PREDICTING PRIMA INDIAN DIABETES DATASET USING NEURAL NETWORK

1. MAIN OBJECTIVE

In this project, we will use neural network models to predict diabetes using the Pima Diabetes Dataset. We will build and train several neural networks using different optimizer methods and different number of hidden layers. We will compare the accuracy results from those models to see how different network structures can affect the performance.

2. DATASET

The dataset used in this project is the UCI Pima Indian Diabetes Dataset from UCI ML Repository:

http://archive.ics.uci.edu/ml/datasets/Pima+Indians+Diabetes

The dataset contains 9 different attributes with one of them contains the diabetes outcome status in binary value that we are going to predict. The details for each attributes can be seen in the table below:

ATTRIBUTES	DESCRIPTION	DTYPE
Pregnancies	Number of times pregnant	int64
Glucose	Plasma glucose concentration a 2 hours in an oral glucose tolerance test	int64
BloodPressure	Diastolic blood pressure (mm Hg)	int64
SkinThickness	Triceps skin fold thickness (mm)	int64
Insulin	2-Hour serum insulin (mu U/ml)	int64
BMI	Body mass index (weight in kg/(height in m)^2)	float64
DiabetesPedigreeFunction	Diabetes pedigree function	float64
Age	Age (years)	int64
Outcome	Class variable (0 or 1)	int64

3. DATA EXPLORATION

3.1. DATA IMPORT

For starter, we will import the dataset into a Panda dataframe using pd.read_csv(). We have a total of 768 samples with 9 columns in the dataframe.

```
file = 'diabetes.csv'
data = pd.read_csv(file)
```

```
data.shape
```

```
(768, 9)
```

3.2. EPLORATORY DATA ANALYSIS

Next, we will show some of the standard information from the dataframe. We can see that we have no missing value for the features and all features are in correct dtypes.

```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 768 entries, 0 to 767
Data columns (total 9 columns):
 #   Column                    Non-Null Count  Dtype
---  ------                    --------------  -----
 0   Pregnancies               768 non-null    int64
 1   Glucose                   768 non-null    int64
 2   BloodPressure             768 non-null    int64
 3   SkinThickness             768 non-null    int64
 4   Insulin                   768 non-null    int64
 5   BMI                       768 non-null    float64
 6   DiabetesPedigreeFunction  768 non-null    float64
 7   Age                       768 non-null    int64
 8   Outcome                   768 non-null    int64
dtypes: float64(2), int64(7)
memory usage: 54.1 KB
```

```
data.head()
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

```
data.describe()
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
count	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000
mean	3.845052	120.894531	69.105469	20.536458	79.799479	31.992578	0.471876	33.240885	0.348958
std	3.369578	31.972618	19.355807	15.952218	115.244002	7.884160	0.331329	11.760232	0.476951
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.078000	21.000000	0.000000
25%	1.000000	99.000000	62.000000	0.000000	0.000000	27.300000	0.243750	24.000000	0.000000
50%	3.000000	117.000000	72.000000	23.000000	30.500000	32.000000	0.372500	29.000000	0.000000
75%	6.000000	140.250000	80.000000	32.000000	127.250000	36.600000	0.626250	41.000000	1.000000
max	17.000000	199.000000	122.000000	99.000000	846.000000	67.100000	2.420000	81.000000	1.000000

We can see that our dataset is unbalanced, where the number of samples that does not contract diabetes (65%) are higher than the one who does (35%). We will need to address this later when we create our train and test datasets.

```
data['Outcome'].value_counts()
```

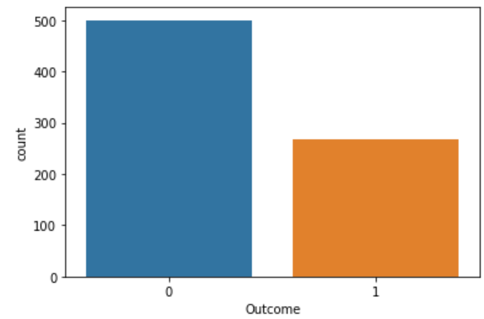
```
0    500
1    268
Name: Outcome, dtype: int64
```

```
data['Outcome'].value_counts(normalize = True)
```

```
0    0.651042
1    0.348958
Name: Outcome, dtype: float64
```

```
sns.countplot(x = data['Outcome'])
```

<AxesSubplot:xlabel='Outcome', ylabel='count'>



We can try to plot the dataframe using sns.pairplot() to see how the features are related to each other. We will also differentiate the samples using the *Outcome* value. Note that the *Outcome* value does not seem to be clearly separable from the feature pairs alone.

```
sns.pairplot(data, hue = 'Outcome')
```

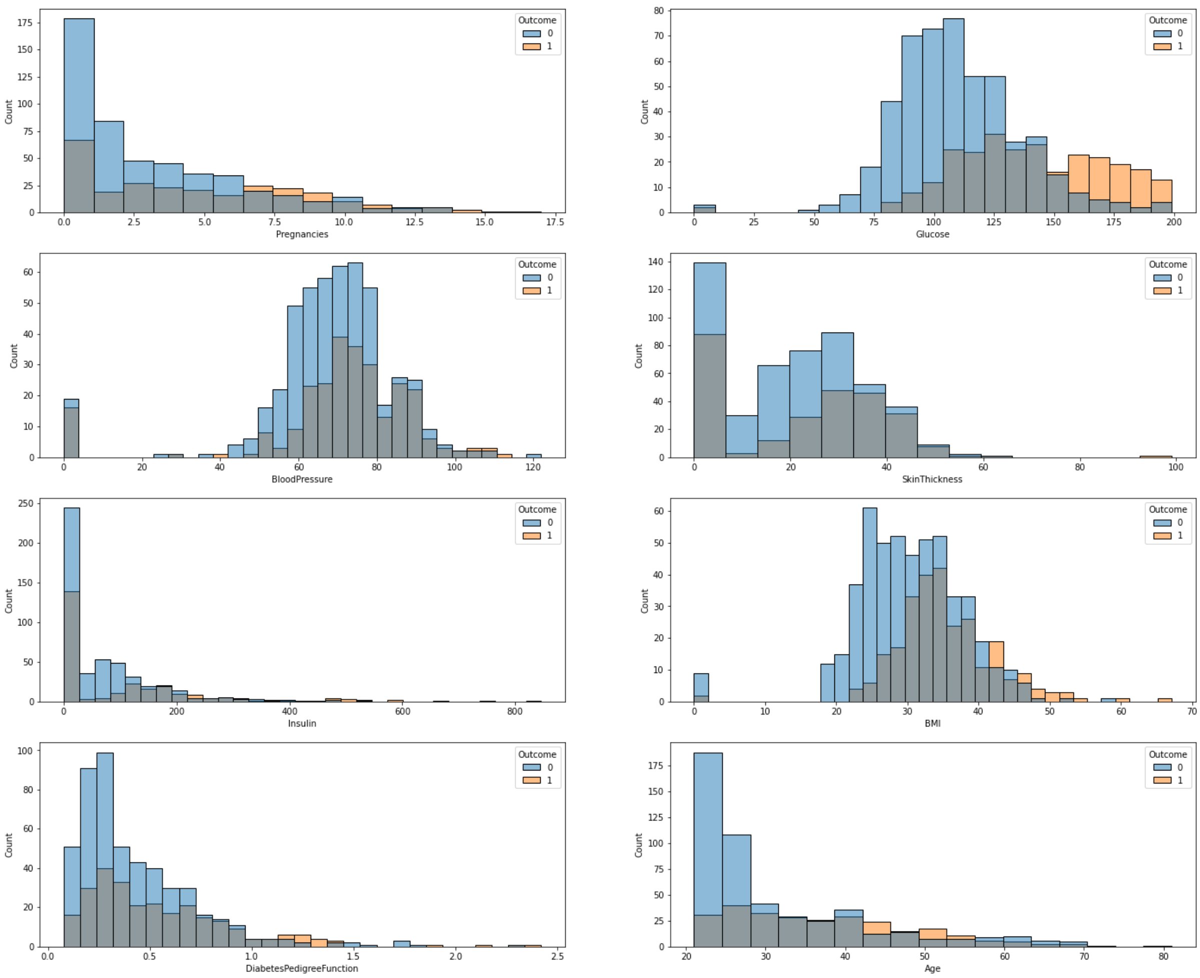
<seaborn.axisgrid.PairGrid at 0x21ee0e19e10>

We will also plot the distribution of the feature values using sns.histplot().

```
feat_cols = data.columns.drop('Outcome')
feat_cols
```

```
Index(['Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness', 'Insulin',
       'BMI', 'DiabetesPedigreeFunction', 'Age'],
      dtype='object')
```

```
fig, axes = plt.subplots(4, 2, figsize=(24, 20))

for idx, feat in enumerate([*feat_cols]):
    row = math.floor(idx/2)
    col = idx - math.floor(idx/2)*2
    sns.histplot(x = data[feat], hue = data['Outcome'], ax = axes[row, col])
```



3.3. DATA SCALING

We will use MinMaxScaler() to scale all the feature values into the same data range, in this case in a range of 0 and 1.

```
from sklearn.preprocessing import MinMaxScaler

mm_scaler = MinMaxScaler()
data[feat_cols] = mm_scaler.fit_transform(data[feat_cols])
```

```
data.describe()
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
count	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000
mean	0.226180	0.607510	0.566438	0.207439	0.094326	0.476790	0.168179	0.204015	0.348958
std	0.198210	0.160666	0.158654	0.161134	0.136222	0.117499	0.141473	0.196004	0.476951
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
25%	0.058824	0.497487	0.508197	0.000000	0.000000	0.406855	0.070773	0.050000	0.000000
50%	0.176471	0.587940	0.590164	0.232323	0.036052	0.476900	0.125747	0.133333	0.000000
75%	0.352941	0.704774	0.655738	0.323232	0.150414	0.545455	0.234095	0.333333	1.000000
max	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000

4. TRAIN, TEST, SPLIT

Now it is time to create out train and test dataset. Remember that we have unbalanced number of *Outcome* values, so we will set stratify = True in our train_test_split() function.

```
from sklearn.model_selection import train_test_split

X = data.drop('Outcome', axis = 1)
y = data['Outcome']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state = 42, stratify = y)
```

```
X_train.shape, y_train.shape
```

```
((576, 8), (576,))
```

```
X_test.shape, y_test.shape
```

```
((192, 8), (192,))
```

We now have 576 samples for our training data and 192 samples for our test data. Note that the ratio of the *Outcome* between the test and train dataset are kept the same.

```
pd.DataFrame(y_train).value_counts(normalize = True)
```

```
Outcome
0          0.651042
1          0.348958
dtype: float64
```

```
pd.DataFrame(y_test).value_counts(normalize = True)
```

```
Outcome
0          0.651042
1          0.348958
dtype: float64
```

5. MODEL TRAINING

Let us start by importing the necessary libraries for building our neural network models.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam, SGD, RMSprop
```

As stated in our objective, we will build several models with different optimizers and different number of hidden layers.

- We will use Adam, SGD, and RMSprop for the optimizers
- For each optimizer we will train 4 models with 1, 2, 3, and 4 hidden layers.
- Each of the hidden layers will be identical Dense layers with 12 nodes for each layer.
- All models will be trained with 10,000 epochs.
- We will use a learning rate of 0.003 for all the models.

First, we set the variables for the number of models (*n_model*) and the number of epochs (*n_epoch*).

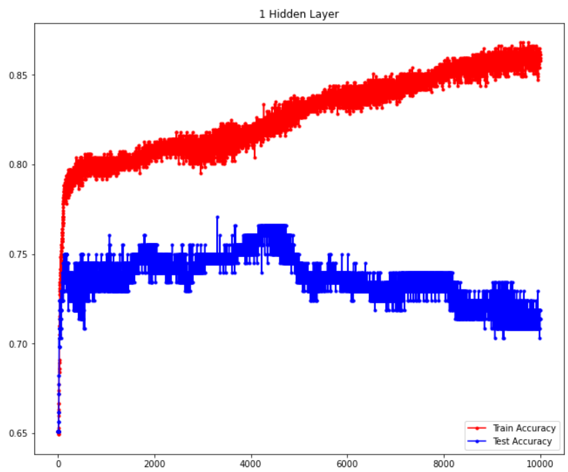
```
n_model = 4
n_epoch = 10000
```

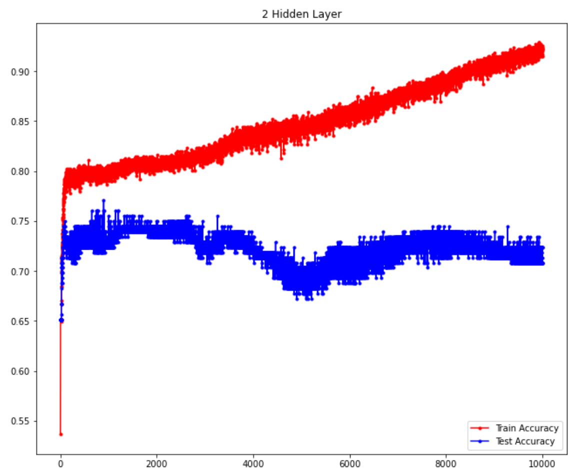
We will then use for loops to create the models and save them in a list. We will do the same with the history for each model.

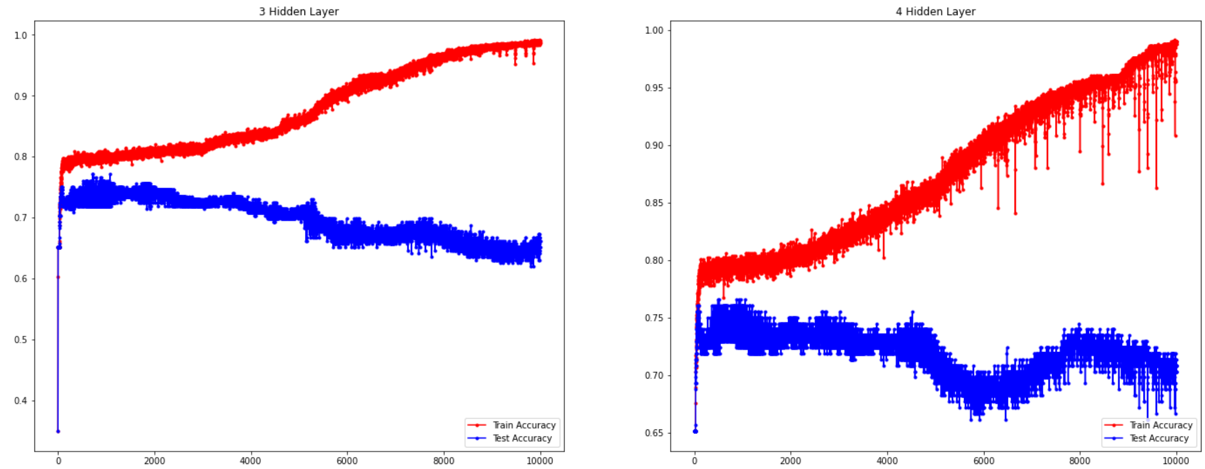
5.1. OPTIMIZER = ADAM

```
model_Adam = []
run_hist_Adam = []

for i in range(0, n_model):
    # initialize the model
    model_Adam.append(Sequential())

    # add hidden layer
    for j in range(0, i + 1):
        model_Adam[i].add(Dense(12,input_shape = (8,),activation = 'sigmoid'))

    # add output layer
    model_Adam[i].add(Dense(1,activation='sigmoid'))

    model_Adam[i].compile(Adam(lr = .003), "binary_crossentropy", metrics = ["accuracy"])
    run_hist_Adam.append(model_Adam[i].fit(X_train, y_train, validation_data = (X_test, y_test), epochs = n_epoch))
```

```
fig, axes = plt.subplots(2, 2, figsize=(24, 20))

for i in range (0, n_model):
    ax = plt.subplot(2, 2, i+1)
    ax.plot(run_hist_Adam[i].history["accuracy"],'r', marker='.', label="Train Accuracy")
    ax.plot(run_hist_Adam[i].history["val_accuracy"],'b', marker='.', label="Test Accuracy")
    ax.legend(loc = 'lower right')
    title = str(i + 1) + ' Hidden Layer'
    ax.title.set_text(title)
```



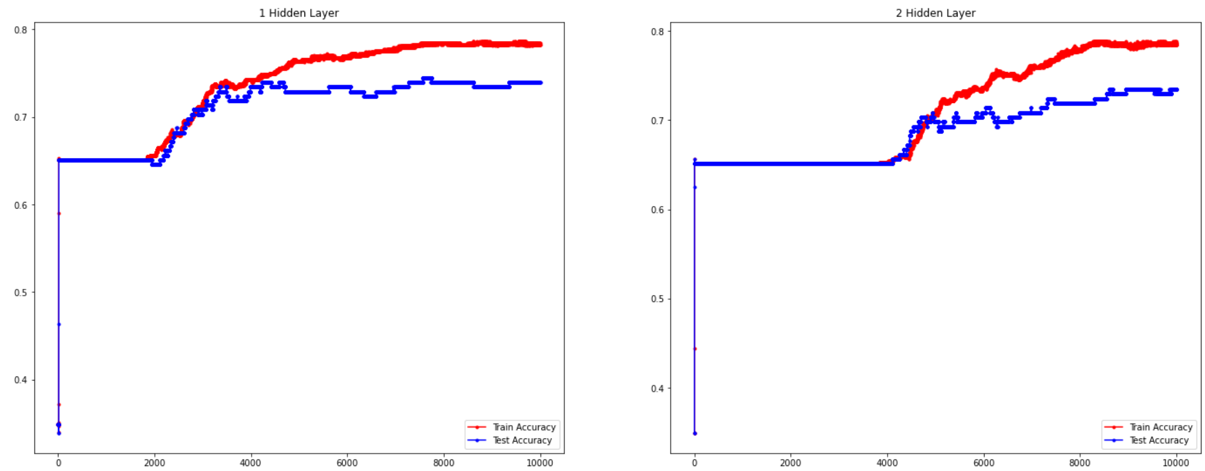


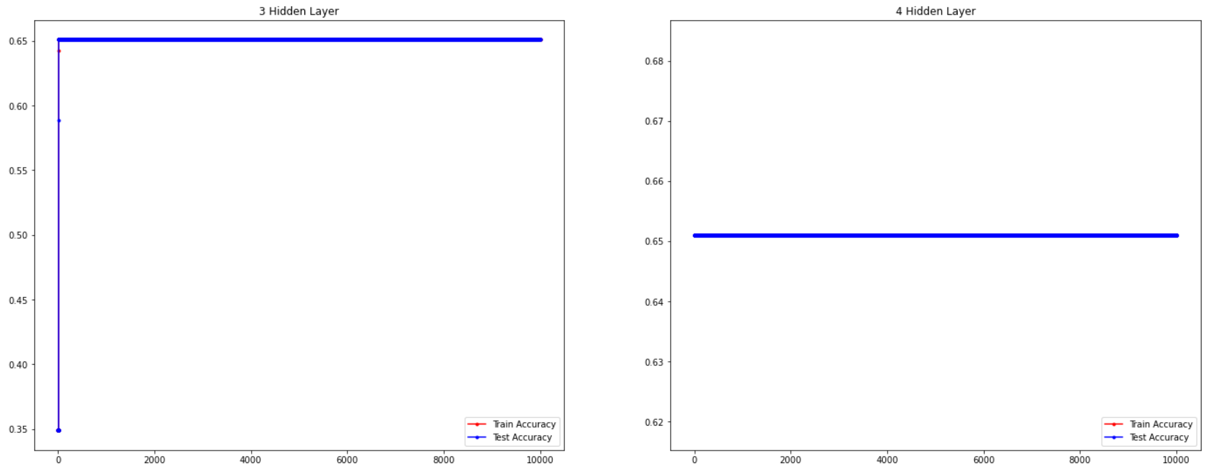


5.2. OPTIMIZER = SGD

```
model_SGD = []
run_hist_SGD = []

for i in range(0, n_model):
    # initialize the model
    model_SGD.append(Sequential())

    # add hidden layer
    for j in range(0, i + 1):
        model_SGD[i].add(Dense(12,input_shape = (8,),activation = 'sigmoid'))

    # add output layer
    model_SGD[i].add(Dense(1,activation='sigmoid'))

    model_SGD[i].compile(SGD(lr = .003), "binary_crossentropy", metrics = ["accuracy"])
    run_hist_SGD.append(model_SGD[i].fit(X_train, y_train, validation_data = (X_test, y_test), epochs = n_epoch))
```

```
fig, axes = plt.subplots(2, 2, figsize=(24, 20))

for i in range (0, n_model):
    ax = plt.subplot(2, 2, i+1)
    ax.plot(run_hist_SGD[i].history["accuracy"],'r', marker='.', label="Train Accuracy")
    ax.plot(run_hist_SGD[i].history["val_accuracy"],'b', marker='.', label="Test Accuracy")
    ax.legend(loc = 'lower right')
    title = str(i + 1) + ' Hidden Layer'
    ax.title.set_text(title)
```

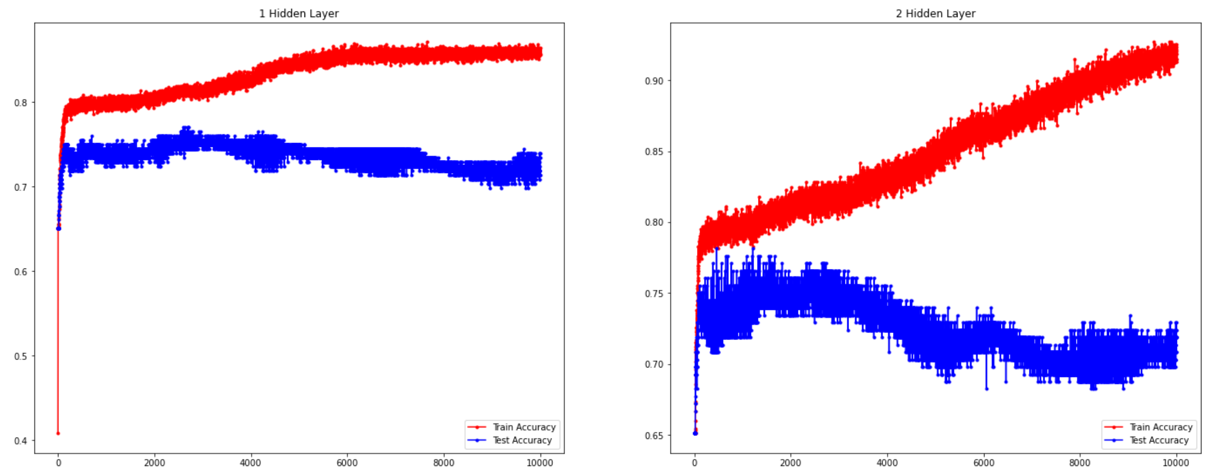


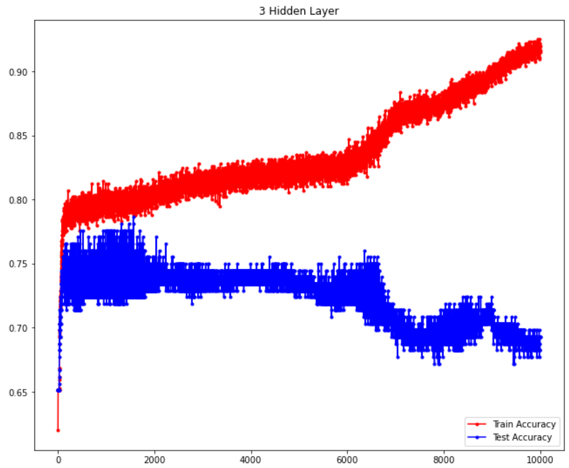


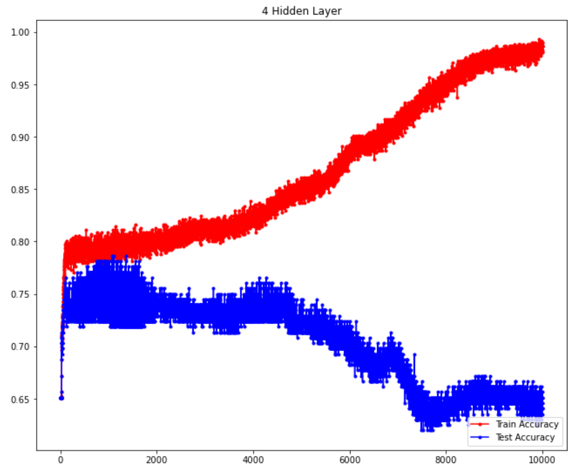
5.3. OPTIMIZER = RMSPROP

```
model_rmsprop = []
run_hist_rmsprop = []

for i in range(0, n_model):
    # initialize the model
    model_rmsprop.append(Sequential())

    # add hidden Layer
    for j in range(0, i + 1):
        model_rmsprop[i].add(Dense(12,input_shape = (8,),activation = 'sigmoid'))

    # add output Layer
    model_rmsprop[i].add(Dense(1,activation='sigmoid'))

    model_rmsprop[i].compile(RMSprop(lr = .003), "binary_crossentropy", metrics = ["accuracy"])
    run_hist_rmsprop.append(model_rmsprop[i].fit(X_train, y_train, validation_data = (X_test, y_test), epochs = n_epoch))
```

```
fig, axes = plt.subplots(2, 2, figsize=(24, 20))

for i in range (0, n_model):
    ax = plt.subplot(2, 2, i+1)
    ax.plot(run_hist_rmsprop[i].history["accuracy"],'r', marker='.', label="Train Accuracy")
    ax.plot(run_hist_rmsprop[i].history["val_accuracy"],'b', marker='.', label="Test Accuracy")
    ax.legend(loc = 'lower right')
    title = str(i + 1) + ' Hidden Layer'
    ax.title.set_text(title)
```







6. MODEL EVALUATION

Based on the results above, we can see that as the number of epochs increase, the more the models start to overfit and the performance for each model starts to decrease. This is true for all models except for the ones using SGD as the optimizer.

For SGD, the accuracy only starts to increase after 2000 epochs for 1 hidden layer and after 4000 epochs for 2 hidden layers. As for 3 and 4 hidden layers, the accuracy for both the train and test datasets flatten out during the duration of the epochs.

We will then calculate the mean accuracy for both the train and dataset from the history for each model and assign them to dataframe for easier comparison.

```
pd_hist_mean_train = pd.DataFrame()

for i in range (0, n_model):
    pd_hist_mean_train.loc[i + 1, 'Adam'] = np.mean(run_hist_Adam[i].history["accuracy"])
    pd_hist_mean_train.loc[i + 1, 'SGD'] = np.mean(run_hist_SGD[i].history["accuracy"])
    pd_hist_mean_train.loc[i + 1, 'RMSprop'] = np.mean(run_hist_rmsprop[i].history["accuracy"])

pd_hist_mean_test = pd.DataFrame()

for i in range (0, n_model):
    pd_hist_mean_test.loc[i + 1, 'Adam'] = np.mean(run_hist_Adam[i].history["val_accuracy"])
    pd_hist_mean_test.loc[i + 1, 'SGD'] = np.mean(run_hist_SGD[i].history["val_accuracy"])
    pd_hist_mean_test.loc[i + 1 , 'RMSprop'] = np.mean(run_hist_rmsprop[i].history["val_accuracy"])
```

Mean Accuracy for Train Dataset:

	Adam	SGD	RMSprop
1	0.827270	0.736129	0.833877
2	0.848484	0.709313	0.850149
3	0.878646	0.650618	0.836251
4	0.874230	0.651042	0.867688

Mean Accuracy for Test Dataset:

	Adam	SGD	RMSprop
1	0.736247	0.710490	0.736125
2	0.722925	0.686009	0.723683
3	0.696694	0.650643	0.722700
4	0.719351	0.651042	0.703260

For the train dataset, using Adam as the optimizer with 3 hidden layers gave us the best mean accuracy, while for the test dataset, it is Adam with only a single hidden layer.

Even though using Adam can give us the best mean accuracy, the accuracy for the test dataset dips down when the models start to overfit to the train dataset. Based on this alone, using SGD as the optimizer with 1 or 2 hidden layers will be a better option as it is more prone to overfitting. For the test dataset, the accuracy even increases along as the train accuracy increases. Also, the accuracies for SGD does not fluctuate as much as the other models using Adam and RMSprop.

7. SUMMARY

From the results above, although using Adam can gives us the best overall accuracy, using SGD with 1 or 2 hidden layers can be a better option since the accuracy does not fluctuate as much as with the other models. SGD also seems more prone to overfitting.

Another takeout is that simply increasing the number of hidden layers alone does not increase the performance, in some models, the performance even drops with the increase number of hidden layers. This is most apparent with models using SGD as the optimizer.

8. SUGGESTION

- Need to investigate on why the accuracy curves flatten out for models using SGD with 3 and 4 hidden layers.
- Need to experiment with different network structures.
- Train the models with more epochs, especially with the ones using SGD as it seems the model can have better performance with additional epochs.
- Experiment with different activation functions for the hidden layers.