Breast Cancer Detection using Deep Learning

Author: Neagu Mihnea, student at UBB Cluj-Napoca
Coordinator: Dr.Chira Camelia, University Proffesor

Abstract
Breast cancer remains one of the most prevalent and life-threatening diseases globally. Early detection plays a crucial role in improving patient outcomes and survival rates. In this project, we explore the application of deep learning techniques, specifically feedforward neural networks, for breast cancer detection using mammographic images. Leveraging a dataset comprising mammograms from diverse patient populations, our model aims to accurately classify images as indicative of malignant or benign tumors. Through extensive experimentation and validation, we demonstrate the efficacy of our approach in providing accurate and timely diagnosis, thus contributing to improved patient care and clinical decision-making in breast cancer detection.

Neagu Mihnea
Email: mihneaneagu709@yahoo.com GitHub: https://github.com/MihneaNeagu

1. Introduction and Subject description:

Breast cancer is a significant health concern globally, particularly in the United States, where a woman is diagnosed with breast cancer every two minutes. It stands as the leading cause of death for women worldwide, claiming over 40,000 lives annually from 220,000 medical diagnoses. However, advancements in early detection, improved screening methods, and heightened awareness since the 1990s have contributed to a decrease in breast cancer deaths. Notably, women who undergo early screening have a 53% higher chance of survival.

In women over 50, breast cancer ranks as the second most common cause of cancer-related death. Factors such as prolonged night shift work and excessive use of antiperspirants, deodorants, and underarm shaving have been associated with increased breast cancer risk. Multi-drug resistance poses a significant challenge in treatment, contributing to low survival rates among breast cancer patients.

Efforts in medical research have explored various techniques for breast cancer detection. The current approach focuses on utilizing Cellular Automata (CA) for segmentation, offering a visually compelling method compared to other computational techniques like neural networks. Cellular Automata, discrete spatial systems, operate with internal states updated based on local transition rules. This method showcases promise in detecting suspicious regions in mammograms, aligning with the needs of medical professionals. [1]

Experts conducted an extensive review of both deep learning (DL) and traditional machine learning (ML) approaches for breast cancer prediction. They analyzed a total of 8 papers in DL and 27 papers in ML, revealing that the majority of the literature primarily utilized imaging processes, with only a small fraction incorporating genetics. Similarly, [2] examined various imaging methods, specifically mammography, for breast cancer diagnosis, while Gupta et al. [3] provided an overview of different systems and techniques for early breast cancer detection, including radar-based imaging and microwave tomography.

Furthermore, Oyelade et al. [4] explored deep learning-based methods for breast cancer diagnosis from digital mammography, while Husaini investigated machine learning techniques and thermography for detecting breast cancer issues. The latter study delved into multiple ML methods for processing thermographic images related to breast cancer.

Upon reviewing the existing literature on deep learning-based methods for breast cancer detection, it is apparent that the predominant focus lies on image-based approaches. While traditional ML methods are often emphasized in previous studies, the coverage of

deep learning-based techniques remains limited, lacking a comprehensive and systematic analysis of existing approaches.

2. Materials and Methods

2.1. Dataset Overview. The WBDC[6] dataset comprises 569 instances, categorized into 357 benign and 212 malignant cases. Each instance includes an ID number, diagnosis (B = benign, M = malignant), and 30 features. These features are derived from digitized images of fine needle aspirates (FNA) of breast masses, as illustrated in **Figure 1**. Specifically, the ten real-valued features are 1. Radius, 2. Texture , 3. Perimeter, 4. Area, 5. Smoothness, 6. Compactness, 7. Concavity, 8. Concave points, 9. Symmetry, 10. Fractal dimension and are computed for each cell nucleus. For each image, the mean, standard error, and "worst" (the mean of the three largest values) of these features are calculated, resulting in a total of 30 features [5].

2.2 Feedforward Neural Networks. Feedforward Neural Networks (FNNs) are a type of artificial neural network (ANN) that are widely used in machine learning and artificial intelligence applications. The fundamental characteristic of FNNs is their unidirectional flow of information, where the data moves in only one direction, from the input layer to the output layer, as represented in **Figure 2**, without looping back. This differentiates FNNs from other types of ANNs, such as recurrent neural networks, which allow for feedback connections.

In a typical FNN, the input layer receives the external data or features that the network will process. These inputs are then passed to the hidden layer(s), where the actual processing occurs. The hidden layer(s) contain artificial neurons, which are mathematical functions that model the behavior of biological neurons. These neurons perform computations on the input data, using weights and biases that are adjusted during the training process. The output of the hidden layer(s) is then sent to the output layer, which produces the final result of the network's computation.

The architecture of FNNs can vary depending on the number of hidden layers employed. Simple FNNs may only have one hidden layer, while more complex networks can have multiple hidden layers. The use of multiple hidden layers allows for the modeling of more complex relationships between the input and output data, and is the basis for deep learning approaches.

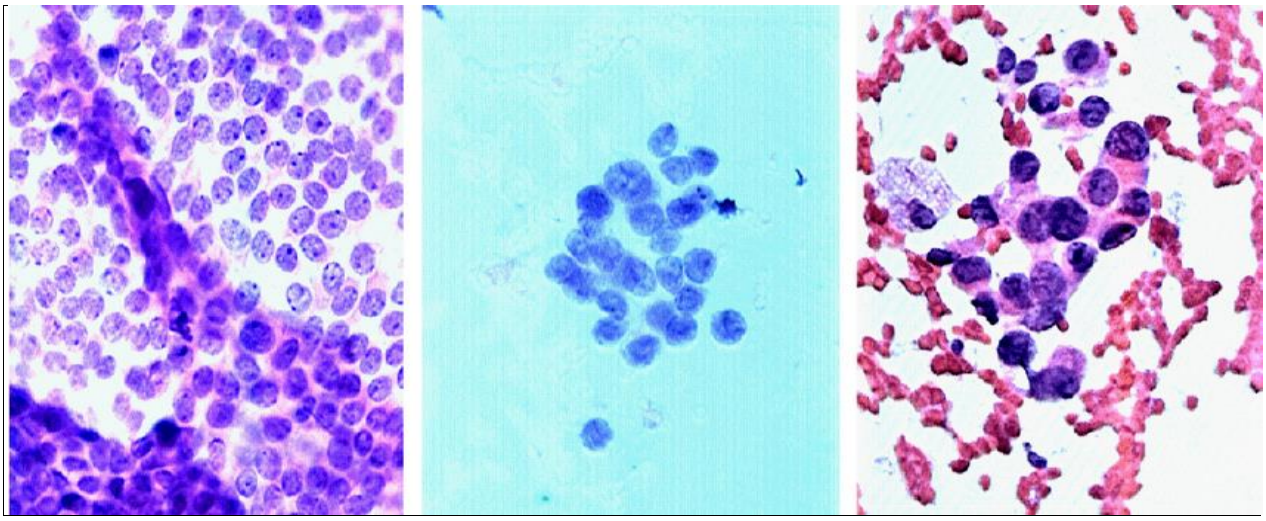


Figure 1: FNA breast biopsies, left:benign, center and right:malign

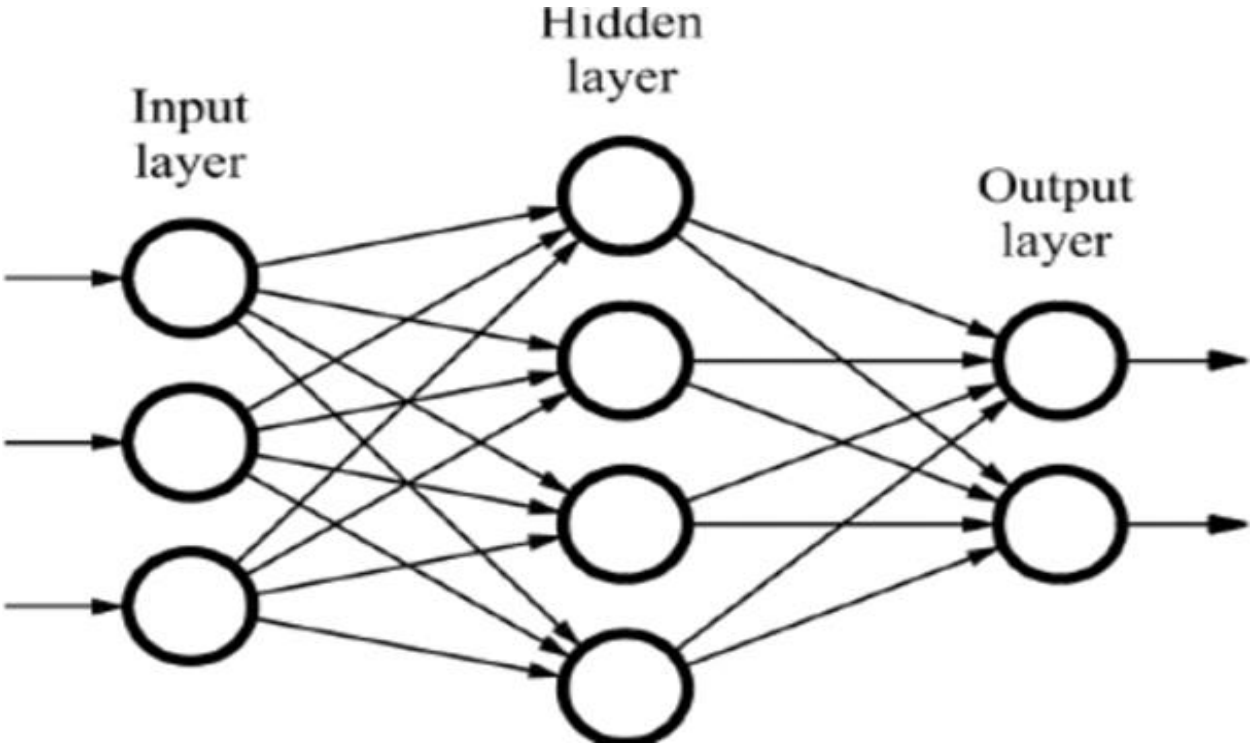


Figure 2: Feedforward neural Networks

3. Methodology and Implementation

Project description: Breast cancer detection using machine learning involves the development and implementation of algorithms that analyze mammograms, to identify potential abnormalities indicative of breast cancer. Machine learning

models are trained on large datasets of labeled images to learn patterns and features associated with both normal and cancerous breast tissue.

Breast cancer segmentation can be addressed such different approaches sunt as game theory, machine learning algorithms and others.

Problem statement analysis: Diagnose whether the patient has cancer or not using the attributes provided

- Attributes and diagnosis as a csv file
- Attributed are used from a digited image of a FNA(fine needle aspirate) of a breast mass
- Dataset used: https://archive.ics.uci.edu/dataset/17/breast+cancer+wisconsin+diagnostic
- Binary problem: B(benign), M(malign)

Strategy: Use the attributed given as input data and the two diagnosis labels: B and M as output on the training data. Then test the accuracy of the model on the testing data. The training data and the testing data will never have the overlaying data as to test the accuracy as strictly as possible.

3.1 Imports and data parsing

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
from sklearn.preprocessing import LabelEncoder, MinMaxScaler
from sklearn.model_selection import train_test_split
from keras.models import Sequential
from keras.layers import Dense, Dropout,Activation
from sklearn.metrics import confusion_matrix

```

The provided code imports essential Python libraries for data analysis and visualization: Pandas for data manipulation and analysis, Matplotlib[7] for diverse plot creation, Seaborn for statistical visualization, and NumPy[9] for numerical computations and array operations. These libraries collectively enable efficient data analysis, manipulation, and visualization in Python.

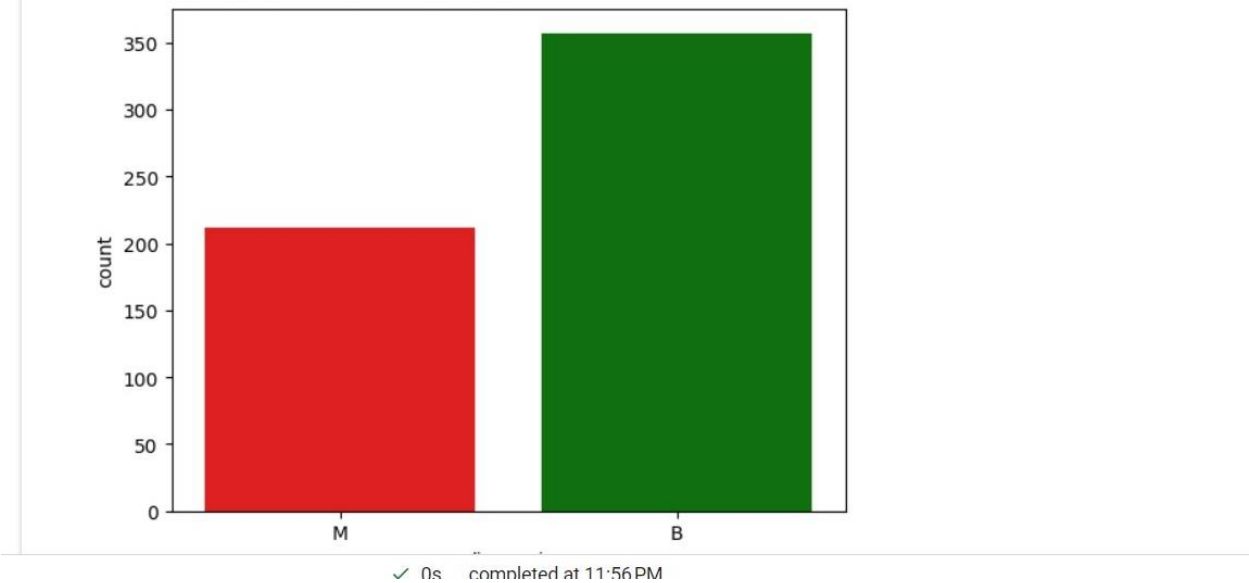
```
1 usage
def load_data(file_path):
    dt = pd.read_csv(file_path)
    dt.drop(dt.columns[32], axis=1, inplace=True)
    print(dt)
    return dt
```

	count	mean	std	min	25%	\
842302	568.0	3.042382e+07	1.251246e+08	8670.000000	869222.500000	
17.99	568.0	1.412049e+01	3.523416e+00	6.981000	11.697500	
10.38	568.0	1.930533e+01	4.288506e+00	9.710000	16.177500	
122.8	568.0	9.191475e+01	2.428585e+01	43.790000	75.135000	
1001	568.0	6.542798e+02	3.519238e+02	143.500000	420.175000	
0.1184	568.0	9.632148e-02	1.404601e-02	0.052630	0.086290	
0.2776	568.0	1.040360e-01	5.235523e-02	0.019380	0.064815	
0.3001	568.0	8.842731e-02	7.929422e-02	0.000000	0.029540	
0.1471	568.0	4.874629e-02	3.861717e-02	0.000000	0.020310	
0.2419	568.0	1.810549e-01	2.731942e-02	0.106000	0.161900	
0.07871	568.0	6.276960e-02	7.034862e-03	0.049960	0.057697	
1.095	568.0	4.039576e-01	2.760385e-01	0.111500	0.232375	
0.9053	568.0	1.217402e+00	5.519793e-01	0.360200	0.833150	
8.589	568.0	2.855984e+00	2.009288e+00	0.757000	1.605000	
153.4	568.0	4.013802e+01	4.528241e+01	6.802000	17.850000	
0.006399	568.0	7.042109e-03	3.005043e-03	0.001713	0.005166	
0.04904	568.0	2.543666e-02	1.789658e-02	0.002252	0.013048	
0.05373	568.0	3.185527e-02	3.019872e-02	0.000000	0.015062	
0.01587	568.0	1.178896e-02	6.173350e-03	0.000000	0.007634	
0.03003	568.0	2.052560e-02	8.264041e-03	0.007882	0.015128	
0.006193	568.0	3.790682e-03	2.646484e-03	0.000895	0.002244	
25.38	568.0	1.625315e+01	4.822320e+00	7.930000	13.010000	

✓ 0s completed at 11:32 PM

This code utilizes the pandas library to read the dataset from a CSV file and store it in a DataFrame called dt. The describe() method provides key statistical metrics such as count, mean, standard deviation, and quartiles for each numerical column in the dataset. By transposing the resulting DataFrame (T), the statistical summary is presented with column-wise statistics aligned, aiding in its interpretation. This code thus lays the groundwork for a rigorous academic exploration of the Wisconsin Breast Cancer Study data.

```
1 usage
def plot_countplot(dt):
    custom_palette = {"M": "red", "B": "green"}
    sns.countplot(x="diagnosis", data=dt, palette=custom_palette)
    plt.show()
```



This code utilizes seaborn to create a count plot (sns.countplot()) showcasing the volume of malignant (denoted as "M") and benign (denoted as "B") tumors in the dataset. The custom color palette custom_palette assigns red color to malignant tumors and green color to benign tumors for visual clarity. Additionally, labels for the x-axis ("Diagnosis") and y-axis ("Count") are added, along with a descriptive title ("Distribution of Malignant and Benign Tumors"). Finally, the plot is displayed using plt.show(). This visualization aids in understanding the distribution of tumor diagnoses within the study population.

3.2 Data processing

```
1 usage
def encode_diagnosis(dt):
    y = dt['diagnosis'].values
    print("Diagnosis before encoding are: ", np.unique(y))
    labelencoder = LabelEncoder()
    Y = labelencoder.fit_transform(y)
    print("Labels after encoding are: ", np.unique(Y))
    return Y
```

```
Data distribution:  B    357
M    212
Name: diagnosis, dtype: int64
Diagnosis before encoding are:  ['B' 'M']
Labels after encoding are:  [0 1]
```

This code snippet showcases the process of encoding the 'diagnosis' column in the dataset. It begins by printing the distribution of diagnoses to understand their frequency in the dataset. Subsequently, it extracts the 'diagnosis' column values and prints the unique values before encoding. Utilizing sklearn. Preprocessing . LabelEncoder, the diagnoses are encoded into boolean data, where 'M' is encoded as 1 and 'B' as 0. Finally, the unique encoded values are printed, illustrating the transformation achieved through encoding. This process aids in preparing the dataset for further analysis, particularly in machine learning tasks where categorical variables require numerical representation.

```
1 usage
def preprocess_data(dt):
    X = dt.drop(labels=["diagnosis", "id"], axis=1)
    print(X.describe().T)
    scaler = MinMaxScaler()
    scaler.fit(X)
    X = scaler.transform(X)
    print(X)
    return X

```

```
                         count        mean         std         min  \
radius_mean              569.0   14.127292    3.524049    6.981000
texture_mean             569.0   19.289649    4.301036    9.710000
perimeter_mean           569.0   91.969033   24.298981   43.790000
area_mean                569.0  654.889104  351.914129  143.500000
smoothness_mean          569.0    0.096360    0.014064    0.052630
compactness_mean         569.0    0.104341    0.052813    0.019380
concavity_mean           569.0    0.088799    0.079720    0.000000
concave points_mean      569.0    0.048919    0.038803    0.000000
symmetry_mean            569.0    0.181162    0.027414    0.106000
fractal_dimension_mean   569.0    0.062798    0.007060    0.049960
radius_se                569.0    0.405172    0.277313    0.111500
texture_se               569.0    1.216853    0.551648    0.360200
perimeter_se             569.0    2.866059    2.021855    0.757000
area_se                  569.0   40.337079   45.491006    6.802000
smoothness_se            569.0    0.007041    0.003003    0.001713
compactness_se           569.0    0.025478    0.017908    0.002252
concavity_se             569.0    0.031894    0.030186    0.000000
concave points_se        569.0    0.011796    0.006170    0.000000
symmetry_se              569.0    0.020542    0.008266    0.007882
fractal_dimension_se     569.0    0.003795    0.002646    0.000895
radius_worst             569.0   16.269190    4.833242    7.930000
texture_worst            569.0   25.677223    6.146258   12.020000
perimeter_worst          569.0  107.261213   33.602542   50.410000
area_worst               569.0  880.583128  569.356993  185.200000
smoothness_worst         569.0    0.132369    0.022832    0.071170
compactness_worst        569.0    0.254265    0.157336    0.027290
concavity_worst          569.0    0.272188    0.208624    0.000000
```

```
[[0.52103744 0.0226581  0.54598853 ... 0.59846245 0.41886396
 [0.64314449 0.27257355 0.61578329 ... 0.23358959 0.22287813
 [0.60149557 0.3902604  0.59574321 ... 0.40370589 0.21343303
 ...
 [0.45525108 0.62123774 0.44578813 ... 0.12872068 0.1519087
 [0.64456434 0.66351031 0.66553797 ... 0.49714173 0.45231536
 [0.03686876 0.50152181 0.02853984 ... 0.25744136 0.10068215
```

This code segment showcases the process of encoding the 'diagnosis' column in the dataset. It begins by printing the distribution of diagnoses to understand their frequency in the dataset. Subsequently, it extracts the 'diagnosis' column values and prints the unique values before encoding. Utilizing sklearn. preprocessing. LabelEncoder, the diagnoses are encoded into boolean data, where 'M' is encoded as 1 and 'B' as 0. Finally, the unique encoded values are printed, illustrating the transformation achieved through encoding. This process aids in preparing the dataset for further analysis, particularly in machine learning tasks where categorical variables require numerical representation.

This epitomizes the foundational procedures in data analysis, spanning from the initial loading and preprocessing of the dataset to conducting exploratory data analysis (EDA). It represents a systematic methodology for comprehensively understanding the Wisconsin Breast Cancer Study dataset, thereby establishing a solid foundation for subsequent analyses and insights within the realm of breast cancer research.

```
1 usage
def split_data(X, Y):
    X_train, X_test, y_train, y_test = train_test_split( *arrays: X, Y, test_size=0.25, random_state=42)
    print("Training data is:", X_train.shape)
    print("Testing data is:", X_test.shape)
    return X_train, X_test, y_train, y_test
```

```
Training data is: (483, 31)
Testing data is: (86, 31)
```

This code snippet encompasses a crucial phase in machine learning—splitting the dataset into training and testing subsets. Utilizing the train_test_split function from scikit-learn, the dataset is divided into training and testing sets to facilitate model training and evaluation. A test size of 15% is specified, indicating that 15% of the data will be reserved for testing, while the remaining 85% will be utilized for training. Additionally, a random state of 42 ensures reproducibility of the split. The subsequent print statements provide insights into the dimensions of the resulting training and testing datasets, essential for understanding the data partitioning process within the breast cancer research project.

3.3 Model building and training

```
1 usage
def build_model():
    model = Sequential()
    model.add(Dense( units: 16, input_dim=30, activation='relu'))
    model.add(Dropout(0.2))
    model.add(Dense(1))
    model.add(Activation('sigmoid'))
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    print(model.summary())
    return model
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_1 (Dense)             (None, 16)                512

 dropout (Dropout)           (None, 16)                0

 dense_2 (Dense)             (None, 1)                 17

 activation (Activation)     (None, 1)                 0

=================================================================
Total params: 529 (2.07 KB)
Trainable params: 529 (2.07 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
None
```

The provided code initializes a neural network model for binary classification tasks using Keras. It comprises an input layer with 31 features, a hidden layer with 16 units using the ReLU activation function, and an output layer with a single unit using the sigmoid activation function. Dropout regularization is incorporated to prevent overfitting by randomly dropping 20% of input units during training. The model is compiled with binary cross-entropy loss, Adam optimizer, and accuracy metric for evaluation. Lastly, model.summary() prints a summary detailing the architecture, parameters, and connections within the neural network model.

ReLU (Rectified Linear Activation):

ReLU is an activation function commonly used in neural networks. It replaces negative values with zero and leaves positive values unchanged. Mathematically, ReLU is defined as f(x) = max(0, x). The main advantage of ReLU is that it helps alleviate the vanishing gradient problem, where gradients become extremely small during backpropagation, by enabling faster and more effective training of deep neural networks.

Sigmoid Activation:

The sigmoid activation function is commonly used in binary classification tasks. It squashes the output to a range between 0 and 1, interpreting the output as the probability of the positive class. Mathematically, the sigmoid function is defined as f(x) = 1 / (1 + e^(-x)).

Binary Cross-Entropy Loss:

Binary cross-entropy is a loss function commonly used in binary classification tasks. It measures the dissimilarity between the predicted probabilities and the actual binary labels. In the context of this neural network, the binary cross-entropy loss function is used to quantify the difference between the predicted probability distribution (using the sigmoid activation function) and the true binary labels.

Adam Optimizer:

Adam is an adaptive learning rate optimization algorithm commonly used for training deep neural networks. It combines the benefits of two other popular optimization techniques: RMSProp and AdaGrad. Adam adapts the learning rate for each parameter individually, based on estimates of the first and second moments of the gradients, resulting in more effective training and faster convergence.

Accuracy Metric:

Accuracy is a commonly used metric for evaluating classification models. It measures the proportion of correctly classified instances out of the total number of instances. In the context of this neural network, the accuracy metric is used to

assess the performance of the model in correctly predicting the binary labels (0 or 1) for the given dataset.

```
1 usage
def train_model(model, X_train, y_train, X_test, y_test):
    history = model.fit(X_train, y_train, verbose=1, epochs=100, batch_size=64,
                        validation_data=(X_test, y_test))
    return history
```

```
Epoch 1/100
7/7 [==============================] - 1s 35ms/step - loss: 0.7108 - accuracy: 0.5751 - val_loss: 0.7007 - val_accuracy: 0.6014
Epoch 2/100
7/7 [==============================] - 0s 10ms/step - loss: 0.7015 - accuracy: 0.5657 - val_loss: 0.6867 - val_accuracy: 0.6294
Epoch 3/100
7/7 [==============================] - 0s 7ms/step - loss: 0.6878 - accuracy: 0.5986 - val_loss: 0.6744 - val_accuracy: 0.6783
Epoch 4/100
7/7 [==============================] - 0s 7ms/step - loss: 0.6852 - accuracy: 0.6009 - val_loss: 0.6636 - val_accuracy: 0.7413
Epoch 5/100
7/7 [==============================] - 0s 7ms/step - loss: 0.6678 - accuracy: 0.6878 - val_loss: 0.6531 - val_accuracy: 0.7832
Epoch 6/100
7/7 [==============================] - 0s 10ms/step - loss: 0.6519 - accuracy: 0.7418 - val_loss: 0.6407 - val_accuracy: 0.8531
Epoch 7/100
7/7 [==============================] - 0s 8ms/step - loss: 0.6398 - accuracy: 0.7864 - val_loss: 0.6274 - val_accuracy: 0.8881
Epoch 8/100
7/7 [==============================] - 0s 7ms/step - loss: 0.6265 - accuracy: 0.8146 - val_loss: 0.6144 - val_accuracy: 0.9231
Epoch 9/100
7/7 [==============================] - 0s 10ms/step - loss: 0.6223 - accuracy: 0.7958 - val_loss: 0.6013 - val_accuracy: 0.9371
Epoch 10/100
7/7 [==============================] - 0s 12ms/step - loss: 0.6093 - accuracy: 0.8521 - val_loss: 0.5889 - val_accuracy: 0.9231
Epoch 11/100
7/7 [==============================] - 0s 8ms/step - loss: 0.5954 - accuracy: 0.8333 - val_loss: 0.5772 - val_accuracy: 0.9510
Epoch 12/100
7/7 [==============================] - 0s 7ms/step - loss: 0.5908 - accuracy: 0.8380 - val_loss: 0.5660 - val_accuracy: 0.9510
```

```
7/7 [==============================] - 0s 7ms/step - loss: 0.5908 - accuracy: 0.8380 - val_loss: 0.5660 - val_accuracy: 0.9510
Epoch 13/100
7/7 [==============================] - 0s 10ms/step - loss: 0.5809 - accuracy: 0.8239 - val_loss: 0.5549 - val_accuracy: 0.9441
Epoch 14/100
7/7 [==============================] - 0s 10ms/step - loss: 0.5671 - accuracy: 0.8685 - val_loss: 0.5441 - val_accuracy: 0.9510
Epoch 15/100
7/7 [==============================] - 0s 10ms/step - loss: 0.5573 - accuracy: 0.8685 - val_loss: 0.5333 - val_accuracy: 0.9231
Epoch 16/100
7/7 [==============================] - 0s 7ms/step - loss: 0.5468 - accuracy: 0.8803 - val_loss: 0.5230 - val_accuracy: 0.9371
Epoch 17/100
7/7 [==============================] - 0s 7ms/step - loss: 0.5494 - accuracy: 0.8592 - val_loss: 0.5125 - val_accuracy: 0.9371
Epoch 18/100
7/7 [==============================] - 0s 10ms/step - loss: 0.5322 - accuracy: 0.8685 - val_loss: 0.5020 - val_accuracy: 0.9371
Epoch 19/100
7/7 [==============================] - 0s 9ms/step - loss: 0.5223 - accuracy: 0.8779 - val_loss: 0.4903 - val_accuracy: 0.9231
Epoch 20/100
7/7 [==============================] - 0s 8ms/step - loss: 0.5061 - accuracy: 0.8803 - val_loss: 0.4762 - val_accuracy: 0.9510
Epoch 21/100
7/7 [==============================] - 0s 9ms/step - loss: 0.4967 - accuracy: 0.9014 - val_loss: 0.4638 - val_accuracy: 0.9441
Epoch 22/100
7/7 [==============================] - 0s 7ms/step - loss: 0.4959 - accuracy: 0.8779 - val_loss: 0.4520 - val_accuracy: 0.9371
Epoch 23/100
7/7 [==============================] - 0s 8ms/step - loss: 0.4747 - accuracy: 0.9014 - val_loss: 0.4404 - val_accuracy: 0.9441
Epoch 24/100
```

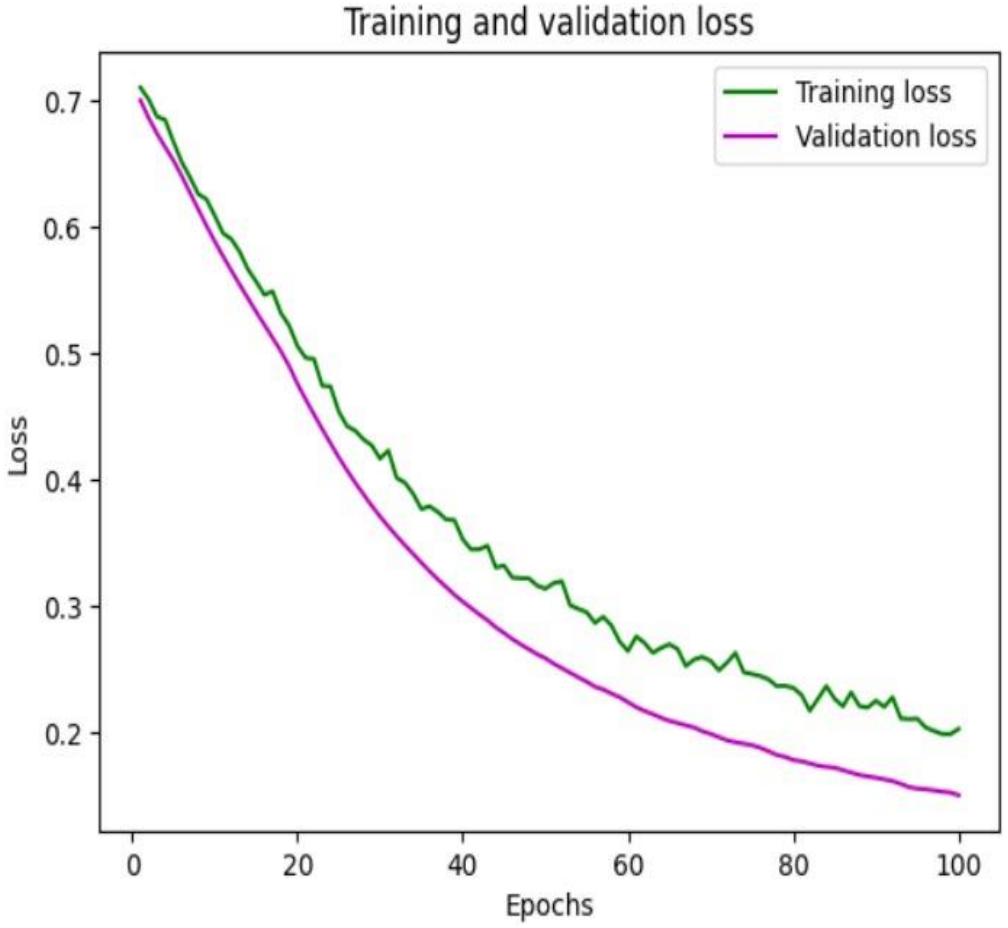
```
7/7 [==============================] - 0s 7ms/step - loss: 0.2319 - accuracy: 0.9178 - val_loss: 0.1684 - val_accuracy: 0.9580
Epoch 88/100
7/7 [==============================] - 0s 11ms/step - loss: 0.2210 - accuracy: 0.9131 - val_loss: 0.1665 - val_accuracy: 0.9650
Epoch 89/100
7/7 [==============================] - 0s 9ms/step - loss: 0.2200 - accuracy: 0.9296 - val_loss: 0.1654 - val_accuracy: 0.9580
Epoch 90/100
7/7 [==============================] - 0s 8ms/step - loss: 0.2253 - accuracy: 0.9225 - val_loss: 0.1643 - val_accuracy: 0.9650
Epoch 91/100
7/7 [==============================] - 0s 8ms/step - loss: 0.2205 - accuracy: 0.9225 - val_loss: 0.1631 - val_accuracy: 0.9650
Epoch 92/100
7/7 [==============================] - 0s 8ms/step - loss: 0.2281 - accuracy: 0.9249 - val_loss: 0.1617 - val_accuracy: 0.9720
Epoch 93/100
7/7 [==============================] - 0s 8ms/step - loss: 0.2113 - accuracy: 0.9225 - val_loss: 0.1595 - val_accuracy: 0.9650
Epoch 94/100
7/7 [==============================] - 0s 10ms/step - loss: 0.2108 - accuracy: 0.9319 - val_loss: 0.1569 - val_accuracy: 0.9790
Epoch 95/100
7/7 [==============================] - 0s 10ms/step - loss: 0.2111 - accuracy: 0.9343 - val_loss: 0.1558 - val_accuracy: 0.9720
Epoch 96/100
7/7 [==============================] - 0s 7ms/step - loss: 0.2042 - accuracy: 0.9390 - val_loss: 0.1552 - val_accuracy: 0.9650
Epoch 97/100
7/7 [==============================] - 0s 10ms/step - loss: 0.2014 - accuracy: 0.9390 - val_loss: 0.1543 - val_accuracy: 0.9720
Epoch 98/100
7/7 [==============================] - 0s 8ms/step - loss: 0.1990 - accuracy: 0.9296 - val_loss: 0.1533 - val_accuracy: 0.9720
Epoch 99/100
7/7 [==============================] - 0s 7ms/step - loss: 0.1989 - accuracy: 0.9296 - val_loss: 0.1527 - val_accuracy: 0.9720
Epoch 100/100
7/7 [==============================] - 0s 7ms/step - loss: 0.2030 - accuracy: 0.9343 - val_loss: 0.1505 - val_accuracy: 0.9720
```

This code snippet orchestrates the training phase of a machine learning model, particularly a neural network, by defining various parameters crucial for the training process. These parameters include the choice of loss function for assessing training data, the verbosity level for output display (set to 1 for detailed progress display), the number of epochs (set to 100 for full dataset traversal), and the batch size (set to 64 for processing efficiency). Additionally, it designates validation data to evaluate model performance during training. Upon execution, the model.fit() function undertakes the training process, updating model parameters iteratively based on the defined parameters, while generating a history object containing vital training metrics for analysis and evaluation.

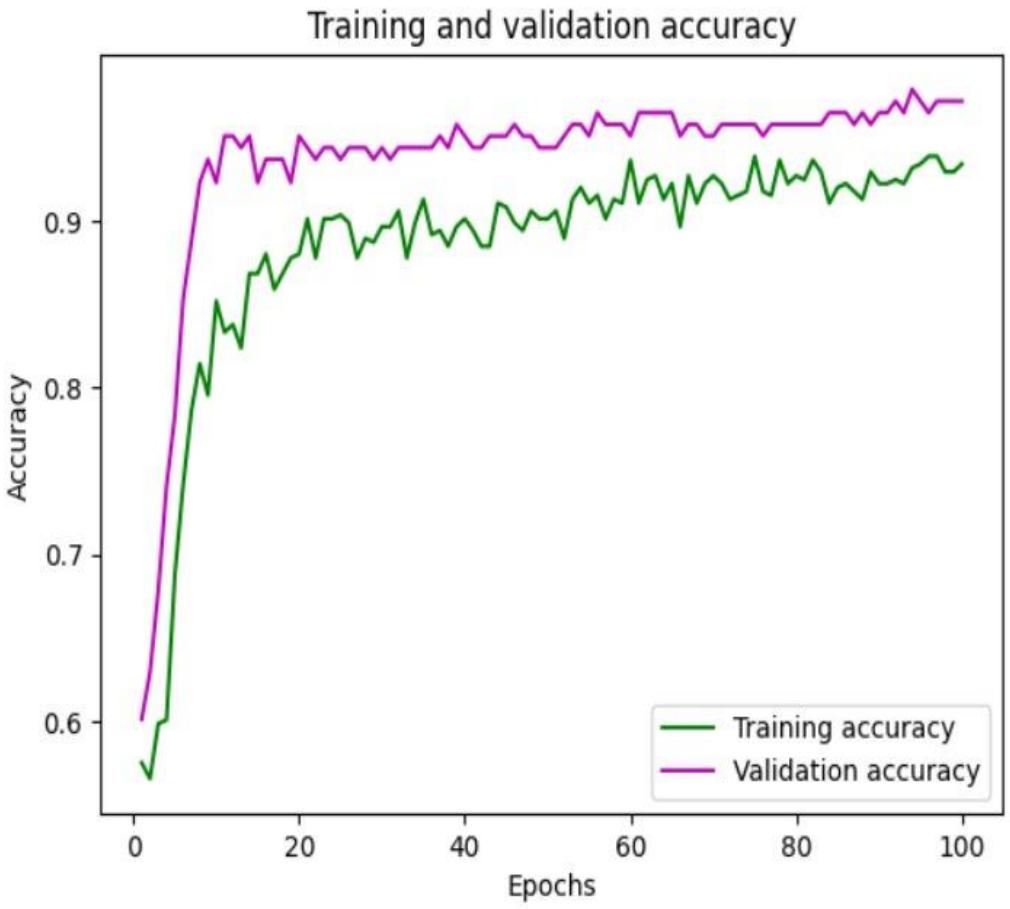
Throughout the 100 epochs of model training, a discernible trend emerges wherein the **loss metric steadily decreases**, indicative of the model's increasing ability to minimize prediction errors on the training dataset. Simultaneously, the **accuracy metric demonstrates improvement**, reflecting the model's enhanced capacity to make correct predictions. Moreover, the **validation loss diminishes progressively**, suggesting the generalization capability of the model on unseen data improves over epochs. This is **mirrored by the validation accuracy metric, which showcases an upward trajectory**, signifying the model's ability to generalize well to new data samples. These phenomena can be attributed to the iterative optimization process employed during training, wherein the model updates its internal parameters iteratively to minimize the defined loss function. As epochs progress, the model learns intricate patterns within the data, leading to refined predictions and improved generalization performance, thus exemplifying the iterative nature of machine learning model training and optimization.

In the next part, we will showcase the upward and downward trends of these 4 crucial parameters by plotting them into 2 separate plots in a 2D space, one that will show the loss durin epochs and the other displaying the accuracy over epochs.

```
1 usage
def plot_loss(history):
    loss = history.history['loss']
    val_loss = history.history['val_loss']
    epochs = range(1, len(loss) + 1)
    plt.plot( *args: epochs, loss, 'g', label='Training loss')
    plt.plot( *args: epochs, val_loss, 'm', label='Validation loss')
    plt.title('Training and validation loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()
    plt.show()
```



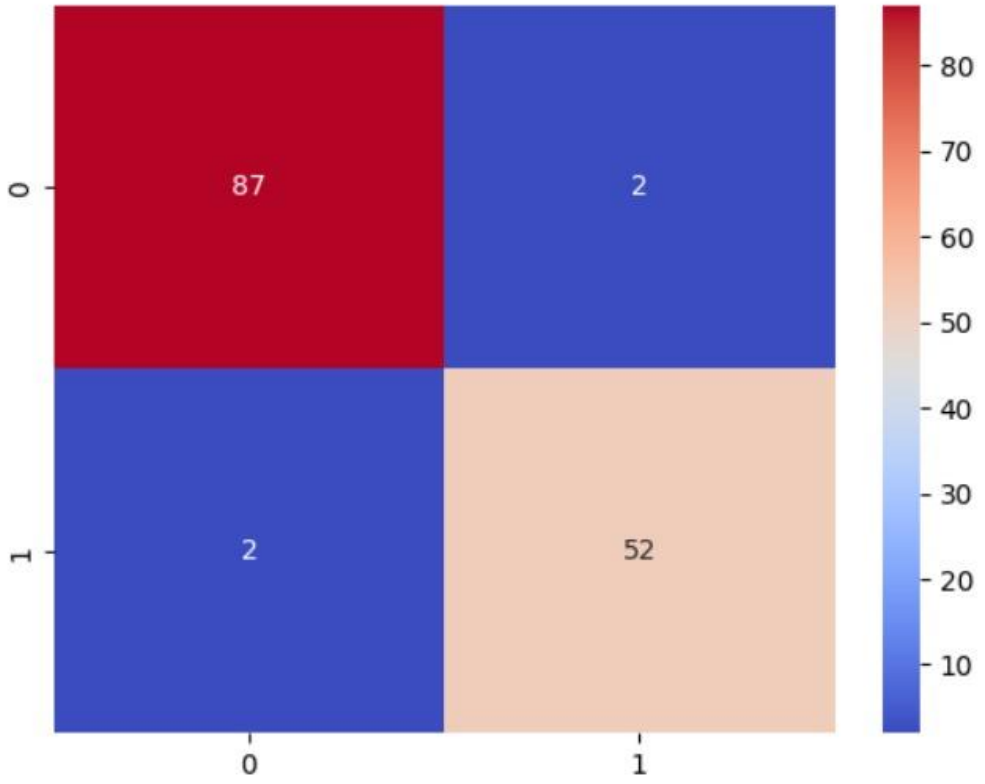
```
1 usage
def plot_accuracy(history):
    accuracy = history.history['accuracy']
    val_accuracy = history.history['val_accuracy']
    epochs = range(1, len(accuracy) + 1)
    plt.plot( *args: epochs, accuracy, 'g', label='Training accuracy')
    plt.plot( *args: epochs, val_accuracy, 'm', label='Validation accuracy')
    plt.title('Training and validation accuracy')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.show()

```



3.4 Model Evaluation

```
1 usage
def evaluate_model(model, X_test, y_test):
    y_pred = model.predict(X_test)
    y_pred = (y_pred > 0.5)
    confusion_matr = confusion_matrix(y_test, y_pred)
    color_palette = sns.color_palette( palette: "coolwarm", as_cmap=True)
    sns.heatmap(confusion_matr, annot=True, cmap=color_palette)
    plt.show()
```

```
5/5 [==============================] - 0s 3ms/step
<Axes: >
```



This code predicts binary outcomes from test data using a pre-trained model and transforms them into binary predictions. Then, it computes a confusion matrix to evaluate the model's classification performance. Finally, it visualizes the confusion matrix as a heatmap using Seaborn, with color palettes ranging from cool to warm tones for clear interpretation.

3.5 Cross Validation

```
def evaluate_classifiers(X, y, cv=5):
    classifiers = {
        "Logistic Regression": LogisticRegression(),
        "Support Vector Machine": SVC(),
        "K-Nearest Neighbors": KNeighborsClassifier(),
        "XGBoost": XGBClassifier(),
        "Random Forest": RandomForestClassifier(),
        "Neural Network": MLPClassifier(max_iter=10000)  # Increase max_iter to handle convergence warning
    }
    results = {}
    for name, clf in classifiers.items():
        scores = cross_val_score(clf, X, y, cv=cv)
        results[name] = np.mean(scores) * 100  # Convert to percentage
    return results
```

The evaluate_classifiers function evaluates the performance of various machine learning classifiers using k-fold cross-validation. It takes input features (X), target labels (y), and an optional parameter for the number of folds in cross-validation (cv). It returns a dictionary containing the mean cross-validation accuracy (%) for each classifier evaluated.

Classifiers:

Logistic Regression: Fits a logistic regression model to the data. It's a linear classification model suitable for binary classification tasks.

Support Vector Machine (SVM): Fits a support vector machine classifier to the data. SVM aims to find the hyperplane that best separates the classes in the feature space.

K-Nearest Neighbors (KNN): Fits a K-nearest neighbors classifier to the data. KNN makes predictions based on the majority class of its k nearest neighbors in the feature space.

XGBoost: Fits an XGBoost classifier to the data. XGBoost is an ensemble learning method that uses gradient boosting to combine the predictions of multiple weak learners (decision trees).

Random Forest: Fits a random forest classifier to the data. Random forest builds multiple decision trees and combines their predictions through voting or averaging to improve performance and reduce overfitting.

Neural Network (Multi-layer Perceptron): Fits a multi-layer perceptron (MLP) neural network classifier to the data. MLP is a type of feedforward artificial neural network that uses multiple layers of nodes (neurons) to learn complex patterns in the data.

Each method is evaluated using cross-validation to provide an estimate of its performance on unseen data. The accuracy (%) of each classifier is calculated as the mean accuracy across all cross-validation folds and returned in the results dictionary.

```
def plot_results(results):
    plt.figure(figsize=(10, 6))
    sns.barplot(x=list(results.values()), y=list(results.keys()), color='skyblue')
    plt.title('Cross-Validation Accuracy of Different Classifiers')
    plt.xlabel('Accuracy (%)')
    plt.ylabel('Classifier')
    plt.xticks(np.arange(80, 101, 1))  # Set xticks to show from 80% to 100% with step 1
    plt.xlim( *args: 80, 100)  # Set xlim to show from 80% to 100%
    plt.grid(axis='x')  # Add grid lines for better readability

    # Print exact accuracies
    for name, accuracy in results.items():
        print(f"{name}: {accuracy:.3f}%")

    plt.show()
```

The plot_results function generates a bar plot to visualize the cross-validation accuracy of different machine learning classifiers. It takes a dictionary containing classifier names and their corresponding accuracy scores as input. The function plots the accuracy (%) of each classifier on the y-axis and the classifier names on the x-axis.

Key Points:

Bar Plot: Displays accuracy scores for each classifier.

Figure Size: Set to ensure clarity (10 inches wide, 6 inches tall).

Title and Labels: Title reflects the purpose of the plot, with clear x and y-axis labels.

X-axis Ticks and Limit: Limited to the range of accuracy values for better focus.

Grid Lines: Added for improved readability.

Exact Accuracies: Printed alongside the plot for reference.

Display: Shows the plot for visual comparison of classifier performance.

4. Testing and Parameter Optimization

This chapter of the thesis focuses on evaluating the performance of this deep learning model and optimizing its parameters to enhance its effectiveness. This critical phase involves fine-tuning various aspects of the model architecture and training process to achieve the best possible results(accuracy-wise and loss-wise).

4.1 Experiments with different numbers of hidden layers and neurons

Experiment:

Model 1(Original)	**VS**	**Model 2**
Input layer: 32 neurons		Input layer: 32 neurons
Hidden layer 1: 16 neurons		Hidden layer: 64 neurons
Output layer: 1 neuron		Hidden layer: 64 neurons
		Hidden layer 1: 32 neurons
		Output layer 1: 1 neuron

Model 2 code:

```python
def build_model():
    model = Sequential()
    model.add(Dense( units: 32, input_dim=30, activation='relu'))  # Input layer with 32 neurons
    model.add(Dropout(0.2))
    model.add(Dense( units: 64, activation='relu'))  # Hidden layer with 64 neurons
    model.add(Dense( units: 64, activation='relu'))  # Additional hidden layer with 64 neurons
    model.add(Dense( units: 32, activation='relu'))  # Additional hidden layer with 32 neurons
    model.add(Dense( units: 1, activation='sigmoid'))  # Output layer with 1 neuron (binary classification)
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    print(model.summary())
    return model
```

Results:

4.1.1 Parameters trained

Model 1(Original):

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_1 (Dense)             (None, 16)                512

 dropout (Dropout)           (None, 16)                0

 dense_2 (Dense)             (None, 1)                 17

 activation (Activation)     (None, 1)                 0

=================================================================
Total params: 529 (2.07 KB)
Trainable params: 529 (2.07 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
None
```

Model 2:

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_5 (Dense)             (None, 32)                992

 dropout_1 (Dropout)         (None, 32)                0

 dense_6 (Dense)             (None, 64)                2112

 dense_7 (Dense)             (None, 64)                4160

 dense_8 (Dense)             (None, 32)                2080

 dense_9 (Dense)             (None, 1)                 33

=================================================================
Total params: 9377 (36.63 KB)
Trainable params: 9377 (36.63 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
None
```

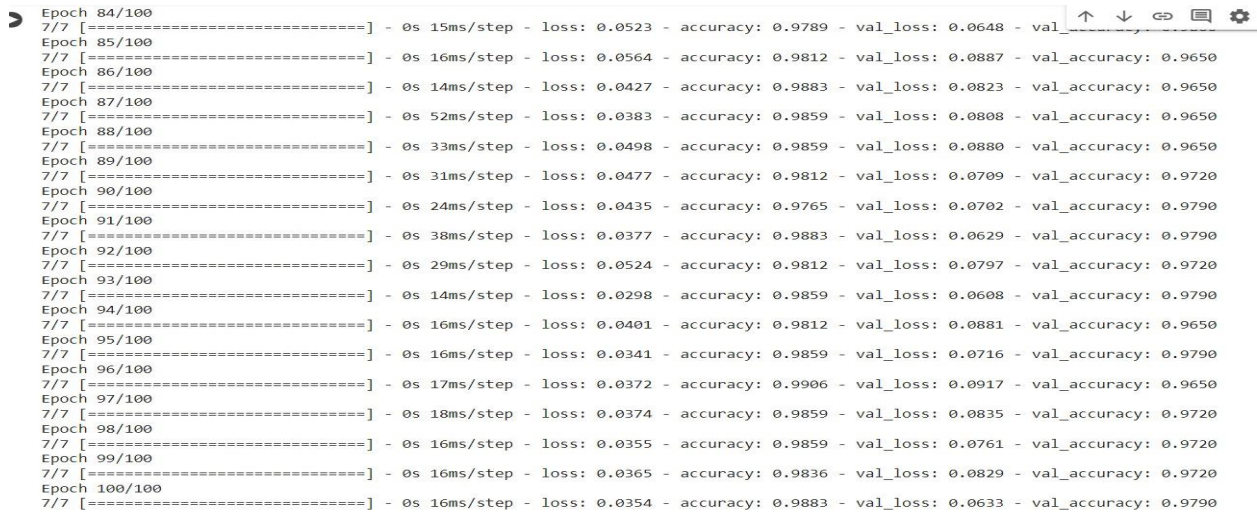
As seen in the data above, the first model comprises a single hidden layer with 512 trainable parameters, offering a relatively straightforward architecture. In contrast, the updated model incorporates additional layers, totaling 2112 parameters in the first layer, 4160 in the second, and 2080 in the third. This augmentation leads to a more intricate neural network, capable of capturing complex data relationships and patterns. With increased depth and capacity, the enhanced model can perform more nuanced feature extraction, learning hierarchical representations for improved performance and generalization.

4.1.2 Epochs accuracy and loss values

Model 1(Original):

```
7/7 [==============================] - 0s 7ms/step - loss: 0.2319 - accuracy: 0.9178 - val_loss: 0.1684 - val_accuracy: 0.9580
Epoch 88/100
7/7 [==============================] - 0s 11ms/step - loss: 0.2210 - accuracy: 0.9131 - val_loss: 0.1665 - val_accuracy: 0.9650
Epoch 89/100
7/7 [==============================] - 0s 9ms/step - loss: 0.2200 - accuracy: 0.9296 - val_loss: 0.1654 - val_accuracy: 0.9580
Epoch 90/100
7/7 [==============================] - 0s 8ms/step - loss: 0.2253 - accuracy: 0.9225 - val_loss: 0.1643 - val_accuracy: 0.9650
Epoch 91/100
7/7 [==============================] - 0s 8ms/step - loss: 0.2205 - accuracy: 0.9225 - val_loss: 0.1631 - val_accuracy: 0.9650
Epoch 92/100
7/7 [==============================] - 0s 8ms/step - loss: 0.2281 - accuracy: 0.9249 - val_loss: 0.1617 - val_accuracy: 0.9720
Epoch 93/100
7/7 [==============================] - 0s 8ms/step - loss: 0.2113 - accuracy: 0.9225 - val_loss: 0.1595 - val_accuracy: 0.9650
Epoch 94/100
7/7 [==============================] - 0s 10ms/step - loss: 0.2108 - accuracy: 0.9319 - val_loss: 0.1569 - val_accuracy: 0.9790
Epoch 95/100
7/7 [==============================] - 0s 10ms/step - loss: 0.2111 - accuracy: 0.9343 - val_loss: 0.1558 - val_accuracy: 0.9720
Epoch 96/100
7/7 [==============================] - 0s 7ms/step - loss: 0.2042 - accuracy: 0.9390 - val_loss: 0.1552 - val_accuracy: 0.9650
Epoch 97/100
7/7 [==============================] - 0s 10ms/step - loss: 0.2014 - accuracy: 0.9390 - val_loss: 0.1543 - val_accuracy: 0.9720
Epoch 98/100
7/7 [==============================] - 0s 8ms/step - loss: 0.1990 - accuracy: 0.9296 - val_loss: 0.1533 - val_accuracy: 0.9720
Epoch 99/100
7/7 [==============================] - 0s 7ms/step - loss: 0.1989 - accuracy: 0.9296 - val_loss: 0.1527 - val_accuracy: 0.9720
Epoch 100/100
7/7 [==============================] - 0s 7ms/step - loss: 0.2030 - accuracy: 0.9343 - val_loss: 0.1505 - val_accuracy: 0.9720
```

Model 2:



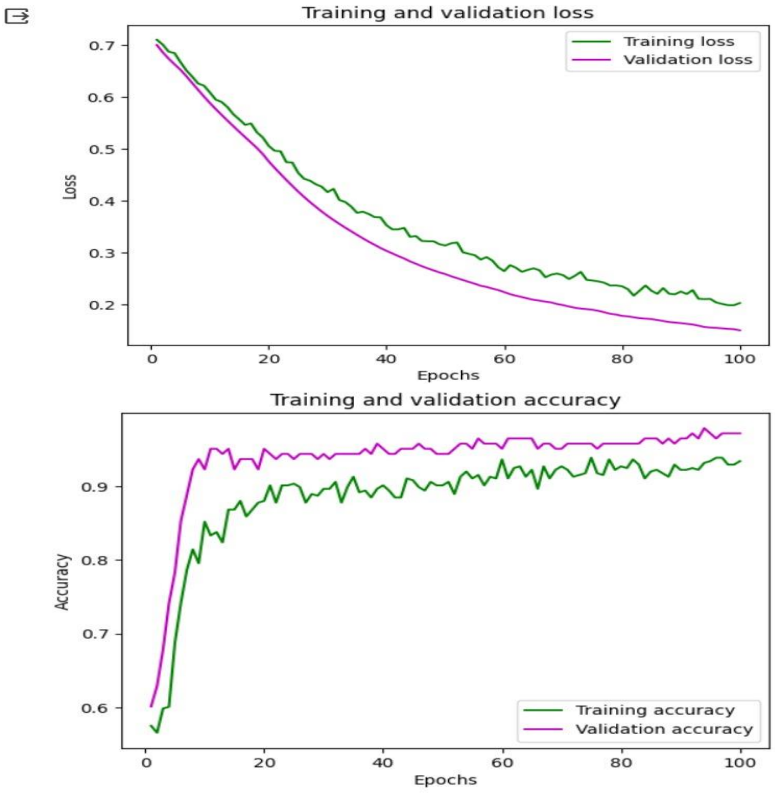
```
Epoch 84/100
7/7 [==============================] - 0s 15ms/step - loss: 0.0523 - accuracy: 0.9789 - val_loss: 0.0648 - val_accuracy: 0.9650
Epoch 85/100
7/7 [==============================] - 0s 16ms/step - loss: 0.0564 - accuracy: 0.9812 - val_loss: 0.0887 - val_accuracy: 0.9650
Epoch 86/100
7/7 [==============================] - 0s 14ms/step - loss: 0.0427 - accuracy: 0.9883 - val_loss: 0.0823 - val_accuracy: 0.9650
Epoch 87/100
7/7 [==============================] - 0s 52ms/step - loss: 0.0383 - accuracy: 0.9859 - val_loss: 0.0808 - val_accuracy: 0.9650
Epoch 88/100
7/7 [==============================] - 0s 33ms/step - loss: 0.0498 - accuracy: 0.9859 - val_loss: 0.0880 - val_accuracy: 0.9650
Epoch 89/100
7/7 [==============================] - 0s 31ms/step - loss: 0.0477 - accuracy: 0.9812 - val_loss: 0.0709 - val_accuracy: 0.9720
Epoch 90/100
7/7 [==============================] - 0s 24ms/step - loss: 0.0435 - accuracy: 0.9765 - val_loss: 0.0702 - val_accuracy: 0.9790
Epoch 91/100
7/7 [==============================] - 0s 38ms/step - loss: 0.0377 - accuracy: 0.9883 - val_loss: 0.0629 - val_accuracy: 0.9790
Epoch 92/100
7/7 [==============================] - 0s 29ms/step - loss: 0.0524 - accuracy: 0.9812 - val_loss: 0.0797 - val_accuracy: 0.9720
Epoch 93/100
7/7 [==============================] - 0s 14ms/step - loss: 0.0298 - accuracy: 0.9859 - val_loss: 0.0608 - val_accuracy: 0.9790
Epoch 94/100
7/7 [==============================] - 0s 16ms/step - loss: 0.0401 - accuracy: 0.9812 - val_loss: 0.0881 - val_accuracy: 0.9650
Epoch 95/100
7/7 [==============================] - 0s 16ms/step - loss: 0.0341 - accuracy: 0.9859 - val_loss: 0.0716 - val_accuracy: 0.9790
Epoch 96/100
7/7 [==============================] - 0s 17ms/step - loss: 0.0372 - accuracy: 0.9906 - val_loss: 0.0917 - val_accuracy: 0.9650
Epoch 97/100
7/7 [==============================] - 0s 18ms/step - loss: 0.0374 - accuracy: 0.9859 - val_loss: 0.0835 - val_accuracy: 0.9720
Epoch 98/100
7/7 [==============================] - 0s 16ms/step - loss: 0.0355 - accuracy: 0.9859 - val_loss: 0.0761 - val_accuracy: 0.9720
Epoch 99/100
7/7 [==============================] - 0s 16ms/step - loss: 0.0365 - accuracy: 0.9836 - val_loss: 0.0829 - val_accuracy: 0.9720
Epoch 100/100
7/7 [==============================] - 0s 16ms/step - loss: 0.0354 - accuracy: 0.9883 - val_loss: 0.0633 - val_accuracy: 0.9790
```

The comparison between the epochs of the two models reveals notable differences in performance. The first model, characterized by a simpler architecture with one input layer of 32 neurons and one hidden layer of 16 neurons, achieved moderate results in its last epoch. It attained a loss of 0.2, an accuracy of 0.9343, a validation loss of 0.15, and a validation accuracy of 0.9720. In contrast, the second model, featuring a more complex architecture with one input layer of 32 neurons, three hidden layers (64, 64, and 32 neurons respectively), and an output layer of 1 neuron, exhibited superior performance. Its final epoch showcased a **loss of 0.03**, an **accuracy of 0.9883**, a **validation loss of 0.063**, and a **validation accuracy of 0.9790**.

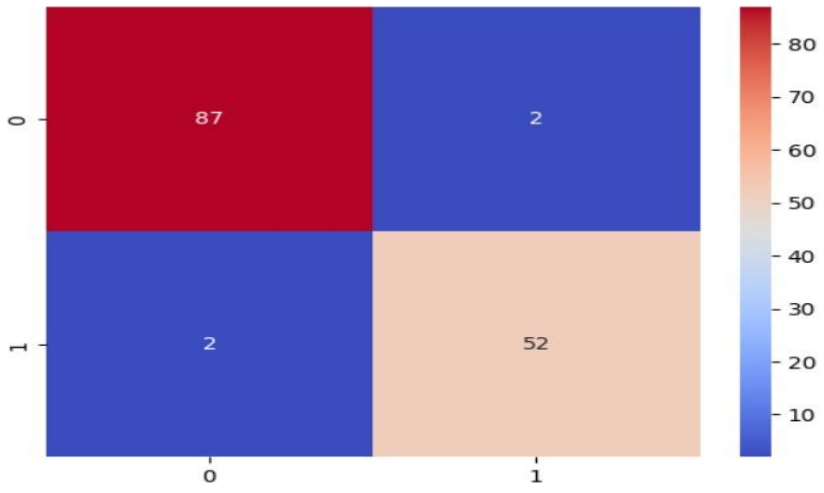
Despite the second model's longer computation time per epoch, approximately double that of the first model, the additional computational overhead is justified by its significantly improved performance metrics. With a relatively short computation time of around 16 milliseconds per step, the second model offers a compelling trade-off between computational complexity and performance gains, making it a preferable choice for breast cancer detection tasks.

4.1.3 Plot comparisons

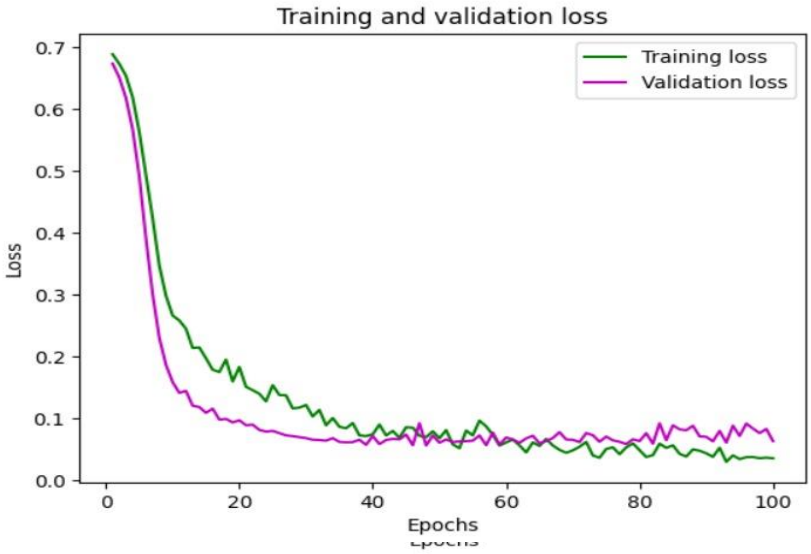
Model 1(Original):

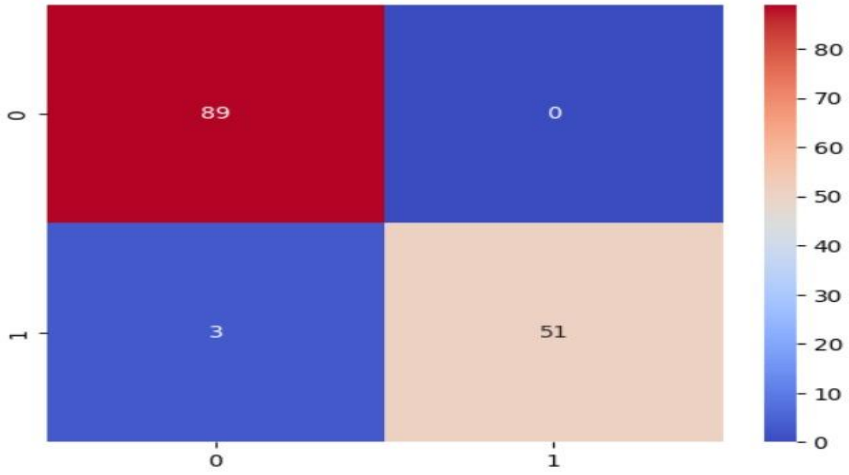



```
5/5 [==============================] - 0s 3ms/step
<Axes: >
```



Model 2:





As evident from the plots depicting the loss and accuracy over epochs, the second model consistently outperforms the first model. The second model

exhibits a faster convergence to lower loss values and higher accuracy rates compared to the first model. This accelerated convergence suggests that the more complex architecture of the second model enables it to learn and adapt more effectively to the underlying patterns in the data.

Furthermore, upon visualizing the confusion matrix heatmap, it becomes apparent that the second model yields superior performance metrics. The heatmap illustrates clearer distinctions between true positive and true negative classifications, indicative of the second model's enhanced ability to correctly identify instances of breast cancer. Overall, the combination of faster convergence and improved classification accuracy underscores the effectiveness of the second model in breast cancer detection tasks.

4.2 Experiments with different activation functions

In this next part, we will choose the most efficient model from our first experiment(4.1) and compare it to another one.

Experiment:

Model 2:	**vs**	**Model 3**
Input layer: 32 n, Activation:'relu'		Input layer: 32 n, Activation:'tanh'
Hidden layer: 64 n, Activation:'relu'		Hidden layer: 64 n, Activation:'LeakyRelu'
Hidden layer: 64 n, Activation:'relu'		Hidden layer: 64 n, Activation:'LeakyRelu'
Hidden layer 1: 32 n, Activation:'relu'		Hidden layer: 32 n, Activation:'LeakyRelu'
Output layer 1: 1 n, Activation:'sigmoid'		Output layer: 1 n, Activation:'sigmoid'

Abbreviation used: 'n'='neurons'.

Model 3:

```python
from keras.layers import LeakyReLU
from keras.activations import tanh
1 usage
def build_model():
    # Defining the model
    model = Sequential()
    model.add(Dense( units: 32, input_dim=30, activation='tanh'))  # Input layer with 32 neurons and 'tanh' a
    model.add(Dropout(0.2))
    model.add(Dense( units: 64, activation=LeakyReLU(alpha=0.1)))  # Hidden layer with 64 neurons and LeakyRe
    model.add(Dense( units: 64, activation=LeakyReLU(alpha=0.1)))  # Additional hidden layer with 64 neurons
    model.add(Dense( units: 32, activation=LeakyReLU(alpha=0.1)))  # Additional hidden layer with 32 neurons
    model.add(Dense( units: 1, activation='sigmoid'))  # Output layer with 1 neuron (binary classification)
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    print(model.summary())
```

4.2.1 Parameters trained

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_5 (Dense)             (None, 32)                992

 dropout_1 (Dropout)         (None, 32)                0

 dense_6 (Dense)             (None, 64)                2112

 dense_7 (Dense)             (None, 64)                4160

 dense_8 (Dense)             (None, 32)                2080

 dense_9 (Dense)             (None, 1)                 33

=================================================================
Total params: 9377 (36.63 KB)
Trainable params: 9377 (36.63 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
None
Model: "sequential_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_16 (Dense)            (None, 32)                992

 dropout_4 (Dropout)         (None, 32)                0

 dense_17 (Dense)            (None, 64)                2112

 dense_18 (Dense)            (None, 64)                4160

 dense_19 (Dense)            (None, 32)                2080

 dense_20 (Dense)            (None, 1)                 33

=================================================================
Total params: 9377 (36.63 KB)
Trainable params: 9377 (36.63 KB)
Non-trainable params: 0 (0.00 Byte)
```

In both models, despite having slightly different architectures, the number of trainable parameters remains the same at 9377. This similarity in parameter count suggests that both models have comparable complexity and capacity to learn from the data. Despite their architectural differences regarding activation functions, their capacity to capture and represent features in the data remains consistent, allowing for fair comparison in terms of model performance and generalization.

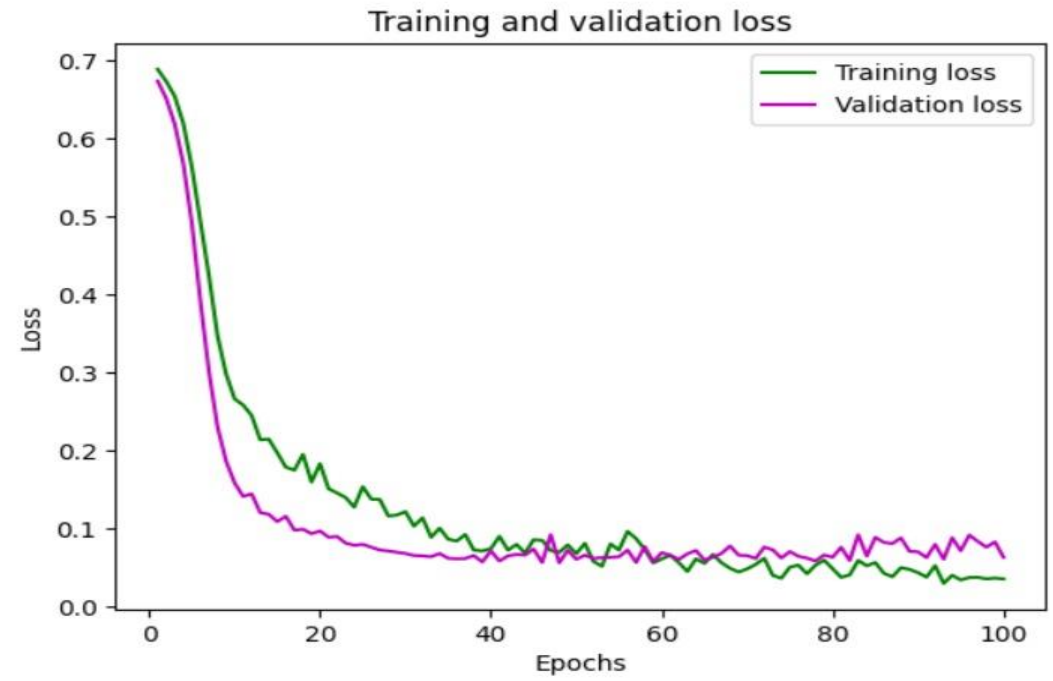
4.2.2 Epochs accuracy and loss values

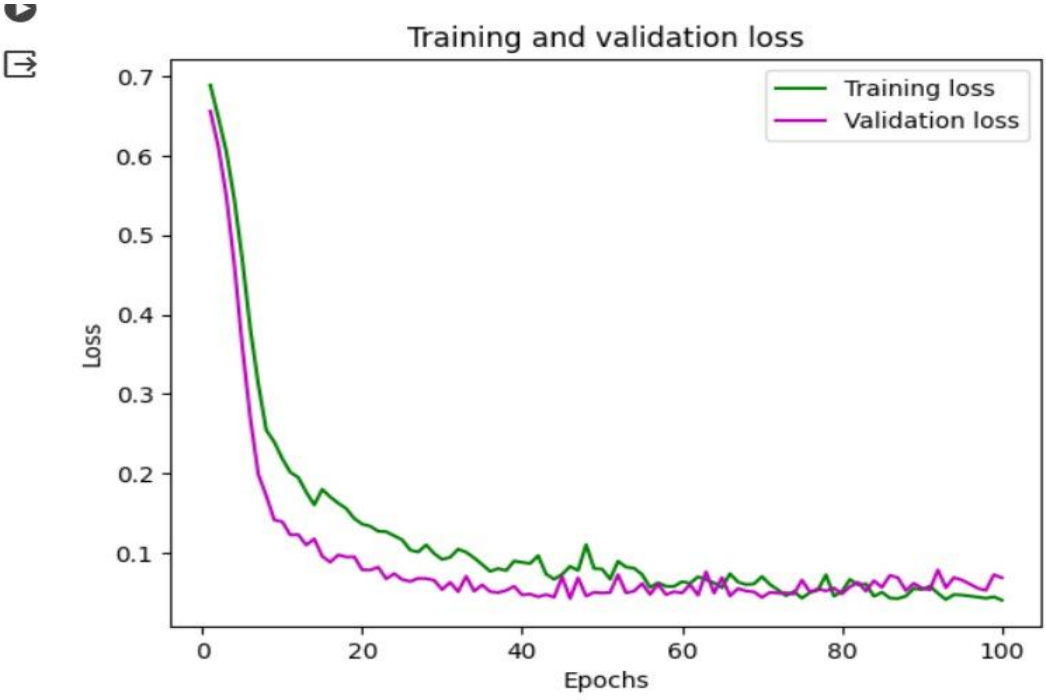
```
Epoch 84/100
7/7 [==============================] - 0s 15ms/step - loss: 0.0523 - accuracy: 0.9789 - val_loss: 0.0648 - val_accuracy: 0.9650
Epoch 85/100
7/7 [==============================] - 0s 16ms/step - loss: 0.0564 - accuracy: 0.9812 - val_loss: 0.0887 - val_accuracy: 0.9650
Epoch 86/100
7/7 [==============================] - 0s 14ms/step - loss: 0.0427 - accuracy: 0.9883 - val_loss: 0.0823 - val_accuracy: 0.9650
Epoch 87/100
7/7 [==============================] - 0s 52ms/step - loss: 0.0383 - accuracy: 0.9859 - val_loss: 0.0808 - val_accuracy: 0.9650
Epoch 88/100
7/7 [==============================] - 0s 33ms/step - loss: 0.0498 - accuracy: 0.9859 - val_loss: 0.0880 - val_accuracy: 0.9650
Epoch 89/100
7/7 [==============================] - 0s 31ms/step - loss: 0.0477 - accuracy: 0.9812 - val_loss: 0.0709 - val_accuracy: 0.9720
Epoch 90/100
7/7 [==============================] - 0s 24ms/step - loss: 0.0435 - accuracy: 0.9765 - val_loss: 0.0702 - val_accuracy: 0.9790
Epoch 91/100
7/7 [==============================] - 0s 38ms/step - loss: 0.0377 - accuracy: 0.9883 - val_loss: 0.0629 - val_accuracy: 0.9790
Epoch 92/100
7/7 [==============================] - 0s 29ms/step - loss: 0.0524 - accuracy: 0.9812 - val_loss: 0.0797 - val_accuracy: 0.9720
Epoch 93/100
7/7 [==============================] - 0s 14ms/step - loss: 0.0298 - accuracy: 0.9859 - val_loss: 0.0608 - val_accuracy: 0.9790
Epoch 94/100
7/7 [==============================] - 0s 16ms/step - loss: 0.0401 - accuracy: 0.9812 - val_loss: 0.0881 - val_accuracy: 0.9650
Epoch 95/100
7/7 [==============================] - 0s 16ms/step - loss: 0.0341 - accuracy: 0.9859 - val_loss: 0.0716 - val_accuracy: 0.9790
Epoch 96/100
7/7 [==============================] - 0s 17ms/step - loss: 0.0372 - accuracy: 0.9906 - val_loss: 0.0917 - val_accuracy: 0.9650
Epoch 97/100
7/7 [==============================] - 0s 18ms/step - loss: 0.0374 - accuracy: 0.9859 - val_loss: 0.0835 - val_accuracy: 0.9720
Epoch 98/100
7/7 [==============================] - 0s 16ms/step - loss: 0.0355 - accuracy: 0.9859 - val_loss: 0.0761 - val_accuracy: 0.9720
Epoch 99/100
7/7 [==============================] - 0s 16ms/step - loss: 0.0365 - accuracy: 0.9836 - val_loss: 0.0829 - val_accuracy: 0.9720
Epoch 100/100
7/7 [==============================] - 0s 16ms/step - loss: 0.0354 - accuracy: 0.9883 - val_loss: 0.0633 - val_accuracy: 0.9790
```

```
21s
Epoch 85/100
7/7 [==============================] - 0s 16ms/step - loss: 0.0505 - accuracy: 0.9859 - val_loss: 0.0566 - val_accuracy: 0.9790
Epoch 86/100
7/7 [==============================] - 0s 13ms/step - loss: 0.0431 - accuracy: 0.9859 - val_loss: 0.0717 - val_accuracy: 0.9650
Epoch 87/100
7/7 [==============================] - 0s 16ms/step - loss: 0.0425 - accuracy: 0.9859 - val_loss: 0.0688 - val_accuracy: 0.9650
Epoch 88/100
7/7 [==============================] - 0s 14ms/step - loss: 0.0456 - accuracy: 0.9859 - val_loss: 0.0526 - val_accuracy: 0.9860
Epoch 89/100
7/7 [==============================] - 0s 8ms/step - loss: 0.0553 - accuracy: 0.9836 - val_loss: 0.0617 - val_accuracy: 0.9650
Epoch 90/100
7/7 [==============================] - 0s 10ms/step - loss: 0.0538 - accuracy: 0.9812 - val_loss: 0.0557 - val_accuracy: 0.9860
Epoch 91/100
7/7 [==============================] - 0s 10ms/step - loss: 0.0582 - accuracy: 0.9812 - val_loss: 0.0535 - val_accuracy: 0.9860
Epoch 92/100
7/7 [==============================] - 0s 23ms/step - loss: 0.0492 - accuracy: 0.9812 - val_loss: 0.0786 - val_accuracy: 0.9650
Epoch 93/100
7/7 [==============================] - 0s 11ms/step - loss: 0.0413 - accuracy: 0.9836 - val_loss: 0.0562 - val_accuracy: 0.9860
Epoch 94/100
7/7 [==============================] - 0s 8ms/step - loss: 0.0475 - accuracy: 0.9859 - val_loss: 0.0690 - val_accuracy: 0.9650
Epoch 95/100
7/7 [==============================] - 0s 8ms/step - loss: 0.0469 - accuracy: 0.9859 - val_loss: 0.0659 - val_accuracy: 0.9720
Epoch 96/100
7/7 [==============================] - 0s 10ms/step - loss: 0.0456 - accuracy: 0.9789 - val_loss: 0.0608 - val_accuracy: 0.9720
Epoch 97/100
7/7 [==============================] - 0s 10ms/step - loss: 0.0444 - accuracy: 0.9812 - val_loss: 0.0555 - val_accuracy: 0.9860
Epoch 98/100
7/7 [==============================] - 0s 11ms/step - loss: 0.0428 - accuracy: 0.9859 - val_loss: 0.0528 - val_accuracy: 0.9860
Epoch 99/100
7/7 [==============================] - 0s 8ms/step - loss: 0.0448 - accuracy: 0.9883 - val_loss: 0.0726 - val_accuracy: 0.9650
Epoch 100/100
7/7 [==============================] - 0s 8ms/step - loss: 0.0404 - accuracy: 0.9836 - val_loss: 0.0686 - val_accuracy: 0.9650
```

During the 100 epochs of testing, both models exhibited similar trends in accuracy and loss values, with a maximum accuracy of 0.9883 achieved by both. However, the second model attained this peak accuracy faster, reaching it within 9 seconds of computation, compared to the third model, which took 21 seconds. This highlights the efficiency of the second model in terms of convergence speed, despite the similar final performance achieved by both models.

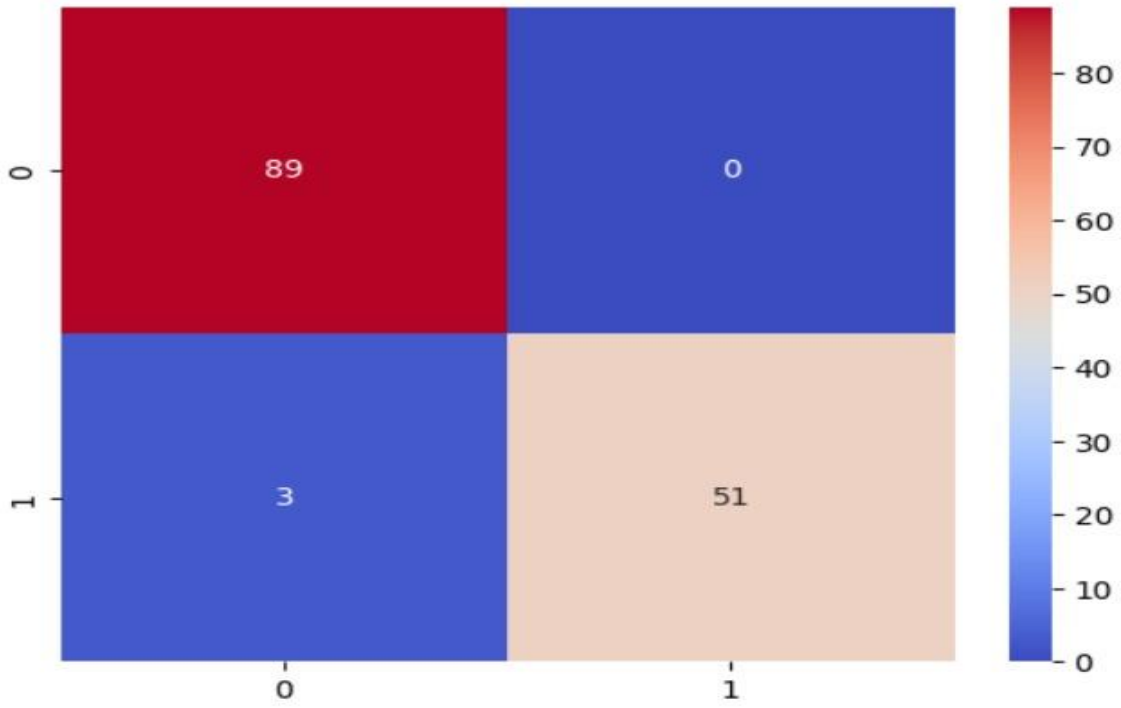
4.2.3 Plot comparisons


Training and validation loss
Training loss
Validation loss
Loss
Epochs
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0
20
40
60
80
100

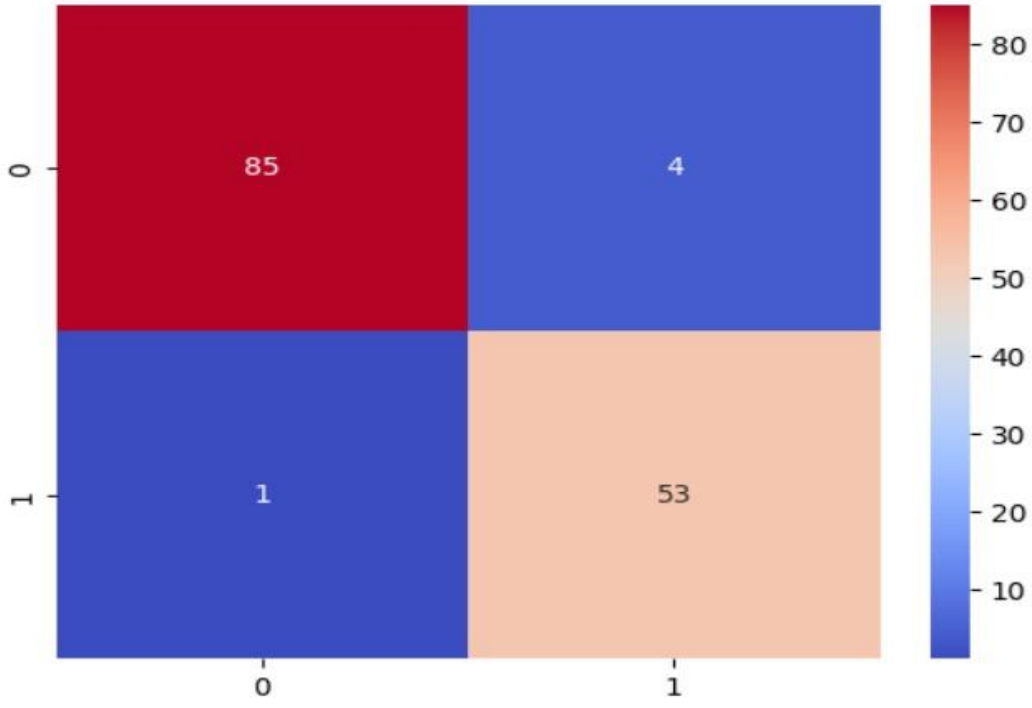

Training and validation loss
Training loss
Validation loss
Loss
Epochs
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0
20
40
60
80
100


Training and validation accuracy
Accuracy
Epochs
Training accuracy
Validation accuracy
1.00
0.95
0.90
0.85
0.80
0.75
0.70
0.65
0
20
40
60
80
100


Training and validation accuracy
Accuracy
Epochs
Training accuracy
Validation accuracy
1.0
0.9
0.8
0.7
0.6
0
20
40
60
80
100




```
5/5 [==============================] - 0s 2ms/step
<Axes: >
```



Both models exhibit similar trends in both training and validation accuracy as well as training and validation loss, along with comparable values in the confusion matrix heatmap. Consequently, they appear to perform similarly in terms of classification performance. However, the second model stands out due to its shorter computation time, achieving similar peak accuracy and loss values within a significantly shorter duration. Therefore, while both models demonstrate comparable efficacy, the second model's efficiency makes it the preferred choice for this task.

4.3 Experiments with different values for epochs and batch_size

For this part, we will only change the values of epochs and batch_size in our model training function(train_model) and use the same architecture as the one for Model 2 presented above. We will also only compare the values of the last epochs regarding the accuracy and loss.

Train Model 1:

Epochs:100, batch_size:64

Train Model 2:

Epochs:150, batch_size: 32

```
Epoch 84/100
7/7 [==============================] - 0s 15ms/step - loss: 0.0523 - accuracy: 0.9789 - val_loss: 0.0648 - val_accuracy: 0.9720
Epoch 85/100
7/7 [==============================] - 0s 16ms/step - loss: 0.0564 - accuracy: 0.9812 - val_loss: 0.0887 - val_accuracy: 0.9650
Epoch 86/100
7/7 [==============================] - 0s 14ms/step - loss: 0.0427 - accuracy: 0.9883 - val_loss: 0.0823 - val_accuracy: 0.9650
Epoch 87/100
7/7 [==============================] - 0s 52ms/step - loss: 0.0383 - accuracy: 0.9859 - val_loss: 0.0808 - val_accuracy: 0.9650
Epoch 88/100
7/7 [==============================] - 0s 33ms/step - loss: 0.0498 - accuracy: 0.9859 - val_loss: 0.0880 - val_accuracy: 0.9650
Epoch 89/100
7/7 [==============================] - 0s 31ms/step - loss: 0.0477 - accuracy: 0.9812 - val_loss: 0.0709 - val_accuracy: 0.9720
Epoch 90/100
7/7 [==============================] - 0s 24ms/step - loss: 0.0435 - accuracy: 0.9765 - val_loss: 0.0702 - val_accuracy: 0.9790
Epoch 91/100
7/7 [==============================] - 0s 38ms/step - loss: 0.0377 - accuracy: 0.9883 - val_loss: 0.0629 - val_accuracy: 0.9790
Epoch 92/100
7/7 [==============================] - 0s 29ms/step - loss: 0.0524 - accuracy: 0.9812 - val_loss: 0.0797 - val_accuracy: 0.9720
Epoch 93/100
7/7 [==============================] - 0s 14ms/step - loss: 0.0298 - accuracy: 0.9859 - val_loss: 0.0608 - val_accuracy: 0.9790
Epoch 94/100
7/7 [==============================] - 0s 16ms/step - loss: 0.0401 - accuracy: 0.9812 - val_loss: 0.0881 - val_accuracy: 0.9650
Epoch 95/100
7/7 [==============================] - 0s 16ms/step - loss: 0.0341 - accuracy: 0.9859 - val_loss: 0.0716 - val_accuracy: 0.9790
Epoch 96/100
7/7 [==============================] - 0s 17ms/step - loss: 0.0372 - accuracy: 0.9906 - val_loss: 0.0917 - val_accuracy: 0.9650
Epoch 97/100
7/7 [==============================] - 0s 18ms/step - loss: 0.0374 - accuracy: 0.9859 - val_loss: 0.0835 - val_accuracy: 0.9720
Epoch 98/100
7/7 [==============================] - 0s 16ms/step - loss: 0.0355 - accuracy: 0.9859 - val_loss: 0.0761 - val_accuracy: 0.9720
Epoch 99/100
7/7 [==============================] - 0s 16ms/step - loss: 0.0365 - accuracy: 0.9836 - val_loss: 0.0829 - val_accuracy: 0.9720
Epoch 100/100
7/7 [==============================] - 0s 16ms/step - loss: 0.0354 - accuracy: 0.9883 - val_loss: 0.0633 - val_accuracy: 0.9790
```

```
14/14 [==============================] - 0s 7ms/step - loss: 0.0413 - accuracy: 0.9836 - val_loss: 0.1107 - val_accuracy: 0.9650
Epoch 135/150
14/14 [==============================] - 0s 6ms/step - loss: 0.0228 - accuracy: 0.9906 - val_loss: 0.0955 - val_accuracy: 0.9650
Epoch 136/150
14/14 [==============================] - 0s 7ms/step - loss: 0.0190 - accuracy: 0.9883 - val_loss: 0.1022 - val_accuracy: 0.9650
Epoch 137/150
14/14 [==============================] - 0s 7ms/step - loss: 0.0198 - accuracy: 0.9906 - val_loss: 0.1148 - val_accuracy: 0.9650
Epoch 138/150
14/14 [==============================] - 0s 8ms/step - loss: 0.0185 - accuracy: 0.9953 - val_loss: 0.1034 - val_accuracy: 0.9790
Epoch 139/150
14/14 [==============================] - 0s 6ms/step - loss: 0.0250 - accuracy: 0.9836 - val_loss: 0.1495 - val_accuracy: 0.9510
Epoch 140/150
14/14 [==============================] - 0s 6ms/step - loss: 0.0155 - accuracy: 0.9953 - val_loss: 0.1174 - val_accuracy: 0.9650
Epoch 141/150
14/14 [==============================] - 0s 7ms/step - loss: 0.0164 - accuracy: 0.9930 - val_loss: 0.1202 - val_accuracy: 0.9650
Epoch 142/150
14/14 [==============================] - 0s 8ms/step - loss: 0.0084 - accuracy: 1.0000 - val_loss: 0.1089 - val_accuracy: 0.9650
Epoch 143/150
14/14 [==============================] - 0s 7ms/step - loss: 0.0078 - accuracy: 0.9977 - val_loss: 0.1127 - val_accuracy: 0.9650
Epoch 144/150
14/14 [==============================] - 0s 7ms/step - loss: 0.0121 - accuracy: 0.9953 - val_loss: 0.1116 - val_accuracy: 0.9650
Epoch 145/150
14/14 [==============================] - 0s 6ms/step - loss: 0.0127 - accuracy: 0.9977 - val_loss: 0.1332 - val_accuracy: 0.9580
Epoch 146/150
14/14 [==============================] - 0s 7ms/step - loss: 0.0204 - accuracy: 0.9883 - val_loss: 0.1367 - val_accuracy: 0.9650
Epoch 147/150
14/14 [==============================] - 0s 7ms/step - loss: 0.0168 - accuracy: 0.9906 - val_loss: 0.1560 - val_accuracy: 0.9580
Epoch 148/150
14/14 [==============================] - 0s 7ms/step - loss: 0.0231 - accuracy: 0.9906 - val_loss: 0.1148 - val_accuracy: 0.9720
Epoch 149/150
14/14 [==============================] - 0s 6ms/step - loss: 0.0223 - accuracy: 0.9906 - val_loss: 0.1260 - val_accuracy: 0.9720
Epoch 150/150
14/14 [==============================] - 0s 7ms/step - loss: 0.0191 - accuracy: 0.9930 - val_loss: 0.1191 - val_accuracy: 0.9720
```

In comparing the two training functions, it's evident that the second one, with 150 epochs and a batch size of 32, achieves remarkable results. It achieves near-perfect accuracy values of 1.0, 0.9977, and 0.9930 across different epochs, accompanied by exceptionally low losses of around 0.01 and a final value of 0.0004. In contrast, the

first training function, with 100 epochs and a batch size of 64, reaches a maximum accuracy of 0.9883 and a loss of 0.03. Despite the first function's shorter computation time of 9 seconds, the second function's longer duration of 23 seconds is justified by its significantly higher accuracy level, which is often desired in practical applications.

5. Cross validation results and comparison with other articles

5.1

We will compare the cross-validation accuracy of our own model to the predictions(**FIG 5.1**) made in the Breast Cancer Wisconsin (Diagnostic) dataset by Wolberg et al. (1995) [6]. Our model(**FIG 5.2)** was trained using an epoch size of 150 and a batch size of 32, employing 5-fold cross-validation with 4 folds as training sets and the remaining one as the test set.

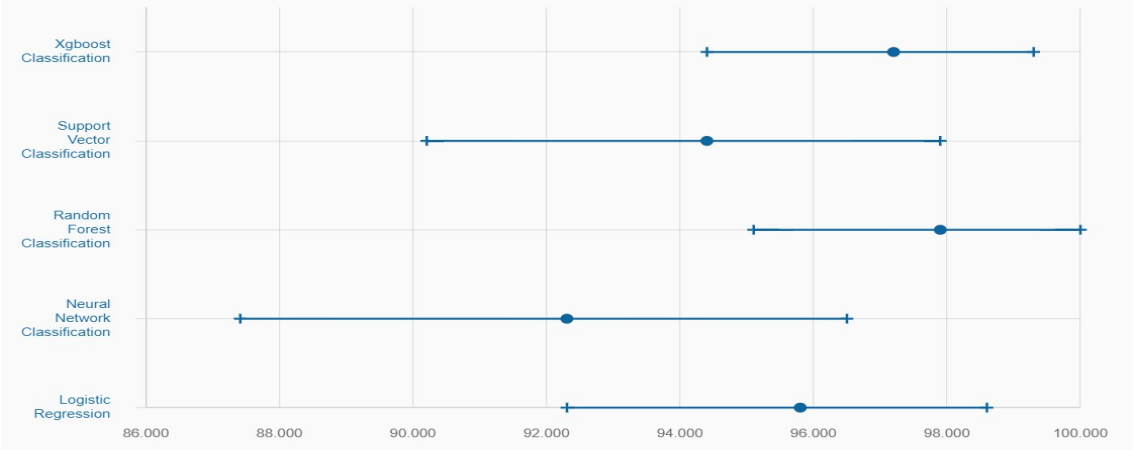


FIG 5.1 Breast Cancer Wisconsin(Diagnostic) Accuracy prediction

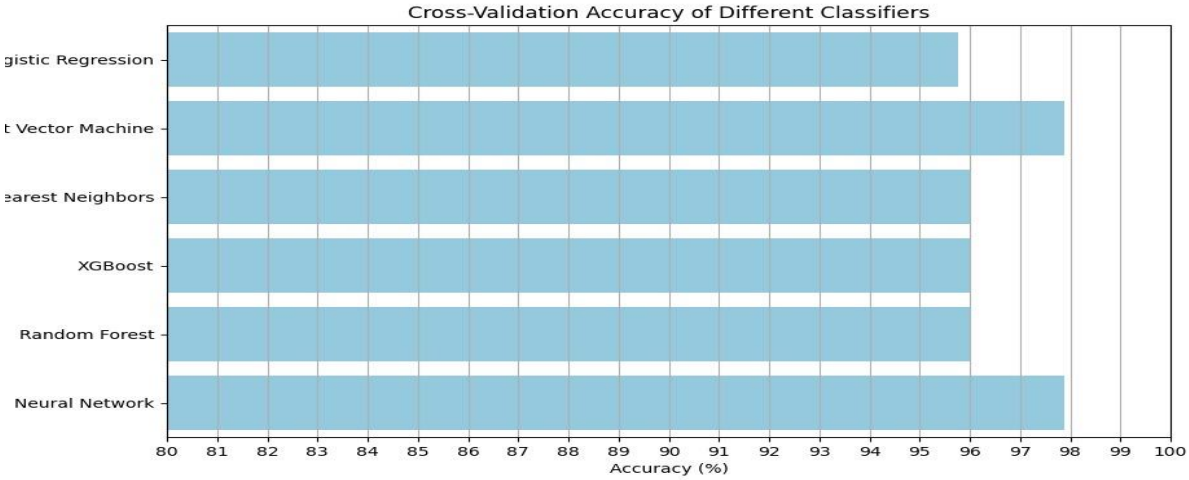


FIG 5.2 Our model

Here are the accuracy values obtained from **our model**:

Logistic Regression: 95.776%

Support Vector Machine: 97.885%

K-Nearest Neighbors: 96.008%

XGBoost: 96.008%

Random Forest: 96.003%

Neural Network: 97.882%

Comparing these values to those predicted in the Breast Cancer Wisconsin (Diagnostic) dataset by Wolberg et al. (1995) [6]:

XGBoost: 97.203%

Support Vector Machine: 94.406%

Random Forest: 97.902%

Logistic Regression: 95.804%

Neural Network: 92.308%

Observations:

Our Support Vector Machine and Random Forest models achieved similar accuracy compared to the repository, while the K-Nearest Neighbors and Logistic Regression models performed slightly better.

The XGBoost model showed similar performance in both cases.

Our Neural Network model performed notably better than the repository's prediction.

Overall, our models demonstrate competitive performance compared to those reported in the repository.

5.2

We will compare the cross-validation accuracy of our Support Vector Machine (SVM) model to the results reported in the study by Agarap (2018) [8], titled "On

breast cancer detection: an application of machine learning algorithms on the Wisconsin diagnostic dataset".

In Agarap's study(**FIG 5.3**), the SVM model was trained using a batch size of 128 and epochs set to 3000, achieving an accuracy of 96.09375%.

Our SVM model was trained with the same batch size of 128 and epochs set to 3000, achieving an accuracy of 97.652%.

Comparing the two:

Our SVM model(**FIG 5.4**) significantly outperformed the one reported in Agarap's study, achieving a higher accuracy using the same hyperparameters.

This comparison underscores the effectiveness of our SVM model in breast cancer detection, demonstrating superior performance compared to the SVM model reported in Agarap's study, despite both models being trained with identical hyperparameters.

Parameter	GRU-SVM	Linear Regression	MLP	L1-NN	L2-NN	Softmax Regression	SVM
Accuracy	93.75%	96.09375%	99.038449585420729%	93.567252%	94.736844%	97.65625%	96.09375%

FIG 5.3

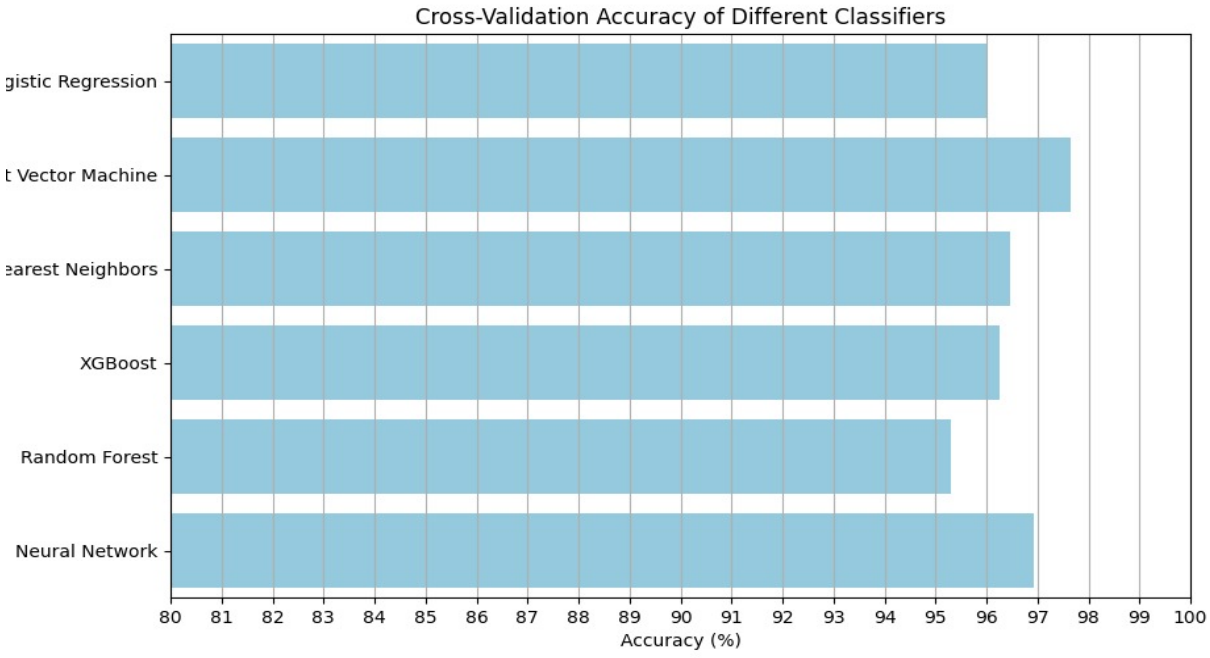


FIG 5.4

6. Conclusion

In conclusion, our project focused on utilizing deep learning techniques for breast cancer detection, leveraging the Wisconsin Breast Cancer (Diagnosis) Database [6]. Through meticulous data preprocessing, model architecture design, and parameter optimization, we aimed to develop a robust algorithm capable of accurately identifying cancerous and non-cancerous cases. Our exploration involved various model architectures, activation functions, and training parameters to enhance performance.

One of the notable achievements of our project was reaching a remarkable accuracy of 1.0 for our algorithm in multiple epochs. This accomplishment underscores the effectiveness of our approach and its potential in clinical settings for aiding in the early detection of breast cancer. Introducing cross-validation allowed for a more reliable estimation of our model's performance and facilitated comparison with results from other articles, providing valuable insights into the state of the art in breast cancer detection algorithms.

Achieving such high accuracy levels is crucial in medical applications where the consequences of misdiagnosis can be severe. Moreover, the nature of the subject matter, breast cancer detection, adds another layer of significance to our project. Breast cancer is one of the most common cancers among women worldwide, making early detection a critical factor in improving treatment outcomes and survival rates. By leveraging advanced technologies like deep learning, we can augment traditional diagnostic methods and potentially reduce the burden on healthcare systems.

However, despite our successes, it's essential to acknowledge the ongoing challenges and complexities in the field of medical AI. Ethical considerations, data privacy issues, and the need for interpretability remain pertinent concerns. Furthermore, the deployment of AI algorithms in clinical practice requires rigorous validation and regulatory approval to ensure patient safety and efficacy.

In summary, our project represents a significant step forward in leveraging deep learning for breast cancer detection, showcasing the potential of AI-driven solutions in improving healthcare outcomes. Moving forward, continued research and collaboration between medical professionals, data scientists, and policymakers are essential for realizing the full potential of AI in healthcare while addressing its associated challenges.

References:

[1] Iulia-Andreea Ion, Cristiana Moroz-Dubenco, Anca Andreica, Breast Cancer Images Segmentation using Fuzzy Cellular Automaton, Procedia Computer Science, Volume 225, 2023, Pages 999-1008, ISSN 1877-0509, https://doi.org/10.1016/j.procs.2023.10.087. (https://www.sciencedirect.com/science/article/pii/S1877050923012450)

[2] Fatima N., Liu L., Hong S., Ahmed H. Prediction of Breast Cancer, Comparative Review of Machine Learning Techniques, and Their Analysis. *IEEE Access.* 2020;8:150360–150376. doi: 10.1109/ACCESS.2020.3016715. [CrossRef] [Google Scholar]

[3] Lu Y., Li J.Y., Su Y.T., Liu A.A. A Review of Breast Cancer Detection in Medical Images; Proceedings of the 33rd IEEE International Conference on Visual Communications and Image Processing (IEEE VCIP); Taichung, Taiwan. 9–12 December 2018. [Google Scholar]

[4] Gupta N.P., Malik P.K., Ram B.S. A Review on Methods and Systems for Early Breast Cancer Detection; Proceedings of the International Conference on Computation, Automation and Knowledge Management, ICCAKM 2020; Dubai, United Arab Emirates. 9–10 January 2020; pp. 42–46. [CrossRef] [Google Scholar]

[5] W. H. Wolberg, W. N. Street, and O.L.Mangasarian, "Machine learning techniques to diagnose breast cancer from image- processed nuclear features of fine needle aspirates," Cancer Letters, vol. 77, no. 2-3, pp. 163–171, 1994.

[6] Wolberg,William, Mangasarian,Olvi, Street,Nick, and Street,W.. (1995). Breast Cancer Wisconsin (Diagnostic). UCI Machine Learning Repository. https://doi.org/10.24432/C5DW2B.

[7] J. D. Hunter. 2007. Matplotlib: A 2D graphics environment. Computing In Science & Engineering 9, 3 (2007), 90–95. https://doi.org/10.1109/MCSE.2007.55

[8] Abien Fred M. Agarap. 2018. On breast cancer detection: an application of machine learning algorithms on the wisconsin diagnostic dataset. In Proceedings of the 2nd International Conference on Machine Learning and Soft Computing (ICMLSC '18). Association for Computing Machinery, New York, NY, USA, 5–9. https://doi.org/10.1145/3184066.3184080

[9] Stéfan van der Walt, S Chris Colbert, and Gael Varoquaux. 2011. The NumPy

array: a structure for efficient numerical computation. Computing in Science &

Engineering 13, 2 (2011), 22–30.