Assignment-3

May 15, 2024

[1]:
```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from keras.models import Sequential
from keras.layers import Dense, Input
import statistics
```

The dataset is about the compressive strength of different samples of concrete based on the volumes of the different ingredients that were used to make them. Ingredients include:

1. Cement
2. Blast Furnace Slag
3. Fly Ash
4. Water
5. Superplasticizer
6. Coarse Aggregate
7. Fine Aggregate

0.0.1 Load Dataset

[2]:
```
data = pd.read_csv("./data/concrete_data.csv")
data.head()
```

[2]:
```
   Cement  Blast Furnace Slag  Fly Ash  Water  Superplasticizer  \
0   540.0                 0.0      0.0  162.0               2.5
1   540.0                 0.0      0.0  162.0               2.5
2   332.5               142.5      0.0  228.0               0.0
3   332.5               142.5      0.0  228.0               0.0
4   198.6               132.4      0.0  192.0               0.0

   Coarse Aggregate  Fine Aggregate  Age  Strength
0            1040.0           676.0   28     79.99
1            1055.0           676.0   28     61.89
2             932.0           594.0  270     40.27
3             932.0           594.0  365     41.05
```

```
 4             978.4           825.5  360     44.30
```

[4]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1030 entries, 0 to 1029
Data columns (total 9 columns):
 #   Column              Non-Null Count  Dtype  
---  ------              --------------  -----  
 0   Cement              1030 non-null   float64
 1   Blast Furnace Slag  1030 non-null   float64
 2   Fly Ash             1030 non-null   float64
 3   Water               1030 non-null   float64
 4   Superplasticizer    1030 non-null   float64
 5   Coarse Aggregate    1030 non-null   float64
 6   Fine Aggregate      1030 non-null   float64
 7   Age                 1030 non-null   int64  
 8   Strength            1030 non-null   float64
dtypes: float64(8), int64(1)
memory usage: 72.6 KB
```

0.0.2 Define Features and Target

[5]:
```
X = data.drop('Strength', axis=1)
y = data['Strength']
```

0.1 Part A: Build a Baseline Model

0.1.1 1. Build the Neural Network Model

[6]:
```
def build_model():
    model = Sequential()
    model.add(Input(shape=(X.shape[1],)))
    model.add(Dense(10, activation='relu'))
    model.add(Dense(1))
    model.compile(optimizer='adam', loss='mean_squared_error')
    return model
```

0.1.2 2. Training and Evaluating the Model

[7]:
```
def train_and_evaluate_model():
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,␣
 ↪random_state=None)
    model = build_model()
    model.fit(X_train, y_train, epochs=50, verbose=0)
    y_pred = model.predict(X_test)
    mse = mean_squared_error(y_test, y_pred)
    return mse
```

```python
mse_list = [train_and_evaluate_model() for _ in range(50)]
mean_mse = np.mean(mse_list)
std_mse = np.std(mse_list)

print(f"Mean MSE: {mean_mse}")
print(f"Standard Deviation of MSE: {std_mse}")
```

```
10/10                 0s 3ms/step
10/10                 0s 3ms/step
10/10                 0s 3ms/step
10/10                 0s 3ms/step
10/10                 0s 3ms/step
10/10                 0s 3ms/step
10/10                 0s 3ms/step
10/10                 0s 3ms/step
10/10                 0s 3ms/step
10/10                 0s 3ms/step
10/10                 0s 3ms/step
10/10                 0s 3ms/step
10/10                 0s 3ms/step
10/10                 0s 3ms/step
10/10                 0s 3ms/step
10/10                 0s 3ms/step
10/10                 0s 3ms/step
10/10                 0s 3ms/step
10/10                 0s 3ms/step
10/10                 0s 3ms/step
10/10                 0s 3ms/step
10/10                 0s 3ms/step
10/10                 0s 3ms/step
10/10                 0s 3ms/step
10/10                 0s 3ms/step
10/10                 0s 3ms/step
10/10                 0s 3ms/step
10/10                 0s 3ms/step
10/10                 0s 3ms/step
10/10                 0s 3ms/step
10/10                 0s 3ms/step
10/10                 0s 3ms/step
10/10                 0s 3ms/step
10/10                 0s 3ms/step
10/10                 0s 3ms/step
10/10                 0s 3ms/step
10/10                 0s 3ms/step
10/10                 0s 3ms/step
10/10                 0s 44ms/step
10/10                 0s 3ms/step
```

```
10/10                 0s 3ms/step
10/10                 0s 3ms/step
10/10                 0s 3ms/step
10/10                 0s 3ms/step
10/10                 0s 3ms/step
10/10                 0s 3ms/step
10/10                 0s 3ms/step
10/10                 0s 3ms/step
10/10                 0s 3ms/step
10/10                 0s 3ms/step
Mean MSE: 347.825765335021
Standard Deviation of MSE: 420.8844012809329
```

0.2 Part B: Normalize the Data

0.2.1 1. Normalize the Data

[9]:
```python
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
X_normalized = scaler.fit_transform(X)
```

0.3 2. Repeat Training and Evaluation with Normalized Data

[10]:
```python
def train_and_evaluate_model_normalized():
    X_train, X_test, y_train, y_test = train_test_split(X_normalized, y,␣
 ↪test_size=0.3, random_state=None)
    model = build_model()
    model.fit(X_train, y_train, epochs=50, verbose=0)
    y_pred = model.predict(X_test)
    mse = mean_squared_error(y_test, y_pred)
    return mse

mse_list_normalized = [train_and_evaluate_model_normalized() for _ in range(50)]
mean_mse_normalized = np.mean(mse_list_normalized)
std_mse_normalized = np.std(mse_list_normalized)

print(f"Mean MSE (Normalized): {mean_mse_normalized}")
print(f"Standard Deviation of MSE (Normalized): {std_mse_normalized}")
```

```
10/10                 0s 3ms/step
10/10                 0s 3ms/step
10/10                 0s 3ms/step
10/10                 0s 3ms/step
10/10                 0s 3ms/step
10/10                 0s 3ms/step
10/10                 0s 3ms/step
10/10                 0s 3ms/step
```

```
10/10             0s 3ms/step
10/10             0s 3ms/step
10/10             0s 3ms/step
10/10             0s 3ms/step
10/10             0s 3ms/step
10/10             0s 3ms/step
10/10             0s 3ms/step
10/10             0s 3ms/step
10/10             0s 3ms/step
10/10             0s 3ms/step
10/10             0s 3ms/step
10/10             0s 3ms/step
10/10             0s 3ms/step
10/10             0s 3ms/step
10/10             0s 3ms/step
10/10             0s 3ms/step
10/10             0s 3ms/step
10/10             0s 3ms/step
10/10             0s 3ms/step
10/10             0s 3ms/step
10/10             0s 3ms/step
10/10             0s 3ms/step
10/10             0s 3ms/step
10/10             0s 3ms/step
10/10             0s 3ms/step
10/10             0s 3ms/step
10/10             0s 3ms/step
10/10             0s 3ms/step
10/10             0s 3ms/step
10/10             0s 3ms/step
10/10             0s 3ms/step
10/10             0s 3ms/step
10/10             0s 3ms/step
10/10             0s 3ms/step
10/10             0s 3ms/step
10/10             0s 3ms/step
10/10             0s 3ms/step
10/10             0s 3ms/step
10/10             0s 3ms/step
10/10             0s 3ms/step
10/10             0s 3ms/step
10/10             0s 3ms/step
Mean MSE (Normalized): 333.549363135248
Standard Deviation of MSE (Normalized): 72.81586897370408
```

0.4 How does the mean of the mean squared errors compare to that from Step A?

- The baseline model's mean MSE (347.826) is slightly higher than the mean MSE (333.549) for the normalized data. This suggests that the model performed slightly better after the data were normalized, as evidenced by the smaller average error.
- Compared to the baseline model (420.884), the standard deviation of the MSE for the normalized data (72.816) is substantially smaller. Given that the errors vary less between iterations, this suggests that the model performs more consistently and steadily when the data is normalized.

0.5 Part C: Increase the Number of Epochs

[12]:
```
def train_and_evaluate_model_normalized_100_epochs():
    X_train, X_test, y_train, y_test = train_test_split(X_normalized, y,␣
 ↪test_size=0.3, random_state=None)
    model = build_model()
    model.fit(X_train, y_train, epochs=100, verbose=0)
    y_pred = model.predict(X_test)
    mse = mean_squared_error(y_test, y_pred)
    return mse

mse_list_normalized_100_epochs =␣
 ↪[train_and_evaluate_model_normalized_100_epochs() for _ in range(50)]
mean_mse_normalized_100_epochs = np.mean(mse_list_normalized_100_epochs)
std_mse_normalized_100_epochs = np.std(mse_list_normalized_100_epochs)

print(f"Mean MSE (Normalized, 100 epochs): {mean_mse_normalized_100_epochs}")
print(f"Standard Deviation of MSE (Normalized, 100 epochs):␣
 ↪{std_mse_normalized_100_epochs}")
```

```
10/10                 0s 5ms/step
10/10                 0s 4ms/step
10/10                 0s 4ms/step
10/10                 0s 4ms/step
10/10                 0s 3ms/step
10/10                 0s 4ms/step
10/10                 0s 4ms/step
10/10                 0s 2ms/step
10/10                 0s 4ms/step
10/10                 0s 4ms/step
10/10                 0s 4ms/step
10/10                 0s 5ms/step
10/10                 0s 6ms/step
10/10                 0s 4ms/step
10/10                 0s 4ms/step
10/10                 0s 4ms/step
10/10                 0s 4ms/step
```

```
10/10            0s 4ms/step
10/10            0s 3ms/step
10/10            0s 4ms/step
10/10            0s 4ms/step
10/10            0s 3ms/step
10/10            0s 4ms/step
10/10            0s 3ms/step
10/10            0s 3ms/step
10/10            0s 3ms/step
10/10            0s 3ms/step
10/10            0s 3ms/step
10/10            0s 3ms/step
10/10            0s 3ms/step
10/10            0s 3ms/step
10/10            0s 3ms/step
10/10            0s 3ms/step
10/10            0s 3ms/step
10/10            0s 3ms/step
10/10            0s 3ms/step
10/10            0s 3ms/step
10/10            0s 3ms/step
10/10            0s 3ms/step
10/10            0s 3ms/step
10/10            0s 3ms/step
10/10            0s 3ms/step
10/10            0s 3ms/step
10/10            0s 3ms/step
10/10            0s 3ms/step
10/10            0s 3ms/step
10/10            0s 3ms/step
10/10            0s 3ms/step
10/10            0s 3ms/step
10/10            0s 3ms/step
Mean MSE (Normalized, 100 epochs): 165.41870796391126
Standard Deviation of MSE (Normalized, 100 epochs): 15.968462810337952
```

0.6 How does the mean of the mean squared errors compare to that from Step B?

- Compared to the mean MSE for the normalized data with 50 epochs (333.549), the mean MSE for the normalized data with 100 epochs (165.419) is significantly lower. This suggests that the model's performance has significantly improved with an increase in epochs, leading to a significantly lower average error.
- Compared to the normalized data with 50 epochs (72.816), the standard deviation of the MSE for the data with 100 epochs is substantially lower (15.968). Given that there is less variation in the errors between iterations, it can be inferred that the model performs better when trained over a longer time span (100 epochs).

0.7 Part D: Increase the Number of Hidden Layers

0.7.1 1. Build a Model with Three Hidden Layers

[15]:
```
def build_model_three_layers():
    model = Sequential()
    model.add(Input(shape=(X.shape[1],)))  # Correct way to specify input shape
    model.add(Dense(10, activation='relu'))
    model.add(Dense(10, activation='relu'))
    model.add(Dense(10, activation='relu'))
    model.add(Dense(1))
    model.compile(optimizer='adam', loss='mean_squared_error')
    return model
```

0.7.2 2. Train and Evaluate the Model with Three Hidden Layers

[17]:
```
def train_and_evaluate_model_three_layers():
    X_train, X_test, y_train, y_test = train_test_split(X_normalized, y,␣
 ↪test_size=0.3, random_state=None)
    model = build_model_three_layers()
    model.fit(X_train, y_train, epochs=50, verbose=0)
    y_pred = model.predict(X_test)
    mse = mean_squared_error(y_test, y_pred)
    return mse

mse_list_three_layers = [train_and_evaluate_model_three_layers() for _ in␣
 ↪range(50)]
mean_mse_three_layers = np.mean(mse_list_three_layers)
std_mse_three_layers = np.std(mse_list_three_layers)

print(f"Mean MSE (Normalized, Three Layers): {mean_mse_three_layers}")
print(f"Standard Deviation of MSE (Normalized, Three Layers):␣
 ↪{std_mse_three_layers}")
```

```
10/10             0s 4ms/step
10/10             0s 4ms/step
10/10             0s 4ms/step
10/10             0s 4ms/step
10/10             0s 4ms/step
10/10             0s 4ms/step
10/10             0s 4ms/step
10/10             0s 4ms/step
10/10             0s 4ms/step
10/10             0s 4ms/step
10/10             0s 4ms/step
10/10             0s 4ms/step
10/10             0s 4ms/step
10/10             0s 4ms/step
```

```
10/10               0s 4ms/step
10/10               0s 4ms/step
10/10               0s 4ms/step
10/10               0s 4ms/step
10/10               0s 4ms/step
10/10               0s 4ms/step
10/10               0s 4ms/step
10/10               0s 5ms/step
10/10               0s 4ms/step
10/10               0s 4ms/step
10/10               0s 4ms/step
10/10               0s 4ms/step
10/10               0s 5ms/step
10/10               0s 4ms/step
10/10               0s 4ms/step
10/10               0s 4ms/step
10/10               0s 4ms/step
10/10               0s 4ms/step
10/10               0s 4ms/step
10/10               0s 5ms/step
10/10               0s 4ms/step
10/10               0s 4ms/step
10/10               0s 4ms/step
10/10               0s 4ms/step
10/10               0s 4ms/step
10/10               0s 4ms/step
10/10               0s 4ms/step
10/10               0s 4ms/step
10/10               0s 4ms/step
10/10               0s 4ms/step
10/10               0s 4ms/step
10/10               0s 4ms/step
10/10               0s 4ms/step
10/10               0s 4ms/step
10/10               0s 4ms/step
10/10               0s 3ms/step
Mean MSE (Normalized, Three Layers): 129.35663797827036
Standard Deviation of MSE (Normalized, Three Layers): 13.459531283588293
```

0.8 How does the mean of the mean squared errors compare to that from Step B?

- Compared to the mean MSE of 333.549 for normalized data with one hidden layer, the mean MSE of 129.357 for normalized data with three hidden layers is substantially lower. This shows that the model's performance has significantly improved with more hidden layers, as evidenced by the significantly lower average error.

- Compared to the normalized data with one hidden layer (72.816), the standard deviation of the MSE for the data with three hidden layers (13.460) is substantially smaller. Given that the errors vary less between iterations, this suggests that the model's performance is more stable and consistent when it has three hidden layers.