Activity 1.2 : Training Neural Networks

Barcelona, Jacob Seth S.

Instructor: Engr. Roman Richard

Objective(s):

This activity aims to demonstrate how to train neural networks using keras

Intended Learning Outcomes (ILOs):

- Demonstrate how to build and train neural networks
- Demonstrate how to evaluate and plot the model using training and validation loss

Resources:

- Jupyter Notebook

CI Pima Diabetes Dataset

- pima-indians-diabetes.csv

Procedures

Load the necessary libraries

In [ ]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import confusion_matrix, precision_recall_curve, roc_auc_s
from sklearn.ensemble import RandomForestClassifier

import seaborn as sns

%matplotlib inline
```

In [ ]:
```
## Import Keras objects for Deep Learning

from keras.models  import Sequential
from keras.layers import Input, Dense, Flatten, Dropout, BatchNormalization
from keras.optimizers import Adam, SGD, RMSprop
```

Load the dataset

In [ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

In [ ]:
```
filepath = "/content/drive/MyDrive/Datasci3/pima-indians-diabetes.csv"
names = ["times_pregnant", "glucose_tolerance_test", "blood_pressure", "skin_th
         "bmi", "pedigree_function", "age", "has_diabetes"]
diabetes_df = pd.read_csv(filepath, names=names)
```

Check the top 5 samples of the data

In [ ]:
```
print(diabetes_df.shape)
diabetes_df.sample(5)
```

```
(768, 9)
```

Out[5]:

	times_pregnant	glucose_tolerance_test	blood_pressure	skin_thickness	insulin	bmi	pedigı
565	2	95	54	14	88	26.1	
656	2	101	58	35	90	21.8	
504	3	96	78	39	0	37.3	
186	8	181	68	36	495	30.1	
271	2	108	62	32	56	25.2	

In [ ]:
```
diabetes_df.dtypes
```

Out[6]:
```
times_pregnant              int64
glucose_tolerance_test      int64
blood_pressure              int64
skin_thickness              int64
insulin                     int64
bmi                       float64
pedigree_function         float64
age                         int64
has_diabetes                int64
dtype: object
```

In [ ]:
```
X = diabetes_df.iloc[:, :-1].values
y = diabetes_df["has_diabetes"].values
```

Split the data to Train, and Test (75%, 25%)

In [ ]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, randc
```

In [ ]:
```
np.mean(y), np.mean(1-y)
```

Out[9]:
```
(0.3489583333333333, 0.6510416666666666)
```

Build a single hidden layer neural network using 12 nodes. Use the sequential model with single layer network and input shape to 8.

Normalize the data

In [ ]:
```
normalizer = StandardScaler()
X_train_norm = normalizer.fit_transform(X_train)
X_test_norm = normalizer.transform(X_test)
```

Define the model:

- Input size is 8-dimensional
- 1 hidden layer, 12 hidden nodes, sigmoid activation
- Final layer with one node and sigmoid activation (standard for binary classification)

In [ ]:
```
model  = Sequential([
    Dense(12, input_shape=(8,), activation="relu"),
    Dense(1, activation="sigmoid")
])
```

View the model summary

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

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 12)                108

 dense_1 (Dense)             (None, 1)                 13

=================================================================
Total params: 121 (484.00 Byte)
Trainable params: 121 (484.00 Byte)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Train the model

- Compile the model with optimizer, loss function and metrics
- Use the fit function to return the run history.

In [ ]:
```
model.compile(SGD(lr = .003), "binary_crossentropy", metrics=["accuracy"])
run_hist_1 = model.fit(X_train_norm, y_train, validation_data=(X_test_norm, y_t
```

```
WARNING:absl:`lr` is deprecated in Keras optimizer, please use `learning_ra
te` or use the legacy optimizer, e.g.,tf.keras.optimizers.legacy.SGD.

Epoch 1/200
18/18 [==============================] - 1s 14ms/step - loss: 0.6772 - accu
racy: 0.6441 - val_loss: 0.7028 - val_accuracy: 0.6042
Epoch 2/200
18/18 [==============================] - 0s 5ms/step - loss: 0.6607 - accur
acy: 0.6684 - val_loss: 0.6846 - val_accuracy: 0.6302
Epoch 3/200
18/18 [==============================] - 0s 5ms/step - loss: 0.6457 - accur
acy: 0.6753 - val_loss: 0.6685 - val_accuracy: 0.6562
Epoch 4/200
18/18 [==============================] - 0s 4ms/step - loss: 0.6319 - accur
acy: 0.6858 - val_loss: 0.6539 - val_accuracy: 0.6562
Epoch 5/200
18/18 [==============================] - 0s 4ms/step - loss: 0.6196 - accur
acy: 0.6927 - val_loss: 0.6407 - val_accuracy: 0.6615
Epoch 6/200
18/18 [==============================] - 0s 4ms/step - loss: 0.6083 - accur
```

In [ ]:
```
## Like we did for the Random Forest, we generate two kinds of predictions
#  One is a hard decision, the other is a probabilitistic score.

y_pred_class_nn_1 = (model.predict(X_test_norm) > 0.5).astype("int32")
y_pred_prob_nn_1 = model.predict(X_test_norm)
```

```
6/6 [==============================] - 0s 3ms/step
6/6 [==============================] - 0s 2ms/step
```

In [ ]:
```
# Let's check out the outputs to get a feel for how keras apis work.
y_pred_class_nn_1[:10]
```

Out[15]:
```
array([[1],
       [1],
       [0],
       [0],
       [0],
       [0],
       [0],
       [0],
       [1],
       [0]], dtype=int32)
```

In [ ]:
```
y_pred_prob_nn_1[:10]
```

Out[16]:
```
array([[0.6247541 ],
       [0.78174627],
       [0.2992891 ],
       [0.30132923],
       [0.17397451],
       [0.46647456],
       [0.02336803],
       [0.3432233 ],
       [0.92992723],
       [0.11837403]], dtype=float32)
```

Create the plot_roc function

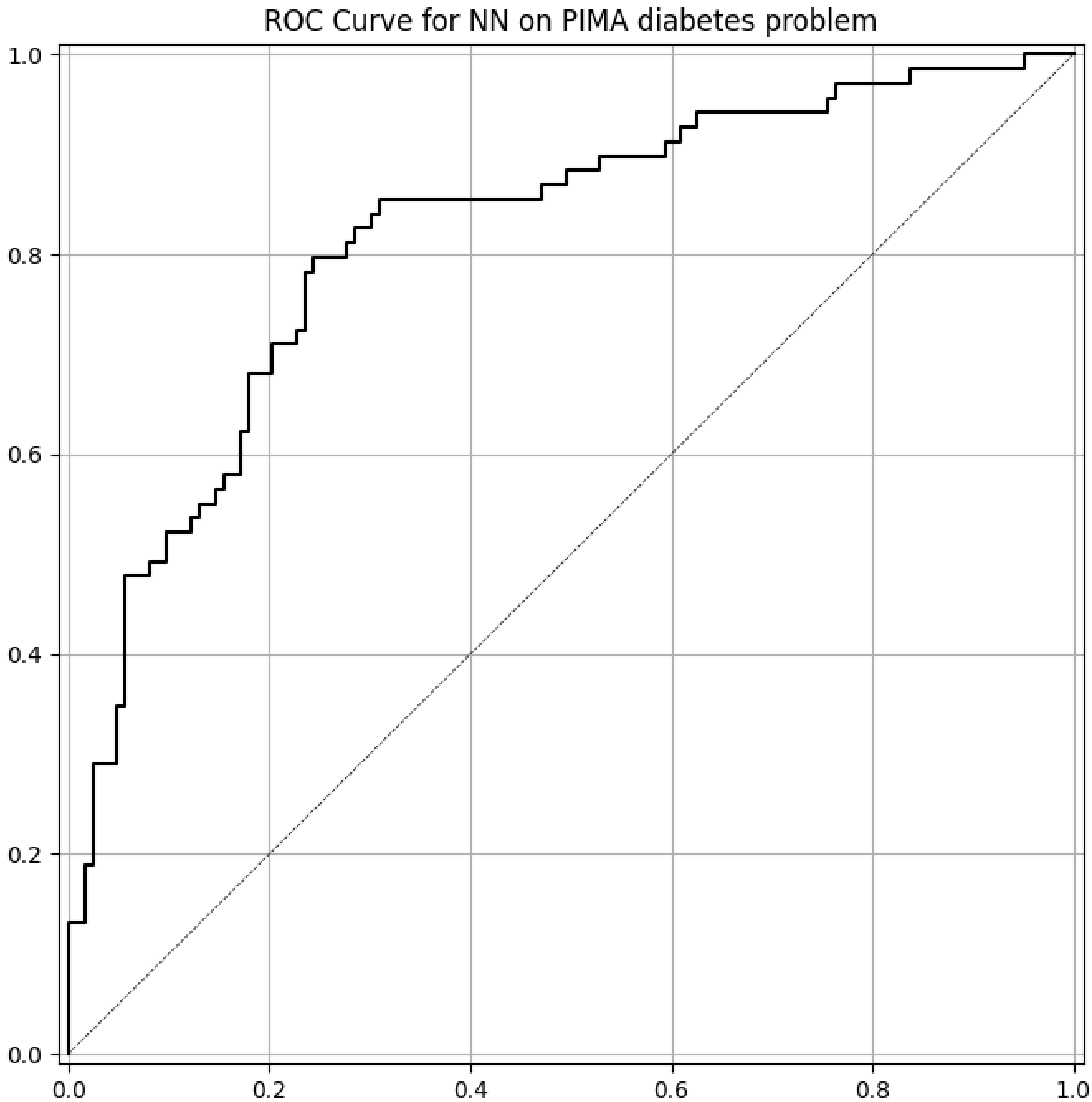
In [ ]:
```
def plot_roc(y_test, y_pred, model_name):
    fpr, tpr, thr = roc_curve(y_test, y_pred)
    fig, ax = plt.subplots(figsize=(8, 8))
    ax.plot(fpr, tpr, 'k-')
    ax.plot([0, 1], [0, 1], 'k--', linewidth=.5)  # roc curve for random model
    ax.grid(True)
    ax.set(title='ROC Curve for {} on PIMA diabetes problem'.format(model_name)
           xlim=[-0.01, 1.01], ylim=[-0.01, 1.01])
```

Evaluate the model performance and plot the ROC CURVE

In [ ]:
```
print('accuracy is {:.3f}'.format(accuracy_score(y_test,y_pred_class_nn_1)))
print('roc-auc is {:.3f}'.format(roc_auc_score(y_test,y_pred_prob_nn_1)))

plot_roc(y_test, y_pred_prob_nn_1, 'NN')
```

```
accuracy is 0.745
roc-auc is 0.817
```



Plot the training loss and the validation loss over the different epochs and see how it looks

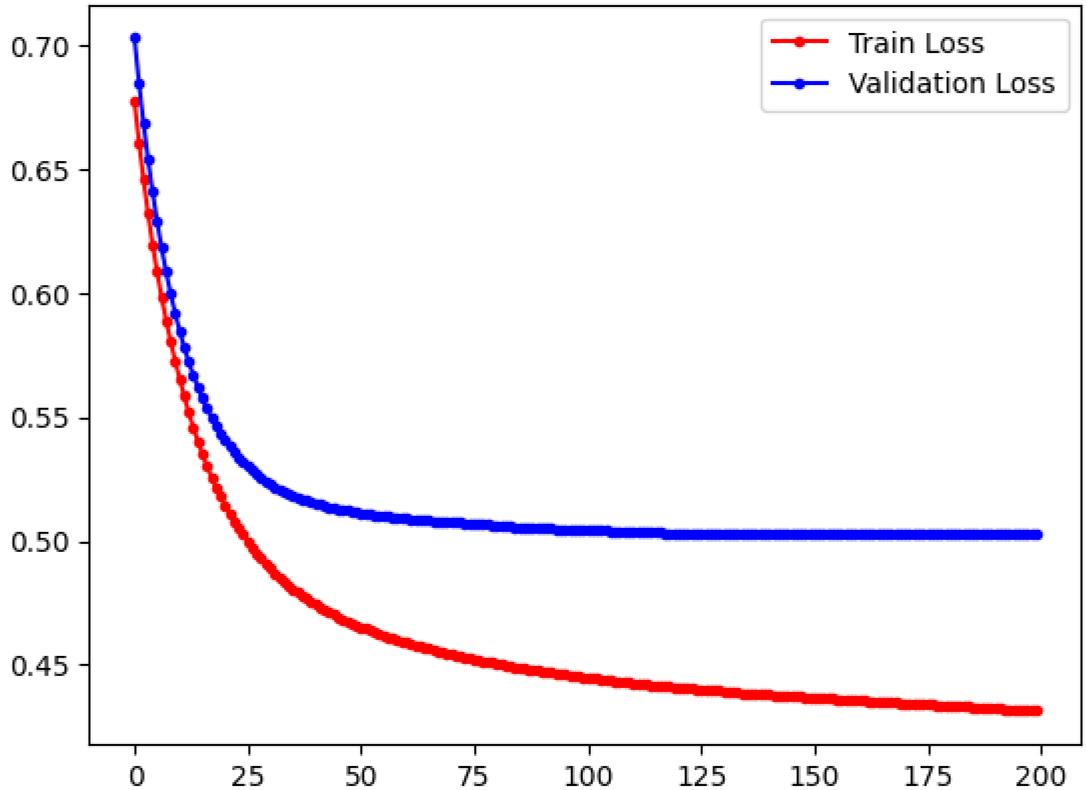
In [ ]:
```
run_hist_1.history.keys()
```

Out[19]:
```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

In [ ]:

```
fig, ax = plt.subplots()
ax.plot(run_hist_1.history["loss"],'r', marker='.', label="Train Loss")
ax.plot(run_hist_1.history["val_loss"],'b', marker='.', label="Validation Loss"
ax.legend()
```

Out[20]: <matplotlib.legend.Legend at 0x7f4fd6f10250>



What is your interpretation about the result of the train and validation loss?

Based on the graph above, it can be observed that the loss for validation is higher compared to the training. To start, training loss is an indication how the model is handling or fitting the training data; wherein validation loss indicates how the model is fitting the new data.

Aside from that, it can be interpreted that the model is overfitting, since the model performs well in training data but performs rather poorly in testing data. At some the 25th epoch, it did not show any signs of underfitting or overfitting. However, as the model was trained for a long period; hence, why it overfitted.

Supplementary Activity

- Build a model with two hidden layers, each with 6 nodes
- Use the "relu" activation function for the hidden layers, and "sigmoid" for the final layer
- Use a learning rate of .003 and train for 1500 epochs
- Graph the trajectory of the loss functions, accuracy on both train and test set
- Plot the roc curve for the predictions
- Use different learning rates, numbers of epochs, and network structures.

- Plot the results of training and validation loss using different learning rates, number of epocgs and network structures
- Interpret your result

In [ ]:
```
filepath2 = "/content/drive/MyDrive/Datasci3/breast-cancer.csv"
breast_df = pd.read_csv(filepath2)
```

In [ ]:
```
print(breast_df.shape)
breast_df.sample(5)
```

```
(569, 32)
```

Out[22]:

	id	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_
491	91376702	B	17.85	13.23	114.60	992.1	0.0
233	88206102	M	20.51	27.81	134.40	1319.0	0.0
323	895100	M	20.34	21.51	135.90	1264.0	0.1
330	896839	M	16.03	15.51	105.80	793.2	0.0
506	91544001	B	12.22	20.04	79.47	453.1	0.1

5 rows × 32 columns

In [ ]:
```
breast_df.dtypes
```

Out[23]:
```
id                           int64
diagnosis                   object
radius_mean                float64
texture_mean               float64
perimeter_mean             float64
area_mean                  float64
smoothness_mean            float64
compactness_mean           float64
concavity_mean             float64
concave points_mean        float64
symmetry_mean              float64
fractal_dimension_mean     float64
radius_se                  float64
texture_se                 float64
perimeter_se               float64
area_se                    float64
smoothness_se              float64
compactness_se             float64
concavity_se               float64
concave points_se          float64
symmetry_se                float64
fractal_dimension_se       float64
radius_worst               float64
texture_worst              float64
perimeter_worst            float64
area_worst                 float64
smoothness_worst           float64
compactness_worst          float64
concavity_worst            float64
concave points_worst       float64
symmetry_worst             float64
fractal_dimension_worst    float64
dtype: object
```

In [ ]:
```
breast_df = breast_df.drop('id', axis=1)
```

In [ ]:
```
breast_df['diagnosis'].astype(str)
```

Out[25]:
```
0      M
1      M
2      M
3      M
4      M
      ..
564    M
565    M
566    M
567    M
568    B
Name: diagnosis, Length: 569, dtype: object
```

In [ ]:
```
breast_df['diagnosis'].value_counts()
```

Out[26]:
```
B    357
M    212
Name: diagnosis, dtype: int64
```

In [ ]:
```
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()

breast_df['diagnosis'] = le.fit_transform(breast_df['diagnosis'])
```

In [ ]:
```
breast_df['diagnosis'].value_counts()
```

Out[28]:
```
0    357
1    212
Name: diagnosis, dtype: int64
```

In [ ]:
```
breast_df['diagnosis'].astype(int)
```

Out[29]:
```
0      1
1      1
2      1
3      1
4      1
      ..
564    1
565    1
566    1
567    1
568    0
Name: diagnosis, Length: 569, dtype: int64
```

In [ ]:
```
X2 = breast_df.drop('diagnosis', axis=1)
y2 = breast_df["diagnosis"]
```

In [ ]:
```
X2_train, X2_test, y2_train, y2_test = train_test_split(X2, y2, test_size=0.25,
```

In [ ]:
```
normalizer = StandardScaler()
X_train_norm2 = normalizer.fit_transform(X2_train)
X_test_norm2 = normalizer.transform(X2_test)
```

- Build a model with two hidden layers, each with 6 nodes
- Use the "relu" activation function for the hidden layers, and "sigmoid" for the final layer

In [ ]:
```
model2 = Sequential([
    Dense(6, input_shape=(30,), activation="relu"),
    Dense(6, input_shape=(30,), activation="relu"),
    Dense(1, activation="sigmoid")
])
```

In [ ]:
```
model2.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_2 (Dense)             (None, 6)                 186

 dense_3 (Dense)             (None, 6)                 42

 dense_4 (Dense)             (None, 1)                 7

=================================================================
Total params: 235 (940.00 Byte)
Trainable params: 235 (940.00 Byte)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Use a learning rate of .003 and train for 1500 epochs

In [ ]:
```
model2.compile(SGD(lr = .003), "binary_crossentropy", metrics=["accuracy"])
run_hist_2 = model2.fit(X_train_norm2, y2_train, validation_data=(X_test_norm2,
```

```
WARNING:absl:`lr` is deprecated in Keras optimizer, please use `learning_ra
te` or use the legacy optimizer, e.g.,tf.keras.optimizers.legacy.SGD.

Epoch 1/1500
14/14 [==============================] - 2s 37ms/step - loss: 0.7157 - accu
racy: 0.3568 - val_loss: 0.7274 - val_accuracy: 0.3566
Epoch 2/1500
14/14 [==============================] - 0s 10ms/step - loss: 0.7032 - accu
racy: 0.3451 - val_loss: 0.7113 - val_accuracy: 0.3986
Epoch 3/1500
14/14 [==============================] - 0s 7ms/step - loss: 0.6914 - accur
acy: 0.3826 - val_loss: 0.6963 - val_accuracy: 0.4266
Epoch 4/1500
14/14 [==============================] - 0s 10ms/step - loss: 0.6804 - accu
racy: 0.4390 - val_loss: 0.6813 - val_accuracy: 0.4755
Epoch 5/1500
14/14 [==============================] - 0s 11ms/step - loss: 0.6694 - accu
racy: 0.4695 - val_loss: 0.6658 - val_accuracy: 0.5105
Epoch 6/1500
14/14 [==============================] - 0s 8ms/step - loss: 0.6579 - accur
```

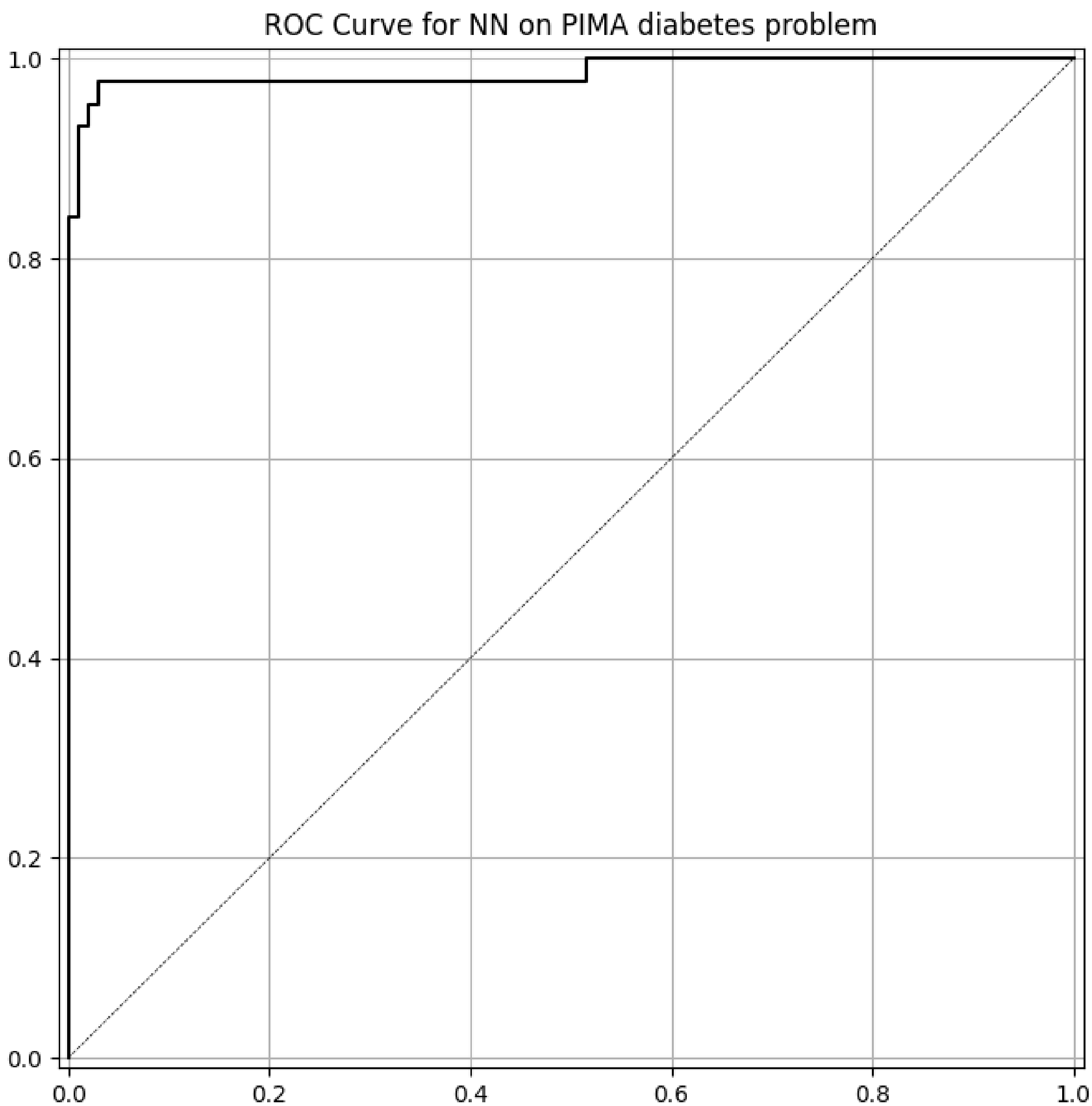
In [ ]:
```
y_pred_class_nn_2 = (model2.predict(X_test_norm2) > 0.5).astype("int32")
y_pred_prob_nn_2 = model2.predict(X_test_norm2)
```

```
5/5 [==============================] - 0s 3ms/step
5/5 [==============================] - 0s 2ms/step
```

In [ ]:
```
print('accuracy is {:.3f}'.format(accuracy_score(y2_test,y_pred_class_nn_2)))
print('roc-auc is {:.3f}'.format(roc_auc_score(y2_test,y_pred_prob_nn_2)))

plot_roc(y2_test, y_pred_prob_nn_2, 'NN')
```

```
accuracy is 0.972
roc-auc is 0.986
```



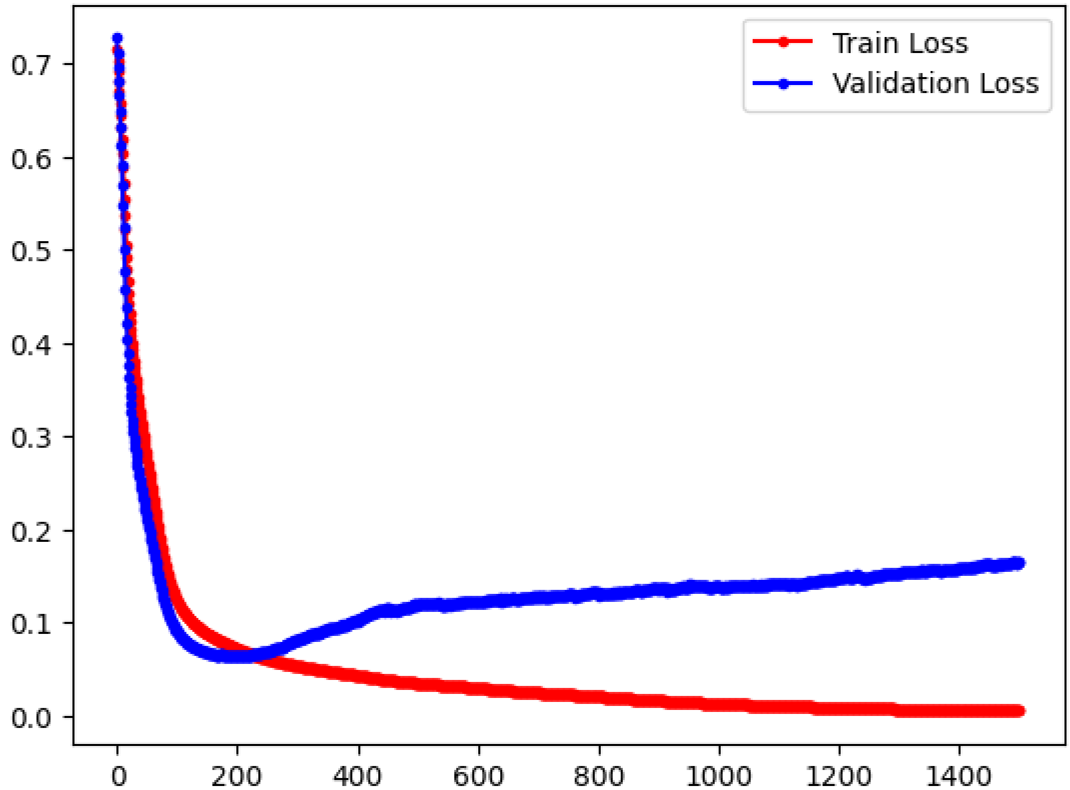
It can be seen that this graph can accurately predict values with a high accuracy. Considering that its accuracy is 97.2, and it's roc-auc is 98.6, it ca be inferred that this model is performing fairly since it does not experience high difficulty in distinguishing both classes. However, it cannot be overlooked that this model is still sensitive to small changes (new data).

In [ ]:
```
run_hist_2.history.keys()
```

Out[41]:
```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

In [ ]:
```
fig, ax = plt.subplots()
ax.plot(run_hist_2.history["loss"],'r', marker='.', label="Train Loss")
ax.plot(run_hist_2.history["val_loss"],'b', marker='.', label="Validation Loss"
ax.legend()
```

Out[42]: <matplotlib.legend.Legend at 0x7f4fd6d7d930>



Based on the given required parameters, such parameters is not a fit for this model. It is highly observable how this model overfitted as the training period is prolonged. Therefore, it can be concluded that this model, with the given parameters, is not suitable in distinguishng between the two classes.

- Use different learning rates, numbers of epochs, and network structures.

In [ ]:
```
model3 = Sequential([
    Dense(10, input_shape=(30,), activation="relu"),
    Dense(10, input_shape=(30,), activation="relu"),
    Dense(10, input_shape=(30,), activation="relu"),
    Dense(10, input_shape=(30,), activation="relu"),
    Dense(1, activation="sigmoid")
])
```

In [ ]:
```
model3.compile(SGD(lr = .001), "binary_crossentropy", metrics=["accuracy"])
run_hist_3 = model3.fit(X_train_norm2, y2_train, validation_data=(X_test_norm2,
```

```
WARNING:absl:`lr` is deprecated in Keras optimizer, please use `learning_ra
te` or use the legacy optimizer, e.g.,tf.keras.optimizers.legacy.SGD.

Epoch 1/500
14/14 [==============================] - 2s 26ms/step - loss: 0.7206 - accu
racy: 0.6127 - val_loss: 0.6999 - val_accuracy: 0.6923
Epoch 2/500
14/14 [==============================] - 0s 7ms/step - loss: 0.7020 - accur
acy: 0.6268 - val_loss: 0.6824 - val_accuracy: 0.6993
Epoch 3/500
14/14 [==============================] - 0s 6ms/step - loss: 0.6862 - accur
acy: 0.6362 - val_loss: 0.6674 - val_accuracy: 0.7203
Epoch 4/500
14/14 [==============================] - 0s 7ms/step - loss: 0.6726 - accur
acy: 0.6901 - val_loss: 0.6538 - val_accuracy: 0.7762
Epoch 5/500
14/14 [==============================] - 0s 8ms/step - loss: 0.6606 - accur
acy: 0.7465 - val_loss: 0.6410 - val_accuracy: 0.8252
Epoch 6/500
14/14 [==============================] - 0s 7ms/step - loss: 0.6493 - accur
```

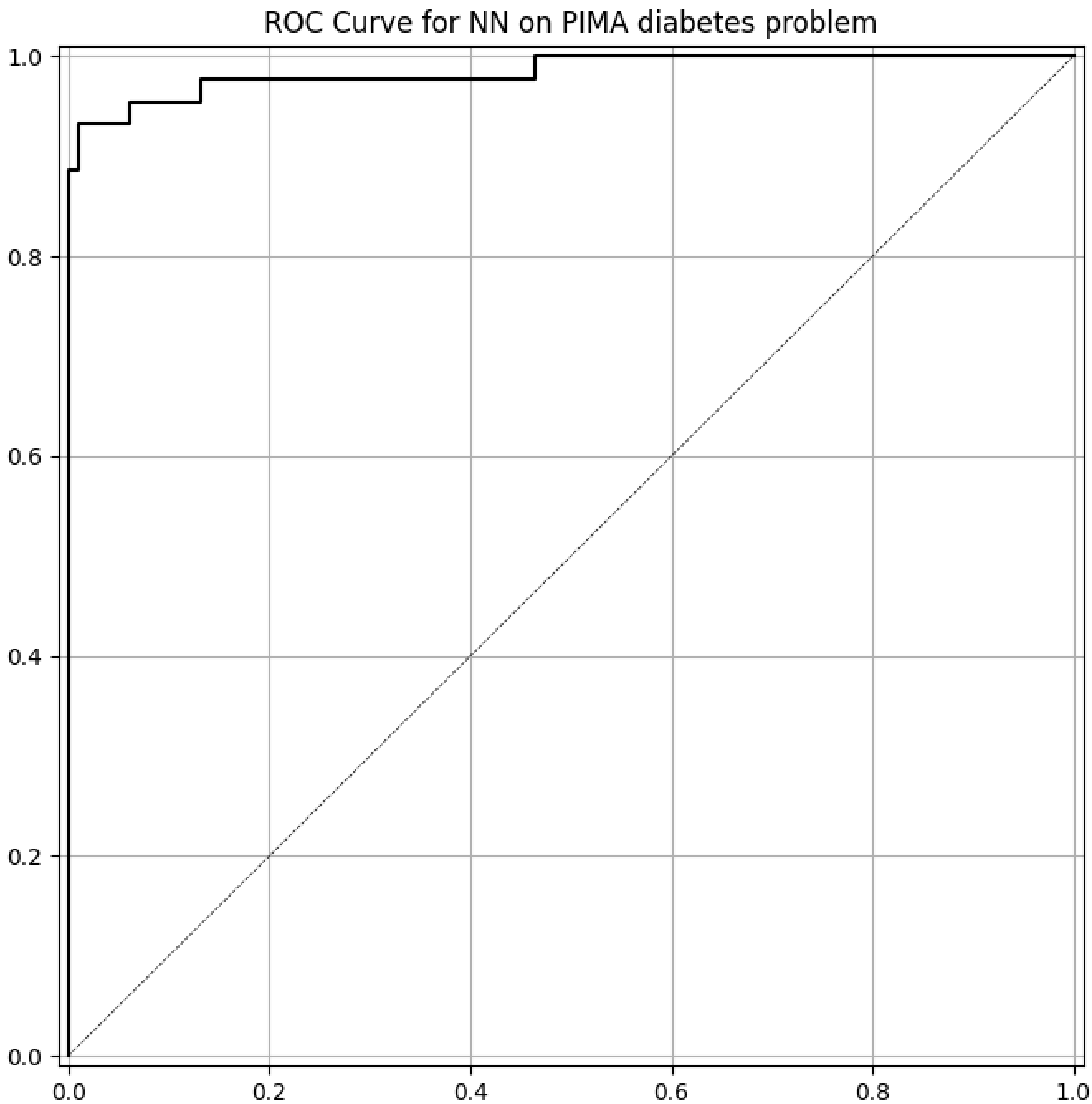
In [ ]:
```
y_pred_class_nn_3 = (model3.predict(X_test_norm2) > 0.5).astype("int32")
y_pred_prob_nn_3 = model3.predict(X_test_norm2)
```

```
5/5 [==============================] - 0s 7ms/step
5/5 [==============================] - 0s 3ms/step
```

In [ ]:
```
print('accuracy is {:.3f}'.format(accuracy_score(y2_test,y_pred_class_nn_3)))
print('roc-auc is {:.3f}'.format(roc_auc_score(y2_test,y_pred_prob_nn_3)))

plot_roc(y2_test, y_pred_prob_nn_3, 'NN')
```

```
accuracy is 0.972
roc-auc is 0.985
```



With the given parameters, it can be observed that the model is still quite having the trouble in distinguishing between the two classes. Albeit its accuracy is 97.2, the threshold in reaching 1.0 was close, and being near 1.0 indicates that this is a fairly-well model.

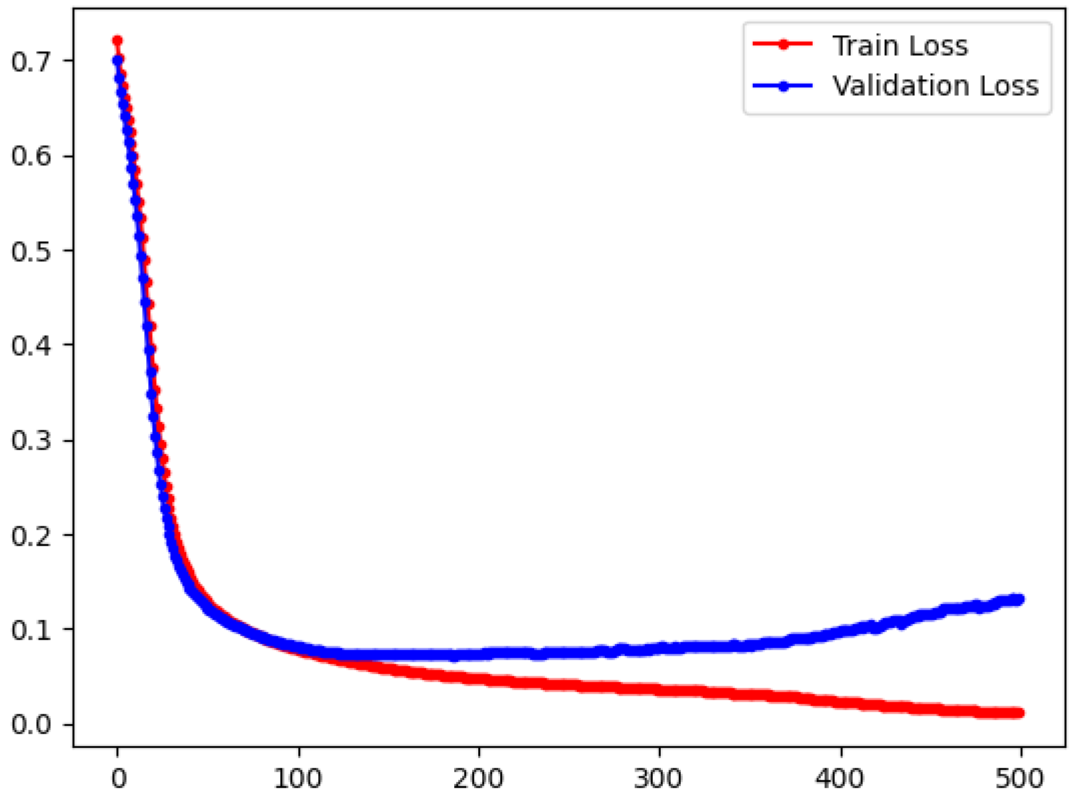
The curve did not show any significant difference compared with the first ROC curve.

In [ ]:
```
run_hist_3.history.keys()
```

Out[48]:
```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

In [ ]:
```
fig, ax = plt.subplots()
ax.plot(run_hist_3.history["loss"],'r', marker='.', label="Train Loss")
ax.plot(run_hist_3.history["val_loss"],'b', marker='.', label="Validation Loss"
ax.legend()
```

Out[49]:
```
<matplotlib.legend.Legend at 0x7f4fd1b2c4f0>
```



It can be seen that the parameters used also causes the model to overfit, after a lot of training periods, the model slowly overfitted. At the range of 120-130th epoch, it showed signs of overfitting. Ergo, the parameters did not fit the model, as it still showed signs of being sensitive to new data.

In [ ]:
```
model4 = Sequential([
    Dense(6, input_shape=(30,), activation="relu"),
    Dense(6, input_shape=(30,), activation="relu"),
    Dense(6, input_shape=(30,), activation="relu"),
    Dense(6, input_shape=(30,), activation="relu"),
    Dense(1, activation="sigmoid")
])
```

In [ ]:
```
model4.compile(SGD(lr = .001), "binary_crossentropy", metrics=["accuracy"])
run_hist_4 = model4.fit(X_train_norm2, y2_train, validation_data=(X_test_norm2,
```

```
WARNING:absl:`lr` is deprecated in Keras optimizer, please use `learning_ra
te` or use the legacy optimizer, e.g.,tf.keras.optimizers.legacy.SGD.

Epoch 1/500
14/14 [==============================] - 3s 48ms/step - loss: 0.7254 - accu
racy: 0.4319 - val_loss: 0.7208 - val_accuracy: 0.4895
Epoch 2/500
14/14 [==============================] - 0s 27ms/step - loss: 0.7097 - accu
racy: 0.4930 - val_loss: 0.7040 - val_accuracy: 0.5664
Epoch 3/500
14/14 [==============================] - 0s 23ms/step - loss: 0.6983 - accu
racy: 0.5469 - val_loss: 0.6917 - val_accuracy: 0.6084
Epoch 4/500
14/14 [==============================] - 0s 10ms/step - loss: 0.6905 - accu
racy: 0.6080 - val_loss: 0.6845 - val_accuracy: 0.6643
Epoch 5/500
14/14 [==============================] - 0s 11ms/step - loss: 0.6860 - accu
racy: 0.6549 - val_loss: 0.6789 - val_accuracy: 0.7063
Epoch 6/500
14/14 [==============================] - 0s 15ms/step - loss: 0.6824 - accu
```

In [ ]:
```
y_pred_class_nn_4 = (model4.predict(X_test_norm2) > 0.5).astype("int32")
y_pred_prob_nn_4 = model4.predict(X_test_norm2)
```

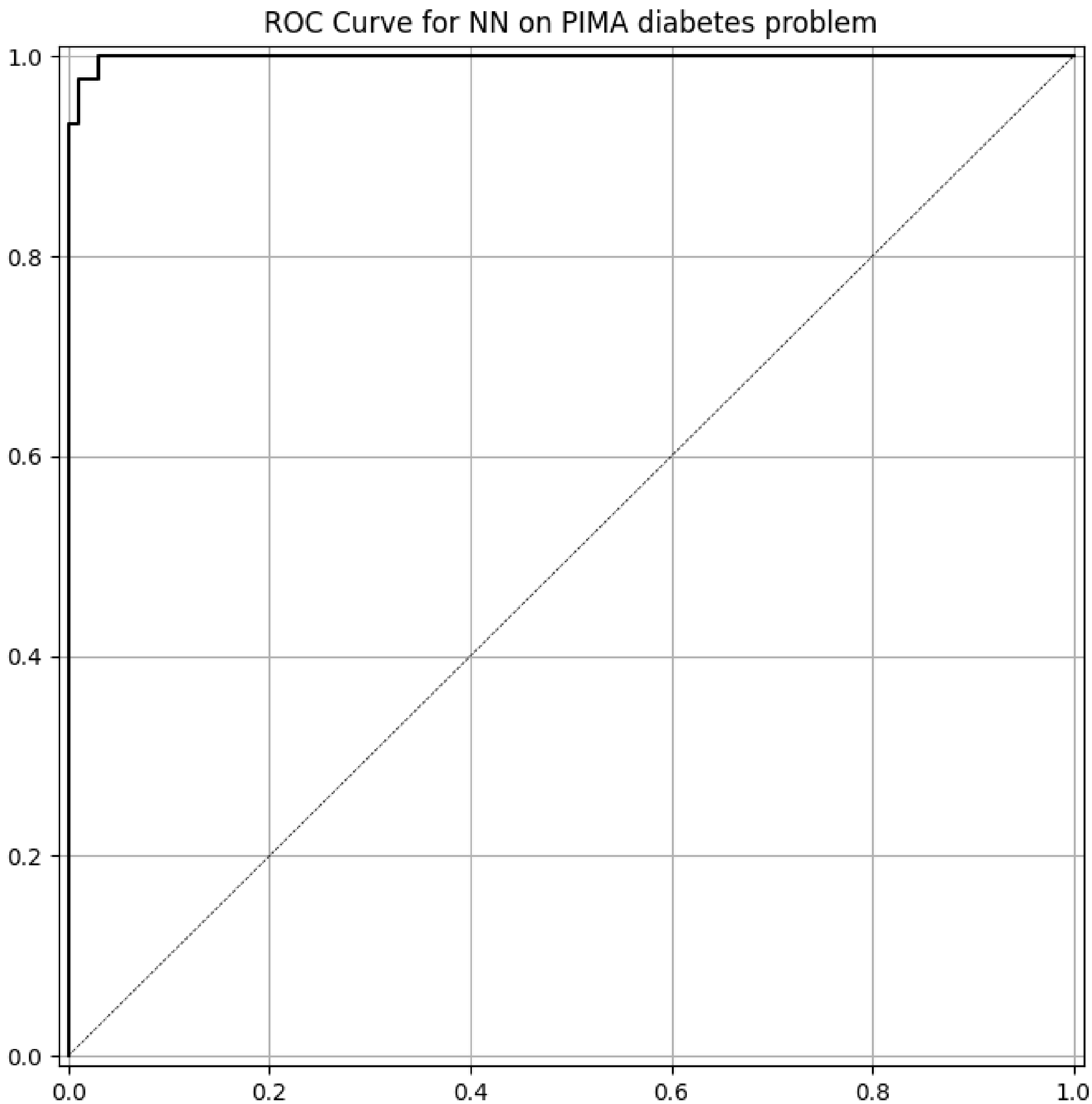
```
5/5 [==============================] - 0s 3ms/step
5/5 [==============================] - 0s 3ms/step
```

In [ ]:

```
print('accuracy is {:.3f}'.format(accuracy_score(y2_test,y_pred_class_nn_4)))
print('roc-auc is {:.3f}'.format(roc_auc_score(y2_test,y_pred_prob_nn_4)))

plot_roc(y2_test, y_pred_prob_nn_4, 'NN')
```

```
accuracy is 0.986
roc-auc is 0.999
```

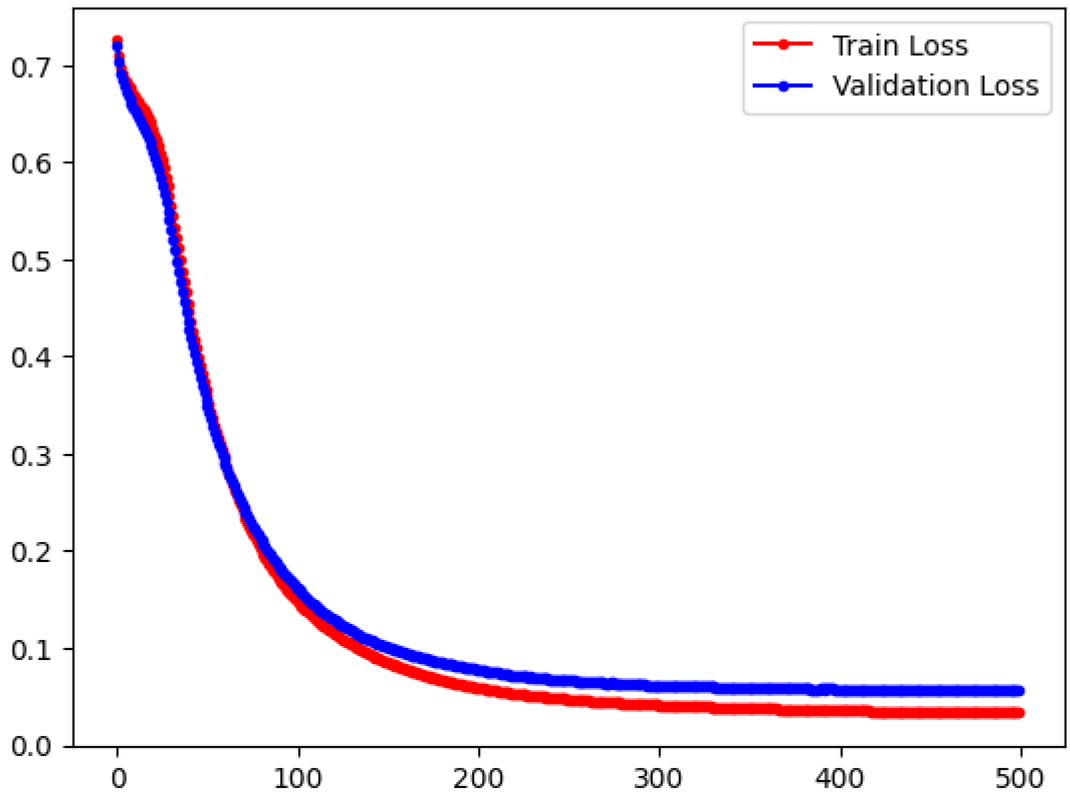


Here, it can be seen that the error is very minimal, and the roc curve is close to have a perfect trend. Considering that the accuracy is 98.6% and the roc-auc is 99%, this parameters served as a great fit for this model. With 4 hidden layers, 1 final layer, .001 learning rate, and a total of 500 epochs. The same activation function and function of the final layer was used with the former network structure.

Hence, this model was able to distinguish between the classes outstandingly since the roc cruve leans towards the corner (as it's an indicator of whether the model is performing well or not.)

In [ ]:
```
run_hist_4.history.keys()
```
Out[270]:
```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

In [ ]:
```
fig, ax = plt.subplots()
ax.plot(run_hist_4.history["loss"],'r', marker='.', label="Train Loss")
ax.plot(run_hist_4.history["val_loss"],'b', marker='.', label="Validation Loss"
ax.legend()
```
Out[271]:
```
<matplotlib.legend.Legend at 0x7f4fc6bd4100>
```



Based on the figure, it can be stated that 4 hidden layer with relu activations, and 1 final layer with a sigmoid function, and alongwith a .001 learning rate, BreastCancer_df proved to be effective with this network structure and parameters. It can be seen that it only slightly overfits, but the trend of the plot is clean and promising. This model can perform well given the set of parameters - comparing it to the others, this model can handle new data and is not sensitive to changes, as it does not overfits completely.

Conclusion

It can be concluded that this activity was a similar in training a machine learning model. There are only different parameters and some functions wasn't performed in machine learning. Moreover, I was able to grasp the concept in building and training a neural network (which is not so different from ML) and evaluating and plotting the model using training and validation loss. Formerly, evaluation and plotting of the model was reliant on the classification report. However, in deep learning, training and validation loss was used.

Other than that, this activity served as a refresher in training models, and another thing that I noticed is that this activity haven't utilized grid search yet, which was always