Predicting Car Auction Prices

By: Reese Karo

Project Overview:

This end-to-end project aims to predict car auction prices using various machine learning techniques, including XGBoost, and a Deep Neural Network. The data has been preprocessed and cleaned, and the models are evaluated based on their performance metrics. Both machine learning models use hyperparameter tuning to determine the best set of hyperparameters using different techniques, including: grid searching of pre-selected parameters, and Hyperband tuning.

In [38]:
```
# import libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import pickle
```

In [39]:
```
path = '/Users/reese/Documents/School/UC Santa Barbra /PSTAT/131/Project'
```

Data Preprocessing

Importing and Cleaning Data

- **Data Source:** The data was initially cleaned using R and Tidyverse, and can be found here: https://github.com/reese-karo/Portfolio/blob/main/Car%20Auction%20Machine%20Learning%20Project/CarAuctionML.pdf. Further cleaning was performed in Python to remove outliers.
- **Outlier Removal:** Vehicles priced over $60,000 or under$1,000, and those with more than 200,000 miles on the odometer, were removed to improve model performance. Also by condensing to the top 8 vehicle colors were kept. We note that age is based off of 2015 results, where 2015 is the newest car at age 0.

In [40]:
```
# Importing the data
cars_df = pd.read_csv(path + '/car_data_cleaned.csv')
# remove the unnamed column because it is an index column
cars_df = cars_df.drop(columns=['Unnamed: 0'])
# convert object types to categorical
cars_df = cars_df.apply(lambda x: x.astype('category') if x.dtype == 'object' else x)

print(cars_df.describe(), '\n')
print(cars_df.info(), '\n')
print(cars_df[:5], '\n')
```

```
          condition       odometer   sellingprice           age
count  78820.000000   78820.000000   78820.000000  78820.000000
mean       3.408368   67752.882479   13495.842806      4.878749
std        0.942517   52359.714698    9518.425494      3.854083
min        1.000000       1.000000     100.000000      0.000000
25%        2.700000   28756.000000    7000.000000      2.000000
50%        3.600000   52515.000000   12000.000000      3.000000
75%        4.200000   98061.500000   18000.000000      7.000000
max        5.000000  999999.000000  160000.000000     25.000000

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 78820 entries, 0 to 78819
Data columns (total 12 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   make          78820 non-null  category
 1   model         78820 non-null  category
 2   trim          78820 non-null  category
 3   body          78820 non-null  category
 4   transmission  78820 non-null  category
 5   condition     78820 non-null  float64
 6   odometer      78820 non-null  int64
 7   color         78820 non-null  category
 8   interior      78820 non-null  category
 9   sellingprice  78820 non-null  int64
 10  region        78820 non-null  category
 11  age           78820 non-null  int64
dtypes: category(8), float64(1), int64(3)
memory usage: 3.2 MB
None

        make             model              trim  ... sellingprice     region  age
0       mini            cooper            cooper  ...        16500  northeast    4
1        bmw                x5                x5  ...        12600       west    7
2     toyota           corolla           corolla  ...        13700      south    1
3       ford              edge              edge  ...        20600    midwest    2
4  chevrolet           equinox           equinox  ...        20300    midwest    3
5  chevrolet  silverado 2500hd  silverado 2500hd  ...        34600       west    2
6       ford        expedition        expedition  ...         7500       west    9
7      dodge     grand caravan     grand caravan  ...         2700      south   10
8  chevrolet              volt              volt  ...        11400    midwest    3
9       ford            escape            escape  ...        13600    midwest    3

[10 rows x 12 columns]
```

In [41]:

```python
# after further eda, we will need to remove outliers from the data
print('Count of cars that are priced more than 60000:', len(cars_df[cars_df['sellingprice'] > 60000]), 'out of', le
print('Removing outliers in price...')
cars_df = cars_df[(cars_df['sellingprice'] > 1000) & (cars_df['sellingprice'] < 60000)] # remove outliers in price
print('Removing outliers in odometer...')
cars_df = cars_df[cars_df['odometer'] < 200000]

print('Keeping the top 8 most common colors...')
cars_df = cars_df[cars_df['color'].isin(cars_df['color'].value_counts()[:8].index)]

cars_df.describe()
```

```
Count of cars that are priced more than 60000: 198 out of 78820
Removing outliers in price...
Removing outliers in odometer...
Keeping the top 8 most common colors...
```

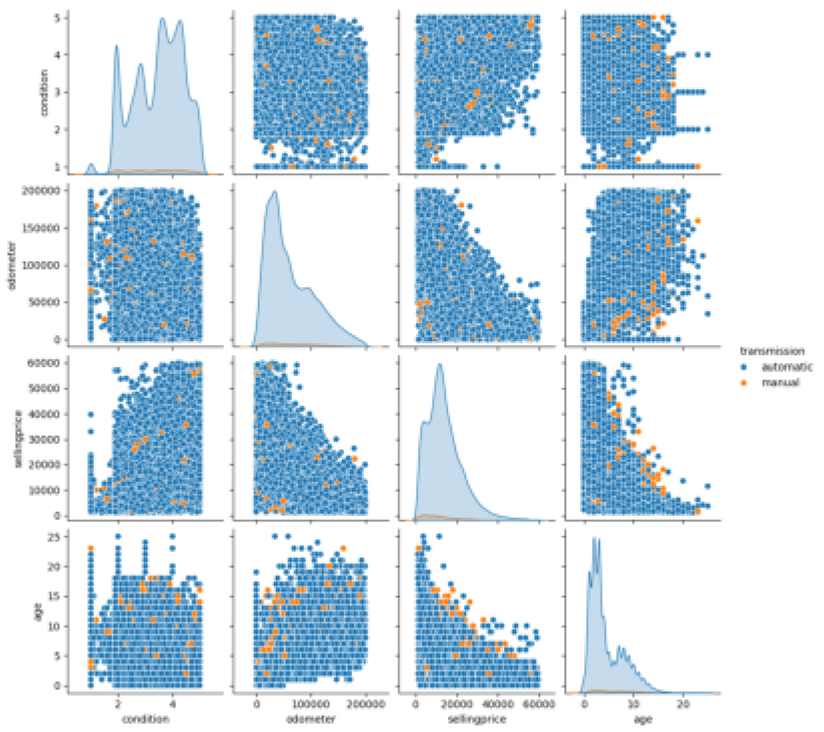
Out[41]:

	condition	odometer	sellingprice	age
count	71550.000000	71550.000000	71550.000000	71550.000000
mean	3.470734	61934.371349	13981.701621	4.457331
std	0.911604	43819.533838	8751.110917	3.411017
min	1.000000	1.000000	1050.000000	0.000000
25%	2.800000	27729.750000	8000.000000	2.000000
50%	3.600000	49139.500000	12500.000000	3.000000
75%	4.200000	91009.500000	18400.000000	7.000000
max	5.000000	199979.000000	59800.000000	25.000000

Exploratory Data Analysis (EDA)

- **Visualizations:** Pair plots and heatmaps were used to understand the relationships between variables.
- **Key Insights:** Age and odometer readings have the highest correlation, indicating that older cars with higher mileage tend to sell for less.

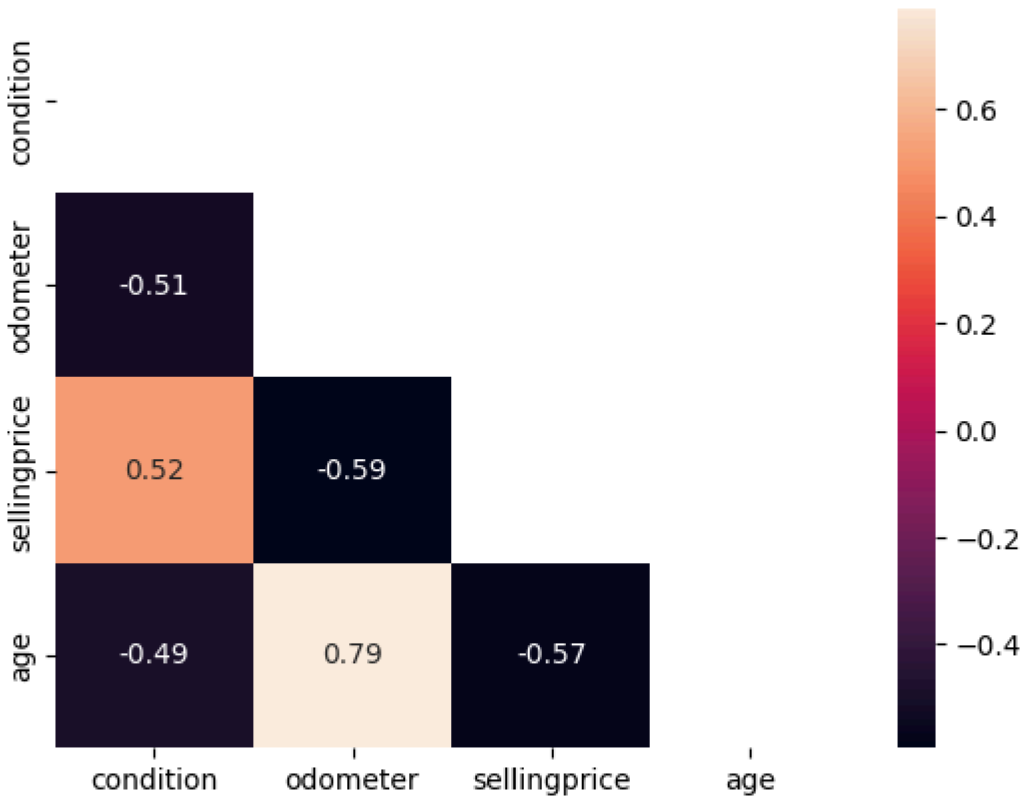
In [42]:
```
# we will first use seaborn to show a few graphs and distributions of the data to get an understanding of the numer
numerics = ['int16', 'int32', 'int64', 'float16', 'float32', 'float64']
numeric_df = cars_df.select_dtypes(include=numerics)
numeric_df['transmission'] = cars_df['transmission'].astype('category')

'''
sns.pairplot(numeric_df, hue = 'transmission')
plt.savefig(path + '/pairplot.png')
plt.show()
'''

# load in the pairplot already created
pairplot = plt.imread(path + '/pairplot.png')
plt.imshow(pairplot)
plt.axis('off')
plt.show()
```



After removing outliers in the prices and the odometer, we have a good representation of some important predictors

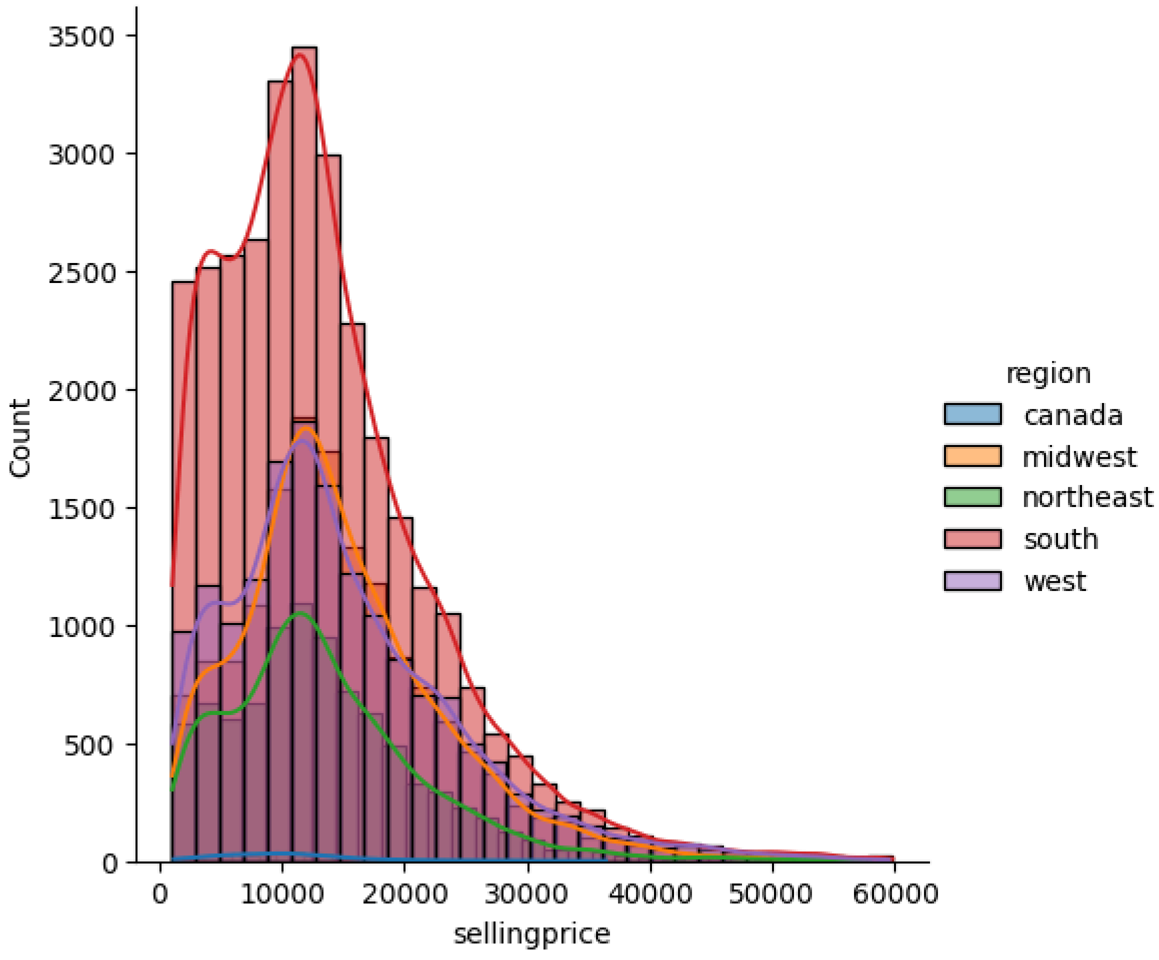
Next we will get our correlation data to see what predictors are most influential on each other, having the highest and lowest correlations

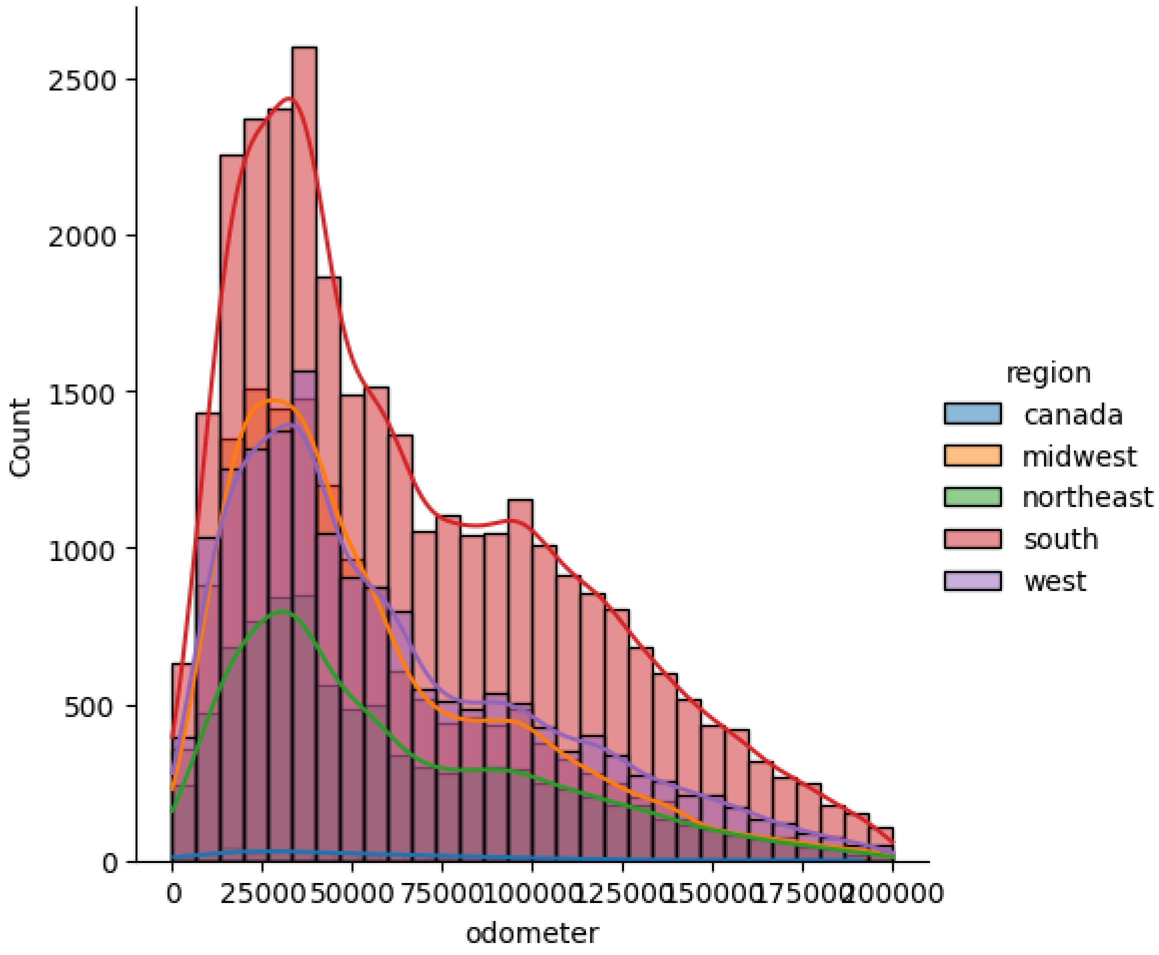
In [43]:
```
# get correlation matrix
numeric_df = numeric_df.drop(columns=['transmission'])
corr_matrix = numeric_df.corr()

# mask
mask = np.triu(np.ones_like(corr_matrix, dtype=bool))

# plot the heatmap
sns.heatmap(corr_matrix, annot = True, mask = mask)
plt.show()
```



In [44]:
```
sns.FacetGrid(cars_df, hue = 'region', height = 5).map(
    sns.histplot, 'sellingprice', bins = 30, kde = True).add_legend()
plt.show()

sns.FacetGrid(cars_df, hue = 'region', height = 5).map(
```

```
    sns.histplot, 'odometer', bins = 30, kde = True).add_legend()
plt.show()
```





Pipeline

Concept:

- Split the data into training and testing first where the X train/test contains all but the prediction variable, and y train/test contains only the selling price data
- Next we collect a list of only numeric and categorical predictors so we can scale the numeric and change the catgeorical into 'One-Hot' or dummy variables
- Then we use a column transformer to combine the numerical and catgeorical columns into a preprocesser for the next step
- Then our official pipeline takes in the preprocesser, and removes any values with 0 variance.

This provides us with a transformed training and testing set to work with to train the models below.

In [45]:
```
# start the preprocessing steps
from sklearn.compose import ColumnTransformer #
from sklearn.pipeline import Pipeline # handle the pipeline of preprocessing steps
from sklearn.preprocessing import OneHotEncoder, StandardScaler # scales the data
from sklearn.feature_selection import VarianceThreshold # removes zero variance predictors
from sklearn.model_selection import train_test_split # split the data into training and testing data

# split the data into training and testing data
X_train, X_test, y_train, y_test = train_test_split(cars_df.drop(columns=['sellingprice']),
                                                    cars_df['sellingprice'], test_size=0.2, random_state=12345)
```

```
##### create the pipeline #####

# create a list of numeric and categorical features
numeric_features = X_train.select_dtypes(include=['float64', 'int64']).columns.tolist()
categorical_features = X_train.select_dtypes(include=['category']).columns.tolist()

# create transformers for numeric and categorical data
numeric_transformer = Pipeline(steps=[
    ('scaler', StandardScaler())]) # normalize numeric features

categorical_transformer = Pipeline(steps=[
    ('onehot', OneHotEncoder(handle_unknown='ignore'))]) # handle novel categories with the ignore option

# combine transformers into a preprocessor
preprocessor = ColumnTransformer(
    transformers=[
        ('num', numeric_transformer, numeric_features),
        ('cat', categorical_transformer, categorical_features)
    ]
)

# create the main pipeline with variance thresholding for zero variance predictors
pipeline = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('variance', VarianceThreshold(threshold=0))
])

# fit and transform
X_train_processed = pipeline.fit_transform(X_train)
X_test_processed = pipeline.transform(X_test)
```

Modeling

XGBoost

- **Hyperparameter Tuning:** Used `GridSearchCV` to find the best parameters, optimizing for minimizing Mean Squared Error (MSE).
- **Model Evaluation:** The model was evaluated on a testing dataset, achieving a Mean Squared Error of `{{ mse_test_xgb }}`.

In [ ]:
```
import xgboost as xgb
from sklearn.model_selection import KFold
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import mean_squared_error
import numpy as np

# create a validation set
X_train_small, X_val, y_train_small, y_val = train_test_split(X_train_processed, y_train, test_size=0.1, random_sta

# init the model
xgb_reg = xgb.XGBRegressor(objective='reg:squarederror', random_state=12345,
                           early_stopping_rounds=20) # including a validation set for early stopping

# set the parameter grid
param_grid_xgb = {
    'learning_rate': [0.01, 0.05], # learning rate
    'max_depth': [9, 12], # depth of each tree, higher is more complex
    'colsample_bytree': [0.5, 1] # controls the fraction of the observations to be randomly samples for each tree
}
# Fold the data into 10 folds
kfold = KFold(n_splits=10, shuffle=True, random_state=12345)
'''
# grid search for xgboost
tuning_xgb = GridSearchCV(estimator=xgb_reg,
                          param_grid=param_grid_xgb,
                          cv=kfold,
                          scoring='neg_mean_squared_error', # optimize for minimizing MSE
                          verbose=1) # show progress
# fit the tuning model
print('Tuning the model...')
tuning_xgb.fit(X_train_small, y_train_small, eval_set=[(X_val, y_val)])

# show the best parameters
best_xgb = tuning_xgb.best_params_
print('Best parameters:', best_xgb)

final_xgb = xgb.XGBRegressor(
    learning_rate=best_xgb['learning_rate'],  # higher learning rate after tuning
    max_depth=best_xgb['max_depth'],
    colsample_bytree=best_xgb['colsample_bytree'],
    early_stopping_rounds=20,
    random_state=12345
)

# Use early stopping while training to determine optimal number of trees
print('Fitting the final model...')
```

```
final_xgb.fit(X_train_processed, y_train,
              eval_set=[(X_val, y_val)],
              verbose=False)

# save the model
with open(path + '/final_xgb.pkl', 'wb') as f:
    pickle.dump(final_xgb, f)
'''
```

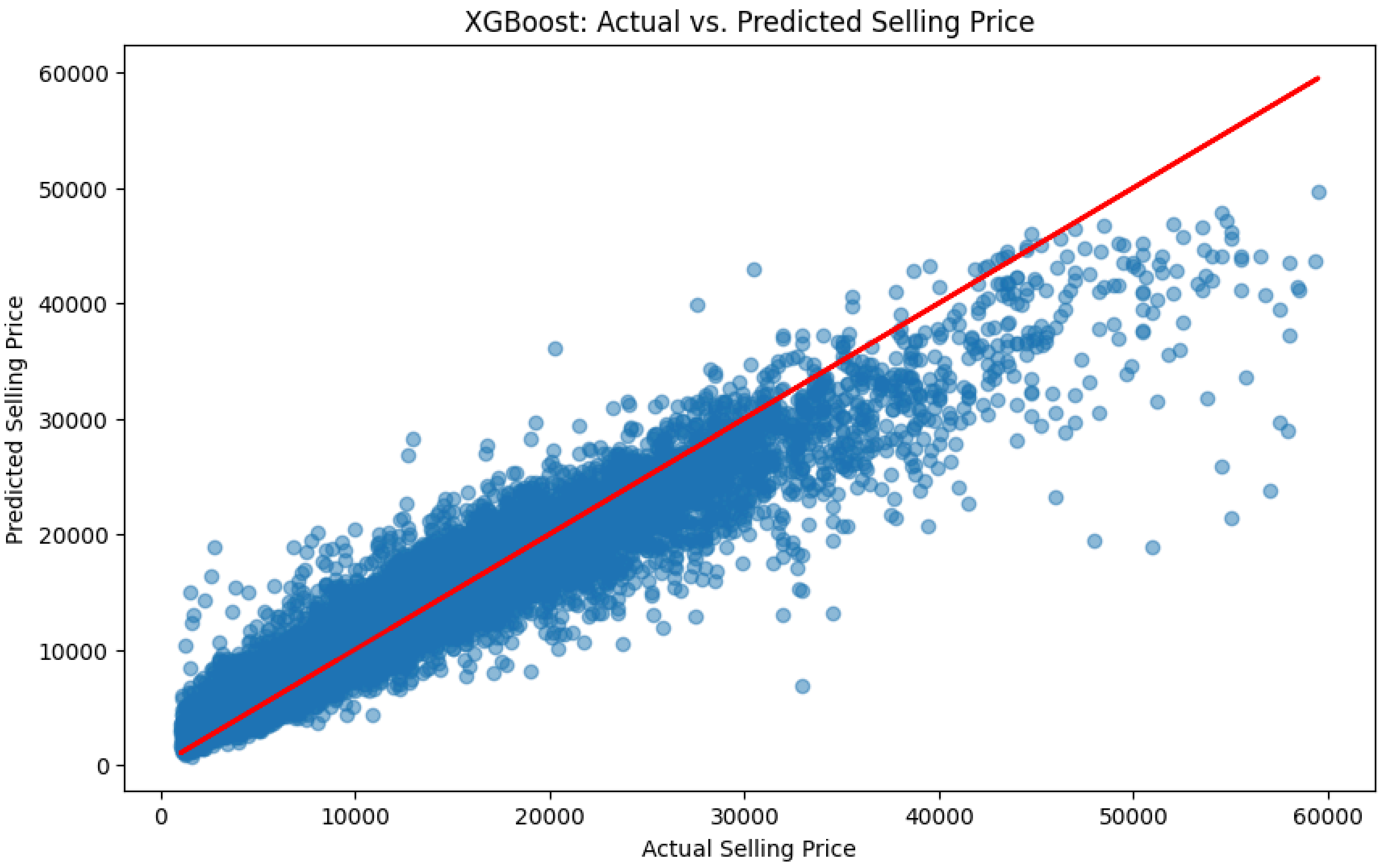
XGBoost Predictions

In [18]:
```
with open(path + '/final_xgb.pkl', 'rb') as f:
    final_xgb = pickle.load(f)
# predict on the testing data
print('Using the final model to predict on the testing data...')
y_pred_xgb = final_xgb.predict(X_test_processed)

# evaluate the model
print('Evaluating the final model...')
mse_test_xgb = mean_squared_error(y_test, y_pred_xgb)
print(f'Mean Squared Error: {mse_test_xgb}')
```

```
Using the final model to predict on the testing data...
Evaluating the final model...
Mean Squared Error: 8817419.028055886
```

Visualizing XGB Performance

In [19]:
```
# Plotting the Performance of XGBoost
plt.figure(figsize=(10, 6))
plt.scatter(y_test, y_pred_xgb, alpha=0.5)
plt.plot(y_test, y_test, color='red', linewidth=2)
plt.xlabel('Actual Selling Price')
plt.ylabel('Predicted Selling Price')
plt.title('XGBoost: Actual vs. Predicted Selling Price')
plt.show()
```



Results

- We see that the model performance follows the 1:1 ratio line of a perfect actual selling versus predicted selling price with predictions falling on both sides of the red line. This happens until the 30,000 actual selling price where we see the model starts to predict more expensive cars (truth) as less expensive cars (pred). Thus some potential reasons could be the model didn't train enough on higher priced cars, since there are fewer cars that cost more than 30,000.

Deep Neural Network (DNN)

- **Hyperband Tuning:** Focused on promising parameter combinations, prioritizing learning rates and network architectures.
- **Model Evaluation:** The DNN was evaluated on a testing dataset, achieving a Mean Squared Error of `{{ mse_test_nn }}`.

Hyperparameter Tuning

Hyperband Tuning first explores a large number of configurations with a small number of training epochs, Then progressively increases the number of epochs for the most promising configurations, narrowing in on the best subset of hyperparameters
We want to prioritize tuning learning rates, and network architectures first. Using dropout and early stopping and regularization, we can avoid overfitting.

The model builder below will generate us the best hyperparameters which we can train our final neural net model on the training dataset.

In [55]:
```
import tensorflow as tf
import keras_tuner as kt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.callbacks import EarlyStopping, CSVLogger
```

In [56]:
```
def model_builder(hp):
    '''
    This function builds neural nets to be passed into the tuner,
    finding the best hyperparameters for the training set.
    '''
    model = Sequential() # initialize the model
    model.add(Dense(units=hp.Int('units', min_value=32, max_value=512, step=32),
                    activation='relu',
                    kernel_regularizer=tf.keras.regularizers.l2(hp.Float('l2_reg', min_value=1e-4, max_value=1e-2,
    model.add(Dropout(rate=hp.Float('dropout', min_value=0.1, max_value=0.5, step=0.1))) # dropout layer with a var
    model.add(Dense(10, activation='relu')) # dense layer with 10 units
    model.add(Dense(1, activation='linear')) # output layer
    model.compile(optimizer=tf.keras.optimizers.Adam(hp.Choice('learning_rate', values=[1e-2, 1e-3, 1e-4])), # comp
                  loss='mean_squared_error',
                  metrics=['root_mean_squared_error'])
    return model

# add in callbacks
early_stopping = EarlyStopping(monitor='val_root_mean_squared_error',
                               patience=10,
                               restore_best_weights=True)
csv_logger = CSVLogger(path + '/training_logs.csv')

# initialize the tuner
tuner_nn = kt.Hyperband(model_builder,
                       objective='val_root_mean_squared_error', # objective is to minimize the validation root mean
                       max_epochs=50, # maximum number of epochs to train the model
                       factor=3, # factor by which the number of epochs is increased
                       directory=path,
                       project_name='Car_price_NN_tuning') # directory to save the model

# search for the best hyperparameters
tuner_nn.search(X_train_processed, y_train, epochs = 50, validation_data =(X_val, y_val), callbacks=[early_stopping

# get the best hyperparameters
best_hps = tuner_nn.get_best_hyperparameters(num_trials=1)[0]

# saving the best model with pickle
with open(path + '/best_hps.pkl', 'wb') as f:
    pickle.dump(best_hps, f)
```

```
Trial 90 Complete [00h 01m 58s]
val_root_mean_squared_error: 2903.47119140625

Best val_root_mean_squared_error So Far: 1893.8212890625
Total elapsed time: 00h 44m 07s
```

In [57]:
```
from sklearn.metrics import mean_squared_error

# opening the best model with pickle
with open(path + '/best_hps.pkl', 'rb') as f:
    best_hps = pickle.load(f)

# build the model with the best hyperparameters
final_nn_model = tuner_nn.hypermodel.build(best_hps)
print(final_nn_model.summary()) # summary of the model

# different csv logger for the final training
train_csv_logger = CSVLogger(path + '/final_nn_training_logs.csv')

# Split the original training data into a new training and validation set
X_train_new, X_val, y_train_new, y_val = train_test_split(X_train_processed, y_train, test_size=0.1, random_state=1

# Retrain the model on the new training set
history = final_nn_model.fit(X_train_new, y_train_new,
                             validation_data=(X_val, y_val),
                             epochs=100,
                             callbacks=[early_stopping, train_csv_logger],
                             verbose=1)
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
dense_3 (Dense)	?	0 (unbuilt)
dropout_1 (Dropout)	?	0
dense_4 (Dense)	?	0 (unbuilt)
dense_5 (Dense)	?	0 (unbuilt)

Total params: 0 (0.00 B)
Trainable params: 0 (0.00 B)
Non-trainable params: 0 (0.00 B)

```
None
Epoch 1/100
1610/1610 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - loss: 52538216.0000 - root_mean_squared_error: 6729.5098 - val_loss: 73
95765.0000 - val_root_mean_squared_error: 2719.4126
Epoch 2/100
1610/1610 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 7496424.0000 - root_mean_squared_error: 2737.5115 - val_loss: 677
0743.0000 - val_root_mean_squared_error: 2601.9207
Epoch 3/100
1610/1610 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 7001836.5000 - root_mean_squared_error: 2645.4338 - val_loss: 632
2325.0000 - val_root_mean_squared_error: 2514.2395
Epoch 4/100
1610/1610 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 6433697.5000 - root_mean_squared_error: 2536.0825 - val_loss: 646
2366.0000 - val_root_mean_squared_error: 2541.9028
Epoch 5/100
1610/1610 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 6131619.5000 - root_mean_squared_error: 2474.8035 - val_loss: 611
6129.5000 - val_root_mean_squared_error: 2472.8223
Epoch 6/100
1610/1610 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 6051137.0000 - root_mean_squared_error: 2459.5840 - val_loss: 579
0528.5000 - val_root_mean_squared_error: 2406.0515
Epoch 7/100
1610/1610 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 5782841.0000 - root_mean_squared_error: 2404.0959 - val_loss: 593
0757.0000 - val_root_mean_squared_error: 2434.9834
Epoch 8/100
1610/1610 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 5520663.0000 - root_mean_squared_error: 2348.9062 - val_loss: 581
8895.0000 - val_root_mean_squared_error: 2411.8682
Epoch 9/100
1610/1610 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 5800606.5000 - root_mean_squared_error: 2407.6091 - val_loss: 564
4052.0000 - val_root_mean_squared_error: 2375.3123
Epoch 10/100
1610/1610 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - loss: 5601505.5000 - root_mean_squared_error: 2365.9141 - val_loss: 566
6425.0000 - val_root_mean_squared_error: 2379.9866
Epoch 11/100
1610/1610 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 5350460.0000 - root_mean_squared_error: 2312.3503 - val_loss: 616
4485.0000 - val_root_mean_squared_error: 2482.3831
Epoch 12/100
1610/1610 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 5350546.0000 - root_mean_squared_error: 2311.8291 - val_loss: 559
9835.0000 - val_root_mean_squared_error: 2365.8860
Epoch 13/100
1610/1610 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 5199671.5000 - root_mean_squared_error: 2279.4956 - val_loss: 566
8199.5000 - val_root_mean_squared_error: 2380.2583
Epoch 14/100
1610/1610 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - loss: 5151341.0000 - root_mean_squared_error: 2268.6167 - val_loss: 572
0488.5000 - val_root_mean_squared_error: 2391.1880
Epoch 15/100
1610/1610 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 5030883.0000 - root_mean_squared_error: 2241.3955 - val_loss: 561
8278.0000 - val_root_mean_squared_error: 2369.6858
Epoch 16/100
1610/1610 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 5044054.5000 - root_mean_squared_error: 2244.9644 - val_loss: 552
9882.5000 - val_root_mean_squared_error: 2350.9314
Epoch 17/100
1610/1610 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 5134553.5000 - root_mean_squared_error: 2264.9336 - val_loss: 544
6362.5000 - val_root_mean_squared_error: 2333.0720
Epoch 18/100
1610/1610 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - loss: 4979764.5000 - root_mean_squared_error: 2230.6987 - val_loss: 556
2031.5000 - val_root_mean_squared_error: 2357.7002
Epoch 19/100
1610/1610 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 4961063.0000 - root_mean_squared_error: 2226.1975 - val_loss: 550
2877.0000 - val_root_mean_squared_error: 2345.0908
Epoch 20/100
1610/1610 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 4822994.5000 - root_mean_squared_error: 2194.6350 - val_loss: 558
9971.5000 - val_root_mean_squared_error: 2363.5586
Epoch 21/100
1610/1610 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 4784075.0000 - root_mean_squared_error: 2185.8489 - val_loss: 562
0366.5000 - val_root_mean_squared_error: 2369.9512
Epoch 22/100
1610/1610 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 4802096.0000 - root_mean_squared_error: 2190.0269 - val_loss: 593
4297.5000 - val_root_mean_squared_error: 2435.2585
Epoch 23/100
1610/1610 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 4697524.5000 - root_mean_squared_error: 2165.8979 - val_loss: 552
9118.5000 - val_root_mean_squared_error: 2350.5698
Epoch 24/100
1610/1610 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - loss: 4748672.5000 - root_mean_squared_error: 2178.0491 - val_loss: 546
5296.5000 - val_root_mean_squared_error: 2336.9292
Epoch 25/100
1610/1610 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 4713213.0000 - root_mean_squared_error: 2169.8535 - val_loss: 559
1959.5000 - val_root_mean_squared_error: 2363.8440
Epoch 26/100
1610/1610 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 4771773.5000 - root_mean_squared_error: 2183.2046 - val_loss: 585
0877.0000 - val_root_mean_squared_error: 2417.9639
Epoch 27/100
1610/1610 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 4540625.5000 - root_mean_squared_error: 2129.4766 - val_loss: 550
0878.5000 - val_root_mean_squared_error: 2344.4456
448/448 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step
Root Mean Squared Error: 2380.489974276855
```

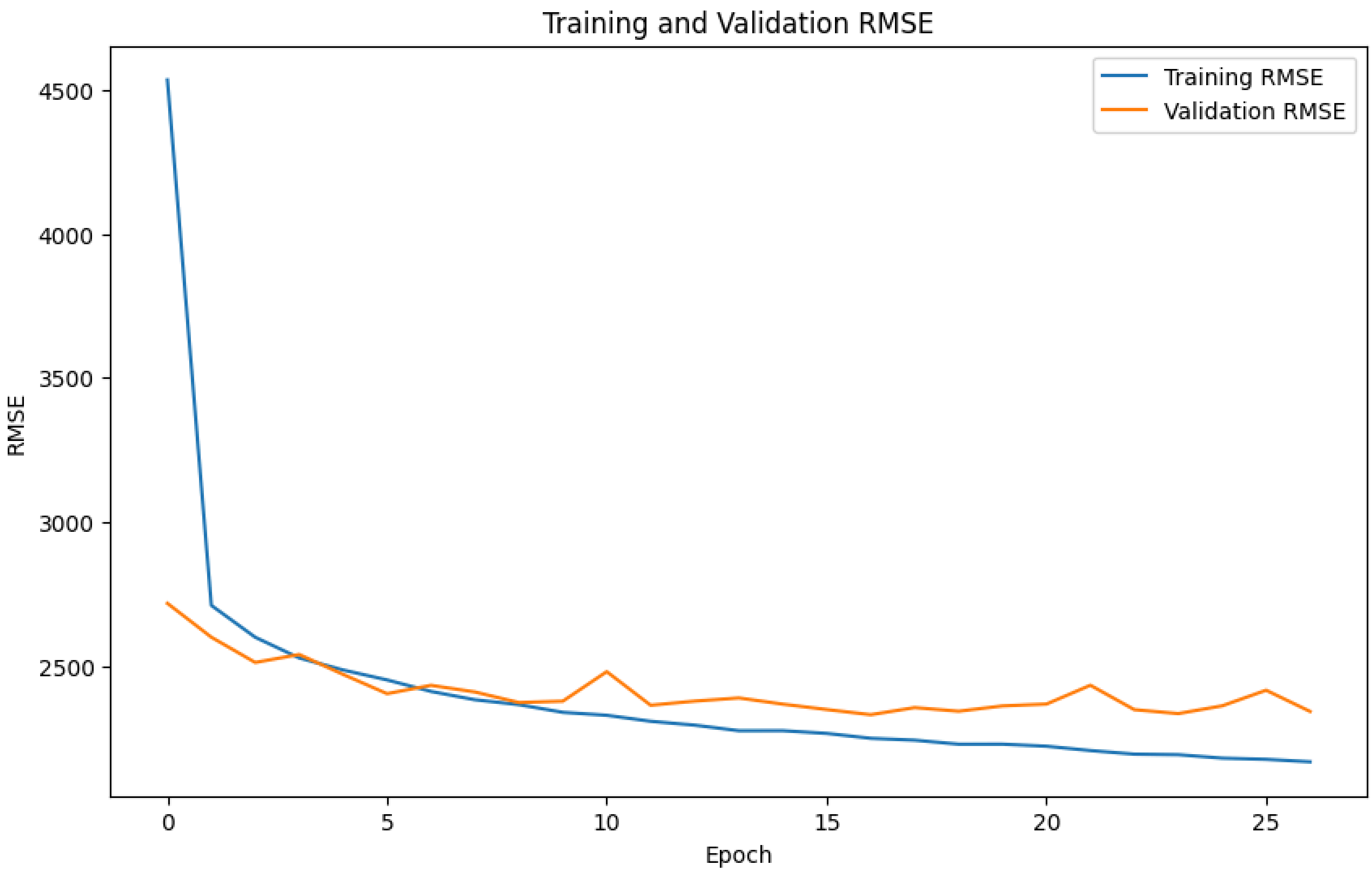
```
/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-packages/sklearn/metrics/_regression.py:483:
FutureWarning: 'squared' is deprecated in version 1.4 and will be removed in 1.6. To calculate the root mean squared
error, use the function'root_mean_squared_error'.
  warnings.warn(
```

In [64]:
```
# Predict on the testing data
y_pred_nn = final_nn_model.predict(X_test_processed)

# Evaluate the model
mse_test_nn = mean_squared_error(y_test, y_pred_nn)
print(f'Mean Squared Error: {mse_test_nn:.2f}', '\n',
      f'Root Mean Squared Error: {np.sqrt(mse_test_nn):.2f}')
```

```
448/448 ━━━━━━━━━━━━━━━━━━━━ 0s 541us/step
Mean Squared Error: 5666732.52
 Root Mean Squared Error: 2380.49
```

Visualizing the Training and Validation RMSE

In [65]:
```
# accessing the log file
log_file = pd.read_csv(path + '/final_nn_training_logs.csv')

# plotting the training and validation loss
plt.figure(figsize=(10, 6))
plt.plot(log_file['epoch'], log_file['root_mean_squared_error'], label='Training RMSE')
plt.plot(log_file['epoch'], log_file['val_root_mean_squared_error'], label='Validation RMSE')
plt.xlabel('Epoch')
plt.ylabel('RMSE')
plt.title('Training and Validation RMSE')
plt.legend()
plt.show()
```

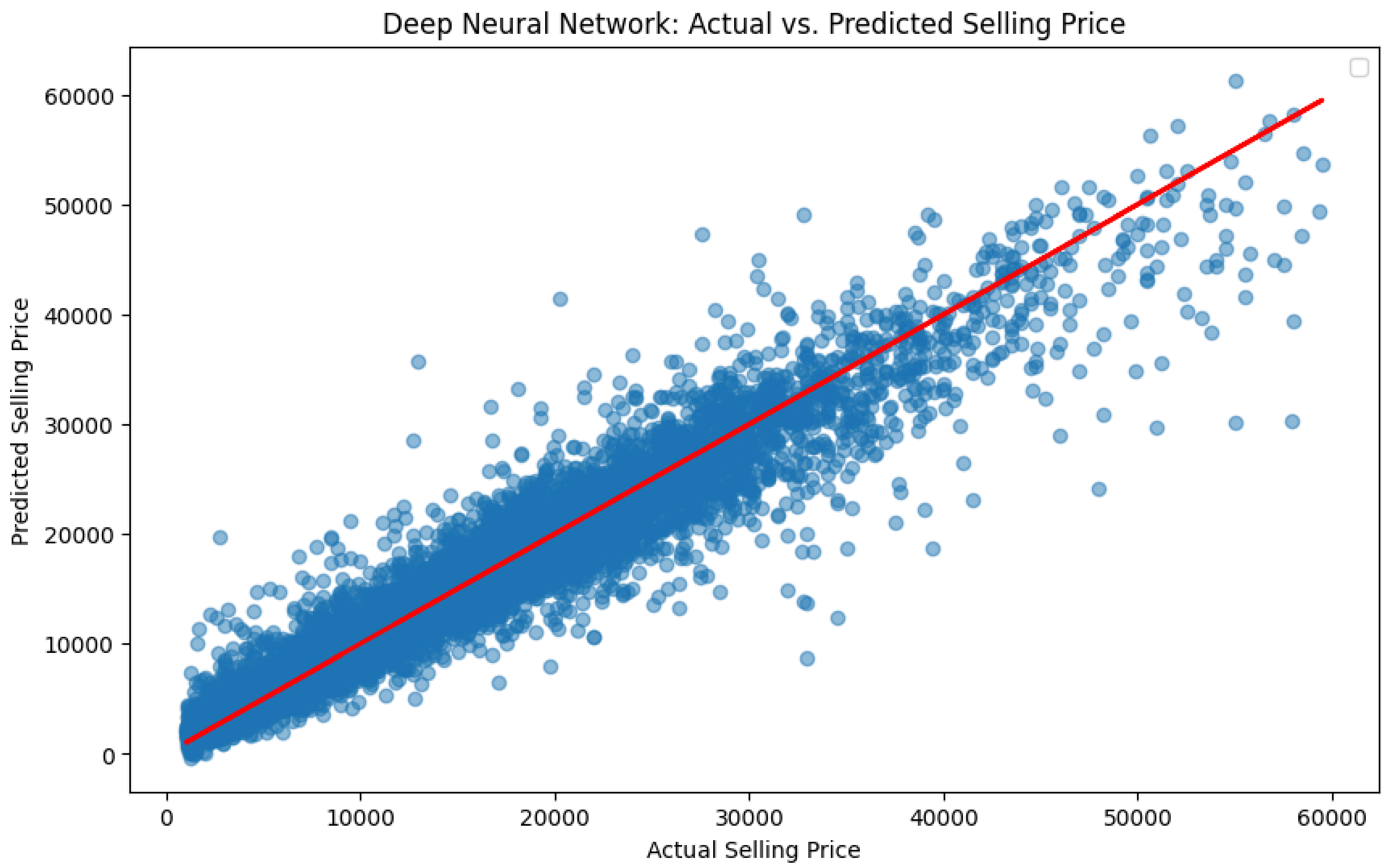


The plot above shows us that our training RMSE drops significatntly within 3 epochs, while our validation RMSE stays relatively low and starts to increase around epoch 20. Having included the callback, `early_stopping`, the model knows when to stop training based off the validation RMSE.

Visualizing the Truth Vs. Predictions

In [67]:
```
plt.figure(figsize=(10, 6))
plt.scatter(y_test, y_pred_nn, alpha=0.5)
plt.plot(y_test, y_test, color='red', linewidth=2)
plt.xlabel('Actual Selling Price')
plt.ylabel('Predicted Selling Price')
plt.title('Deep Neural Network: Actual vs. Predicted Selling Price')
plt.legend()
plt.show()
```

```
/var/folders/xh/4xpwy9dj7glg3lpzyn0wwqgw0000gn/T/ipykernel_4991/1656358250.py:7: UserWarning: No artists with labels
found to put in legend.  Note that artists whose label start with an underscore are ignored when legend() is called
with no argument.
  plt.legend()
```



Results

The plot above shows the fitting of the trained neural network predictions on the testing data. We see that the data fits much better to the true selling prices of the cars at auction. As we start to increase in price, the model ended up training better to the dataset than before, improving our results from the XGBoost Model from earlier.

Conclusion

In this project, I successfully developed a XGBoost, and deep neural network model to predict the selling prices of cars at auction. Through careful data preprocessing and model training, we observed a significant reduction in training RMSE within the first few epochs, indicating that our model was learning effectively. The validation RMSE, however, began to increase after roughly 15-20 epochs, suggesting potential overfitting, which we addressed using the early stopping callback. Similar preventions to overfitting in the XGBoost model were taken, comparing the validation set and training performance.

The visualizations of actual versus predicted selling prices demonstrated that our models performed well. Particularly at higher price points, the Deep Neural Net outperformed the XGBoost model. This indicates that the neural network is capable of capturing complex patterns in the data, leading to more accurate predictions.

Overall, this project highlights the effectiveness of deep learning techniques in the domain of car auction price prediction and opens avenues for further exploration, such as hyperparameter tuning and the inclusion of additional features to enhance model performance.