Multi-task Learning for Predicting House Prices and House Category

Krishan Gupta | 261151116

Introduction

The objective of this project is to construct a multi-task learning model capable of simultaneously predicting house prices and categorizing house types. This dual-task model is a sophisticated approach in machine learning that leverages a shared architecture to perform both regression (predicting house prices) and classification (categorizing house types based on style, building type, and remodeling history). The project utilizes the House Prices - Advanced Regression Techniques Dataset, enhancing it with a newly engineered feature, 'House Category', derived from multiple existing variables.

Utilizing PyTorch Lightning, a powerful framework that simplifies complex model training, the project employs its advanced features to manage the intricacies of multi-task learning efficiently. These features include automated logging, a robust callback system, and a flexible Trainer API that aids in handling the computational challenges of training a multi-task model. Additionally, the integration of Optuna for hyperparameter tuning ensures that the model achieves optimal performance by fine-tuning its parameters based on the training outcomes. This report details the methodology, experiments, and results of building and evaluating a model that adeptly handles the complexities of predicting multiple outputs from a single, cohesive neural network architecture.

Dataset and Preprocessing

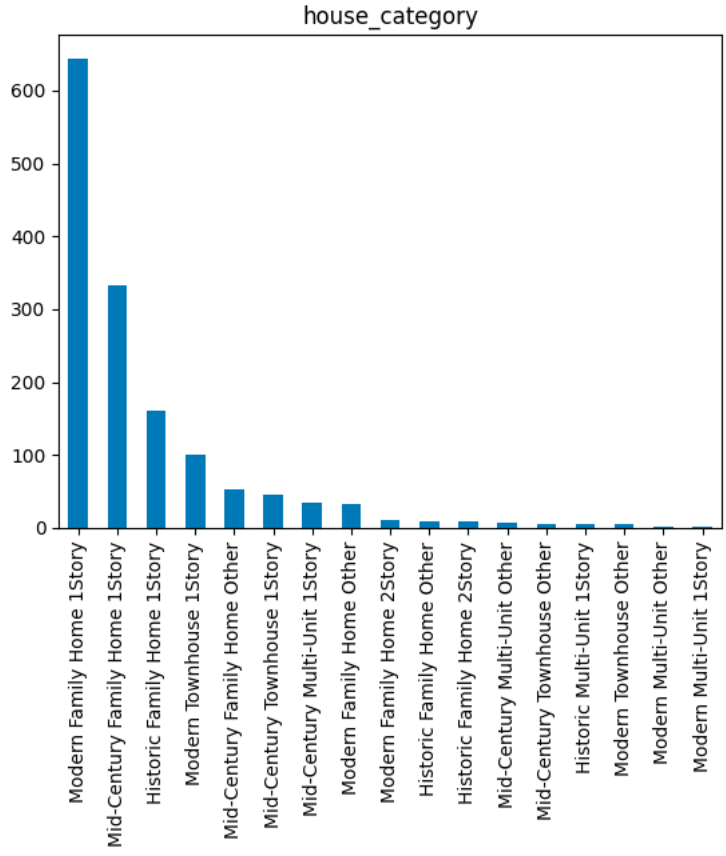
The first step in the analysis involved loading the data and performing initial explorations to understand its structure and content. The dataset initially contained several features, including both numerical and categorical data.

Handling Missing Values : The exploration phase revealed that some columns had a significant amount of missing data. Columns with more than 40% missing values were dropped, as imputing these could introduce bias. For other columns with fewer missing values, numerical columns were imputed with the mean, and categorical columns were imputed with the mode, ensuring that no data point was left out due to missing values.

LotFrontage	17.739726
Alley	93.767123
MasVnrType	0.547945
MasVnrArea	0.547945
BsmtQual	2.534247
BsmtCond	2.534247
BsmtExposure	2.602740
BsmtFinType1	2.534247
BsmtFinType2	2.602740
Electrical	0.068493
FireplaceQu	47.260274
GarageType	5.547945
GarageYrBlt	5.547945
GarageFinish	5.547945
GarageQual	5.547945
GarageCond	5.547945
PoolQC	99.520548
Fence	80.753425
MiscFeature	96.301370

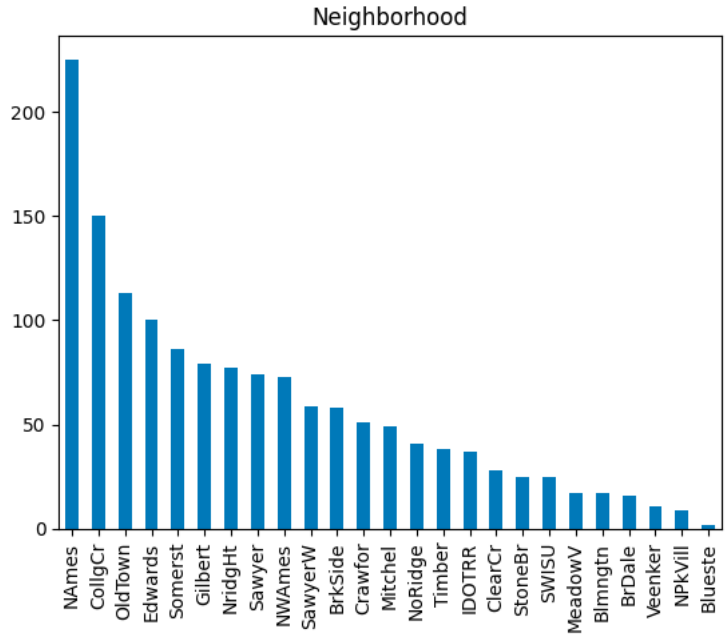
Missing Values in %

Feature Engineering : Significant efforts were made in feature engineering, particularly with the creation of a new feature named 'house_category.' This feature was engineered from 'House Style', 'Bldg Type', 'Year Built', and 'Year Remod/Add' to aid in classification tasks. This categorical feature was divided into categories based on the house era, type, and style, defined by the provided features and using specific conditions to categorize them into Modern, Historic, and Mid-Century eras.

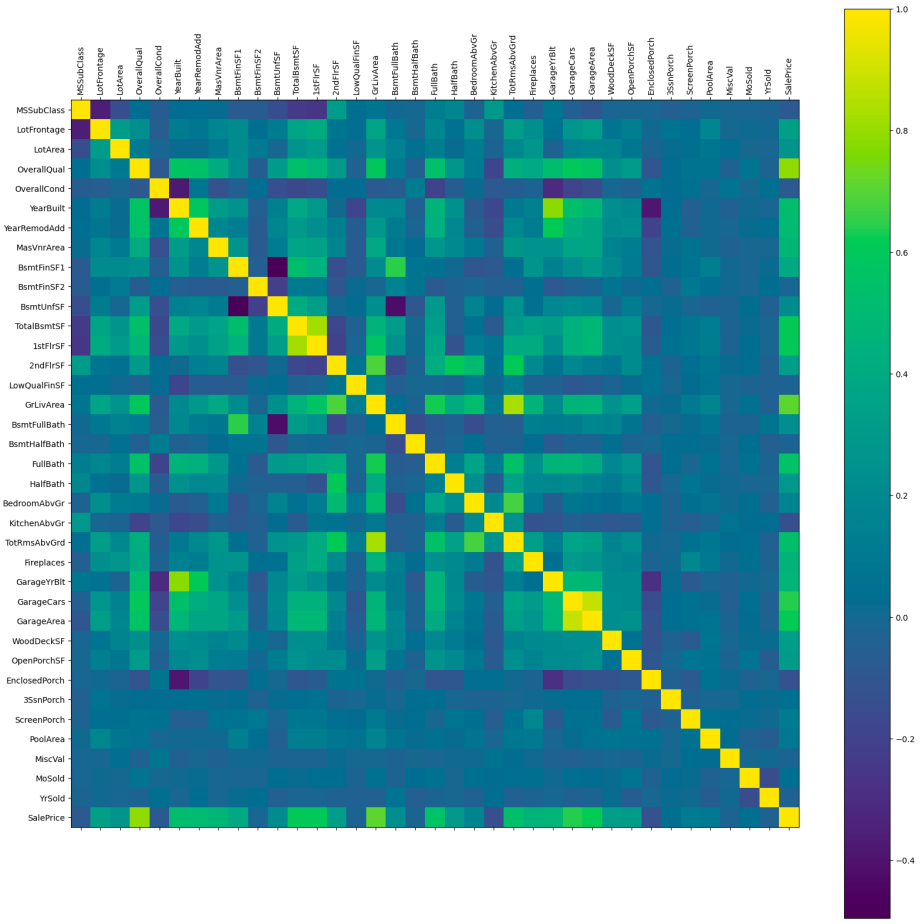


House Categories Distribution(17 Classes)

Exploring data distributions and correlations using plots like value counts bar charts and a correlation matrix heatmap. This helped identify important features and relationships.



Neighbourhood Distribution



Correlation Matrix

One-hot encoding categorical variables in the input features (X) and converting all features to float64.

Splitting data into train/validation/test sets (60/20/20) using sklearn's train_test_split.

Creating PyTorch TensorDatasets and DataLoaders for efficient batch processing during training and evaluation.

Multi-task Model Building with PyTorch Lightning

I built a feed-forward neural network model using PyTorch Lightning to predict both house prices (regression) and category (classification). Key aspects:

- Shared bottom layers: 2 hidden layers with ReLU activation and BatchNorm. The hidden layer sizes were tuned using Optuna.
- Task-specific output layers: single neuron for price regression, num_classes neurons for category classification

```
# Model architecture
self.hidden_layer1 = nn.Linear(input_size, hidden_size1)
self.batch_norm1 = nn.BatchNorm1d(hidden_size1)
self.hidden_layer2 = nn.Linear(hidden_size1, hidden_size2)
self.batch_norm2 = nn.BatchNorm1d(hidden_size2)
self.price_layer = nn.Linear(hidden_size2, 1)
self.category_layer = nn.Linear(hidden_size2, num_classes)
```

Model Architecture

- Proper weight initialization: Kaiming initialization for ReLU layers, Xavier for output layers

```
# Kaiming (He) initialization for layers with ReLU activation
nn.init.kaiming_normal_(self.hidden_layer1.weight, mode='fan_out', nonlinearity='relu')
nn.init.kaiming_normal_(self.hidden_layer2.weight, mode='fan_out', nonlinearity='relu')
# For the output layers, you might use a simpler initialization since they might not be followed by ReLU
nn.init.xavier_normal_(self.price_layer.weight)
nn.init.xavier_normal_(self.category_layer.weight)
```

Weight Initialisation

- Logging of training and validation metrics using PyTorch Lightning's automatic logging

The model architecture was defined in the HousingModel class, a subclass of pl.LightningModule. The forward method defined the forward pass, while training_step, validation_step and test_step computed losses and metrics for each batch during the respective phases.

Activation Functions and Optimizers

I experimented with ReLU activation (worked well) and tried Adam, SGD and RMSprop optimizers. Adam gave the best results.

Loss Functions and Metrics

- Used nn.MSELoss for price regression and nn.CrossEntropyLoss for category classification.
- Combined them into a weighted sum for the overall multi-task loss. The loss weights were tuned using Optuna.
- Logged and evaluated the model using:
    - RMSE for price
    - Accuracy, Precision, Recall, F1 Score for category
- Used PyTorch Lightning's validation_step and test_step to automatically compute and log metrics.

Hyperparameter Tuning with Optuna

Performed hyperparameter optimization using Optuna with PyTorch Lightning. Tuned:

- Hidden layer sizes: suggested int values for hidden_size1 and hidden_size2
- Learning rate: suggested log-uniform values for lr
- Optimizer type: categorical suggestion of 'adam', 'sgd', 'rmsprop'
- Loss function weights: suggested uniform values for w_loss, with w_acc = 1 - w_loss

The objective function defined the Optuna optimization process:

- Instantiate the HousingModel with hyperparameters suggested by Optuna
- Create a PyTorch Lightning Trainer with early stopping callback
- Fit the model and return the validation loss as the objective to minimize

Optuna searched the space of hyperparameters to find the configuration that minimized validation loss. The best configuration was:

- **hidden_size1: 152**
- **hidden_size2: 495**
- **lr: 0.09**
- **optimizer_type: 'adam'**
- **w_loss: 0.052**

```
Number of finished trials:  100
Best trial:
  Value:  69980248.0
  Params:
    hidden_size1: 152
    hidden_size2: 495
    lr: 0.09933770172211563
    optimizer_type: adam
    w_loss: 0.0528856173253155
```

Best Hyperparameters after 100 Optuna iterations

Final Model Evaluation

Model performance was assessed using various metrics:

- Mean Squared Error (MSE) for regression output (SalePrice).
- Accuracy, precision, recall, and F1 score for the classification output (house_category).

The final model was trained with the best hyperparameters found by Optuna.

On the Validation set, it achieved:

- Price RMSE of $ 40381.01
- Category Accuracy of 52.14%

On the Test set, it achieved:

- **Price RMSE of $ 45535.71** (House Mean price is $180921)
- **Category Accuracy of 45.89%** (among 17 Classes)

Test metric	DataLoader 0
test_accuracy	0.45890411734580994
test_category_loss	246.90370178222656
test_f1	0.3783092200756073
test_loss	109646928.0
test_precision	0.5660455226898193
test_price_loss	2073500672.0
test_recall	0.45890411734580994

```
RMSE Price Test: 45535.71
Accuracy: 45.89
```

Test Set Metrics

This demonstrates the capability of multi-task learning to leverage shared representations to perform well on multiple related tasks. PyTorch Lightning enabled productive experimentation, while Optuna boosted final performance via automated hyperparameter optimization.

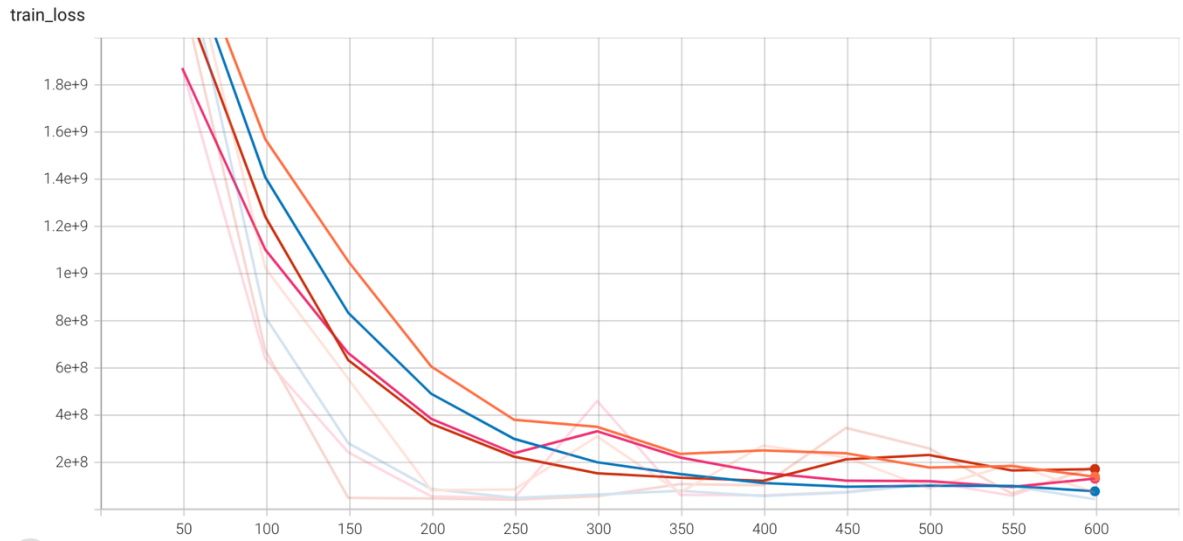
Advanced PyTorch Lightning Features

The Project utilized advanced PyTorch Lightning features to streamline the model training process and enhance the organization and scalability of the code. Here's a breakdown of how these features are employed:

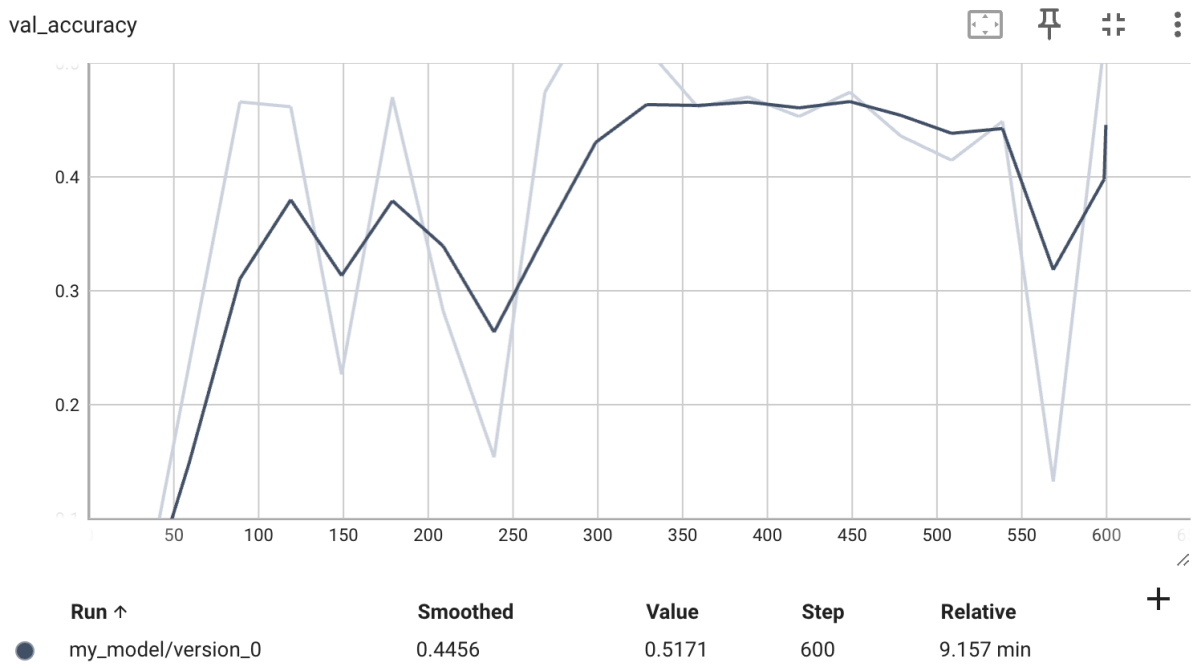
1. LightningModule for Clean Code Organization
The project uses the **LightningModule** class extensively, which is a core component of PyTorch Lightning. This class allows for the organization of code into logical sections, separating the model architecture, data handling, training step, validation step, and configuration of optimizers into neat, manageable methods. Here's how it's used:

- **Model Architecture**: The **__init__** method defines the model layers and initialization.
- **Forward Pass**: The **forward** method handles the forward computation—defining how data passes through the network.
- **Training Step**: The **training_step** method defines how the model should process data and calculate loss during training.
- **Validation Step**: The **validation_step** is used to evaluate the model on the validation set.
- **Test Step**: The **test_step** method assesses model performance on the test set.
- **Optimizer Configuration**: The **configure_optimizers** method sets up the optimizer based on the hyperparameters chosen during tuning.

2. Automatic Logging with TensorBoard
TensorBoard is integrated via PyTorch Lightning's logging capabilities. In the project, a **TensorBoardLogger** is instantiated, which automatically logs training and validation metrics like loss, accuracy, precision, recall, and F1 score. This integration simplifies the monitoring of model training progress and performance over epochs, directly storing logs in a structured format compatible with TensorBoard.



Train Loss logged using Tensorboard



Run ↑	Smoothed	Value	Step	Relative
my_model/version_0	0.4456	0.5171	600	9.157 min

Accuracy on Validation set logged using Tensorboard

3. Early Stopping Callback to Prevent Overfitting

The project employs an **EarlyStopping** callback, a feature of PyTorch Lightning that helps prevent overfitting. This callback monitors a specified metric, in this case, 'val_loss', and stops the training process if the metric does not improve for a defined number of epochs (**patience=10**). This ensures that the model doesn't continue to learn noise from the training data once it has ceased making genuine improvements in validation loss.

4. Trainer for Easy Training on GPUs

The **Trainer** class in PyTorch Lightning abstracts away much of the boilerplate training logic. In the script, it's used to manage the training process, including running the training loops, validation, and testing phases. The **Trainer** handles device placement (GPUs if available) and enables easy scaling of training to more complex hardware setups without changes to the model code. Although the GPU-specific settings are not explicitly set in the provided script, PyTorch Lightning's Trainer is designed to automatically leverage GPUs if they are available and configured appropriately.

Conclusion

Key learnings included the importance of careful data preprocessing, the effectiveness of shared representations in multi-task learning, and the power of PyTorch Lightning and Optuna in streamlining the deep learning workflow. Going forward, next steps could include exploring more advanced architectures like CNNs or Transformers, and deploying the model to a production environment.

The use of PyTorch Lightning in the project clearly demonstrates an advanced, organized approach to model development and training. It not only enhances code

readability and maintenance but also leverages modern best practices in machine learning workflows, such as callbacks for early stopping, automatic logging for monitoring, and abstraction layers that allow data scientists to focus more on modeling and less on the boilerplate of training loops.

References

torchMTL: A simple multi-task learning module for PyTorch: https://www.reddit.com/r/pytorch/comments/kgeb73/torchmtl_a_simple_multitask_learning_module_for/

How to tune Pytorch Lightning hyperparameters: https://towardsdatascience.com/how-to-tune-pytorch-lightning-hyperparameters-80089a281646

pytorch lightning hyperparameter tuning: https://www.youtube.com/watch?v=Y9mlwwRsid8

How to do multi-task training?: https://discuss.pytorch.org/t/how-to-do-multi-task-training/14879

TUTORIAL 2: ACTIVATION FUNCTIONS: https://lightning.ai/docs/pytorch/stable/notebooks/course_UvA-DL/02-activation-functions.html

Feasibility of multi-task training in lightning with dynamic model size: https://github.com/Lightning-AI/pytorch-lightning/issues/1502