m7vxdqs4n

April 18, 2025

0.1 Linear regression by using Deep Neural network: Implement Boston housing price prediction problem by Linear regression using Deep Neural Network. Use Boston House Price prediction Dataset.

[1]:
```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from tensorflow.keras.optimizers import Adam
```

[2]:
```
df = pd.read_csv("HousingData.csv")
df.head()
```

[2]:
```
      CRIM    ZN  INDUS  CHAS    NOX     RM   AGE     DIS  RAD  TAX  PTRATIO  \
0  0.00632  18.0   2.31   0.0  0.538  6.575  65.2  4.0900    1  296     15.3
1  0.02731   0.0   7.07   0.0  0.469  6.421  78.9  4.9671    2  242     17.8
2  0.02729   0.0   7.07   0.0  0.469  7.185  61.1  4.9671    2  242     17.8
3  0.03237   0.0   2.18   0.0  0.458  6.998  45.8  6.0622    3  222     18.7
4  0.06905   0.0   2.18   0.0  0.458  7.147  54.2  6.0622    3  222     18.7

        B  LSTAT  MEDV
0  396.90   4.98  24.0
1  396.90   9.14  21.6
2  392.83   4.03  34.7
3  394.63   2.94  33.4
4  396.90    NaN  36.2
```

[3]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 506 entries, 0 to 505
Data columns (total 14 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
```

```
 0   CRIM     486 non-null    float64
 1   ZN       486 non-null    float64
 2   INDUS    486 non-null    float64
 3   CHAS     486 non-null    float64
 4   NOX      506 non-null    float64
 5   RM       506 non-null    float64
 6   AGE      486 non-null    float64
 7   DIS      506 non-null    float64
 8   RAD      506 non-null    int64
 9   TAX      506 non-null    int64
 10  PTRATIO  506 non-null    float64
 11  B        506 non-null    float64
 12  LSTAT    486 non-null    float64
 13  MEDV     506 non-null    float64
dtypes: float64(12), int64(2)
memory usage: 55.5 KB
```

[4]:
```
df.fillna(df.mean(), inplace=True)
```

[5]:
```
X = df.drop(columns=['MEDV'])
y = df['MEDV']
```

[6]:
```
scaler = StandardScaler()
X = scaler.fit_transform(X)
```

[7]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
↪random_state=42)
```

[8]:
```
model = keras.Sequential([
    keras.Input(shape=(X.shape[1],)),  # Explicit Input Layer
    layers.Dense(64, activation='relu'),  # Hidden Layer 1
    layers.Dense(32, activation='relu'),  # Hidden Layer 2
    layers.Dense(1, activation='linear')  # Output layer (Regression)
])
```

[9]:
```
model.compile(optimizer=Adam(learning_rate=0.01), loss='mse', metrics=['mae'])
```

[10]:
```
history = model.fit(X_train, y_train, epochs=100, validation_data=(X_test,␣
↪y_test), batch_size=16, verbose=1)
```

```
Epoch 1/100
26/26                 1s 8ms/step - loss:
440.8634 - mae: 18.5338 - val_loss: 57.2412 - val_mae: 5.5565
Epoch 2/100
26/26                 0s 2ms/step - loss:
45.3372 - mae: 5.1544 - val_loss: 21.6366 - val_mae: 3.2939
Epoch 3/100
26/26                 0s 3ms/step - loss:
```

```
20.0994 - mae: 3.3215 - val_loss: 17.6297 - val_mae: 2.8288
Epoch 4/100
26/26            0s 4ms/step - loss:
17.6236 - mae: 3.0781 - val_loss: 14.9943 - val_mae: 2.6249
Epoch 5/100
26/26            0s 3ms/step - loss:
14.2821 - mae: 2.7823 - val_loss: 14.5623 - val_mae: 2.3381
Epoch 6/100
26/26            0s 2ms/step - loss:
13.6029 - mae: 2.6609 - val_loss: 12.9750 - val_mae: 2.4203
Epoch 7/100
26/26            0s 3ms/step - loss:
12.8029 - mae: 2.5281 - val_loss: 13.4118 - val_mae: 2.3723
Epoch 8/100
26/26            0s 3ms/step - loss:
14.5553 - mae: 2.7077 - val_loss: 13.8585 - val_mae: 2.4914
Epoch 9/100
26/26            0s 3ms/step - loss:
13.7882 - mae: 2.5705 - val_loss: 15.7087 - val_mae: 2.6962
Epoch 10/100
26/26            0s 3ms/step - loss:
10.6948 - mae: 2.4719 - val_loss: 15.3429 - val_mae: 2.6446
Epoch 11/100
26/26            0s 3ms/step - loss:
14.5759 - mae: 2.7952 - val_loss: 14.6829 - val_mae: 2.7426
Epoch 12/100
26/26            0s 3ms/step - loss:
12.2882 - mae: 2.5158 - val_loss: 12.0716 - val_mae: 2.3263
Epoch 13/100
26/26            0s 3ms/step - loss:
9.0725 - mae: 2.1205 - val_loss: 14.1640 - val_mae: 2.4194
Epoch 14/100
26/26            0s 3ms/step - loss:
8.8355 - mae: 2.2726 - val_loss: 14.3075 - val_mae: 2.5564
Epoch 15/100
26/26            0s 3ms/step - loss:
10.8129 - mae: 2.4245 - val_loss: 13.6643 - val_mae: 2.4626
Epoch 16/100
26/26            0s 4ms/step - loss:
9.6793 - mae: 2.4046 - val_loss: 13.4059 - val_mae: 2.5891
Epoch 17/100
26/26            0s 2ms/step - loss:
10.1957 - mae: 2.4241 - val_loss: 15.5057 - val_mae: 2.7194
Epoch 18/100
26/26            0s 2ms/step - loss:
10.0053 - mae: 2.4237 - val_loss: 14.4027 - val_mae: 2.6199
Epoch 19/100
26/26            0s 3ms/step - loss:
```

```
8.8556 - mae: 2.3036 - val_loss: 15.0273 - val_mae: 2.5188
Epoch 20/100
26/26            0s 2ms/step - loss:
11.1176 - mae: 2.5842 - val_loss: 13.6674 - val_mae: 2.4945
Epoch 21/100
26/26            0s 3ms/step - loss:
8.4296 - mae: 2.2924 - val_loss: 12.5635 - val_mae: 2.4181
Epoch 22/100
26/26            0s 2ms/step - loss:
8.5037 - mae: 2.1967 - val_loss: 13.1299 - val_mae: 2.3947
Epoch 23/100
26/26            0s 2ms/step - loss:
9.4739 - mae: 2.2657 - val_loss: 12.7473 - val_mae: 2.5449
Epoch 24/100
26/26            0s 2ms/step - loss:
8.6330 - mae: 2.2670 - val_loss: 13.1207 - val_mae: 2.4739
Epoch 25/100
26/26            0s 3ms/step - loss:
8.5585 - mae: 2.2271 - val_loss: 11.9104 - val_mae: 2.2847
Epoch 26/100
26/26            0s 2ms/step - loss:
8.1119 - mae: 2.1407 - val_loss: 13.2038 - val_mae: 2.5670
Epoch 27/100
26/26            0s 2ms/step - loss:
8.2387 - mae: 2.2445 - val_loss: 12.3271 - val_mae: 2.4717
Epoch 28/100
26/26            0s 2ms/step - loss:
10.2736 - mae: 2.4373 - val_loss: 13.2581 - val_mae: 2.4999
Epoch 29/100
26/26            0s 3ms/step - loss:
9.6498 - mae: 2.2557 - val_loss: 13.1113 - val_mae: 2.5810
Epoch 30/100
26/26            0s 2ms/step - loss:
8.0430 - mae: 2.0943 - val_loss: 16.5543 - val_mae: 2.6920
Epoch 31/100
26/26            0s 2ms/step - loss:
9.1912 - mae: 2.3014 - val_loss: 13.0773 - val_mae: 2.4636
Epoch 32/100
26/26            0s 2ms/step - loss:
6.8917 - mae: 2.0088 - val_loss: 13.4819 - val_mae: 2.4822
Epoch 33/100
26/26            0s 2ms/step - loss:
8.8703 - mae: 2.3257 - val_loss: 13.6802 - val_mae: 2.7276
Epoch 34/100
26/26            0s 2ms/step - loss:
9.3434 - mae: 2.3046 - val_loss: 11.9395 - val_mae: 2.2687
Epoch 35/100
26/26            0s 3ms/step - loss:
```

```
6.3509 - mae: 1.9543 - val_loss: 13.7159 - val_mae: 2.5541
Epoch 36/100
26/26            0s 2ms/step - loss:
8.4139 - mae: 2.1992 - val_loss: 14.7841 - val_mae: 2.5076
Epoch 37/100
26/26            0s 2ms/step - loss:
5.0453 - mae: 1.6810 - val_loss: 11.9874 - val_mae: 2.4051
Epoch 38/100
26/26            0s 2ms/step - loss:
7.4600 - mae: 2.0501 - val_loss: 13.0326 - val_mae: 2.4529
Epoch 39/100
26/26            0s 2ms/step - loss:
9.1731 - mae: 2.3376 - val_loss: 12.4445 - val_mae: 2.4682
Epoch 40/100
26/26            0s 2ms/step - loss:
9.4816 - mae: 2.3027 - val_loss: 11.6693 - val_mae: 2.4574
Epoch 41/100
26/26            0s 3ms/step - loss:
7.4163 - mae: 2.0914 - val_loss: 11.5701 - val_mae: 2.3401
Epoch 42/100
26/26            0s 1ms/step - loss:
6.8931 - mae: 2.0371 - val_loss: 13.6548 - val_mae: 2.4897
Epoch 43/100
26/26            0s 2ms/step - loss:
7.0245 - mae: 2.0263 - val_loss: 12.0620 - val_mae: 2.4173
Epoch 44/100
26/26            0s 2ms/step - loss:
6.6387 - mae: 1.9324 - val_loss: 12.6175 - val_mae: 2.3677
Epoch 45/100
26/26            0s 2ms/step - loss:
4.9877 - mae: 1.6471 - val_loss: 11.8034 - val_mae: 2.3359
Epoch 46/100
26/26            0s 2ms/step - loss:
6.4296 - mae: 1.8254 - val_loss: 14.0951 - val_mae: 2.4861
Epoch 47/100
26/26            0s 2ms/step - loss:
6.7578 - mae: 1.8863 - val_loss: 12.3211 - val_mae: 2.4534
Epoch 48/100
26/26            0s 2ms/step - loss:
7.8392 - mae: 2.1540 - val_loss: 19.0252 - val_mae: 2.9707
Epoch 49/100
26/26            0s 2ms/step - loss:
7.9342 - mae: 2.1704 - val_loss: 11.3792 - val_mae: 2.3989
Epoch 50/100
26/26            0s 3ms/step - loss:
8.6149 - mae: 2.0973 - val_loss: 12.0656 - val_mae: 2.4236
Epoch 51/100
26/26            0s 2ms/step - loss:
```

```
6.7459 - mae: 1.9316 - val_loss: 12.6347 - val_mae: 2.4684
Epoch 52/100
26/26            0s 2ms/step - loss:
8.0481 - mae: 2.1807 - val_loss: 12.8758 - val_mae: 2.3664
Epoch 53/100
26/26            0s 1ms/step - loss:
6.2552 - mae: 1.8587 - val_loss: 12.6247 - val_mae: 2.7060
Epoch 54/100
26/26            0s 2ms/step - loss:
6.8880 - mae: 2.0309 - val_loss: 11.8173 - val_mae: 2.3182
Epoch 55/100
26/26            0s 2ms/step - loss:
5.4851 - mae: 1.8217 - val_loss: 11.3842 - val_mae: 2.2862
Epoch 56/100
26/26            0s 2ms/step - loss:
5.5749 - mae: 1.8342 - val_loss: 13.7648 - val_mae: 2.4022
Epoch 57/100
26/26            0s 2ms/step - loss:
7.1245 - mae: 1.9695 - val_loss: 15.4390 - val_mae: 2.6652
Epoch 58/100
26/26            0s 2ms/step - loss:
6.0535 - mae: 1.8371 - val_loss: 12.1460 - val_mae: 2.4247
Epoch 59/100
26/26            0s 6ms/step - loss:
5.2226 - mae: 1.6742 - val_loss: 14.3516 - val_mae: 2.4435
Epoch 60/100
26/26            0s 2ms/step - loss:
5.8027 - mae: 1.7942 - val_loss: 11.3104 - val_mae: 2.3171
Epoch 61/100
26/26            0s 3ms/step - loss:
5.5879 - mae: 1.7765 - val_loss: 13.8340 - val_mae: 2.4806
Epoch 62/100
26/26            0s 3ms/step - loss:
5.5349 - mae: 1.7566 - val_loss: 13.4828 - val_mae: 2.4362
Epoch 63/100
26/26            0s 2ms/step - loss:
6.2231 - mae: 1.8632 - val_loss: 16.7503 - val_mae: 2.9819
Epoch 64/100
26/26            0s 2ms/step - loss:
7.2343 - mae: 2.0410 - val_loss: 12.8258 - val_mae: 2.7086
Epoch 65/100
26/26            0s 2ms/step - loss:
6.7012 - mae: 1.9879 - val_loss: 12.6499 - val_mae: 2.4648
Epoch 66/100
26/26            0s 2ms/step - loss:
6.2187 - mae: 1.9058 - val_loss: 16.2132 - val_mae: 2.6801
Epoch 67/100
26/26            0s 2ms/step - loss:
```

```
7.3492 - mae: 2.1787 - val_loss: 12.5516 - val_mae: 2.3207
Epoch 68/100
26/26              0s 2ms/step - loss:
8.4214 - mae: 2.2212 - val_loss: 13.8353 - val_mae: 2.8145
Epoch 69/100
26/26              0s 3ms/step - loss:
7.3707 - mae: 2.1419 - val_loss: 15.5721 - val_mae: 2.8678
Epoch 70/100
26/26              0s 2ms/step - loss:
6.5717 - mae: 2.0063 - val_loss: 13.7526 - val_mae: 2.4975
Epoch 71/100
26/26              0s 2ms/step - loss:
7.7291 - mae: 2.0831 - val_loss: 12.3777 - val_mae: 2.3361
Epoch 72/100
26/26              0s 5ms/step - loss:
5.2945 - mae: 1.7679 - val_loss: 13.3363 - val_mae: 2.7306
Epoch 73/100
26/26              0s 2ms/step - loss:
5.7156 - mae: 1.8765 - val_loss: 12.8553 - val_mae: 2.5685
Epoch 74/100
26/26              0s 3ms/step - loss:
5.7010 - mae: 1.8624 - val_loss: 14.7535 - val_mae: 2.3943
Epoch 75/100
26/26              0s 2ms/step - loss:
5.3147 - mae: 1.7076 - val_loss: 13.4687 - val_mae: 2.3820
Epoch 76/100
26/26              0s 2ms/step - loss:
4.5659 - mae: 1.5840 - val_loss: 13.4049 - val_mae: 2.4064
Epoch 77/100
26/26              0s 3ms/step - loss:
6.5707 - mae: 1.9099 - val_loss: 11.9541 - val_mae: 2.2850
Epoch 78/100
26/26              0s 2ms/step - loss:
5.2888 - mae: 1.7650 - val_loss: 14.1747 - val_mae: 2.6666
Epoch 79/100
26/26              0s 2ms/step - loss:
5.9275 - mae: 1.8753 - val_loss: 12.7242 - val_mae: 2.4817
Epoch 80/100
26/26              0s 2ms/step - loss:
4.5076 - mae: 1.6957 - val_loss: 11.2360 - val_mae: 2.3203
Epoch 81/100
26/26              0s 2ms/step - loss:
5.9954 - mae: 1.8458 - val_loss: 14.4281 - val_mae: 2.4375
Epoch 82/100
26/26              0s 2ms/step - loss:
5.1949 - mae: 1.7030 - val_loss: 11.7536 - val_mae: 2.2650
Epoch 83/100
26/26              0s 1ms/step - loss:
```

```
5.1825 - mae: 1.7165 - val_loss: 12.7035 - val_mae: 2.3677
Epoch 84/100
26/26            0s 3ms/step - loss:
5.5240 - mae: 1.7834 - val_loss: 12.4507 - val_mae: 2.4028
Epoch 85/100
26/26            0s 3ms/step - loss:
5.3263 - mae: 1.6635 - val_loss: 11.4906 - val_mae: 2.4123
Epoch 86/100
26/26            0s 2ms/step - loss:
5.4103 - mae: 1.7492 - val_loss: 11.7925 - val_mae: 2.2501
Epoch 87/100
26/26            0s 2ms/step - loss:
5.1477 - mae: 1.6802 - val_loss: 12.8947 - val_mae: 2.3426
Epoch 88/100
26/26            0s 2ms/step - loss:
4.7069 - mae: 1.6337 - val_loss: 12.2875 - val_mae: 2.2313
Epoch 89/100
26/26            0s 2ms/step - loss:
6.4071 - mae: 1.8286 - val_loss: 11.5631 - val_mae: 2.5627
Epoch 90/100
26/26            0s 3ms/step - loss:
5.6254 - mae: 1.8210 - val_loss: 12.5988 - val_mae: 2.2553
Epoch 91/100
26/26            0s 1ms/step - loss:
4.3570 - mae: 1.5867 - val_loss: 13.4870 - val_mae: 2.4412
Epoch 92/100
26/26            0s 2ms/step - loss:
3.7779 - mae: 1.4996 - val_loss: 13.0153 - val_mae: 2.3395
Epoch 93/100
26/26            0s 2ms/step - loss:
5.1520 - mae: 1.7093 - val_loss: 13.6739 - val_mae: 2.5711
Epoch 94/100
26/26            0s 2ms/step - loss:
5.6690 - mae: 1.7843 - val_loss: 12.1259 - val_mae: 2.3143
Epoch 95/100
26/26            0s 3ms/step - loss:
5.1884 - mae: 1.7702 - val_loss: 12.9545 - val_mae: 2.4932
Epoch 96/100
26/26            0s 2ms/step - loss:
4.8350 - mae: 1.6868 - val_loss: 11.5383 - val_mae: 2.2410
Epoch 97/100
26/26            0s 2ms/step - loss:
4.8329 - mae: 1.6504 - val_loss: 14.1497 - val_mae: 2.6293
Epoch 98/100
26/26            0s 2ms/step - loss:
5.1155 - mae: 1.7440 - val_loss: 11.5848 - val_mae: 2.3331
Epoch 99/100
26/26            0s 2ms/step - loss:
```

```
5.3956 - mae: 1.7245 - val_loss: 11.9814 - val_mae: 2.3072
Epoch 100/100
26/26            0s 2ms/step - loss:
3.7539 - mae: 1.5173 - val_loss: 11.3920 - val_mae: 2.2617
```

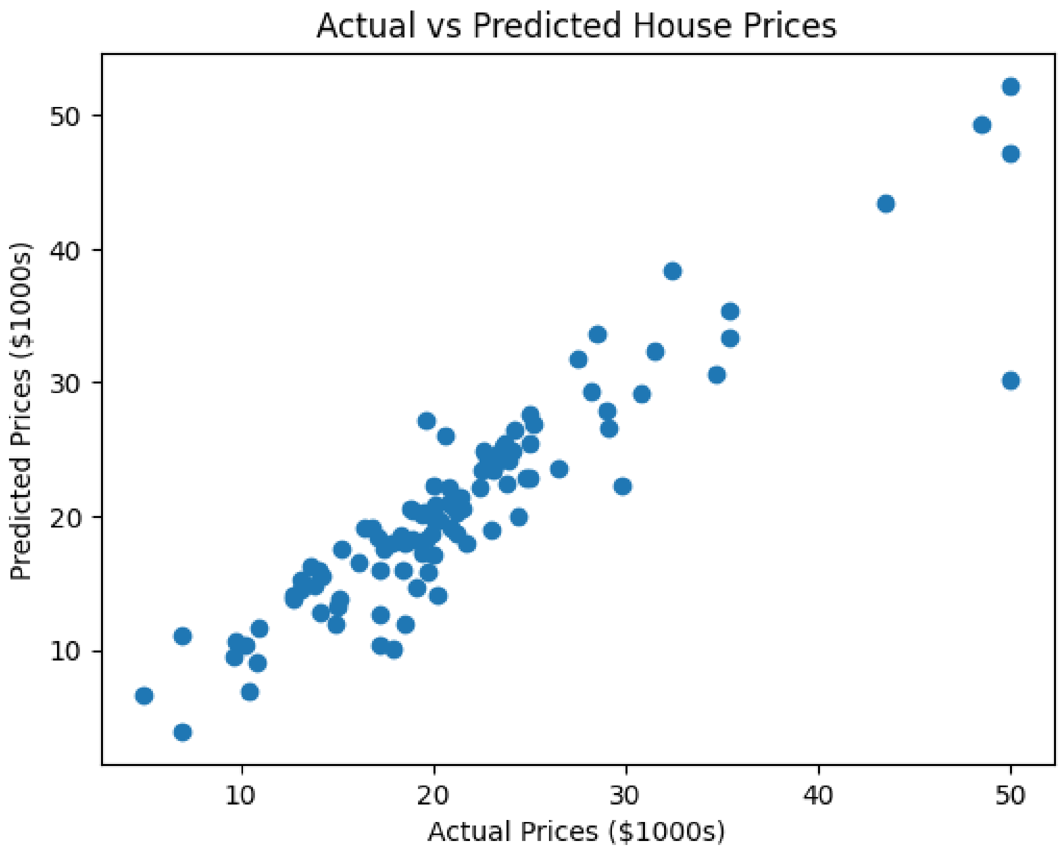
[11]:
```
loss, mae = model.evaluate(X_test, y_test)
print(f"Test Loss (MSE): {loss}")
print(f"Test Mean Absolute Error (MAE): {mae}")
```

```
4/4            0s 0s/step - loss:
8.9360 - mae: 2.1187
Test Loss (MSE): 11.391955375671387
Test Mean Absolute Error (MAE): 2.261707305908203
```

[12]:
```
predictions = model.predict(X_test)
```

```
4/4            0s 6ms/step
```

[13]:
```
plt.scatter(y_test, predictions)
plt.xlabel("Actual Prices ($1000s)")
plt.ylabel("Predicted Prices ($1000s)")
plt.title("Actual vs Predicted House Prices")
plt.show()
```



wxs98airg

April 18, 2025

0.1 Binary classification using Deep Neural Networks Example: Classify movie reviews into positive" reviews and "negative" reviews, just based on the text content of the reviews. Use IMDB dataset.

[1]:
```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import matplotlib.pyplot as plt
```

[2]:
```
vocab_size = 10000
max_length = 200
(x_train, y_train), (x_test, y_test) = keras.datasets.imdb.
 ↪load_data(num_words=vocab_size)
```

[3]:
```
x_train = keras.preprocessing.sequence.pad_sequences(x_train,␣
 ↪maxlen=max_length, padding='post')
x_test = keras.preprocessing.sequence.pad_sequences(x_test, maxlen=max_length,␣
 ↪padding='post')
```

[4]:
```
model = keras.Sequential([
    layers.Embedding(input_dim=vocab_size, output_dim=64),  # Removed␣
 ↪input_length
    layers.Conv1D(32, 5, activation='relu'),  # 1D Convolution for feature␣
 ↪extraction
    layers.GlobalMaxPooling1D(),  # Reduce dimensions
    layers.Dense(64, activation='relu'),  # Fully connected layer
    layers.Dense(1, activation='sigmoid')  # Output layer (Binary␣
 ↪classification)
])
```

[5]:
```
model.compile(optimizer='adam', loss='binary_crossentropy',␣
 ↪metrics=['accuracy'])
```

[6]:
```
history = model.fit(x_train, y_train, epochs=5, validation_data=(x_test,␣
 ↪y_test), batch_size=64, verbose=1)
```

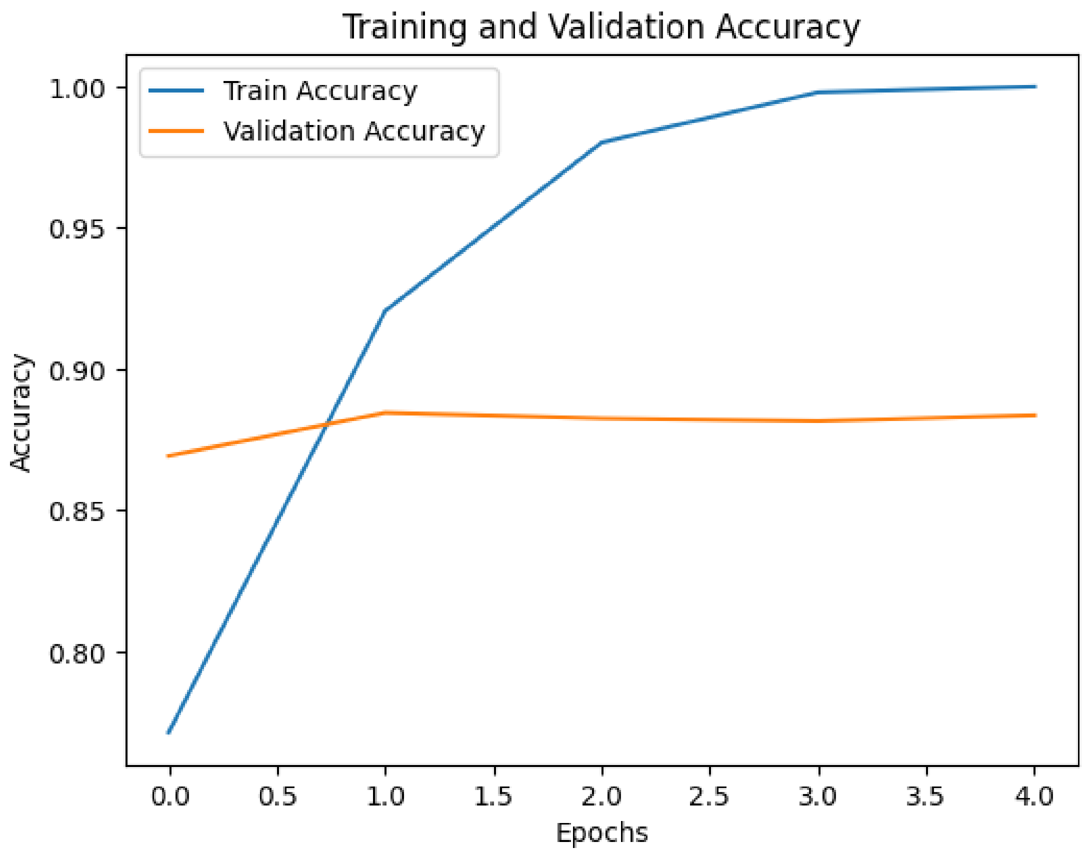
```
Epoch 1/5
391/391             8s 18ms/step -
```

```
accuracy: 0.6687 - loss: 0.5754 - val_accuracy: 0.8692 - val_loss: 0.3026
Epoch 2/5
391/391              7s 18ms/step -
accuracy: 0.9185 - loss: 0.2122 - val_accuracy: 0.8845 - val_loss: 0.2767
Epoch 3/5
391/391              7s 18ms/step -
accuracy: 0.9827 - loss: 0.0740 - val_accuracy: 0.8825 - val_loss: 0.3220
Epoch 4/5
391/391              7s 18ms/step -
accuracy: 0.9979 - loss: 0.0180 - val_accuracy: 0.8816 - val_loss: 0.3672
Epoch 5/5
391/391              7s 17ms/step -
accuracy: 1.0000 - loss: 0.0031 - val_accuracy: 0.8836 - val_loss: 0.4037
```

[7]:
```python
loss, accuracy = model.evaluate(x_test, y_test)
print(f"Test Accuracy: {accuracy:.4f}")
print(f"Test Loss: {loss:.4f}")
```

```
782/782              3s 4ms/step -
accuracy: 0.8828 - loss: 0.4049
Test Accuracy: 0.8836
Test Loss: 0.4037
```

[8]:
```python
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.title('Training and Validation Accuracy')
plt.show()
```


Training and Validation Accuracy
Train Accuracy
Validation Accuracy
Accuracy
Epochs
1.00
0.95
0.90
0.85
0.80
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0

rbezg8v4z

April 18, 2025

0.1 Convolutional neural network (CNN) (Any One from the following)

#### Use any dataset of plant disease and design a plant disease detection system using CNN. #### Use MNIST Fashion Dataset and create a classifier to classify fashion clothing into categories.

[1]:
```
import pandas as pd
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers, models
```

[2]:
```
train_df = pd.read_csv('fashion-mnist_train.csv')
test_df = pd.read_csv('fashion-mnist_test.csv')
```

[3]:
```
train_df.head(2)
```

[3]:
```
   label  pixel1  pixel2  pixel3  pixel4  pixel5  pixel6  pixel7  pixel8  \
0      2       0       0       0       0       0       0       0       0
1      9       0       0       0       0       0       0       0       0

   pixel9  …  pixel775  pixel776  pixel777  pixel778  pixel779  pixel780  \
0       0  …         0         0         0         0         0         0
1       0  …         0         0         0         0         0         0

   pixel781  pixel782  pixel783  pixel784
0         0         0         0         0
1         0         0         0         0

[2 rows x 785 columns]
```

[4]:
```
test_df.head(2)
```

[4]:
```
   label  pixel1  pixel2  pixel3  pixel4  pixel5  pixel6  pixel7  pixel8  \
0      0       0       0       0       0       0       0       0       9
1      1       0       0       0       0       0       0       0       0

   pixel9  …  pixel775  pixel776  pixel777  pixel778  pixel779  pixel780  \
0       8  …       103        87        56         0         0         0
```

```
1       0  …        34          0          0          0          0          0

   pixel781  pixel782  pixel783  pixel784
0         0         0         0         0
1         0         0         0         0

[2 rows x 785 columns]
```

[5]:
```
# Split features and labels
x_train = train_df.iloc[:, 1:].values
y_train = train_df.iloc[:, 0].values
```

[6]:
```
x_test = test_df.iloc[:, 1:].values
y_test = test_df.iloc[:, 0].values
```

[7]:
```
# Normalize pixel values
x_train = x_train / 255.0
x_test = x_test / 255.0
```

[8]:
```
# Reshape for CNN: (samples, height, width, channels)
x_train = x_train.reshape(-1, 28, 28, 1)
x_test = x_test.reshape(-1, 28, 28, 1)
```

[9]:
```
# Build CNN model
model = models.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
    layers.MaxPooling2D(2, 2),

    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D(2, 2),

    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dense(10, activation='softmax')
])
```

```
C:\Python312\Lib\site-packages\keras\src\layers\convolutional\base_conv.py:107:
UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When
using Sequential models, prefer using an `Input(shape)` object as the first
layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

[10]:
```
# Compile model
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

[11]:
```
# Train
model.fit(x_train, y_train, epochs=5, validation_split=0.1)
```

```
Epoch 1/5
1688/1688            13s 7ms/step -
accuracy: 0.7610 - loss: 0.6718 - val_accuracy: 0.8638 - val_loss: 0.3871
Epoch 2/5
1688/1688            12s 7ms/step -
accuracy: 0.8786 - loss: 0.3398 - val_accuracy: 0.8882 - val_loss: 0.3101
Epoch 3/5
1688/1688            13s 7ms/step -
accuracy: 0.8976 - loss: 0.2819 - val_accuracy: 0.8890 - val_loss: 0.3041
Epoch 4/5
1688/1688            13s 8ms/step -
accuracy: 0.9079 - loss: 0.2526 - val_accuracy: 0.8978 - val_loss: 0.2800
Epoch 5/5
1688/1688            12s 7ms/step -
accuracy: 0.9167 - loss: 0.2238 - val_accuracy: 0.8975 - val_loss: 0.2802
```

[11]:
```
<keras.src.callbacks.history.History at 0x25f324b7050>
```

[12]:
```
# Evaluate
loss, acc = model.evaluate(x_test, y_test)
print(f"\nTest Accuracy: {acc}")
```

```
313/313            1s 4ms/step -
accuracy: 0.9019 - loss: 0.2680

Test Accuracy: 0.9064000248908997
```

[13]:
```
import matplotlib.pyplot as plt
class_names=['T-shirt/
 ↪top','Trouser','Pullover','Dress','Coat','Sandal','Shirt','Sneaker','Bag','Ankle␣
 ↪boot']
plt.figure(figsize=(10,10))
for i in range(25):
    plt.subplot(5,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(x_train[i],cmap=plt.cm.binary)
    plt.xlabel(class_names[y_train[i]])
plt.show()
```

Pullover
Ankle boot
Shirt
T-shirt/top
Dress
Coat
Coat
Sandal
Coat
Bag
T-shirt/top
Bag
Ankle boot
T-shirt/top
Pullover
Pullover
Ankle boot
Dress
Dress
Dress
Bag
Sneaker
Coat
Coat
T-shirt/top

```
[ ]:
```

```
//Design and implement Parallel Breadth First Search and Depth First Search
based on existing algorithms using OpenMP. Use a Tree or an undirected graph for
BFS and DFS .

#include <iostream>
#include <vector>
#include <queue>
#include <stack>
#include <omp.h>
using namespace std;

const int N = 6;  // Number of nodes
vector<int> graph[N];
bool visited_bfs[N], visited_dfs[N];

// Add edge to undirected graph
void addEdge(int u, int v) {
    graph[u].push_back(v);
    graph[v].push_back(u);
}

// Parallel BFS using OpenMP
void parallelBFS(int start) {
    queue<int> q;
    q.push(start);
    visited_bfs[start] = true;

    while (!q.empty()) {
        int size = q.size();

        #pragma omp parallel for
        for (int i = 0; i < size; i++) {
            int node;
            #pragma omp critical
            {
                node = q.front(); q.pop();
                cout << "BFS visited: " << node << endl;
            }

            for (int neighbor : graph[node]) {
                #pragma omp critical
                {
                    if (!visited_bfs[neighbor]) {
                        visited_bfs[neighbor] = true;
                        q.push(neighbor);
                    }
                }
            }
        }
    }
}

// Parallel DFS using OpenMP
void parallelDFS(int start) {
```

```cpp
    stack<int> s;
    s.push(start);
    visited_dfs[start] = true;

    while (!s.empty()) {
        int node;
        #pragma omp critical
        {
            node = s.top(); s.pop();
            cout << "DFS visited: " << node << endl;
        }

        #pragma omp parallel for
        for (int i = 0; i < graph[node].size(); i++) {
            int neighbor = graph[node][i];
            #pragma omp critical
            {
                if (!visited_dfs[neighbor]) {
                    visited_dfs[neighbor] = true;
                    s.push(neighbor);
                }
            }
        }
    }
}

int main() {
    addEdge(0, 1);
    addEdge(0, 2);
    addEdge(1, 3);
    addEdge(1, 4);
    addEdge(2, 5);

    cout << "Parallel BFS:\n";
    parallelBFS(0);

    cout << "\nParallel DFS:\n";
    parallelDFS(0);

    return 0;
}

//run= g++ -fopenmp HPC_Practical_1.cpp -o HPC_Practical_1
//Windows:- HPC_Practical_1.exe
//Linux:- ./HPC_Practical_1
```

//Write a program to implement Parallel Bubble Sort and Merge sort using OpenMP. Use existing algorithms and measure the performance of sequential and parallel algorithms.

```
#include <iostream>
#include <vector>
#include <omp.h>
using namespace std;

// Sequential Bubble Sort
void bubbleSortSeq(vector<int>& arr) {
    int n = arr.size();
    for (int i = 0; i < n-1; i++)
        for (int j = 0; j < n-i-1; j++)
            if (arr[j] > arr[j+1])
                swap(arr[j], arr[j+1]);
}

// Parallel Bubble Sort
void bubbleSortPar(vector<int>& arr) {
    int n = arr.size();
    for (int i = 0; i < n; i++) {
        #pragma omp parallel for
        for (int j = i % 2; j < n - 1; j += 2)
            if (arr[j] > arr[j + 1])
                swap(arr[j], arr[j + 1]);
    }
}

// Merge function
void merge(vector<int>& arr, int l, int m, int r) {
    vector<int> left(arr.begin() + l, arr.begin() + m + 1);
    vector<int> right(arr.begin() + m + 1, arr.begin() + r + 1);
    int i = 0, j = 0, k = l;
    while (i < left.size() && j < right.size())
        arr[k++] = (left[i] < right[j]) ? left[i++] : right[j++];
    while (i < left.size()) arr[k++] = left[i++];
    while (j < right.size()) arr[k++] = right[j++];
}

// Sequential Merge Sort
void mergeSortSeq(vector<int>& arr, int l, int r) {
    if (l < r) {
        int m = (l + r) / 2;
        mergeSortSeq(arr, l, m);
        mergeSortSeq(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}

// Parallel Merge Sort
void mergeSortPar(vector<int>& arr, int l, int r) {
    if (l < r) {
        int m = (l + r) / 2;
        #pragma omp parallel sections
        {
            #pragma omp section
            mergeSortPar(arr, l, m);
            #pragma omp section
            mergeSortPar(arr, m + 1, r);
        }
        merge(arr, l, m, r);
    }
}

int main() {
    vector<int> data = {8, 5, 2, 9, 1, 4};
    vector<int> arr1 = data, arr2 = data;
    vector<int> arr3 = data, arr4 = data;

    bubbleSortSeq(arr1);
    bubbleSortPar(arr2);
    mergeSortSeq(arr3, 0, arr3.size() - 1);
    mergeSortPar(arr4, 0, arr4.size() - 1);

    cout << "Sorted (Seq Bubble): ";
    for (int x : arr1) cout << x << " ";
    cout << "\nSorted (Par Bubble): ";
    for (int x : arr2) cout << x << " ";
    cout << "\nSorted (Seq Merge):  ";
    for (int x : arr3) cout << x << " ";
    cout << "\nSorted (Par Merge):  ";
    for (int x : arr4) cout << x << " ";
}


//Complie Windows:- g++ -fopenmp HPC_Practical_2.cpp -o HPC_Practical_2
//Run:- HPC_Practical_2.exe

//Complie linux:- sudo apt update
//sudo apt install g++ libomp-dev
//g++ -fopenmp HPC_Practical_2.cpp -o HPC_Practical_2
//./HPC_Practical_2
```

bqcofxpye

April 18, 2025

0.1 Implement Min, Max, Sum and Average oprations using Parallel Reduction.

[1]:
```
import multiprocessing
import random

def parallel_reduction(operation, arr):
    with multiprocessing.Pool() as pool:
        if operation == "min":
            return min(pool.map(min, arr))
        elif operation == "max":
            return max(pool.map(max, arr))
        elif operation == "sum":
            return sum(pool.map(sum, arr))
        elif operation == "avg":
            return sum(pool.map(sum, arr)) / len(arr)

if __name__ == "__main__":
    arr = [random.randint(0, 10000) for _ in range(10000)]
    chunked_arr = [arr[i::multiprocessing.cpu_count()] for i in␣
 ↪range(multiprocessing.cpu_count())]

    print(f"Min: {parallel_reduction('min', chunked_arr)}")
    print(f"Max: {parallel_reduction('max', chunked_arr)}")
    print(f"Sum: {parallel_reduction('sum', chunked_arr)}")
    print(f"Average: {parallel_reduction('avg', chunked_arr)}")
```

```
Min: 0
Max: 10000
Sum: 49891056
Average: 6236382.0
```

[ ]:

Write a cuda program for

1. Addition of two large vectors
2. Matrics Multiplication using CUDA C

```
import cupy as cp
import numpy as np

# Initialize two large vectors
n = 1 << 20  # 1 million elements
a = cp.ones(n, dtype=cp.float32)
b = cp.ones(n, dtype=cp.float32) * 2

# Vector Addition (using CuPy)
c = a + b

# Sample Output
print("c[0] =", c[0])  # This should print 3.0 (1 + 2)

import cupy as cp
import numpy as np

# Define matrix size
N = 512  # Matrix dimensions (N x N)

# Initialize two large matrices
a = cp.ones((N, N), dtype=cp.float32)
b = cp.ones((N, N), dtype=cp.float32)

# Matrix Multiplication (using CuPy)
c = cp.matmul(a, b)

# Sample Output
print("c[0, 0] =", c[0, 0])  # This should print N (512) as the result
of 1 * N
```