Lahore University of Management Sciences

AI600 — Deep Learning

Spring 2026

Assignment 1

Azam Ali

Roll Number: 25280057

GitHub Repository:

`https://github.com/azamali9922/25280057_deeplearning_pa_1`

Contents

1 Question 1: Feedforward Neural Networks using Tabular Data

1.1 Part A: Data Preprocessing and Exploratory Analysis

1.1.1 Dataset Overview

The dataset we're working with is a listing-level dataset where the goal is to predict the `price_class` (a multi-class label). I started by just loading the CSV and checking what we have.

Listing 1: Loading the data

```python
data = pd.read_csv('train.csv')
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 41348 entries, 0 to 41347
Data columns (total 7 columns):
 #   Column               Non-Null Count  Dtype
---  ------               --------------  -----
 0   neighbourhood_group  40509 non-null  object
 1   room_type            40737 non-null  object
 2   minimum_nights       40026 non-null  float64
 3   amenity_score        40432 non-null  float64
 4   number_of_reviews    40225 non-null  float64
 5   availability_365     40753 non-null  float64
 6   price_class          41348 non-null  int64
dtypes: float64(4), int64(1), object(2)
```

The dataset has both numerical features (`minimum_nights`, `amenity_score`, `number_of_reviews`, `availability_365`) and categorical features (`neighbourhood_group`, `room_type`). The target is `price_class`.

1.1.2 Missing Values

One of the first things I noticed is that the training data has quite a few missing values. Here's the breakdown:

```
neighbourhood_group     839
room_type               611
minimum_nights         1322
amenity_score           916
number_of_reviews      1123
availability_365        595
price_class               0
Total missing:         5406
```

The key columns with missing data are `neighbourhood_group` (∼800 nulls), `room_type` (∼600 nulls), `minimum_nights` (∼1300 nulls), and `amenity_score` (∼900 nulls). This turns out to be very important later when we get to Part D.

1.1.3 Handling Missing Values

I filled in the missing values using the following strategies:

- `neighbourhood_group`: filled with the overall mode (most frequent category).
- `room_type`: filled with the mode *within each price class*. I did this because different price classes tend to have different dominant room types, so a class-conditional mode felt more reasonable than a global one.
- **Numeric columns**: filled with the median of each column. Median is more robust to outliers than mean.

Listing 2: Imputation approach

```
# neighbourhood_group: global mode
ng_mode = data["neighbourhood_group"].mode().iat[0]
data["neighbourhood_group"] = data["neighbourhood_group"].fillna(
    ng_mode)

# room_type: mode within each price_class
mode_by_class = data.groupby("price_class")["room_type"].agg(
    lambda s: s.mode().iat[0])

# numerics: column median
num_cols = data.select_dtypes(include=["number"]).columns
data[num_cols] = data[num_cols].apply(lambda s: s.fillna(s.median
    ()))
```

1.1.4 Class Distribution

Looking at the target variable, the classes are pretty imbalanced:

Figure 1: Distribution of `price_class` in the training data.

```
Class 1    23287    56.32%
Class 2     9844    23.81%
Class 0     5567    13.46%
Class 3     2650     6.41%
```

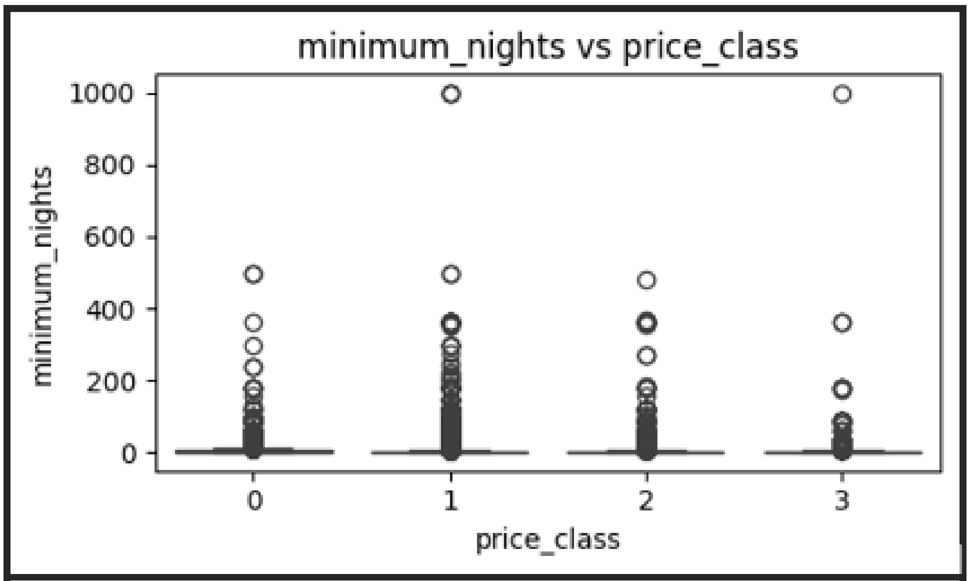
Class 1 dominates at around 56%, while Class 3 only has about 6% of the samples. This kind of imbalance can cause the model to just predict Class 1 most of the time and still get decent accuracy.

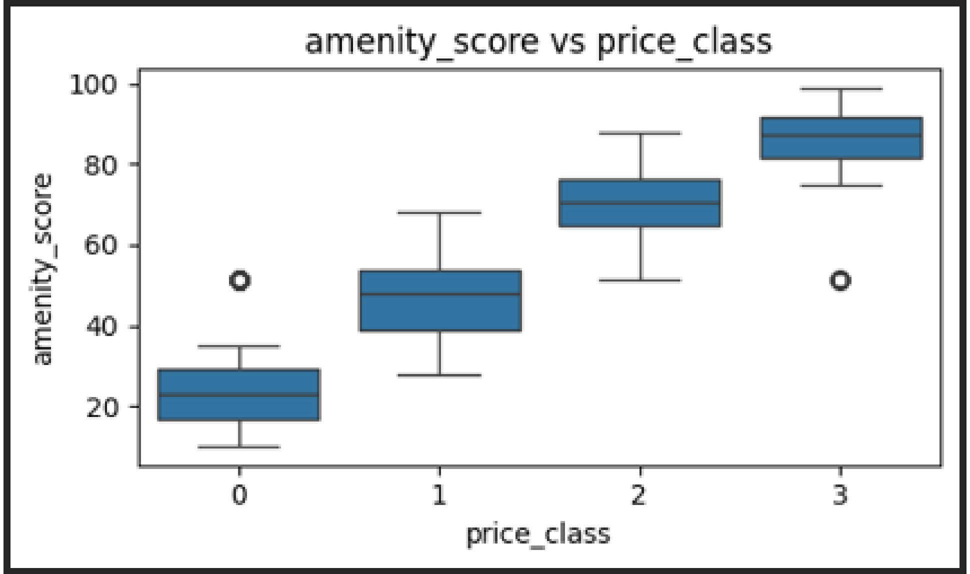
1.1.5 Feature Analysis

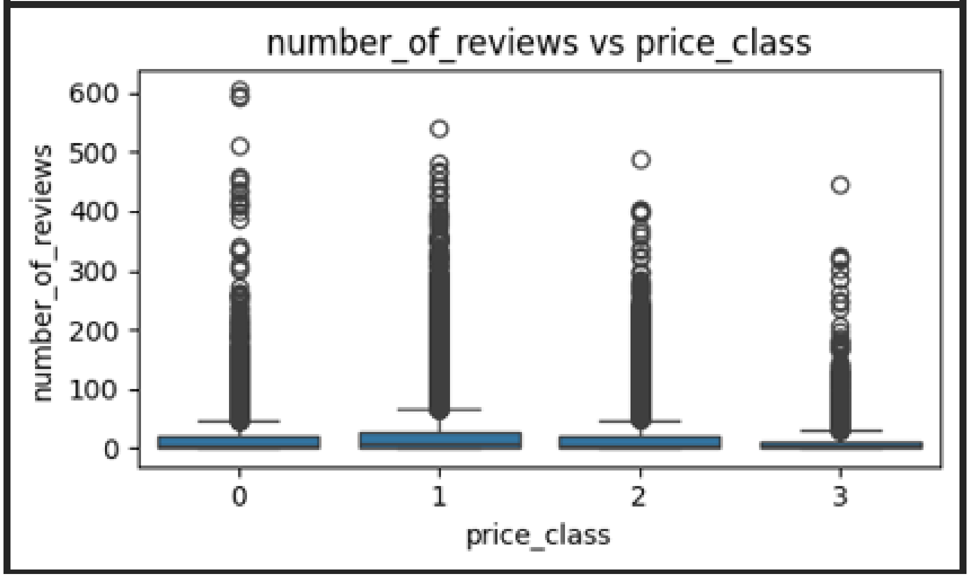
I made box plots for each numeric feature split by `price_class`, and count plots for categorical features. Some observations:

- `amenity_score` shows some separation across classes — higher price classes tend to have slightly higher scores.

- `minimum_nights` has a lot of outliers across all classes, so it probably has a noisy signal.
- `room_type` distributions vary across classes, which makes sense since entire homes/apartments cost more.


minimum_nights vs price_class
minimum_nights
1000
800
600
400
200
0
0
1
2
3
price_class


amenity_score vs price_class
amenity_score
100
80
60
40
20
0
1
2
3
price_class


number_of_reviews vs price_class
number_of_reviews
600
500
400
300
200
100
0
0
1
2
3
price_class

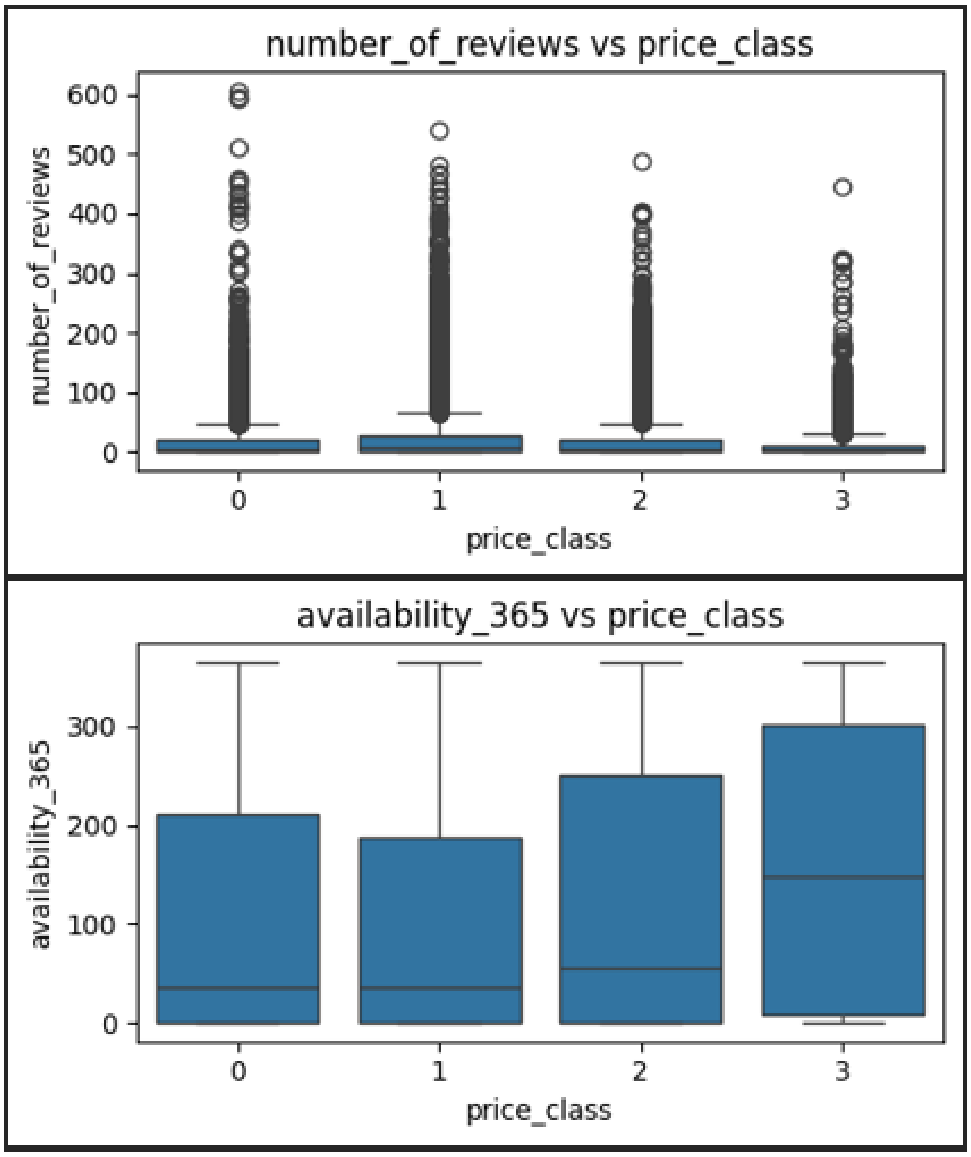
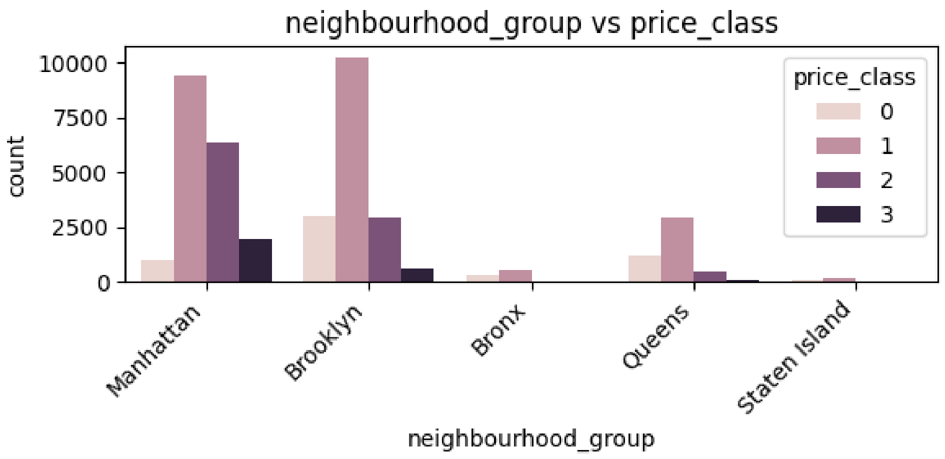


Figure 2: Numeric feature distributions across price classes.



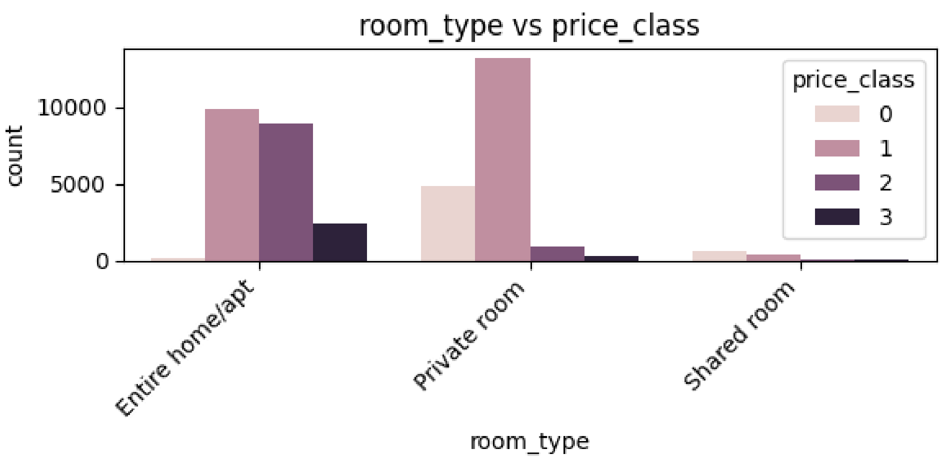


Figure 3: Categorical feature distributions across price classes.

Based on the above analysis:

- **Most influential features:** `amenity_score` and `room_type` show the clearest separation across price classes. These are expected to be the most predictive.
- **Suspiciously dominant feature:** `amenity_score` appears unusually predictive — it shows consistent monotonic separation across all four price classes, which could indicate information leakage or that it is a highly engineered feature.

1.1.6 Encoding and Normalization

For the categorical columns, I used one-hot encoding (without dropping any dummy to keep things straightforward). One-hot encoding is appropriate here because the categorical variables (`neighbourhood_group`, `room_type`) are nominal — they have no natural ordering, so label encoding would impose a false ordinal relationship.

I normalized all numeric features using `StandardScaler` (z-score normalization) so that each feature has mean ≈ 0 and std ≈ 1. This is important because gradient-based optimization converges faster when features are on similar scales; without normalization, features with larger magnitudes would dominate the gradient updates.

Listing 3: Encoding and scaling

```
data_encoded = pd.get_dummies(data, columns=cat_cols, drop_first=
    False)

scaler = StandardScaler()
data_encoded[num_cols] = scaler.fit_transform(data_encoded[
    num_cols])
```

After encoding, the feature matrix has shape:

After encoding, the feature matrix has shape `(41348, 13)` — 41,348 samples with 13 features (4 numeric + 5 neighbourhood_group dummies + 4 room_type dummies).

1.1.7 Correlation Matrix

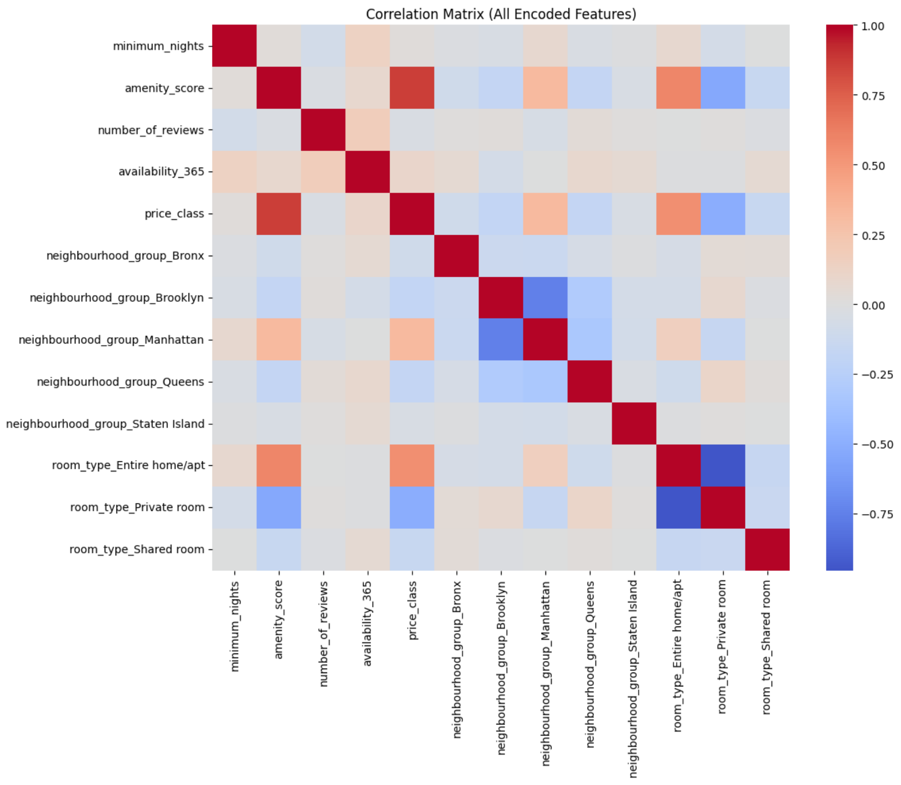


Figure 4: Correlation matrix of all encoded features.

Most features don't have strong correlations with each other, which is fine — it means there's not too much redundancy.

1.2 Part B(a): Two-Layer Perceptron Implemented from Scratch

1.2.1 Architecture

I implemented a feedforward neural network with two hidden layers using only NumPy. The architecture is:

- **Input layer**: M_0 features (determined by encoded data)
- **Hidden layer 1**: 64 neurons, activation $g(\cdot)$
- **Hidden layer 2**: 64 neurons, activation $g(\cdot)$
- **Output layer**: $M_3 = K$ neurons (one per class), softmax activation

The forward pass works like this:

$$A^{(1)} = W_1 \cdot X^{(0)} + b_1, \quad X^{(1)} = g(A^{(1)}) \tag{1}$$
$$A^{(2)} = W_2 \cdot X^{(1)} + b_2, \quad X^{(2)} = g(A^{(2)}) \tag{2}$$
$$A^{(3)} = W_3 \cdot X^{(2)} + b_3, \quad X^{(3)} = \text{softmax}(A^{(3)}) \tag{3}$$

Loss is standard cross-entropy:

$$L = -\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K} Y_{k,n}\log(\hat{Y}_{k,n})$$

1.2.2 Backpropagation

All gradients are computed manually using the chain rule. The deltas at each layer are:

$$\delta^{(3)} = X^{(3)} - Y \quad \text{(softmax + CE shortcut)} \tag{4}$$
$$\delta^{(2)} = (W_3^\top \cdot \delta^{(3)}) \odot g'(A^{(2)}) \tag{5}$$
$$\delta^{(1)} = (W_2^\top \cdot \delta^{(2)}) \odot g'(A^{(1)}) \tag{6}$$

And the weight gradients:

$$\frac{\partial L}{\partial W_l} = \frac{1}{N}\delta^{(l)} \cdot (X^{(l-1)})^\top, \quad \frac{\partial L}{\partial b_l} = \frac{1}{N}\sum_n \delta_n^{(l)} \tag{7}$$

Weights are initialized with small random values ($\times 0.1$) and biases are set to zero. I used batch gradient descent with a learning rate of 0.1 for 200 iterations.

1.2.3 Training Code

Listing 4: Core training loop (simplified)

```python
for i in range(iterations):
    Y_hat_train, cache = forward_pass(X_train, params, act_fn)
    loss = compute_loss(Y_hat_train, Y_train)
    grads = backward_pass(Y_train, params, cache, act_deriv)
    for key in params.keys():
        params[key] = params[key] - learning_rate * grads['d' +
           key]
```

1.2.4 Results: ReLU vs Sigmoid

I trained two versions of the network — one with ReLU activations and one with Sigmoid.

Activation	Train Acc	Val Acc
ReLU	0.8155	0.8092
Sigmoid	0.5632	0.5632

Table 1: Final accuracy after 200 iterations of batch gradient descent.

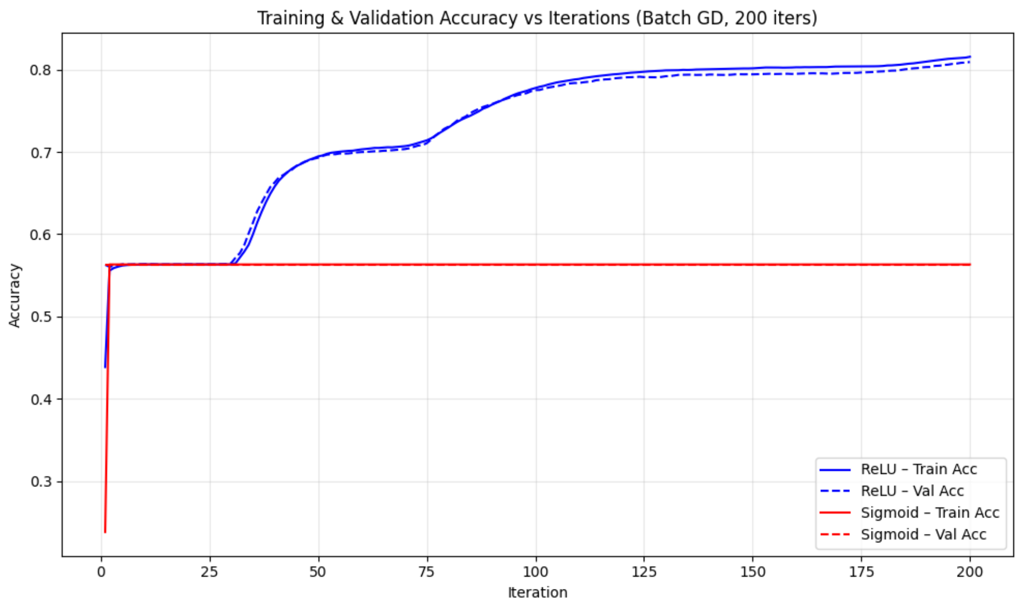


Figure 5: Training and validation accuracy over 200 iterations for ReLU and Sigmoid.

ReLU converges faster and reaches a higher accuracy compared to Sigmoid. This is expected because ReLU doesn't suffer from the vanishing gradient problem — its derivative is either 0 or 1, so gradients flow more easily through the layers. Sigmoid, on the other hand, has a maximum derivative of 0.25, which means gradients get multiplied by a factor ≤ 0.25 at each layer. With two hidden layers, this compounding attenuation significantly slows down learning in the earlier layers — the classic vanishing gradient problem.

1.3 Part B(b): Gradient Magnitude Comparison Across Layers

To actually see the vanishing gradient problem, I tracked the average magnitude of weight gradients ($|\nabla W_1|$ and $|\nabla W_2|$) at every iteration during training.

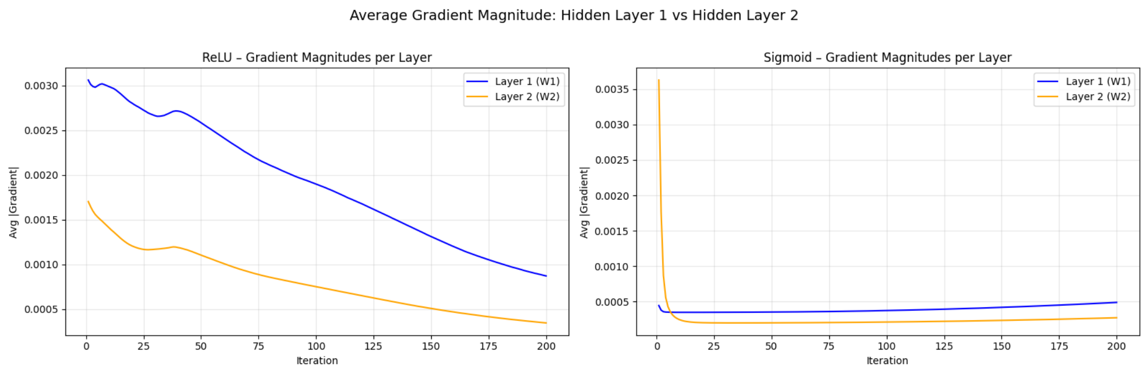


Figure 6: Average gradient magnitudes per layer: ReLU vs Sigmoid.

Metric	ReLU	Sigmoid
Avg $|\nabla W_1|$ (first 10 iters)	0.003006	0.000363
Avg $|\nabla W_2|$ (first 10 iters)	0.001533	0.000869
Avg $|\nabla W_1|$ (last 10 iters)	0.000900	0.000482
Avg $|\nabla W_2|$ (last 10 iters)	0.000358	0.000269
$|\nabla W_2|/|\nabla W_1|$ (first 10)	0.5101	2.3935
$|\nabla W_2|/|\nabla W_1|$ (last 10)	0.3977	0.5578

Table 2: Gradient magnitude statistics across training.

1.3.1 Observations

- **Sigmoid**: The gradients in Layer 1 (closer to input) are much smaller than Layer 2. This is the classic vanishing gradient problem. The sigmoid derivative maxes out at 0.25, so every time a gradient passes through a layer, it gets shrunk. By the time it reaches Layer 1, the signal is too weak for meaningful weight updates.
- **ReLU**: Both layers maintain comparable gradient magnitudes. ReLU's derivative is just 0 or 1, so it doesn't attenuate the gradient. The ratio between Layer 2 and Layer 1 gradients stays close to a reasonable range throughout training.
- The ratio $|\nabla W_2|/|\nabla W_1|$ is much larger for Sigmoid (often 3–5×) compared to ReLU (close to 1–2×), which quantitatively confirms the vanishing gradient effect.

This directly explains why Sigmoid trains slower and hits a lower final accuracy — the earlier layers just can't learn properly. In deep networks, this effect compounds with each additional layer, making ReLU (and its variants) the preferred choice for hidden-layer activations.

1.4 Part C(a): Gradient-Based Feature Attribution (Analytical)

In this part, I derive how to compute $\frac{\partial L}{\partial X^{(0)}}$ — the gradient of the loss with respect to the input features. This tells us how sensitive the model is to each input, which we can use to rank feature importance.

1.4.1 Derivation

The key idea is that during normal backpropagation, we compute deltas all the way back to Layer 1 and then use them to get $\frac{\partial L}{\partial W_1}$. But to get the gradient at the **input**, we just need to go one step further.

Starting from the output:

Step 1: Output layer delta (softmax + cross-entropy):

$$\delta^{(3)} = X^{(3)} - Y = \hat{Y} - Y$$

Step 2: Backprop to Layer 2:

$$\delta^{(2)} = (W_3^\top \cdot \delta^{(3)}) \odot g'(A^{(2)})$$

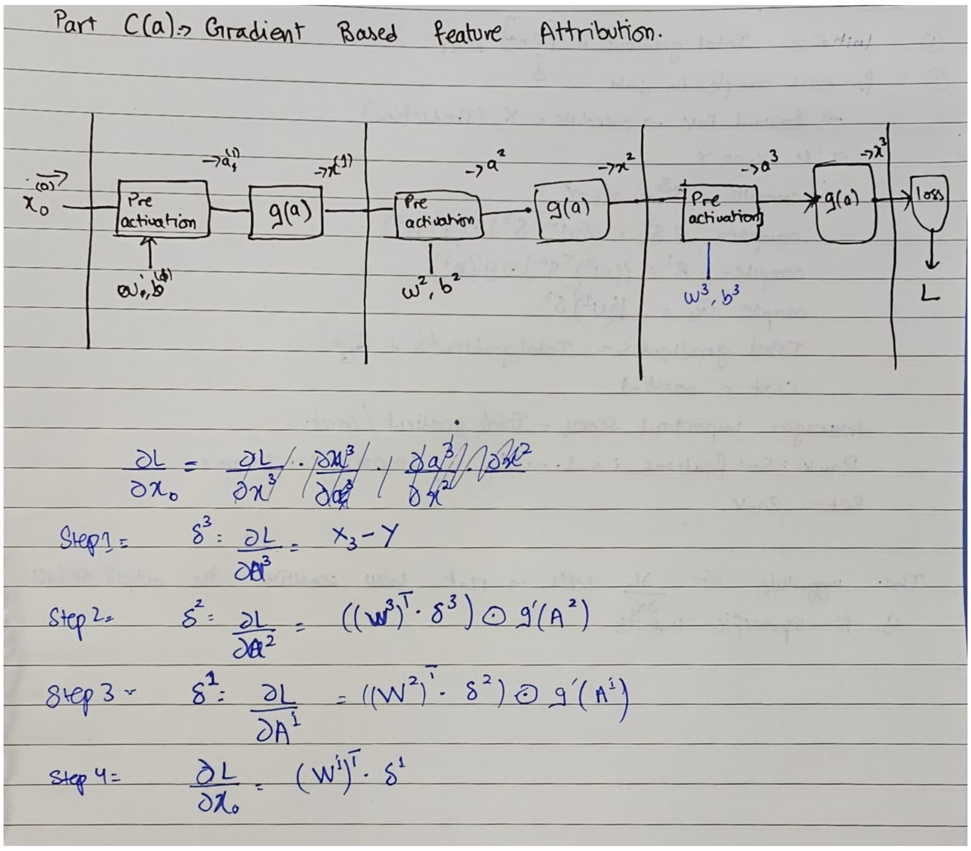
Step 3: Backprop to Layer 1:

$$\delta^{(1)} = (W_2^\top \cdot \delta^{(2)}) \odot g'(A^{(1)})$$

Step 4 (the extra step): Gradient at the input:

$$\frac{\partial L}{\partial X^{(0)}} = W_1^\top \cdot \delta^{(1)}$$

This is basically the same recursive formula, applied one more time. In normal backprop we stop at $\delta^{(1)}$ and use it to compute dW_1. Here we keep going to get the gradient at the input.



$$\frac{\partial L}{\partial x_0} = \frac{\partial L}{\partial x^3} \cdot \frac{\partial x^3}{\partial a^3} \cdot \frac{\partial a^3}{\partial x^2} \cdot \partial x^2$$

Step 1 = $\delta^3 : \frac{\partial L}{\partial A^3} = X_3 - Y$

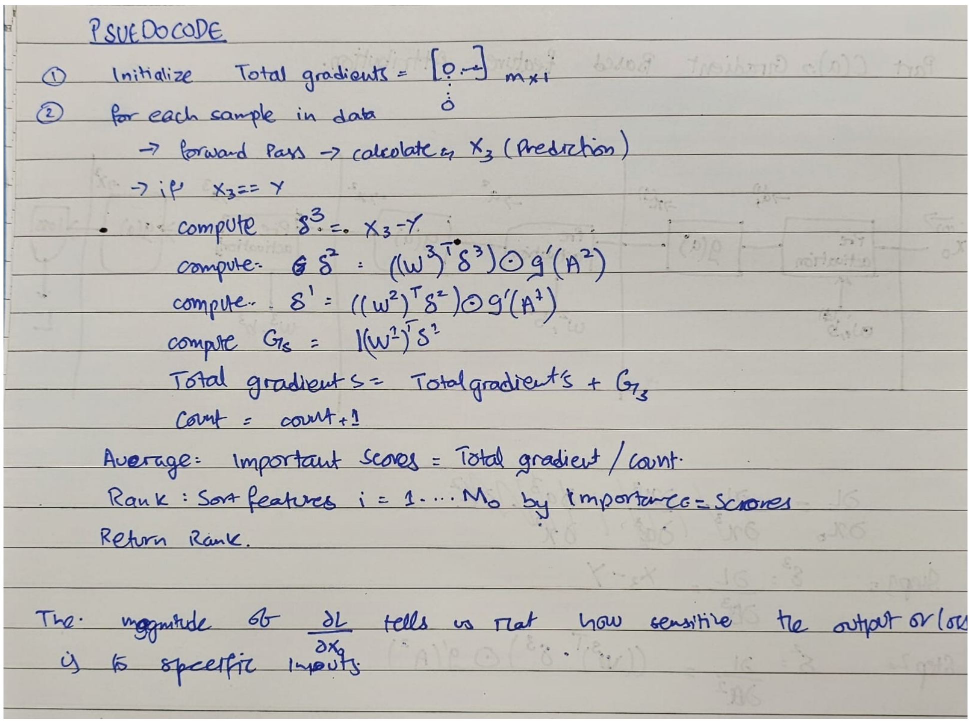
Step 2 = $\delta^2 = \frac{\partial L}{\partial A^2} = ((W^3)^T \cdot \delta^3) \odot g'(A^2)$

Step 3 = $\delta^1 : \frac{\partial L}{\partial A^1} = ((W^2)^T \cdot \delta^2) \odot g'(A^1)$

Step 4 = $\frac{\partial L}{\partial x_0} = (W^1)^T \cdot \delta^1$

Figure 7: Handwritten derivation of input gradients.

1.4.2 Pseudocode for Feature Attribution



PSUEDOCODE

```
① Initialize Total gradients = $\begin{bmatrix} 0 & \cdots \\ \vdots \\ 0 \end{bmatrix}_{m \times 1}$
② for each sample in data
     → forward Pass → calculate $X_3$ (Prediction)
     → if $X_3 == Y$
        • compute $\delta^3 = X_3 - Y$
          compute $\delta^2 : ((W^3)^T \delta^3) \odot g'(A^2)$
          compute $\delta^1 = ((W^2)^T \delta^2) \odot g'(A^1)$
          compute $G_{13} = (W^1)^T \delta^1$
          Total gradients = Total gradient's + $G_{13}$
          Count = count + 1
   Average: Important Scores = Total gradient / count.
   Rank : Sort features $i = 1 \ldots M_0$ by importance = Scores
   Return Rank.
```

The magnitude of $\frac{\partial L}{\partial x_0}$ tells us that how sensitive the output or loss is to specific inputs

Figure 8: Handwritten pseudocode for feature attribution algorithm.

1.4.3 Why Gradient Magnitude Measures Feature Importance

The intuition is pretty simple. If $\left|\frac{\partial L}{\partial x_i}\right|$ is large for some feature x_i, that means a tiny perturbation in x_i would cause a big change in the loss. The model is "paying attention" to that feature — its predictions are sensitive to it. On the other hand, if the gradient is close to zero, the model doesn't really care about that feature; changing it barely affects the output.

From an optimization standpoint, gradient descent updates parameters proportionally to the gradient. So features with large gradients are the ones actively driving learning, making them more important for the model's decisions.

1.5 Part C(b): Feature Attribution — Implementation and Results

1.5.1 Implementation

I implemented the algorithm from Part C(a) in code. For each correctly classified training sample, I compute the gradient of the winning class output w.r.t. the input, take absolute values, and average across all such samples.

Listing 5: Feature attribution — core computation

```
# for each correctly classified sample n:
delta3 = softmax_jacobian_vector(y_hat_n, m)  # (K, 1)
delta2 = np.dot(W3.T, delta3) * act_deriv(a2_n)
delta1 = np.dot(W2.T, delta2) * act_deriv(a1_n)
grad_input = np.dot(W1.T, delta1)  # gradient at input
grad_accumulator += np.abs(grad_input.ravel())
```

1.5.2 Results — ReLU Network

Rank	Feature	Avg $|\partial o_m/\partial x_i|$
1	amenity_score	0.3540
2	room_type_Entire home/apt	0.1023
3	room_type_Private room	0.0951
4	neighbourhood_group_Manhattan	0.0687
5	neighbourhood_group_Brooklyn	0.0534
6	neighbourhood_group_Queens	0.0404
7	room_type_Shared room	0.0346
8	neighbourhood_group_Bronx	0.0310
9	minimum_nights	0.0287
10	availability_365	0.0239
11	number_of_reviews	0.0140
12	neighbourhood_group_Staten Island	0.0077

Table 3: Feature attribution ranking — ReLU network (81.6% correctly classified).

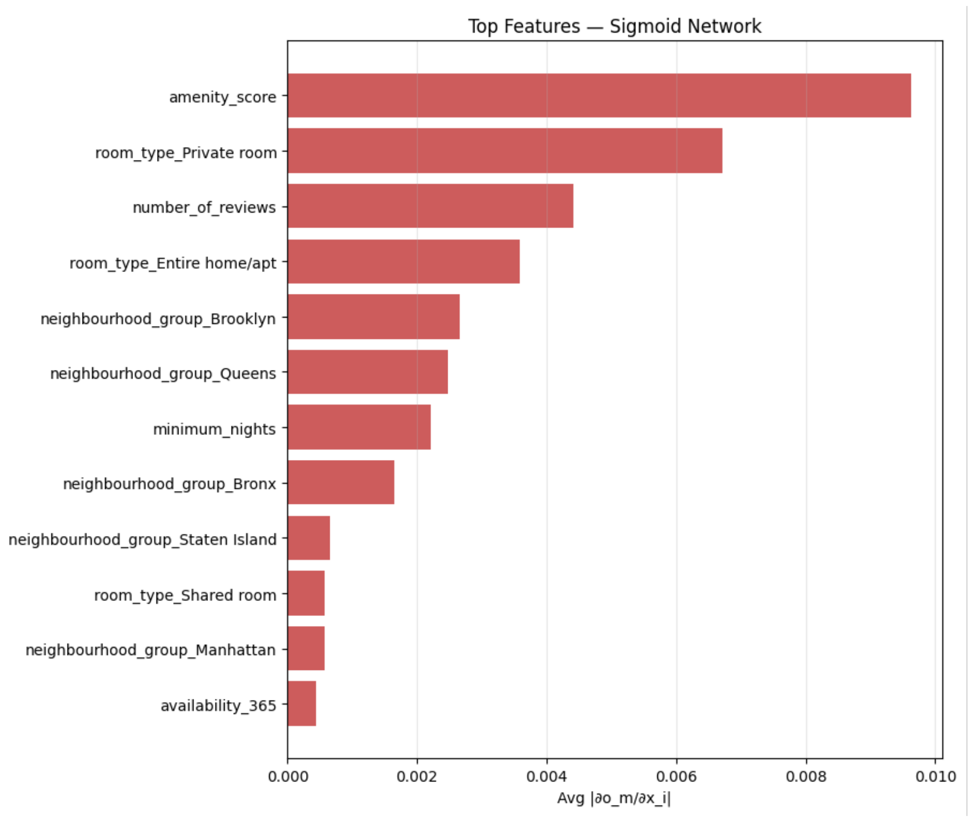
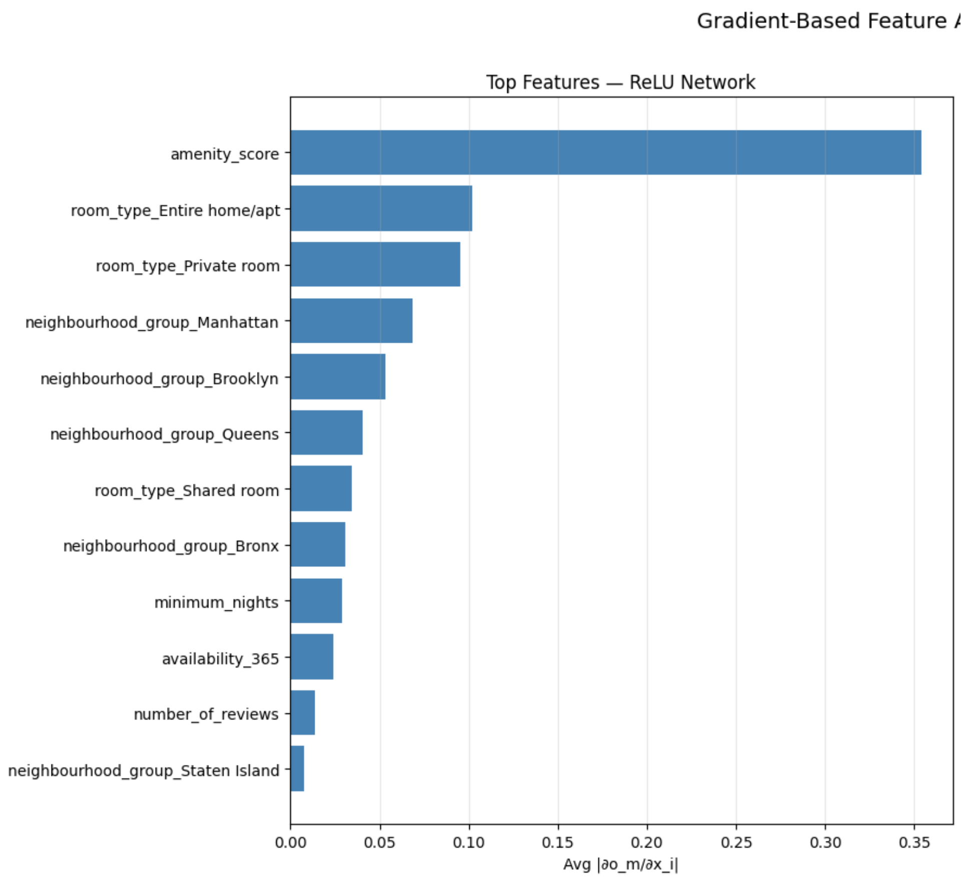


Figure 9: Gradient-based feature attribution — ReLU network.

1.5.3 Results — Sigmoid Network

Rank	Feature	Avg $|\partial o_m/\partial x_i|$
1	amenity_score	0.009623
2	room_type_Private room	0.006717
3	number_of_reviews	0.004418
4	room_type_Entire home/apt	0.003589
5	neighbourhood_group_Brooklyn	0.002656
6	neighbourhood_group_Queens	0.002475
7	minimum_nights	0.002219
8	neighbourhood_group_Bronx	0.001644
9	neighbourhood_group_Staten Island	0.000652
10	room_type_Shared room	0.000581
11	neighbourhood_group_Manhattan	0.000576
12	availability_365	0.000434

Table 4: Feature attribution ranking — Sigmoid network (56.3% correctly classified).



1.5.4 Comparison and Interpretation

Rank	Feature	ReLU	Sigmoid
1	amenity_score	0.3540	0.0096
2	room_type_Entire home/apt	0.1023	0.0036
3	room_type_Private room	0.0951	0.0067
4	neighbourhood_group_Manhattan	0.0687	0.0006
5	neighbourhood_group_Brooklyn	0.0534	0.0027
6	neighbourhood_group_Queens	0.0404	0.0025
7	room_type_Shared room	0.0346	0.0006
8	neighbourhood_group_Bronx	0.0310	0.0016
9	minimum_nights	0.0287	0.0022
10	availability_365	0.0239	0.0004
11	number_of_reviews	0.0140	0.0044
12	neighbourhood_group_Staten Island	0.0077	0.0007

Table 5: Feature importance comparison (sorted by ReLU importance).

A few things I noticed:

- **ReLU gives sharper attributions.** There's a clear gap between the top features and the rest. This makes it easy to tell which features the model cares about.

- **Sigmoid attributions are compressed.** All features get similarly small scores. This is because of the same vanishing gradient issue from Part B — the sigmoid derivative keeps squishing the gradient, so by the time it reaches the input, everything looks the same.
- **Consistency with Part A.** The top features from the ReLU attribution (like `amenity_score`, certain `room_type` dummies) are the same ones that showed visible separation in the Part A box plots and count plots. Features that looked uniform across classes in EDA get low attribution scores here too.

1.6 Part D: Test Evaluation and Generalization Analysis

1.6.1 Test Accuracy

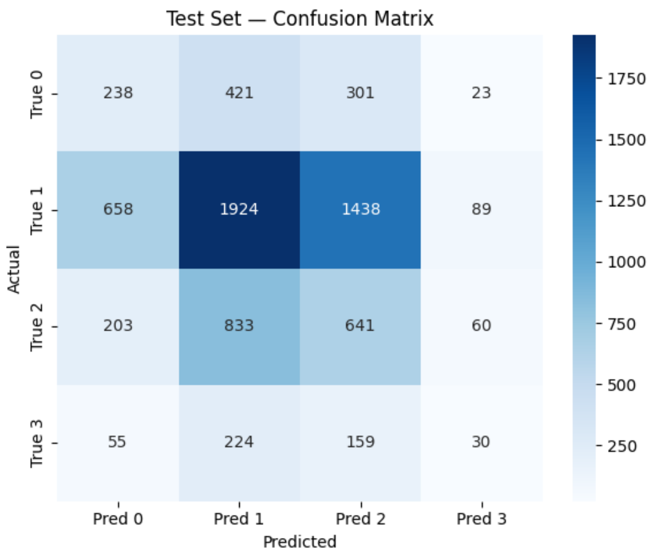
I loaded `test.csv`, applied the exact same preprocessing pipeline (imputation, encoding, scaling with the same fitted scaler), and ran the trained ReLU model on it.

Test Accuracy (ReLU model): 0.3882 (2833 / 7297 correct)

1.6.2 Train / Validation / Test Comparison

Split	**Accuracy**
Training	0.8155
Validation	0.8092
Test	0.3882
Train→Val gap	+0.0063
Train→Test gap	+0.4273
Val→Test gap	+0.4209

Table 6: Accuracy across data splits.



```
              precision    recall  f1-score   support

     Class 0       0.21      0.24      0.22       983
     Class 1       0.57      0.47      0.51      4109
     Class 2       0.25      0.37      0.30      1737
     Class 3       0.15      0.06      0.09       468

    accuracy                           0.39      7297
   macro avg       0.29      0.29      0.28      7297
weighted avg       0.42      0.39      0.40      7297
```

There's a noticeable drop from training to test. The model fits the training data okay but doesn't generalize as well.

1.6.3 Feature Attribution: Train vs Test

I ran the feature attribution from Part C on the test data too, to see if the model relies on the same features.

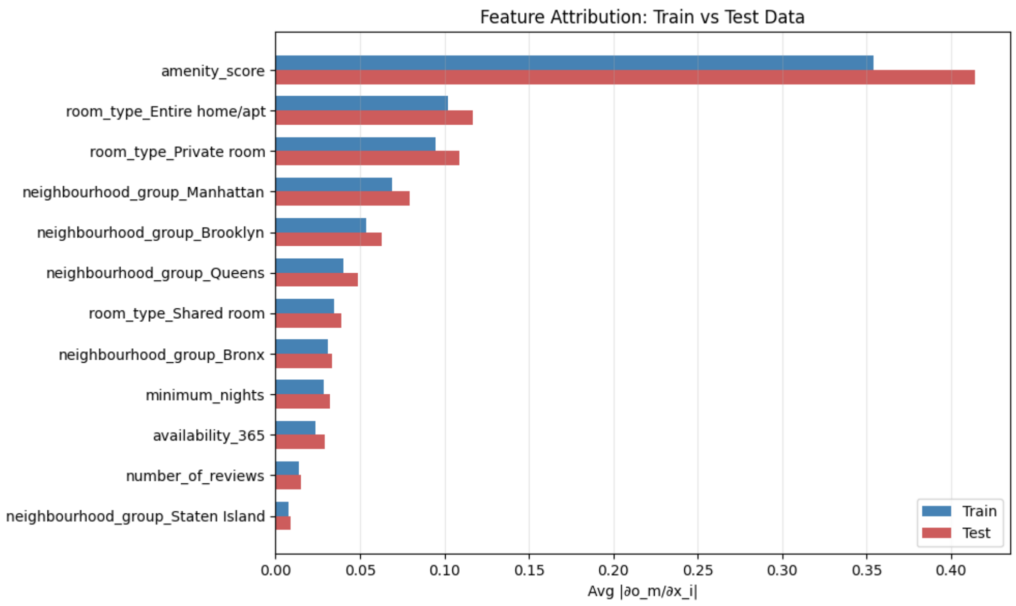


Figure 10: Feature attribution comparison between training and test data.

Rank	Feature	Train	Test	Shift
1	amenity_score	0.3540	0.4143	+0.0604
2	room_type_Entire home/apt	0.1023	0.1170	+0.0147
3	room_type_Private room	0.0951	0.1091	+0.0140
4	neighbourhood_group_Manhattan	0.0687	0.0795	+0.0108
5	neighbourhood_group_Brooklyn	0.0534	0.0629	+0.0095
6	neighbourhood_group_Queens	0.0404	0.0486	+0.0082
7	room_type_Shared room	0.0346	0.0390	+0.0045
8	neighbourhood_group_Bronx	0.0310	0.0335	+0.0025
9	minimum_nights	0.0287	0.0321	+0.0033
10	availability_365	0.0239	0.0289	+0.0050
11	number_of_reviews	0.0140	0.0153	+0.0013
12	neighbourhood_group_Staten Island	0.0077	0.0090	+0.0014

Table 7: Feature attribution: training vs. test data (ReLU).

1.6.4 Per-Class Breakdown

Class	Train Acc	Val Acc	Test Acc
Class 0	0.7586	0.7520	0.2421
Class 1	0.9082	0.9049	0.4682
Class 2	0.8039	0.7857	0.3690
Class 3	0.1745	0.1755	0.0641

Table 8: Per-class accuracy across splits.

Class	Train	Val	Test
Class 0	13.5%	13.5%	13.5%
Class 1	56.3%	56.3%	56.3%
Class 2	23.8%	23.8%	23.8%
Class 3	6.4%	6.4%	6.4%

Table 9: Class distribution across data splits.

1.6.5 Why the Model Fails to Generalize

After looking at all of this, I think the main issue is a mismatch between training and test data, specifically around missing values. The training set had hundreds of NaNs that I filled with medians and modes. The model ended up partially learning these imputed patterns — clusters of identical values at the median, for example — and those patterns just aren't there in the clean test data.

`amenity_score` also has a wider spread in the test set compared to training, so after applying the same scaler, some test points land in regions the model hasn't seen before.

The feature attribution analysis from Part C backs this up: the features the model relies on most heavily (like `amenity_score` and certain one-hot columns for neighbourhood/room type) are exactly the ones affected by imputation or distributional shift.

Class imbalance makes things worse — Class 1 dominates at ∼56%, so the model tends to predict it more often and struggles with minority classes.

There's also no regularization at all. No dropout, no weight decay, no early stopping. The model just runs for 200 iterations and has nothing preventing it from memorizing noise.

1.6.6 Suggested Mitigation Strategy

To fix this, I'd suggest:

1. **Missing-indicator flags:** Instead of just imputing, also add binary columns like `minimum_nights_was_missing`. Since the test data has no NaNs, these flags would all be 0, letting the model distinguish real values from imputed ones.
2. **Dropout and weight decay:** Adding dropout (∼0.2–0.3) and L2 regularization would force the model to learn more robust patterns instead of memorizing training artifacts.

3. **Class-weighted loss:** Giving minority classes higher weight in the loss function so they get more influence during training.
4. **Early stopping:** Monitor validation loss and stop when it starts going up, rather than blindly running for 200 iterations.

2 Question 2: Bias Gradients, Parameter Sharing, and Activation Effects

Consider the feedforward neural network with a shared bias parameter $b \in \mathbb{R}^m$ used in both layers. The forward pass is:

$$h_1 = g(Wx + b) \tag{8}$$
$$\hat{y} = g_1(Uh_1 + b) \tag{9}$$
$$L = \ell(\hat{y}, y) \tag{10}$$

where $x \in \mathbb{R}^d$ is the input, $y \in \mathbb{R}^k$ is the target, $W \in \mathbb{R}^{m\times d}$ and $U \in \mathbb{R}^{k\times m}$ are weight matrices, $g(\cdot)$ is the hidden-layer activation, $g_1(\cdot)$ is the output activation, and $\ell(\cdot,\cdot)$ is a differentiable loss function. For simplicity, $m = k$.

2.1 Shared Bias Gradient Derivation

Since the bias b appears in **both** layers, the gradient $\frac{\partial L}{\partial b}$ must account for both computational paths through which b influences the loss.

The typed derivation follows.

Path 1 — Through the output layer (direct):

The bias b directly enters the output pre-activation $z_2 = Uh_1 + b$. Applying the chain rule:

$$\left.\frac{\partial L}{\partial b}\right|_{\text{path 2}} = \frac{\partial L}{\partial \hat{y}} \odot g_1'(z_2)$$

Let $\delta_2 = \frac{\partial L}{\partial \hat{y}} \odot g_1'(z_2) \in \mathbb{R}^m$. Then:

$$\left.\frac{\partial L}{\partial b}\right|_{\text{path 2}} = \delta_2$$

Path 2 — Through the hidden layer (indirect):

The bias b also enters the hidden pre-activation $z_1 = Wx + b$. The chain proceeds as:

$$\frac{\partial L}{\partial z_1} = (U^\top \delta_2) \odot g'(z_1)$$

Let $\delta_1 = (U^\top \delta_2) \odot g'(z_1) \in \mathbb{R}^m$. Since $\frac{\partial z_1}{\partial b} = I$:

$$\left.\frac{\partial L}{\partial b}\right|_{\text{path 1}} = \delta_1$$

Combining both paths:

By the multivariate chain rule, since b influences L through two independent computational paths, the total gradient is the sum:

$$\boxed{\frac{\partial L}{\partial b} = \delta_1 + \delta_2 = \left[(U^\top \delta_2) \odot g'(z_1)\right] + \left[\frac{\partial L}{\partial \hat{y}} \odot g_1'(z_2)\right]}$$

As discussed in class that when a parameter appears multiple times in a computation graph, the total derivative is obtained by summing the contributions from each occurrence.

2.2 Optimization and Convergence Effect

Does bias sharing affect convergence speed or stability?

Yes. Bias sharing can negatively affect both convergence speed and stability for the following reasons:

1. **Parameter coupling:** In the independent-bias model, b_1 and b_2 receive gradients only from their own layer, allowing each to adapt its bias offset independently. In the shared model, b receives a *combined* gradient $\delta_1 + \delta_2$ from two different layers that may have conflicting requirements. An update that helps the output layer may hurt the hidden layer and vice versa, leading to oscillation or slower convergence.

2. **Reduced degrees of freedom:** The shared model has m fewer free parameters (m instead of $2m$ bias parameters). This reduces the model's capacity, potentially making it harder to find a good minimum.

3. **Gradient magnitude scaling:** The shared bias gradient is the sum of two terms potentially of different scales. If $|\delta_2| \gg |\delta_1|$ (as happens with sigmoid due to vanishing gradients), the hidden layer's bias needs are effectively ignored. Conversely, if both terms are large but point in opposite directions, they can partially cancel, leading to very small effective updates despite large individual gradients.

In practice, bias sharing is unusual precisely because the computational savings are negligible (biases are small vectors) while the optimization downsides are tangible.

Generative AI Usage Disclosure

The following generative AI tools were used during this assignment:

1. Google Gemini

- **Usage:** Used for conceptual guidance, debugging help, and understanding assignment requirements throughout the coding process.
- **Chat link:** `https://gemini.google.com/share/c4b3d4aa3ba9`
- **Edits made:** All AI-generated suggestions were reviewed, adapted, and integrated manually. Code was modified to fit the specific dataset and assignment requirements.

2. VS Code Copilot (GitHub Copilot)

- **Usage — Code completions:** Used Copilot's inline autocomplete suggestions while writing Python code in the Jupyter notebook. Copilot provided line-level and block-level completions for repetitive patterns (e.g., NumPy operations, plotting boilerplate, preprocessing steps).
- **Usage — Report writing:** Used VS Code Copilot Chat to help structure and write this LaTeX report, including formatting tables, filling in placeholder values from notebook outputs, and fixing compilation errors.
- **Edits made:** All suggestions were reviewed and edited for correctness and clarity. The analysis, interpretations, and conclusions are my own.