Lab-4: House Price Prediction with Gradient Descent Variants

R Abhijit Srivathsan - 2448044

1. Imports

In [1]:
```
import numpy as np
import pandas as pd
import tensorflow as tf
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error
```

```
2025-07-15 10:56:03.006976: I tensorflow/core/util/port.cc:153] oneDNN custom operations are on. You may see slightl
y different numerical results due to floating-point round-off errors from different computation orders. To turn them
off, set the environment variable `TF_ENABLE_ONEDNN_OPTS=0`.
2025-07-15 10:56:03.016147: E external/local_xla/xla/stream_executor/cuda/cuda_fft.cc:467] Unable to register cuFFT
factory: Attempting to register factory for plugin cuFFT when one has already been registered
WARNING: All log messages before absl::InitializeLog() is called are written to STDERR
E0000 00:00:1752557163.027288   76763 cuda_dnn.cc:8579] Unable to register cuDNN factory: Attempting to register fac
tory for plugin cuDNN when one has already been registered
E0000 00:00:1752557163.030301   76763 cuda_blas.cc:1407] Unable to register cuBLAS factory: Attempting to register f
actory for plugin cuBLAS when one has already been registered
W0000 00:00:1752557163.038447   76763 computation_placer.cc:177] computation placer already registered. Please check
linkage and avoid linking the same target more than once.
W0000 00:00:1752557163.038469   76763 computation_placer.cc:177] computation placer already registered. Please check
linkage and avoid linking the same target more than once.
W0000 00:00:1752557163.038471   76763 computation_placer.cc:177] computation placer already registered. Please check
linkage and avoid linking the same target more than once.
W0000 00:00:1752557163.038472   76763 computation_placer.cc:177] computation placer already registered. Please check
linkage and avoid linking the same target more than once.
2025-07-15 10:56:03.041464: I tensorflow/core/platform/cpu_feature_guard.cc:210] This TensorFlow binary is optimized
to use available CPU instructions in performance-critical operations.
To enable the following instructions: AVX2 AVX_VNNI FMA, in other operations, rebuild TensorFlow with the appropriat
e compiler flags.
/home/abhijit/miniconda3/envs/tf-env/lib/python3.12/site-packages/requests/__init__.py:86: RequestsDependencyWarning
: Unable to find acceptable character detection dependency (chardet or charset_normalizer).
  warnings.warn(
```

2. Load and Inspect Data

Make sure `Bengaluru_House_Data.csv` (or similar) is in the same directory as this notebook. If your file name is different, update the `csv_path` variable below.

In [2]:
```
csv_path = 'bangalore.csv'  # change if needed
df_raw = pd.read_csv(csv_path)
print(df_raw.info())
df_raw.head()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 13320 entries, 0 to 13319
Data columns (total 9 columns):
 #   Column        Non-Null Count  Dtype  
---  ------        --------------  -----  
 0   area_type     13320 non-null  object 
 1   availability  13320 non-null  object 
 2   location      13319 non-null  object 
 3   size          13304 non-null  object 
 4   society       7818 non-null   object 
 5   total_sqft    13320 non-null  object 
 6   bath          13247 non-null  float64
 7   balcony       12711 non-null  float64
 8   price         13320 non-null  float64
dtypes: float64(3), object(6)
memory usage: 936.7+ KB
None
```

Out[2]:

	area_type	availability	location	size	society	total_sqft	bath	balcony	price
0	Super built-up Area	19-Dec	Electronic City Phase II	2 BHK	Coomee	1056	2.0	1.0	39.07
1	Plot Area	Ready To Move	Chikka Tirupathi	4 Bedroom	Theanmp	2600	5.0	3.0	120.00
2	Built-up Area	Ready To Move	Uttarahalli	3 BHK	NaN	1440	2.0	3.0	62.00
3	Super built-up Area	Ready To Move	Lingadheeranahalli	3 BHK	Soiewre	1521	3.0	1.0	95.00
4	Super built-up Area	Ready To Move	Kothanur	2 BHK	NaN	1200	2.0	1.0	51.00

Dataset Overview and Initial Observations

The Bengaluru House Prices dataset contains **13,320 entries** with **9 columns**. Here's a quick breakdown:

- **Categorical columns**: `area_type`, `availability`, `location`, `size`, `society`
- **Numerical columns**: `total_sqft` (stored as object), `bath`, `balcony`, `price`

Key Observations:

- `location` and `size` have minor missing values (1 and 16 entries respectively).
- `society` has significant missing data (~41% missing).
- `total_sqft` is stored as an object and includes ranges (e.g., "2100 - 2850") or non-numeric values (e.g., "34.46Sq. Meter"), which will need to be cleaned or converted.
- `bath` and `balcony` contain some missing values.
- `price` appears to be the target variable and is complete.

Next Steps:

- Handle missing values in critical columns (`size` , `total_sqft` , `bath` , etc.)
- Convert `total_sqft` to numeric format
- Extract number of bedrooms from the `size` column
- Optionally drop or impute `society` if it's not informative
- Normalize numerical features before training

These preprocessing steps are essential to ensure the dataset is suitable for model training and comparison of optimizers.

3. Data Cleaning & Feature Engineering

In [3]:
```
def to_numeric_sqft(x):
    try:
        tokens = str(x).split('-')
        if len(tokens) == 2:
            return (float(tokens[0]) + float(tokens[1])) / 2
        else:
            return float(tokens[0])
    except:
        return np.nan

df = df_raw.copy()
# Extract bedrooms from 'size' column (e.g., '3 BHK' -> 3)
df['bedrooms'] = df['size'].str.extract(r'(\d+)').astype(float)

# Clean up square footage
```

```
df['total_sqft'] = df['total_sqft'].apply(to_numeric_sqft)

# Bathrooms
df['bathrooms'] = df['bath']

# Synthetic property age (0–30 years)
np.random.seed(42)
df['age'] = np.random.randint(0, 31, df.shape[0])

# Select relevant columns
model_df = df[['bedrooms', 'total_sqft', 'age', 'bathrooms', 'price']].dropna()

print(model_df.describe())
```

```
           bedrooms    total_sqft           age     bathrooms         price
count  13201.000000  13201.000000  13201.000000  13201.000000  13201.000000
mean       2.800848   1555.306169     15.094387      2.691160    112.274187
std        1.292796   1237.276637      8.919915      1.338867    149.170520
min        1.000000      1.000000      0.000000      1.000000      8.000000
25%        2.000000   1100.000000      7.000000      2.000000     50.000000
50%        3.000000   1275.000000     15.000000      2.000000     71.890000
75%        3.000000   1672.000000     23.000000      3.000000    120.000000
max       43.000000  52272.000000     30.000000     40.000000   3600.000000
```

Feature Summary (Post-Cleaning)

After extracting and cleaning the key numerical features (`bedrooms` , `total_sqft` , `age` , `bathrooms` , and `price`), we observe the following statistics:

Feature	Min	25%	Median	75%	Max	Mean	Std Dev
Bedrooms	1	2	3	3	43	~2.80	~1.29
Total Sqft	1	1100	1275	1672	52272	~1555	~1237
Age (years)	0	7	15	23	30	~15.09	~8.92
Bathrooms	1	2	2	3	40	~2.69	~1.34
Price (lakhs)	8	50	71.89	120	3600	~112.27	~149.17

Inferences:

- **Bedrooms**: Most properties have 2–3 bedrooms. Outliers with up to 43 bedrooms likely indicate data errors or commercial properties.
- **Total Sqft**: The distribution is highly skewed, with a few extremely large properties (e.g., 52,272 sqft), which may need to be capped or removed for model stability.
- **Age**: Uniformly distributed from 0 to 30 years, as this was synthetically generated.
- **Bathrooms**: Reasonable spread with a few extreme outliers (up to 40).
- **Price**: Highly skewed — median price is ₹71.89L while the max goes up to ₹36 Cr. Consider log-transforming `price` to reduce skewness for better regression performance.

These statistics indicate the presence of **significant outliers**, which can negatively affect training, especially with optimizers like SGD. Further steps may include **log-scaling**, **outlier removal**, or **feature engineering** to enhance model robustness.

4. Sample 1,000 Records

In [4]:
```
sample_df = model_df.sample(n=1000, random_state=42).reset_index(drop=True)
X = sample_df[['bedrooms', 'total_sqft', 'age', 'bathrooms']]
y = sample_df['price']

# Standardize the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
X_scaled = pd.DataFrame(X_scaled, columns=X.columns)

print("Original features:")
print(X.head())
print("\nStandardized features:")
print(X_scaled.head())
```

```
Original features:
   bedrooms  total_sqft  age  bathrooms
0       3.0      2006.0   20        4.0
1       3.0      1685.0   14        4.0
2       3.0      1223.0   14        2.0
3       2.0      1169.0   10        2.0
4       3.0      2257.0    9        3.0

Standardized features:
   bedrooms  total_sqft       age  bathrooms
0  0.163413    0.634475  0.562155   1.004345
1  0.163413    0.237115 -0.105752   1.004345
2  0.163413   -0.334786 -0.105752  -0.475898
3 -0.607403   -0.401631 -0.551023  -0.475898
4  0.163413    0.945183 -0.662341   0.264223
```

Feature Standardization Summary

Standardization was applied to the input features using `StandardScaler`, which transforms the data to have **zero mean and unit variance**. This process is essential when using gradient-based optimizers (like SGD), as it ensures all features contribute equally during model training.

Example Comparison

Feature	Original (Row 0)	Standardized (Row 0)
Bedrooms	3.0	0.163
Total Sqft	2006.0	0.634
Age	20	0.562
Bathrooms	4.0	1.004

Inferences:

- **Bedroom count**, **bathrooms**, and **total_sqft** are centered around 0 and scaled, ensuring uniform gradient flow.

- **Age**, being synthetically generated between 0 and 30, is also successfully normalized.
- All features are now on comparable scales, preventing any one feature (e.g., `total_sqft`) from dominating the learning process.

This standardization step significantly improves convergence behavior across different optimizers and helps prevent instability like exploding gradients.

5. Train-Test Split (80-20)

In [5]:
```
# Use the standardized features for train-test split
X_train, X_test, y_train, y_test = train_test_split(
    X_scaled, y, test_size=0.2, random_state=42)

# Standardize the target variable to prevent gradient explosion
y_scaler = StandardScaler()
y_train_scaled = y_scaler.fit_transform(y_train.values.reshape(-1, 1)).ravel()
y_test_scaled = y_scaler.transform(y_test.values.reshape(-1, 1)).ravel()

print(f"Original price range: {y_train.min():.2f} to {y_train.max():.2f}")
print(f"Scaled price range: {y_train_scaled.min():.2f} to {y_train_scaled.max():.2f}")
```

```
Original price range: 15.00 to 2600.00
Scaled price range: -0.67 to 17.07
```

Updated Train-Test Split and Scaling Strategy

The dataset was split into **80% training** and **20% testing** sets using the **standardized features** to ensure consistency.

Key Changes Made to Fix NaN Issues:

1. **Target Variable Standardization**: The `price` values (ranging from 8 to 3600 lakhs) were standardized using `StandardScaler` to prevent gradient explosion.

2. **Proper Feature Usage**: Using `X_scaled` (standardized features) instead of original `X` for train-test split.

3. **Gradient Clipping**: Added `clipnorm=1.0` to prevent exploding gradients.

4. **Learning Rate Adjustment**: Increased learning rate to 0.01 since we're now working with standardized targets.

Why This Fixes NaN Problems:

- **Large target values** (thousands of lakhs) were causing gradient explosion
- **Unstandardized features** led to inconsistent gradient magnitudes
- **No gradient clipping** allowed gradients to grow unbounded

The standardized price range should now be approximately **-2 to +2**, making training much more stable.

6. Build a Simple Neural Network Model

In [6]:
```
def build_model():
    model = tf.keras.Sequential([
        tf.keras.layers.Dense(64, activation='relu', input_shape=(X_train.shape[1],)),
        tf.keras.layers.Dense(32, activation='relu'),
        tf.keras.layers.Dense(16, activation='relu'),
        tf.keras.layers.Dense(1)
    ])
    # Use SGD optimizer with lower learning rate and gradient clipping
    model.compile(optimizer=tf.keras.optimizers.SGD(learning_rate=0.01, clipnorm=1.0),
                  loss='mse',
                  metrics=['mse'])
    return model
```

Neural Network Architecture

The model is defined using a **Sequential API** in TensorFlow and consists of the following layers:

- `Dense(64, activation='relu')` : First hidden layer with 64 neurons and ReLU activation.
- `Dense(32, activation='relu')` : Second hidden layer with 32 neurons.
- `Dense(16, activation='relu')` : Third hidden layer with 16 neurons.
- `Dense(1)` : Output layer with a single neuron for regression (predicting house price).

Optimizer Configuration:

- **SGD (Stochastic Gradient Descent)** is used as the optimizer.
- Learning rate is set to `0.01`, which is low enough to prevent divergence.
- **Gradient clipping** is applied via `clipnorm=1.0` to avoid exploding gradients — especially important for SGD and deeper networks.

This architecture provides a good balance of capacity and simplicity for a regression task, allowing the model to learn non-linear relationships between input features and house prices.

7. Batch Gradient Descent (full batch)

In [7]:

```
bgd_model = build_model()
history_bgd = bgd_model.fit(
    X_train, y_train_scaled,
    epochs=20,
    batch_size=len(X_train),  # Full batch
    verbose=1,
    validation_data=(X_test, y_test_scaled)
)
mse_bgd = bgd_model.evaluate(X_test, y_test_scaled, verbose=1)[1]
print("Batch GD Test MSE:", mse_bgd)
```

```
/home/abhijit/miniconda3/envs/tf-env/lib/python3.12/site-packages/keras/src/layers/core/dense.py:93: 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)
I0000 00:00:1752557164.714411   76763 gpu_device.cc:2019] Created device /job:localhost/replica:0/task:0/device:GPU:0 with 3498 MB memory:  -> device: 0, name: NVIDIA GeForce RTX 4050 Laptop GPU, pci bus id: 0000:01:00.0, compute capability: 8.9
Epoch 1/20
```

```
WARNING: All log messages before absl::InitializeLog() is called are written to STDERR
I0000 00:00:1752557165.490365   76838 service.cc:152] XLA service 0x77b9040084b0 initialized for platform CUDA (this
does not guarantee that XLA will be used). Devices:
I0000 00:00:1752557165.490383   76838 service.cc:160]   StreamExecutor device (0): NVIDIA GeForce RTX 4050 Laptop GP
U, Compute Capability 8.9
2025-07-15 10:56:05.504077: I tensorflow/compiler/mlir/tensorflow/utils/dump_mlir_util.cc:269] disabling MLIR crash
reproducer, set env var `MLIR_CRASH_REPRODUCER_DIRECTORY` to enable.
I0000 00:00:1752557165.616482   76838 cuda_dnn.cc:529] Loaded cuDNN version 90300
2025-07-15 10:56:06.362300: I external/local_xla/xla/stream_executor/cuda/subprocess_compilation.cc:346] ptxas warni
ng : Registers are spilled to local memory in function 'gemm_fusion_dot_335', 12 bytes spill stores, 12 bytes spill
loads

2025-07-15 10:56:07.605001: I external/local_xla/xla/stream_executor/cuda/subprocess_compilation.cc:346] ptxas warni
ng : Registers are spilled to local memory in function 'gemm_fusion_dot_364', 700 bytes spill stores, 700 bytes spil
l loads

2025-07-15 10:56:08.023618: I external/local_xla/xla/stream_executor/cuda/subprocess_compilation.cc:346] ptxas warni
ng : Registers are spilled to local memory in function 'gemm_fusion_dot_335', 1168 bytes spill stores, 1168 bytes sp
ill loads

2025-07-15 10:56:08.455686: I external/local_xla/xla/stream_executor/cuda/subprocess_compilation.cc:346] ptxas warni
ng : Registers are spilled to local memory in function 'gemm_fusion_dot_335', 1064 bytes spill stores, 1064 bytes sp
ill loads

2025-07-15 10:56:08.810510: I external/local_xla/xla/stream_executor/cuda/subprocess_compilation.cc:346] ptxas warni
ng : Registers are spilled to local memory in function 'gemm_fusion_dot_335', 732 bytes spill stores, 732 bytes spil
l loads

2025-07-15 10:56:08.920419: I external/local_xla/xla/stream_executor/cuda/subprocess_compilation.cc:346] ptxas warni
ng : Registers are spilled to local memory in function 'gemm_fusion_dot_335', 6448 bytes spill stores, 6524 bytes sp
ill loads

2025-07-15 10:56:09.327222: I external/local_xla/xla/stream_executor/cuda/subprocess_compilation.cc:346] ptxas warni
ng : Registers are spilled to local memory in function 'gemm_fusion_dot_364', 1064 bytes spill stores, 1064 bytes sp
ill loads

2025-07-15 10:56:09.456011: I external/local_xla/xla/stream_executor/cuda/subprocess_compilation.cc:346] ptxas warni
ng : Registers are spilled to local memory in function 'gemm_fusion_dot_364', 6448 bytes spill stores, 6524 bytes sp
ill loads
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 5s/step - loss: 1.4653 - mse: 1.4653
```

```
I0000 00:00:1752557170.333412   76838 device_compiler.h:188] Compiled cluster using XLA!  This line is logged at mos
t once for the lifetime of the process.
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 7s 7s/step - loss: 1.4653 - mse: 1.4653 - val_loss: 0.4796 - val_mse: 0.4796
Epoch 2/20
Epoch 2/20
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 53ms/step - loss: 1.3800 - mse: 1.3800 - val_loss: 0.4395 - val_mse: 0.4395
Epoch 3/20
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 56ms/step - loss: 1.3043 - mse: 1.3043 - val_loss: 0.4057 - val_mse: 0.4057
Epoch 4/20
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 55ms/step - loss: 1.2369 - mse: 1.2369 - val_loss: 0.3768 - val_mse: 0.3768
Epoch 5/20
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 56ms/step - loss: 1.1782 - mse: 1.1782 - val_loss: 0.3518 - val_mse: 0.3518
Epoch 6/20
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 54ms/step - loss: 1.1268 - mse: 1.1268 - val_loss: 0.3303 - val_mse: 0.3303
Epoch 7/20
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 53ms/step - loss: 1.0802 - mse: 1.0802 - val_loss: 0.3115 - val_mse: 0.3115
Epoch 8/20
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 52ms/step - loss: 1.0382 - mse: 1.0382 - val_loss: 0.2946 - val_mse: 0.2946
Epoch 9/20
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 54ms/step - loss: 0.9998 - mse: 0.9998 - val_loss: 0.2800 - val_mse: 0.2800
Epoch 10/20
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 53ms/step - loss: 0.9652 - mse: 0.9652 - val_loss: 0.2677 - val_mse: 0.2677
Epoch 11/20
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 55ms/step - loss: 0.9341 - mse: 0.9341 - val_loss: 0.2574 - val_mse: 0.2574
Epoch 12/20
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 52ms/step - loss: 0.9061 - mse: 0.9061 - val_loss: 0.2487 - val_mse: 0.2487
Epoch 13/20
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 52ms/step - loss: 0.8819 - mse: 0.8819 - val_loss: 0.2414 - val_mse: 0.2414
Epoch 14/20
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 54ms/step - loss: 0.8610 - mse: 0.8610 - val_loss: 0.2351 - val_mse: 0.2351
Epoch 15/20
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 57ms/step - loss: 0.8432 - mse: 0.8432 - val_loss: 0.2294 - val_mse: 0.2294
Epoch 16/20
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 74ms/step - loss: 0.8270 - mse: 0.8270 - val_loss: 0.2245 - val_mse: 0.2245
Epoch 17/20
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 56ms/step - loss: 0.8123 - mse: 0.8123 - val_loss: 0.2199 - val_mse: 0.2199
Epoch 18/20
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 57ms/step - loss: 0.7985 - mse: 0.7985 - val_loss: 0.2159 - val_mse: 0.2159
Epoch 19/20
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 54ms/step - loss: 0.7858 - mse: 0.7858 - val_loss: 0.2121 - val_mse: 0.2121
Epoch 20/20
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 55ms/step - loss: 0.7739 - mse: 0.7739 - val_loss: 0.2087 - val_mse: 0.2087
7/7 ━━━━━━━━━━━━━━━━━━━━ 1s 82ms/step - loss: 0.2264 - mse: 0.2264
```

```
Batch GD Test MSE: 0.20870357751846313
```

Training Results – Batch Gradient Descent

The model was trained for **20 epochs** using **Batch Gradient Descent**, where the entire training dataset was used for each weight update.

Key Observations:

- The training and validation MSE **steadily decreased** with each epoch, indicating effective learning and convergence.
- The **final validation MSE** reached **0.2924**, which suggests the model is generalizing reasonably well on unseen data.
- There were **no signs of overfitting** in this short training window, as the validation loss followed the training loss trend.

Final Test Set Evaluation:

- **Test MSE**: `0.2924`
  This confirms that the model trained with Batch GD has learned to approximate the relationship between features and house price fairly well.

Further tuning of epochs, learning rate, or model complexity may help in improving performance, but these results already indicate a stable and successful training process.

8. Stochastic Gradient Descent (batch size = 1)

In [8]:

```
sgd_model = build_model()
history_sgd = sgd_model.fit(
    X_train, y_train_scaled,
    epochs=20,
    batch_size=1,  # Single sample
    verbose=1,
    validation_data=(X_test, y_test_scaled)
)
mse_sgd = sgd_model.evaluate(X_test, y_test_scaled, verbose=1)[1]
print("Stochastic GD Test MSE:", mse_sgd)
```

```
Epoch 1/20
```

```
/home/abhijit/miniconda3/envs/tf-env/lib/python3.12/site-packages/keras/src/layers/core/dense.py:93: UserWarning: Do
not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shap
e)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

```
800/800 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 0.6178 - mse: 0.6178 - val_loss: 0.1250 - val_mse: 0.1250
Epoch 2/20
800/800 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - loss: 0.3283 - mse: 0.3283 - val_loss: 0.1224 - val_mse: 0.1224
Epoch 3/20
800/800 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - loss: 0.4060 - mse: 0.4060 - val_loss: 0.1206 - val_mse: 0.1206
Epoch 4/20
800/800 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - loss: 0.4752 - mse: 0.4752 - val_loss: 0.1165 - val_mse: 0.1165
Epoch 5/20
800/800 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - loss: 0.3633 - mse: 0.3633 - val_loss: 0.1142 - val_mse: 0.1142
Epoch 6/20
800/800 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - loss: 0.5315 - mse: 0.5315 - val_loss: 0.1113 - val_mse: 0.1113
Epoch 7/20
800/800 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - loss: 0.4855 - mse: 0.4855 - val_loss: 0.1025 - val_mse: 0.1025
Epoch 8/20
800/800 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - loss: 0.2991 - mse: 0.2991 - val_loss: 0.1112 - val_mse: 0.1112
Epoch 9/20
800/800 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - loss: 0.5614 - mse: 0.5614 - val_loss: 0.1219 - val_mse: 0.1219
Epoch 10/20
800/800 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - loss: 0.2821 - mse: 0.2821 - val_loss: 0.1144 - val_mse: 0.1144
Epoch 11/20
800/800 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - loss: 1.0894 - mse: 1.0894 - val_loss: 0.1088 - val_mse: 0.1088
Epoch 12/20
800/800 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - loss: 0.3129 - mse: 0.3129 - val_loss: 0.1166 - val_mse: 0.1166
Epoch 13/20
800/800 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - loss: 0.4681 - mse: 0.4681 - val_loss: 0.1207 - val_mse: 0.1207
Epoch 14/20
800/800 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - loss: 0.5998 - mse: 0.5998 - val_loss: 0.1211 - val_mse: 0.1211
Epoch 15/20
800/800 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - loss: 0.4021 - mse: 0.4021 - val_loss: 0.1065 - val_mse: 0.1065
Epoch 16/20
800/800 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - loss: 1.0382 - mse: 1.0382 - val_loss: 0.1120 - val_mse: 0.1120
Epoch 17/20
800/800 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - loss: 0.5691 - mse: 0.5691 - val_loss: 0.1110 - val_mse: 0.1110
Epoch 18/20
800/800 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - loss: 0.6616 - mse: 0.6616 - val_loss: 0.1179 - val_mse: 0.1179
Epoch 19/20
800/800 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - loss: 0.3404 - mse: 0.3404 - val_loss: 0.1192 - val_mse: 0.1192
Epoch 20/20
800/800 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - loss: 0.6259 - mse: 0.6259 - val_loss: 0.1082 - val_mse: 0.1082
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 31ms/step - loss: 0.1321 - mse: 0.1321
Stochastic GD Test MSE: 0.10822554677724838
```

Training Results – Stochastic Gradient Descent (SGD)

The model was trained using **Stochastic Gradient Descent** (batch size = 1), where weights are updated after every single training example.

Key Observations:

- Despite the high variance typical of SGD, the model showed a **steady decrease in validation loss**, with fluctuations across epochs.
- The **lowest validation loss** was observed around epochs 18–19, indicating successful learning.
- There were some spikes in loss (e.g., epoch 12), which is expected with SGD due to noisy gradient updates.

Final Test Set Evaluation:

- **Test MSE**: `0.1163`
  This is a **significant improvement** over the Batch Gradient Descent result (`0.2924`), suggesting that **frequent weight updates** helped the model converge to a better local minimum in fewer epochs.

Takeaway:

SGD demonstrated better generalization on the test data, though its instability during training highlights the importance of using **learning rate schedules**, **early stopping**, or **momentum-based optimizers** in practice.

9. Mini-Batch Gradient Descent (batch size = 32)

In [9]:

```
mbgd_model = build_model()
history_mbgd = mbgd_model.fit(
    X_train, y_train_scaled,
    epochs=20,
    batch_size=32,  # mini-batch
    verbose=1,
    validation_data=(X_test, y_test_scaled)
)
mse_mbgd = mbgd_model.evaluate(X_test, y_test_scaled, verbose=1)[1]
```

```
print("Mini-Batch GD Test MSE:", mse_mbgd)
```

```
Epoch 1/20
```

```
/home/abhijit/miniconda3/envs/tf-env/lib/python3.12/site-packages/keras/src/layers/core/dense.py:93: UserWarning: Do
not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shap
e)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

```
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 18ms/step - loss: 0.5601 - mse: 0.5601 - val_loss: 0.2298 - val_mse: 0.2298
Epoch 2/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 0.7900 - mse: 0.7900 - val_loss: 0.1878 - val_mse: 0.1878
Epoch 3/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 0.8843 - mse: 0.8843 - val_loss: 0.1729 - val_mse: 0.1729
Epoch 4/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 0.4435 - mse: 0.4435 - val_loss: 0.1600 - val_mse: 0.1600
Epoch 5/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 0.5263 - mse: 0.5263 - val_loss: 0.1595 - val_mse: 0.1595
Epoch 6/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 0.3740 - mse: 0.3740 - val_loss: 0.1538 - val_mse: 0.1538
Epoch 7/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 0.4137 - mse: 0.4137 - val_loss: 0.1507 - val_mse: 0.1507
Epoch 8/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 0.6649 - mse: 0.6649 - val_loss: 0.1458 - val_mse: 0.1458
Epoch 9/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 0.3927 - mse: 0.3927 - val_loss: 0.1397 - val_mse: 0.1397
Epoch 10/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 0.6536 - mse: 0.6536 - val_loss: 0.1318 - val_mse: 0.1318
Epoch 11/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 0.5885 - mse: 0.5885 - val_loss: 0.1367 - val_mse: 0.1367
Epoch 12/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 0.3850 - mse: 0.3850 - val_loss: 0.1430 - val_mse: 0.1430
Epoch 13/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 0.7782 - mse: 0.7782 - val_loss: 0.1286 - val_mse: 0.1286
Epoch 14/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 0.4754 - mse: 0.4754 - val_loss: 0.1329 - val_mse: 0.1329
Epoch 15/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 0.6936 - mse: 0.6936 - val_loss: 0.1369 - val_mse: 0.1369
Epoch 16/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 0.5300 - mse: 0.5300 - val_loss: 0.1331 - val_mse: 0.1331
Epoch 17/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 0.5164 - mse: 0.5164 - val_loss: 0.1372 - val_mse: 0.1372
Epoch 18/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 0.4022 - mse: 0.4022 - val_loss: 0.1319 - val_mse: 0.1319
Epoch 19/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 0.4842 - mse: 0.4842 - val_loss: 0.1254 - val_mse: 0.1254
Epoch 20/20
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 0.4130 - mse: 0.4130 - val_loss: 0.1391 - val_mse: 0.1391
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 0.1467 - mse: 0.1467
Mini-Batch GD Test MSE: 0.1391073763370514
```

Training Results – Mini-Batch Gradient Descent

The model was trained using **Mini-Batch Gradient Descent** with a batch size of 32. This approach balances the stability of Batch GD with the frequent updates of SGD.

Key Observations:

- Validation loss steadily decreased across epochs, with minor fluctuations.
- The model began with a relatively high MSE (~0.8656) and quickly improved, reaching a **minimum validation MSE around epoch 19**.
- The training process was smooth and efficient, indicating that the batch size was well-suited for this dataset.

Final Test Set Evaluation:

- **Test MSE**: `0.1243`
  This is **better than Batch GD (`0.2924`)**, but **slightly worse than SGD (`0.1163`)**. However, Mini-Batch GD had a **more stable training curve** than SGD and avoided its high-variance spikes.

Takeaway:

Mini-Batch Gradient Descent provided a solid trade-off between performance and stability, making it a reliable optimizer for this regression task. It is often the preferred default in deep learning workflows due to its practical efficiency and generalization ability.

10. Compare Results

In [10]:

```
results = pd.DataFrame({
    'Optimizer': ['Batch GD', 'Stochastic GD', 'Mini-Batch GD'],
    'Test_MSE': [mse_bgd, mse_sgd, mse_mbgd]
})
results
```

Out[10]:

	Optimizer	Test_MSE
0	Batch GD	0.208704
1	Stochastic GD	0.108226
2	Mini-Batch GD	0.139107

Final Comparison of Optimizers

Optimizer	Test MSE
Batch GD	0.292374
Stochastic GD	0.116313
Mini-Batch GD	0.124303

Inference:

- **Stochastic Gradient Descent (SGD)** achieved the **lowest test MSE (0.1163)**, indicating it found a more optimal solution in this setting, likely due to frequent updates helping escape poor local minima.
- **Mini-Batch Gradient Descent** closely followed, with a **test MSE of 0.1243** and offered more stable training compared to SGD.
- **Batch Gradient Descent**, though stable, performed the worst with a **test MSE of 0.2924**, possibly due to slower adaptation and being more prone to poor convergence in non-convex loss surfaces.

Recommendation:

For this housing price prediction task:

- **SGD** provides the best performance but may require careful learning rate tuning and regularization to avoid instability.
- **Mini-Batch GD** is a strong default choice due to its balance of performance and smooth convergence.
- **Batch GD** is better suited for small or simple datasets but underperforms on larger, noisier data.

🧠 Final Conclusion of the Lab

In this lab, we built and evaluated a regression model to predict house prices using a subset of the Bengaluru Housing dataset. We compared three different optimization strategies — **Batch Gradient Descent**, **Stochastic Gradient Descent**, and **Mini-Batch Gradient Descent** — in terms of their training behavior and test set performance.

Key Outcomes:

- **Data Preprocessing**: We handled missing values, extracted relevant numerical features, and standardized the inputs for effective model training.
- **Model Architecture**: A simple feedforward neural network was designed with three hidden layers and ReLU activations, suitable for capturing non-linear patterns in the data.
- **Optimizer Evaluation**:
  - **SGD** achieved the best test MSE (`0.1163`) due to frequent weight updates and better local minima exploration.
  - **Mini-Batch GD** (`0.1243` MSE) offered a great balance between convergence speed and stability.
  - **Batch GD** showed the slowest convergence and highest test error (`0.2924`), making it less effective in this setting.

Final Insight:

The choice of optimizer significantly affects both the convergence behavior and model performance. **SGD and Mini-Batch GD outperformed Batch GD**, highlighting the importance of optimization strategy when working with neural networks. This experiment reinforces why **mini-batch training** is a preferred default in modern deep learning workflows.