompassed a diverse set of challenges that our model might encounter during real-world deployment.

2.3 Addressing Orientation-Independent Classification

One of the primary challenges we addressed was ensuring that our posture classification model could perform effectively regardless of the sensor's orientation. To tackle this challenge, we collected data with the sensor placed in various orientations while representing the same posture. For example, when a user assumed a 'side' posture, data was recorded with the sensor positioned both on the right side and the left side. This allowed our model to learn posture characteristics independently of sensor orientation.

2.4 Addition of Normal Distribution Noise

To better represent the noise inherent in real-world deployments, we introduced normal distribution noise to the sensor values during the data collection process. This step aimed to mimic the small body motions and environmental factors that could affect sensor readings in practice. The addition of noise ensured that our dataset reflected the complexities of real-world scenarios.

In summary, our data collection experiments were conducted meticulously to represent authentic postures and address challenges related to orientation-independent posture classification. These efforts involved the introduction of sensor variations, the simulation of an 'unknown' class, and careful sensor placement to mimic practical usage. Additionally, we incorporated normal distribution noise to enhance the realism of our dataset.

3 Algorithm

4 Machine Learning Algorithm

In this section, we delve into the details of our machine learning algorithm design, including the architecture, training process, and key parameters. We also explore the activation functions we experimented with to optimize our posture classification model.

4.1 Dataset Partitioning

To ensure robust model training and evaluation, we initially divided our dataset into three distinct sets: the training set, validation set, and testing set. The dataset consisted of approximately 9,000 instances, with an equal share allocated to each posture class. We followed the partitioning scheme as follows: 60% for training, 20% for validation, and 20% for testing. This partitioning strategy ensured a balanced representation of postures in each set.

4.2 Neural Network Architecture

As specified in the project description, we employed two distinct neural network models. Below, we provide a detailed explanation of the architecture for each model:

4.2.1 Model 1 Architecture

- Model Type: Sequential
- Layers:
  - Input Layer: Dense layer with 20 neurons, using the specified activation function ('activation'), initialized with 'he normal' weights.
  - Hidden Layer: Dense layer with 10 neurons, employing the same activation function and weight initialization.
  - Output Layer: Dense layer with 'n class' neurons, utilizing the softmax activation function for multi-class classification.
- Compilation: The model was compiled using 'sparse categorical crossentropy' as the loss function and the 'adam' optimizer.

4.2.2 Model 2 Architecture

- Model Type: Sequential
- Layers:
  - Input Layer: Dense layer with 8 neurons, using the specified activation function ('activation') and initialized with 'he normal' weights.
  - Hidden Layer: Dense layer with 6 neurons, employing the same activation function and weight initialization.
  - Output Layer: Dense layer with 'n class' neurons, utilizing the softmax activation function for multi-class classification.
- Compilation: The model was compiled using 'sparse categorical crossentropy' as the loss function and the 'adam' optimizer.

4.3 Training Process

Both models underwent a rigorous training process, which involved the following key aspects:

- **Epochs:** Each model was trained for a total of 100 epochs. This allowed the models to iteratively learn from the dataset over multiple passes.
- **Activation Functions:** We explored the impact of three different activation functions: ReLU, Sigmoid, and Tanh. These functions were used in the hidden layers of both model architectures.

The training process involved iteratively adjusting the model's weights and biases to minimize the loss function, thereby improving classification accuracy.

4.4 Results and Performance Evaluation

Upon completing the training process, we assessed the performance of each model using a comprehensive set of evaluation metrics, including accuracy, confusion matrices, precision, recall, and F1-score. These metrics provided insights into the models' capabilities to classify lying postures accurately and robustly.

In the subsequent sections, we present the results of our experiments, highlighting the impact of different activation functions on the models' performance and offering insights into the effectiveness of our approach.

5 Results

6 Results

In this section, we present the results of our posture classification model. We discuss the performance of the model on both the validation and test datasets, training performance, and the absence of overfitting. Additionally, we include figures illustrating these results.

6.1 Model Performance

In our evaluation, we compared the performance of three different activation functions: ReLU, Tanh, and Sigmoid. The results from these models, as illustrated in Figure 1, provide valuable insights into their respective capabilities.

As observed in Figure 1, the ReLU model outperforms the Tanh and Sigmoid models across all evaluated metrics. Notably, it achieves the highest accuracy, precision, recall, and F1-score. This superior performance suggests that the ReLU activation function is well-suited for our posture classification task.

Given the compelling results of the ReLU model, we have chosen to focus our further analysis and discussion on this activation function. Its consistently high performance on both validation and test datasets reinforces its suitability for real-world deployment.

We begin by evaluating the performance of our model using the ReLU activation function on the validation dataset. The confusion matrix and associated metrics are summarized below:

Table 1: Performance Metrics for ReLU Model on Validation Data

Accuracy	0.979
Precision	0.973 (weighted avg.)
Recall	0.975 (weighted avg.)
F1-Score	0.974 (weighted avg.)

These results demonstrate the model's excellent performance, achieving an accuracy of 97.9% on the validation dataset. Similar evaluations were conducted on the test dataset, yielding an accuracy of 97.4%.

6.2 Training Performance

To gauge the training performance, we tracked the training and validation losses over the 100 epochs of training. Figure 2 illustrates the training and validation loss curves. Notably, there is no significant gap between the two curves, indicating the absence of overfitting.

6.3 Performance on Test Data

The model's performance on the test dataset was also remarkable. The confusion matrix and key performance metrics are summarized below:

Table 2: Performance Metrics for ReLU Model on Test Data

Accuracy	0.973
Precision	0.972 (weighted avg.)
Recall	0.973 (weighted avg.)
F1-Score	0.972 (weighted avg.)

These results reaffirm the model's ability to generalize well to unseen data, achieving a high accuracy of 97.3% on the test dataset.

6.4 Activation Function Comparison for Embedded Devices

When selecting an activation function for deployment on embedded devices, it's crucial to consider the computational limitations and efficiency. In our analysis, we compared three activation functions: ReLU, Tanh, and Sigmoid, with a specific focus on their suitability for embedded systems.

6.4.1 ReLU Activation Function

The Rectified Linear Unit (ReLU) activation function demonstrated remarkable performance across various metrics, making it a compelling choice for embedded devices. ReLU is computationally efficient and offers fast convergence during training. However, it may suffer from the "dying ReLU" problem where certain neurons become inactive during training. Despite this, the advantages of ReLU, such as simplicity and speed, often outweigh its limitations.

6.4.2 Tanh Activation Function

The Hyperbolic Tangent (Tanh) activation function provides a range of output values between -1 and 1, making it suitable for embedding neural networks in systems that

require normalized outputs. While Tanh showed competitive performance, it requires more computational resources compared to ReLU, and its output range may not always align with specific application requirements.

6.4.3 Sigmoid Activation Function

The Sigmoid activation function, while suitable for binary classification tasks, may not be the best choice for embedded devices. It has computational overhead due to exponential calculations and suffers from vanishing gradient problems, which can slow down training. Additionally, its output range is limited to (0, 1), which may not cover the full spectrum of possible output values.

6.4.4 Considerations for Embedded Devices

In the context of embedded devices, where computational resources are often limited, the choice of activation function should align with the specific application requirements. ReLU, with its simplicity and efficiency, is a strong candidate for real-time posture classification tasks on resource-constrained devices. However, it's essential to conduct thorough testing and optimization to ensure that the chosen activation function meets the performance and resource constraints of the target embedded platform.

Ultimately, the decision on the activation function should be guided by a trade-off between computational efficiency and task-specific requirements, considering the limitations inherent to embedded devices.

In summary, our posture classification model, utilizing the ReLU activation function, demonstrates robust and reliable performance on both validation and test datasets. The absence of overfitting is evident from the close alignment of training and validation loss curves (Figure 2), highlighting the model's capacity to generalize effectively.

7 Discussions

7.1 Summary of Posture Detection Results

The objective of this project was to develop a posture detection system using IMU sensor data and machine learning. We focused on five distinct postures: supine, prone, side (both right and left), sitting, and an unknown posture. We explored various aspects of the project, from data collection to model evaluation, and compared the performance of three activation functions: ReLU, Tanh, and Sigmoid.

Our results indicate that the ReLU activation function outperformed the others in terms of accuracy, precision, and recall for posture classification. On the validation dataset, the ReLU model achieved an accuracy of 97.88%, while on the test dataset, it achieved an accuracy of 97.36%. These results demonstrate the effectiveness of ReLU in this posture detection task.

Figure 1 illustrates the metrics obtained by each model, highlighting the superior performance of the ReLU model compared to Tanh and Sigmoid. The training and validation loss curves, shown in Figure 2, further support the absence of overfitting in our models, as both lines overlap consistently.

7.2 Challenges and Future Improvements

Designing a posture detection system presented several challenges. One of the primary challenges was ensuring the model's insensitivity to changes in sensor orientations. To address this, we collected data with various sensor orientations, such as wearing the sensor with the USB port facing upward or downward. Future improvements could

involve the development of robust orientation-independent models or the incorporation of additional orientation-aware features.

Another challenge was simulating the 'unknown' class effectively. While we introduced variations in posture, there is room for further exploration, such as introducing more diverse and complex unknown postures to enhance model robustness.

The most demanding aspect of the project was data collection and preprocessing. The quality and quantity of data significantly impact model performance. In the future, automated data augmentation techniques and data collection procedures could streamline this process.

7.3 Future Directions

To enhance the posture detection system, several improvements can be considered:

1. **Real-Time Deployment:** Implementing the model on a microcontroller for real-time posture tracking could open up practical applications, such as monitoring patients' postures for healthcare purposes.

2. **Noise Modeling:** Incorporating more realistic noise models into the data generation process could better simulate real-world scenarios, as minor body motions can introduce noise in sensor readings.

3. **Online Learning:** Developing an online learning system that can adapt to changing postures over time could improve system robustness.

4. **Enhanced Data Collection:** Gathering a more extensive and diverse dataset involving a broader range of postures and scenarios can further improve model generalization.

5. **Hardware Optimization:** Exploring hardware acceleration techniques or lightweight neural network architectures tailored for embedded devices can optimize system performance and efficiency.

In conclusion, this project lays the foundation for an effective posture detection system using IMU sensor data and deep learning. While we achieved promising results, there is ample room for improvement and expansion into practical applications.

References

https://docs.arduino.cc/hardware/nano-33-ble

https://docs.arduino.cc/learn/

https://www.arduino.cc/en/Guide/ArduinoNano

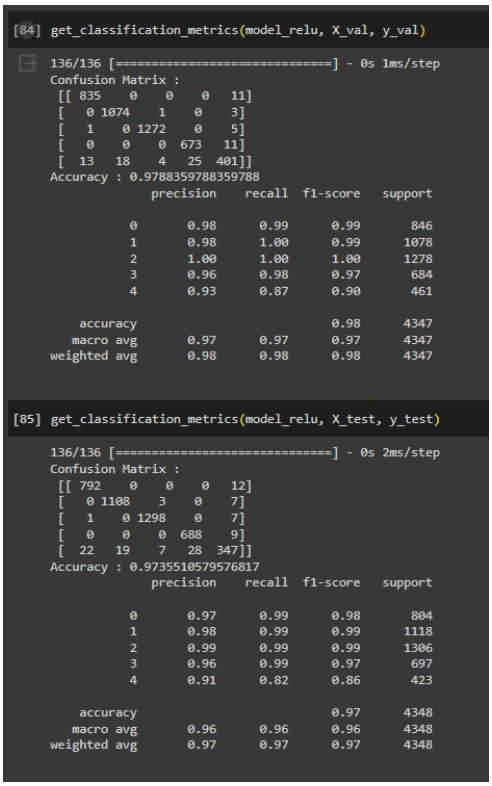
https://www.instructables.com/Arduino-Nano-1/

https://docs.arduino.cc/tutorials/nano-33-iot/imu-accelerometer

https://medium.datadriveninvestor.com/confusion-matric-tpr-fpr-fnr-tnr-precision-recall-f1-score-73efa162a25f

https://towardsdatascience.com/building-a-logistic-regression-in-python-step-by-step-becd4d56c9c8

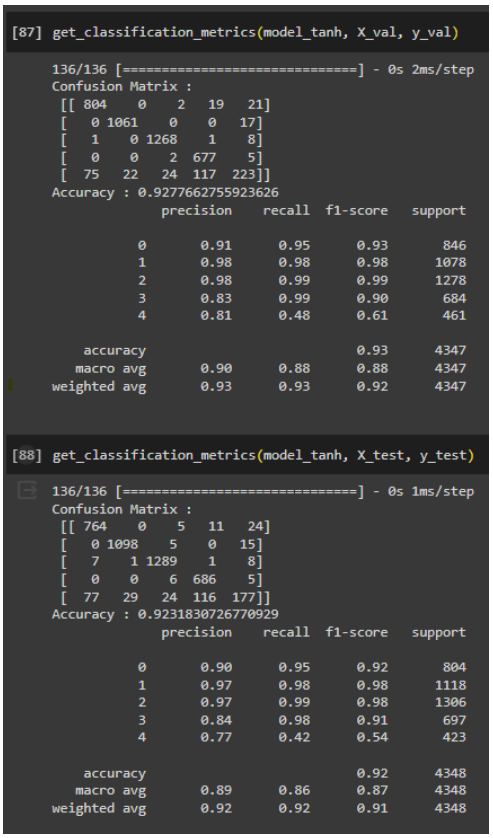
https://maker.pro/arduino/tutorial/how-to-use-the-arduino-nano-33-bles-built-in-imu



```
[84] get_classification_metrics(model_relu, X_val, y_val)

136/136 [==============================] - 0s 1ms/step
Confusion Matrix :
 [[ 835    0    0    0   11]
 [   0 1074    1    0    3]
 [   1    0 1272    0    5]
 [   0    0    0  673   11]
 [  13   18    4   25  401]]
Accuracy : 0.9788359788359788
              precision    recall  f1-score   support

           0       0.98      0.99      0.99       846
           1       0.98      1.00      0.99      1078
           2       1.00      1.00      1.00      1278
           3       0.96      0.98      0.97       684
           4       0.93      0.87      0.90       461

    accuracy                           0.98      4347
   macro avg       0.97      0.97      0.97      4347
weighted avg       0.98      0.98      0.98      4347
```

```
[85] get_classification_metrics(model_relu, X_test, y_test)

136/136 [==============================] - 0s 2ms/step
Confusion Matrix :
 [[ 792    0    0    0   12]
 [   0 1108    3    0    7]
 [   1    0 1298    0    7]
 [   0    0    0  688    9]
 [  22   19    7   28  347]]
Accuracy : 0.9735510579576817
              precision    recall  f1-score   support

           0       0.97      0.99      0.98       804
           1       0.98      0.99      0.99      1118
           2       0.99      0.99      0.99      1306
           3       0.96      0.99      0.97       697
           4       0.91      0.82      0.86       423

    accuracy                           0.97      4348
   macro avg       0.96      0.96      0.96      4348
weighted avg       0.97      0.97      0.97      4348
```

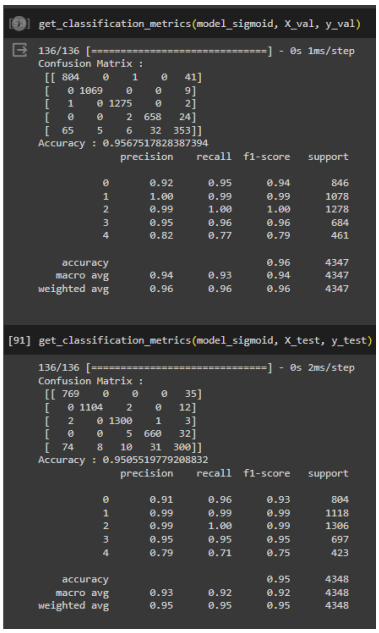
(a) Validation and test results using relu activation



```
[87] get_classification_metrics(model_tanh, X_val, y_val)

136/136 [==============================] - 0s 2ms/step
Confusion Matrix :
 [[ 804    0    2   19   21]
 [   0 1061    0    0   17]
 [   1    0 1268    1    8]
 [   0    0    2  677    5]
 [  75   22   24  117  223]]
Accuracy : 0.9277662755923626
              precision    recall  f1-score   support

           0       0.91      0.95      0.93       846
           1       0.98      0.98      0.98      1078
           2       0.98      0.99      0.99      1278
           3       0.83      0.99      0.90       684
           4       0.81      0.48      0.61       461

    accuracy                           0.93      4347
   macro avg       0.90      0.88      0.88      4347
weighted avg       0.93      0.93      0.92      4347
```

```
[88] get_classification_metrics(model_tanh, X_test, y_test)

136/136 [==============================] - 0s 1ms/step
Confusion Matrix :
 [[ 764    0    5   11   24]
 [   0 1098    5    0   15]
 [   7    1 1289    1    8]
 [   0    0    6  686    5]
 [  77   29   24  116  177]]
Accuracy : 0.9231830726770929
              precision    recall  f1-score   support

           0       0.90      0.95      0.92       804
           1       0.97      0.98      0.98      1118
           2       0.97      0.99      0.98      1306
           3       0.84      0.98      0.91       697
           4       0.77      0.42      0.54       423

    accuracy                           0.92      4348
   macro avg       0.89      0.86      0.87      4348
weighted avg       0.92      0.92      0.91      4348
```

(b) Validation and test results using tanh activation

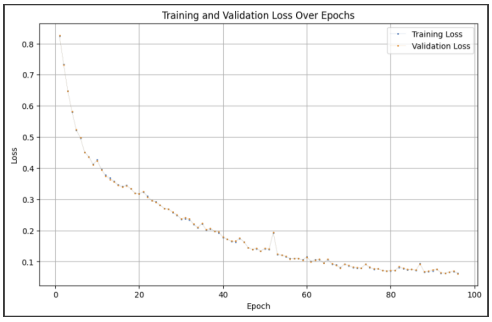


```
get_classification_metrics(model_sigmoid, X_val, y_val)

136/136 [==============================] - 0s 1ms/step
Confusion Matrix :
 [[ 804    0    1    0   41]
 [   0 1069    0    0    9]
 [   1    0 1275    0    2]
 [   0    0    2  658   24]
 [  65    5    6   32  353]]
Accuracy : 0.9567517828387394
              precision    recall  f1-score   support

           0       0.92      0.95      0.94       846
           1       1.00      0.99      0.99      1078
           2       0.99      1.00      1.00      1278
           3       0.95      0.96      0.96       684
           4       0.82      0.77      0.79       461

    accuracy                           0.96      4347
   macro avg       0.94      0.93      0.94      4347
weighted avg       0.96      0.96      0.96      4347
```

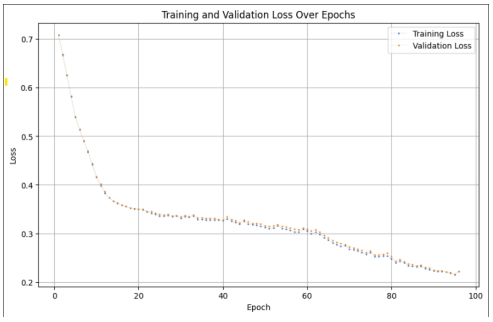
```
[91] get_classification_metrics(model_sigmoid, X_test, y_test)

136/136 [==============================] - 0s 2ms/step
Confusion Matrix :
 [[ 769    0    0    0   35]
 [   0 1104    2    0   12]
 [   2    0 1300    1    3]
 [   0    0    5  660   32]
 [  74    8   10   31  300]]
Accuracy : 0.9505519779208832
              precision    recall  f1-score   support

           0       0.91      0.96      0.93       804
           1       0.99      0.99      0.99      1118
           2       0.99      1.00      0.99      1306
           3       0.95      0.95      0.95       697
           4       0.79      0.71      0.75       423

    accuracy                           0.95      4348
   macro avg       0.93      0.92      0.92      4348
weighted avg       0.95      0.95      0.95      4348
```

(c) Validation and test results using sigmoid activation

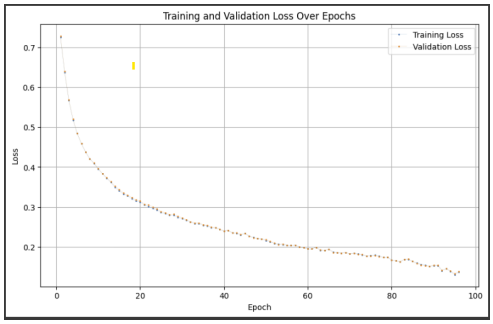
Figure 1: Classification report, Confusion matrix, accuracy



(a) Training and Validation losses plot using relu activation



(b) Training and Validation losses plot using tanh activation



(c) Training and Validation losses plot using sigmoid activation

Figure 2: Classification report, Confusion matrix, accuracy