

Used Car Sales:

*Deep Learning vs. Tree Based Models on Tabular Data*

MSCA 31009 Machine Learning: Dr. Arnab Bose

Benjamin Cox



# Problem Statement

---

## Deep Learning vs. Boosted Tree Models on Tabular Data

Personal Case: For years, my career existed in finance where regulations made applying deep learning models a large corporate risk and highly avoided. In later roles focused on researching explainability/interpretability in ML, our teams often advocated against DL when it was not absolutely necessary. Leading tree-based methods could outperform on tabular data, provided more means for interpretability (less black-box), and much more user friendly. In this analysis, I plan to test whether what is currently available in open-source packages whether Deep Learning models can outperform leading 'machine learning' methods, can provide similarly useful interpretability/explainability tools, and whether we can replicate the double descent phenomenon in tabular methods.

Business Case: For NON-Financial institutions, the luxury of testing the latest and greatest deep learning methods to increase revenue and cut OPEX exists without the same regulatory risk, at the same time many of these companies are extremely overwhelmed in the scale of their data and are looking to drive value from it. We aim to test if a high-capacity Deep Learning model provide meaningful uptick in performance on large scale (10GB) data sets of used car sales in the US. Should this be effective, we enable a meaningful new data driven approach to used car sales strategy for dealerships.



# Key Tests

Hypotheses In Focus



## Performance

Can large deep learning models outperform SOTA tree-based methods?



## Interpretability

Can large deep learning models provide sufficient explanatory/interpretability tools to be trustworthy?

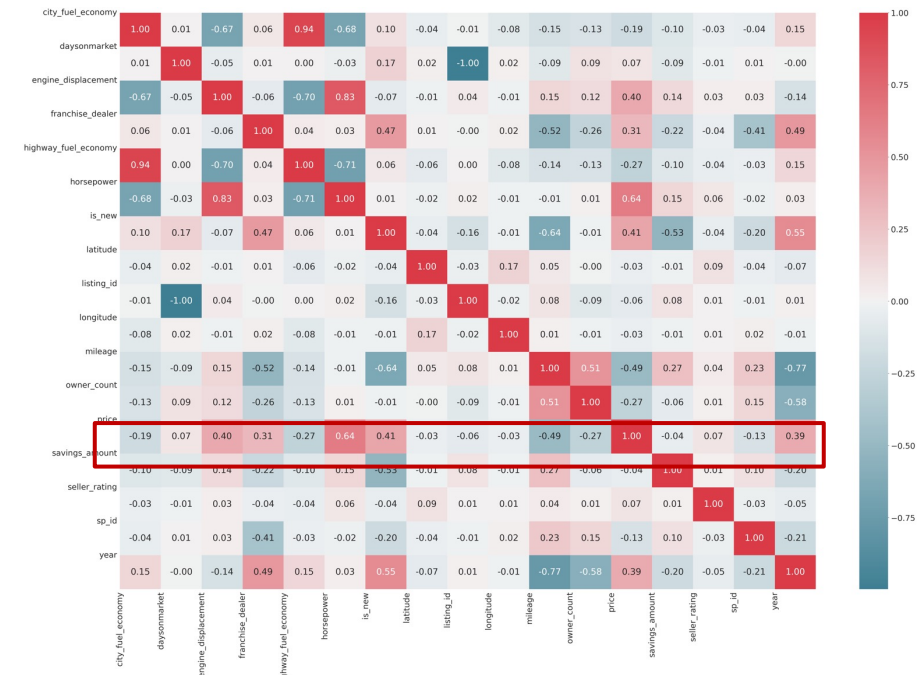
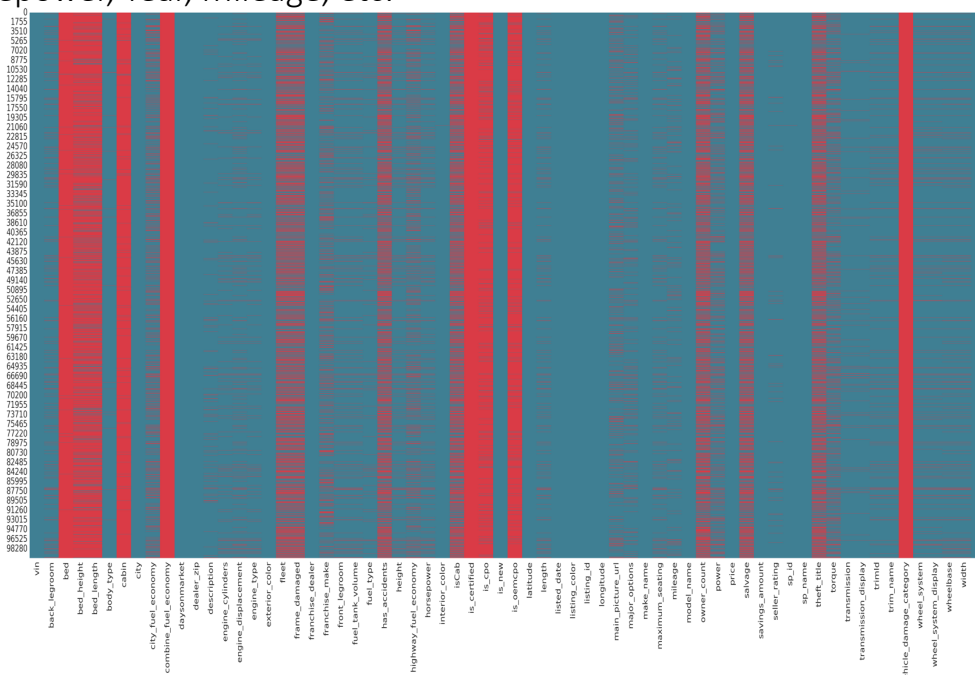


## Double Descent

Can we replicate the double-descent phenomenon given our data set size and computational bandwidth?

# Exploratory Data Analysis

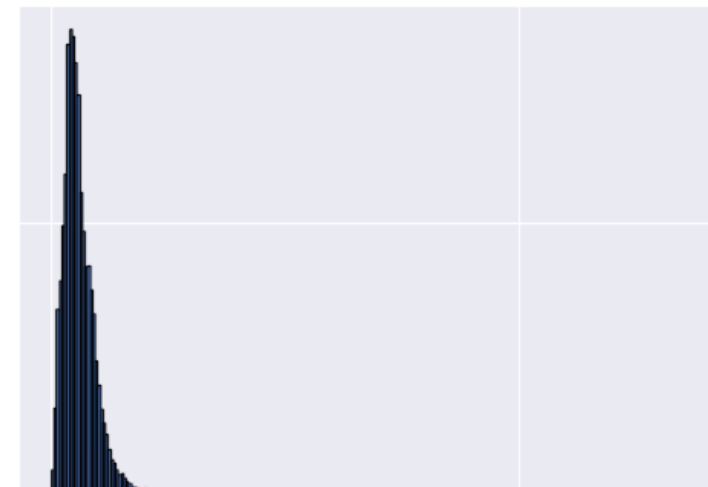
- The data set for this analysis is Used Car Sales Data from Kaggle: (<https://www.kaggle.com/datasets/ananymital/us-used-cars-dataset>)
- The data set choice was in order to find the largest possible quality tabular data set available (9.85 GB) to test whether deep learning models could outperform leading tabular boosted tree (XGBoost methods)
- There has been significant debate on whether deep learning with enough data and enough parameters can consistently outperform “standard Machine Learning” on prediction tasks, we aim to test that in this analysis.
- The dataset contains 66 features across 3 Million rows. These features are very standard expected used car data: Year, Make, Model, etc. and our target variable Price.
- A very quick heatmap can show us our data set is missing a large amount of data, with some columns being completely empty. We address this in preprocessing.
- Finally, a correlation heatmap is run to get some early insight on feature correlation. We can see some obvious correlations with Price early such as Horsepower, Year, Mileage, etc.



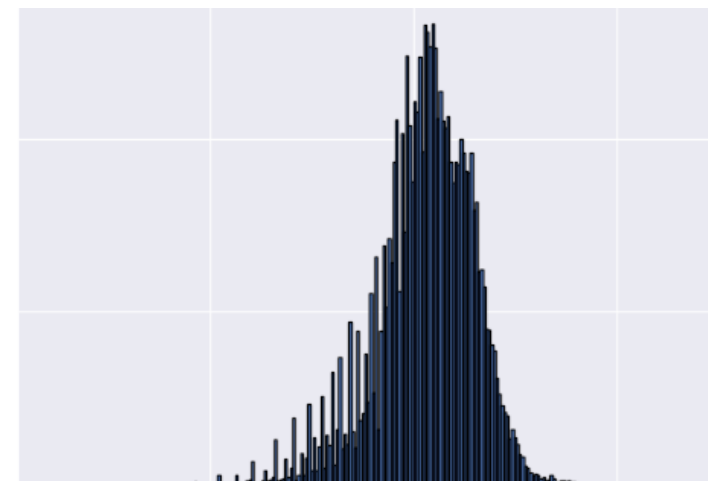
# EDA: Preprocessing

- As shown in data population heatmaps, there were a variety of data quality issues:
  - 100% Empty columns
  - Columns missing over 75% of population
  - Miscategorizations across data types (Numerical, Categorical, Boolean etc)
  - Duplicate Iterations of Columns
  - User ID/Sale ID codings
- However, shown in our price distributions, the target variable was highly skewed. This serves to reason as a vast majority of cars made and sold exist within a very normal price range, with a long tail for luxury and supercars.
- The most key model preparation pre-processing that was conducted were:
  - **Log Scale Price:** As shown, by taking the log of price, we are able to create a significantly more normal distribution, which will enable our models to perform better.
  - **MinMax Scalar:** Neural Networks work best when feature ranges are between 0-1, so we need to apply that transformation to our dataset, our tree based ML models do not need the same preprocessing, but it will not negatively hurt those models either.
- Additionally, the following generalized data prep steps were taken:
  - Remove duplicate columns
  - Remove spaces and text from intended numerical features
  - One-Hot encoding categorical variables with 10 levels or less (except Make/Model), dropping those with too many levels
  - Dropping columns missing over 75% of rows, dropping ID columns
  - Removing locational features like longitude, latitude, and zipcode
- Note: In a longer study with higher compute budget, many of the locational features or larger category features might be kept and treated accordingly, but with a single GPU, 3 Million rows 150+ features, and high capacity deep learning models, the data set was trimmed aggressively

Price Distribution



Price Distribution  
(Log Scale)



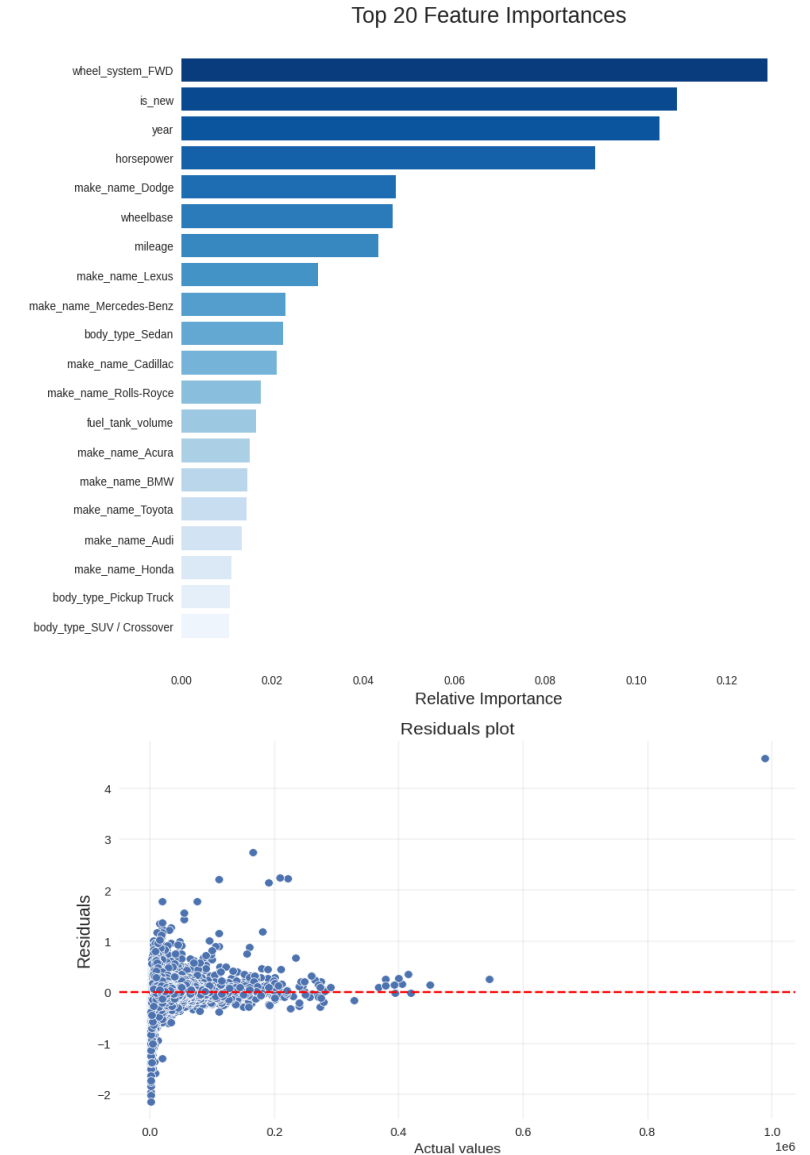


# XGBoost

- Our primary classical machine learning model tested as a challenger for our deep learning models was an Extreme Gradient Boosting (XGBoost) model.
- XGBoost has largely been the state-of-the-art tree based model for several years due to its efficient incorporation of leading tree ML methodologies like boosting, bagging, flexible hyperparameter tuning, and large data handling.
- The champion XGBoost model used random search for hyperparameter tuning with wide ranges of parameter options. Ultimately the champion model yielded a performance of:

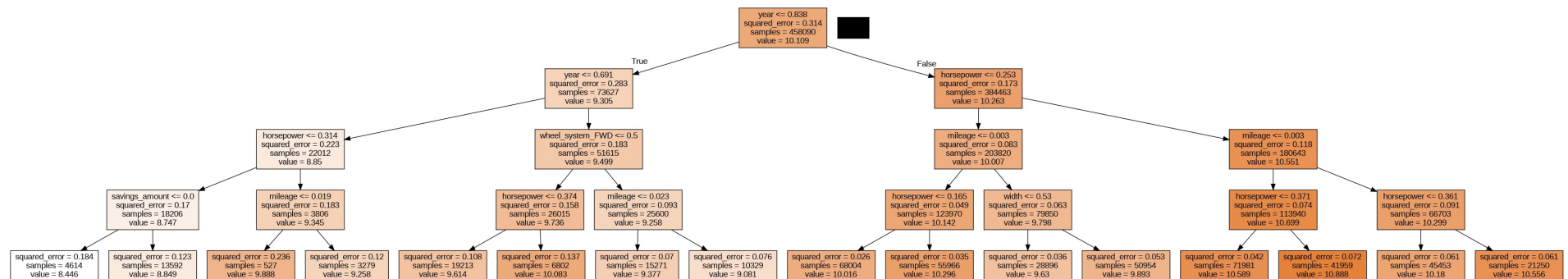
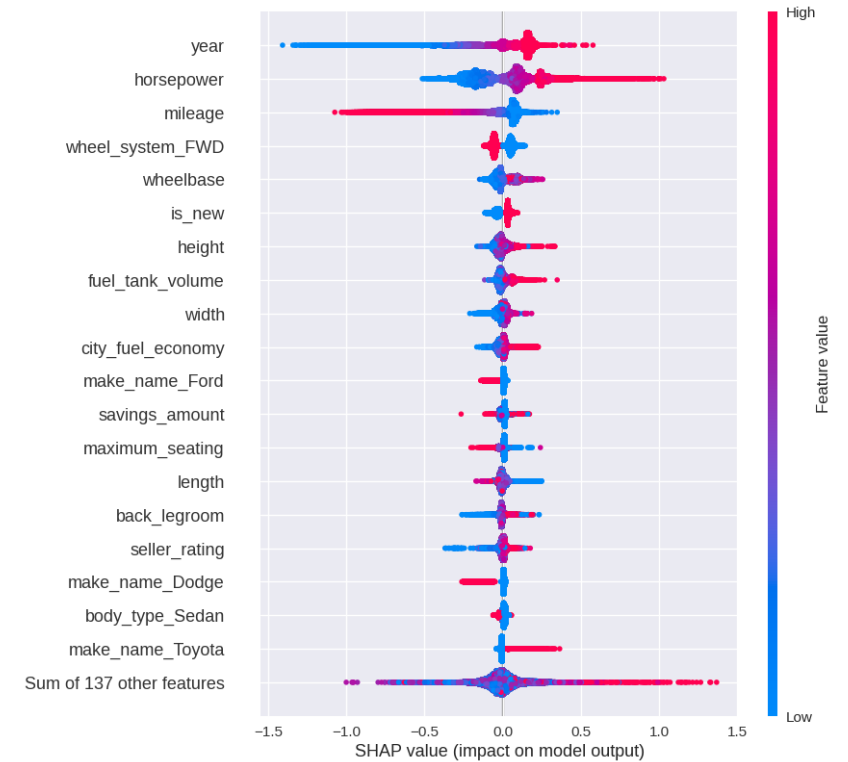
- Mean Squared Error(MSE): 0.01055
- Root Mean Squared Error(RMSE): 0.10271
- R-Squared: 0.96754

- The parameters selected by the model were:
  - **N\_estimators:** 1000
  - **Max\_depth:** 5
  - **Learning\_rate:** 0.30
  - **Gamma:** 0
  - **ColSample\_bytree:** .70
  - **Tree\_method:** gpu\_hist
- Additionally, this XGBoost model showed no signs of overfitting as test results were extremely comparable with training results.
- Overall, this model was extremely effective and we see in the feature importance plot that much of our initial correlation analysis was correct.



# XGBoost Model Interpretation

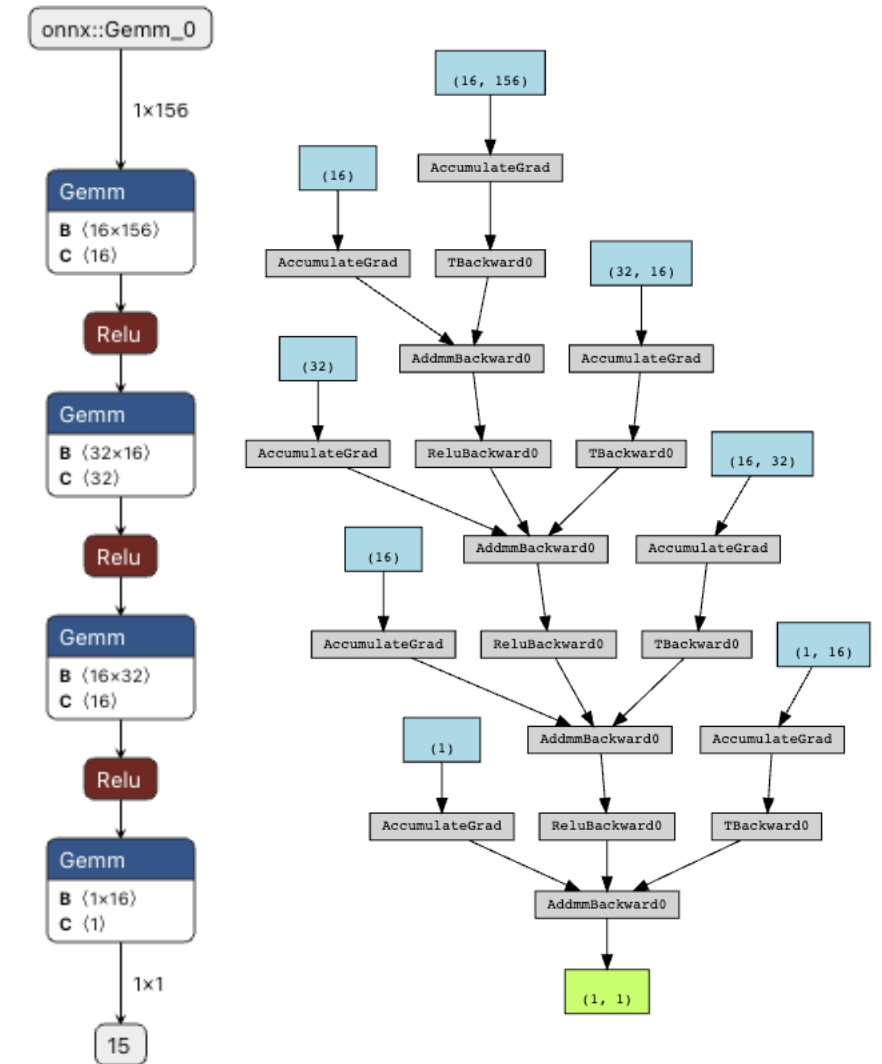
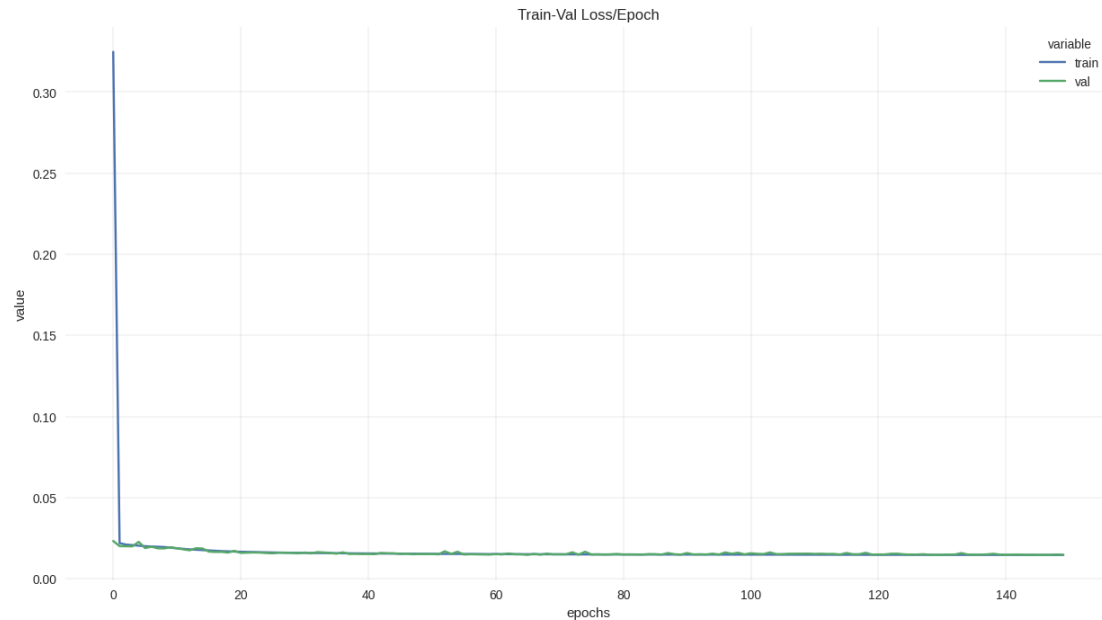
- One of the other often cited advantages of XGBoost and other tree based methods for tabular data is the suite of interpretability options available for these models.
- By creating a beeswarm plot using Shapley Values we can gain a better understanding at the sub-feature level what is driving predictions in our models.
- Some key takeaways from this plot are:
  - The higher the year (newer the car) has a heavier positive impact on price
  - Similarly, higher horsepower also has a higher positive impact on predicted price
  - Conversely, **lower** mileage also has a high positive impact on price (as is supported by is\_new)
- Additionally, below we can plot the ultimate decision tree for our model and follow the paths for most expensive and least expensive cars:
  - Most Expensive: High (recent) year, top tranche of horse power, lowest tranche of mileage yields the highest predicted value 10.888
  - Least Expensive: Oldest tranche of vehicles, low horsepower, with positive savings amounts yields the lowest predicted value of 8.446



# Neural Network Baseline

- The first neural network tested was a feed forward MLP model with 3 layers (+output) with largely out of the box settings. The following parameters were selected:
  - Batch Size: 64
  - Learning Rate = .001
  - Epochs: 150
- Ultimately, this model yielded relatively strong results but not outperforming the XGBoost:

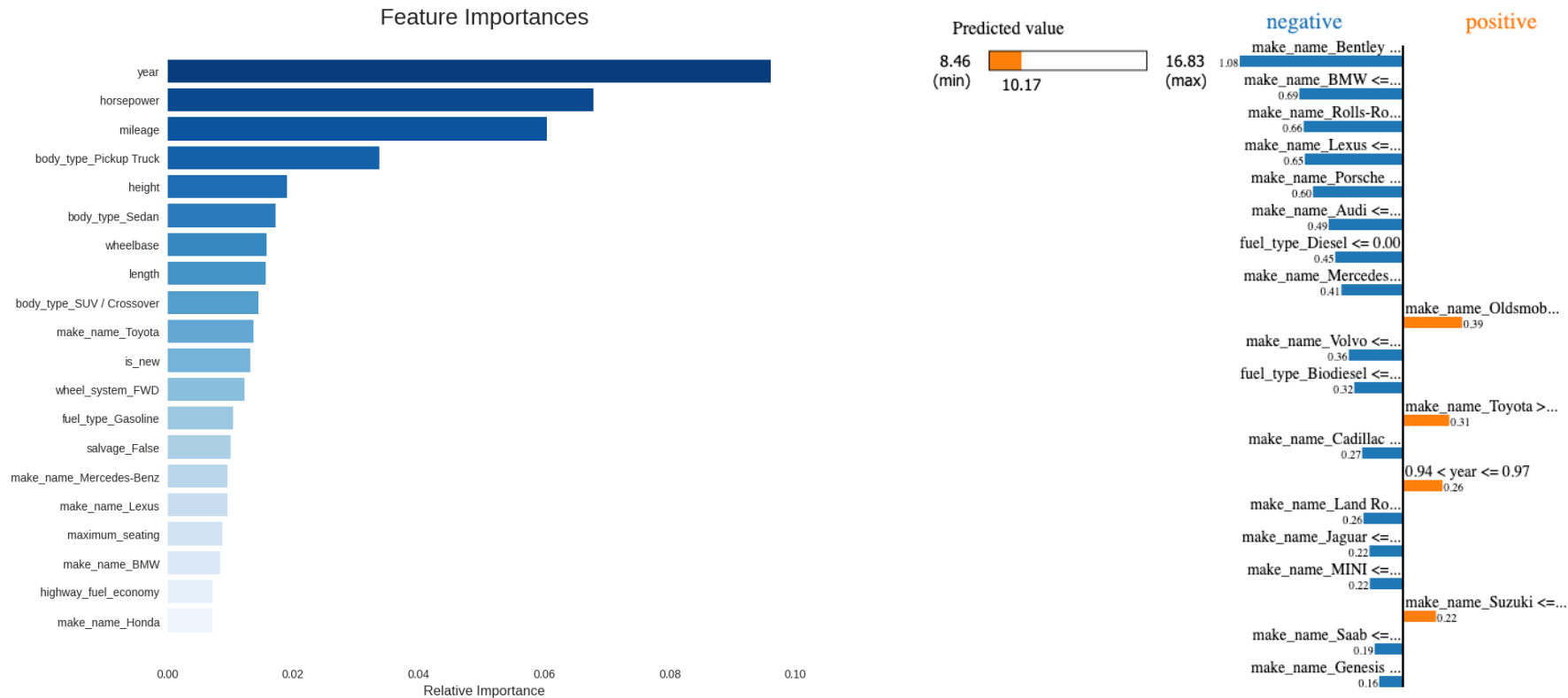
- Mean Squared Error(MSE): 0.01456
- Root Mean Squared Error(RMSE): 0.12066
- R-Squared: 0.95506





# Deep Learning Model Interpretability

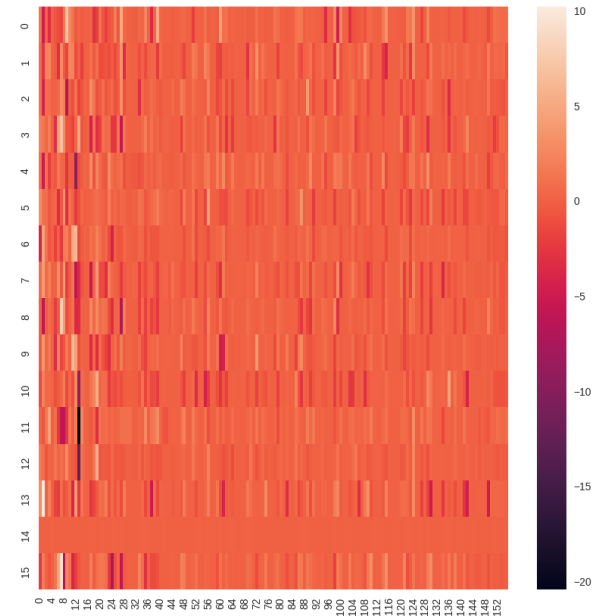
- A variety of “deep learning/neural network compatible” interpretability metrics were tested to varying degrees of efficacy, the following methods were attempted:
  - **InterpretML**: Does not support Pytorch models, focused solely on scikitlearn, and workarounds with multiple numpy to tensor conversions that ultimately were unsuccessful
  - **SHAP**: Also not directly supporting for Pytorch, wrappers to convert to numpy were build but ultimately flagged for memory requirements and NVIDIA T4 GPU timed out.  
Additionally, subsamples and clustered approaches were attempted to bypass this, additional time outs were observed.
- However, feature importances and LIME plots were able to be run on our Baseline Neural Network which affirmed what we saw in the XGBoost, but was also able to give us a directional idea of the importances of these features



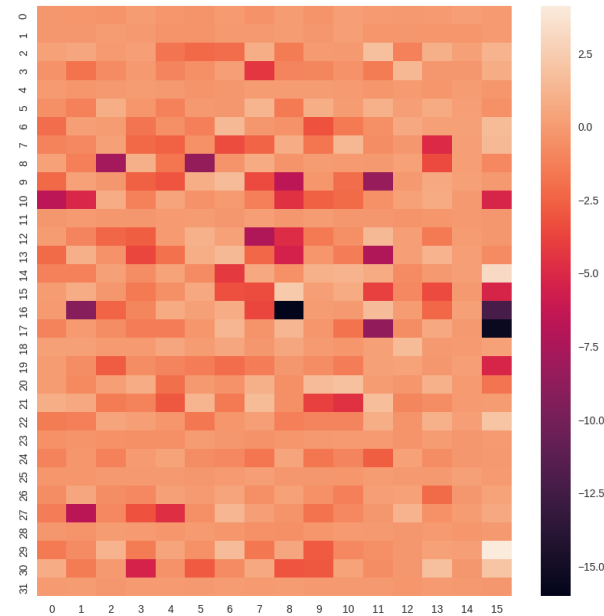
# Deep Learning Model Interpretability Contd.

- Finally, we were able to extract and map out the layer by layer weightings, to get a better visualizations of neuron per layer, the the number of synapses between the neurons, and the subsequent weighting of those connections. Of course, this is extremely hard to visually pull much out of, but we can mathematically confirm and visualize how weightings are flowing through our model. Which is a useful validation tool to have when deploying these models in production cases.

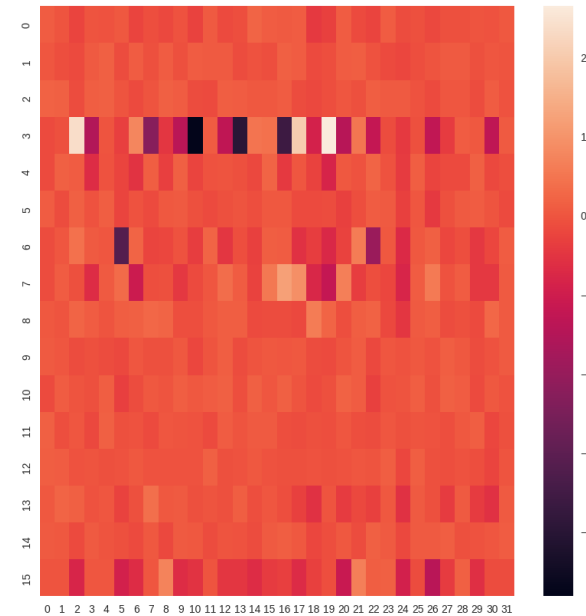
Layer 1



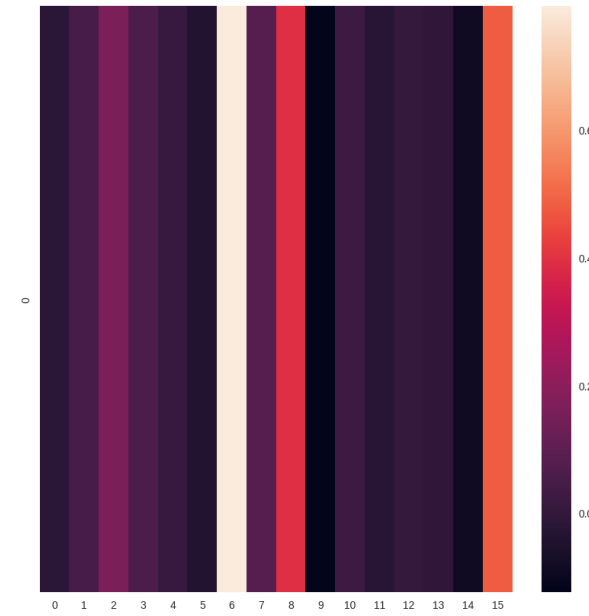
Layer 2



Layer 3

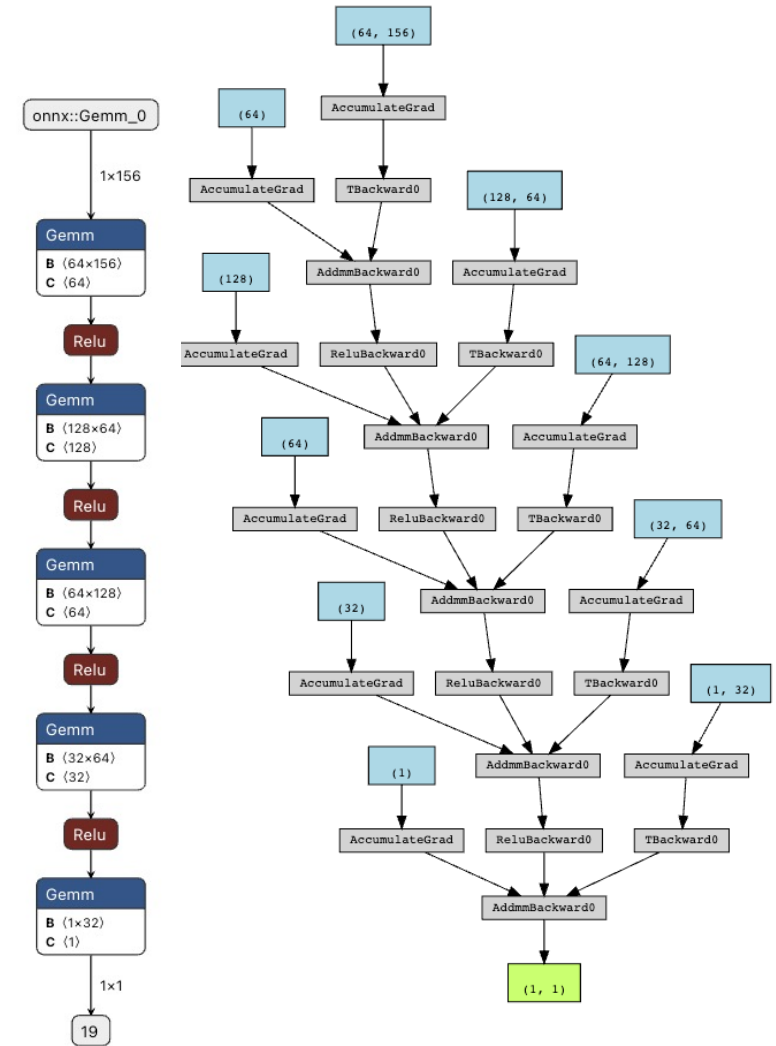
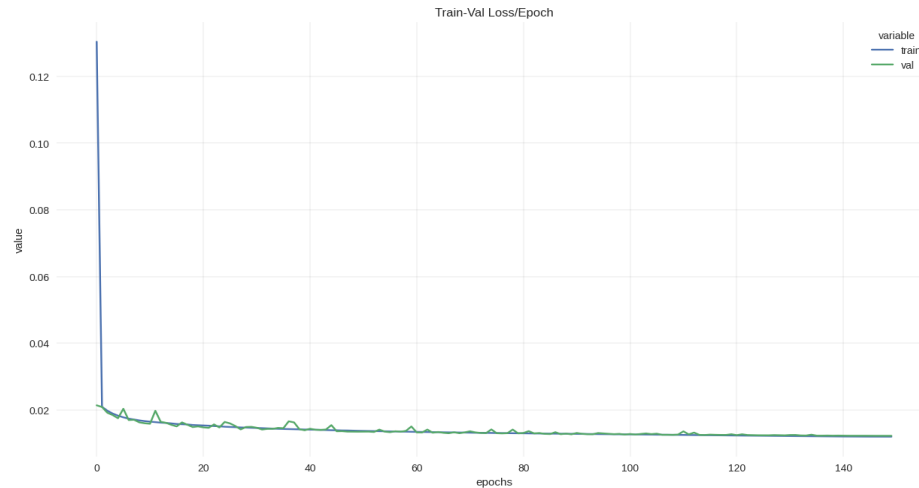


Layer Out



# Neural Network: Deep Learner One Cycle LR

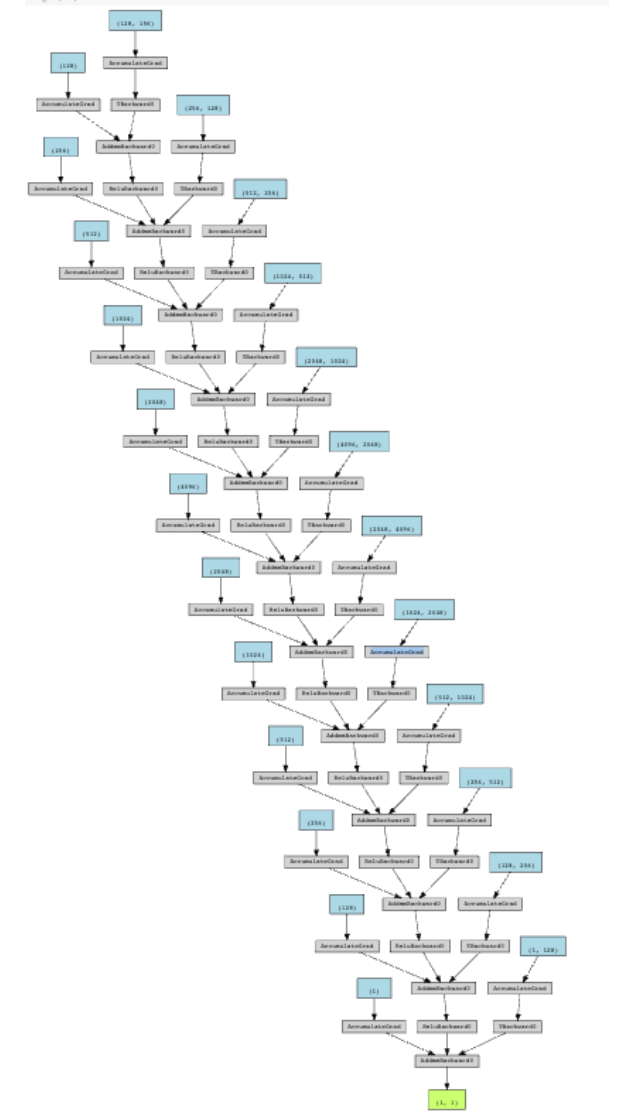
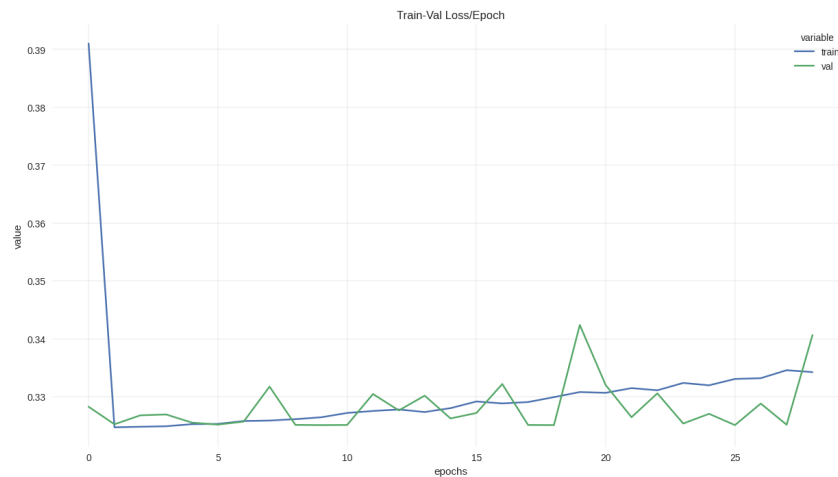
- The next model attempted was the first deep learning model leveraging 4 layers, and increasing the parameter sizing
- The key addition to this model was bringing in OneCycleLR, to attempt to incorporate the idea of super convergence into our model, in the interest of time (<https://arxiv.org/abs/1708.07120>). This brings in a learning rate scheduler into our Pytorch model that attempts to accelerate convergence by testing different learning rates over the training period
- The scores in this iteration were:
  - MSE: 0.1232
  - RMSE: 0.11101
  - R2: 0.96196
- This model actually performed quite well and showed minimal signs of overfitting, by the final epochs the train test numbers were extremely comparable.





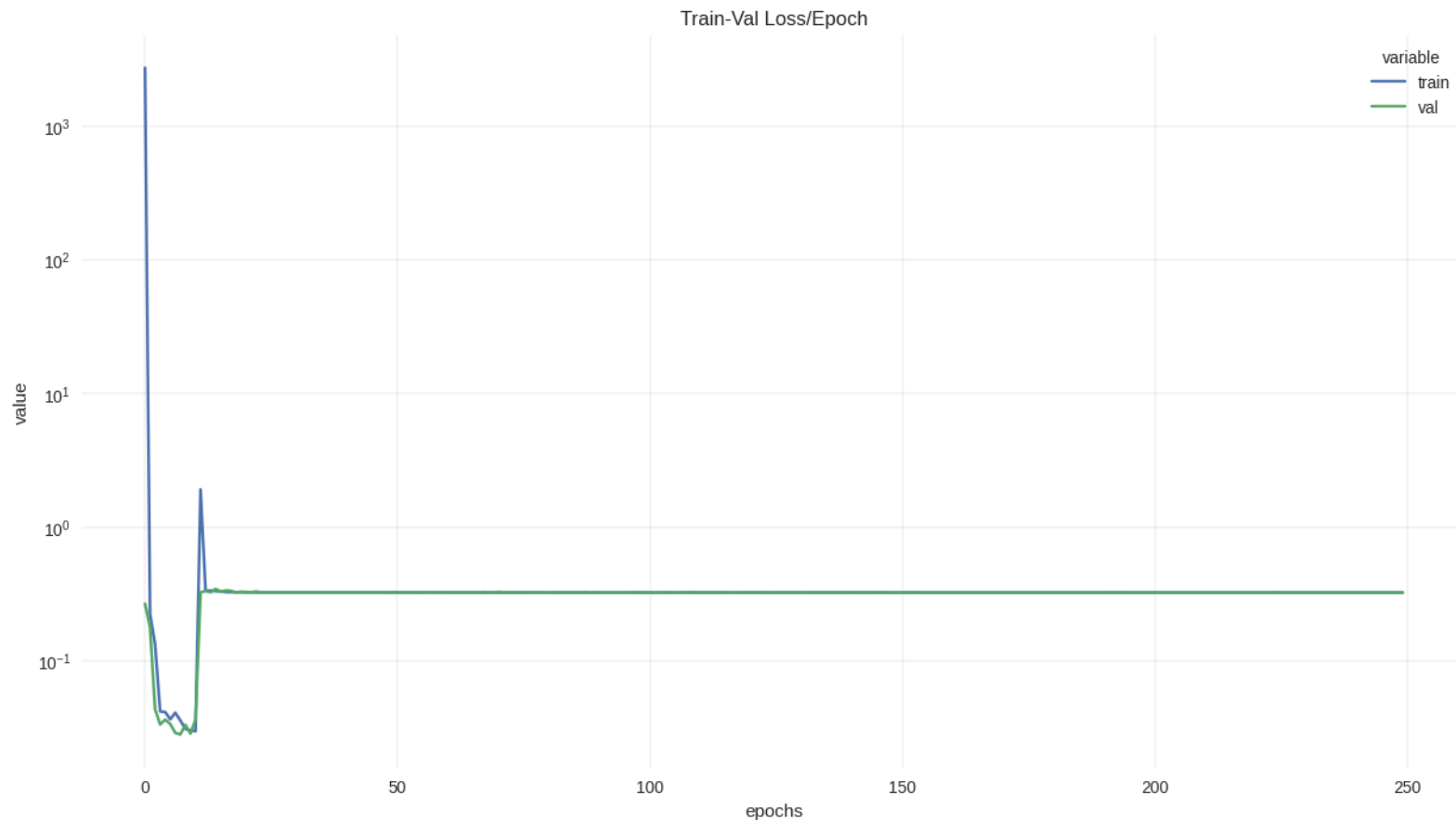
# Neural Network: DL 12 Layer OneCycleLR

- The next method was a hybrid attempt leveraging a much deeper neural network with much higher parameters, while also leveraging some methods known to help optimize for convergence and minimize computer burden
- The same OneCycleLR scheduler was adopted here with a much wider range of learning rates to be tested from .0001 to .9
- Early stopping was also integrated in order to minimize unnecessary compute and remove the risk of google colab disconnects and memory constraints
- The scores in this iteration were:
  - MSE: 0.33962
  - RMSE: 0.58277
- This high capacity model was the worst performing model in the study, showing virtually no advantages over the other baselines and deep learning models



# Neural Network: DL Double Descent Attempt

- Lastly, we attempted to replicate double descent by creating an extremely high-capacity deep learning model with 22,379,905 parameters. In early iterations we saw an optimistic drop and spike in loss score, but ultimately the model flatlined quite early around a very high error rate relative to our more tuned models. In order to replicate the double descent phenomenon, we likely would need an order of magnitude larger parameters, and significantly more compute, and ideally higher density data like text or image



```
EPOCHS = 250 # changed epochs to 250
BATCH_SIZE = 512 # changed batch size to 512
LEARNING_RATE = 0.01 # set a static learning rate of 0.01
NUM_FEATURES = len(X.columns)
train_loader = DataLoader(dataset=train_dataset, batch_size=BATCH_SIZE, shuffle=True)
val_loader = DataLoader(dataset=val_dataset, batch_size=1)
test_loader = DataLoader(dataset=test_dataset, batch_size=1)
import torch
import torch.nn as nn

class MultipleRegression(nn.Module):
    def __init__(self, num_features):
        super(MultipleRegression, self).__init__()

        self.layer_1 = nn.Linear(num_features, 128)
        self.layer_2 = nn.Linear(128, 256)
        self.layer_3 = nn.Linear(256, 512)
        self.layer_4 = nn.Linear(512, 1024)
        self.layer_5 = nn.Linear(1024, 2048)
        self.layer_6 = nn.Linear(2048, 4096) # Extra layer
        self.layer_7 = nn.Linear(4096, 2048) # Extra layer
        self.layer_8 = nn.Linear(2048, 1024)
        self.layer_9 = nn.Linear(1024, 512)
        self.layer_10 = nn.Linear(512, 256)
        self.layer_11 = nn.Linear(256, 128)
        self.layer_out = nn.Linear(128, 1) # The output layer

        self.relu = nn.ReLU()

    def forward(self, inputs):
        x = self.relu(self.layer_1(inputs))
        x = self.relu(self.layer_2(x))
        x = self.relu(self.layer_3(x))
        x = self.relu(self.layer_4(x))
        x = self.relu(self.layer_5(x))
        x = self.relu(self.layer_6(x))
        x = self.relu(self.layer_7(x))
        x = self.relu(self.layer_8(x))
        x = self.relu(self.layer_9(x))
        x = self.relu(self.layer_10(x))
        x = self.relu(self.layer_11(x))
        x = self.layer_out(x)
        return x

    def predict(self, test_inputs):
        x = self.relu(self.layer_1(test_inputs))
        x = self.relu(self.layer_2(x))
        x = self.relu(self.layer_3(x))
        x = self.relu(self.layer_4(x))
        x = self.relu(self.layer_5(x))
        x = self.relu(self.layer_6(x))
        x = self.relu(self.layer_7(x))
        x = self.relu(self.layer_8(x))
        x = self.relu(self.layer_9(x))
        x = self.relu(self.layer_10(x))
        x = self.relu(self.layer_11(x))
        x = self.layer_out(x)
        return x
```

# Final Model Results & Observations



- MSE: 0.01232
- RMSE: 0.11101
- R2: 0.96196
- Highly flexible and scalable
- GPU efficient, but computationally slow
- Solid amount of interpretability features
- Impressive performance on relatively small tabular data

## Champion Model

**dmlc**  
**XGBoost**

- MSE: 0.01055
- RMSE: 0.10271
- R2: 0.96754
- State of the Art Performance
- GPU Intensive but computationally very fast
  - Extensive interpretability options
- Extremely user friendly and easy to develop



# Results

## Deep Learning vs. Machine Learning Findings

- Our initial findings shows that leading tabular machine learning methods such as XGBoost leveraging random search for hyperparameter optimization outperform a variety of architectures of deep learning models both relatively simply to very complex and high capacity.
- What we also found however, is that for deep learning on tabular data, while not all standardized explainability metrics are available, many feature importances and contribution metrics are available as well as intra layer weight and bias metrics.
- Ultimately, in this case, double descent was not able to be replicated. However, I believe this is a function of the data set being too small (<10GB), it being tabular, and model capacity restrictions (in tandem with time) on the google colab single NVIDIA T4 GPU.

## Business Case Interpretation

- The initial interpretation for a hypothetical used car sales business client would be that there are a variety of machine learning and deep learning methods that can provide highly predictive analytics to these businesses.
  - These key metrics like shapley values, lime, visualized decision trees, can enable operators to make much more informed strategic decisions about the cars they choose to hold in inventory or clusters of automobiles they choose to focus on
  - Two next steps to be considered at the microeconomic/firm level would be to analyze the demographics of the given auto dealership and determine which bucket of the price range fits your locale, additionally incorporating any information on profit yield from each car sale would be immensely helpful in understanding which cars drive the most margin and how to weight that against cost of carry and time to sale
-

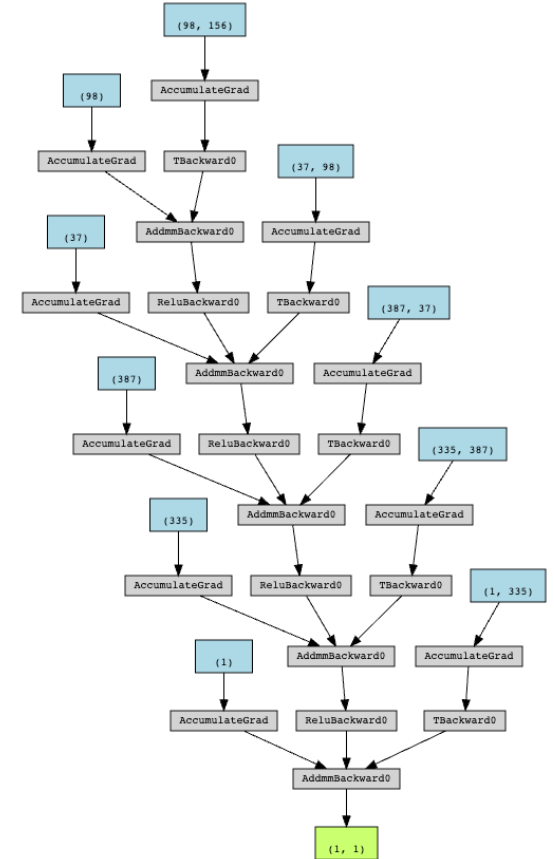
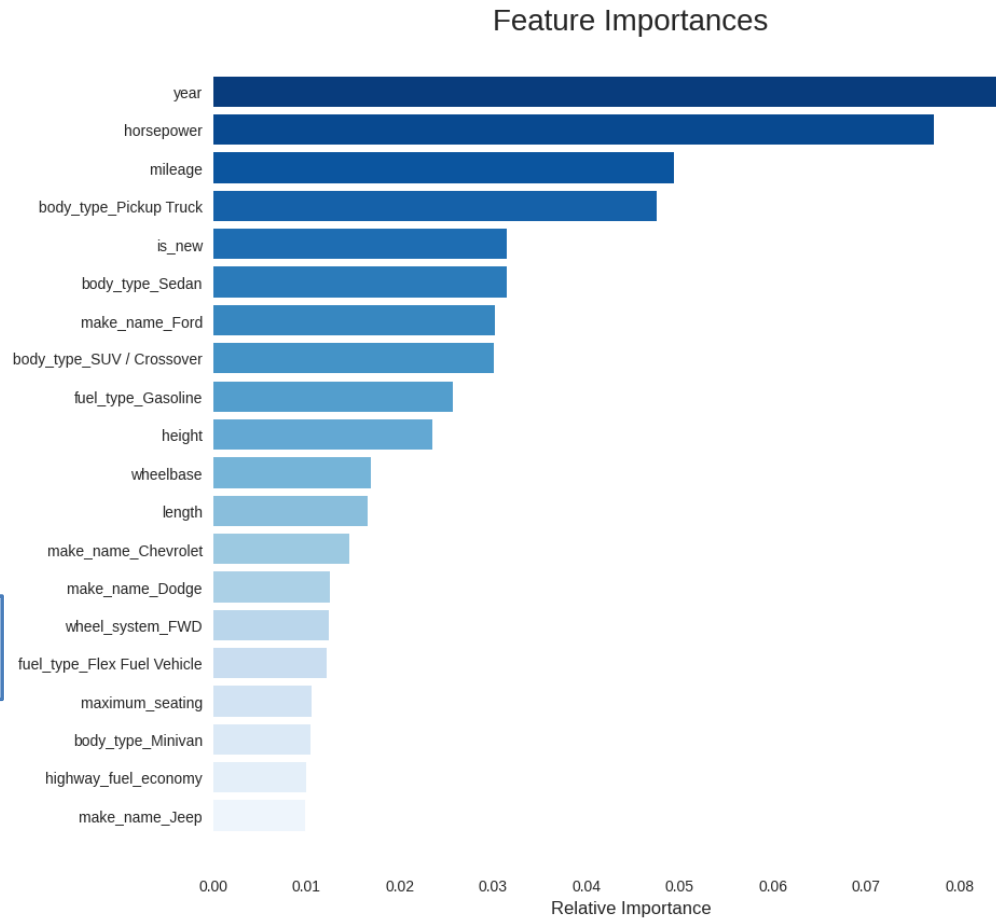
A high-angle, rear-view shot of a multi-lane highway completely clogged with cars. The vehicles are packed closely together, filling the frame from the foreground into the distance. The scene is dimly lit, suggesting dusk or dawn, with some brake lights visible as small red glows. The word "Appendix" is centered in the middle of the image in a white, sans-serif font.

# Appendix

# Neural Network: DL Random Search

- An additional deep learning model deploying random search for hyperparameter optimization was tested, however ultimately realized there was a bug in the code, that did not optimize on the random parameters
- This model simply selected random parameters within a range and then trained
- While performance was surprisingly good, it was not better than our previous champion model


Mean Squared Error : 0.012709852494375338  
R<sup>2</sup> : 0.9607662606125476  
RMSE : 0.11273798159615657





# Neural Network: DL Random Search #2

- Finally, a true random search for hyperparameter optimization model was developed and upon its first iteration the error metrics on the final epochs appeared to approach our top performers with the following architecture and parameter selections:

```
cuda:0
Training with parameters: {'batch_size': 368, 'learning_rate': 0.010656343659147235, 'neurons_layer_1': 64, 'neurons_layer_2': 294, 'neurons_layer_3': 304, 'neurons_layer_4': 109}
100%  150/150 [5:17:35<00:00, 127.03s/it]
```

- However, the entirety of this code including the subsequent 9 iterations was allocated to take over 50+ additional hours in order to complete, so output metrics and train test learning curve plots were unable to be achieved.
- Additional information noted in results page.

```
Epoch 145: | Train Loss: 0.01165 | Val Loss: 0.01199
Epoch 146: | Train Loss: 0.01165 | Val Loss: 0.01199
Epoch 147: | Train Loss: 0.01164 | Val Loss: 0.01199
Epoch 148: | Train Loss: 0.01164 | Val Loss: 0.01199
Epoch 149: | Train Loss: 0.01163 | Val Loss: 0.01199
Epoch 150: | Train Loss: 0.01163 | Val Loss: 0.01199

# Hyperparameters
EPOCHS = 150
NUM_FEATURES = len(X.columns)
# Hyperparameter space
hyperparam_space = {
    'learning_rate': expon(0.001, 0.1),
    'batch_size': randint(32, 512),
    'neurons_layer_1': randint(32, 512),
    'neurons_layer_2': randint(32, 512),
    'neurons_layer_3': randint(32, 512),
    'neurons_layer_4': randint(32, 512),
}

# Number of iterations for random search
n_iter = 10
parameter_sampler = ParameterSampler(hyperparam_space, n_iter)

best_val_loss = np.inf
best_params = None

for params in parameter_sampler:
    BATCH_SIZE = params['batch_size']
    LEARNING_RATE = params['learning_rate']
    neurons_layer_1 = params['neurons_layer_1']
    neurons_layer_2 = params['neurons_layer_2']
    neurons_layer_3 = params['neurons_layer_3']
    neurons_layer_4 = params['neurons_layer_4']
```