The Implementation of SGD for NN using MPI

Part 1: Data Preprocessing (Python)

Packages

Before interpreting the implementation of SGD with MPI, the discussion of data preview and data washing is necessary, as it's closely related to the structure of data sent to the main programme, and determines the split number of the dataset, which is closely related to the processes. Here's the packages we used in our Python part.

In [ ]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import time
from datetime import datetime
import os
import joblib
import json
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
```

Among the packages, numpy and pandas are used to deal with the dataframes read from original csv. matplotlib.pyplot and seaborn are used for data visualization, which is related to data washing. time is used to record the time that the process uses, and datetime is used to regularize datetime data in the dataframe. os, joblib and json are used to deal with data saving and loading. LabelEncoder is used to deal with label encoding problem. And train_test_split is used to split train data and test data.

Data Washing

Date Time Data

In [ ]:
```
df["trip_duration"] = (df["tpep_dropoff_datetime"] - df["tpep_pickup_datetime"])

before = len(df_clean)
df_clean = df_clean[(df_clean['trip_duration'] > 30) & (df_clean['trip_duration'
removed = before - len(df_clean)
print(f"Removed 'trip_duration' outliers: {removed} records ({removed/before:.2%
```

Firstly, we removed rows where 'trip_duration', the difference between 'tpep_dropoff_datetime' and 'tpep_pickup_datetime', didn't satisfy normal condition. In practice, we deleted rows with 'trip_duration' smaller than 30

seconds or greater than 24*60*60 seconds (one day), where click farming records and abnormal records were excluded.

Numerical Data

In [ ]:
```
df_clean['passenger_count'] = pd.to_numeric(df_clean['passenger_count'], errors=
df_clean['RatecodeID'] = pd.to_numeric(df_clean['RatecodeID'], errors='coerce')
df_clean['payment_type'] = pd.to_numeric(df_clean['payment_type'], errors='coerc
before = len(df_clean)
df_clean.loc[~df_clean['RatecodeID'].isin([1,2,3,4,5,99]), 'RatecodeID'] = np.na
df_clean.loc[~df_clean['payment_type'].isin([1,2,3,4]), 'payment_type'] = np.nan
df_clean.loc[(df_clean['passenger_count'] == 0) | (df_clean['passenger_count'] >
df_clean = df_clean.dropna()
removed = before - len(df_clean)
print(f"Removed categorical outliers: {removed} records ({removed/before:.2%})")
```

Secondly, we removed rows where 'passenger_count', 'RatecodeID', 'payment_type' don't satisfy normal conditions. For the 'passenger_count', we removed rows with passenger counts greater than 6 or equal to 0. For the 'RatecodeID', we removed rows with ratings out of range. For the 'payment_type', we removed rows with abnormal payment types.

In [ ]:
```
numeric_filters = {
    'trip_distance': (0.1, 100),
    'extra': (0, 10),
    'total_amount': (2, 1000),
}

for col, (min_val, max_val) in numeric_filters.items():
    before = len(df_clean)
    df_clean = df_clean[(df_clean[col] >= min_val) & (df_clean[col] <= max_val)]
    removed = before - len(df_clean)
    print(f"Removed {col} outliers: {removed} records ({removed/before:.2%})")
```

Besides, we removed rows with abnormal numbers accordingly. For the 'trip_distance', we removed rows with distance smaller than 0.1 mile or greater than 100 miles. For the 'extra', we removed rows with extra fees greater than 10 dollars. For the 'total_amount', we removed rows with total amount smaller than 2 dollars or greater than 1000 dollars.

Data Split

In [ ]:
```
train_df, test_df = train_test_split(df_clean, test_size=0.3, random_state=rando

# Process train and test data
def process_data(df):
    """Process the dataset (training or testing)"""
    # Standardize numeric features - using statistics from the training set
    for col in numeric_cols + ["trip_duration", target_col]:
        if col in df.columns:
            mean, std = numeric_global_stats[col]['mean'], numeric_global_stats[
            df[col] = (df[col] - mean) / (std if std > 0 else 1)

    # Perform embedding lookup for high cardinality columns
```

```
    embedding_features = []
    for col in high_cardinality_id_cols:
        if col in df.columns:
            # Handle unseen categories - map to 0
            encoded = np.zeros(len(df), dtype=int)
            known_categories = set(label_encoders[col].classes_)
            mask = df[col].isin(known_categories)
            encoded[mask] = label_encoders[col].transform(df.loc[mask, col])
            # Lookup embedding
            embeddings = embedding_matrices[col][encoded]
            embedding_features.append(embeddings)

    # One-hot encode low cardinality categorical columns
    one_hot_features = []
    for col in low_cardinality_cat_cols:
        if col in df.columns:
            # Create one-hot encoding
            one_hot = np.zeros((len(df), len(one_hot_mappings[col])))
            for j, category in enumerate(one_hot_mappings[col]):
                one_hot[:, j] = (df[col] == category).astype(float)
            one_hot_features.append(one_hot)

    # Combine all features
    numeric_features = df[numeric_cols].values
    time_feature = df['trip_duration'].values.reshape(-1, 1)

    all_features_list = [numeric_features, time_feature]
    if embedding_features:
        all_features_list.extend(embedding_features)
    if one_hot_features:
        all_features_list.extend(one_hot_features)

    all_features = np.hstack(all_features_list)
    labels = df[target_col].values.reshape(-1, 1)

    return all_features, labels

# Process training and testing sets
X_train, y_train = process_data(train_df)
X_test, y_test = process_data(test_df)
```

Firstly, we split data into training data and testing data, and then, extracted the labels from features. During the process, we performed embedding for high cardinality columns and used one-hot encoding for low cardinality categorical columns.

In [ ]:
```
# Calculate how many rows each partition should have (discard remainder)
train_rows_per_part = len(train_data) // nparts
train_data = train_data[:train_rows_per_part * nparts]

for j in range(nparts):
    start_idx = j * train_rows_per_part
    end_idx = (j + 1) * train_rows_per_part

    # Assign data to partitions
    partition_data = train_data[start_idx:end_idx]
    np.savetxt(train_files[j], partition_data, delimiter=',', fmt='%.6f')
```

```
# Distribute test data into different partitions
test_data = np.hstack([X_test, y_test])
np.random.shuffle(test_data)

test_rows_per_part = len(test_data) // nparts
test_data = test_data[:test_rows_per_part * nparts]

for j in range(nparts):
    start_idx = j * test_rows_per_part
    end_idx = (j + 1) * test_rows_per_part

    # Assign data to partitions
    partition_data = test_data[start_idx:end_idx]
    np.savetxt(test_files[j], partition_data, delimiter=',', fmt='%.6f')

for f in train_files.values():
    f.close()
for f in test_files.values():
    f.close()
```

Secondly, we split training data and testing data into different partitions, which is related to the number of processors. For convenience, we discarded the remaining part of the data.

Part 2: SGD for NN using MPI (C)

Hyperparameters

In [ ]:
```
#define M_FEATURES 35
#define N_HIDDEN 64
#define EPOCHS 20
#define LR 0.01
#define BATCH_SIZE 512
```

In our implementation, we defined 5 hyperparameters for the whole process. 'M_features' stands for the total amount of the features, where datetime features, 'tpep_dropoff_datetime' and 'tpep_pickup_datetime', are combined into 'trip_duration', nominal features are embedded or one-hot encoded and numerical features are regularized, thus having 35 features altogether, a much bigger digit compared with 9 original columns. 'N_HIDDEN' stands for the number of the hidden layer, whose default input is 64, yet in practice, we found 8 hidden layer is good enough, so there's a great difference between results and origin setting. 'EPOCHS' stands for the number of training epoches, where 20 is a good default digit for programmes of such scale. 'LR' stands for the learning rate, which determines the learning speed of the programme and final results. Similarly, 0.01 is a usually good default. 'BATCH_SIZE' stands for the size of batch, literally, determines the training speed and the final results.

SGD for NN

In [ ]:
```
if (rank == 0) {
    for (int j = 0; j < n; j++) {
        for (int k = 0; k < m; k++) {
            W[j][k] = ((double)rand() / RAND_MAX - 0.5) * 0.1;
        }
        b[j] = 0.0;
        v[j] = ((double)rand() / RAND_MAX - 0.5) * 0.1;
    }
    *c = 0.0;
}
```

In our implementation, the initialization was performed only by the root process (where rank equals to 0), to ensure that all processes shared the same start parameters. Weight values in the weight matrix for the hidden layer 'W' and in the weight vector for the ouput 'v' was randomly initialized using a uniform distribution within the range from -0.05 to 0.05, where small initial weights avoided problems like explosion of gradients in the early stages of training. Biases for the hidden layer 'b' and for the output 'c' were initialized to 0.0, meaning initial biases didn't influence the first learning process, which is a common practice.

In [ ]:
```
for (int j = 0; j < n; j++) {
    MPI_Bcast(W[j], m, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(&b[j], 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(&v[j], 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
}
MPI_Bcast(c, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

Then, the initialized parameters were broadcast to all processors. After each epoch, every parameters were updated and brodacast to all processors again.

Training

In [ ]:
```
shuffle_local_data(X, Y, N, m, rank, epoch);abs

// definition
void shuffle_local_data(double **X, double *Y, int N, int m, int rank, int epoch
    srand(123 + rank + epoch * 1000);
    for (int i = N-1; i > 0; i--) {
        int j = rand() % (i+1);
        double *temp = X[i];
        X[i] = X[j];
        X[j] = temp;

        double tempY = Y[i];
        Y[i] = Y[j];
        Y[j] = tempY;
    }
}
```

In our implementation of training part, firstly, we shuffled data in each epoch to avoid mislearning on the sequence of th data, leading to a more generalizable

and stable model. We set different random seed based on rank and epoches to avoid same shuffling in each step.

In [ ]:
```
double local_loss = 0.0;
for (int i = 0; i < N; i++) {
    double *x = X[i];
    double y = Y[i];

    for (int j = 0; j < n; j++) {
        double s = b[j];
        for (int k = 0; k < m; k++) s += W[j][k] * x[k];

        if (activation_type == ACTIVATION_SIGMOID) {
            hidden[j] = sigmoid(s);
        } else if (activation_type == ACTIVATION_RELU) {
            hidden[j] = relu(s);
        } else if (activation_type == ACTIVATION_TANH) {
            hidden[j] = tanh_activation(s);
        }
    }
    double out = *c;
    for (int j = 0; j < n; j++) out += v[j] * hidden[j];

    double err = out - y;
    local_loss += 0.5 * err * err;
}

double global_loss;
MPI_Allreduce(&local_loss, &global_loss, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD)
double avg_loss = global_loss / (N * size);
```

Secondly, we count local loss and aggregated the local losses of all processors to compute average loss to evaluate training effects of each epoch and see whether exists situations need early stopping.

In [ ]:
```
if (rank == 0) {
    if (fabs(prev_loss - avg_loss) < early_stop_delta) {
        stop_counter++;
        if (stop_counter >= early_stop_patience) {
            printf("Early stopping at epoch %d\n", epoch+1);
            int stop_signal = 1;
            MPI_Bcast(&stop_signal, 1, MPI_INT, 0, MPI_COMM_WORLD);
            break;
        }
    } else {
        stop_counter = 0;
    }
    prev_loss = avg_loss;

    int stop_signal = 0;
    MPI_Bcast(&stop_signal, 1, MPI_INT, 0, MPI_COMM_WORLD);
} else {
    int stop_signal;
    MPI_Bcast(&stop_signal, 1, MPI_INT, 0, MPI_COMM_WORLD);
    if (stop_signal) break;
}
```

Besides, before entering the iterations, we also included an early-stopping part. Firstly, we distinguished the early sopping situation at rank 0 from other situations, then we broadcast the stop signal to all processors.

In [ ]:
```
for (int iter = 0; iter < local_batches; iter++) {
    // Initialize accumulated gradients

    // Local gradients computation
        // Forward

        // Back propagation

    // Aggregated gradients of all processors

    // Update parameters

}
```

Here's the stucture of the iterations, the implementation of each part will be discussed in the following parts.

In [ ]:
```
// Initialize accumulated gradients
for (int j = 0; j < n; j++) {
    accum_grad_v[j] = 0.0;
    accum_grad_b[j] = 0.0;
    for (int k = 0; k < m; k++) {
        accum_grad_W[j][k] = 0.0;
    }
}
accum_grad_c = 0.0;
```

For the part of initializing accumulated gradients, we set every gradient of every parameters equal to 0.

In [ ]:
```
// Forward
for (int j = 0; j < n; j++) {
    double s = b[j];
    for (int k = 0; k < m; k++) s += W[j][k] * x[k];
    z[j] = s;

    if (activation_type == ACTIVATION_SIGMOID) {
        hidden[j] = sigmoid(s);
    } else if (activation_type == ACTIVATION_RELU) {
        hidden[j] = relu(s);
    } else if (activation_type == ACTIVATION_TANH) {
        hidden[j] = tanh_activation(s);
    }
}
double out = *c;
for (int j = 0; j < n; j++) out += v[j] * hidden[j];

double err = out - y;
```

For the forward part, we computed the weighted sum of the inputs, added biases to them, and then applied activation function to them, where activation functions

are determined by the hyperparameters in the beginning. Finally, we get the error for the back propagation.

In [ ]:
```
// Back propagation
for (int j = 0; j < n; j++) {
    double derivative;
    if (activation_type == ACTIVATION_SIGMOID) {
        derivative = sigmoid_derivative(hidden[j]);
    } else if (activation_type == ACTIVATION_RELU) {
        derivative = relu_derivative(hidden[j]);
    } else if (activation_type == ACTIVATION_TANH) {
        derivative = tanh_derivative(hidden[j]);
    }

    grad_v[j] = err * hidden[j];
    grad_b[j] = err * v[j] * derivative;

    for (int k = 0; k < m; k++) {
        accum_grad_W[j][k] += grad_b[j] * x[k];
    }
    accum_grad_v[j] += grad_v[j];
    accum_grad_b[j] += grad_b[j];
}
accum_grad_c += err;
```

For the back-propagation part, we computed the derivatives of each part of each processor for updating parameters later.

In [ ]:
```
// Aggregated gradients of all processors
for (int j = 0; j < n; j++) {
    MPI_Allreduce(MPI_IN_PLACE, accum_grad_W[j], m, MPI_DOUBLE, MPI_SUM, MPI_COM
    MPI_Allreduce(MPI_IN_PLACE, &accum_grad_v[j], 1, MPI_DOUBLE, MPI_SUM, MPI_CO
    MPI_Allreduce(MPI_IN_PLACE, &accum_grad_b[j], 1, MPI_DOUBLE, MPI_SUM, MPI_CO
}
MPI_Allreduce(MPI_IN_PLACE, &accum_grad_c, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORL
```

For the part of aggregating gradients, we get the sum of all derivatives correspondingly for updating parameters.

In [ ]:
```
// Update parameters
double scale = lr / (batch_size * size);
for (int j = 0; j < n; j++) {
    for (int k = 0; k < m; k++) {
        W[j][k] -= scale * accum_grad_W[j][k];
    }
    b[j] -= scale * accum_grad_b[j];
    v[j] -= scale * accum_grad_v[j];
}
*c -= scale * accum_grad_c;
```

For the part of updating parameters, we used the gradients to adjust the parameters in the direction that reduces the error together with the hyperparameter 'learning rate'.

Output

In [ ]:
```
double compute_sse(double **X, double *Y, int N, int m, int n,
    double **W, double *b, double *v, double c, int activation_type,
    int rank, const char* dataset_name) {
    double sse = 0.0;

    // Debug with the SSE of the first 5 samples
    printf("Rank %d: %s - First 5 samples predictions:\n", rank, dataset_name);
    for (int i = 0; i < 5 && i < N; i++) {
        double y_pred = forward(X[i], m, n, W, b, v, c, activation_type);
        double err = y_pred - Y[i];
        printf("  Rank %d: %s Sample %d: y_true=%.6f, y_pred=%.6f, err=%.6f\n",
               rank, dataset_name, i, Y[i], y_pred, err);
        sse += err * err;
    }

    // Compute the SSE of all samples
    for (int i = 5; i < N; i++) {
        double y_pred = forward(X[i], m, n, W, b, v, c, activation_type);
        double err = y_pred - Y[i];
        sse += err * err;
    }

    printf("Rank %d: %s - Computed SSE for %d samples\n", rank, dataset_name, N)

    return sse;
}
```

In the implemetation of the output, we computed and displayed SSE for debugging and tracking training history.

Main Function

In the implementation of the programme, we used the MPI function and many other functions to construct the main function. In the main function, we firstly set the default parameters and broadcast to all processors. Then, we extracted training and testing data from the corresponding csv files generated by the python file of Part 1. Afterwards, we initialized the model parameters, trained the model and computed the global RMSE for the final output. The results will be further discussed in Part 3.

Part 3: Result Explanations

Epoch	1024-1	1024-2	1024-4	1024-8	1024-16	1024-32
1	0.4999	0.5003	0.5012	0.5061	0.5011	0.5031
2	0.2052	0.1026	0.0796	0.0717	0.0687	0.0642
3	0.0973	0.0737	0.0685	0.0640	0.0629	0.0607
4	0.0812	0.0694	0.0654	0.0614	0.0601	0.0585
5	0.0739	0.0671	0.0633	0.0592	0.0578	0.0565
6	0.0695	0.0655	0.0617	0.0573	0.0558	0.0545

Epoch	1024-1	1024-2	1024-4	1024-8	1024-16	1024-32
7	0.0667	0.0643	0.0604	0.0553	0.0539	0.0526
8	0.0649	0.0635	0.0593	0.0535	0.0521	0.0507
9	0.0637	0.0629	0.0584	0.0517	0.0504	0.0490
10	0.0630	0.0625	0.0575	0.0499	0.0489	0.0473
11	0.0624	0.0621	0.0567	0.0483	0.0474	0.0459
12	0.0620	0.0618	0.0559	0.0468	0.0461	0.0446
13	0.0616	0.0615	0.0552	0.0454	0.0449	0.0435
14	0.0613	0.0612	0.0544	0.0442	0.0439	0.0426
15	0.0610	0.0609	0.0535	0.0431	0.0429	0.0418
16	0.0608	0.0605	0.0527	0.0421	0.0421	0.0412
17	0.0606	0.0602	0.0519	0.0412	0.0414	0.0406
18	0.0605	0.0599	0.0510	0.0405	0.0407	0.0401
19	0.0603	0.0595	0.0502	0.0398	0.0402	0.0397
20	0.0602	0.0592	0.0493	0.0392	0.0397	0.0393
Time	307.33	59.22	69.84	97.77	155.56	267.86
Train RMSE	0.3467	0.3430	0.3114	0.2780	0.2801	0.2792
Test RMSE	0.3483	0.3446	0.3130	0.2796	0.2817	0.2809

Epoch	512-1	512-2	512-4	512-8	512-16	512-32
1	0.4999	0.5003	0.5012	0.5061	0.5011	0.5031
2	0.0973	0.0737	0.0685	0.0640	0.0629	0.0607
3	0.0739	0.0671	0.0633	0.0592	0.0578	0.0565
4	0.0667	0.0643	0.0604	0.0553	0.0539	0.0526
5	0.0637	0.0629	0.0584	0.0517	0.0504	0.0490
6	0.0624	0.0621	0.0567	0.0483	0.0474	0.0459
7	0.0616	0.0615	0.0552	0.0454	0.0449	0.0435
8	0.0610	0.0609	0.0535	0.0431	0.0429	0.0418
9	0.0606	0.0602	0.0519	0.0412	0.0414	0.0406
10	0.0603	0.0595	0.0501	0.0398	0.0402	0.0397
11	0.0601	0.0588	0.0485	0.0386	0.0392	0.0390
12	0.0599	0.0582	0.0469	0.0377	0.0384	0.0383
13	0.0598	0.0576	0.0456	0.0370	0.0377	0.0378
14	0.0597	0.0571	0.0445	0.0365	0.0371	0.0373
15	0.0596	0.0567	0.0437	0.0360	0.0365	0.0369

Epoch	512-1	512-2	512-4	512-8	512-16	512-32
16	0.0595	0.0563	0.0430	0.0356	0.0361	0.0365
17	0.0595	0.0559	0.0423	0.0352	0.0357	0.0362
18	0.0594	0.0556	0.0418	0.0349	0.0354	0.0359
19	0.0594	0.0553	0.0413	0.0346	0.0351	0.0356
20	0.0594	0.0551	0.0409	0.0343	0.0349	0.0354
Time	47.4	54.98	71.07	101.7	157.52	277.85
Train RMSE	0.3445	0.3312	0.2846	0.2611	0.2632	0.2652
Test RMSE	0.3461	0.3328	0.2863	0.2629	0.2650	0.2670

Epoch	256-1	256-2	256-4	256-8	256-16	256-32
1	0.4999	0.5003	0.5012	0.5061	0.5011	0.5031
2	0.0739	0.0671	0.0633	0.0593	0.0578	0.0565
3	0.0637	0.0629	0.0583	0.0517	0.0504	0.0490
4	0.0616	0.0615	0.0552	0.0455	0.0450	0.0435
5	0.0606	0.0602	0.0518	0.0412	0.0415	0.0406
6	0.0601	0.0588	0.0485	0.0386	0.0393	0.0390
7	0.0598	0.0576	0.0456	0.0370	0.0377	0.0378
8	0.0596	0.0567	0.0437	0.0360	0.0365	0.0369
9	0.0595	0.0559	0.0423	0.0352	0.0357	0.0362
10	0.0594	0.0553	0.0413	0.0346	0.0351	0.0356
11	0.0593	0.0548	0.0405	0.0341	0.0346	0.0352
12	0.0593	0.0544	0.0398	0.0336	0.0342	0.0348
13	0.0593	0.0421	0.0391	0.0332	0.0338	0.0345
14	0.0592	0.0400	0.0385	0.0328	0.0335	0.0342
15	0.0592	0.0388	0.0380	0.0325	0.0332	0.0339
16	0.0592	0.0380	0.0375	0.0321	0.0329	0.0337
17	0.0592	0.0373	0.0371	0.0318	0.0327	0.0334
18	0.0591	0.0367	0.0367	0.0315	0.0324	0.0332
19	0.0591	0.0361	0.0363	0.0313	0.0322	0.0330
20	0.0591	0.0357	0.0360	0.0310	0.0320	0.0328
Time	43.78	56.52	75.07	109.48	175.12	308.61
Train RMSE	0.3438	0.2658	0.2673	0.2480	0.2522	0.2552
Test RMSE	0.3453	0.2678	0.2692	0.2500	0.2541	0.2571

Epoch	128-1	128-2	128-4	128-8	128-16	128-32
1	0.4999	0.5003	0.5012	0.5061	0.5011	0.5031
2	0.0637	0.0629	0.0583	0.0517	0.0504	0.0489
3	0.0606	0.0602	0.0518	0.0412	0.0414	0.0406
4	0.0598	0.0576	0.0456	0.0370	0.0377	0.0378
5	0.0595	0.0559	0.0423	0.0353	0.0357	0.0362
6	0.0593	0.0548	0.0405	0.0341	0.0346	0.0352
7	0.0593	0.0420	0.0391	0.0332	0.0339	0.0345
8	0.0592	0.0388	0.0380	0.0325	0.0332	0.0339
9	0.0592	0.0372	0.0371	0.0318	0.0327	0.0334
10	0.0591	0.0361	0.0363	0.0312	0.0322	0.0330
11	0.0591	0.0353	0.0357	0.0308	0.0318	0.0326
12	0.0591	0.0347	0.0352	0.0303	0.0314	0.0322
13	0.0590	0.0342	0.0347	0.0299	0.0310	0.0319
14	0.0590	0.0338	0.0344	0.0296	0.0306	0.0316
15	0.0589	0.0335	0.0340	0.0293	0.0303	0.0313
16	0.0589	0.0332	0.0337	0.0291	0.0300	0.0310
17	0.0589	0.0330	0.0334	0.0289	0.0297	0.0308
18	0.0589	0.0328	0.0331	0.0287	0.0294	0.0305
19	0.0588	0.0326	0.0329	0.0285	0.0292	0.0302
20	0.0588	0.0324	0.0327	0.0284	0.0289	0.0300
Time	46.89	60.46	83.28	125.39	210.83	371.81
Train RMSE	0.3429	0.2540	0.2548	0.2377	0.2397	0.2440
Test RMSE	0.3444	0.2562	0.2569	0.2399	0.2419	0.2461

Epoch	64-1	64-2	64-4	64-8	64-16	64-32
1	0.4999	0.5003	0.5012	0.5061	0.5011	0.5031
2	0.0606	0.0602	0.0518	0.0412	0.0414	0.0406
3	0.0595	0.0560	0.0423	0.0352	0.0357	0.0362
4	0.0593	0.0422	0.0391	0.0332	0.0338	0.0345
5	0.0592	0.0373	0.0371	0.0318	0.0327	0.0334
6	0.0591	0.0353	0.0357	0.0307	0.0317	0.0326
7	0.0590	0.0342	0.0348	0.0299	0.0309	0.0319
8	0.0589	0.0335	0.0340	0.0293	0.0302	0.0313
9	0.0589	0.0330	0.0334	0.0289	0.0295	0.0307

Epoch	64-1	64-2	64-4	64-8	64-16	64-32
10	0.0588	0.0326	0.0329	0.0285	0.0290	0.0302
11	0.0588	0.0323	0.0325	0.0283	0.0287	0.0298
12	0.0587	0.0320	0.0321	0.0280	0.0283	0.0293
13	0.0587	0.0318	0.0318	0.0277	0.0280	0.0288
14	0.0587	0.0316	0.0315	0.0274	0.0277	0.0284
15	0.0587	0.0314	0.0313	0.0272	0.0275	0.0280
16	0.0587	0.0313	0.0310	0.0270	0.0273	0.0276
17	0.0587	0.0312	0.0309	0.0268	0.0269	0.0271
18	0.0587	0.0310	0.0307	0.0267	0.0267	0.0268
19	0.0587	0.0309	0.0305	0.0265	0.0264	0.0265
20	0.0587	0.0308	0.0304	0.0264	0.0262	0.0263
Time	54.05	70.17	97.99	156.33	266.79	495.59
Train RMSE	0.3425	0.2475	0.2460	0.2290	0.2397	0.2440
Test RMSE	0.3440	0.2497	0.2481	0.2314	0.2307	0.2311

The five tables above show the training epoch loss, training time, training and test RMSE of different hyperparameters. The title shows the combination of batch size and hidden layer (e.g., 1024-1 means the batch size is 1024 and the number of the hidden layer is 1). We notice that time increases as batch size decrease (except 1024-1), time increases as number of hidden layer increases. Trian RMSE and Test RMSE reaches minimum when number of hidden layer is around 8. The reason is that for number of hidden layer smaller than 8, they converged earlier than 20 epoches limited to their performance. Number of hidden layer greater than 8, they were somehow underfitted. So, after considertion, we took batch size equal to 512 and number of hidden layer equal to 8 for further discussion.

Epoch	0.1	0.01	0.001(20)	0.001(100)						
Time	109.12	109.33	107.98	514.77						
Train RMSE	0.2210	0.2480	0.3214	0.2611						
Test RMSE	0.2235	0.2500	0.3232	0.2629						
1	0.5061	0.5061	0.5061	0.5061	21	0.0517	41	0.0412	61	0.0370
2	0.0341	0.0593	0.3713	0.3713	22	0.0510	42	0.0409	62	0.0369
3	0.0308	0.0517	0.0790	0.0790	23	0.0503	43	0.0406	63	0.0368
4	0.0291	0.0455	0.0693	0.0693	24	0.0496	44	0.0403	64	0.0367
5	0.0283	0.0412	0.0659	0.0659	25	0.0490	45	0.0400	65	0.0366
6	0.0278	0.0386	0.0640	0.0640	26	0.0483	46	0.0398	66	0.0365

Epoch	0.1	0.01	0.001(20)	0.001(100)						
7	0.0274	0.0370	0.0628	0.0628	27	0.0477	47	0.0395	67	0.0364
8	0.0270	0.0360	0.0618	0.0618	28	0.0471	48	0.0393	68	0.0363
9	0.0264	0.0352	0.0609	0.0609	29	0.0465	49	0.0391	69	0.0362
10	0.0260	0.0346	0.0601	0.0601	30	0.0460	50	0.0388	70	0.0361
11	0.0256	0.0341	0.0592	0.0592	31	0.0454	51	0.0386	71	0.0360
12	0.0253	0.0336	0.0584	0.0584	32	0.0449	52	0.0384	72	0.0359
13	0.0252	0.0332	0.0576	0.0576	33	0.0444	53	0.0383	73	0.0358
14	0.0250	0.0328	0.0569	0.0569	34	0.0440	54	0.0381	74	0.0358
15	0.0249	0.0325	0.0561	0.0561	35	0.0435	55	0.0379	75	0.0357
16	0.0248	0.0321	0.0553	0.0553	36	0.0431	56	0.0377	76	0.0356
17	0.0247	0.0318	0.0546	0.0546	37	0.0427	57	0.0376	77	0.0355
18	0.0246	0.0315	0.0538	0.0538	38	0.0423	58	0.0374	78	0.0355
19	0.0245	0.0313	0.0531	0.0531	39	0.0419	59	0.0373	79	0.0354
20	0.0245	0.0310	0.0524	0.0524	40	0.0416	60	0.0372	80	0.0353

Then, we tested the effect of learning rate. We noticed when learning rate equals to 0.001, it didn't converge after 100 epoches. When learning rate equals to 0.01, it was about to converge when epoche reaches 20. When learning rate equals to 0.1, it converges before 20 epoches. Yet consider that a big learning rate may lead to poor performance due to long step, we chose 0.01 as the final hyperparameter.

Epoch	0	1	2
1	0.5061	0.5048	0.5044
2	0.0593	0.0370	0.0447
3	0.0517	0.0323	0.0381
4	0.0455	0.0309	0.0353
5	0.0412	0.0294	0.0331
6	0.0386	0.0283	0.0316
7	0.0370	0.0277	0.0306
8	0.0360	0.0272	0.0299
9	0.0352	0.0269	0.0293
10	0.0346	0.0267	0.0289
11	0.0341	0.0265	0.0285
12	0.0336	0.0263	0.0282
13	0.0332	0.0262	0.0279

Epoch	0	1	2
14	0.0328	0.0260	0.0277
15	0.0325	0.0259	0.0273
16	0.0321	0.0258	0.0271
17	0.0318	0.0256	0.0269
18	0.0315	0.0255	0.0268
19	0.0313	0.0255	0.0267
20	0.0310	0.0254	0.0265
Time	109.48	93.21	115.41
Train RMSE	0.2480	0.2248	0.2299
Test RMSE	0.2500	0.2274	0.2323

Finally, we tested the effects of different activation function, 0 stands for sigmoid function, 1 stands for ReLu function and 2 stands for tanh function. We noticed that ReLu performed the best in time and RMSE, which may somehow explain why so many deep learning models choose ReLu. However, ReLu does have a problem, that is the parameters cannot be updated when the output of ReLu is 0. So, in further possible improvment, we may consider leaky ReLu as final activation function.

AI Tool Usage

We used ChatGPT 5 to generate the tables above in markdown environment (as it's a torture to type the tables digit by digit, hyphen by hyphen).