Predict Price using Sneaker Image

Sreevar Rao Patiyara
https://colab.research.google.com/drive/1SfP1nwZP3W8EOysRpXfYotu886lSCU1i
https://github.com/SreevarP/SneakerClassificationModel/tree/main

Abstract -

The Sneaker Classification Model presented in this project utilizes TensorFlow's Keras library to develop a machine learning model for the purpose of classifying sneakers based on their images and predicting their associated prices. The model is trained on a dataset containing diverse images of sneakers along with corresponding price labels. Convolutional Neural Networks (CNNs) are employed to capture complex features from the input images, enabling the model to recognize patterns relevant to sneaker classification. The training process involves optimizing the model's parameters to minimize the disparity between predicted and actual prices. The resulting model demonstrates the ability to accurately classify unseen sneaker images and estimate their prices. This project contributes to the field of computer vision and demonstrates the practical application of deep learning in the domain of fashion and retail like figuring out whether prices should go down (markdown) or up (markup).

Introduction -

The obsession with sneakers began almost four decades ago, triggered by the introduction of the first Air Jordan, thereby initiating a cultural phenomenon. In a price war, competitors engage in unhealthy competition by consistently lowering the prices of their products and services in an attempt to secure a larger market share, leading to a downward spiral in pricing. Consequently, competing businesses offering comparable products find themselves compelled to lower their prices, leading to a reduction in their profit margins. This model will help the retailers to get the lowest price of the sneaker by providing the image.

Data Acquisition -

Gather the necessary data from the provided URL while specifying 100 results per page. Retrieve information on sneaker details, prices, and extract corresponding images. Retailers can collect this data inhouse or leverage third party data provided who scrapes the web.

Sneaker Data - Demographics & Images

Data Preparation & Build Training Dataset -

- Parse the JSON data and extract the image of the shoe with CV2 package.
- Extract the price from the json data and convert to dollars
- Read the image and resize to 50 using CV2 library
- Create a new dataset with Image and Price

Build Model -
The overall structure of the model includes two sets of Convolutional, Activation, and MaxPooling layers. This is an architecture used in CNNs for image classification tasks. Followed by flattening and fully connected layers. The output layer is designed for binary classification using a sigmoid activation function. The model is then compiled and trained on the provided data

Sequential - This function creates a sequential model, which is a linear stack of layers. In a sequential model, you can simply add one layer at a time, starting from the input.
This adds a 2D convolutional layer to the model with 64 filters, each of size (3, 3).
Activation - This adds a rectified linear unit (ReLU) activation function after the convolutional layer. ReLU is commonly used to introduce non-linearity in neural networks.
MaxPooling2D - This adds a max-pooling layer with a pool size of (2, 2). Max-pooling is employed to reduce the spatial dimensions of the input through down-sampling
Activation('sigmoid') - The sigmoid activation function is utilized on the output neuron, a common practice in binary classification tasks. This function compresses the output values within the range of 0 to 1, serving to represent probabilities.

Model Testing, Predict & Validation -

- Acquire a new dataset from the url above and prepare the results in the similar format to how we trained.
- Parse the JSON data and extract the image of the shoe with CV2 package.
- Extract the price from the json data and convert to dollars
- Read the image and resize to 50 using CV2 library
- Create a validation dataset with Image and Price
- Use the model built in the above steps and predict with a new validation dataset.
- Find the average error with Training and predicted values.

Conclusion -
Adding more layers to a convolutional neural network (CNN) can potentially enhance its ability to learn intricate patterns and features, which may contribute to improved accuracy. However, it's essential to strike a balance and consider factors such as model complexity, computational resources, and the risk of overfitting. In addition to trimming the initial training dataset

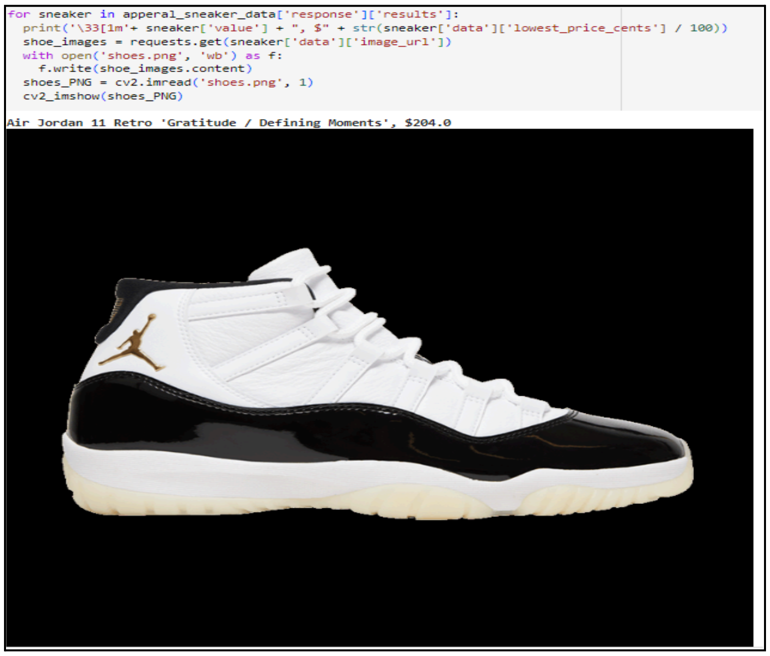
Depth of the Network - Increasing the depth of the network by adding more convolutional, activation, and pooling layers can allow the model to capture more complex hierarchical features in the data.
Hardware Requirements - Deeper networks require more computational resources for training and inference.

Code Snippet for adding more layers & filters -

```
model.add(Conv2D(128, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(128, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Flatten())
model.add(Dense(128))
model.add(Activation('relu'))

model.add(Dense(64))
model.add(Activation('relu'))

model.add(Dense(1))
model.add(Activation('sigmoid'))
```

Code Run Screenshots -

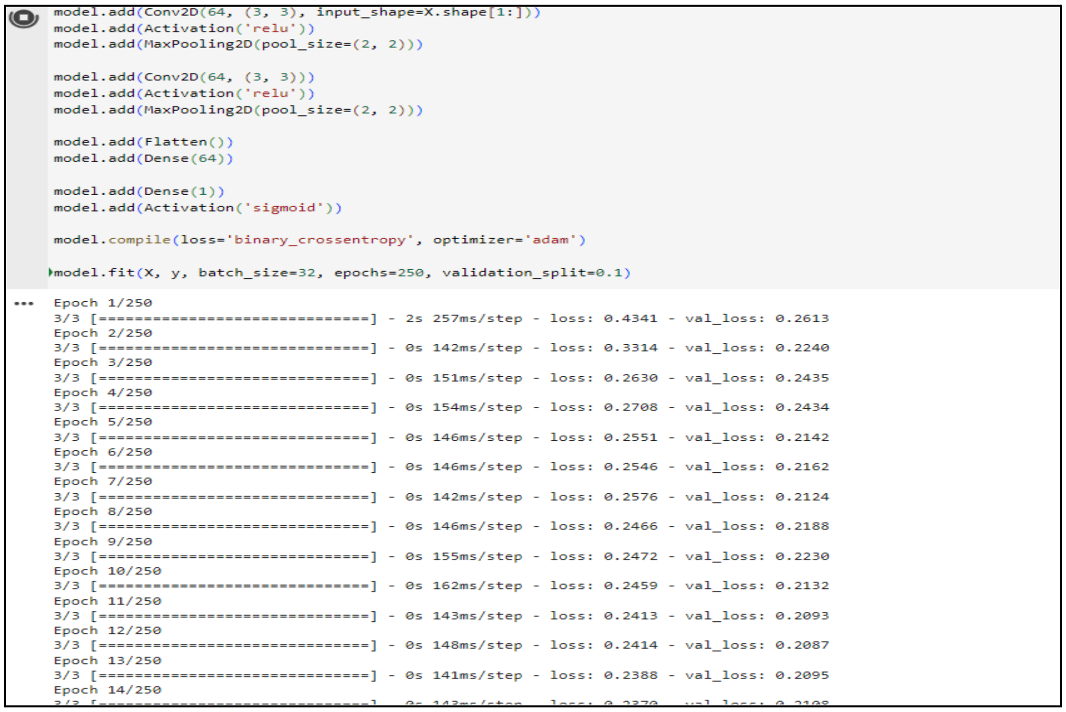
Data Prep and Image show



```
for sneaker in apperal_sneaker_data['response']['results']:
  print('\33[1m'+ sneaker['value'] + ", $" + str(sneaker['data']['lowest_price_cents'] / 100))
  shoe_images = requests.get(sneaker['data']['image_url'])
  with open('shoes.png', 'wb') as f:
    f.write(shoe_images.content)
  shoes_PNG = cv2.imread('shoes.png', 1)
  cv2_imshow(shoes_PNG)
```

```
Air Jordan 11 Retro 'Gratitude / Defining Moments', $204.0
```

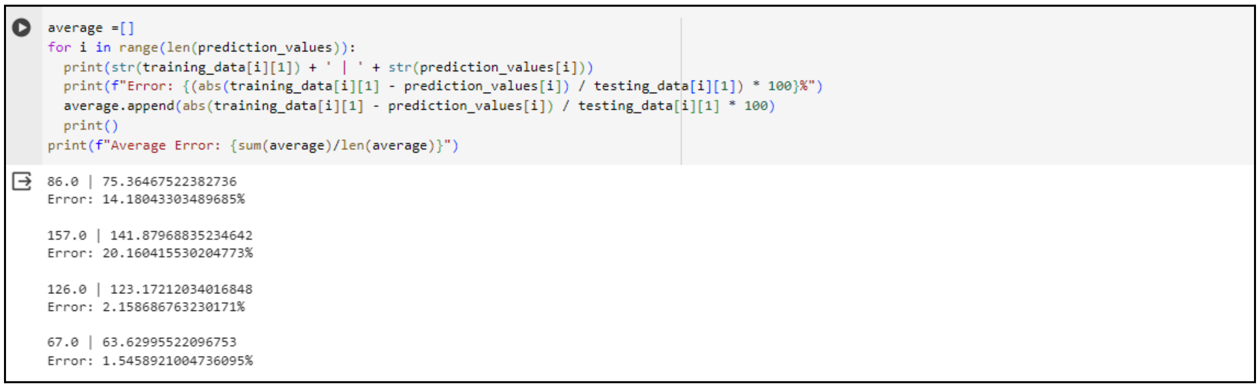
Model Training with 64 filters -



```
model.add(Conv2D(64, (3, 3), input_shape=X.shape[1:]))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(64, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Flatten())
model.add(Dense(64))

model.add(Dense(1))
model.add(Activation('sigmoid'))

model.compile(loss='binary_crossentropy', optimizer='adam')

model.fit(X, y, batch_size=32, epochs=250, validation_split=0.1)
```

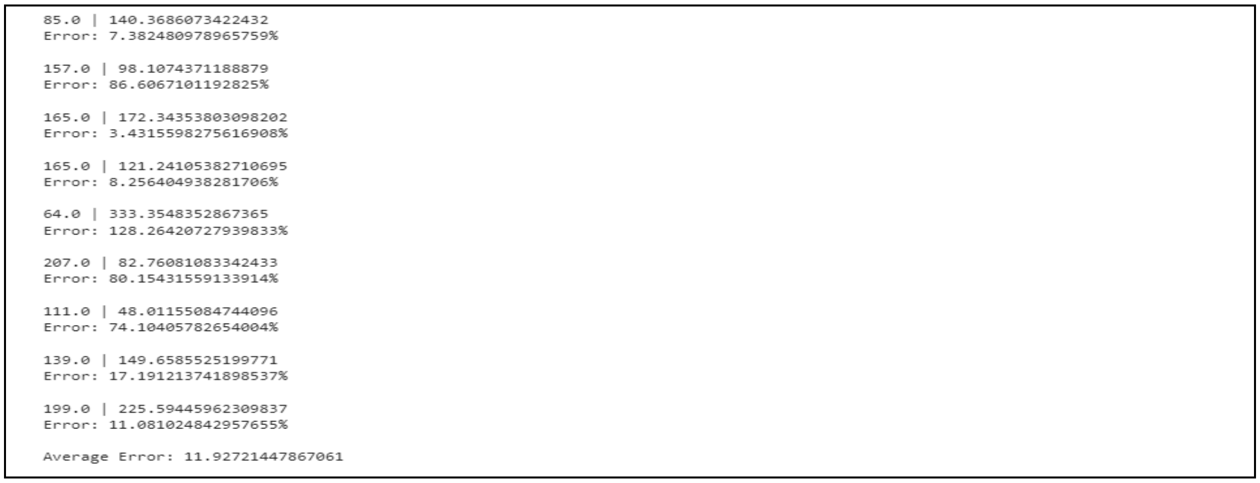
```
Epoch 1/250
3/3 [==============================] - 2s 257ms/step - loss: 0.4341 - val_loss: 0.2613
Epoch 2/250
3/3 [==============================] - 0s 142ms/step - loss: 0.3314 - val_loss: 0.2240
Epoch 3/250
3/3 [==============================] - 0s 151ms/step - loss: 0.2630 - val_loss: 0.2435
Epoch 4/250
3/3 [==============================] - 0s 154ms/step - loss: 0.2708 - val_loss: 0.2434
Epoch 5/250
3/3 [==============================] - 0s 146ms/step - loss: 0.2551 - val_loss: 0.2142
Epoch 6/250
3/3 [==============================] - 0s 146ms/step - loss: 0.2546 - val_loss: 0.2162
Epoch 7/250
3/3 [==============================] - 0s 142ms/step - loss: 0.2576 - val_loss: 0.2124
Epoch 8/250
3/3 [==============================] - 0s 146ms/step - loss: 0.2466 - val_loss: 0.2188
Epoch 9/250
3/3 [==============================] - 0s 155ms/step - loss: 0.2472 - val_loss: 0.2230
Epoch 10/250
3/3 [==============================] - 0s 162ms/step - loss: 0.2459 - val_loss: 0.2132
Epoch 11/250
3/3 [==============================] - 0s 143ms/step - loss: 0.2413 - val_loss: 0.2093
Epoch 12/250
3/3 [==============================] - 0s 148ms/step - loss: 0.2414 - val_loss: 0.2087
Epoch 13/250
3/3 [==============================] - 0s 141ms/step - loss: 0.2388 - val_loss: 0.2095
Epoch 14/250
```

Model Prediction with 64 filters -



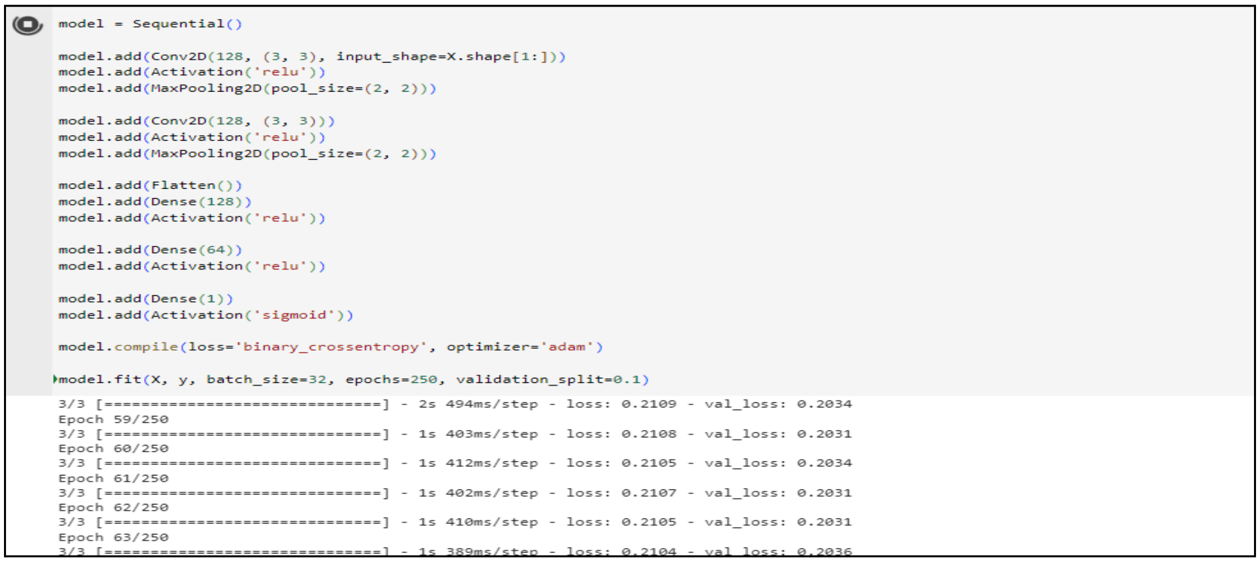
```
average =[]
for i in range(len(prediction_values)):
  print(str(training_data[i][1]) + ' | ' + str(prediction_values[i]))
  print(f"Error: {(abs(training_data[i][1] - prediction_values[i]) / testing_data[i][1]) * 100}%")
  average.append(abs(training_data[i][1] - prediction_values[i]) / testing_data[i][1] * 100)
  print()
print(f"Average Error: {sum(average)/len(average)}")
```

```
86.0 | 75.36467522382736
Error: 14.18043303489685%

157.0 | 141.87968835234642
Error: 20.160415530204773%

126.0 | 123.17212034016848
Error: 2.158686763230171%

67.0 | 63.62995522096753
Error: 1.5458921004736095%
```



```
85.0 | 140.3686073422432
Error: 7.382480978965759%

157.0 | 98.1074371188879
Error: 86.6067101192825%

165.0 | 172.34353803098202
Error: 3.4315598275616908%

165.0 | 121.24105382710695
Error: 8.256404938281706%

64.0 | 333.3548352867365
Error: 128.26420727939833%

207.0 | 82.76081083342433
Error: 80.15431559133914%

111.0 | 48.01155084744096
Error: 74.10405782654004%

139.0 | 149.6585525199771
Error: 17.191213741898537%

199.0 | 225.59445962309837
Error: 11.081024842957655%

Average Error: 11.92721447867061
```

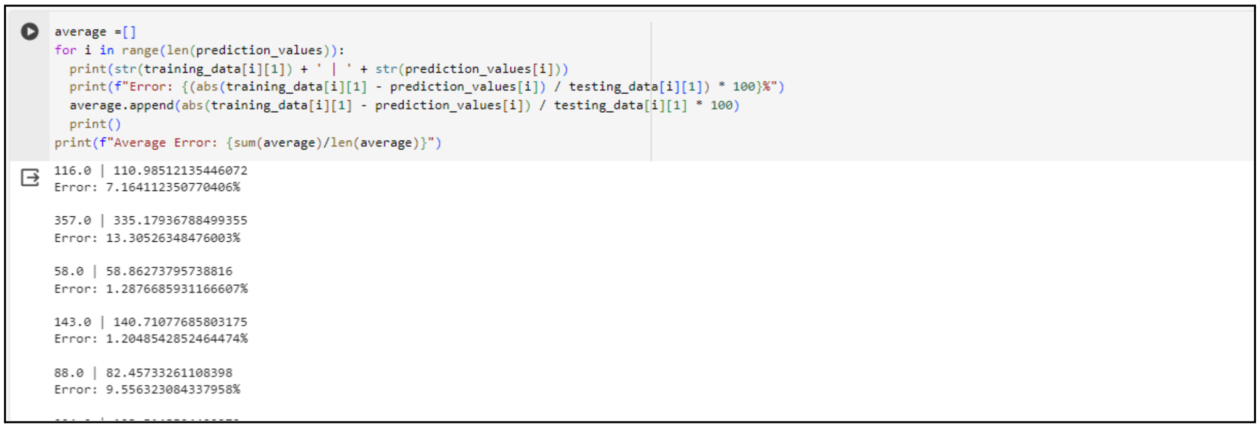
Model Training with 128 filters -



```
model = Sequential()

model.add(Conv2D(128, (3, 3), input_shape=X.shape[1:]))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(128, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Flatten())
model.add(Dense(128))
model.add(Activation('relu'))

model.add(Dense(64))
model.add(Activation('relu'))

model.add(Dense(1))
model.add(Activation('sigmoid'))

model.compile(loss='binary_crossentropy', optimizer='adam')

model.fit(X, y, batch_size=32, epochs=250, validation_split=0.1)
```

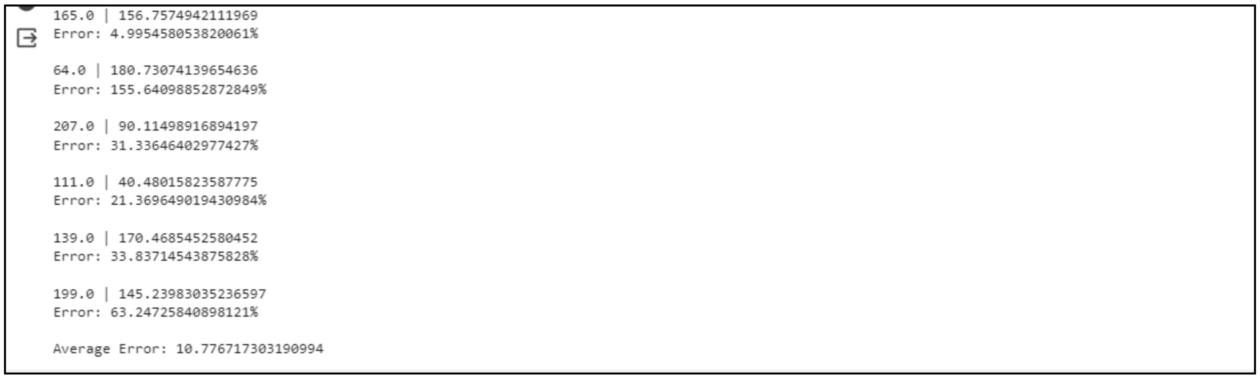
```
3/3 [==============================] - 2s 494ms/step - loss: 0.2109 - val_loss: 0.2034
Epoch 59/250
3/3 [==============================] - 1s 403ms/step - loss: 0.2108 - val_loss: 0.2031
Epoch 60/250
3/3 [==============================] - 1s 412ms/step - loss: 0.2105 - val_loss: 0.2034
Epoch 61/250
3/3 [==============================] - 1s 402ms/step - loss: 0.2107 - val_loss: 0.2031
Epoch 62/250
3/3 [==============================] - 1s 410ms/step - loss: 0.2105 - val_loss: 0.2031
Epoch 63/250
3/3 [==============================] - 1s 389ms/step - loss: 0.2104 - val_loss: 0.2036
```

Model Prediction with 128 filters -



```
average =[]
for i in range(len(prediction_values)):
  print(str(training_data[i][1]) + ' | ' + str(prediction_values[i]))
  print(f"Error: {(abs(training_data[i][1] - prediction_values[i]) / testing_data[i][1]) * 100}%")
  average.append(abs(training_data[i][1] - prediction_values[i]) / testing_data[i][1] * 100)
  print()
print(f"Average Error: {sum(average)/len(average)}")
```

```
116.0 | 110.98512135446072
Error: 7.164112350770406%

357.0 | 335.17936788499355
Error: 13.30526348476003%

58.0 | 58.86273795738816
Error: 1.2876685931166607%

143.0 | 140.71077685803175
Error: 1.2048542852464474%

88.0 | 82.45733261108398
Error: 9.556323084337958%
```



```
165.0 | 156.7574942111969
Error: 4.995458053820061%

64.0 | 180.73074139654636
Error: 155.64098852872849%

207.0 | 90.11498916894197
Error: 31.33646402977427%

111.0 | 40.48015823587775
Error: 21.369649019430984%

139.0 | 170.4685452580452
Error: 33.83714543875828%

199.0 | 145.23983035236597
Error: 63.24725840898121%

Average Error: 10.776717303190994
```