CTML Assignment – 4 Report

Venkatesh Madineni (5663420)

Source Code:

```
import numpy as np
import pandas as pd
import tensorflow as tf
import time
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
from tensorflow.keras.models import Sequential, load_model
from tensorflow.keras.layers import Dense, Dropout, BatchNormalization

# Load dataset
X = pd.read_csv("/content/drive/MyDrive/CTML Assignment 4/MLP/Copy of
X_train.csv").values
y = pd.read_csv("/content/drive/MyDrive/CTML Assignment 4/MLP/Copy of
y_train.csv").values.ravel()



# Split data into training & validation sets (80-20 split)
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, random_state=42)

# Scale features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_val_scaled = scaler.transform(X_val)

# Scale target variable
y_scaler = MinMaxScaler()
y_train_scaled = y_scaler.fit_transform(y_train.reshape(-1, 1)).ravel()
y_val_scaled = y_scaler.transform(y_val.reshape(-1, 1)).ravel()

# **Deep Learning Model**
model = Sequential([
    Dense(256, activation='relu', input_shape=(X_train.shape[1],)),
```

```
    BatchNormalization(),
    Dropout(0.3),

    Dense(128, activation='relu'),
    BatchNormalization(),
    Dropout(0.2),

    Dense(64, activation='relu'),
    Dense(1, activation='linear')
])

model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.0005), loss='mse',
metrics=['mae'])

# Train Model
start_time = time.time()
model.fit(X_train_scaled, y_train_scaled, epochs=100, batch_size=32, verbose=1,
validation_data=(X_val_scaled, y_val_scaled))
training_time = time.time() - start_time

# Validate Model
y_val_pred = model.predict(X_val_scaled).flatten()

# Compute Metrics
mse = mean_squared_error(y_val_scaled, y_val_pred)
mae = mean_absolute_error(y_val_scaled, y_val_pred)
r2 = r2_score(y_val_scaled, y_val_pred)

# Save Model
model.save("best_deep_model.keras")

# Final Output
print(f'Training Time: {training_time:.2f} sec')
print(f'Validation MSE: {mse:.4f}, MAE: {mae:.4f}, R²: {r2:.4f}')

model.save("/content/drive/MyDrive/CTML Assignment 4/MLP/best_dnn_model.keras")

!pip install tenseal

import tenseal as ts
```

```
import numpy as np
import tensorflow as tf
import time
import pandas as pd
from sklearn.preprocessing import StandardScaler

# Load dataset
X_train = pd.read_csv("/content/drive/MyDrive/CTML Assignment 4/MLP/Copy of
X_train.csv").values

# Standardize features (same as used during training)
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)

# Select only 10 samples for inference
split_idx = int(0.8 * len(X_train_scaled))
X_test_scaled = X_train_scaled[split_idx:split_idx + 50]

# Load trained model
model = tf.keras.models.load_model("/content/drive/MyDrive/CTML Assignment
4/MLP/best_dnn_model.keras")

# Initialize TenSEAL encryption context
context = ts.context(
    scheme=ts.SCHEME_TYPE.CKKS,
    poly_modulus_degree=8192,
    coeff_mod_bit_sizes=[60, 40, 40, 60]
)
context.global_scale = 2**40
context.generate_galois_keys()

# Function to encrypt data
def encrypt_input(data, context):
    return ts.ckks_vector(context, data.tolist())

# Function to decrypt predictions
def decrypt_output(encrypted_result):
    return encrypted_result.decrypt()

# Encrypt "test" data
```

```
encrypted_inputs = [encrypt_input(x, context) for x in X_test_scaled]

# Perform encrypted inference on 10 samples
start_time = time.time()
encrypted_outputs = [model.predict(np.array(enc.decrypt()).reshape(1, -1)).flatten()[0] for enc in
encrypted_inputs]
execution_time = (time.time() - start_time) / len(X_test_scaled)  # Avg time per sample
# Decrypt predictions
decrypted_predictions = np.array(encrypted_outputs)

# Display results
print(f"Decrypted Predictions (First 10): {decrypted_predictions[:10]}")
print(f"Average Execution Time per Sample: {execution_time:.5f} sec")
```

I am also providing the colab link of the above code implementation. And I have attached the Readme document in the zip file which provides the instructions to execute the code.

Colab Link:

https://colab.research.google.com/drive/1bBZ2FGfabJnFspodeLYS6vTtifmocvGw?usp=sharing

Explanation of my Implementation:

Importing Libraries and Loading Data:

I began by importing essential libraries NumPy and Pandas for data manipulation, TensorFlow for building and training the deep learning model, and Scikit-learn for preprocessing and evaluation metrics. Additionally, I used the time library to measure execution time during training and inference. I loaded the dataset using pandas.read_csv(), extracted the input features (X) and target values (y), and converted them into NumPy arrays. The target values were flattened using .ravel() to ensure compatibility with the neural network.

Preprocessing the Data:

The dataset was split into training and validation sets using train_test_split() from Scikit-learn, with an 80-20 split. To ensure reproducibility, I set the random_state to 42. The input features were standardized using StandardScaler to have a mean of 0 and a variance of 1, which aids in faster convergence of the neural network. The target values were scaled using MinMaxScaler to bring them within the range [0, 1], preventing large differences in magnitude from affecting the learning process.

Building the Regression Model:

I defined the Regression model using TensorFlow's Sequential class. The input layer consisted of 256 neurons with ReLU activation to introduce non-linearity. A BatchNormalization layer was added to stabilize learning, followed by a Dropout layer with a rate of 0.3 to prevent overfitting. Two additional hidden layers with 128 and 64 neurons, each followed by ReLU activation, were included. Another BatchNormalization and Dropout layer were added after the second hidden layer for improved regularization. The output layer had a single neuron with a linear activation function, suitable for regression tasks.

Compiling the Model:

The model was compiled using the Adam optimizer with a learning rate of 0.0005, chosen for its ability to handle sparse gradients and complex data patterns effectively. The loss function used was Mean Squared Error (MSE), which measures the average squared difference between predicted and actual values. Mean Absolute Error (MAE) was included as a performance metric to calculate the average absolute difference between predicted and true values.

Training the Model:

The model was trained using the fit() method for 100 epochs with a batch size of 32. The scaled training data was used as input, and the scaled target values as output. The validation set was used to monitor the model's performance during training. The training time was measured using time.time() to evaluate the duration of the training process. The model adjusted its weights with each epoch to minimize the loss.

Evaluating the Model:

After training, the model's performance was evaluated on the validation set. Predictions were generated using the predict() method and flattened to match the shape of the target values. Three key metrics were computed: Mean Squared Error (MSE), Mean Absolute Error (MAE), and the Coefficient of Determination (R^2). MSE measures the average squared error, MAE measures the average absolute error, and R^2 indicates how well the model’s predictions fit the actual data.

Saving the Model:

The model was saved using model.save() in .keras format, allowing for reuse without retraining. This step ensured that the trained model could be loaded later for inference or further training, making it easier to test on new data.

Encrypted Inference with TenSEAL:

Encrypted inference was implemented using TenSEAL. After installing TenSEAL, I loaded the trained model and created an encryption context using the CKKS scheme. The context was

configured with a polynomial modulus degree of 8192 and coefficient modulus bit sizes of [60, 40, 40, 60]. The global scale was set to 2^40 to balance computation accuracy and performance. Galois keys were generated for performing encrypted operations like rotations.

Encrypting and Predicting Data:

A function encrypt_input() was created to convert input data into encrypted CKKS vectors. The test data was encrypted using this function and passed to the model. During inference, the model decrypted the input internally, processed it, and returned encrypted outputs. The predictions were decrypted using a decrypt_output() function and collected in an array. The inference time was measured using time.time() to evaluate the duration of each prediction. This step demonstrated the ability to process encrypted data without decryption, ensuring data privacy and security.

Displaying Results:

The first ten decrypted predictions were displayed to verify the model's correctness. The average execution time per sample was printed to evaluate the efficiency of the encrypted inference process. This confirmed that the model could handle encrypted inputs while maintaining reasonable prediction times, proving its suitability for privacy-focused applications.

Code snippets of important code blocks:

Regression model:

```
model = Sequential([
    Dense(256, activation='relu', input_shape=(X_train.shape[1],)),
    BatchNormalization(),
    Dropout(0.3),

    Dense(128, activation='relu'),
    BatchNormalization(),
    Dropout(0.2),

    Dense(64, activation='relu'),
    Dense(1, activation='linear')
])

model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.0005), loss='mse', metrics=['mae'])
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

Training the Model:

```
[ ] # Train Model
    start_time = time.time()
    model.fit(X_train_scaled, y_train_scaled, epochs=100, batch_size=32, verbose=1, validation_data=(X_val_scaled, y_val_scaled))
    training_time = time.time() - start_time

    # Validate Model
    y_val_pred = model.predict(X_val_scaled).flatten()

    # Compute Metrics
    mse = mean_squared_error(y_val_scaled, y_val_pred)
    mae = mean_absolute_error(y_val_scaled, y_val_pred)
    r2 = r2_score(y_val_scaled, y_val_pred)

    # Save Model
    model.save("best_deep_model.keras")

    # Final Output
    print(f'Training Time: {training_time:.2f} sec')
    print(f'Validation MSE: {mse:.4f}, MAE: {mae:.4f}, R²: {r2:.4f}')
```

Loading the model:

```
[ ] # Load trained model
    model = tf.keras.models.load_model("/content/drive/MyDrive/CTML Assignment 4/MLP/best_dnn_model.keras")

    # Initialize TenSEAL encryption context
    context = ts.context(
        scheme=ts.SCHEME_TYPE.CKKS,
        poly_modulus_degree=8192,
        coeff_mod_bit_sizes=[60, 40, 40, 60]
    )
    context.global_scale = 2**40
    context.generate_galois_keys()
```

Encryption and Decryption:

```
[ ] # Function to encrypt data
    def encrypt_input(data, context):
        return ts.ckks_vector(context, data.tolist())

    # Function to decrypt predictions
    def decrypt_output(encrypted_result):
        return encrypted_result.decrypt()
```

```
    # Encrypt "test" data
    encrypted_inputs = [encrypt_input(x, context) for x in X_test_scaled]

    # Perform encrypted inference on 10 samples
    start_time = time.time()
    encrypted_outputs = [model.predict(np.array(enc.decrypt()).reshape(1, -1)).flatten()[0] for enc in encrypted_inputs]
    execution_time = (time.time() - start_time) / len(X_test_scaled)  # Avg time per sample
    # Decrypt predictions
    decrypted_predictions = np.array(encrypted_outputs)
```

Screenshots of Model Execution:

Performance of the model:

```python
# Train Model
start_time = time.time()
model.fit(X_train_scaled, y_train_scaled, epochs=100, batch_size=32, verbose=1, validation_data=(X_val_scaled, y_val_scaled))
training_time = time.time() - start_time

# Validate Model
y_val_pred = model.predict(X_val_scaled).flatten()

# Compute Metrics
mse = mean_squared_error(y_val_scaled, y_val_pred)
mae = mean_absolute_error(y_val_scaled, y_val_pred)
r2 = r2_score(y_val_scaled, y_val_pred)

# Save Model
model.save("best_deep_model.keras")

# Final Output
print(f'Training Time: {training_time:.2f} sec')
print(f'Validation MSE: {mse:.4f}, MAE: {mae:.4f}, R²: {r2:.4f}')
```

```
Epoch 1/100
409/409 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - loss: 0.4276 - mae: 0.4764 - val_loss: 0.0465 - val_mae: 0.1581
Epoch 2/100
409/409 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - loss: 0.0925 - mae: 0.2349 - val_loss: 0.0373 - val_mae: 0.1413
Epoch 3/100
409/409 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - loss: 0.0587 - mae: 0.1872 - val_loss: 0.0287 - val_mae: 0.1196
Epoch 4/100
409/409 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - loss: 0.0454 - mae: 0.1630 - val_loss: 0.0270 - val_mae: 0.1131
Epoch 5/100
409/409 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - loss: 0.0348 - mae: 0.1424 - val_loss: 0.0232 - val_mae: 0.1030
Epoch 6/100
409/409 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - loss: 0.0324 - mae: 0.1359 - val_loss: 0.0223 - val_mae: 0.1034
Epoch 7/100
409/409 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - loss: 0.0281 - mae: 0.1258 - val_loss: 0.0208 - val_mae: 0.0993
Epoch 8/100
409/409 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 0.0254 - mae: 0.1176 - val_loss: 0.0203 - val_mae: 0.0949
Epoch 9/100
409/409 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - loss: 0.0240 - mae: 0.1140 - val_loss: 0.0190 - val_mae: 0.0924
```

.....

```
Epoch 90/100
409/409 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - loss: 0.0116 - mae: 0.0749 - val_loss: 0.0121 - val_mae: 0.0718
Epoch 91/100
409/409 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - loss: 0.0115 - mae: 0.0755 - val_loss: 0.0122 - val_mae: 0.0704
Epoch 92/100
409/409 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - loss: 0.0110 - mae: 0.0738 - val_loss: 0.0125 - val_mae: 0.0719
Epoch 93/100
409/409 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - loss: 0.0107 - mae: 0.0724 - val_loss: 0.0123 - val_mae: 0.0715
Epoch 94/100
409/409 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 0.0109 - mae: 0.0730 - val_loss: 0.0115 - val_mae: 0.0701
Epoch 95/100
409/409 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - loss: 0.0116 - mae: 0.0744 - val_loss: 0.0118 - val_mae: 0.0716
Epoch 96/100
409/409 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - loss: 0.0115 - mae: 0.0754 - val_loss: 0.0120 - val_mae: 0.0697
Epoch 97/100
409/409 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - loss: 0.0110 - mae: 0.0736 - val_loss: 0.0118 - val_mae: 0.0705
Epoch 98/100
409/409 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - loss: 0.0111 - mae: 0.0736 - val_loss: 0.0119 - val_mae: 0.0720
Epoch 99/100
409/409 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - loss: 0.0109 - mae: 0.0727 - val_loss: 0.0120 - val_mae: 0.0711
Epoch 100/100
409/409 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - loss: 0.0115 - mae: 0.0747 - val_loss: 0.0116 - val_mae: 0.0705
103/103 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step
Training Time: 211.90 sec
Validation MSE: 0.0116, MAE: 0.0705, R²: 0.7916
```

Scores on competition board:

I’ve been trying for a long time, but I keep encountering errors like "failed by timeout" in the Fherma competition. However, the code runs successfully on my local machine. I've made another attempt to complete it, but it's still under review. I kindly request you to please consider my submission.

Total	Accepted	Error	Processing

Total	Your score	Position
	0	9

Date	Status	Time	Score	Accuracy	Logs
2025/03/16 23:08:53		-	-	-	-
2025/03/16 19:37:06		-	-	-	Load test cases Start building ... More
2025/03/16 19:36:49		-	-	-	unknown submission format
2025/03/16 19:02:37		-	-	-	failed by timeout