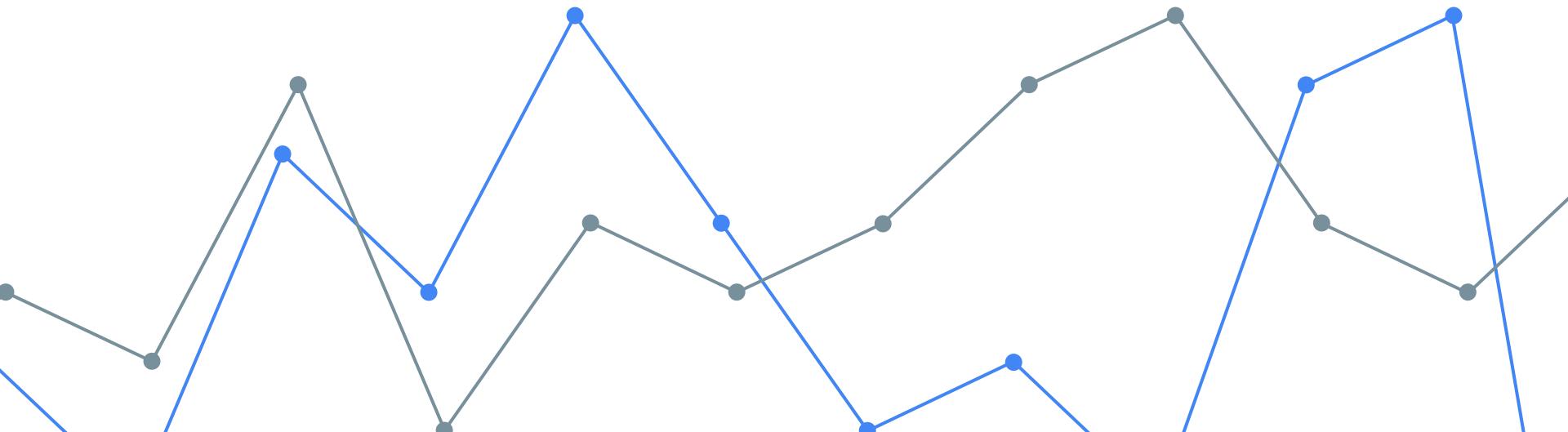


# Neural Networks vs. Classical Methods

By Charles Dilger & Ahsan Imran



# Dataset | NY Housing

Our first dataset will be on housing in NY.

It comes from the Federal Reserve Economic Data (FRED) API.

Our target variable will be average housing price.

It will be an engineered feature, taking the mean of all cities' housing prices.

This dataset contains 12 rows that are missing values.

It contains features such as the date, cities' housing prices, mortgage rate, NY housing permits, NY price growth, and more.

AIM 460 Project 3 - Dataset 1: NY Housing Data  
Source: FRED API (Federal Reserve Economic Data)

Loading New York Housing Data from FRED API...

Fetching 10 NY housing metrics...

- ny\_house\_price\_index: 202 records
- ny\_housing\_permits: 451 records
- ny\_housing\_starts: 451 records
- albany\_house\_price: 183 records
- buffalo\_house\_price: 192 records
- rochester\_house\_price: 188 records
- syracuse\_house\_price: 189 records
- ny\_median\_income: HTTP 400
- ny\_unemployment\_rate: 595 records
- mortgage\_rate\_30yr: 2843 records
- Total NY records collected: 5294

Transforming NY data for machine learning...

Creating NY-specific features...

- NY dataset ready: 3363 rows, 26 features

# Dataset | ADRE ETF

Our second database is the ADRE ETF, available at Kaggle's Huge Stock Market Dataset.

It is the value of BLDRS Emerging Markets 50 ADR Index Fund.

It contains the opening value, closing value, high, low, volume, and return.

We will try to predict the closing value, based on the other features.



```
🚀 Loading Dataset 2: ADRE ETF Stock Data
Target: Close Price (Stock Price Prediction)
🕒 Loading ADRE ETF Stock Data from GitHub...
    📁 Downloading ADRE ETF data...
    ✅ ADRE data loaded: 3201 records
🕒 Transforming ADRE data for ML...
    🚀 Creating ADRE stock features...
    ✅ ADRE dataset ready: 3201 rows, 25 features

=====
🕒 ADRE ETF DATA VERIFICATION (Dataset 2)
=====
Dataset Shape: 3201 rows x 25 columns

🕒 REQUIREMENTS:
≥3,000 examples: ✅ PASS (3,201 rows)
≥5 features: ✅ PASS (25 features)
Download source: ✅ PASS (GitHub)
Different domain: ✅ PASS (Stocks vs Housing)
```

# NY Housing EDA Pt.1

## Result 1

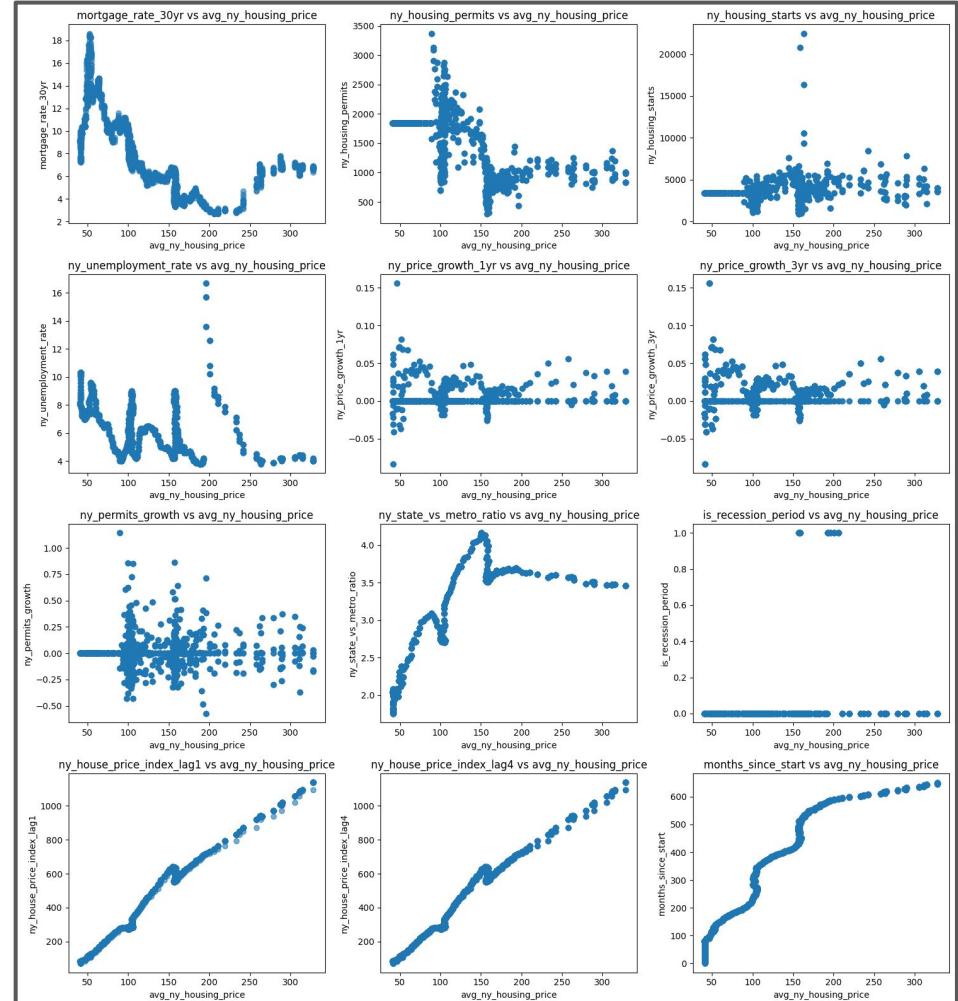
Engineered a target feature called Average NY Housing Price, that takes the mean of all cities' housing prices.

## Result 2

Engineered a feature called Months Since Start, which counts the months past since the beginning of the data.

## Result 3

12 rows were removed for having missing values, leaving 3350 rows



# NY Housing EDA Pt.2

From then on, we:

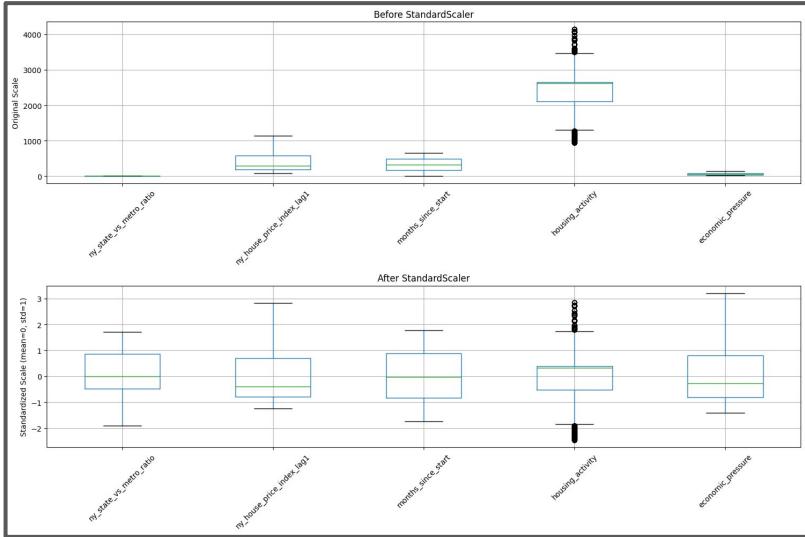
- Removed seemingly redundant features.
- Applied IQR in order to remove outliers.
- Checked the VIF score, found a score as high as 81.
- Decided to engineer 2 new features:
  - Housing Activity, the combination of NY Housing Permits and NY Housing Starts
  - Economic Pressure, the combination of NY Unemployment Rate and 30-Year Mortgage Rate

```
Combining highly correlated features to reduce multicollinearity...
Created: housing_activity (average of permits and starts)
Created: economic_pressure (unemployment × mortgage rate)

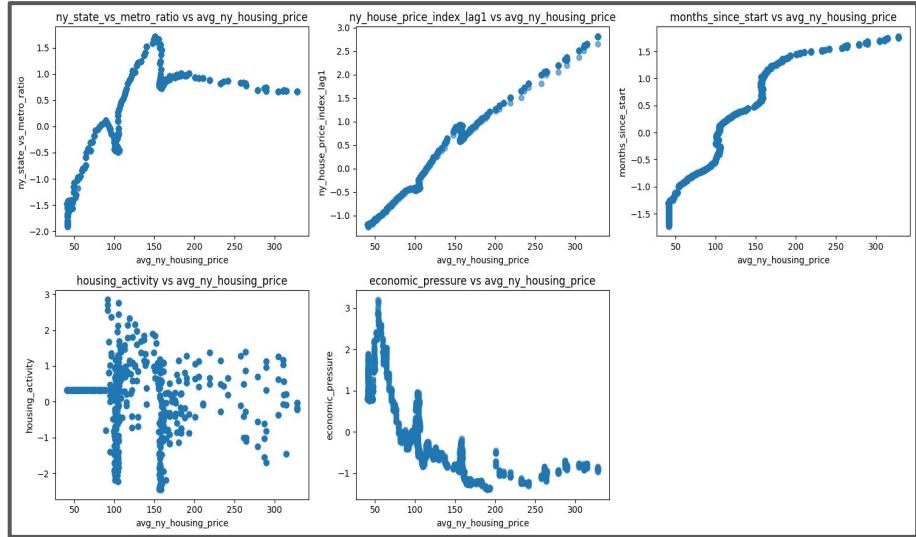
Removing original features: ['ny_housing_permits', 'ny_housing_starts', 'ny_unemployment_rate', 'mortgage_rate_30yr']

Final feature set (5 features):
1. ny_state_vs_metro_ratio
2. ny_house_price_index_lag1
3. months_since_start
4. housing_activity
5. economic_pressure
```

# NY Housing StandardScaler

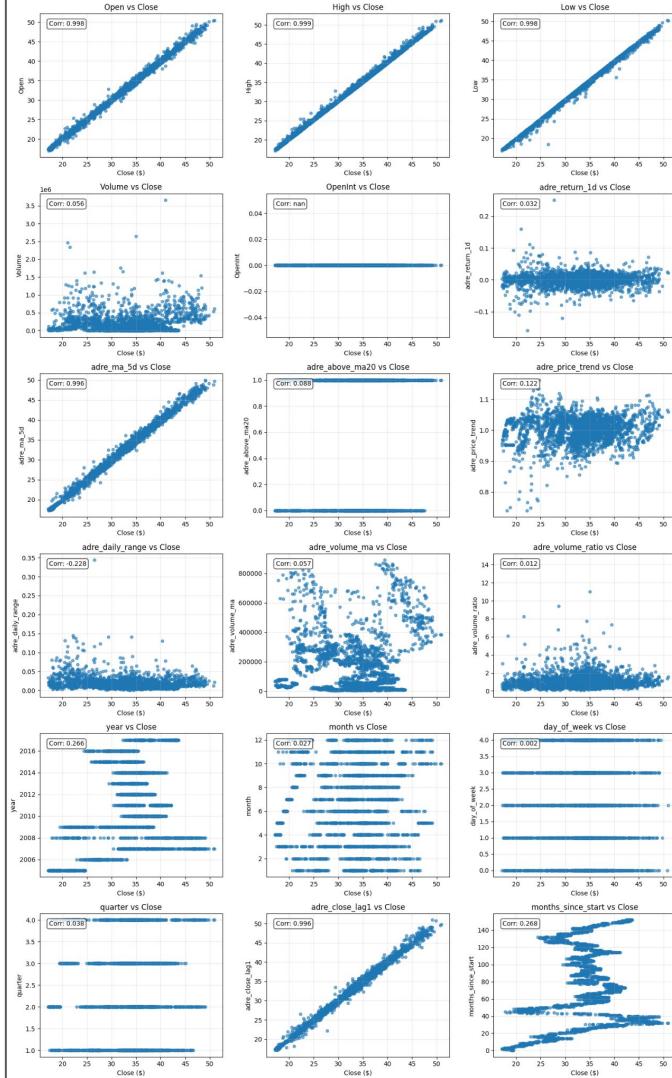


## Pre-scaling Scatterplot



# ADRE ETF Dataset Pt.1

No missing values were found.  
We removed a few redundant features right away, such as delayed returns and lagging variables, that would likely have extreme multicollinearity.  
After plotting, we removed weak correlations and time bands.



# ADRE ETF Dataset Pt.2

Volume had a skew, so we applied a logarithmic transformation to it.

Our VIF scores were insanely high, so we removed adre\_close\_lag1 and adre\_ma\_5d. We left Open because we wanted to predict the closing stock value based primarily on the opening stock value. It is the most important part of our models.

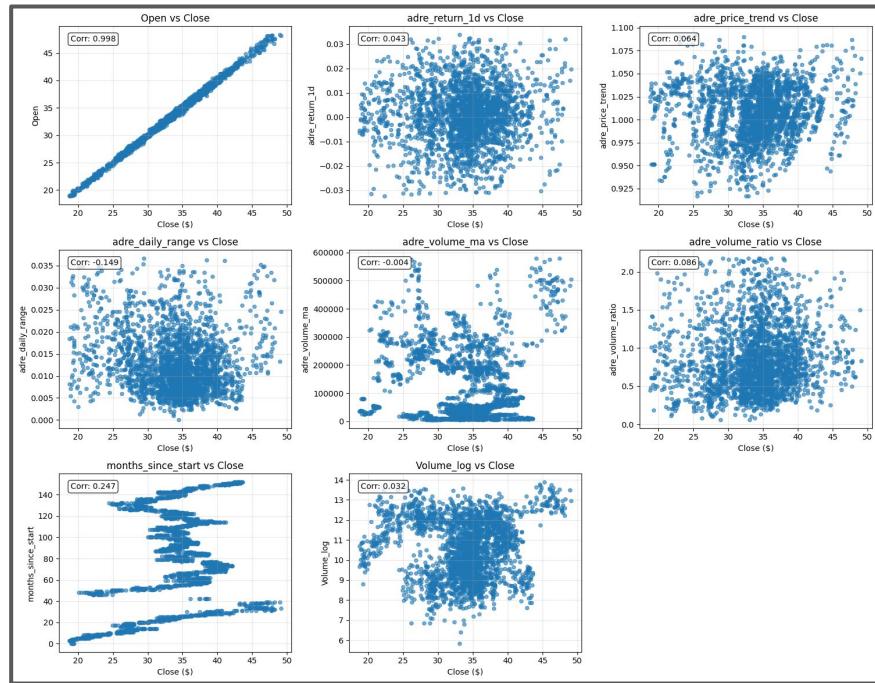
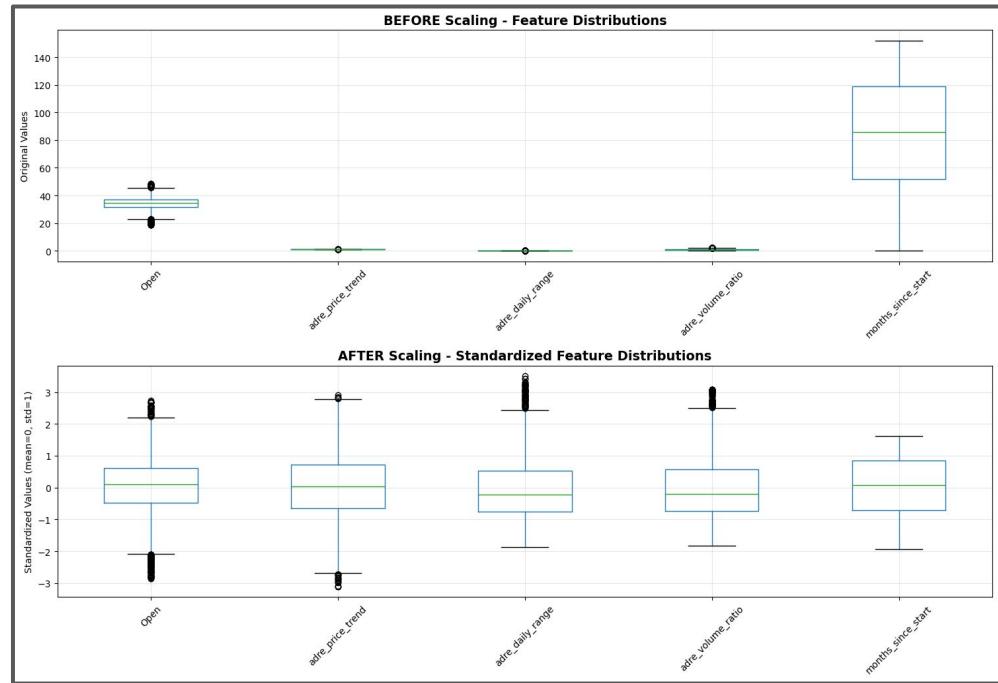
We then removed poorly correlated features.

VIF Scores (>10 indicates high multicollinearity):

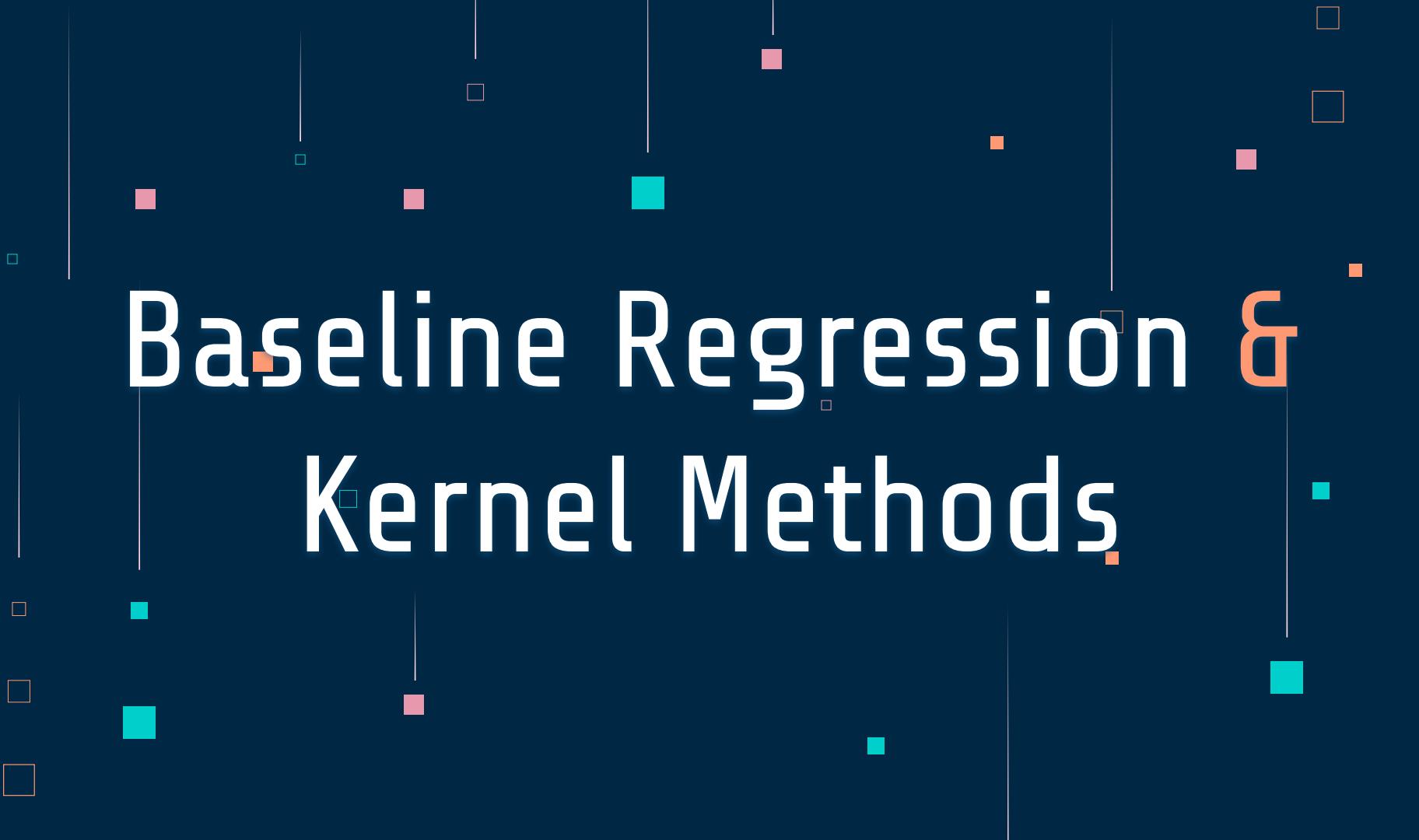
adre_close_lag1	:	36152.25	HIGH
Open	:	25321.53	HIGH
adre_ma_5d	:	11471.20	HIGH
Volume_log	:	451.28	HIGH
adre_price_trend	:	365.00	HIGH
months_since_start	:	24.62	HIGH
adre_volume_ratio	:	8.91	MODERATE
adre_daily_range	:	6.95	MODERATE
adre_volume_ma	:	6.93	MODERATE
adre_return_1d	:	1.85	LOW

# ADRE StandardScaler

# Pre-scaling Scatterplot

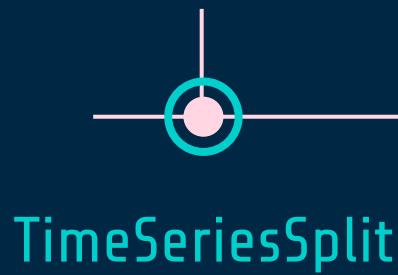


# Baseline Regression & Kernel Methods



# Evaluation Setup

5 fold - since data is chronological



## Pipeline

Impute → Standardize  
→ Model

OLS, Ridge, LASSO,  
Elastic Net, Kernel  
Ridge



## Models

MSE (primary), plus  
RMSE & MAE

## Metrics

# Linear Baseline

<u>Model</u>	<u>What it does</u>	<u>How we tuned</u>	<u>When it helps</u>
OLS	Reference fit & No Penalty	-	Quick floor to beat
Ridge	Shrinks all coeffs	$\alpha \in \{0.01, 0.1, 1, 10, 100\}$	Features correlated
LASSO	Drives Coeff to 0	$\alpha \in \{0.0005, 0.001, 0.01, 0.1, 1\}$	feature selection via sparsity
Elastic Net	Balance of shrink & selection	$\alpha \in \{0.001, 0.01, 0.1, 1\}$ , $\text{l1\_ratio} \in \{0.1...0.9\}$	Correlated features + some pruning

# Linear Baselines Results



Using TimeSeriesSplit for housing data

Top 10 Linear Models:

	family	model	val_MSE	val_RMSE	val_MAE	fit_time_s	alpha
15	Linear	ElasticNet	487.505795	16.342489	12.919377	0.011601	0.0010
1	Linear	Ridge	499.175165	17.124968	13.385103	0.004056	0.0100
2	Linear	Ridge	500.678205	17.352979	13.761011	0.007477	0.1000
7	Linear	LASSO	507.594547	17.602253	13.778482	0.010070	0.0010
8	Linear	LASSO	512.510929	17.672512	14.039539	0.011123	0.0100
6	Linear	LASSO	512.975345	17.827460	13.955985	0.007461	0.0005
0	Linear	OLS	518.903167	18.053405	14.135162	0.006170	NaN
14	Linear	ElasticNet	515.261795	18.128285	14.495841	0.010756	0.0010
13	Linear	ElasticNet	534.980997	18.840802	15.147588	0.014883	0.0010
12	Linear	ElasticNet	548.841016	19.236168	15.527127	0.012880	0.0010

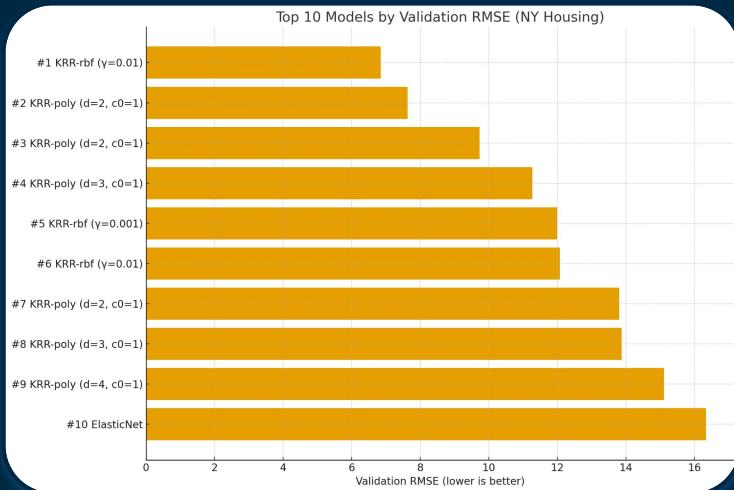
# Kernel Ridge

Since our data isn't Linear, Our dataset is benefiting from the kernel methods in comparison to our linear models

Our Top 5 Kernel Models:

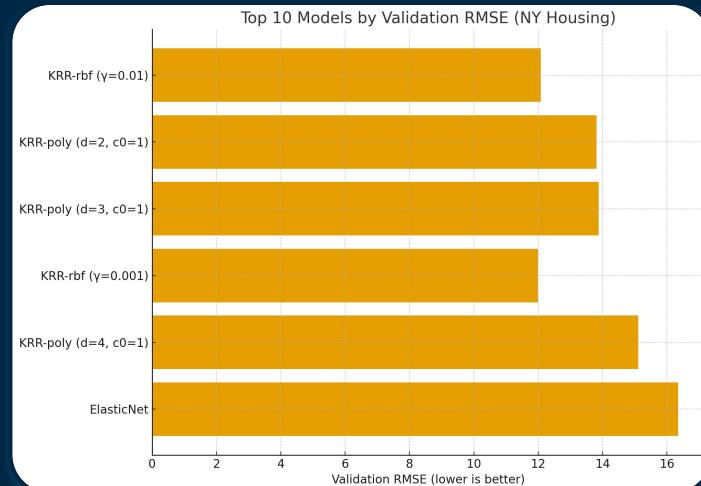
	family	model	alpha	gamma	val_MSE	val_RMSE	val_MAE
1	Kernel	KRR-rbf	0.001	0.010	88.991282	6.842321	5.061987
26	Kernel	KRR-poly	0.001	NaN	65.024493	7.628858	5.467877
32	Kernel	KRR-poly	0.010	NaN	135.964599	9.734397	6.869843
28	Kernel	KRR-poly	0.001	NaN	174.701644	11.271754	8.704205
0	Kernel	KRR-rbf	0.001	0.001	182.589075	11.993725	9.898700

# Top 10 Models by Validation RMSE



## KRR-RBF

Kernel Ridge with an RBF kernel clearly wins on validation with RMSE



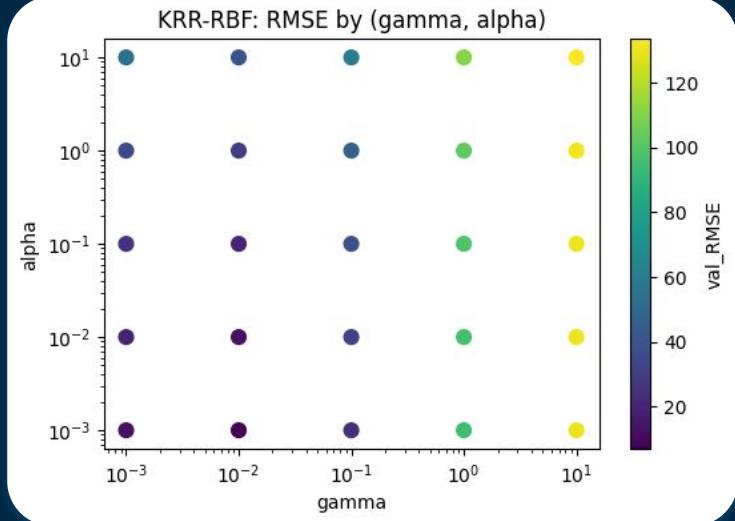
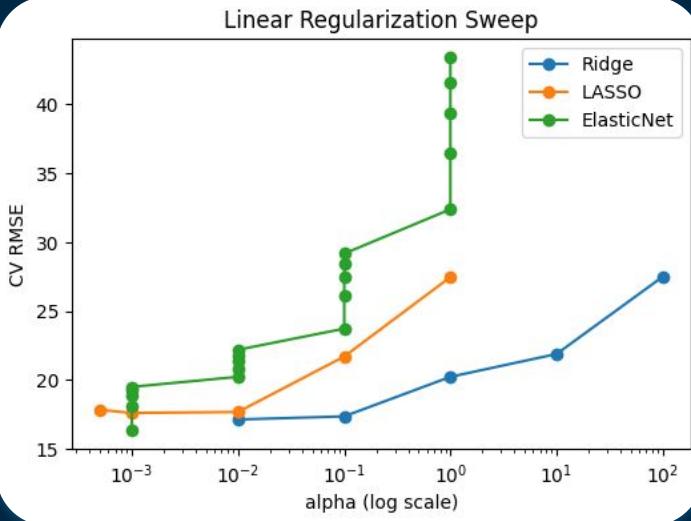
## What the data shows

Our recommended default RBF KRR with a quick  $\gamma$  sweep around 0.001-0.01

# Regularization Sweep

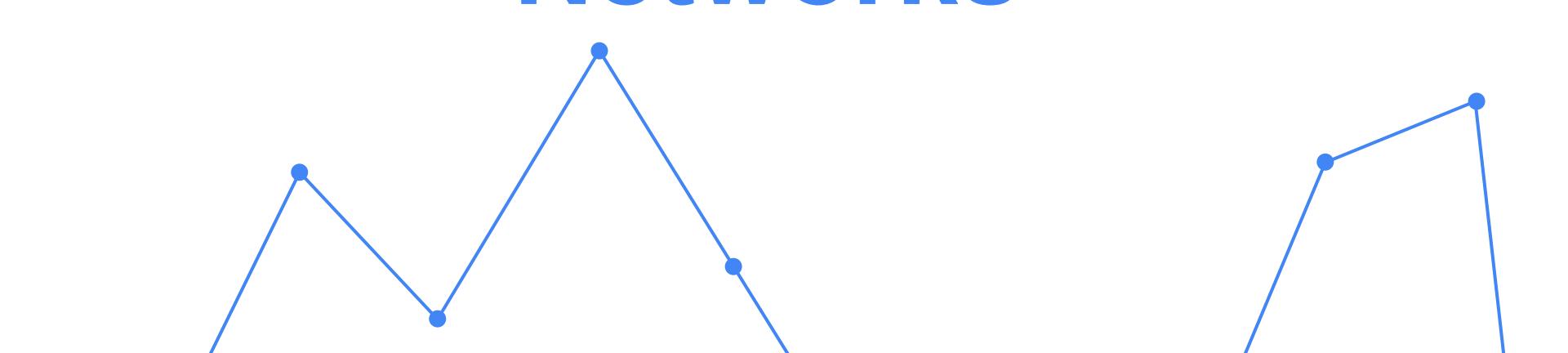
&

# KRR-RBF Landscape





# Feedforward Neural Networks



# Shallow Neural Network

Our first neural network was a shallow neural network.

We did a baseline neural network of 64 neurons, and then we did an enhanced version with batch normalization, a 20% dropout rate, and replacing ReLU with Swish.

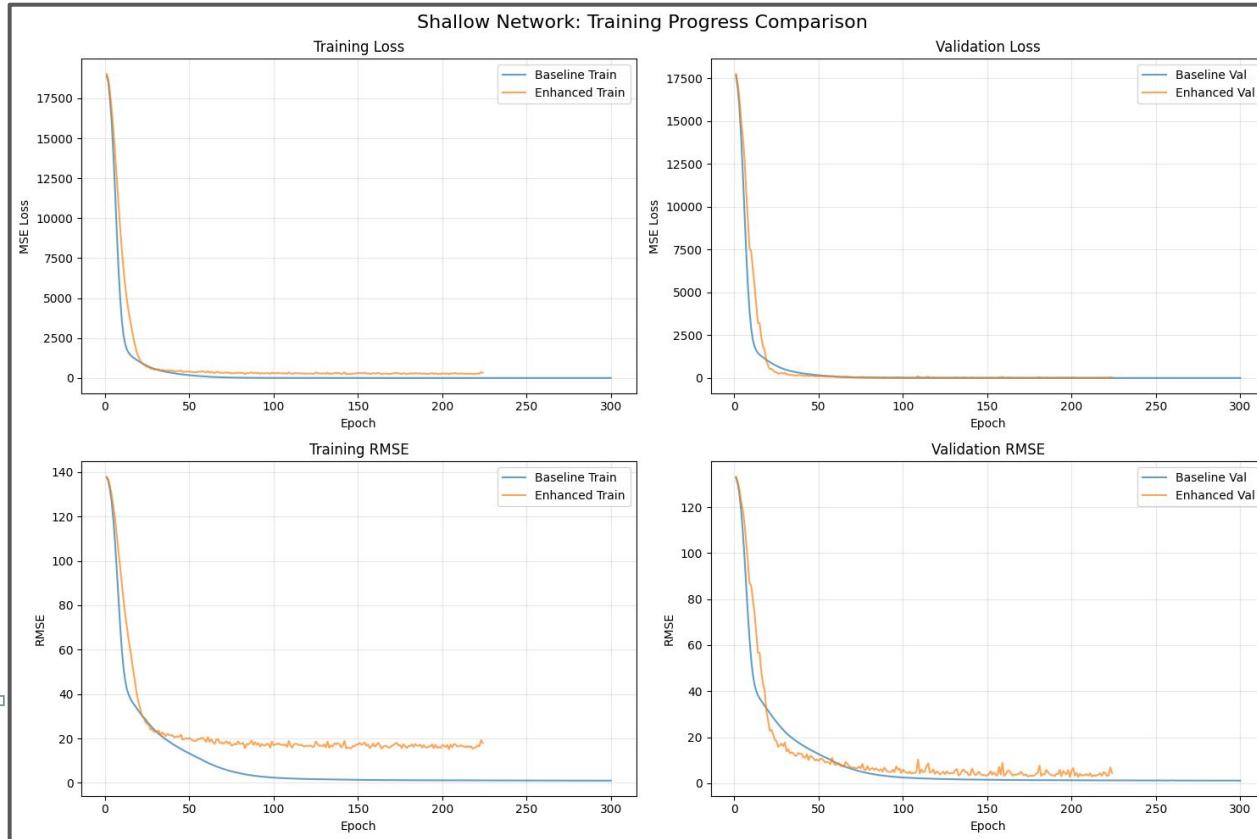
The baseline outperformed, with a RMSE of 0.9466 compared to the enhanced's 4.5857.

```
== Shallow Network Architecture ==
Baseline Shallow Net:
ShallowNet(
    (hidden): Linear(in_features=5, out_features=64, bias=True)
    (output): Linear(in_features=64, out_features=1, bias=True)
)
Parameters: 449

Enhanced Shallow Net:
ShallowNet(
    (hidden): Linear(in_features=5, out_features=64, bias=True)
    (output): Linear(in_features=64, out_features=1, bias=True)
    (batch_norm): BatchNorm1d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (dropout): Dropout(p=0.2, inplace=False)
)
Parameters: 577

Forward pass test:
Input shape: torch.Size([32, 5])
Baseline output shape: torch.Size([32, 1])
Enhanced output shape: torch.Size([32, 1])
✓ Architecture working correctly!
```

# Shallow Neural Network - Results



# Deep Neural Network

We repeated the process with a deep neural network. This model had 4 layers; they were 128, 64, 32, and 16 neurons, respectively. The baseline got a RMSE of 1.0305, worse than the shallow neural network.

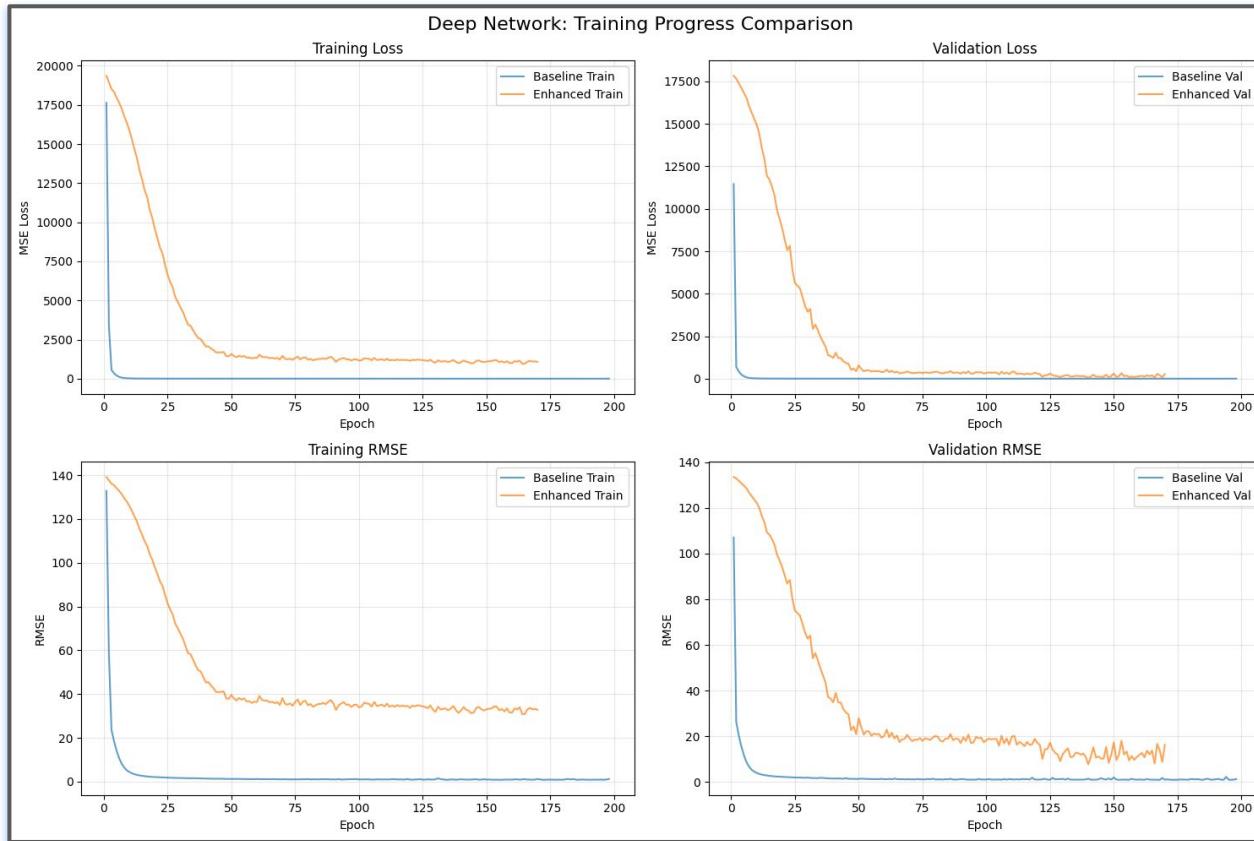
We tried an enhanced version, with Mish activation, batch normal layers, and 30% dropout.

This performed significantly worse due to over-regularization. It had a RMSE of 16.8254.

```
== Deep Network Architecture ==
Baseline Deep Net:
DeepNet(
    (hidden_layers): ModuleList(
        (0): Linear(in_features=5, out_features=128, bias=True)
        (1): Linear(in_features=128, out_features=64, bias=True)
        (2): Linear(in_features=64, out_features=32, bias=True)
        (3): Linear(in_features=32, out_features=16, bias=True)
    )
    (batch_norm_layers): ModuleList()
    (output): Linear(in_features=16, out_features=1, bias=True)
)
Parameters: 11,649
```

Architecture Summary:  
Hidden layers: 4 (meets ≥4 requirement)  
Layer sizes: 5 → 128 → 64 → 32 → 16 → 1  
 Deep architecture working correctly!

# Deep Neural Network - Results



# Wide Neural Network

This time, we tried it with a model that had 1 hidden layer of 512 neurons.

This performed shockingly well, with a RMSE of only 0.8031, our lowest yet.

We tried an enhanced version, as well. It had Swish activation, batch norm, 40% dropout. As expected, this performed worse, with a RMSE of 3.8199. It was still the best performer of all enhanced models, further asserting that a wide neural network is the best fit for our dataset.

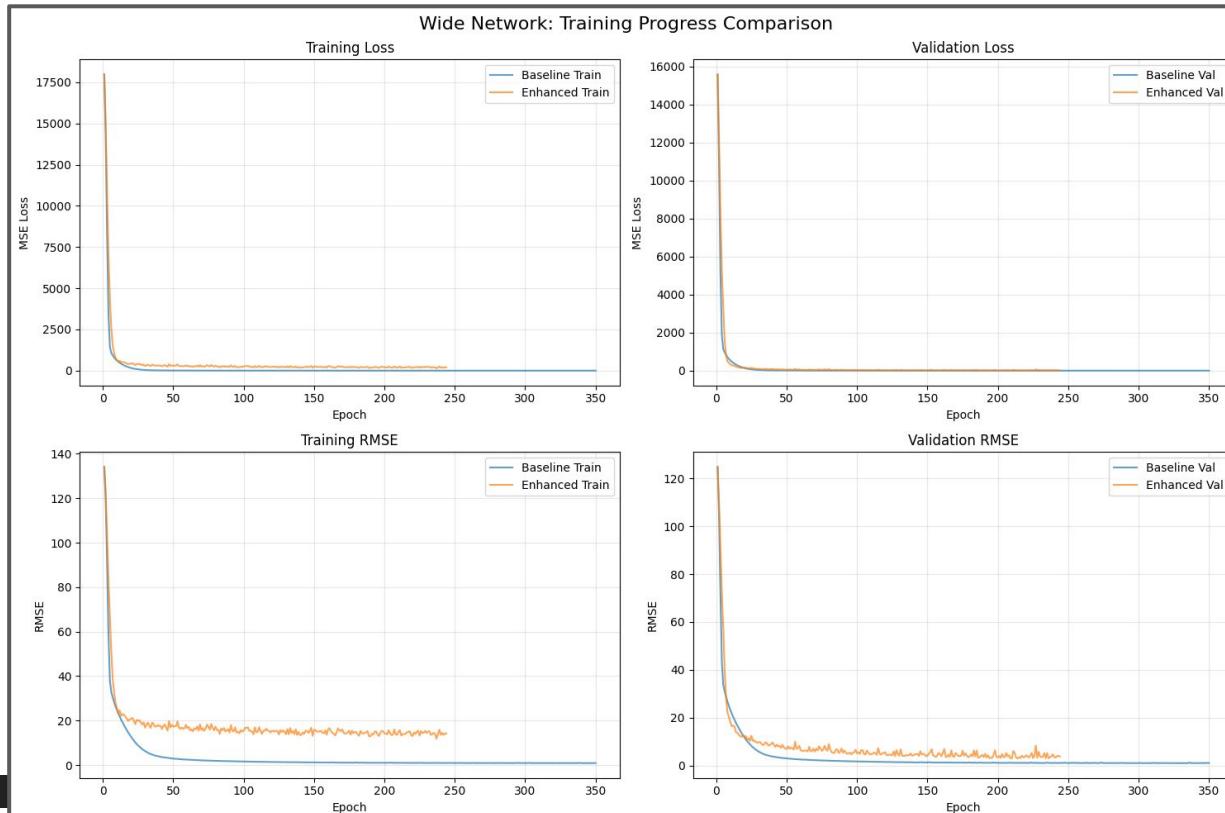
Training complete! Time: 38.94s  
 Best validation RMSE: 0.9822

Wide Baseline Results:  
Test RMSE: 0.8031  
Test MAE: 0.5159  
vs Shallow Baseline: 0.8031 vs 0.9466

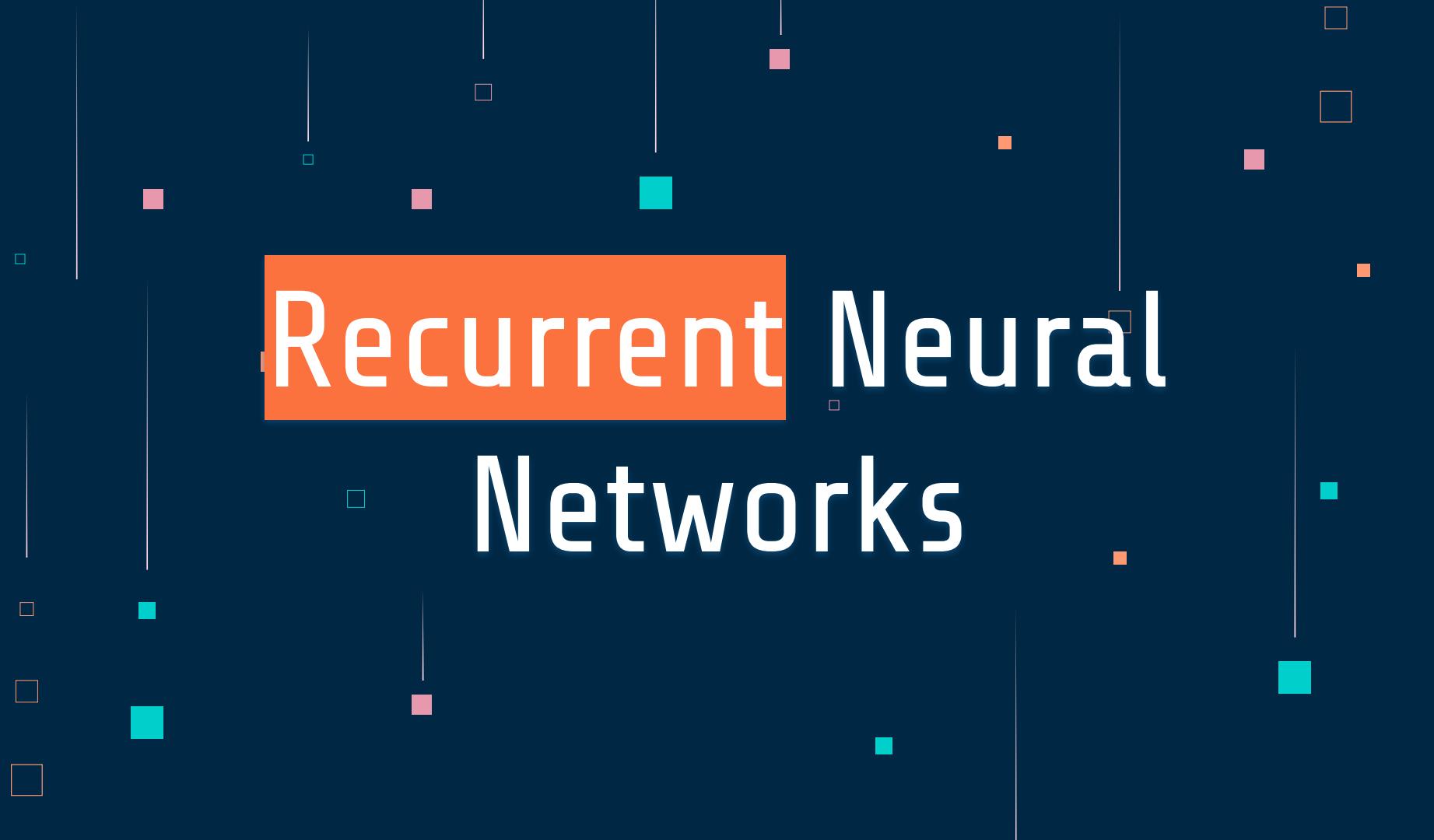
Training complete! Time: 46.67s  
 Best validation RMSE: 2.9480

Wide Enhanced Results:  
Test RMSE: 3.8199  
Test MAE: 3.2305  
vs Shallow Baseline: 3.8199 vs 0.9466

# Wide Neural Network Results



# Recurrent Neural Networks



# Recurrent Neural Networks

For our RNN, we will be using our NY Housing data. It is chronologically ordered, had explicit time progression, has lag features, momentum, and cycles. It is an ideal dataset for running an RNN.

We will be using LSTM instead of GRU, due to long-term memory needs, and the complex relationships. LSTM is better at learning long-term, complex patterns.

We first implemented Sliding Window Transformation to convert our chronological housing data into sequences.

```
=====
NY HOUSING DATA → LSTM SEQUENCES
=====

Input data verification:
Features (X): (2966, 5)
Target (y): (2966,)
Data type: Time series (chronologically ordered)
Creating sequences from time series data...
Original data shape: (2966, 5)
Sequence length: 12 time steps
Stride: 1
Sequences created!
Sequence shape: (2954, 12, 5)
Target shape: (2954,)
Total sequences: 2,954
Data transformation summary:
Original samples: 2,966
Sequence samples: 2,954
Data loss: 0.4% (due to sequence creation)

Splitting sequences temporally (preserving time order)...
Split sizes:
Train: 1,772 sequences (60.0%)
Val: 591 sequences (20.0%)
Test: 591 sequences (20.0%)
LSTM data preparation complete!

Sequence Structure Analysis:
Input sequence shape: (2954, 12, 5)
- Dimension 0: 2,954 sequences
- Dimension 1: 12 time steps (lookback)
- Dimension 2: 5 features per time step

Example sequence:
Sequence 0 input shape: (12, 5) (12 months x 5 features)
Sequence 0 target: 41.3800 (price to predict)

Testing LSTM DataLoader:
Batch input shape: torch.Size([32, 12, 5]) # (batch_size, sequence_length, features)
Batch target shape: torch.Size([32, 1]) # (batch_size, 1)

Sequence preparation complete!
Variables created:
• X_sequences, y_sequences: Raw sequence arrays
• lstm_train/val/test_loader: PyTorch DataLoaders
• lstm_data_info: Metadata for LSTM training

Ready to build LSTM architecture!
```

# Recurrent Neural Networks

For our RNN, we will be using our NY Housing data. It is chronologically ordered, had explicit time progression, has lag features, momentum, and cycles. It is an ideal dataset for running an RNN.

We will be using LSTM instead of GRU, due to long-term memory needs, and the complex relationships. LSTM is better at learning long-term, complex patterns.

We first implemented Sliding Window Transformation to convert our chronological housing data into sequences.

```
=====
NY HOUSING DATA → LSTM SEQUENCES
=====

Input data verification:
Features (X): (2966, 5)
Target (y): (2966,)
Data type: Time series (chronologically ordered)
Creating sequences from time series data...
Original data shape: (2966, 5)
Sequence length: 12 time steps
Stride: 1
Sequences created!
Sequence shape: (2954, 12, 5)
Target shape: (2954,)
Total sequences: 2,954
Data transformation summary:
Original samples: 2,966
Sequence samples: 2,954
Data loss: 0.4% (due to sequence creation)

Splitting sequences temporally (preserving time order)...
Split sizes:
Train: 1,772 sequences (60.0%)
Val: 591 sequences (20.0%)
Test: 591 sequences (20.0%)
LSTM data preparation complete!

Sequence Structure Analysis:
Input sequence shape: (2954, 12, 5)
- Dimension 0: 2,954 sequences
- Dimension 1: 12 time steps (lookback)
- Dimension 2: 5 features per time step

Example sequence:
Sequence 0 input shape: (12, 5) (12 months x 5 features)
Sequence 0 target: 41.3800 (price to predict)

Testing LSTM DataLoader:
Batch input shape: torch.Size([32, 12, 5]) # (batch_size, sequence_length, features)
Batch target shape: torch.Size([32, 1]) # (batch_size, 1)

Sequence preparation complete!
Variables created:
• X_sequences, y_sequences: Raw sequence arrays
• lstm_train/val/test_loader: PyTorch DataLoaders
• lstm_data_info: Metadata for LSTM training

Ready to build LSTM architecture!
```

# Recurrent Neural Networks Pt.2

We ended up with a shocking 51,521 parameters to work with in our RNN model.

This added complexity backfired, as our model was gravely overfitted.

We ended up with a RMSE of 5.0388, much higher than our greatest performing neural network.

It even took much longer to run the model.

```
=====
# LSTM ARCHITECTURE TESTING
=====

# LSTM Model Information:
architecture: LSTM
input_size: 5
hidden_size: 64
num_layers: 2
dropout: 0.1
total_parameters: 51521
trainable_parameters: 51521

# Forward Pass Test:
Input shape: torch.Size([32, 12, 5]) # (batch=32, seq=12, features=5)
Target shape: torch.Size([32, 1])
Output shape: torch.Size([32, 1]) # (batch=32, 1)
✓ LSTM architecture working correctly!

# Model Complexity Comparison:
Model      Parameters      Type
Wide MLP      3,585      Feedforward
LSTM         51,521      Recurrent

⌚ Ready to train LSTM and compare with MLP champion (0.8031 RMSE)!
```



# Feature-Transfer Experiment

# Feature-Transfer Experiment

We checked whether or not the learned representations of our best performer, the wide baseline network, can outperform our original features. We tested this on both the same domain, our housing dataset, and a different domain, our ADRE stock.

Our model had remarkable ability to generalize. The neural features we extracted from the housing-trained model improved the prediction power of our stock by 16.1-37.3% when implemented on our ADRE model. The model performed excellently on ADRE, maintaining a RMSE of 0.1889 on the Kernel+Neutral learning test.

# Feature-Transfer Experiment output

## FEATURE-TRANSFER LEARNING EXPERIMENT

- Using Wide Baseline Network (0.8031 RMSE champion)
- Extracting 512-dimensional learned representations
- Comparing transfer performance vs original features

### Transfer Learning Experiment 1: NY Housing (same dataset)

#### TRANSFER LEARNING: NY HOUSING

- Extracting features from hidden layer...
  - Extracted features shape: (2966, 512)
  - Extracting features from hidden layer...
  - Extracted features shape: (2966, 512)
- Feature extraction summary:  
Source features: (2966, 512)  
Target features: (2966, 512)  
Original → Neural features: 5 → 512

Baseline: Direct regression on original NY Housing features  
Baseline RMSE:  $22.3422 \pm 26.3850$

Transfer Learning: Ridge on neural network features  
Transfer RMSE:  $5.9297 \pm 5.8843$

Advanced Transfer: RBF Kernel on neural features  
Kernel Transfer RMSE:  $99.2762 \pm 115.3576$

#### NY Housing Transfer Learning Results:

Method	RMSE	Improvement
--------	------	-------------

Original Features	22.3422	-
Neural Features	5.9297	+73.5%
Kernel+Neural	99.2762	-344.3%

- Transfer Learning Experiment 2: ADRE Stock (cross-domain)
- ADRE dataset: 2563 samples, 5 features

## TRANSFER LEARNING: ADRE STOCK

- Extracting features from hidden layer...
  - Extracted features shape: (2966, 512)
  - Extracting features from hidden layer...
  - Extracted features shape: (2563, 512)
- Feature extraction summary:  
Source features: (2966, 512)  
Target features: (2563, 512)  
Original → Neural features: 5 → 512

Baseline: Direct regression on original ADRE Stock features  
Baseline RMSE:  $0.3013 \pm 0.0000$

Transfer Learning: Ridge on neural network features  
Transfer RMSE:  $0.2527 \pm 0.0000$

Advanced Transfer: RBF Kernel on neural features  
Kernel Transfer RMSE:  $0.1889 \pm 0.0000$

#### ADRE Stock Transfer Learning Results:

Method	RMSE	Improvement
Original Features	0.3013	-
Neural Features	0.2527	+16.1%
Kernel+Neural	0.1889	+37.3%

## TRANSFER LEARNING CONCLUSIONS

- Same-Domain Transfer (NY Housing):
  - Neural features improve performance by 73.5%
  - The network learned useful representations beyond original features
- Cross-Domain Transfer (Housing → Stock):
  - Significant transfer! Neural features improve by 16.1%
  - Learned representations generalize across domains
- Feature-transfer experiment complete!
- Key insight: Transfer learning reveals whether neural networks learn generalizable representations beyond task-specific feature engineering