

Machine Learning HW3

Yara Alfouzan
443203087

DATA SET DESCRIPTION:

The Graduate Admission Dataset contains information about applicants to graduate programs. It includes various features that are typically considered during the admission process(GRE Scores,TOEFL Scores,University Rating,Statement of Purpose (SOP),Letter of Recommendation (LOR) Strength,Undergraduate GPA,Research Experience,Chance of Admit) , in this homework we will implement and train a neural network for a regression task using Keras using these various features to predict the "Chance of Admit."

Clear screenshots of the code, evaluations and outputs:

The screenshot shows a Jupyter Notebook interface with two main sections of code.

Part 1: Dataset Loading & Preprocessing

```
[2] # Import necessary libraries
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split, KFold
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
import matplotlib.pyplot as plt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam
import tensorflow as tf

[3] from google.colab import drive
drive.mount('/content/drive')

# 1. Load the dataset and display the first 5 rows.
df = pd.read_csv('/content/drive/MyDrive/graduate_admission_dataset(1).csv')
print(df.head())

# 2. Select the chance of admit as the target variable (y) and use all other features as the input features (X).
X = df.drop('Chance of Admit ', axis=1)
y = df['Chance of Admit ']

# 3. Standardize the features using the standard scaler.
scaler = StandardScaler()
```

Part 2: Build and Compile the Neural Network Model

```
# 3. Standardize the features using the standard scaler.
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# 4. Split the dataset into train (60%), validation (20%), and test (20%) sets with random state 42
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2, random_state=42)
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.25, random_state=42) # 0.25 x 0.8 = 0.2

# Convert y_test to a Numpy array for correct indexing
y_test = y_test.to_numpy()

# 5. Build the neural network model
model = Sequential([
    Dense(64, activation='relu', input_shape=(X_scaled.shape[1],)),
    Dense(32, activation='relu'),
    Dense(1, activation='linear')
])

model.compile(optimizer=Adam(), loss='mean_squared_error')

# Train the model
history = model.fit(X_train, y_train, validation_data=(X_val, y_val), epochs=50, batch_size=32)
```

Data Output:

	Serial No.	GRE Score	TOEFL Score	University Rating	SOP	LOR	CGPA	\
0	1	337	118	4	4.5	4.5	9.65	
1	2	324	107	4	4.0	4.5	8.87	
2	3	316	104	3	3.0	3.5	8.00	
3	4	322	110	3	3.5	2.5	8.67	
4	5	314	103	2	2.0	3.0	8.21	

	Research	Chance of Admit
0	1	0.92
1	1	0.76
2	1	0.72
3	1	0.80
4	0	0.65

Part 2: Build and Compile the Neural Network Model

```

# 1. Complete the function build_model to build a simple feedforward neural network with the following architecture:
#   * Input layer: Based on the number of features in the dataset.
#   * Hidden layer: 10 neurons with the ReLU activation function.
#   * Output layer: A single neuron with a linear activation function for regression.

def build_model():
    model = Sequential()
    model.add(Dense(10, activation='relu', input_dim=X_train.shape[1]))
    model.add(Dense(1, activation='linear'))

    # 2. Compile the model using Mean Squared Error (MSE) as the loss function and the Adam optimizer with a learning rate of 0.01.

    model.compile(loss='mean_squared_error', optimizer=Adam(learning_rate=0.01))

    # 3. Train the model using the training set and validate on the validation set with 50 epochs and batch size 32.
    model = build_model()
    # Train the model
    history = model.fit(X_train, y_train, epochs=50, batch_size=32, validation_data=(X_val, y_val))

    # Plot training and validation loss over epochs
    plt.plot(history.history['loss'], label='Training Loss')
    plt.plot(history.history['val_loss'], label='Validation Loss')
    plt.title('Training and Validation Loss Over Epochs')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()
    plt.grid()
    plt.show()

```

... /usr/local/lib/python3.10/dist-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an `input_shape`/'input_dim' argument to a layer. We will ignore it.
super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Epoch 1/50
10/10 [0s 47ms/step - loss: 0.4430 - val_loss: 0.2122]



Q ▾ Part 3: Evaluation Using Cross-Validation

```
[x] 16s
# 1. Perform 5-fold cross-validation on the test set.
kf = KFold(n_splits=5, shuffle=True, random_state=42)

# Store metrics for each fold
fold_mse = []
fold_mae = []
fold_r2 = []

for train_index, test_index in kf.split(X_test):
    X_fold_train, X_fold_test = X_test[train_index], X_test[test_index]
    y_fold_train, y_fold_test = y_test[train_index], y_test[test_index]

    # Train the model on the training fold
    model = build_model() # Create a new model for each fold
    model.fit(X_fold_train, y_fold_train, epochs=50, batch_size=32, verbose=0) # Train on the training fold

    # Make predictions on the test fold
    y_pred = model.predict(X_fold_test)

    # 2. For each fold, evaluate the model using the Mean Absolute Error (MAE), Mean Squared Error (MSE), and R2 score.
    mse = mean_squared_error(y_fold_test, y_pred)
    mae = mean_absolute_error(y_fold_test, y_pred)
    r2 = r2_score(y_fold_test, y_pred)

    fold_mse.append(mse)
    fold_mae.append(mae)
    fold_r2.append(r2)

# 3. Report the average MAE, MSE, and R2 for the test set.
print("LR=0.01 and batch size=32")
print("Average MSE across folds:", np.mean(fold_mse))
15s completed at 11:13 PM
```

```
# 3. Report the average MAE, MSE, and R2 for the test set.
print("LR=0.01 and batch size=32")
print("Average MSE across folds:", np.mean(fold_mse))
print("Average MAE across folds:", np.mean(fold_mae))
print("Average R2 across folds:", np.mean(fold_r2))

super().__init__(activity_regularizer=activity_regularizer, **kwargs)
1/1 0s 50ms/step
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an `input_shape`/'input_dim' argument to a layer. When using functional API, pass input_shape to the first layer in your model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
1/1 0s 47ms/step
LR=0.01 and batch size=32
Average MSE across folds: 0.02605137974181109
Average MAE across folds: 0.13184069597125053
Average R2 across folds: -0.3001517723133986
```

A table displaying printed results (e.g., model parameters and/or predicted values):

MSE , MAE ,AND R²:

Here are some of the outputs at several LR and batch sizes specified in the printed statement

Learning Rate 0.01:

```
1/1 0s 71ms/step
LR=0.01 and batch size=64
Average MSE across folds: 0.022494376408904886
Average MAE across folds: 0.11348837693929673
Average R2 across folds: -0.1731125999887509
```

```
1/1 0s 46ms/step
LR=0.01 and batch size=16
Average MSE across folds: 0.027150647594965893
Average MAE across folds: 0.11649344310760497
Average R2 across folds: -0.41071177729256475
```

```

1/1 ━━━━━━ 0s 67ms/step
LR=0.01 and batch size=32
Average MSE across folds: 0.015125461536795535
Average MAE across folds: 0.10042175749540327
Average R2 across folds: 0.22646966545138278

```

Learning Rate 0.001

```

1/1 ━━━━━━ 0s 45ms/step
LR=0.001 and batch size=64
Average MSE across folds: 0.6290488938318151
Average MAE across folds: 0.6137113001346588
Average R2 across folds: -34.098611630351236

```

```

1/1 ━━━━━━ 0s 61ms/step
LR=0.001 and batch size=32
Average MSE across folds: 0.23625051766685906
Average MAE across folds: 0.39365089815855
Average R2 across folds: -11.475281351952836

```

```

1/1 ━━━━━━ 0s 43ms/step
LR=0.001 and batch size=16
Average MSE across folds: 0.16972428482415342
Average MAE across folds: 0.326173982706666
Average R2 across folds: -7.441868100188134

```

Learning Rate 0.1:

```

1/1 ━━━━━━ 0s 49ms/step
LR=0.1 and batch size=16
Average MSE across folds: 0.006227111004786278
Average MAE across folds: 0.06007657756805419
Average R2 across folds: 0.6817006173976485

```

```

super().__init__(activity_regularizer=activity_regularizer, **kwargs)
1/1 ━━━━━━ 0s 66ms/step
LR=0.1 and batch size=32
Average MSE across folds: 0.005704663499543468
Average MAE across folds: 0.056855614328384395
Average R2 across folds: 0.7104786662381615

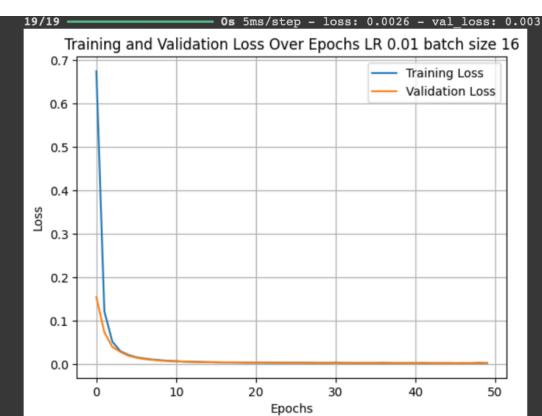
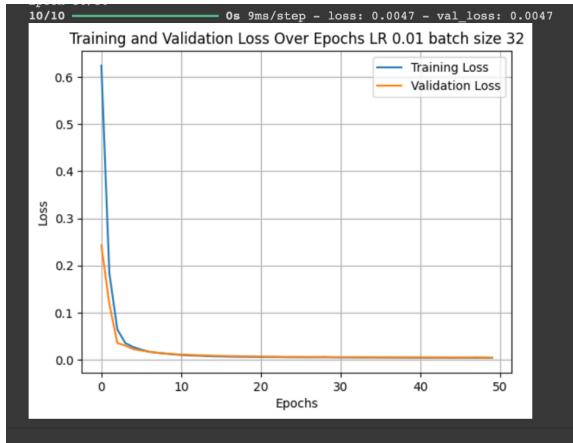
```

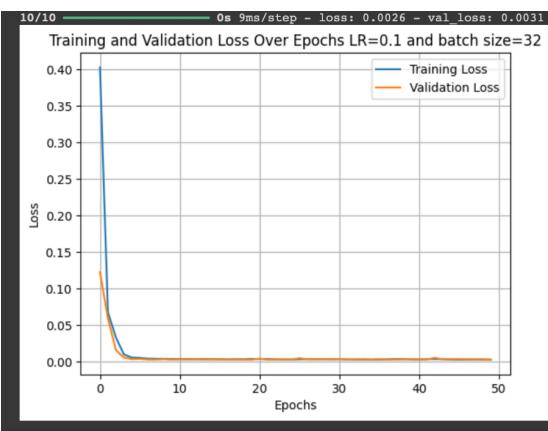
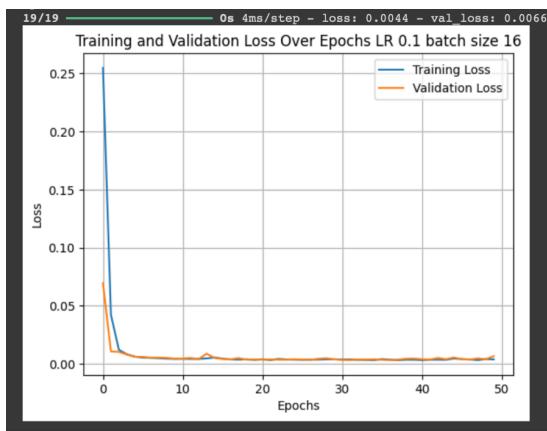
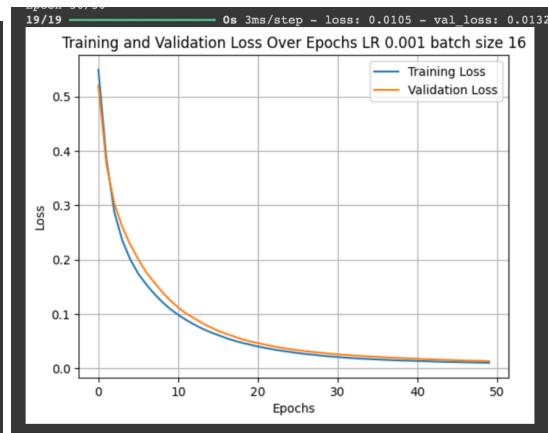
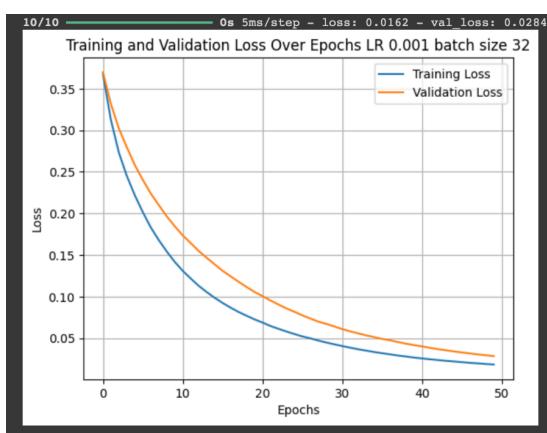
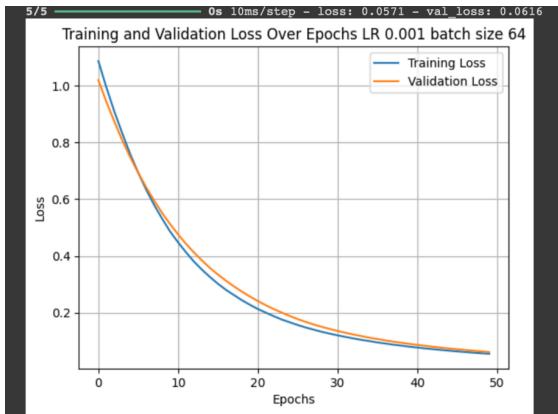
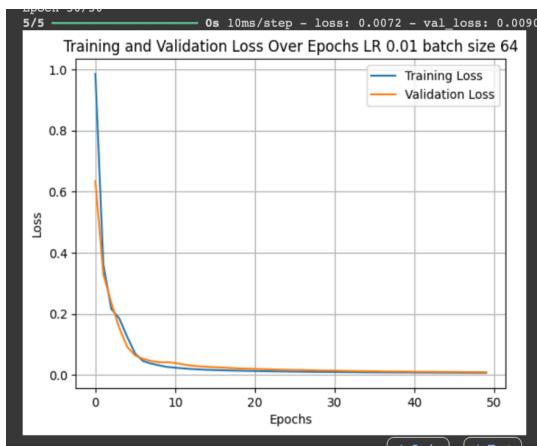
```

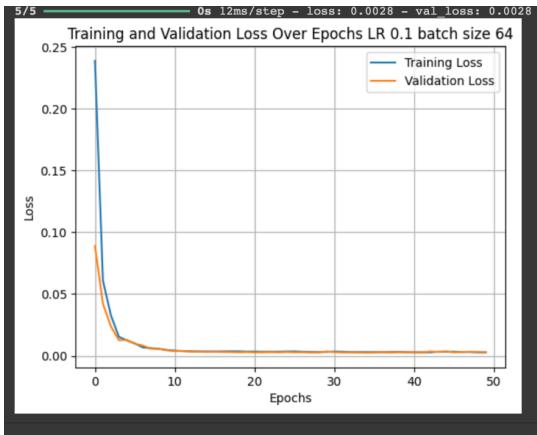
super().__init__(activity_regularizer=activity_regularizer, **kwargs)
1/1 ━━━━━━ 0s 43ms/step
LR=0.1 and batch size=64
Average MSE across folds: 0.010539361052626284
Average MAE across folds: 0.07542648433446883
Average R2 across folds: 0.5026643365211059

```

Some graph outputs using several learning rate and batch sizes as specified in the title of the graph:







Part 3:

1.Experiment with different learning rates (e.g., 0.001, 0.01, 0.1) and compare the effect on convergence.

Since the outputs are slightly different with every run due to that when a neural network is created, the weights of the neurons are initialized randomly. These weights determine how the network processes inputs and learns over time. Even slight differences in weight initialization can lead to different learning dynamics and, consequently, different results. A few other reasons also affect the variability of the outputs such as data shuffling.

Therefore in general :

Low LR :

Leads to slower convergence, but it may help the model find a more precise local minimum. If too small, it might take too long to converge or get stuck in local minima.

High LR :

Leads to faster convergence initially, but the risk of overshooting the optimal minimum is higher, which can cause the model to diverge.

In the **screenshot graph results** presented earlier if we fix batch size to 32 and compare we can see that a learning rate of **0.1** seems to converge the fastest , A learning rate of **0.01** converges almost the same as 0.1 but slightly slower, and a learning rate of **0.001** presented slow convergence.

2. Experiment with different batch sizes (e.g., 16, 32, 64) and evaluate the performance.

From the results presented in the screenshots, batch size 32 seemed to give the best performance since it gave lowest MAE, MSE and highest R² when the LR was 0.1 and 0.01 but batch size 16 performed slightly better on LR 0.001, batch size 64 was the worst in LR 0.1 and 0.001 and second worst at 0.01; in general small batch sizes might offer better generalization but with more variability in loss per epoch. It can sometimes improve performance metrics due to better exploration of the loss surface. Larger batch sizes are more likely to lead to poorer generalization, and potential increases in MAE and MSE and a lower R² on the test set. larger batch size converges faster but the model could get stuck in local minima.

In our case batch size 32 seems to be the most appropriate.

With LR fixed to 0.01:

As we can see in this example the best performance was presented by batch size 32.

Batch size	MAE	MSE	R ²
16	0.116	0.027	-0.410
32	0.100	0.015	0.226
64	0.113	0.022	-0.173