case_18.1

May 30, 2020

1 How do I predict the fair market prices of used cars?

[3]:
```
!pip install numpy scipy scikit-learn
```

```
Requirement already satisfied: numpy in
/root/.virtualenvs/ds4a-py3/lib/python3.6/site-packages (1.18.1)
Requirement already satisfied: scipy in
/root/.virtualenvs/ds4a-py3/lib/python3.6/site-packages (1.4.1)
Requirement already satisfied: scikit-learn in
/root/.virtualenvs/ds4a-py3/lib/python3.6/site-packages (0.22.1)
Requirement already satisfied: joblib>=0.11 in
/root/.virtualenvs/ds4a-py3/lib/python3.6/site-packages (from scikit-learn)
(0.14.1)
WARNING: You are using pip version 20.0.2; however, version 20.1.1 is
available.
You should consider upgrading via the '/root/.virtualenvs/ds4a-py3/bin/python -m
pip install --upgrade pip' command.
```

1.1 Introduction (5 mts)

Business Context. There are many companies selling used (refurbished) cars across the United States. As automobiles depreciate in value as they age, this is an extremely competitive industry, and they have to price the car right in order to win business. You are a data scientist tasked with building a predictive model for the prices of used car sales around the country. We have already seen some methods that will be useful for this purpose, such as a linear regression. In this case, we will look at another approach to this problem, called a **neural network**. Neural networks are the basic building block of **deep learning algorithms**.

Business Problem. Your task is to **predict the fair market price of a used car given its attributes**.

Analytical Context. The provided dataset on used cars was scraped from Craigslist. We have already pre-cleaned the dataset, by removing some outliers that are irrelevant, whittling down the features to relevant columns, and standardizing missing data that may cause trouble for our analysis.

The case will proceed as follows: we will (1) build a predictive model using linear regression; (2) discuss the challenges of feature engineering in order to implement regression in real-world contexts; (3) look at neural networks as an alternative to explicit feature engineering; and finaly (4) build a neural network to solve this problem.

1.2 Reading in the data (5 mts)

[5]:
```
import statsmodels.formula.api as sm
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

#from sklearn.preprocessing import Imputer
from sklearn.impute import SimpleImputer as Imputer
from sklearn.metrics import mean_squared_error as mse
```

[2]:
```
df = pd.read_csv('used_cars_clean.csv')
```

[3]:
```
print(df.columns)
df.head()
```

```
Index(['city', 'year', 'manufacturer', 'condition', 'cylinders', 'fuel',
       'odometer', 'title_status', 'transmission', 'drive', 'size', 'type',
       'paint_color', 'desc', 'price'],
      dtype='object')
```

[3]:

	city	year	manufacturer	condition	cylinders	fuel	odometer	title_status	transmission	drive	size	type	paint_color	desc	price
0	abilene, TX	2009.0	chevrolet	good	8 cylinders	gas	217743.0	clean	automatic	rwd	full-size	SUV	white	2WD 1/2 ton\nLeather Captains Chairs\nIn good …	9000
1	abilene, TX	2002.0	gmc	good	8 cylinders	gas	195000.0	clean	automatic	4wd	NaN	pickup	white	2002 GMC Sierra Extended Cab Truck For Sale! R…	6000
2	abilene, TX	2007.0	pontiac	excellent	4 cylinders	gas	NaN	clean	automatic	fwd	compact	convertible	red	34,965 original miles excellent condition.\n\n…	7000
3	abilene, TX	2012.0	chevrolet	excellent	8 cylinders	diesel	178000.0	clean	automatic	4wd	full-size	pickup	silver	2012 Chevrolet 3500 178k miles, runs and drive…	37000
4	abilene, TX	2003.0	NaN	fair	8 cylinders	gas	269000.0	clean	automatic	4wd	NaN	pickup	silver		

```
4  Silver 2003 F150 Triton v8-plugs have been cha…   3700
```

[4]:
```
df.shape
```

[4]:
```
(435653, 15)
```

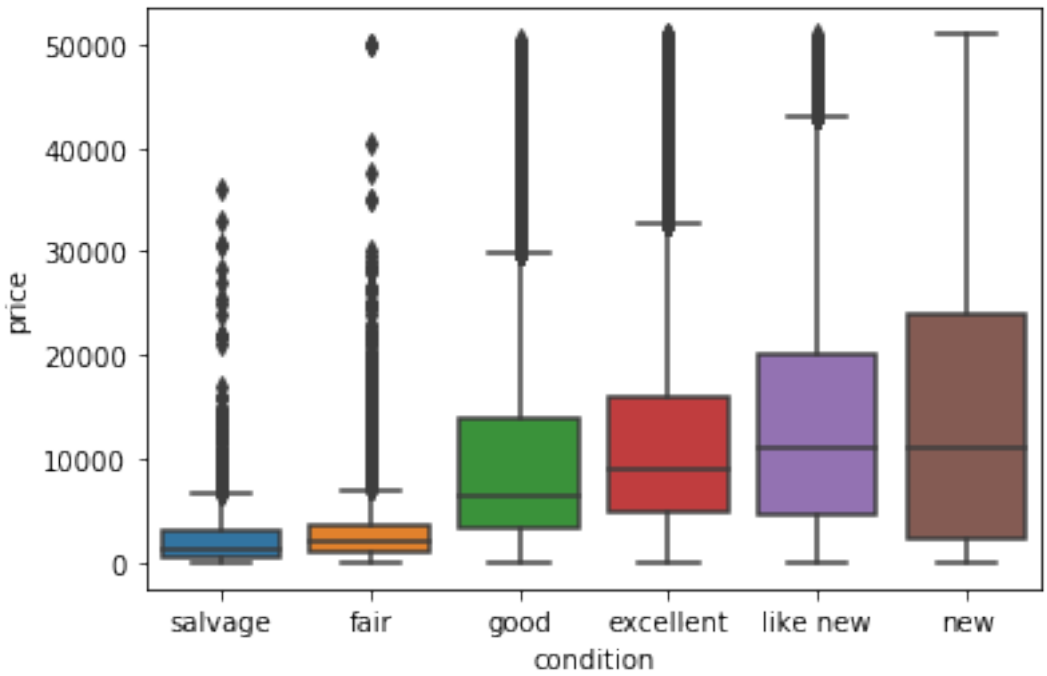
[5]:
```
df.transmission.value_counts()
```

[5]:
```
automatic    386150
manual        33474
other         11731
Name: transmission, dtype: int64
```

1.3 Linear regression and feature engineering (35 mts)

A natural first step to building a model is to consider different variables that impact the price of a used car. For example, let's take a look at the following graph, which gives boxplots of price for each condition category:

[6]:
```
ax = sns.boxplot(x='condition', y='price', data=df, order=['salvage', 'fair',␣
↪'good', 'excellent', 'like new', 'new'])
```



Unsurprisingly, we see that price increases as the condition of the car improves from `salvage` to `new`. Let's build a linear model using the different category labels within this feature:

[7]:
```
# Clean condition
df = pd.concat([df, pd.get_dummies(df['condition'])], axis=1)
np.random.seed(0)
mask = np.random.randn(len(df)) < 0.75
df_train = df[mask]
df_test = df[~mask]
```

[8]:
```
model1 = 'price~salvage + fair + good + excellent + Q("like new") + new'
lml = sm.ols(formula=model1, data=df_train).fit()
print(lml.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                  price   R-squared:                       0.066
Model:                            OLS   Adj. R-squared:                  0.066
Method:                 Least Squares   F-statistic:                     3958.
Date:                Fri, 29 Nov 2019   Prob (F-statistic):               0.00
Time:                        18:43:35   Log-Likelihood:            -3.5809e+06
No. Observations:              336778   AIC:                         7.162e+06
Df Residuals:                  336771   BIC:                         7.162e+06
Df Model:                           6
Covariance Type:            nonrobust
==============================================================================
=
                     coef    std err          t      P>|t|      [0.025
0.975]
------------------------------------------------------------------------------
-
Intercept      1.481e+04     28.417    521.148      0.000    1.48e+04
1.49e+04
salvage       -1.208e+04    397.125    -30.429      0.000   -1.29e+04
-1.13e+04
fair          -1.201e+04    107.577   -111.640      0.000   -1.22e+04
-1.18e+04
good          -5267.6879     45.673   -115.335      0.000   -5357.205
-5178.171
excellent     -3376.5298     42.447    -79.547      0.000   -3459.724
-3293.335
Q("like new") -1238.6643     74.584    -16.608      0.000   -1384.847
-1092.482
new            -387.0998    367.579     -1.053      0.292   -1107.544
333.344
==============================================================================
Omnibus:                    43002.221   Durbin-Watson:                   1.514
Prob(Omnibus):                  0.000   Jarque-Bera (JB):            62357.858
Skew:                           0.971   Prob(JB):                         0.00
Kurtosis:                       3.820   Cond. No.                         24.8
```

```
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
```

[9]:
```
pred = lml.predict(df_test[['salvage', 'fair', 'good', 'excellent', 'like new',␣
 ↪'new']])
mse(pred, df_test['price'])
```

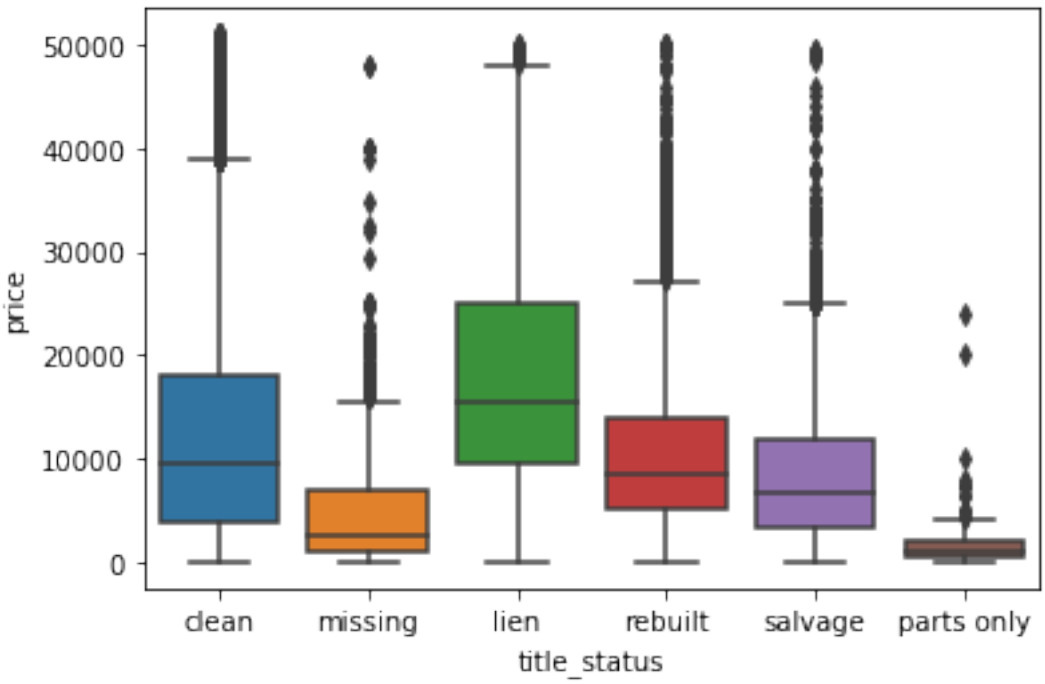
[9]:
```
101412127.78212239
```

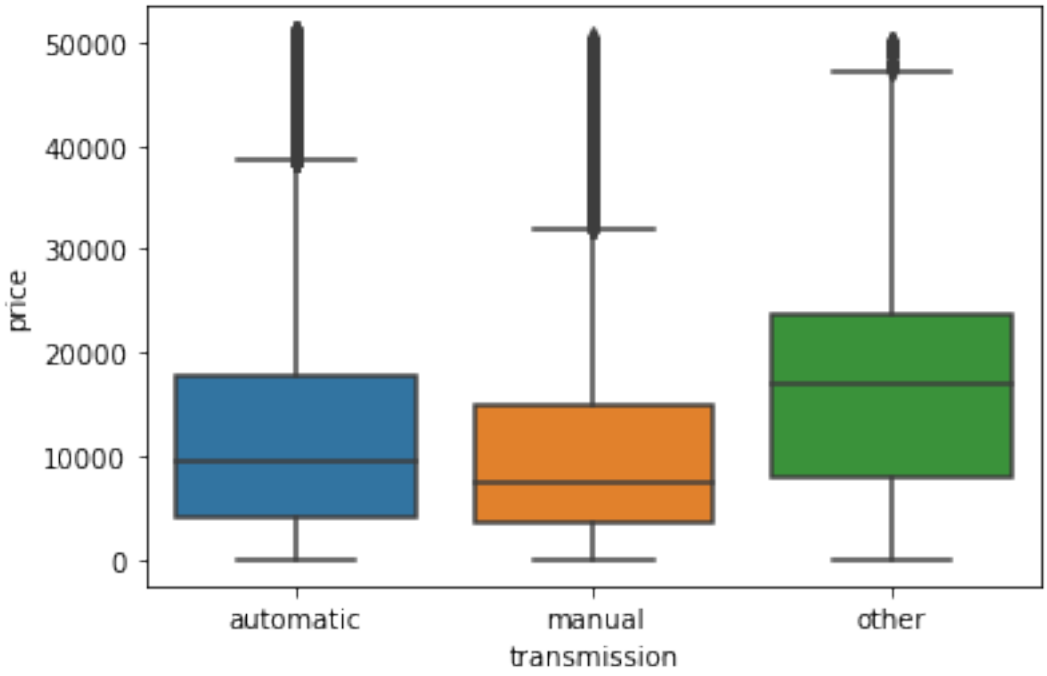
1.3.1 Exercise 1: (15 mts)

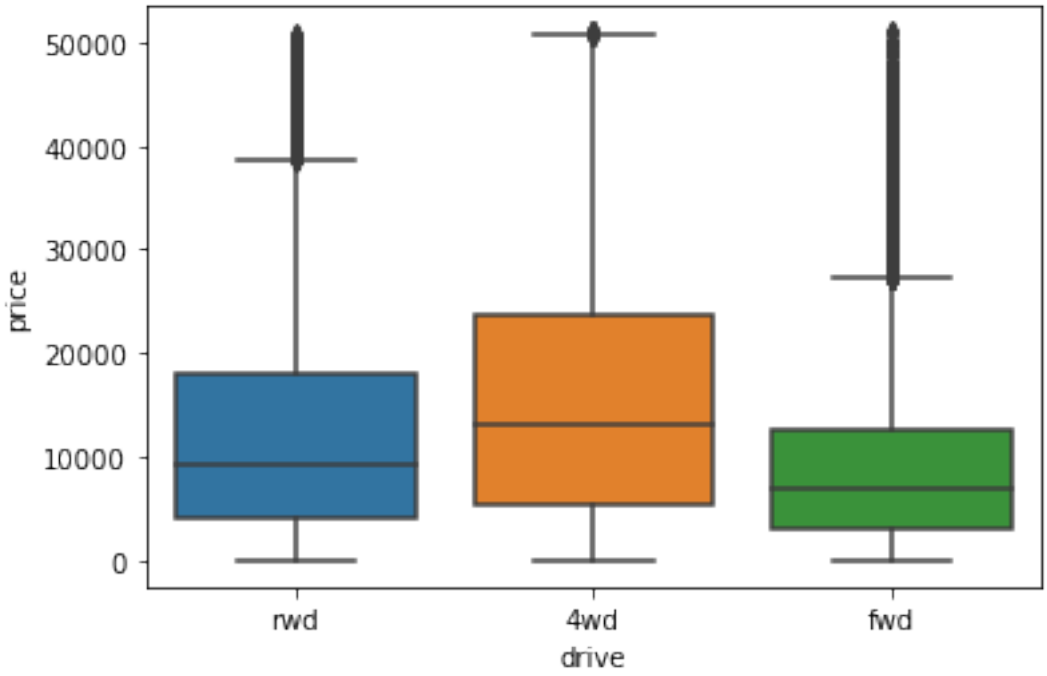
Perform an exploratory analysis of the dataset to find other features that may be correlated with price. Use these features to create a linear model regressing price against those features. How is the model-fit? Why do you think that is?

Answer. One possible solution is shown below:

[10]:
```
### Suggested answer

for col in ['title_status', 'transmission', 'drive']:
    sns.boxplot(x=col, y='price', data=df)
    plt.show()
    df = pd.concat([df, pd.get_dummies(df[col])], axis=1)
```




50000
40000
30000
20000
10000
0
price
automatic
manual
other
transmission


50000
40000
30000
20000
10000
0
price
rwd
4wd
fwd
drive

[11]:
```
for col in ['title_status', 'transmission', 'drive']:
    df_train = pd.concat([df_train, pd.get_dummies(df_train[col])], axis=1)
    df_test = pd.concat([df_test, pd.get_dummies(df_test[col])], axis=1)
```

[12]:
```
model2 = 'price~Q("parts only") + missing + salvage + rebuilt + clean + lien'
lml = sm.ols(formula=model2, data=df_train).fit()
print(lml.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                  price   R-squared:                       0.007
Model:                            OLS   Adj. R-squared:                  0.007
Method:                 Least Squares   F-statistic:                     337.3
Date:                Fri, 29 Nov 2019   Prob (F-statistic):               0.00
Time:                        18:44:16   Log-Likelihood:            -3.5912e+06
No. Observations:              336778   AIC:                         7.182e+06
Df Residuals:                  336770   BIC:                         7.183e+06
Df Model:                           7
Covariance Type:            nonrobust
==============================================================================
===
                       coef    std err          t      P>|t|      [0.025
0.975]
------------------------------------------------------------------------------
---
Intercept        1.634e+04    211.981     77.077      0.000    1.59e+04
1.68e+04
Q("parts only") -1.239e+04   1103.201    -11.234      0.000   -1.46e+04
-1.02e+04
missing         -9704.9651    509.055    -19.065      0.000   -1.07e+04
-8707.232
salvage[0]      -7658.6264    416.059    -18.408      0.000   -8474.090
-6843.163
salvage[1]      -7346.6684    279.917    -26.246      0.000   -7895.297
-6798.040
rebuilt         -5698.0997    248.494    -22.931      0.000   -6185.140
-5211.060
clean           -4204.2683    212.765    -19.760      0.000   -4621.282
-3787.255
lien             1522.7858    291.413      5.226      0.000     951.624
2093.947
==============================================================================
Omnibus:                    47788.715   Durbin-Watson:                   1.508
Prob(Omnibus):                  0.000   Jarque-Bera (JB):            71114.242
Skew:                           1.060   Prob(JB):                         0.00
Kurtosis:                       3.758   Cond. No.                         86.5
==============================================================================
```

```
Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
```

[13]:
```
pred = lml.predict(df_test[['parts only', 'missing', 'salvage', 'rebuilt',␣
 ↪'clean', 'lien']])
mse(pred, df_test['price'])
```

[13]: 108044577.24286044

We have even worse results (although only slightly) for this model than for our previous, naive model. This could be happening for a couple of reasons:

1. We may have wrongly weighted the relative values of these factors. For example, we weighted a `salvage` car as 1 and a `fair` car as 2, when maybe we should have weighted a `fair` car as 3 (i.e. three times as good as a `salvage` car).
2. The interaction between these variables may be highly non-linear. By definition, a linear model cannot accommodate this without additional data wrangling & engineering.

1.3.2 Exercise 2: (5 mts)

Write code to create dummy variables for each categorical value in this dataframe, and create a new dataframe with each categorical column turned into either numerical data or a set of dummy columns.

Answer. Our suggested answer is below:

[14]:
```
cols = ['manufacturer', 'fuel', 'size', 'type','paint_color']
for col in cols:
    df = pd.concat([df, pd.get_dummies(df[col])], axis=1)

df.drop('other', axis=1, inplace=True)

df.head()
```

[14]:

	city	year	manufacturer	condition	cylinders	fuel	odometer	\
0	abilene, TX	2009.0	chevrolet	good	8 cylinders	gas	217743.0	
1	abilene, TX	2002.0	gmc	good	8 cylinders	gas	195000.0	
2	abilene, TX	2007.0	pontiac	excellent	4 cylinders	gas	NaN	
3	abilene, TX	2012.0	chevrolet	excellent	8 cylinders	diesel	178000.0	
4	abilene, TX	2003.0	NaN	fair	8 cylinders	gas	269000.0	

	title_status	transmission	drive	…	brown	custom	green	grey	orange	\
0	clean	automatic	rwd	…	0	0	0	0	0	
1	clean	automatic	4wd	…	0	0	0	0	0	
2	clean	automatic	fwd	…	0	0	0	0	0	
3	clean	automatic	4wd	…	0	0	0	0	0	

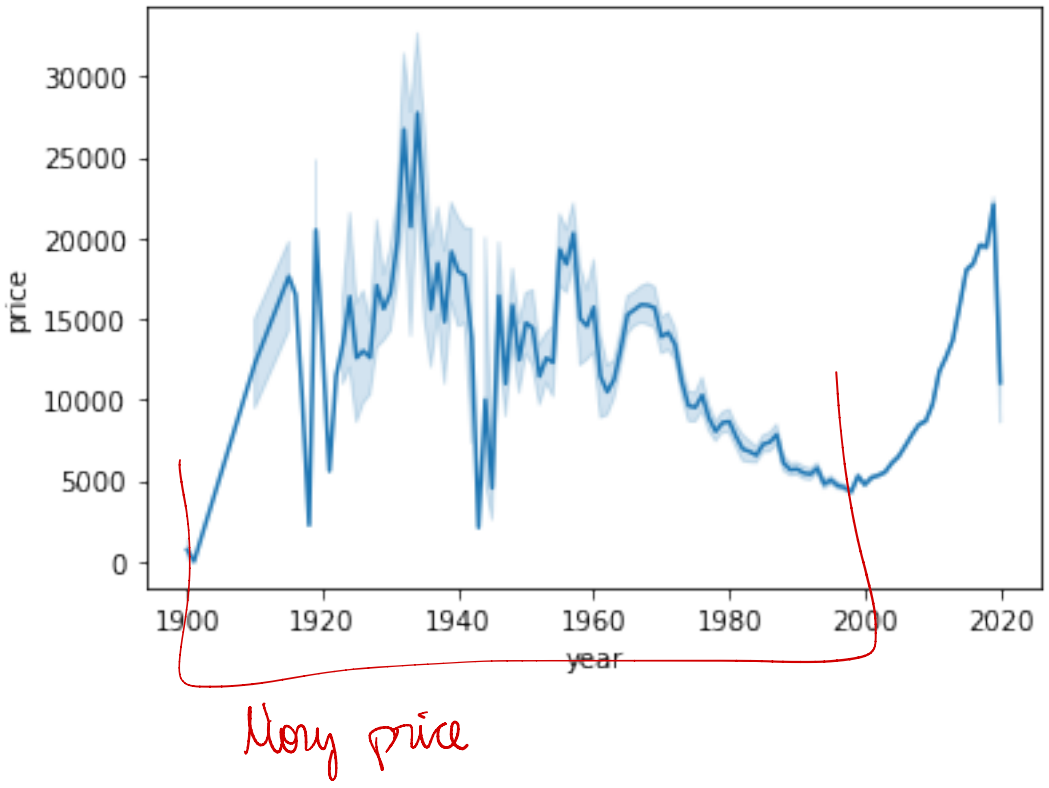
```
4        clean    automatic   4wd  …     0      0      0    0       0

   purple  red  silver  white  yellow
0       0    0       0      1       0
1       0    0       0      1       0
2       0    1       0      0       0
3       0    0       1      0       0
4       0    0       1      0       0

[5 rows x 106 columns]
```

There are already a lot of new columns with a lot of potential for errors! In order to use this categorical data in a linear regression, we need some understanding of how this data can be converted to numbers. This may require significant domain knowledge. With cars, it's straightforward to understand that "good" will be more expensive than "salvage". However, our intuitions may not be so accurate in other situations.

We can try to make some more quick plots to understand how these features impact the price of a used car. Then we could restrict ourselves to only looking at the features that seem to have a major impact:

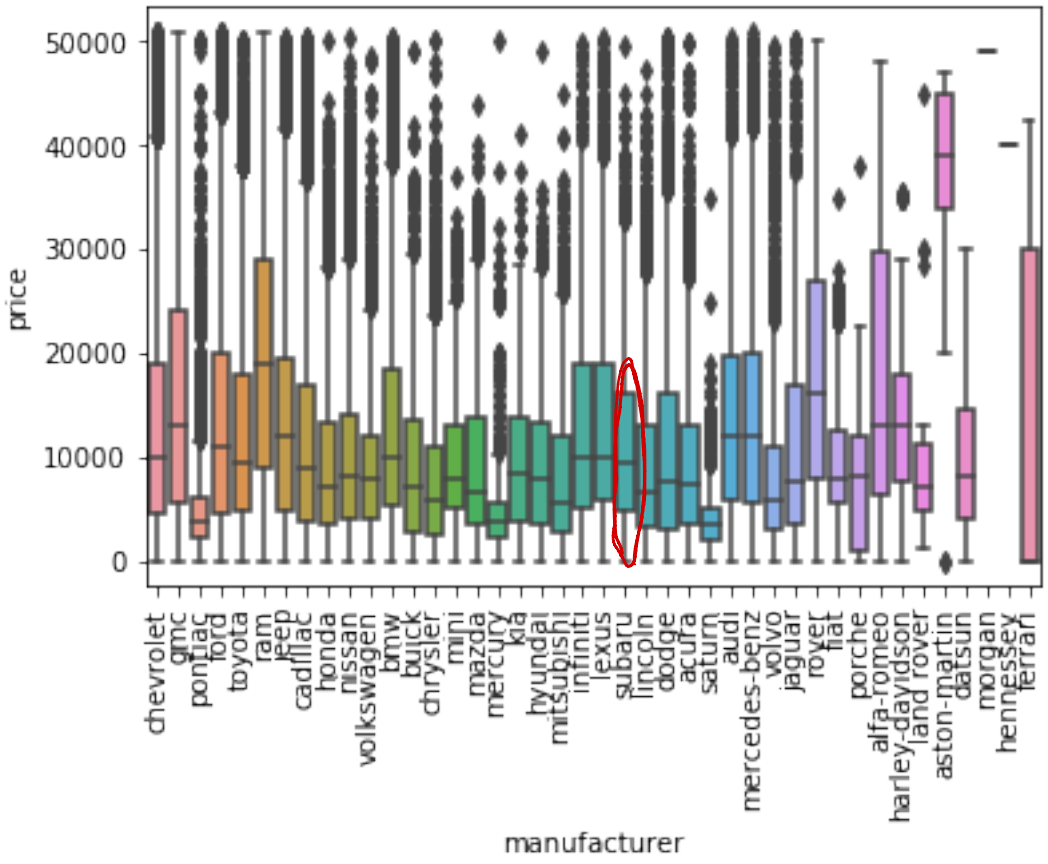
[15]:
```
sns.lineplot(x='year', y='price', data=df)
```

[15]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x101b92d30>
```



[16]:
```
sns.boxplot(x='manufacturer', y='price', data=df)
plt.xticks(rotation=90)
```

[16]:
```
(array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
        17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33,
        34, 35, 36, 37, 38, 39, 40, 41]),
 <a list of 42 Text xticklabel objects>)
```



Without a doubt, these graphs are interesting, and this kind of exploratory data analysis, which we've been doing since the beginning of the training, is incredibly important. For instance, it's surprising that price does not increase linearly with the condition of the car, and it's also notable how noisy the prices are for years before 1980.

1.3.3 Exercise 3: (5 mts)

Which of the following are difficulties with a traditional feature engineering approach to this problem?

I. It is difficult to think of enough relevant features

II. Many desired features may not be quantitative

III. It is difficult to extract those features from the photos

IV. Some features may be too abstract to immediately recognize

Answer. All of the above. These are all reasons for which we may want to consider more sophisticated approaches to predicting the prices of used cars.

1.4 A workaround: neural networks (50 mts)

One way to avoid the difficulties of explicit feature engineering is by using **neural networks**. In a neural network, the computer automatically optimizes relevant features from the data and uses those features in a model to test the effects of various parameters and tune those parameters to fit the model to the given labels. This is essentially a giant calculus problem, where the computer minimizes the amount by which various functions on the data make incorrect predictions. If you've studied multivariable calculus, you may recognize this solution as requiring the **gradient vector**:

A more-precise characterization of the functions that a neural network uses is given above. The inner workings of the algorithm take inputs on the left-hand side to provide outputs on the right-hand side by multiplying each layer by the edges between neurons to get the next layer. Then, the algorithm uses multivariable calculus (in a process known as **gradient descent**) to optimize each edge.

We can add more hidden layers in the middle to allow the neural network to create more complex functions, like in the following diagram:

This is then called a **deep neural network**. Modern deep neural networks can have hundreds of layers.

This image shows an example of how the nodes in a neural network are calculated:

(*Source:* https://medium.com/wwblog/transformation-in-neural-networks-cdf74cbd8da8).

Each node is a linear combination of the layer before it. Between each layer, there is an **activation function** (usually the logistic regression function), which allows our network to develop non-linear sensitivities. When there are multiple layers, we can keep adding more linear combinations, which can lead to more complex *non-linear* models. This is the main strength of neural networks over more elementary regressions. One simple non-linear activation function that we can use is the *relu* function, which looks like the following:

[17]:
```
df_clean = pd.concat([df['year'], df.iloc[:,14:]], axis=1)#.
 ↪dropna(inplace=True, axis=0)
df_clean.dropna(inplace=True)
df_clean.head()
```

[17]:

	year	price	excellent	fair	good	like new	new	salvage	clean	lien	\
0	2009.0	9000	0	0	1	0	0	0	1	0	
1	2002.0	6000	0	0	1	0	0	0	1	0	
2	2007.0	7000	1	0	0	0	0	0	1	0	
3	2012.0	37000	1	0	0	0	0	0	1	0	

```
4  2003.0   3700          0     1     0          0    0         0      1     0

   …  brown  custom  green  grey  orange  purple  red  silver  white  yellow
0  …      0       0      0     0       0       0    0       0      1       0
1  …      0       0      0     0       0       0    0       0      1       0
2  …      0       0      0     0       0       0    1       0      0       0
3  …      0       0      0     0       0       0    0       1      0       0
4  …      0       0      0     0       0       0    0       1      0       0

[5 rows x 93 columns]
```

Let's split our data into training and testing data so we can properly evaluate how our model does later on:

[18]:
```
np.random.seed(0)
mask = np.random.randn(len(df_clean)) < 0.75
df_train = df_clean[mask]
df_test = df_clean[~mask]

print('Training:', len(df_train))
print('Testing:', len(df_test))
print('Total:', len(df))
```

```
Training: 335934
Testing: 98608
Total: 435653
```

[19]:
```
# Reindex
df_train.index = np.arange(len(df_train))
df_test.index = np.arange(len(df_test))
```

1.4.1 Exercise 4: (5 mts)

Which of the following is a suitable application for a neural network?

A. Classifying images of handwritten numerals

B. Analyzing Don Quixote

C. Deciding how to caption a photograph

D. Sorting an array

Answer. A is correct. Certain types of neural networks are able to interpret images as pixels and then make sense of those pixels in various ways, such as identifying numerals. Neural networks alone are still unable to understand text data well enough to analyze a poem or caption a photograph (this requires insights from **natural language processing**). Sorting an array using a neural network is overkill and quite inefficient; it's better to simply use a standard sorting algorithm from computer science theory for this purpose.

1.4.2 A simple first model (5 mts)

[20]:
```
from tensorflow import keras
from tensorflow.keras import layers
```

```
/Users/haris.jaliawala/anaconda3/envs/week8/lib/python3.7/site-
packages/tensorflow/python/framework/dtypes.py:526: FutureWarning: Passing
(type, 1) or '1type' as a synonym of type is deprecated; in a future version of
numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint8 = np.dtype([("qint8", np.int8, 1)])
/Users/haris.jaliawala/anaconda3/envs/week8/lib/python3.7/site-
packages/tensorflow/python/framework/dtypes.py:527: FutureWarning: Passing
(type, 1) or '1type' as a synonym of type is deprecated; in a future version of
numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_quint8 = np.dtype([("quint8", np.uint8, 1)])
/Users/haris.jaliawala/anaconda3/envs/week8/lib/python3.7/site-
packages/tensorflow/python/framework/dtypes.py:528: FutureWarning: Passing
(type, 1) or '1type' as a synonym of type is deprecated; in a future version of
numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint16 = np.dtype([("qint16", np.int16, 1)])
/Users/haris.jaliawala/anaconda3/envs/week8/lib/python3.7/site-
packages/tensorflow/python/framework/dtypes.py:529: FutureWarning: Passing
(type, 1) or '1type' as a synonym of type is deprecated; in a future version of
numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_quint16 = np.dtype([("quint16", np.uint16, 1)])
/Users/haris.jaliawala/anaconda3/envs/week8/lib/python3.7/site-
packages/tensorflow/python/framework/dtypes.py:530: FutureWarning: Passing
(type, 1) or '1type' as a synonym of type is deprecated; in a future version of
numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint32 = np.dtype([("qint32", np.int32, 1)])
/Users/haris.jaliawala/anaconda3/envs/week8/lib/python3.7/site-
packages/tensorflow/python/framework/dtypes.py:535: FutureWarning: Passing
(type, 1) or '1type' as a synonym of type is deprecated; in a future version of
numpy, it will be understood as (type, (1,)) / '(1,)type'.
  np_resource = np.dtype([("resource", np.ubyte, 1)])
```

To make this even more clear, let's consider the following simple toy example with a nonlinearity:

[21]:
```
toy = df_train.iloc[:1000].loc[:,['year', 'price']]
toy['price'] = (toy['price'] - toy['price'].mean()) / toy['price'].std() #␣
 ↪Normalization
toy['year'] = (toy['year'] - toy['year'].mean()) / toy['year'].std()

toy['toy'] = (toy['price'] ** 2 - toy['year'] ** 3)
toy.head()
```

[21]:

	year	price	toy
0	-0.650935	-0.700488	0.766495
1	0.430189	0.599313	0.279564
2	0.921609	2.462827	5.282735
3	0.626757	1.903773	3.378144
4	-0.454367	-0.905940	0.914532

Let's try regressing on this with both a linear model and a simple neural network:

[22]:
```
failure_model = 'toy~year + price'
lml = sm.ols(formula=failure_model, data=toy).fit()
print(lml.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                    toy   R-squared:                       0.646
Model:                            OLS   Adj. R-squared:                  0.646
Method:                 Least Squares   F-statistic:                     911.4
Date:                Fri, 29 Nov 2019   Prob (F-statistic):          8.34e-226
Time:                        18:54:32   Log-Likelihood:                -4190.0
No. Observations:                1000   AIC:                             8386.
Df Residuals:                     997   BIC:                             8401.
Df Model:                           2
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept      4.4639      0.506      8.823      0.000       3.471       5.457
year         -23.1585      0.543    -42.623      0.000     -24.225     -22.092
price          9.6612      0.543     17.781      0.000       8.595      10.727
==============================================================================
Omnibus:                     1268.126   Durbin-Watson:                   2.067
Prob(Omnibus):                  0.000   Jarque-Bera (JB):           319509.412
Skew:                           6.323   Prob(JB):                         0.00
Kurtosis:                      89.650   Cond. No.                         1.46
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
```

[23]:
```
pred = lml.predict(toy[['year', 'price']])
mse(pred, toy['toy'])
```

[23]: 255.1942716717411

[24]:
```
toy_model = keras.Sequential([layers.Dense(1000,activation='relu',␣
↪input_shape=[2,]),
                              layers.Dense(500, activation='relu'),
                              layers.Dense(1)])

opt = keras.optimizers.Adam(decay=0.01/500)
toy_model.compile(loss='mse', optimizer=opt, metrics=['mse'])
toy_model.fit(toy[['price','year']], toy['toy'], epochs=200, validation_split =␣
↪0.2, verbose=1)
```

```
WARNING:tensorflow:From
/Users/haris.jaliawala/anaconda3/envs/week8/lib/python3.7/site-
packages/tensorflow/python/ops/resource_variable_ops.py:435: colocate_with (from
tensorflow.python.framework.ops) is deprecated and will be removed in a future
version.
Instructions for updating:
Colocations handled automatically by placer.
WARNING:tensorflow:From
/Users/haris.jaliawala/anaconda3/envs/week8/lib/python3.7/site-
packages/tensorflow/python/keras/utils/losses_utils.py:170: to_float (from
tensorflow.python.ops.math_ops) is deprecated and will be removed in a future
version.
Instructions for updating:
Use tf.cast instead.
Train on 800 samples, validate on 200 samples
WARNING:tensorflow:From
/Users/haris.jaliawala/anaconda3/envs/week8/lib/python3.7/site-
packages/tensorflow/python/ops/math_ops.py:3066: to_int32 (from
tensorflow.python.ops.math_ops) is deprecated and will be removed in a future
version.
Instructions for updating:
Use tf.cast instead.
Epoch 1/200
800/800 [==============================] - 0s 413us/sample - loss: 762.5631 -
mean_squared_error: 762.5630 - val_loss: 162.8607 - val_mean_squared_error:
162.8607
Epoch 2/200
800/800 [==============================] - 0s 232us/sample - loss: 505.0434 -
mean_squared_error: 505.0434 - val_loss: 62.1170 - val_mean_squared_error:
62.1170
Epoch 3/200
800/800 [==============================] - 0s 200us/sample - loss: 277.3989 -
mean_squared_error: 277.3990 - val_loss: 45.2612 - val_mean_squared_error:
45.2612
Epoch 4/200
800/800 [==============================] - 0s 182us/sample - loss: 173.6310 -
mean_squared_error: 173.6310 - val_loss: 35.9034 - val_mean_squared_error:
```

```
35.9034
Epoch 5/200
800/800 [==============================] - 0s 176us/sample - loss: 151.4288 -
mean_squared_error: 151.4288 - val_loss: 51.0662 - val_mean_squared_error:
51.0662
Epoch 6/200
800/800 [==============================] - 0s 180us/sample - loss: 117.4726 -
mean_squared_error: 117.4726 - val_loss: 43.4914 - val_mean_squared_error:
43.4914
Epoch 7/200
800/800 [==============================] - 0s 186us/sample - loss: 97.0397 -
mean_squared_error: 97.0397 - val_loss: 41.3227 - val_mean_squared_error:
41.3227
Epoch 8/200
800/800 [==============================] - 0s 175us/sample - loss: 96.0617 -
mean_squared_error: 96.0617 - val_loss: 50.0189 - val_mean_squared_error:
50.0189
Epoch 9/200
800/800 [==============================] - 0s 176us/sample - loss: 78.1716 -
mean_squared_error: 78.1716 - val_loss: 32.3096 - val_mean_squared_error:
32.3096
Epoch 10/200
800/800 [==============================] - 0s 177us/sample - loss: 72.4263 -
mean_squared_error: 72.4263 - val_loss: 35.9732 - val_mean_squared_error:
35.9732
Epoch 11/200
800/800 [==============================] - 0s 183us/sample - loss: 64.2477 -
mean_squared_error: 64.2477 - val_loss: 41.9641 - val_mean_squared_error:
41.9641
Epoch 12/200
800/800 [==============================] - 0s 177us/sample - loss: 96.4449 -
mean_squared_error: 96.4449 - val_loss: 23.5706 - val_mean_squared_error:
23.5706
Epoch 13/200
800/800 [==============================] - 0s 195us/sample - loss: 45.2161 -
mean_squared_error: 45.2161 - val_loss: 37.9678 - val_mean_squared_error:
37.9678
Epoch 14/200
800/800 [==============================] - 0s 184us/sample - loss: 47.4200 -
mean_squared_error: 47.4200 - val_loss: 33.8760 - val_mean_squared_error:
33.8760
Epoch 15/200
800/800 [==============================] - 0s 187us/sample - loss: 55.4782 -
mean_squared_error: 55.4782 - val_loss: 17.6419 - val_mean_squared_error:
17.6419
Epoch 16/200
800/800 [==============================] - 0s 182us/sample - loss: 44.7575 -
mean_squared_error: 44.7575 - val_loss: 23.6382 - val_mean_squared_error:
```

```
23.6382
Epoch 17/200
800/800 [==============================] - 0s 183us/sample - loss: 32.5180 -
mean_squared_error: 32.5180 - val_loss: 19.4743 - val_mean_squared_error:
19.4743
Epoch 18/200
800/800 [==============================] - 0s 184us/sample - loss: 30.8619 -
mean_squared_error: 30.8619 - val_loss: 32.6552 - val_mean_squared_error:
32.6552
Epoch 19/200
800/800 [==============================] - 0s 181us/sample - loss: 34.1978 -
mean_squared_error: 34.1978 - val_loss: 30.1265 - val_mean_squared_error:
30.1265
Epoch 20/200
800/800 [==============================] - 0s 178us/sample - loss: 25.0806 -
mean_squared_error: 25.0806 - val_loss: 19.0922 - val_mean_squared_error:
19.0922
Epoch 21/200
800/800 [==============================] - 0s 172us/sample - loss: 25.5284 -
mean_squared_error: 25.5284 - val_loss: 19.6463 - val_mean_squared_error:
19.6463
Epoch 22/200
800/800 [==============================] - 0s 182us/sample - loss: 19.3211 -
mean_squared_error: 19.3211 - val_loss: 13.8132 - val_mean_squared_error:
13.8132
Epoch 23/200
800/800 [==============================] - 0s 178us/sample - loss: 17.9847 -
mean_squared_error: 17.9847 - val_loss: 12.4209 - val_mean_squared_error:
12.4209
Epoch 24/200
800/800 [==============================] - 0s 179us/sample - loss: 17.4213 -
mean_squared_error: 17.4213 - val_loss: 7.7200 - val_mean_squared_error: 7.7200
Epoch 25/200
800/800 [==============================] - 0s 184us/sample - loss: 20.8184 -
mean_squared_error: 20.8184 - val_loss: 27.6040 - val_mean_squared_error:
27.6040
Epoch 26/200
800/800 [==============================] - 0s 187us/sample - loss: 20.7894 -
mean_squared_error: 20.7894 - val_loss: 9.0809 - val_mean_squared_error: 9.0809
Epoch 27/200
800/800 [==============================] - 0s 183us/sample - loss: 12.9934 -
mean_squared_error: 12.9934 - val_loss: 9.1808 - val_mean_squared_error: 9.1808
Epoch 28/200
800/800 [==============================] - 0s 174us/sample - loss: 10.2111 -
mean_squared_error: 10.2111 - val_loss: 5.4440 - val_mean_squared_error: 5.4440
Epoch 29/200
800/800 [==============================] - 0s 171us/sample - loss: 10.5331 -
mean_squared_error: 10.5331 - val_loss: 5.2673 - val_mean_squared_error: 5.2673
```

```
Epoch 30/200
800/800 [==============================] - 0s 174us/sample - loss: 11.0753 -
mean_squared_error: 11.0753 - val_loss: 11.8064 - val_mean_squared_error:
11.8064
Epoch 31/200
800/800 [==============================] - 0s 177us/sample - loss: 7.2014 -
mean_squared_error: 7.2014 - val_loss: 3.8470 - val_mean_squared_error: 3.8470
Epoch 32/200
800/800 [==============================] - 0s 181us/sample - loss: 7.7914 -
mean_squared_error: 7.7914 - val_loss: 4.9637 - val_mean_squared_error: 4.9637
Epoch 33/200
800/800 [==============================] - 0s 182us/sample - loss: 5.0249 -
mean_squared_error: 5.0249 - val_loss: 3.3038 - val_mean_squared_error: 3.3038
Epoch 34/200
800/800 [==============================] - 0s 184us/sample - loss: 9.3910 -
mean_squared_error: 9.3910 - val_loss: 2.4789 - val_mean_squared_error: 2.4789
Epoch 35/200
800/800 [==============================] - 0s 172us/sample - loss: 5.7940 -
mean_squared_error: 5.7940 - val_loss: 8.0047 - val_mean_squared_error: 8.0047
Epoch 36/200
800/800 [==============================] - 0s 177us/sample - loss: 7.9613 -
mean_squared_error: 7.9613 - val_loss: 11.0188 - val_mean_squared_error: 11.0188
Epoch 37/200
800/800 [==============================] - 0s 180us/sample - loss: 9.3736 -
mean_squared_error: 9.3736 - val_loss: 8.1940 - val_mean_squared_error: 8.1940
Epoch 38/200
800/800 [==============================] - 0s 180us/sample - loss: 5.3783 -
mean_squared_error: 5.3783 - val_loss: 8.9841 - val_mean_squared_error: 8.9841
Epoch 39/200
800/800 [==============================] - 0s 200us/sample - loss: 27.8564 -
mean_squared_error: 27.8564 - val_loss: 6.6697 - val_mean_squared_error: 6.6697
Epoch 40/200
800/800 [==============================] - 0s 189us/sample - loss: 46.9768 -
mean_squared_error: 46.9768 - val_loss: 2.1302 - val_mean_squared_error: 2.1302
Epoch 41/200
800/800 [==============================] - 0s 183us/sample - loss: 19.2287 -
mean_squared_error: 19.2287 - val_loss: 1.8493 - val_mean_squared_error: 1.8493
Epoch 42/200
800/800 [==============================] - 0s 178us/sample - loss: 4.3843 -
mean_squared_error: 4.3843 - val_loss: 2.5032 - val_mean_squared_error: 2.5032
Epoch 43/200
800/800 [==============================] - 0s 193us/sample - loss: 8.9607 -
mean_squared_error: 8.9607 - val_loss: 2.0871 - val_mean_squared_error: 2.0871
Epoch 44/200
800/800 [==============================] - 0s 181us/sample - loss: 3.2821 -
mean_squared_error: 3.2821 - val_loss: 3.5322 - val_mean_squared_error: 3.5322
Epoch 45/200
800/800 [==============================] - 0s 189us/sample - loss: 2.3990 -
```

```
mean_squared_error: 2.3990 - val_loss: 0.6879 - val_mean_squared_error: 0.6879
Epoch 46/200
800/800 [==============================] - 0s 184us/sample - loss: 2.2561 -
mean_squared_error: 2.2561 - val_loss: 2.3543 - val_mean_squared_error: 2.3543
Epoch 47/200
800/800 [==============================] - 0s 183us/sample - loss: 2.4227 -
mean_squared_error: 2.4227 - val_loss: 0.4441 - val_mean_squared_error: 0.4441
Epoch 48/200
800/800 [==============================] - 0s 186us/sample - loss: 11.9196 -
mean_squared_error: 11.9196 - val_loss: 1.2038 - val_mean_squared_error: 1.2038
Epoch 49/200
800/800 [==============================] - 0s 183us/sample - loss: 4.9905 -
mean_squared_error: 4.9905 - val_loss: 3.6717 - val_mean_squared_error: 3.6717
Epoch 50/200
800/800 [==============================] - 0s 181us/sample - loss: 2.7276 -
mean_squared_error: 2.7276 - val_loss: 0.3122 - val_mean_squared_error: 0.3122
Epoch 51/200
800/800 [==============================] - 0s 191us/sample - loss: 2.2717 -
mean_squared_error: 2.2717 - val_loss: 3.4552 - val_mean_squared_error: 3.4552
Epoch 52/200
800/800 [==============================] - 0s 183us/sample - loss: 2.6251 -
mean_squared_error: 2.6251 - val_loss: 0.5749 - val_mean_squared_error: 0.5749
Epoch 53/200
800/800 [==============================] - 0s 191us/sample - loss: 3.3848 -
mean_squared_error: 3.3848 - val_loss: 2.3102 - val_mean_squared_error: 2.3102
Epoch 54/200
800/800 [==============================] - 0s 175us/sample - loss: 2.5502 -
mean_squared_error: 2.5502 - val_loss: 0.2735 - val_mean_squared_error: 0.2735
Epoch 55/200
800/800 [==============================] - 0s 189us/sample - loss: 2.5586 -
mean_squared_error: 2.5586 - val_loss: 0.7491 - val_mean_squared_error: 0.7491
Epoch 56/200
800/800 [==============================] - 0s 190us/sample - loss: 2.2323 -
mean_squared_error: 2.2323 - val_loss: 1.5366 - val_mean_squared_error: 1.5366
Epoch 57/200
800/800 [==============================] - 0s 182us/sample - loss: 2.8701 -
mean_squared_error: 2.8701 - val_loss: 0.4120 - val_mean_squared_error: 0.4120
Epoch 58/200
800/800 [==============================] - 0s 183us/sample - loss: 1.5777 -
mean_squared_error: 1.5777 - val_loss: 0.0651 - val_mean_squared_error: 0.0651
Epoch 59/200
800/800 [==============================] - 0s 189us/sample - loss: 1.4511 -
mean_squared_error: 1.4511 - val_loss: 0.2478 - val_mean_squared_error: 0.2478
Epoch 60/200
800/800 [==============================] - 0s 182us/sample - loss: 3.4262 -
mean_squared_error: 3.4262 - val_loss: 0.1989 - val_mean_squared_error: 0.1989
Epoch 61/200
800/800 [==============================] - 0s 185us/sample - loss: 1.6043 -
```

```
mean_squared_error: 1.6043 - val_loss: 0.0856 - val_mean_squared_error: 0.0856
Epoch 62/200
800/800 [==============================] - 0s 181us/sample - loss: 2.5863 -
mean_squared_error: 2.5863 - val_loss: 0.0838 - val_mean_squared_error: 0.0838
Epoch 63/200
800/800 [==============================] - 0s 178us/sample - loss: 2.6676 -
mean_squared_error: 2.6676 - val_loss: 0.5659 - val_mean_squared_error: 0.5659
Epoch 64/200
800/800 [==============================] - 0s 189us/sample - loss: 1.0938 -
mean_squared_error: 1.0938 - val_loss: 0.3269 - val_mean_squared_error: 0.3269
Epoch 65/200
800/800 [==============================] - 0s 190us/sample - loss: 2.3930 -
mean_squared_error: 2.3930 - val_loss: 0.1785 - val_mean_squared_error: 0.1785
Epoch 66/200
800/800 [==============================] - 0s 194us/sample - loss: 3.8265 -
mean_squared_error: 3.8265 - val_loss: 0.2383 - val_mean_squared_error: 0.2383
Epoch 67/200
800/800 [==============================] - 0s 181us/sample - loss: 3.1506 -
mean_squared_error: 3.1506 - val_loss: 2.1097 - val_mean_squared_error: 2.1097
Epoch 68/200
800/800 [==============================] - 0s 190us/sample - loss: 4.7730 -
mean_squared_error: 4.7730 - val_loss: 2.6578 - val_mean_squared_error: 2.6578
Epoch 69/200
800/800 [==============================] - 0s 194us/sample - loss: 10.6819 -
mean_squared_error: 10.6819 - val_loss: 17.2629 - val_mean_squared_error:
17.2629
Epoch 70/200
800/800 [==============================] - 0s 178us/sample - loss: 92.4018 -
mean_squared_error: 92.4018 - val_loss: 39.2494 - val_mean_squared_error:
39.2494
Epoch 71/200
800/800 [==============================] - 0s 183us/sample - loss: 76.4770 -
mean_squared_error: 76.4770 - val_loss: 4.2490 - val_mean_squared_error: 4.2490
Epoch 72/200
800/800 [==============================] - 0s 182us/sample - loss: 17.7897 -
mean_squared_error: 17.7897 - val_loss: 1.3622 - val_mean_squared_error: 1.3622
Epoch 73/200
800/800 [==============================] - 0s 177us/sample - loss: 3.8088 -
mean_squared_error: 3.8088 - val_loss: 1.0829 - val_mean_squared_error: 1.0829
Epoch 74/200
800/800 [==============================] - 0s 175us/sample - loss: 3.6369 -
mean_squared_error: 3.6369 - val_loss: 2.4302 - val_mean_squared_error: 2.4302
Epoch 75/200
800/800 [==============================] - 0s 186us/sample - loss: 1.7890 -
mean_squared_error: 1.7890 - val_loss: 2.1636 - val_mean_squared_error: 2.1636
Epoch 76/200
800/800 [==============================] - 0s 215us/sample - loss: 5.6225 -
mean_squared_error: 5.6225 - val_loss: 0.1183 - val_mean_squared_error: 0.1183
```

```
Epoch 77/200
800/800 [==============================] - 0s 223us/sample - loss: 1.5739 -
mean_squared_error: 1.5739 - val_loss: 0.1972 - val_mean_squared_error: 0.1972
Epoch 78/200
800/800 [==============================] - 0s 213us/sample - loss: 1.4197 -
mean_squared_error: 1.4197 - val_loss: 0.7496 - val_mean_squared_error: 0.7496
Epoch 79/200
800/800 [==============================] - 0s 206us/sample - loss: 2.4141 -
mean_squared_error: 2.4141 - val_loss: 2.1127 - val_mean_squared_error: 2.1127
Epoch 80/200
800/800 [==============================] - 0s 175us/sample - loss: 1.6270 -
mean_squared_error: 1.6270 - val_loss: 0.3136 - val_mean_squared_error: 0.3136
Epoch 81/200
800/800 [==============================] - 0s 169us/sample - loss: 1.8515 -
mean_squared_error: 1.8515 - val_loss: 0.3448 - val_mean_squared_error: 0.3448
Epoch 82/200
800/800 [==============================] - 0s 171us/sample - loss: 1.3141 -
mean_squared_error: 1.3141 - val_loss: 0.2942 - val_mean_squared_error: 0.2942
Epoch 83/200
800/800 [==============================] - 0s 180us/sample - loss: 1.0199 -
mean_squared_error: 1.0199 - val_loss: 0.1007 - val_mean_squared_error: 0.1007
Epoch 84/200
800/800 [==============================] - 0s 178us/sample - loss: 2.0925 -
mean_squared_error: 2.0925 - val_loss: 0.3204 - val_mean_squared_error: 0.3204
Epoch 85/200
800/800 [==============================] - 0s 170us/sample - loss: 2.6628 -
mean_squared_error: 2.6628 - val_loss: 1.0162 - val_mean_squared_error: 1.0162
Epoch 86/200
800/800 [==============================] - 0s 173us/sample - loss: 1.1597 -
mean_squared_error: 1.1597 - val_loss: 0.2579 - val_mean_squared_error: 0.2579
Epoch 87/200
800/800 [==============================] - 0s 172us/sample - loss: 1.2107 -
mean_squared_error: 1.2107 - val_loss: 0.7083 - val_mean_squared_error: 0.7083
Epoch 88/200
800/800 [==============================] - 0s 179us/sample - loss: 0.3820 -
mean_squared_error: 0.3820 - val_loss: 0.5650 - val_mean_squared_error: 0.5650
Epoch 89/200
800/800 [==============================] - 0s 177us/sample - loss: 0.7389 -
mean_squared_error: 0.7389 - val_loss: 0.0540 - val_mean_squared_error: 0.0540
Epoch 90/200
800/800 [==============================] - 0s 179us/sample - loss: 1.1665 -
mean_squared_error: 1.1665 - val_loss: 0.0523 - val_mean_squared_error: 0.0523
Epoch 91/200
800/800 [==============================] - 0s 174us/sample - loss: 0.9669 -
mean_squared_error: 0.9669 - val_loss: 0.3810 - val_mean_squared_error: 0.3810
Epoch 92/200
800/800 [==============================] - 0s 175us/sample - loss: 1.4180 -
mean_squared_error: 1.4180 - val_loss: 1.4182 - val_mean_squared_error: 1.4182
```

```
Epoch 93/200
800/800 [==============================] - 0s 167us/sample - loss: 2.1358 -
mean_squared_error: 2.1358 - val_loss: 0.1590 - val_mean_squared_error: 0.1590
Epoch 94/200
800/800 [==============================] - 0s 165us/sample - loss: 0.7673 -
mean_squared_error: 0.7673 - val_loss: 0.0465 - val_mean_squared_error: 0.0465
Epoch 95/200
800/800 [==============================] - 0s 164us/sample - loss: 0.6899 -
mean_squared_error: 0.6899 - val_loss: 0.0595 - val_mean_squared_error: 0.0595
Epoch 96/200
800/800 [==============================] - 0s 165us/sample - loss: 0.5023 -
mean_squared_error: 0.5023 - val_loss: 0.6344 - val_mean_squared_error: 0.6344
Epoch 97/200
800/800 [==============================] - 0s 164us/sample - loss: 2.6792 -
mean_squared_error: 2.6792 - val_loss: 1.1585 - val_mean_squared_error: 1.1585
Epoch 98/200
800/800 [==============================] - 0s 164us/sample - loss: 1.7344 -
mean_squared_error: 1.7344 - val_loss: 0.3540 - val_mean_squared_error: 0.3540
Epoch 99/200
800/800 [==============================] - 0s 165us/sample - loss: 1.2201 -
mean_squared_error: 1.2201 - val_loss: 0.5813 - val_mean_squared_error: 0.5813
Epoch 100/200
800/800 [==============================] - 0s 164us/sample - loss: 0.9112 -
mean_squared_error: 0.9112 - val_loss: 0.8957 - val_mean_squared_error: 0.8957
Epoch 101/200
800/800 [==============================] - 0s 165us/sample - loss: 2.1542 -
mean_squared_error: 2.1542 - val_loss: 0.4457 - val_mean_squared_error: 0.4457
Epoch 102/200
800/800 [==============================] - 0s 165us/sample - loss: 2.0311 -
mean_squared_error: 2.0311 - val_loss: 0.8584 - val_mean_squared_error: 0.8584
Epoch 103/200
800/800 [==============================] - 0s 161us/sample - loss: 0.7315 -
mean_squared_error: 0.7315 - val_loss: 0.2436 - val_mean_squared_error: 0.2436
Epoch 104/200
800/800 [==============================] - 0s 164us/sample - loss: 1.1734 -
mean_squared_error: 1.1734 - val_loss: 0.2997 - val_mean_squared_error: 0.2997
Epoch 105/200
800/800 [==============================] - 0s 168us/sample - loss: 0.6261 -
mean_squared_error: 0.6261 - val_loss: 0.6293 - val_mean_squared_error: 0.6293
Epoch 106/200
800/800 [==============================] - 0s 163us/sample - loss: 1.8225 -
mean_squared_error: 1.8225 - val_loss: 1.3874 - val_mean_squared_error: 1.3874
Epoch 107/200
800/800 [==============================] - 0s 165us/sample - loss: 0.7992 -
mean_squared_error: 0.7992 - val_loss: 0.7367 - val_mean_squared_error: 0.7367
Epoch 108/200
800/800 [==============================] - 0s 164us/sample - loss: 1.4605 -
mean_squared_error: 1.4605 - val_loss: 0.8900 - val_mean_squared_error: 0.8900
```

```
Epoch 109/200
800/800 [==============================] - 0s 165us/sample - loss: 1.3735 -
mean_squared_error: 1.3735 - val_loss: 1.2125 - val_mean_squared_error: 1.2125
Epoch 110/200
800/800 [==============================] - 0s 163us/sample - loss: 2.4864 -
mean_squared_error: 2.4864 - val_loss: 0.1325 - val_mean_squared_error: 0.1325
Epoch 111/200
800/800 [==============================] - 0s 167us/sample - loss: 4.0480 -
mean_squared_error: 4.0480 - val_loss: 3.7808 - val_mean_squared_error: 3.7808
Epoch 112/200
800/800 [==============================] - 0s 164us/sample - loss: 5.0440 -
mean_squared_error: 5.0440 - val_loss: 2.8994 - val_mean_squared_error: 2.8994
Epoch 113/200
800/800 [==============================] - 0s 165us/sample - loss: 14.2369 -
mean_squared_error: 14.2369 - val_loss: 3.1774 - val_mean_squared_error: 3.1774
Epoch 114/200
800/800 [==============================] - 0s 161us/sample - loss: 4.7504 -
mean_squared_error: 4.7504 - val_loss: 2.2319 - val_mean_squared_error: 2.2319
Epoch 115/200
800/800 [==============================] - 0s 164us/sample - loss: 4.0247 -
mean_squared_error: 4.0247 - val_loss: 1.3910 - val_mean_squared_error: 1.3910
Epoch 116/200
800/800 [==============================] - 0s 164us/sample - loss: 3.1407 -
mean_squared_error: 3.1407 - val_loss: 2.1261 - val_mean_squared_error: 2.1261
Epoch 117/200
800/800 [==============================] - 0s 163us/sample - loss: 1.4969 -
mean_squared_error: 1.4969 - val_loss: 0.8010 - val_mean_squared_error: 0.8010
Epoch 118/200
800/800 [==============================] - 0s 165us/sample - loss: 1.0634 -
mean_squared_error: 1.0634 - val_loss: 0.0772 - val_mean_squared_error: 0.0772
Epoch 119/200
800/800 [==============================] - 0s 165us/sample - loss: 1.0480 -
mean_squared_error: 1.0480 - val_loss: 0.7540 - val_mean_squared_error: 0.7540
Epoch 120/200
800/800 [==============================] - 0s 165us/sample - loss: 0.7412 -
mean_squared_error: 0.7412 - val_loss: 0.4868 - val_mean_squared_error: 0.4868
Epoch 121/200
800/800 [==============================] - 0s 162us/sample - loss: 0.8776 -
mean_squared_error: 0.8776 - val_loss: 0.0580 - val_mean_squared_error: 0.0580
Epoch 122/200
800/800 [==============================] - 0s 164us/sample - loss: 1.4105 -
mean_squared_error: 1.4105 - val_loss: 0.3351 - val_mean_squared_error: 0.3351
Epoch 123/200
800/800 [==============================] - 0s 164us/sample - loss: 1.0251 -
mean_squared_error: 1.0251 - val_loss: 0.1224 - val_mean_squared_error: 0.1224
Epoch 124/200
800/800 [==============================] - 0s 164us/sample - loss: 2.2066 -
mean_squared_error: 2.2066 - val_loss: 0.1259 - val_mean_squared_error: 0.1259
```

```
Epoch 125/200
800/800 [==============================] - 0s 164us/sample - loss: 0.8138 -
mean_squared_error: 0.8138 - val_loss: 0.2388 - val_mean_squared_error: 0.2388
Epoch 126/200
800/800 [==============================] - 0s 165us/sample - loss: 0.5980 -
mean_squared_error: 0.5980 - val_loss: 0.0403 - val_mean_squared_error: 0.0403
Epoch 127/200
800/800 [==============================] - 0s 168us/sample - loss: 1.4867 -
mean_squared_error: 1.4867 - val_loss: 0.2377 - val_mean_squared_error: 0.2377
Epoch 128/200
800/800 [==============================] - 0s 164us/sample - loss: 0.7055 -
mean_squared_error: 0.7055 - val_loss: 0.8655 - val_mean_squared_error: 0.8655
Epoch 129/200
800/800 [==============================] - 0s 162us/sample - loss: 1.0175 -
mean_squared_error: 1.0175 - val_loss: 0.1218 - val_mean_squared_error: 0.1218
Epoch 130/200
800/800 [==============================] - 0s 163us/sample - loss: 0.6498 -
mean_squared_error: 0.6498 - val_loss: 0.0590 - val_mean_squared_error: 0.0590
Epoch 131/200
800/800 [==============================] - 0s 164us/sample - loss: 2.8154 -
mean_squared_error: 2.8154 - val_loss: 3.3983 - val_mean_squared_error: 3.3983
Epoch 132/200
800/800 [==============================] - 0s 168us/sample - loss: 4.4634 -
mean_squared_error: 4.4634 - val_loss: 1.2875 - val_mean_squared_error: 1.2875
Epoch 133/200
800/800 [==============================] - 0s 176us/sample - loss: 3.6024 -
mean_squared_error: 3.6024 - val_loss: 0.0608 - val_mean_squared_error: 0.0608
Epoch 134/200
800/800 [==============================] - 0s 173us/sample - loss: 3.0473 -
mean_squared_error: 3.0473 - val_loss: 1.9100 - val_mean_squared_error: 1.9100
Epoch 135/200
800/800 [==============================] - 0s 169us/sample - loss: 4.0936 -
mean_squared_error: 4.0936 - val_loss: 0.1507 - val_mean_squared_error: 0.1507
Epoch 136/200
800/800 [==============================] - 0s 169us/sample - loss: 1.7221 -
mean_squared_error: 1.7221 - val_loss: 0.3062 - val_mean_squared_error: 0.3062
Epoch 137/200
800/800 [==============================] - 0s 170us/sample - loss: 1.5338 -
mean_squared_error: 1.5338 - val_loss: 0.0822 - val_mean_squared_error: 0.0822
Epoch 138/200
800/800 [==============================] - 0s 173us/sample - loss: 1.2638 -
mean_squared_error: 1.2638 - val_loss: 0.5463 - val_mean_squared_error: 0.5463
Epoch 139/200
800/800 [==============================] - 0s 177us/sample - loss: 1.9048 -
mean_squared_error: 1.9048 - val_loss: 0.3713 - val_mean_squared_error: 0.3713
Epoch 140/200
800/800 [==============================] - 0s 187us/sample - loss: 0.4721 -
mean_squared_error: 0.4721 - val_loss: 1.0615 - val_mean_squared_error: 1.0615
```

```
Epoch 141/200
800/800 [==============================] - 0s 167us/sample - loss: 1.3769 -
mean_squared_error: 1.3769 - val_loss: 0.3407 - val_mean_squared_error: 0.3407
Epoch 142/200
800/800 [==============================] - 0s 181us/sample - loss: 3.6319 -
mean_squared_error: 3.6319 - val_loss: 0.4527 - val_mean_squared_error: 0.4527
Epoch 143/200
800/800 [==============================] - 0s 171us/sample - loss: 1.6032 -
mean_squared_error: 1.6032 - val_loss: 0.0542 - val_mean_squared_error: 0.0542
Epoch 144/200
800/800 [==============================] - 0s 166us/sample - loss: 0.8050 -
mean_squared_error: 0.8050 - val_loss: 0.9822 - val_mean_squared_error: 0.9822
Epoch 145/200
800/800 [==============================] - 0s 184us/sample - loss: 1.6522 -
mean_squared_error: 1.6522 - val_loss: 0.1523 - val_mean_squared_error: 0.1523
Epoch 146/200
800/800 [==============================] - 0s 184us/sample - loss: 0.9615 -
mean_squared_error: 0.9615 - val_loss: 2.0972 - val_mean_squared_error: 2.0972
Epoch 147/200
800/800 [==============================] - 0s 187us/sample - loss: 0.9668 -
mean_squared_error: 0.9668 - val_loss: 0.0719 - val_mean_squared_error: 0.0719
Epoch 148/200
800/800 [==============================] - 0s 171us/sample - loss: 1.8674 -
mean_squared_error: 1.8674 - val_loss: 0.7999 - val_mean_squared_error: 0.7999
Epoch 149/200
800/800 [==============================] - 0s 171us/sample - loss: 1.4414 -
mean_squared_error: 1.4414 - val_loss: 0.0463 - val_mean_squared_error: 0.0463
Epoch 150/200
800/800 [==============================] - 0s 176us/sample - loss: 0.8697 -
mean_squared_error: 0.8697 - val_loss: 0.4092 - val_mean_squared_error: 0.4092
Epoch 151/200
800/800 [==============================] - 0s 170us/sample - loss: 1.7901 -
mean_squared_error: 1.7901 - val_loss: 0.6613 - val_mean_squared_error: 0.6613
Epoch 152/200
800/800 [==============================] - 0s 178us/sample - loss: 3.6607 -
mean_squared_error: 3.6607 - val_loss: 0.6957 - val_mean_squared_error: 0.6957
Epoch 153/200
800/800 [==============================] - 0s 169us/sample - loss: 0.5969 -
mean_squared_error: 0.5969 - val_loss: 0.1174 - val_mean_squared_error: 0.1174
Epoch 154/200
800/800 [==============================] - 0s 167us/sample - loss: 3.0329 -
mean_squared_error: 3.0329 - val_loss: 2.1010 - val_mean_squared_error: 2.1010
Epoch 155/200
800/800 [==============================] - 0s 168us/sample - loss: 2.6653 -
mean_squared_error: 2.6653 - val_loss: 0.9864 - val_mean_squared_error: 0.9864
Epoch 156/200
800/800 [==============================] - 0s 167us/sample - loss: 3.0555 -
mean_squared_error: 3.0555 - val_loss: 5.8271 - val_mean_squared_error: 5.8271
```

```
Epoch 157/200
800/800 [==============================] - 0s 169us/sample - loss: 4.9626 -
mean_squared_error: 4.9626 - val_loss: 1.4493 - val_mean_squared_error: 1.4493
Epoch 158/200
800/800 [==============================] - 0s 163us/sample - loss: 8.1173 -
mean_squared_error: 8.1173 - val_loss: 1.2280 - val_mean_squared_error: 1.2280
Epoch 159/200
800/800 [==============================] - 0s 172us/sample - loss: 3.6938 -
mean_squared_error: 3.6938 - val_loss: 0.3014 - val_mean_squared_error: 0.3014
Epoch 160/200
800/800 [==============================] - 0s 172us/sample - loss: 1.3357 -
mean_squared_error: 1.3357 - val_loss: 0.7937 - val_mean_squared_error: 0.7937
Epoch 161/200
800/800 [==============================] - 0s 168us/sample - loss: 6.0258 -
mean_squared_error: 6.0258 - val_loss: 9.0951 - val_mean_squared_error: 9.0951
Epoch 162/200
800/800 [==============================] - 0s 170us/sample - loss: 25.3645 -
mean_squared_error: 25.3645 - val_loss: 7.1941 - val_mean_squared_error: 7.1941
Epoch 163/200
800/800 [==============================] - 0s 176us/sample - loss: 6.4119 -
mean_squared_error: 6.4119 - val_loss: 0.2369 - val_mean_squared_error: 0.2369
Epoch 164/200
800/800 [==============================] - 0s 172us/sample - loss: 1.0405 -
mean_squared_error: 1.0405 - val_loss: 1.7439 - val_mean_squared_error: 1.7439
Epoch 165/200
800/800 [==============================] - 0s 174us/sample - loss: 1.4930 -
mean_squared_error: 1.4930 - val_loss: 1.0095 - val_mean_squared_error: 1.0095
Epoch 166/200
800/800 [==============================] - 0s 174us/sample - loss: 0.6536 -
mean_squared_error: 0.6536 - val_loss: 1.3059 - val_mean_squared_error: 1.3059
Epoch 167/200
800/800 [==============================] - 0s 167us/sample - loss: 1.3529 -
mean_squared_error: 1.3529 - val_loss: 0.3710 - val_mean_squared_error: 0.3710
Epoch 168/200
800/800 [==============================] - 0s 167us/sample - loss: 2.6145 -
mean_squared_error: 2.6145 - val_loss: 1.0135 - val_mean_squared_error: 1.0135
Epoch 169/200
800/800 [==============================] - 0s 169us/sample - loss: 1.1393 -
mean_squared_error: 1.1393 - val_loss: 1.5369 - val_mean_squared_error: 1.5369
Epoch 170/200
800/800 [==============================] - 0s 166us/sample - loss: 2.2096 -
mean_squared_error: 2.2096 - val_loss: 0.5578 - val_mean_squared_error: 0.5578
Epoch 171/200
800/800 [==============================] - 0s 168us/sample - loss: 1.8896 -
mean_squared_error: 1.8896 - val_loss: 4.3285 - val_mean_squared_error: 4.3285
Epoch 172/200
800/800 [==============================] - 0s 165us/sample - loss: 21.5844 -
mean_squared_error: 21.5844 - val_loss: 24.7272 - val_mean_squared_error:
```

```
24.7272
Epoch 173/200
800/800 [==============================] - 0s 171us/sample - loss: 100.7402 -
mean_squared_error: 100.7402 - val_loss: 21.8386 - val_mean_squared_error:
21.8386
Epoch 174/200
800/800 [==============================] - 0s 177us/sample - loss: 93.4701 -
mean_squared_error: 93.4701 - val_loss: 10.9511 - val_mean_squared_error:
10.9511
Epoch 175/200
800/800 [==============================] - 0s 173us/sample - loss: 22.2627 -
mean_squared_error: 22.2627 - val_loss: 1.3486 - val_mean_squared_error: 1.3486
Epoch 176/200
800/800 [==============================] - 0s 183us/sample - loss: 2.7636 -
mean_squared_error: 2.7636 - val_loss: 1.1765 - val_mean_squared_error: 1.1765
Epoch 177/200
800/800 [==============================] - 0s 208us/sample - loss: 1.8887 -
mean_squared_error: 1.8887 - val_loss: 0.2239 - val_mean_squared_error: 0.2239
Epoch 178/200
800/800 [==============================] - 0s 203us/sample - loss: 1.6538 -
mean_squared_error: 1.6538 - val_loss: 0.0929 - val_mean_squared_error: 0.0929
Epoch 179/200
800/800 [==============================] - 0s 191us/sample - loss: 0.9651 -
mean_squared_error: 0.9651 - val_loss: 0.9761 - val_mean_squared_error: 0.9761
Epoch 180/200
800/800 [==============================] - 0s 169us/sample - loss: 2.4304 -
mean_squared_error: 2.4304 - val_loss: 0.6722 - val_mean_squared_error: 0.6722
Epoch 181/200
800/800 [==============================] - 0s 175us/sample - loss: 1.5966 -
mean_squared_error: 1.5966 - val_loss: 1.1192 - val_mean_squared_error: 1.1192
Epoch 182/200
800/800 [==============================] - 0s 167us/sample - loss: 2.6022 -
mean_squared_error: 2.6022 - val_loss: 0.1911 - val_mean_squared_error: 0.1911
Epoch 183/200
800/800 [==============================] - 0s 167us/sample - loss: 1.6265 -
mean_squared_error: 1.6265 - val_loss: 0.5233 - val_mean_squared_error: 0.5233
Epoch 184/200
800/800 [==============================] - 0s 165us/sample - loss: 1.8476 -
mean_squared_error: 1.8476 - val_loss: 0.4470 - val_mean_squared_error: 0.4470
Epoch 185/200
800/800 [==============================] - 0s 162us/sample - loss: 0.9064 -
mean_squared_error: 0.9064 - val_loss: 0.6652 - val_mean_squared_error: 0.6652
Epoch 186/200
800/800 [==============================] - 0s 162us/sample - loss: 0.4888 -
mean_squared_error: 0.4888 - val_loss: 0.9737 - val_mean_squared_error: 0.9737
Epoch 187/200
800/800 [==============================] - 0s 163us/sample - loss: 0.6484 -
mean_squared_error: 0.6484 - val_loss: 0.4239 - val_mean_squared_error: 0.4239
```

```
Epoch 188/200
800/800 [==============================] - 0s 169us/sample - loss: 0.4810 -
mean_squared_error: 0.4810 - val_loss: 0.1709 - val_mean_squared_error: 0.1709
Epoch 189/200
800/800 [==============================] - 0s 167us/sample - loss: 0.7558 -
mean_squared_error: 0.7558 - val_loss: 0.1735 - val_mean_squared_error: 0.1735
Epoch 190/200
800/800 [==============================] - 0s 167us/sample - loss: 0.5442 -
mean_squared_error: 0.5442 - val_loss: 1.3083 - val_mean_squared_error: 1.3083
Epoch 191/200
800/800 [==============================] - 0s 164us/sample - loss: 1.2037 -
mean_squared_error: 1.2037 - val_loss: 0.2205 - val_mean_squared_error: 0.2205
Epoch 192/200
800/800 [==============================] - 0s 166us/sample - loss: 0.6694 -
mean_squared_error: 0.6694 - val_loss: 0.5916 - val_mean_squared_error: 0.5916
Epoch 193/200
800/800 [==============================] - 0s 166us/sample - loss: 0.3951 -
mean_squared_error: 0.3951 - val_loss: 0.1963 - val_mean_squared_error: 0.1963
Epoch 194/200
800/800 [==============================] - 0s 169us/sample - loss: 0.4530 -
mean_squared_error: 0.4530 - val_loss: 0.8369 - val_mean_squared_error: 0.8369
Epoch 195/200
800/800 [==============================] - 0s 170us/sample - loss: 1.5567 -
mean_squared_error: 1.5567 - val_loss: 2.0729 - val_mean_squared_error: 2.0729
Epoch 196/200
800/800 [==============================] - 0s 172us/sample - loss: 0.4819 -
mean_squared_error: 0.4819 - val_loss: 0.2141 - val_mean_squared_error: 0.2141
Epoch 197/200
800/800 [==============================] - 0s 173us/sample - loss: 0.4528 -
mean_squared_error: 0.4528 - val_loss: 0.1054 - val_mean_squared_error: 0.1054
Epoch 198/200
800/800 [==============================] - 0s 169us/sample - loss: 0.8813 -
mean_squared_error: 0.8813 - val_loss: 0.1139 - val_mean_squared_error: 0.1139
Epoch 199/200
800/800 [==============================] - 0s 169us/sample - loss: 0.3837 -
mean_squared_error: 0.3837 - val_loss: 0.6312 - val_mean_squared_error: 0.6312
Epoch 200/200
800/800 [==============================] - 0s 175us/sample - loss: 0.6771 -
mean_squared_error: 0.6771 - val_loss: 0.5067 - val_mean_squared_error: 0.5067
```

[24]: `<tensorflow.python.keras.callbacks.History at 0x11e593c18>`

1.4.3 Nonlinearities (15 mts)

The reason why our linear model fails here while a neural network trains quickly and effectively is because of the *nonlinearities* in our dataset. We defined our `toy` column as a cubic polynomial of `year` and `price` – by definition, a linear model will not be able to understand the underlying

factors. A neural network, on the other hand, is composed of a variety of linear functions and activation functions, which allows it to predict more complex nonlinear models.

In the above contrived example, we can see why the data was fundamentally nonlinear. In the real world, these connections are far less clear. This is the reason why linear models are often reasonably good at approximating relationships in most real-world datasets. However, there are times when neural networks can really demonstrate their power.

Let's go ahead and build our model. We'll use a simple *Sequential* constructor from Keras. Take a look at the description of how to use Keras Sequential models here: https://keras.io/getting-started/sequential-model-guide/. The following code constructs a simple Keras Sequential instance with one layer. We need to specify the *activation function* and the *input shape* in our first model:

[25]:
```
neurons = 128

model = keras.Sequential([layers.Dense(neurons, activation='relu',␣
↪input_shape=[len(df_train.columns) - 1,]), # Input layer
                          layers.Dense(1)]) # Output layer
```

In addition to the layers of our model, we need to specify three more parameters:

1. The *loss function* – this function will define how "wrong" our final answer is.
2. The *optimizer* – this is the algorithm that minimizes our loss function by fine-tuning the weights in our neural network.
3. A list of metrics that determine the performance of our model – you can understand the metrics that Keras uses here: https://keras.io/metrics/

These specifications go in the model's compile function, with the following parameters:

[26]:
```
model.compile(loss='mse', # This uses Mean-Squared Error (https://en.wikipedia.
↪org/wiki/Mean_squared_error)
              optimizer = 'adam', # The algorithm to optimize, root means␣
↪squared is useful for regression
              metrics = ['mse'] # This is a good metric to use by default for␣
↪any classification problem
             )
```

For a neural network, training the model consists of tuning each of the weights between every node by minimizing the difference between the model's predicted value and the actual value in the training dataset. After this, the model is run on the test set to see how well it generalizes:

[27]:
```
X = df_train.drop('price', axis=1)
X['year'] = (X['year'] - df_train['year'].mean()) / df_train['year'].std()

y = df_train['price']
y = (y - df_train['price'].mean()) / df_train['price'].std()

X_test = df_test.drop('price', axis=1)
X_test['year'] = (X_test['year'] - df_test['year'].mean()) / df_test['year'].
↪std()
```

```
y_test = df_test['price']
y_test = (y_test - df_test['price'].mean()) / df_test['price'].std()
```

```
/Users/haris.jaliawala/anaconda3/envs/week8/lib/python3.7/site-
packages/ipykernel_launcher.py:2: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy

/Users/haris.jaliawala/anaconda3/envs/week8/lib/python3.7/site-
packages/ipykernel_launcher.py:8: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
```

We're now ready to finally train the model! This uses the Keras `fit` function, which uses the following parameters:

1. Input data
2. Input labels
3. Number of epochs
4. Verbosity: 0, 1, or 2, depending on how frequently you want your model to give information while logging information
5. Validation split: A fraction of the training data to *avoid* use while training, in order to prevent the model from overfitting on a subset of the data.
6. Batch size: This tells you how many data points will train at once. A higher batch size will be faster until a certain point, until the algorithm hits diminishing returns from overhead. Higher batch size may also reduce accuracy.

Here's an example of fitting the data with some sample parameters.

[28]:
```
history = model.fit(X, y, epochs=10, validation_split = 0.2, verbose=1,␣
↪batch_size=1000)
```

```
Train on 268747 samples, validate on 67187 samples
Epoch 1/10
268747/268747 [==============================] - 2s 6us/sample - loss: 0.5691 -
mean_squared_error: 0.5691 - val_loss: 0.5029 - val_mean_squared_error: 0.5029
Epoch 2/10
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4994 -
mean_squared_error: 0.4994 - val_loss: 0.4905 - val_mean_squared_error: 0.4905
Epoch 3/10
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4864 -
mean_squared_error: 0.4864 - val_loss: 0.4840 - val_mean_squared_error: 0.4840
```

```
Epoch 4/10
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4778 -
mean_squared_error: 0.4778 - val_loss: 0.4791 - val_mean_squared_error: 0.4791
Epoch 5/10
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4714 -
mean_squared_error: 0.4714 - val_loss: 0.4711 - val_mean_squared_error: 0.4711
Epoch 6/10
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4665 -
mean_squared_error: 0.4665 - val_loss: 0.4693 - val_mean_squared_error: 0.4693
Epoch 7/10
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4628 -
mean_squared_error: 0.4628 - val_loss: 0.4654 - val_mean_squared_error: 0.4654
Epoch 8/10
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4593 -
mean_squared_error: 0.4593 - val_loss: 0.4644 - val_mean_squared_error: 0.4644
Epoch 9/10
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4568 -
mean_squared_error: 0.4568 - val_loss: 0.4604 - val_mean_squared_error: 0.4604
Epoch 10/10
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4547 -
mean_squared_error: 0.4547 - val_loss: 0.4599 - val_mean_squared_error: 0.4599
```

[29]:
```
model.evaluate(X_test, y_test)
```

```
98608/98608 [==============================] - 1s 10us/sample - loss: 0.4685 -
mean_squared_error: 0.4685
```

[29]:
```
[0.46847368223716856, 0.46847305]
```

Now that we've seen a basic model, let's try making this model deeper by adding more hidden layers. To add more layers, we just use the `add()` method, which takes layer instances as input in order to upgrade the model.

1.4.4 Exercise 5: (10 mts)

Use Keras to create a deep model with three layers to fit on our training data, then run it against our test data to see how well it fits. Compare this model to a linear regression which uses all features on our new normalized dataset.

Answer. One possible solution is given below:

[30]:
```
deep_model = keras.Sequential([layers.Dense(128, activation='relu',␣
 ↪input_shape=[len(X.columns),]),])
deep_model.add(layers.Dense(64, activation='softmax'))
deep_model.add(layers.Dense(1))

deep_model.compile(loss='mse', optimizer = 'adam', metrics = ['mse'])
```

[31]:
```
history = deep_model.fit(X, y, epochs=100, validation_split=0.2, verbose=1,␣
↪batch_size=2000)
```

```
Train on 268747 samples, validate on 67187 samples
Epoch 1/100
268747/268747 [==============================] - 2s 6us/sample - loss: 0.8179 -
mean_squared_error: 0.8179 - val_loss: 0.6503 - val_mean_squared_error: 0.6503
Epoch 2/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.6408 -
mean_squared_error: 0.6408 - val_loss: 0.6041 - val_mean_squared_error: 0.6041
Epoch 3/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.5961 -
mean_squared_error: 0.5961 - val_loss: 0.5729 - val_mean_squared_error: 0.5729
Epoch 4/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.5587 -
mean_squared_error: 0.5587 - val_loss: 0.5398 - val_mean_squared_error: 0.5398
Epoch 5/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.5353 -
mean_squared_error: 0.5353 - val_loss: 0.5208 - val_mean_squared_error: 0.5208
Epoch 6/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.5201 -
mean_squared_error: 0.5201 - val_loss: 0.5112 - val_mean_squared_error: 0.5112
Epoch 7/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.5090 -
mean_squared_error: 0.5090 - val_loss: 0.5030 - val_mean_squared_error: 0.5030
Epoch 8/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.5004 -
mean_squared_error: 0.5004 - val_loss: 0.4992 - val_mean_squared_error: 0.4992
Epoch 9/100
268747/268747 [==============================] - 1s 6us/sample - loss: 0.4932 -
mean_squared_error: 0.4932 - val_loss: 0.4908 - val_mean_squared_error: 0.4908
Epoch 10/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4872 -
mean_squared_error: 0.4872 - val_loss: 0.4852 - val_mean_squared_error: 0.4852
Epoch 11/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4829 -
mean_squared_error: 0.4829 - val_loss: 0.4801 - val_mean_squared_error: 0.4801
Epoch 12/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4787 -
mean_squared_error: 0.4787 - val_loss: 0.4802 - val_mean_squared_error: 0.4802
Epoch 13/100
268747/268747 [==============================] - 2s 6us/sample - loss: 0.4758 -
mean_squared_error: 0.4758 - val_loss: 0.4849 - val_mean_squared_error: 0.4849
Epoch 14/100
268747/268747 [==============================] - 2s 6us/sample - loss: 0.4730 -
mean_squared_error: 0.4730 - val_loss: 0.4745 - val_mean_squared_error: 0.4745
Epoch 15/100
```

```
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4694 -
mean_squared_error: 0.4694 - val_loss: 0.4712 - val_mean_squared_error: 0.4712
Epoch 16/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4679 -
mean_squared_error: 0.4679 - val_loss: 0.4793 - val_mean_squared_error: 0.4793
Epoch 17/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4666 -
mean_squared_error: 0.4666 - val_loss: 0.4672 - val_mean_squared_error: 0.4672
Epoch 18/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4632 -
mean_squared_error: 0.4632 - val_loss: 0.4660 - val_mean_squared_error: 0.4660
Epoch 19/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4622 -
mean_squared_error: 0.4622 - val_loss: 0.4632 - val_mean_squared_error: 0.4632
Epoch 20/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4604 -
mean_squared_error: 0.4604 - val_loss: 0.4632 - val_mean_squared_error: 0.4632
Epoch 21/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4591 -
mean_squared_error: 0.4591 - val_loss: 0.4625 - val_mean_squared_error: 0.4625
Epoch 22/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4577 -
mean_squared_error: 0.4577 - val_loss: 0.4618 - val_mean_squared_error: 0.4618
Epoch 23/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4566 -
mean_squared_error: 0.4566 - val_loss: 0.4585 - val_mean_squared_error: 0.4585
Epoch 24/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4552 -
mean_squared_error: 0.4552 - val_loss: 0.4637 - val_mean_squared_error: 0.4637
Epoch 25/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4542 -
mean_squared_error: 0.4542 - val_loss: 0.4585 - val_mean_squared_error: 0.4585
Epoch 26/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4525 -
mean_squared_error: 0.4525 - val_loss: 0.4573 - val_mean_squared_error: 0.4573
Epoch 27/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4523 -
mean_squared_error: 0.4523 - val_loss: 0.4607 - val_mean_squared_error: 0.4607
Epoch 28/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4513 -
mean_squared_error: 0.4513 - val_loss: 0.4554 - val_mean_squared_error: 0.4554
Epoch 29/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4502 -
mean_squared_error: 0.4502 - val_loss: 0.4569 - val_mean_squared_error: 0.4569
Epoch 30/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4503 -
mean_squared_error: 0.4503 - val_loss: 0.4600 - val_mean_squared_error: 0.4600
Epoch 31/100
```

```
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4485 -
mean_squared_error: 0.4485 - val_loss: 0.4568 - val_mean_squared_error: 0.4568
Epoch 32/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4478 -
mean_squared_error: 0.4478 - val_loss: 0.4542 - val_mean_squared_error: 0.4542
Epoch 33/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4472 -
mean_squared_error: 0.4472 - val_loss: 0.4530 - val_mean_squared_error: 0.4530
Epoch 34/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4467 -
mean_squared_error: 0.4467 - val_loss: 0.4572 - val_mean_squared_error: 0.4572
Epoch 35/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4454 -
mean_squared_error: 0.4454 - val_loss: 0.4563 - val_mean_squared_error: 0.4563
Epoch 36/100
268747/268747 [==============================] - 1s 6us/sample - loss: 0.4448 -
mean_squared_error: 0.4448 - val_loss: 0.4583 - val_mean_squared_error: 0.4583
Epoch 37/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4444 -
mean_squared_error: 0.4444 - val_loss: 0.4537 - val_mean_squared_error: 0.4537
Epoch 38/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4437 -
mean_squared_error: 0.4437 - val_loss: 0.4545 - val_mean_squared_error: 0.4545
Epoch 39/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4435 -
mean_squared_error: 0.4435 - val_loss: 0.4543 - val_mean_squared_error: 0.4543
Epoch 40/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4430 -
mean_squared_error: 0.4430 - val_loss: 0.4530 - val_mean_squared_error: 0.4530
Epoch 41/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4423 -
mean_squared_error: 0.4423 - val_loss: 0.4524 - val_mean_squared_error: 0.4524
Epoch 42/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4420 -
mean_squared_error: 0.4420 - val_loss: 0.4517 - val_mean_squared_error: 0.4517
Epoch 43/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4415 -
mean_squared_error: 0.4415 - val_loss: 0.4613 - val_mean_squared_error: 0.4613
Epoch 44/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4403 -
mean_squared_error: 0.4403 - val_loss: 0.4505 - val_mean_squared_error: 0.4505
Epoch 45/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4400 -
mean_squared_error: 0.4400 - val_loss: 0.4550 - val_mean_squared_error: 0.4550
Epoch 46/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4394 -
mean_squared_error: 0.4394 - val_loss: 0.4551 - val_mean_squared_error: 0.4551
Epoch 47/100
```

```
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4391 -
mean_squared_error: 0.4391 - val_loss: 0.4576 - val_mean_squared_error: 0.4576
Epoch 48/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4385 -
mean_squared_error: 0.4385 - val_loss: 0.4516 - val_mean_squared_error: 0.4516
Epoch 49/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4386 -
mean_squared_error: 0.4386 - val_loss: 0.4487 - val_mean_squared_error: 0.4487
Epoch 50/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4376 -
mean_squared_error: 0.4376 - val_loss: 0.4492 - val_mean_squared_error: 0.4492
Epoch 51/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4371 -
mean_squared_error: 0.4371 - val_loss: 0.4501 - val_mean_squared_error: 0.4501
Epoch 52/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4369 -
mean_squared_error: 0.4369 - val_loss: 0.4511 - val_mean_squared_error: 0.4511
Epoch 53/100
268747/268747 [==============================] - 1s 6us/sample - loss: 0.4363 -
mean_squared_error: 0.4363 - val_loss: 0.4506 - val_mean_squared_error: 0.4506
Epoch 54/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4364 -
mean_squared_error: 0.4364 - val_loss: 0.4572 - val_mean_squared_error: 0.4572
Epoch 55/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4352 -
mean_squared_error: 0.4352 - val_loss: 0.4516 - val_mean_squared_error: 0.4516
Epoch 56/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4353 -
mean_squared_error: 0.4353 - val_loss: 0.4474 - val_mean_squared_error: 0.4474
Epoch 57/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4352 -
mean_squared_error: 0.4352 - val_loss: 0.4524 - val_mean_squared_error: 0.4524
Epoch 58/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4349 -
mean_squared_error: 0.4349 - val_loss: 0.4531 - val_mean_squared_error: 0.4531
Epoch 59/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4349 -
mean_squared_error: 0.4349 - val_loss: 0.4500 - val_mean_squared_error: 0.4500
Epoch 60/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4336 -
mean_squared_error: 0.4336 - val_loss: 0.4493 - val_mean_squared_error: 0.4493
Epoch 61/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4329 -
mean_squared_error: 0.4329 - val_loss: 0.4488 - val_mean_squared_error: 0.4488
Epoch 62/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4322 -
mean_squared_error: 0.4322 - val_loss: 0.4487 - val_mean_squared_error: 0.4487
Epoch 63/100
```

```
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4338 -
mean_squared_error: 0.4338 - val_loss: 0.4479 - val_mean_squared_error: 0.4479
Epoch 64/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4322 -
mean_squared_error: 0.4322 - val_loss: 0.4573 - val_mean_squared_error: 0.4573
Epoch 65/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4328 -
mean_squared_error: 0.4328 - val_loss: 0.4481 - val_mean_squared_error: 0.4481
Epoch 66/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4322 -
mean_squared_error: 0.4322 - val_loss: 0.4545 - val_mean_squared_error: 0.4545
Epoch 67/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4310 -
mean_squared_error: 0.4310 - val_loss: 0.4498 - val_mean_squared_error: 0.4498
Epoch 68/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4305 -
mean_squared_error: 0.4305 - val_loss: 0.4495 - val_mean_squared_error: 0.4495
Epoch 69/100
268747/268747 [==============================] - 1s 6us/sample - loss: 0.4304 -
mean_squared_error: 0.4304 - val_loss: 0.4510 - val_mean_squared_error: 0.4510
Epoch 70/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4305 -
mean_squared_error: 0.4305 - val_loss: 0.4479 - val_mean_squared_error: 0.4479
Epoch 71/100
268747/268747 [==============================] - 1s 6us/sample - loss: 0.4309 -
mean_squared_error: 0.4309 - val_loss: 0.4485 - val_mean_squared_error: 0.4485
Epoch 72/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4293 -
mean_squared_error: 0.4293 - val_loss: 0.4467 - val_mean_squared_error: 0.4467
Epoch 73/100
268747/268747 [==============================] - 1s 6us/sample - loss: 0.4311 -
mean_squared_error: 0.4311 - val_loss: 0.4480 - val_mean_squared_error: 0.4480
Epoch 74/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4292 -
mean_squared_error: 0.4292 - val_loss: 0.4477 - val_mean_squared_error: 0.4477
Epoch 75/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4289 -
mean_squared_error: 0.4289 - val_loss: 0.4513 - val_mean_squared_error: 0.4513
Epoch 76/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4281 -
mean_squared_error: 0.4281 - val_loss: 0.4466 - val_mean_squared_error: 0.4466
Epoch 77/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4279 -
mean_squared_error: 0.4279 - val_loss: 0.4475 - val_mean_squared_error: 0.4475
Epoch 78/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4276 -
mean_squared_error: 0.4276 - val_loss: 0.4498 - val_mean_squared_error: 0.4498
Epoch 79/100
```

```
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4279 -
mean_squared_error: 0.4279 - val_loss: 0.4467 - val_mean_squared_error: 0.4467
Epoch 80/100
268747/268747 [==============================] - 1s 6us/sample - loss: 0.4279 -
mean_squared_error: 0.4279 - val_loss: 0.4484 - val_mean_squared_error: 0.4484
Epoch 81/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4268 -
mean_squared_error: 0.4268 - val_loss: 0.4538 - val_mean_squared_error: 0.4538
Epoch 82/100
268747/268747 [==============================] - 2s 6us/sample - loss: 0.4267 -
mean_squared_error: 0.4267 - val_loss: 0.4485 - val_mean_squared_error: 0.4485
Epoch 83/100
268747/268747 [==============================] - 1s 6us/sample - loss: 0.4263 -
mean_squared_error: 0.4263 - val_loss: 0.4456 - val_mean_squared_error: 0.4456
Epoch 84/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4264 -
mean_squared_error: 0.4264 - val_loss: 0.4467 - val_mean_squared_error: 0.4467
Epoch 85/100
268747/268747 [==============================] - 1s 6us/sample - loss: 0.4261 -
mean_squared_error: 0.4261 - val_loss: 0.4532 - val_mean_squared_error: 0.4532
Epoch 86/100
268747/268747 [==============================] - 2s 6us/sample - loss: 0.4253 -
mean_squared_error: 0.4253 - val_loss: 0.4457 - val_mean_squared_error: 0.4457
Epoch 87/100
268747/268747 [==============================] - 2s 6us/sample - loss: 0.4255 -
mean_squared_error: 0.4255 - val_loss: 0.4450 - val_mean_squared_error: 0.4450
Epoch 88/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4251 -
mean_squared_error: 0.4251 - val_loss: 0.4538 - val_mean_squared_error: 0.4538
Epoch 89/100
268747/268747 [==============================] - 1s 6us/sample - loss: 0.4250 -
mean_squared_error: 0.4250 - val_loss: 0.4522 - val_mean_squared_error: 0.4522
Epoch 90/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4241 -
mean_squared_error: 0.4241 - val_loss: 0.4524 - val_mean_squared_error: 0.4524
Epoch 91/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4243 -
mean_squared_error: 0.4243 - val_loss: 0.4472 - val_mean_squared_error: 0.4472
Epoch 92/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4239 -
mean_squared_error: 0.4239 - val_loss: 0.4453 - val_mean_squared_error: 0.4453
Epoch 93/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4247 -
mean_squared_error: 0.4247 - val_loss: 0.4456 - val_mean_squared_error: 0.4456
Epoch 94/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4233 -
mean_squared_error: 0.4233 - val_loss: 0.4445 - val_mean_squared_error: 0.4445
Epoch 95/100
```

```
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4233 -
mean_squared_error: 0.4233 - val_loss: 0.4516 - val_mean_squared_error: 0.4516
Epoch 96/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4232 -
mean_squared_error: 0.4232 - val_loss: 0.4471 - val_mean_squared_error: 0.4471
Epoch 97/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4230 -
mean_squared_error: 0.4230 - val_loss: 0.4439 - val_mean_squared_error: 0.4439
Epoch 98/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4230 -
mean_squared_error: 0.4230 - val_loss: 0.4433 - val_mean_squared_error: 0.4433
Epoch 99/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4218 -
mean_squared_error: 0.4218 - val_loss: 0.4530 - val_mean_squared_error: 0.4530
Epoch 100/100
268747/268747 [==============================] - 1s 5us/sample - loss: 0.4220 -
mean_squared_error: 0.4220 - val_loss: 0.4495 - val_mean_squared_error: 0.4495
```

Notice how in this case, the MSE is generally higher, but it's falling more consistently. In small neural networks, minor changes in the gradient optimization can cause the MSE to fluctuate wildly, whereas large neural networks have the possibility of fitting more closely, but much more slowly. If we let it train for 10^8 epochs, we can see this eventually training perfectly. This is one of the main tradeoffs of neural networks – the tradeoff between accuracy and efficiency.

[32]:
```
deep_model.evaluate(X_test, y_test, batch_size=2000)
```

```
98608/98608 [==============================] - 0s 1us/sample - loss: 0.4533 -
mean_squared_error: 0.4533
```

[32]:
```
[0.45327300783026125, 0.45327297]
```

[33]:
```
comparison_model = 'price~year'
problematic_columns = set(['4wd', 'alfa-romeo', 'aston-martin',␣
↪'harley-davidson',
                           'mercedes-benz', 'full-size', 'mid-size',␣
↪'sub-compact', 'mini-van'])
for col in X.columns[1:]:
    if len(col.split()) == 1 and col not in problematic_columns:
        comparison_model = comparison_model + ' + ' + col
    else:
        comparison_model = comparison_model + ' + Q("' + col + '")'

lml = sm.ols(formula=comparison_model, data=pd.concat([X, y], axis=1)).fit()
lml.summary()
```

[33]:
```
<class 'statsmodels.iolib.summary.Summary'>
"""
                            OLS Regression Results
```

```
==============================================================================
Dep. Variable:                  price   R-squared:                       0.292
Model:                            OLS   Adj. R-squared:                  0.291
Method:                 Least Squares   F-statistic:                     1503.
Date:                Fri, 29 Nov 2019   Prob (F-statistic):               0.00
Time:                        19:02:26   Log-Likelihood:            -4.1875e+05
No. Observations:              335934   AIC:                         8.377e+05
Df Residuals:                  335841   BIC:                         8.387e+05
Df Model:                          92
Covariance Type:            nonrobust
==============================================================================
========
                           coef    std err          t      P>|t|      [0.025
0.975]
------------------------------------------------------------------------------
--------
Intercept                0.3920      0.022     17.730      0.000       0.349
0.435
year                     0.2830      0.002    163.834      0.000       0.280
0.286
excellent               -0.1283      0.004    -32.257      0.000      -0.136
-0.121
fair                    -0.6583      0.009    -69.631      0.000      -0.677
-0.640
good                    -0.2713      0.004    -63.857      0.000      -0.280
-0.263
Q("like new")            0.0255      0.007      3.871      0.000       0.013
0.038
new                      0.1195      0.032      3.790      0.000       0.058
0.181
salvage[0]              -0.6292      0.034    -18.704      0.000      -0.695
-0.563
salvage[1]              -0.2062      0.023     -8.861      0.000      -0.252
-0.161
clean                   -0.0822      0.018     -4.570      0.000      -0.117
-0.047
lien                     0.3621      0.024     14.880      0.000       0.314
0.410
missing                  0.3247      0.042      7.720      0.000       0.242
0.407
Q("parts only")         -0.2260      0.089     -2.527      0.011      -0.401
-0.051
rebuilt                 -0.1078      0.021     -5.160      0.000      -0.149
-0.067
automatic               -0.2115      0.008    -26.566      0.000      -0.227
-0.196
manual                  -0.0904      0.010     -9.362      0.000      -0.109
```

```
-0.072
Q("4wd")                 0.2317      0.005     45.990      0.000        0.222
0.242
fwd                     -0.1667      0.005    -32.362      0.000       -0.177
-0.157
rwd                      0.1567      0.006     27.552      0.000        0.146
0.168
acura                   -0.1757      0.018     -9.993      0.000       -0.210
-0.141
Q("alfa-romeo")          0.6314      0.104      6.073      0.000        0.428
0.835
Q("aston-martin")        1.7221      0.243      7.081      0.000        1.245
2.199
audi                    -0.0883      0.016     -5.654      0.000       -0.119
-0.058
bmw                     -0.1370      0.012    -11.479      0.000       -0.160
-0.114
buick                   -0.1340      0.015     -8.937      0.000       -0.163
-0.105
cadillac                -0.0236      0.014     -1.653      0.098       -0.052
0.004
chevrolet               -0.1108      0.009    -12.747      0.000       -0.128
-0.094
chrysler                -0.3301      0.014    -24.293      0.000       -0.357
-0.303
datsun                   0.5399      0.104      5.189      0.000        0.336
0.744
dodge                   -0.2928      0.011    -26.896      0.000       -0.314
-0.272
ferrari                 -0.6037      0.421     -1.434      0.152       -1.429
0.221
fiat                    -0.3656      0.038     -9.587      0.000       -0.440
-0.291
ford                    -0.1906      0.009    -22.273      0.000       -0.207
-0.174
gmc                     -0.0440      0.011     -4.136      0.000       -0.065
-0.023
Q("harley-davidson")    -0.3369      0.079     -4.267      0.000       -0.492
-0.182
hennessey                1.9937      0.842      2.368      0.018        0.344
3.644
honda                   -0.2214      0.010    -21.557      0.000       -0.241
-0.201
hyundai                 -0.3132      0.012    -25.427      0.000       -0.337
-0.289
infiniti                -0.1441      0.018     -7.867      0.000       -0.180
-0.108
```

jaguar	-0.0025	0.032	-0.079	0.937	-0.065 0.060
jeep	-0.1361	0.011	-12.923	0.000	-0.157 -0.115
kia	-0.2975	0.014	-21.581	0.000	-0.325 -0.271
Q("land rover")	-0.0698	0.204	-0.342	0.732	-0.470 0.331
lexus	0.0939	0.015	6.246	0.000	0.064 0.123
lincoln	-0.1900	0.019	-9.854	0.000	-0.228 -0.152
mazda	-0.2665	0.015	-17.662	0.000	-0.296 -0.237
Q("mercedes-benz")	0.0025	0.013	0.190	0.850	-0.023 0.028
mercury	-0.4123	0.024	-17.121	0.000	-0.459 -0.365
mini	-0.2405	0.023	-10.267	0.000	-0.286 -0.195
mitsubishi	-0.3954	0.021	-18.753	0.000	-0.437 -0.354
morgan	3.0799	0.842	3.659	0.000	1.430 4.730
nissan	-0.3176	0.010	-30.985	0.000	-0.338 -0.298
pontiac	-0.2690	0.018	-14.549	0.000	-0.305 -0.233
porche	-0.2190	0.234	-0.937	0.349	-0.677 0.239
ram	-0.0539	0.011	-5.053	0.000	-0.075 -0.033
rover	0.3626	0.024	15.153	0.000	0.316 0.409
saturn	-0.4758	0.025	-19.191	0.000	-0.524 -0.427
subaru	-0.2622	0.013	-19.983	0.000	-0.288 -0.236
toyota	-0.0540	0.010	-5.659	0.000	-0.073 -0.035
volkswagen	-0.2980	0.013	-22.934	0.000	-0.323 -0.273
volvo	-0.1578	0.020	-7.719	0.000	-0.198 -0.118
diesel	0.5082	0.009	54.639	0.000	0.490 0.526
electric	0.2213	0.038	5.848	0.000	0.147

```
0.295
gas                  -0.0069      0.007     -0.923      0.356      -0.021
0.008
hybrid                0.0331      0.018      1.874      0.061      -0.002
0.068
compact              -0.2976      0.006    -46.766      0.000      -0.310
-0.285
Q("full-size")       -0.1723      0.004    -40.168      0.000      -0.181
-0.164
Q("mid-size")        -0.2219      0.005    -43.827      0.000      -0.232
-0.212
Q("sub-compact")     -0.3627      0.015    -24.706      0.000      -0.391
-0.334
SUV                  -0.0695      0.005    -13.236      0.000      -0.080
-0.059
bus                   0.1151      0.038      3.053      0.002       0.041
0.189
convertible           0.3148      0.011     29.763      0.000       0.294
0.335
coupe                 0.2977      0.008     37.235      0.000       0.282
0.313
hatchback            -0.1043      0.010    -10.899      0.000      -0.123
-0.086
Q("mini-van")        -0.0387      0.012     -3.218      0.001      -0.062
-0.015
offroad               0.2301      0.031      7.442      0.000       0.170
0.291
pickup                0.3944      0.006     63.144      0.000       0.382
0.407
sedan                -0.1450      0.005    -26.682      0.000      -0.156
-0.134
truck                 0.1904      0.006     29.390      0.000       0.178
0.203
van                   0.0499      0.011      4.671      0.000       0.029
0.071
wagon                -0.1378      0.010    -13.788      0.000      -0.157
-0.118
black                 0.1941      0.005     38.543      0.000       0.184
0.204
blue                  0.0877      0.006     14.243      0.000       0.076
0.100
brown                 0.1408      0.011     12.331      0.000       0.118
0.163
custom                0.3078      0.010     29.748      0.000       0.287
0.328
green                 0.0273      0.010      2.694      0.007       0.007
0.047
```

```
grey                    0.1245      0.006     19.407      0.000       0.112
0.137
orange                  0.3163      0.020     15.509      0.000       0.276
0.356
purple                  0.1920      0.031      6.128      0.000       0.131
0.253
red                     0.1498      0.006     24.178      0.000       0.138
0.162
silver                  0.0695      0.006     12.611      0.000       0.059
0.080
white                   0.1679      0.005     34.386      0.000       0.158
0.177
yellow                  0.2679      0.018     14.958      0.000       0.233
0.303
==============================================================================
Omnibus:                    33144.849   Durbin-Watson:                   1.587
Prob(Omnibus):                  0.000   Jarque-Bera (JB):            69119.441
Skew:                           0.637   Prob(JB):                         0.00
Kurtosis:                       4.820   Cond. No.                     1.21e+03
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
[2] The condition number is large, 1.21e+03. This might indicate that there are
strong multicollinearity or other numerical problems.
"""
```

[34]:
```
pred = lml.predict(X_test)
mse(pred, y_test)
```

[34]: 0.713143388089478

As we can see, the 3-layer neural network performs notably better on the test set.

1.5 The dark side of neural networks (15 mts)

Around this time, you may have a natural thought: "Does this mean that if we have a single supercomputer and a neural network algorithm, we can solve *every problem in the world???*" It's a pretty natural thought, because these neural networks truly seem infinitely flexible without having to put in careful feature engineering work. It would seem that they run contrary to all the scientific principles that we've been touting throughout this entire course.

As you might suspect, this isn't true. But why? Let's illustrate this with the following exercise:

1.5.1 Exercise 6: (10 mts)

Code and begin to fit a neural network with 10 layers.

Answer. One possible solution is given below:

[35]:
```
stupid_model = keras.Sequential([layers.Dense(10000, activation='relu',␣
 ↪input_shape=[len(X.columns),])])
stupid_model = keras.Sequential([layers.Dense(5000, activation='relu',␣
 ↪input_shape=[len(X.columns),])])
stupid_model = keras.Sequential([layers.Dense(2000, activation='relu',␣
 ↪input_shape=[len(X.columns),])])
for _ in np.arange(8):
    stupid_model.add(layers.Dense(100, activation='relu'))
stupid_model.add(layers.Dense(1))

stupid_model.compile(loss='mse', optimizer='adam')
```

[36]:
```
stupid_model.fit(X.iloc[:20000], y.iloc[:20000], epochs=100, validation_split=0.
 ↪2, verbose=1, batch_size=1000)
```

```
Train on 16000 samples, validate on 4000 samples
Epoch 1/100
16000/16000 [==============================] - 1s 74us/sample - loss: 0.7103 -
val_loss: 0.6652
Epoch 2/100
16000/16000 [==============================] - 1s 32us/sample - loss: 0.4472 -
val_loss: 0.5691
Epoch 3/100
16000/16000 [==============================] - 0s 31us/sample - loss: 0.4042 -
val_loss: 0.5608
Epoch 4/100
16000/16000 [==============================] - 0s 31us/sample - loss: 0.3875 -
val_loss: 0.5971
Epoch 5/100
16000/16000 [==============================] - 1s 32us/sample - loss: 0.3755 -
val_loss: 0.5642
Epoch 6/100
16000/16000 [==============================] - 1s 32us/sample - loss: 0.3510 -
val_loss: 0.5668
Epoch 7/100
16000/16000 [==============================] - 0s 31us/sample - loss: 0.3298 -
val_loss: 0.5585
Epoch 8/100
16000/16000 [==============================] - 1s 32us/sample - loss: 0.3134 -
val_loss: 0.5716
Epoch 9/100
16000/16000 [==============================] - 0s 31us/sample - loss: 0.3051 -
```

```
val_loss: 0.5888
Epoch 10/100
16000/16000 [==============================] - 1s 31us/sample - loss: 0.2939 -
val_loss: 0.6366
Epoch 11/100
16000/16000 [==============================] - 0s 30us/sample - loss: 0.3021 -
val_loss: 0.5875
Epoch 12/100
16000/16000 [==============================] - 0s 31us/sample - loss: 0.2635 -
val_loss: 0.6704
Epoch 13/100
16000/16000 [==============================] - 1s 31us/sample - loss: 0.2628 -
val_loss: 0.6245
Epoch 14/100
16000/16000 [==============================] - 0s 31us/sample - loss: 0.2498 -
val_loss: 0.6034
Epoch 15/100
16000/16000 [==============================] - 0s 31us/sample - loss: 0.2368 -
val_loss: 0.6626
Epoch 16/100
16000/16000 [==============================] - 0s 31us/sample - loss: 0.2300 -
val_loss: 0.6447
Epoch 17/100
16000/16000 [==============================] - 1s 32us/sample - loss: 0.2231 -
val_loss: 0.6634
Epoch 18/100
16000/16000 [==============================] - 1s 32us/sample - loss: 0.2050 -
val_loss: 0.6482
Epoch 19/100
16000/16000 [==============================] - 1s 35us/sample - loss: 0.1977 -
val_loss: 0.7768
Epoch 20/100
16000/16000 [==============================] - 1s 32us/sample - loss: 0.1934 -
val_loss: 0.6910
Epoch 21/100
16000/16000 [==============================] - 1s 32us/sample - loss: 0.1827 -
val_loss: 0.7417
Epoch 22/100
16000/16000 [==============================] - 1s 32us/sample - loss: 0.1827 -
val_loss: 0.6777
Epoch 23/100
16000/16000 [==============================] - 1s 32us/sample - loss: 0.1736 -
val_loss: 0.6878
Epoch 24/100
16000/16000 [==============================] - 0s 31us/sample - loss: 0.1640 -
val_loss: 0.7270
Epoch 25/100
16000/16000 [==============================] - 1s 31us/sample - loss: 0.1563 -
```

```
val_loss: 0.7061
Epoch 26/100
16000/16000 [==============================] - 1s 31us/sample - loss: 0.1676 -
val_loss: 0.6750
Epoch 27/100
16000/16000 [==============================] - 1s 32us/sample - loss: 0.1744 -
val_loss: 0.6715
Epoch 28/100
16000/16000 [==============================] - 1s 32us/sample - loss: 0.1791 -
val_loss: 0.7592
Epoch 29/100
16000/16000 [==============================] - 1s 32us/sample - loss: 0.1697 -
val_loss: 0.6753
Epoch 30/100
16000/16000 [==============================] - 1s 32us/sample - loss: 0.1454 -
val_loss: 0.6843
Epoch 31/100
16000/16000 [==============================] - 1s 31us/sample - loss: 0.1369 -
val_loss: 0.7372
Epoch 32/100
16000/16000 [==============================] - 1s 32us/sample - loss: 0.1395 -
val_loss: 0.7499
Epoch 33/100
16000/16000 [==============================] - 1s 32us/sample - loss: 0.1397 -
val_loss: 0.7362
Epoch 34/100
16000/16000 [==============================] - 0s 31us/sample - loss: 0.1356 -
val_loss: 0.6947
Epoch 35/100
16000/16000 [==============================] - 1s 33us/sample - loss: 0.1369 -
val_loss: 0.7014
Epoch 36/100
16000/16000 [==============================] - 1s 35us/sample - loss: 0.1343 -
val_loss: 0.6781
Epoch 37/100
16000/16000 [==============================] - 1s 45us/sample - loss: 0.1287 -
val_loss: 0.6847
Epoch 38/100
16000/16000 [==============================] - 1s 32us/sample - loss: 0.1315 -
val_loss: 0.7797
Epoch 39/100
16000/16000 [==============================] - 1s 32us/sample - loss: 0.1254 -
val_loss: 0.7913
Epoch 40/100
16000/16000 [==============================] - 1s 37us/sample - loss: 0.1196 -
val_loss: 0.7261
Epoch 41/100
16000/16000 [==============================] - 1s 35us/sample - loss: 0.1208 -
```

```
val_loss: 0.7169
Epoch 42/100
16000/16000 [==============================] - 0s 31us/sample - loss: 0.1147 -
val_loss: 0.7272
Epoch 43/100
16000/16000 [==============================] - 0s 30us/sample - loss: 0.1179 -
val_loss: 0.6692
Epoch 44/100
16000/16000 [==============================] - 1s 32us/sample - loss: 0.1180 -
val_loss: 0.7616
Epoch 45/100
16000/16000 [==============================] - 1s 47us/sample - loss: 0.1164 -
val_loss: 0.7064
Epoch 46/100
16000/16000 [==============================] - 1s 38us/sample - loss: 0.1213 -
val_loss: 0.8145
Epoch 47/100
16000/16000 [==============================] - 1s 33us/sample - loss: 0.1179 -
val_loss: 0.7335
Epoch 48/100
16000/16000 [==============================] - 1s 31us/sample - loss: 0.1111 -
val_loss: 0.7308
Epoch 49/100
16000/16000 [==============================] - 1s 33us/sample - loss: 0.1102 -
val_loss: 0.7225
Epoch 50/100
16000/16000 [==============================] - 1s 34us/sample - loss: 0.1129 -
val_loss: 0.7023
Epoch 51/100
16000/16000 [==============================] - 1s 32us/sample - loss: 0.1103 -
val_loss: 0.7702
Epoch 52/100
16000/16000 [==============================] - 0s 31us/sample - loss: 0.1075 -
val_loss: 0.7405
Epoch 53/100
16000/16000 [==============================] - 0s 30us/sample - loss: 0.1107 -
val_loss: 0.7059
Epoch 54/100
16000/16000 [==============================] - 0s 29us/sample - loss: 0.1066 -
val_loss: 0.8499
Epoch 55/100
16000/16000 [==============================] - 0s 30us/sample - loss: 0.1090 -
val_loss: 0.7267
Epoch 56/100
16000/16000 [==============================] - 1s 33us/sample - loss: 0.1063 -
val_loss: 0.7624
Epoch 57/100
16000/16000 [==============================] - 0s 30us/sample - loss: 0.1013 -
```

```
val_loss: 0.7234
Epoch 58/100
16000/16000 [==============================] - 1s 34us/sample - loss: 0.1049 -
val_loss: 0.6983
Epoch 59/100
16000/16000 [==============================] - 1s 35us/sample - loss: 0.1020 -
val_loss: 0.7287
Epoch 60/100
16000/16000 [==============================] - 1s 34us/sample - loss: 0.0986 -
val_loss: 0.7151
Epoch 61/100
16000/16000 [==============================] - 1s 44us/sample - loss: 0.0995 -
val_loss: 0.8248
Epoch 62/100
16000/16000 [==============================] - 0s 29us/sample - loss: 0.1095 -
val_loss: 0.7527
Epoch 63/100
16000/16000 [==============================] - 1s 47us/sample - loss: 0.1018 -
val_loss: 0.7271
Epoch 64/100
16000/16000 [==============================] - 0s 31us/sample - loss: 0.1001 -
val_loss: 0.6899
Epoch 65/100
16000/16000 [==============================] - 1s 33us/sample - loss: 0.0965 -
val_loss: 0.7445
Epoch 66/100
16000/16000 [==============================] - 1s 34us/sample - loss: 0.1006 -
val_loss: 0.7715
Epoch 67/100
16000/16000 [==============================] - 0s 31us/sample - loss: 0.1000 -
val_loss: 0.7388
Epoch 68/100
16000/16000 [==============================] - 1s 32us/sample - loss: 0.1048 -
val_loss: 0.7358
Epoch 69/100
16000/16000 [==============================] - 0s 29us/sample - loss: 0.0939 -
val_loss: 0.7264
Epoch 70/100
16000/16000 [==============================] - 0s 31us/sample - loss: 0.0928 -
val_loss: 0.7246
Epoch 71/100
16000/16000 [==============================] - 0s 30us/sample - loss: 0.0947 -
val_loss: 0.7153
Epoch 72/100
16000/16000 [==============================] - 0s 30us/sample - loss: 0.0936 -
val_loss: 0.7310
Epoch 73/100
16000/16000 [==============================] - 1s 31us/sample - loss: 0.0991 -
```

```
val_loss: 0.7067
Epoch 74/100
16000/16000 [==============================] - 0s 31us/sample - loss: 0.0950 -
val_loss: 0.7513
Epoch 75/100
16000/16000 [==============================] - 0s 31us/sample - loss: 0.0931 -
val_loss: 0.7010
Epoch 76/100
16000/16000 [==============================] - 0s 31us/sample - loss: 0.0889 -
val_loss: 0.7045
Epoch 77/100
16000/16000 [==============================] - 0s 31us/sample - loss: 0.0887 -
val_loss: 0.7151
Epoch 78/100
16000/16000 [==============================] - 1s 31us/sample - loss: 0.0880 -
val_loss: 0.7055
Epoch 79/100
16000/16000 [==============================] - 0s 30us/sample - loss: 0.0894 -
val_loss: 0.7098
Epoch 80/100
16000/16000 [==============================] - 0s 31us/sample - loss: 0.0908 -
val_loss: 0.7079
Epoch 81/100
16000/16000 [==============================] - 0s 30us/sample - loss: 0.0863 -
val_loss: 0.7495
Epoch 82/100
16000/16000 [==============================] - 0s 31us/sample - loss: 0.0853 -
val_loss: 0.7613
Epoch 83/100
16000/16000 [==============================] - 0s 31us/sample - loss: 0.0878 -
val_loss: 0.7402
Epoch 84/100
16000/16000 [==============================] - 0s 31us/sample - loss: 0.0851 -
val_loss: 0.7322
Epoch 85/100
16000/16000 [==============================] - 0s 30us/sample - loss: 0.0833 -
val_loss: 0.7196
Epoch 86/100
16000/16000 [==============================] - 0s 31us/sample - loss: 0.0890 -
val_loss: 0.7588
Epoch 87/100
16000/16000 [==============================] - 0s 30us/sample - loss: 0.0865 -
val_loss: 0.7457
Epoch 88/100
16000/16000 [==============================] - 0s 30us/sample - loss: 0.0864 -
val_loss: 0.7042
Epoch 89/100
16000/16000 [==============================] - 0s 31us/sample - loss: 0.0842 -
```

```
val_loss: 0.7341
Epoch 90/100
16000/16000 [==============================] - 0s 30us/sample - loss: 0.0852 -
val_loss: 0.7455
Epoch 91/100
16000/16000 [==============================] - 0s 30us/sample - loss: 0.0886 -
val_loss: 0.6959
Epoch 92/100
16000/16000 [==============================] - 0s 30us/sample - loss: 0.0888 -
val_loss: 0.7203
Epoch 93/100
16000/16000 [==============================] - 0s 31us/sample - loss: 0.0811 -
val_loss: 0.7327
Epoch 94/100
16000/16000 [==============================] - 0s 31us/sample - loss: 0.0791 -
val_loss: 0.7474
Epoch 95/100
16000/16000 [==============================] - 1s 31us/sample - loss: 0.0841 -
val_loss: 0.7457
Epoch 96/100
16000/16000 [==============================] - 0s 31us/sample - loss: 0.0872 -
val_loss: 0.7305
Epoch 97/100
16000/16000 [==============================] - 0s 31us/sample - loss: 0.0896 -
val_loss: 0.8145
Epoch 98/100
16000/16000 [==============================] - 0s 29us/sample - loss: 0.0850 -
val_loss: 0.7323
Epoch 99/100
16000/16000 [==============================] - 0s 31us/sample - loss: 0.0810 -
val_loss: 0.6949
Epoch 100/100
16000/16000 [==============================] - 0s 30us/sample - loss: 0.0787 -
val_loss: 0.7006
```

[36]:
```
<tensorflow.python.keras.callbacks.History at 0x132313748>
```

Let's check how the model does on data that it hasn't seen before:

[37]:
```
stupid_model.evaluate(X_test, y_test)
```

```
98608/98608 [==============================] - 3s 27us/sample - loss: 0.6600
```

[37]:
```
0.6599767346085617
```

What happened? The model trained well on the training data (albeit after a long time and with inefficient memory usage), but utterly failed on the testing data. Essentially, the neural network is suffering from overfitting.

Of course, we can get overfitting problems with simpler models like linear and logistic regression. So why is it so particularly bad when it occurs with neural networks?

1.5.2 Exercise 7: (5 mts)

Which of the following is a potential drawback of solving this problem with a neural network instead of linear regression? Select all that apply.

I. Lack of transparency in the algorithm

II. Difficulty of use

III. Speed of runtime

Answer. I and III. The lack of transparency in the algorithm usually necessitates working at a very high level, which can make it difficult to debug and identify ways to improve performance. II is incorrect – most of the difficulty in incorporating a neural network comes from properly tuning parameters. This can be tedious, but in general isn't as difficult as thinking about how to properly engineer good features. III is a potential difficulty of neural networks. Because neural networks can get complicated and opaque,can be slow, and often prohibitively so in many real-world applications.

Thus, a lot of the challenge in devising neural networks comes from the "black box" nature of implementation. Because the mathematics isn't so intuitive, it requires careful effort and patience to determine the best parameters and optimizers to use. Furthermore, if our model overfits and something goes wrong, the opaque nature of the algorithm makes it much more difficult to interpret, diagnose, and improve compared to a standard linear regression model.

1.6 Conclusion (5 mts)

In this case, we saw some of the shortcomings of a traditional linear model in performing predictions on a dataset with many complex non-linearities. Using a simple neural network, we're able to predict within 50% of the fair price of a used car. This isn't stellar accuracy, but is better than the 71% interval that we can predict with a similarly naive linear model. Nonetheless, we learned that neural networks have a dark side – when they are not as accurate as we would like them to be, they are very difficult to diagnose and improve. On the other hand, linear models remain quite interpretable even as we make them more complex.

1.7 Takeaways (5 mts)

We've learned a few important facts about neural networks:

1. Neural networks can be used to optimize quantitative transformations of data into meaningful information
2. In some situations, a neural network may be more suitable and more efficient than manual feature engineering
3. The biggest drawback of neural networks is their lack of interpretability and difficulty of improvement, while the biggest strength is their ease of use and flexibility