MGSC 673: Project
Multi-task Learning for Predicting House Prices and House Category

Lakshya Agarwal - 261149449

22nd April 2024

1 Introduction

This report presents a multi-task learning model for predicting house prices and categories using the House Prices - Advanced Regression Techniques Dataset from Kaggle. The objective is to build a feed-forward neural network model using PyTorch Lightning that can handle both regression (price prediction) and classification (category prediction) tasks simultaneously. The project involves data exploration, preprocessing, model building, experimentation with activation functions and optimizers, implementation of suitable loss functions, model evaluation, and hyperparameter tuning.

2 Data Preprocessing

The dataset was obtained from Kaggle. There are 1460 rows and 81 columns. The data preprocessing steps are implemented in the `preprocessing.py` file. The key preprocessing tasks are broken down into the following subsections.

2.1 Constructing House Category Variable

A new variable called `HouseCategory` is created based on the combination of 'House Style', 'Bldg Type', 'Year Built', and 'Year Remod/Add' features. This categorical variable is used as the target for the classification task in the multi-task learning model.

2.2 Exploratory Data Analysis (EDA)

Exploratory data analysis is performed to gain insights into the dataset.

- Target price distribution: After removing outliers, we can see that most of the observation lie in the range of 100K-200K

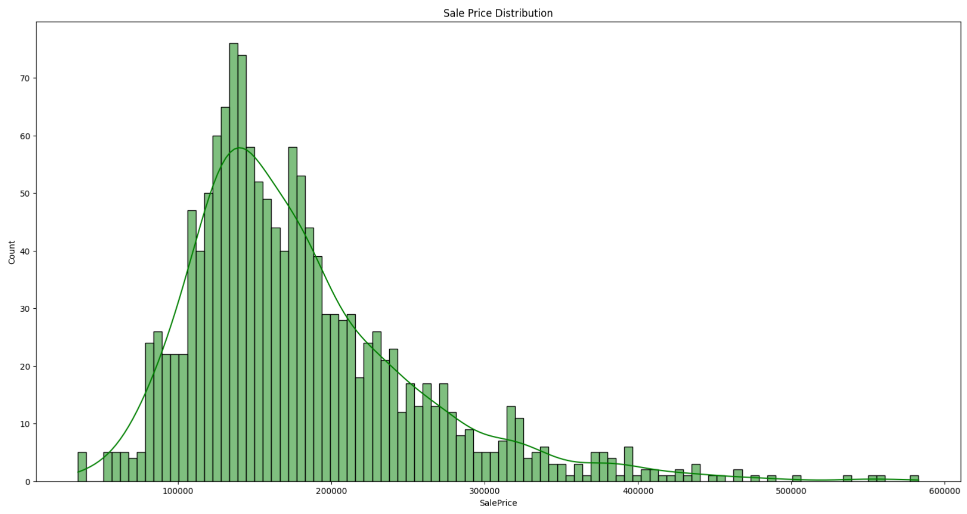

Figure 1: Sale Price Distribution

- Target category distribution: The target categories are unbalanced with class 14 and 12 having the highest representation. To mitigate this, we will calculate class weights and utilize them when calculating the categorical loss.

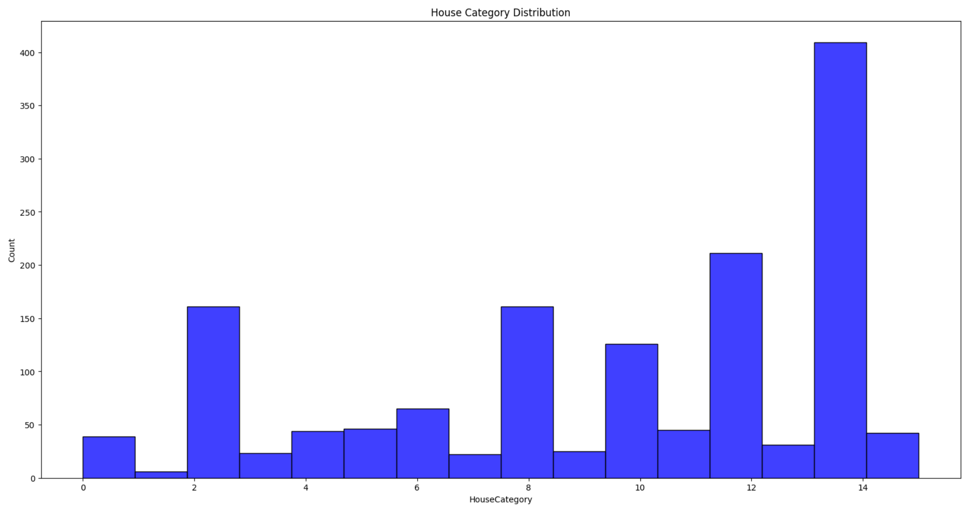

Figure 2: Sale Price Distribution

- Scatter plots: We see a linear and possibly exponential relationship for ‘GrLivArea‘ and ‘TotalBsmtSf‘ respectively, with ‘SalePrice‘

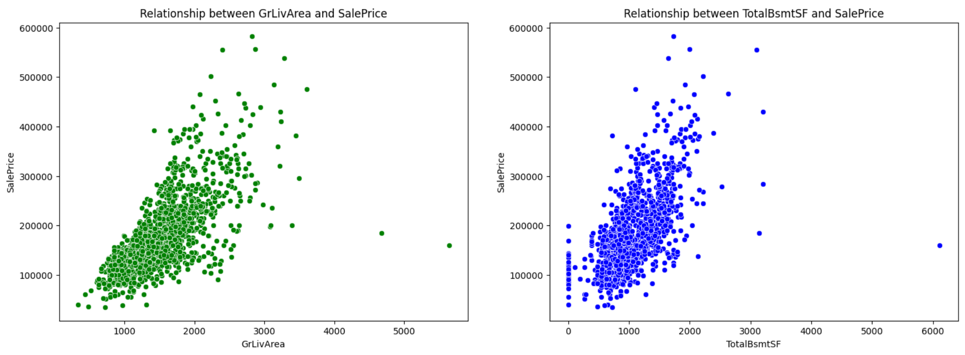

Figure 3: Scatterplot with SalePrice

2.3 Train-Validation Split

The preprocessed data is split into train and validation sets using the `split_data` function implemented in `preprocessing.py`, with a validation size of 30%. Stratified splitting is performed based on the ’House Category’ variable to ensure balanced representation of each category in both the train and validation sets.

2.4 Missing Value Imputation

Missing values in the dataset are handled using different imputation strategies for specific features. The imputation techniques used include:

- Filling NAs with None / 0: A lot of the missing values in the dataset actually correspond to an absence of a feature in a house. For such values, we replace the NA value with a None or 0 (depending on the column type) to signify the absence.

- Median imputation: Missing values for ‘LotFrontage‘ are imputed using the median value

2.5 Converting Numerical to Categorical

Some numerical features are converted to categorical features based on domain knowledge or data distribution. This conversion is performed to capture the categorical nature of certain variables and to leverage the power of embeddings in the neural network model.

In particular, the following variables are converted into categorical: `"MSSubClass"`, `"OverallCond"`, `"OverallQual"`, `"YrSold"`, `"MoSold"`

2.6 Scaling Numerical Variables

Numerical features are scaled using the `StandardScaler` from scikit-learn. Scaling ensures that all numerical features have zero mean and unit variance, which helps in the convergence of the optimization algorithm during model training.

2.7 Label Encoding Categorical Features

Categorical features are encoded using the `LabelEncoder` from scikit-learn. Label encoding assigns a unique integer value to each category in a categorical feature. This encoding is necessary to convert the categorical data into a format suitable for the neural network model.

2.8 Creating a PyTorch Dataset

The `HousingDataset` class in `dataset.py` is a custom PyTorch Dataset that handles loading and providing access to the preprocessed data. It takes care of both categorical and numerical features and returns them as tensors along with the corresponding price and category labels. A batch size of 32 samples was chosen.

3 Model Architecture

The `HousingNetwork` class in `network.py` defines the neural network architecture for the multi-task learning model.

The `HousingNetwork` is initialized with an instance of the `NetworkParameters` dataclass from `classes.py`, which holds all the necessary hyperparameters for the model, such as the number of layers, embedding dimensions, hidden dimensions, and output dimensions.

3.1 Layers

A visual representation of the general form of the neural network can be seen in Figure 4.

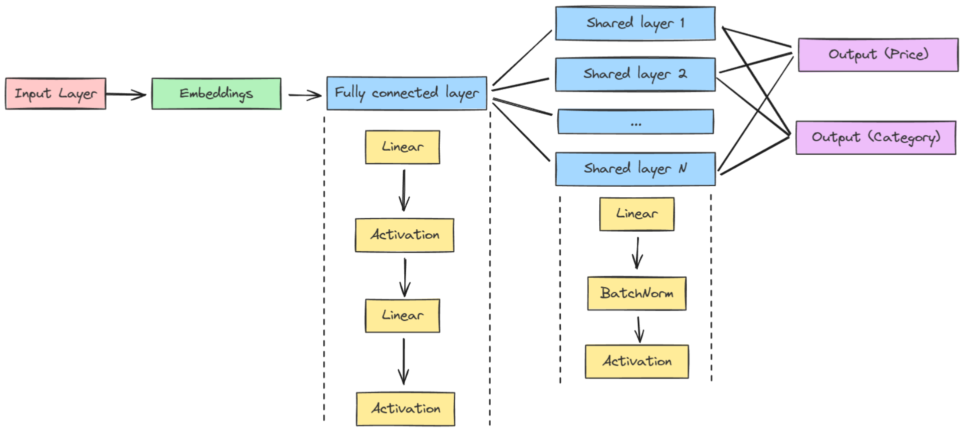

Figure 4: General representation of the neural network

3.1.1 Embedding Layers

The `HousingNetwork` class uses embedding layers to handle categorical features. For each categorical feature, an `nn.Embedding` layer is created. The dimensions of each embedding layer is of the form $(num_embd, embd_dim)$, where the first parameter specifies the number of unique categories.

The embedding layers map each unique category to a dense vector representation of size `embedding_dim`. This allows the model to learn meaningful representations for the categorical features, capturing their semantic relationships. The embeddings are initialized using the Kaiming-uniform distribution.

3.1.2 Shared Layers

After the embedding layers, the `HousingNetwork` class defines shared hidden layers that process the concatenated embeddings of categorical features and the numerical features.

The shared layers consist of a series of linear layers (`nn.Linear`) followed by an activation function. The activation function used in the shared layers is configurable and defaults to `nn.LeakyReLU`. The choice of activation function, and the number of shared layers is determined by the parameters in the `NetworkParameters` dataclass.

The input to the first shared layer is the concatenation of the flattened embeddings of categorical features and the numerical features. The output dimension of each shared layer is specified by the `hidden_dim` parameter. After the first shared layer, we add a batch normalization layer (`nn.BatchNorm1d`) to normalize the weights for each batch.

The shared layers allow the model to learn common representations and patterns across both the price prediction and category classification tasks.

3.1.3 Output Layers

Since we are building a multi-task learning model, the `HousingNetwork` class has separate output layers for price prediction (regression) and category classification.

For price prediction, a linear layer (`nn.Linear`) is used to map the output of the last shared layer to a single value representing the predicted price. The output dimension of this layer is 1.

For category classification, another linear layer (`nn.Linear`) is used to map the output of the last shared layer to a vector of size `output_dim_category`, representing the logits for each category.

The output layers allow the model to specialize in the specific tasks of price prediction and category classification while leveraging the shared representations learned by the shared layers.

3.2 Loss function

The model uses different loss functions for the price prediction and category classification tasks.

For price prediction, the Mean Squared Error (MSE) loss is used, which is implemented using `nn.MSELoss()`. MSE loss calculates the average squared difference between the predicted prices and the true prices, penalizing larger errors more heavily.

For category classification, the Cross-Entropy loss is used, which is implemented using `nn.CrossEntropyLoss()`. Cross-Entropy loss measures the dissimilarity between the predicted class probabilities and the true class labels, and encourages the model to assign high probabilities to the correct classes. To handle imbalanced data in the category classification task, we use class weights, which gives higher importance to underrepresented classes.

To combine the MSE loss and Cross-Entropy loss, we use an uncertainty weighting scheme [1]. The combined loss is calculated as:

$$L = (L_R \cdot \exp(-\eta_1) + \eta_1) + (L_C \cdot \exp(-\eta_2) + \eta_2)$$

where L_R and L_C are the MSE and CrossEntropy loss values respectively. η_1 and η_2 are learnable parameters that control the relative importance of each task's loss. The $\exp(-\eta)$ term acts as a weighting factor, allowing the model to adaptively balance the contributions of each task during training. The η values are initialized as a tensor in the constructor of the `HousingModel` class and are updated during the optimization process.

The uncertainty weighting scheme helps the model to dynamically adjust the emphasis on each task based on the model's confidence in its predictions. This approach is particularly useful in multi-task learning scenarios where the tasks may have different scales or difficulties.

During training, the combined loss is used to update the model's parameters using backpropagation. The individual MSE loss and Cross-Entropy loss values are also logged for monitoring and analysis purposes.

3.3 Activation Function

By default, the activation function of choice is the `LeakyReLU`, which is defined as:

$$\text{LeakyReLU}(x) = \begin{cases} x, & \text{if } x \geq 0 \\ \alpha x, & \text{otherwise} \end{cases}$$

where α is a small negative slope (default value is 0.01) that allows for a small gradient when the input is negative. LeakyReLU helps alleviate the "dying ReLU" problem, where neurons with negative inputs become inactive and stop contributing to the learning process.

The choice of activation function can have a significant impact on the model's performance and convergence. The `HousingNetwork` class is flexible and can accommodate other activation functions as well such as `ReLU`, `ELU`, or `Tanh`, depending on the specific requirements of the problem.

The selected activation function is applied element-wise to the output of each hidden layer, introducing non-linearity and enabling the model to learn complex patterns and relationships in the data.

4 Experimental Setup

With the model architecture, loss function, and activation functions in place, we turn to training the model, validating it, and performing hyperparameter optimization. PyTorch Lightning and TensorBoard were used to effectively manage the training, evaluation, and logging across experiments.

4.1 Training and Validation Steps

The `HousingModel` class in `learner.py` defines the `training_step` and `validation_step` methods, which handle the forward pass, loss calculation, and logging of relevant metrics during training and validation, respectively.

In the `training_step`, the model takes the categorical features, numerical features, and corresponding price and category labels as input. It performs the forward pass and computes the combined loss, price loss (MSE), and category loss (Cross-Entropy). The losses are logged to TensorBoard and the console for monitoring and visualization purposes. Additionally, the values of the learnable parameters `eta` are logged as well.

In the `validation_step`, the model switches to inference mode, performs the forward pass and computes the losses. The validation losses are logged alongside additional metrics such as accuracy, precision, recall, and F1 score.

4.2 Optimizer

The `configure_optimizers` method in the `HousingModel` class sets up the optimizer for training the model. For this experiment, the `AdamW` optimizer is used with a specified learning rate (`self.lr`) and weight decay of 1×10^{-2}. The learning rate is a hyperparameter that can be tuned.

Other optimizers, namely `RMSProp` and `SGD` were also used, but model performance suffered in initial experiments, so `AdamW` was chosen.

4.3 Initial Value of η

The η parameter is a learnable tensor that balances the contributions of the price loss and category loss in the combined loss function. Since the MSE of the price prediction task is of the order 3.72×10^{10}, while the CrossEntropy loss of the category prediction task is of the order 0.2269×10^{1}, the loss weight parameter η is initialized as:

$$\eta = \begin{bmatrix} 20 & 0 \end{bmatrix}$$

Here, $\eta_1 = 20$ represents the scaling factor for the MSE loss relative to the CrossEntropy loss. After experimenting with different values for η_1, 20 was chosen.

4.4 Hyperparameters

The hyperparameters available to be customized are:

- Number of layers (`num_layers`): An integer value between 2 and 6.
- Embedding dimension (`embedding_dim`): A value selected from 2^4 to 2^9
- Hidden dimension (`hidden_dim`): A value selected from 2^5 to 2^10
- Learning rate (`lr`): A float value sampled from a log-uniform distribution between 1×10^{-5} and 1×10^{-2}.

These hyperparameters are used to create an instance of the `NetworkParameters` dataclass, which is then passed to the `HousingNetwork` model.

4.5 Hyperparameter Tuning using Optuna

Optuna is used to tune the hyperparameters of the neural network. The search space for hyperparameters is defined using Optuna's suggest methods, as mentioned in subsection 4.4.

The `HousingModel` is then instantiated with the `HousingNetwork` model, initial `eta` values, learning rate, and class weights.

Finally, the PyTorch Lightning `Trainer` is set up with the specified maximum number of epochs (100), early stopping callback, Optuna callback [2] for pruning, and the configured logger.

The objective function returns the final validation loss, which Optuna aims to minimize during the hyperparameter optimization process.

5 Results & Discussion

The model's performance is evaluated based on the logged metrics during training and validation, such as the training and validation losses for both price prediction and category classification tasks, as well as accuracy, precision, recall, and F1 score for the category prediction task.

The effectiveness of the multi-task learning approach can be assessed by comparing the performance of the joint model with separate models trained for each task individually. The impact of different hyperparameter configurations obtained through Optuna can also be analyzed to identify the best-performing model.

A total of 15 trials were run using Optuna, with each trial running for a maximum of 100 epochs. The list of hyperparameters used for each trial can be seen in Table 3

Among the best trials, the top three (0, 11, 12) were chosen for further validation. The training and validation loss curves can be seen in Figure 5

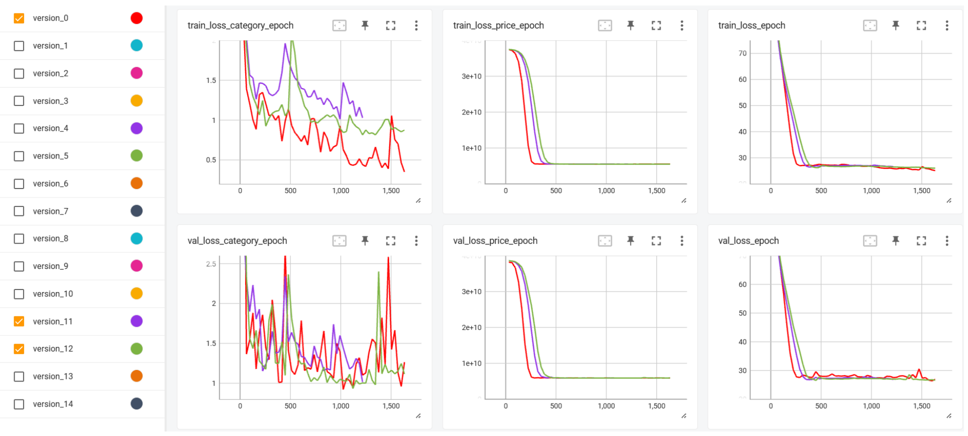

Figure 5: Loss curves for best trials

After running these models on the validation dataset, the task-specific performance is presented in Table 1. Please refer to Appendix B for the detailed image of loss curves across trials.

Trial #	lr	num_layers	embedding_dim	hidden_dim	RMSE	Accuracy
0	8.12×10^{-3}	3	16	64	76766.60	72.54%
11	9.12×10^{-3}	6	64	32	76777.61	56.06%
12	8.05×10^{-3}	5	64	32	76952.62	58.58%

Table 1: Best trials

For the best model, the validation metrics reported are in Table 2

Metric	Validation Dataset
Accuracy	72.54%
F1 Score	72.19%
Precision	77.65%
Recall	72.54%
Loss	26.95
Category Loss	1.26
Price Loss	5.89×10^{9}

Table 2: Model performance

6 Conclusion

In conclusion, this project demonstrates the implementation of a multi-task learning model for predicting house prices and categories using PyTorch Lightning.

The model architecture incorporates embedding layers for categorical features, shared hidden layers, and separate output layers for each task. The training process is managed by the PyTorch Lightning Trainer, which handles logging, checkpointing, and early stopping.

The modular structure of the codebase, with separate files for data preprocessing, model definition, training logic, and hyperparameter tuning, promotes code reusability and maintainability. The use of PyTorch Lightning simplifies the training process and provides a standardized way to organize the code.

Further improvements can be explored, such as experimenting with different model architectures (CNNs, RNNs, Transformers), incorporating additional features, and applying more advanced techniques for handling imbalanced data. The model's performance can be further validated using cross-validation or by evaluating it on a separate test set.

Appendix A Hyperparamter Tuning

Trial #	lr	num_layers	embedding_dim	hidden_dim
0	8.12×10^{-3}	3	16	64
1	1.26×10^{-4}	6	256	128
2	2.66×10^{-3}	4	16	512
3	3.63×10^{-4}	3	32	64
4	9.46×10^{-5}	4	32	64
5	3.95×10^{-5}	3	32	256
6	1.55×10^{-5}	2	16	128
7	3.03×10^{-4}	3	128	512
8	1.85×10^{-3}	4	256	128
9	4.15×10^{-4}	3	128	512
10	7.33×10^{-3}	6	64	32
11	9.12×10^{-3}	6	64	32
12	8.05×10^{-3}	5	64	32
13	2.28×10^{-3}	5	64	32
14	8.24×10^{-3}	5	64	32

Table 3: Hyperparameter tuning trials

Appendix B Loss curves

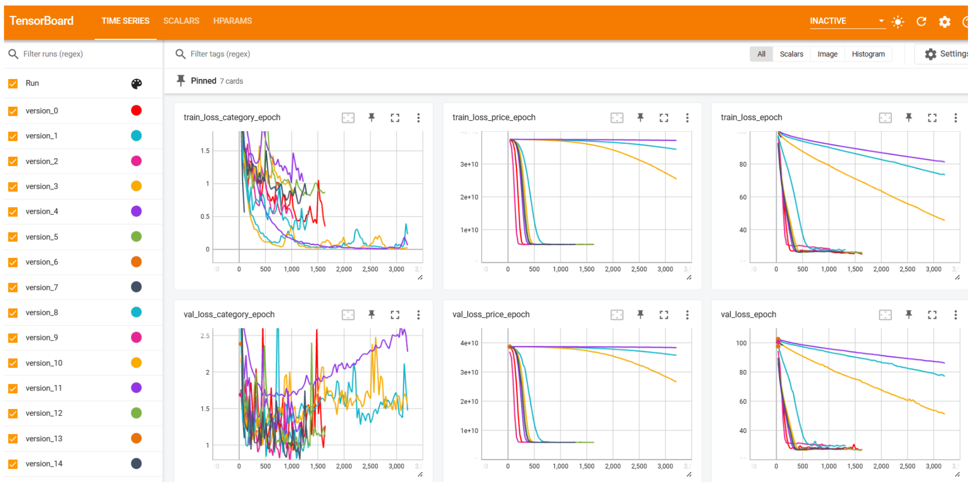

Figure 6: Loss curves for all trials

7 References

[1] Alex Kendall, Yarin Gal, and Roberto Cipolla. "Multi-Task Learning Using Uncertainty to Weigh Losses for Scene Geometry and Semantics". In: *CoRR* abs/1705.07115 (2017). arXiv: 1705.07115. URL: http://arxiv.org/abs/1705.07115.

[2] Ahmet Zamanis. *ValueError: Expected a parent with PyTorchLightningPruningCallback*. Issue 4689, Optuna GitHub repository. Apr. 2024.

[3] *Introduction to Lightning AI*. Lightning AI. 2023. URL: https://lightning.ai/docs/pytorch/stable/starter/introduction.html.

[4] Yu Zhang and Qiang Yang. "An overview of multi-task learning". In: *National Science Review* 5.1 (Sept. 2017), pp. 30–43. ISSN: 2095-5138. DOI: 10.1093/nsr/nwx105. eprint: https://academic.oup.com/nsr/article-pdf/5/1/30/31567358/nwx105.pdf. URL: https://doi.org/10.1093/nsr/nwx105.