Deep Learning - Assignment

Roll Number: 25280072
MS in Artificial Intelligence, LUMS

February 14, 2026

QUESTION 1

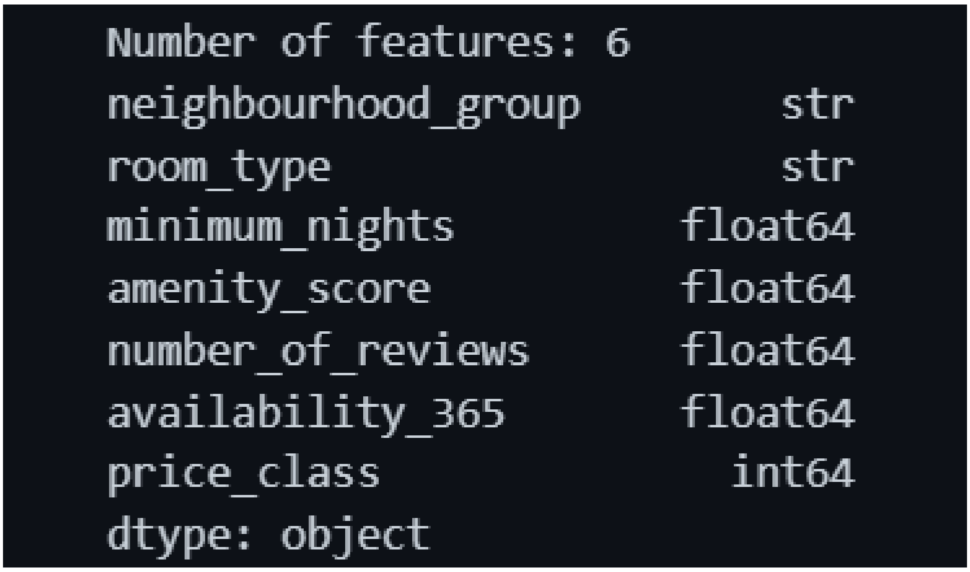
In this assignment, I have implemented a simple feedforward neural network from scratch using Python and NumPy. The network consists of an input layer, one hidden layer with activation (sigmoid and ReLU), and an output layer with softmax activation.

I have trained the network on the AirBnB listings dataset for digit classification. The purpose of this problem is not only to train a feedforward neural network, but also to develop a principled understanding of how data characteristics influence model behavior and generalization

Finally, I have analyzed the generalization performance of the model by evaluating it on a test set and comparing the training and test accuracies to identify any potential overfitting issues.

1 Exploratory Data Analysis (Part A)

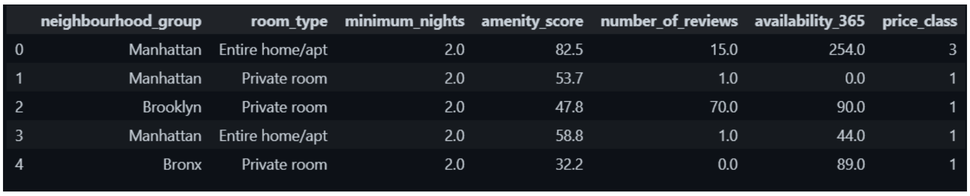
So for part A,I loaded the data and discovered the important features of the dataset, datatypes, no. of feature and their total count.



```
Number of features: 6
neighbourhood_group       str
room_type                 str
minimum_nights        float64
amenity_score         float64
number_of_reviews     float64
availability_365      float64
price_class             int64
dtype: object
```

Figure 1: Initial datatypes.

I have performed necessary data preprocessing steps, which include clearing for the missing values in both categorical column and numerical columns. Filled the categorical data with the mode and numerical data with the mean, Except amenity score and reviews, because these affect the targeted feature directly, thus filled with 0 as they are not available. Then I encoded the categorical features using one-hot encoding to convert them into a format suitable for training the neural network. Finally, I normalized the numerical features to ensure that they are on a similar scale, which helps in faster convergence during training.



	neighbourhood_group	room_type	minimum_nights	amenity_score	number_of_reviews	availability_365	price_class
0	Manhattan	Entire home/apt	2.0	82.5	15.0	254.0	3
1	Manhattan	Private room	2.0	53.7	1.0	0.0	1
2	Brooklyn	Private room	2.0	47.8	70.0	90.0	1
3	Manhattan	Entire home/apt	2.0	58.8	1.0	44.0	1
4	Bronx	Private room	2.0	32.2	0.0	89.0	1

Figure 2: After preprocessing.

Based on the EDA, I selected the feature ”price” as the target variable for the classification task. I created a binary classification problem by categorizing the prices, and during EDA i was able to identify the feature ”amenity score” to have a strong correlation with the target variable, which can be useful for training the neural network. I also visualized the distribution of the target variable and the relationship between the features and the target variable to gain insights into the data.

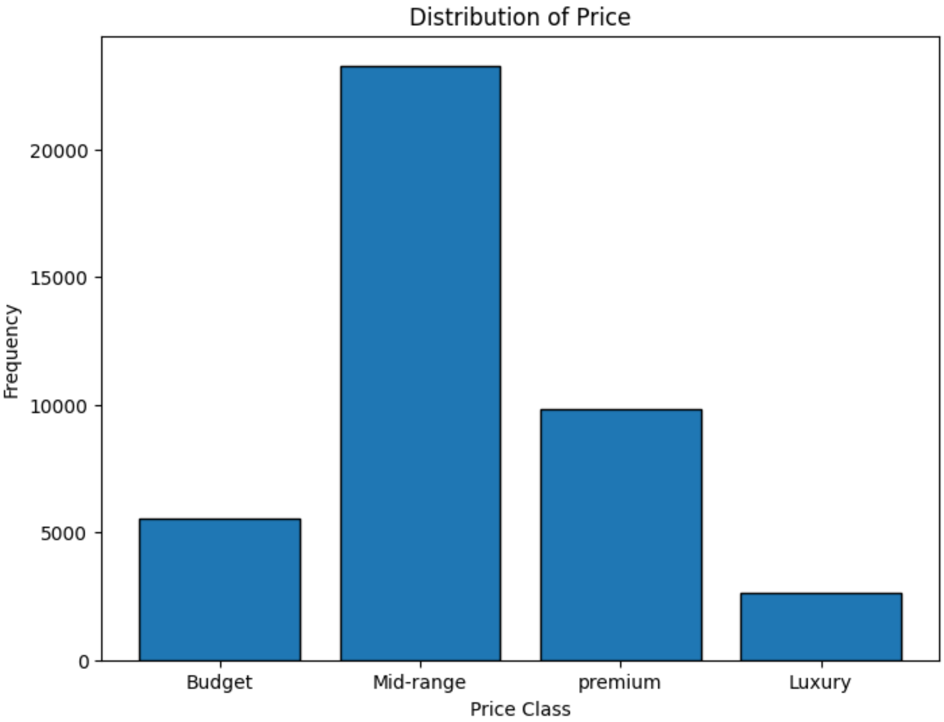


Figure 3: Distribution of target variable

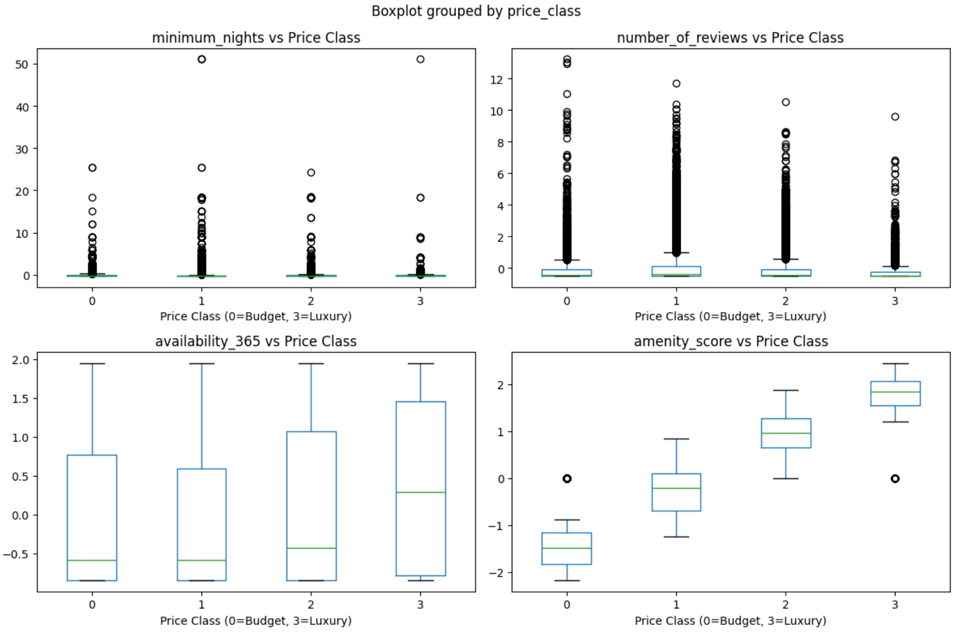
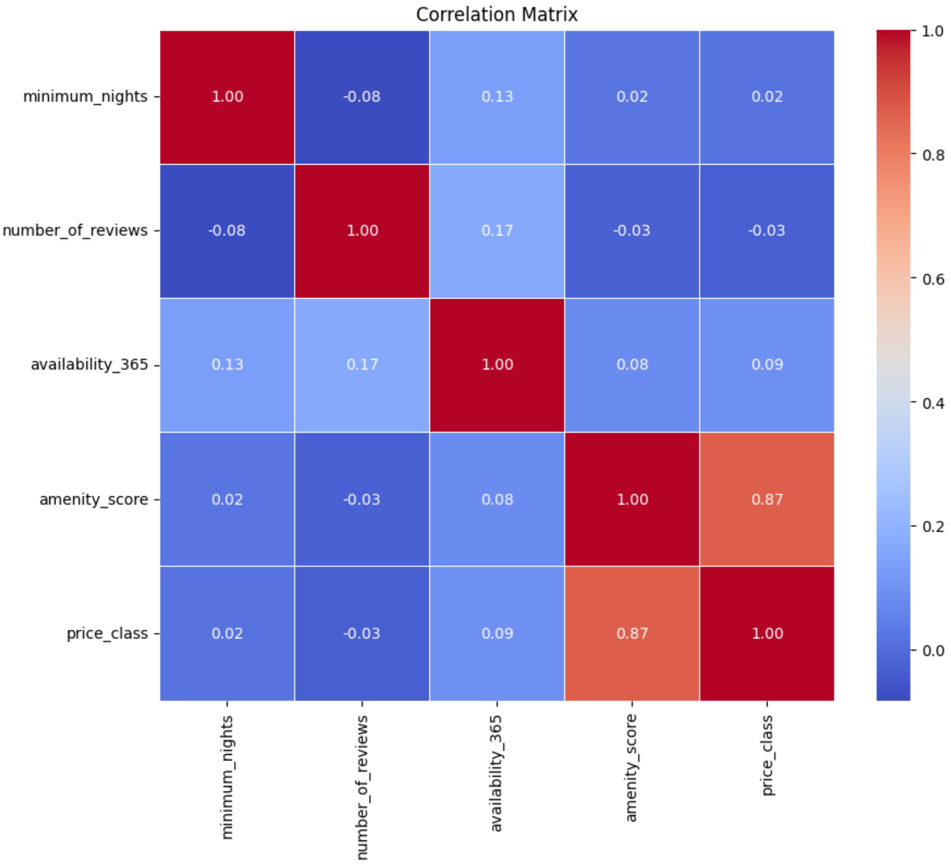


Figure 4: boxplot of price class with respect to other variables



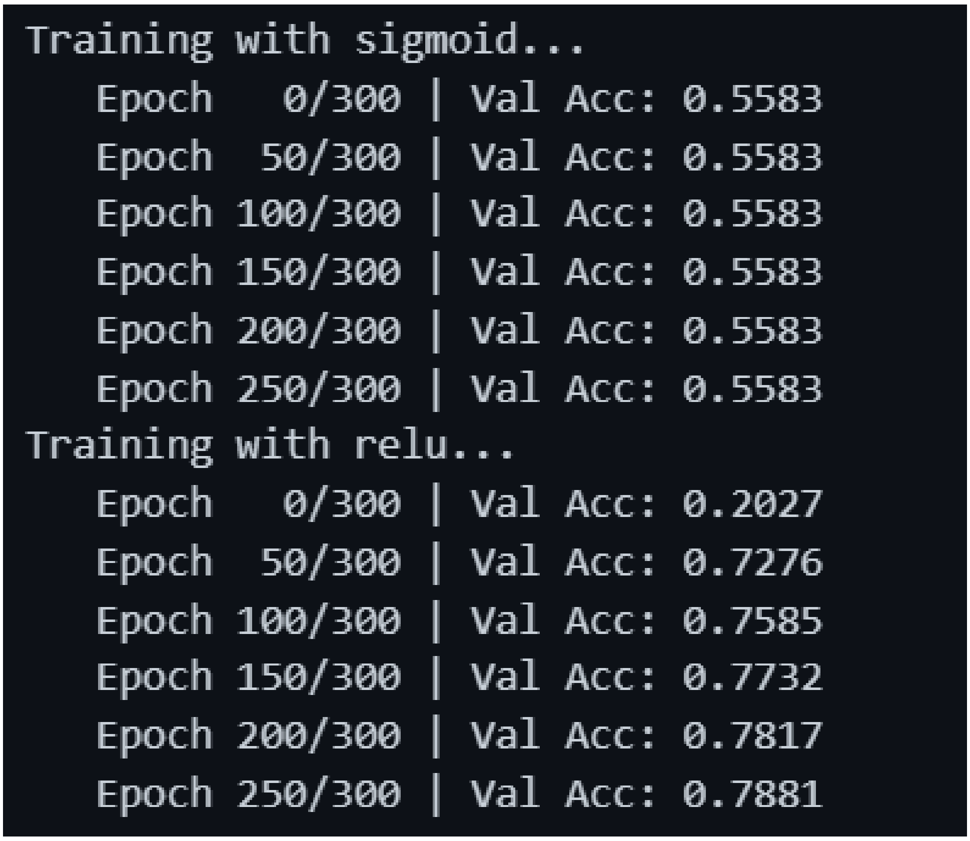
	minimum_nights	number_of_reviews	availability_365	amenity_score	price_class
minimum_nights	1.00	-0.08	0.13	0.02	0.02
number_of_reviews	-0.08	1.00	0.17	-0.03	-0.03
availability_365	0.13	0.17	1.00	0.08	0.09
amenity_score	0.02	-0.03	0.08	1.00	0.87
price_class	0.02	-0.03	0.09	0.87	1.00

Figure 5: correlation of amenity score with price class

2 Neural Network (Part B)

First, I initialized the weights and biases for the hidden layer and output layer. The weights were initialized using a random normal distribution, and the biases were initialized to zero. Set up the activation(including sigmoid and ReLU) and softmax functions for the hidden and output layers, respectively.

Next, I implemented the forward pass to compute the activations for the hidden layer and the output layer. The loss was calculated using the cross-entropy loss function. Backpropagation was implemented to compute the gradients of the loss with respect to the weights and biases, which were then updated using gradient descent.



```
Training with sigmoid...
   Epoch   0/300 | Val Acc: 0.5583
   Epoch  50/300 | Val Acc: 0.5583
   Epoch 100/300 | Val Acc: 0.5583
   Epoch 150/300 | Val Acc: 0.5583
   Epoch 200/300 | Val Acc: 0.5583
   Epoch 250/300 | Val Acc: 0.5583
Training with relu...
   Epoch   0/300 | Val Acc: 0.2027
   Epoch  50/300 | Val Acc: 0.7276
   Epoch 100/300 | Val Acc: 0.7585
   Epoch 150/300 | Val Acc: 0.7732
   Epoch 200/300 | Val Acc: 0.7817
   Epoch 250/300 | Val Acc: 0.7881
```

Figure 6: Training loss and accuracy curves.

I trained the model for a specified number of epochs and monitored the training loss and accuracy to ensure that the model was learning effectively. while tracking gradients history to analyze the learning process and identify any potential issues such as vanishing or exploding gradients.

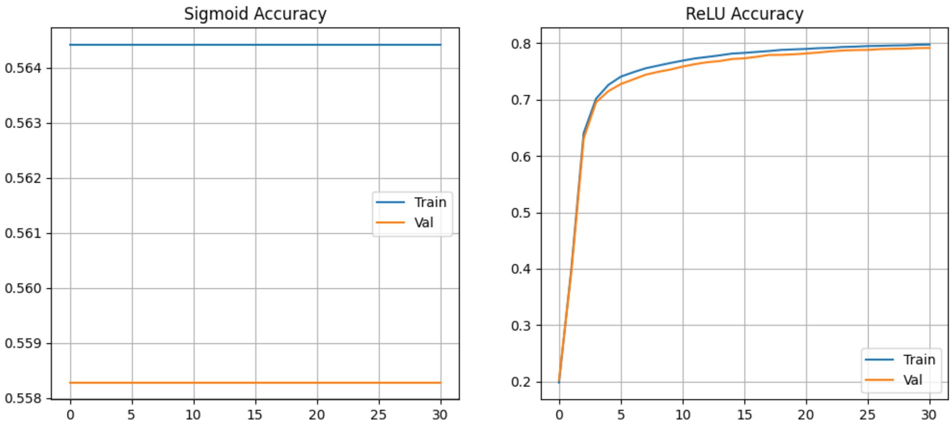


Figure 7: Gradient history for the hidden layer weights.

Gradient properties directly dictate the efficiency and stability of neural network optimization. During backpropagation, error signals are propagated from the output layer back to the input layer using the chain rule. Because gradients are continuously multiplied across successive layers, the derivative properties of the chosen activation function determine whether this signal is preserved, exploded, or destroyed.

Impact of Gradient Properties on Optimization and Gradient Flow

Gradient properties directly dictate the efficiency and stability of neural network optimization. During backpropagation, error signals are propagated from the output layer back to the input layer using the chain rule. Because gradients are continuously multiplied across successive layers, the derivative properties of the chosen activation function determine whether this signal is preserved, exploded, or destroyed.

- **Sigmoid and Vanishing Gradients:** The derivative of the Sigmoid function has a maximum value of 0.25. When these small, fractional gradients are multiplied across multiple hidden layers, the overall gradient magnitude shrinks exponentially. This chokes the *gradient flow*, causing the well-known "vanishing gradient problem." As a result, optimization stalls; the earlier layers receive almost zero update signal, preventing the network from learning complex representations. [add plot]

- **ReLU and Stable Gradient Flow:** In contrast, the ReLU activation function has a constant derivative of 1 for all positive pre-activations. This linear property allows gradients to flow backwards through the network without shrinking. Because the gradient flow remains strong and intact, the optimizer receives clear, non-decaying signals. This drastically accelerates convergence speed and ensures stable optimization across all layers. [add plot]

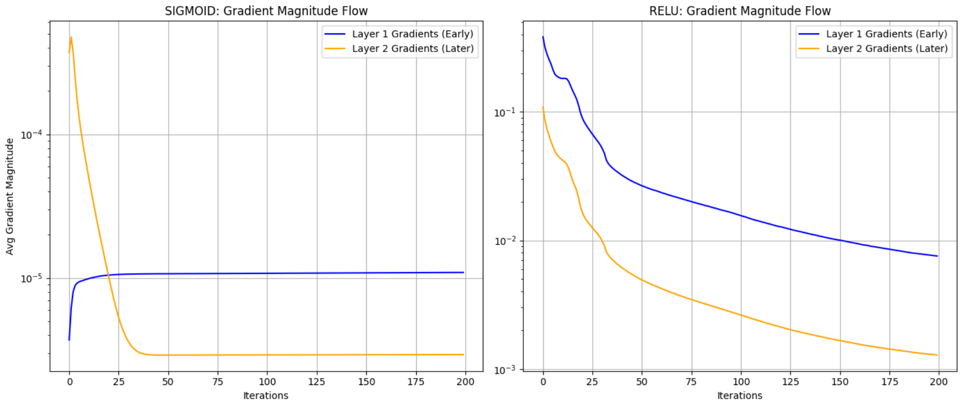


Figure 8: vanishing gradient problem with sigmoid activation.

3 Saliency Maps & Feature Attribution (Part C)

Derivation

The below equations show the step-by-step derivation of the gradient of the loss with respect to the input features, which is used to compute the saliency maps for feature attribution.

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z_2}\frac{\partial z_2}{\partial h_1}\frac{\partial h_1}{\partial z_1}\frac{\partial z_1}{\partial x},$$

$$\frac{\partial \mathcal{L}}{\partial x} = \mathbf{W}_1^\top \left((\mathbf{W}_2^\top (\hat{y} - y)) \odot \sigma'(z_1) \right)$$

where:

$$\frac{\partial \mathcal{L}}{\partial x} = \delta_1 \cdot W_1^T \tag{1}$$

Algorithm

The algorithm that computes analytically the gradient from given inputs become as follows:

1. **Inputs:** a sample vector and its target label. Initialize the weights and biases for the two layers.

2. **Forward pass:**

   - Compute the first layer pre-activation from the input and the first-layer weights and bias.
   - Apply the activation function to get the hidden representation.
   - Compute the logits using the second-layer weights and bias.
   - Apply softmax to obtain the predicted probabilities.

3. **Backward pass (sensitivities):**

   - Compute the output error as prediction minus target.

- Backpropagate this error to the hidden layer using the second-layer weights and the activation derivative.
- Backpropagate to the input using the first-layer weights to obtain the input gradient.

4. **Feature ranking:**
   - For each input feature, take the absolute value of its gradient as importance.
   - Sort features by importance from highest to lowest.
5. **Return:** the sorted list of feature indices.

To compute the saliency maps, I calculated the gradient of the loss with respect to the input features. This was done by performing backpropagation from the output layer back to the input layer. The saliency map highlights the importance of each input feature in contributing to the model's prediction. The gradient of the loss with respect to the input is calculated as:

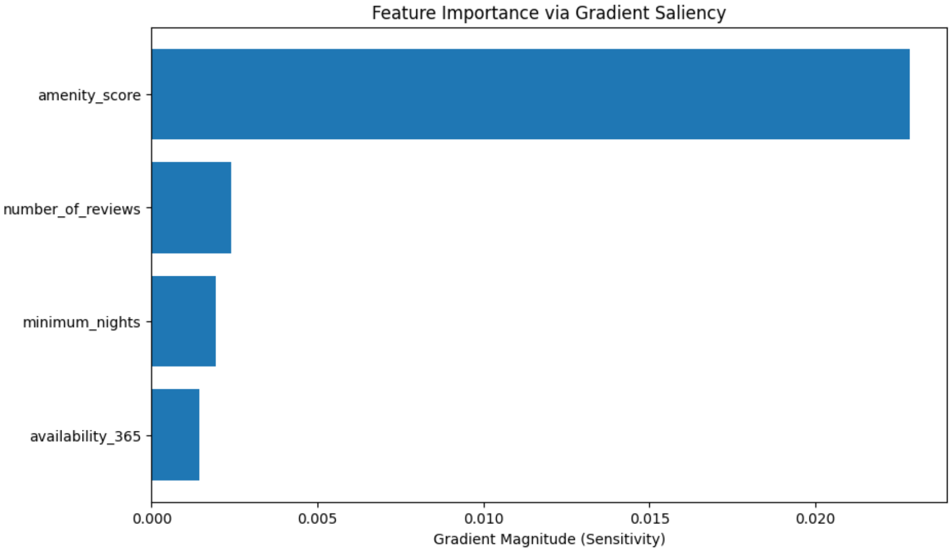


Figure 9: Saliency map for input features.

4 Generalization Analysis (Part D)

To analyze the generalization performance of the model, I evaluated it on a separate test set that was not used during training. I calculated the test accuracy and compared it with the training accuracy to identify any potential overfitting issues. If the training accuracy is significantly higher than the test accuracy, it may indicate that the model is overfitting to the training data and not generalizing well to unseen data. I also plotted the training and test accuracies over epochs to visualize the learning process and identify any trends in the model's performance over time.

The model achieves high accuracy on the training and validation sets but experiences severe drop in performance on the held-out test set. This massive generalization gap is an indicator of overfitting; the network memorized specific patterns in the training data rather than learning universally applicable rules.

- **Generalization Gap Analysis Final Test Accuracy: 0.0692**

  Training Accuracy: (Check your training plots, e.g. 0.80) Test Accuracy: 0.0692 Gap Analysis: The drop in accuracy is likely due to the model overfitting on amenity_score.

- **Evidence from EDA**: Initial data exploration revealed that amenity_score was suspiciously dominant and highly correlated with the price_class target.

- **Evidence from Feature Attribution**: Gradient-based saliency maps ($\frac{\partial \mathcal{L}}{\partial x}$) mathematically confirmed this bias. The gradient magnitude for amenity_score eclipsed all other features, proving the model heavily weighted this single input.

why it fails on test set? This shortcut worked well on the training data, where amenity_score was a strong predictor. However, in the test set, the distribution of amenity_score was different, and the model's over-reliance on this feature caused it to fail to generalize, leading to poor performance.

The only way resolve this issue is to remove the amenity_score feature from the training data, forcing the model to learn from other features and develop a more robust understanding of the underlying patterns in the data. This would help improve the generalization performance on the test set and reduce the risk of overfitting.

QUESTION 2 In this question, I have evaluated my understanding of backpropagation at a symbolic level, with particular emphasis on bias parameters that are shared across multiple layers.

4.1 Shared Bias Setting and Gradient Derivation

The forward pass of the network is:

$$h_1 = g(Wx + b), \tag{2}$$

$$\hat{y} = g_1(Uh_1 + b), \tag{3}$$

$$L = \ell(\hat{y}, y). \tag{4}$$

where:

- $x \in \mathbb{R}^d$ is the input feature vector,
- $y \in \mathbb{R}^k$ is the target vector,
- $W \in \mathbb{R}^{m \times d}$ and $U \in \mathbb{R}^{k \times m}$ are weight matrices,
- $b \in \mathbb{R}^m$ is a single shared bias vector used in both affine transformations,
- $g(\cdot)$ is an element-wise hidden-layer activation function,
- $g_1(\cdot)$ is an element-wise output activation function,
- $\ell(\cdot, \cdot)$ is a differentiable loss function mapping $(\hat{y}, y)$ to a scalar.

Shared Bias Gradient

Modify the backpropagation algorithm discussed in class/notes to compute the gradient vector

$$\frac{\partial L}{\partial b}$$

for the case when biases are shared. For simplicity, assume hidden and output dimensions are equal, i.e., $m = k$.

Your derivation must:

- apply the chain rule step by step, and clearly identify all computational paths through which the bias parameter b influences the loss;
- explicitly show how gradient contributions from multiple occurrences of the same parameter are combined.

Derivation

The gradient of the loss with respect to the shared bias b can be derived by applying the chain rule of calculus. The bias b influences the loss through both the hidden layer and the output layer. Therefore, we need to consider both paths when computing the gradient. The gradient can be expressed as:

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial h_1} \cdot \frac{\partial h_1}{\partial b} + \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial b}$$

Combining contributions for a shared bias. Because the same bias vector b is used in both the hidden layer and the output layer (with $m = k$), the loss depends on b through two distinct computational paths. The total gradient is the sum of the pathwise contributions:

$$\frac{\partial \mathcal{L}}{\partial b} = \left.\frac{\partial \mathcal{L}}{\partial b}\right|_{\text{via hidden}} + \left.\frac{\partial \mathcal{L}}{\partial b}\right|_{\text{via output}}.$$

Equivalently, if we denote the hidden-layer sensitivity by δ_{hidden} and the output-layer sensitivity by δ_{out}, then

$$\frac{\partial \mathcal{L}}{\partial b} = \delta_{\text{hidden}} + \delta_{\text{out}},$$

which explicitly shows how gradients from multiple occurrences of the shared parameter are combined by summation.

It is expected that sharing bias parameters affect stability and convergence speed due to parameter coupling.

- **stability** Gradient magnitude for b accumlates if both hidden layer share the same signs compared to weights.
- **convergence** As independant bias model has flexibility to shift activation threshold, but if $b1 = b2$ then we restrict the network capacity to learn optimal thresholds for both layers, which can slow down convergence and lead to suboptimal minima.

  The optimizer must find single compromising value for the shared bias that satisfies both layers which slows down convergence to optimal solution.