Open in Colab

T81-558: Applications of Deep Neural Networks

Module 6: Convolutional Neural Networks (CNN) for Computer Vision

- Instructor: Jeff Heaton, McKelvey School of Engineering, Washington University in St. Louis
- For more information visit the class website.

Module 6 Material

- Part 6.1: Image Processing in Python [Video] [Notebook]
- Part 6.2: Using Convolutional Neural Networks [Video] [Notebook]
- **Part 6.3: Using Pretrained Neural Networks with Keras** [Video] [Notebook]
- Part 6.4: Looking at Keras Generators and Image Augmentation [Video] [Notebook]
- Part 6.5: Recognizing Multiple Images with YOLOv5 [Video] [Notebook]

Google CoLab Instructions

The following code ensures that Google CoLab is running the correct version of TensorFlow.

In [1]:

```
# Detect Colab if present
try:
    from google.colab import drive
    COLAB = True
    print("Note: using Google CoLab")
    %tensorflow_version 2.x
except:
    print("Note: not using Google CoLab")
    COLAB = False

# Nicely formatted time string
def hms_string(sec_elapsed):
    h = int(sec_elapsed / (60 * 60))
    m = int((sec_elapsed % (60 * 60)) / 60)
    s = sec_elapsed % 60
    return f"{h}:{m:>02}:{s:>05.2f}"
```

```
Note: using Google CoLab
```

Part 6.3: Transfer Learning for

Computer Vision

Many advanced prebuilt neural networks are available for computer vision, and Keras provides direct access to many networks. Transfer learning is the technique where you use these prebuilt neural networks. Module 9 takes a deeper look at transfer learning.

There are several different levels of transfer learning.

- Use a prebuilt neural network in its entirety
- Use a prebuilt neural network's structure
- Use a prebuilt neural network's weights

We will begin by using the MobileNet prebuilt neural network in its entirety. MobileNet will be loaded and allowed to classify simple images. We can already classify 1,000 images through this technique without ever having trained the network.

In [2]:

```
import pandas as pd
import numpy as np
import os
import tensorflow.keras
import matplotlib.pyplot as plt
from tensorflow.keras.layers import Dense,GlobalAveragePooling2D
from tensorflow.keras.applications import MobileNet
from tensorflow.keras.preprocessing import image
from tensorflow.keras.applications.mobilenet import preprocess_input
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
```

We begin by downloading weights for a MobileNet trained for the imagenet dataset, which will take some time to download the first time you train the network.

In [3]:

```
# HIDE OUTPUT
model = MobileNet(weights='imagenet',include_top=True)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-appl
ications/mobilenet/mobilenet_1_0_224_tf.h5
17227776/17225924 [==============================] - 0s 0us/step
17235968/17225924 [==============================] - 0s 0us/step
```

The loaded network is a Keras neural network. However, this is a neural network that a third party engineered on advanced hardware. Merely looking at the structure of an advanced state-of-the-art neural network can be educational.

In [4]:

```
model.summary()
```

```
Model: "mobilenet_1.00_224"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 224, 224, 3)]     0

 conv1 (Conv2D)              (None, 112, 112, 32)      864

 conv1_bn (BatchNormalizatio  (None, 112, 112, 32)     128
 n)

 conv1_relu (ReLU)           (None, 112, 112, 32)      0

 conv_dw_1 (DepthwiseConv2D)  (None, 112, 112, 32)     288

 conv_dw_1_bn (BatchNormaliz  (None, 112, 112, 32)     128
 ation)

 conv_dw_1_relu (ReLU)       (None, 112, 112, 32)      0

 conv_pw_1 (Conv2D)          (None, 112, 112, 64)      2048

 conv_pw_1_bn (BatchNormaliz  (None, 112, 112, 64)     256
 ation)

 conv_pw_1_relu (ReLU)       (None, 112, 112, 64)      0

 conv_pad_2 (ZeroPadding2D)  (None, 113, 113, 64)      0

 conv_dw_2 (DepthwiseConv2D)  (None, 56, 56, 64)       576

 conv_dw_2_bn (BatchNormaliz  (None, 56, 56, 64)       256
 ation)

 conv_dw_2_relu (ReLU)       (None, 56, 56, 64)        0

 conv_pw_2 (Conv2D)          (None, 56, 56, 128)       8192

 conv_pw_2_bn (BatchNormaliz  (None, 56, 56, 128)      512
 ation)

 conv_pw_2_relu (ReLU)       (None, 56, 56, 128)       0

 conv_dw_3 (DepthwiseConv2D)  (None, 56, 56, 128)      1152

 conv_dw_3_bn (BatchNormaliz  (None, 56, 56, 128)      512
 ation)

 conv_dw_3_relu (ReLU)       (None, 56, 56, 128)       0

 conv_pw_3 (Conv2D)          (None, 56, 56, 128)       16384

 conv_pw_3_bn (BatchNormaliz  (None, 56, 56, 128)      512
 ation)

 conv_pw_3_relu (ReLU)       (None, 56, 56, 128)       0

 conv_pad_4 (ZeroPadding2D)  (None, 57, 57, 128)       0

 conv_dw_4 (DepthwiseConv2D)  (None, 28, 28, 128)      1152

 conv_dw_4_bn (BatchNormaliz  (None, 28, 28, 128)      512
 ation)
```

```
 conv_dw_4_relu (ReLU)       (None, 28, 28, 128)       0

 conv_pw_4 (Conv2D)          (None, 28, 28, 256)       32768

 conv_pw_4_bn (BatchNormaliz  (None, 28, 28, 256)      1024
 ation)

 conv_pw_4_relu (ReLU)       (None, 28, 28, 256)       0

 conv_dw_5 (DepthwiseConv2D)  (None, 28, 28, 256)      2304

 conv_dw_5_bn (BatchNormaliz  (None, 28, 28, 256)      1024
 ation)

 conv_dw_5_relu (ReLU)       (None, 28, 28, 256)       0

 conv_pw_5 (Conv2D)          (None, 28, 28, 256)       65536

 conv_pw_5_bn (BatchNormaliz  (None, 28, 28, 256)      1024
 ation)

 conv_pw_5_relu (ReLU)       (None, 28, 28, 256)       0

 conv_pad_6 (ZeroPadding2D)  (None, 29, 29, 256)       0

 conv_dw_6 (DepthwiseConv2D)  (None, 14, 14, 256)      2304

 conv_dw_6_bn (BatchNormaliz  (None, 14, 14, 256)      1024
 ation)

 conv_dw_6_relu (ReLU)       (None, 14, 14, 256)       0

 conv_pw_6 (Conv2D)          (None, 14, 14, 512)       131072

 conv_pw_6_bn (BatchNormaliz  (None, 14, 14, 512)      2048
 ation)

 conv_pw_6_relu (ReLU)       (None, 14, 14, 512)       0

 conv_dw_7 (DepthwiseConv2D)  (None, 14, 14, 512)      4608

 conv_dw_7_bn (BatchNormaliz  (None, 14, 14, 512)      2048
 ation)

 conv_dw_7_relu (ReLU)       (None, 14, 14, 512)       0

 conv_pw_7 (Conv2D)          (None, 14, 14, 512)       262144

 conv_pw_7_bn (BatchNormaliz  (None, 14, 14, 512)      2048
 ation)

 conv_pw_7_relu (ReLU)       (None, 14, 14, 512)       0

 conv_dw_8 (DepthwiseConv2D)  (None, 14, 14, 512)      4608

 conv_dw_8_bn (BatchNormaliz  (None, 14, 14, 512)      2048
 ation)

 conv_dw_8_relu (ReLU)       (None, 14, 14, 512)       0

 conv_pw_8 (Conv2D)          (None, 14, 14, 512)       262144
```

```
conv_pw_8_bn (BatchNormaliz  (None, 14, 14, 512)      2048
ation)

conv_pw_8_relu (ReLU)       (None, 14, 14, 512)       0

conv_dw_9 (DepthwiseConv2D)  (None, 14, 14, 512)      4608

conv_dw_9_bn (BatchNormaliz  (None, 14, 14, 512)      2048
ation)

conv_dw_9_relu (ReLU)       (None, 14, 14, 512)       0

conv_pw_9 (Conv2D)          (None, 14, 14, 512)       262144

conv_pw_9_bn (BatchNormaliz  (None, 14, 14, 512)      2048
ation)

conv_pw_9_relu (ReLU)       (None, 14, 14, 512)       0

conv_dw_10 (DepthwiseConv2D  (None, 14, 14, 512)      4608
)

conv_dw_10_bn (BatchNormali  (None, 14, 14, 512)      2048
zation)

conv_dw_10_relu (ReLU)      (None, 14, 14, 512)       0

conv_pw_10 (Conv2D)         (None, 14, 14, 512)       262144

conv_pw_10_bn (BatchNormali  (None, 14, 14, 512)      2048
zation)

conv_pw_10_relu (ReLU)      (None, 14, 14, 512)       0

conv_dw_11 (DepthwiseConv2D  (None, 14, 14, 512)      4608
)

conv_dw_11_bn (BatchNormali  (None, 14, 14, 512)      2048
zation)

conv_dw_11_relu (ReLU)      (None, 14, 14, 512)       0

conv_pw_11 (Conv2D)         (None, 14, 14, 512)       262144

conv_pw_11_bn (BatchNormali  (None, 14, 14, 512)      2048
zation)

conv_pw_11_relu (ReLU)      (None, 14, 14, 512)       0

conv_pad_12 (ZeroPadding2D)  (None, 15, 15, 512)      0

conv_dw_12 (DepthwiseConv2D  (None, 7, 7, 512)        4608
)

conv_dw_12_bn (BatchNormali  (None, 7, 7, 512)        2048
zation)

conv_dw_12_relu (ReLU)      (None, 7, 7, 512)         0

conv_pw_12 (Conv2D)         (None, 7, 7, 1024)        524288

conv_pw_12_bn (BatchNormali  (None, 7, 7, 1024)       4096
zation)
```

```
 conv_pw_12_relu (ReLU)      (None, 7, 7, 1024)        0

 conv_dw_13 (DepthwiseConv2D  (None, 7, 7, 1024)       9216
 )

 conv_dw_13_bn (BatchNormali  (None, 7, 7, 1024)       4096
 zation)

 conv_dw_13_relu (ReLU)      (None, 7, 7, 1024)        0

 conv_pw_13 (Conv2D)         (None, 7, 7, 1024)        1048576

 conv_pw_13_bn (BatchNormali  (None, 7, 7, 1024)       4096
 zation)

 conv_pw_13_relu (ReLU)      (None, 7, 7, 1024)        0

 global_average_pooling2d (G  (None, 1, 1, 1024)       0
 lobalAveragePooling2D)

 dropout (Dropout)           (None, 1, 1, 1024)        0

 conv_preds (Conv2D)         (None, 1, 1, 1000)        1025000

 reshape_2 (Reshape)         (None, 1000)              0

 predictions (Activation)    (None, 1000)              0

=================================================================
Total params: 4,253,864
Trainable params: 4,231,976
Non-trainable params: 21,888
_________________________________________________________________
```

Several clues to neural network architecture become evident when examining the above structure.

We will now use the MobileNet to classify several image URLs below. You can add additional URLs of your own to see how well the MobileNet can classify.

In [5]:

```python
%matplotlib inline
from PIL import Image, ImageFile
from matplotlib.pyplot import imshow
import requests
import numpy as np
from io import BytesIO
from IPython.display import display, HTML
from tensorflow.keras.applications.mobilenet import decode_predictions

IMAGE_WIDTH = 224
IMAGE_HEIGHT = 224
IMAGE_CHANNELS = 3

ROOT = "https://data.heatonresearch.com/data/t81-558/images/"

def make_square(img):
    cols,rows = img.size

    if rows>cols:
        pad = (rows-cols)/2
```

```
        img = img.crop((pad,0,cols,cols))
    else:
        pad = (cols-rows)/2
        img = img.crop((0,pad,rows,rows))

    return img


def classify_image(url):
  x = []
  ImageFile.LOAD_TRUNCATED_IMAGES = False
  response = requests.get(url)
  img = Image.open(BytesIO(response.content))
  img.load()
  img = img.resize((IMAGE_WIDTH,IMAGE_HEIGHT),Image.ANTIALIAS)

  x = image.img_to_array(img)
  x = np.expand_dims(x, axis=0)
  x = preprocess_input(x)
  x = x[:,:,:,:3] # maybe an alpha channel
  pred = model.predict(x)

  display(img)
  print(np.argmax(pred,axis=1))

  lst = decode_predictions(pred, top=5)
  for itm in lst[0]:
      print(itm)
```

We can now classify an example image. You can specify the URL of any image you wish to classify.

In [6]:
```
classify_image(ROOT+"soccer_ball.jpg")
```

```
[805]
Downloading data from https://storage.googleapis.com/download.tensorflow.o
rg/data/imagenet_class_index.json
40960/35363 [==================================] - 0s 0us/step
49152/35363 [==========================================] - 0s 0us/step
('n04254680', 'soccer_ball', 0.9999938)
('n03530642', 'honeycomb', 3.862412e-06)
('n03255030', 'dumbbell', 4.442458e-07)
('n02782093', 'balloon', 3.7038987e-07)
('n04548280', 'wall_clock', 3.143911e-07)
```

In [7]:
```
classify_image(ROOT+"race_truck.jpg")
```

Overall, the neural network is doing quite well.

For many applications, MobileNet might be entirely acceptable as an image classifier. However, if you need to classify very specialized images, not in the 1,000 image types supported by imagenet, it is necessary to use transfer learning.

Using the Structure of ResNet

We will train a neural network to count the number of paper clips in images. We will make use of the structure of the ResNet neural network. There are several significant changes that we will make to ResNet to apply to this task. First, ResNet is a classifier; we wish to perform a regression to count. Secondly, we want to change the image resolution that ResNet uses. We will not use the weights from ResNet; changing this resolution invalidates the current weights. Thus, it will be necessary to retrain the network.

In [8]:
```
import os
URL = "https://github.com/jeffheaton/data-mirror/"
DOWNLOAD_SOURCE = URL+"releases/download/v1/paperclips.zip"
DOWNLOAD_NAME = DOWNLOAD_SOURCE[DOWNLOAD_SOURCE.rfind('/')+1:]

if COLAB:
  PATH = "/content"
else:
  # I used this locally on my machine, you may need different
  PATH = "/Users/jeff/temp"

EXTRACT_TARGET = os.path.join(PATH,"clips")
SOURCE = os.path.join(EXTRACT_TARGET, "paperclips")
```

```
[751]
('n04037443', 'racer', 0.7131951)
('n03100240', 'convertible', 0.100896776)
('n04285008', 'sports_car', 0.0770768)
('n03930630', 'pickup', 0.02635305)
('n02704792', 'amphibian', 0.011636169)
```

Next, we download the images. This part depends on the origin of your images. The following code downloads images from a URL, where a ZIP file contains the images. The code unzips the ZIP file.

In [9]:

```
# HIDE OUTPUT
!wget -O {os.path.join(PATH,DOWNLOAD_NAME)} {DOWNLOAD_SOURCE}
!mkdir -p {SOURCE}
!mkdir -p {TARGET}
!mkdir -p {EXTRACT_TARGET}
!unzip -o -j -d {SOURCE} {os.path.join(PATH, DOWNLOAD_NAME)} >/dev/null
```

```
--2021-11-28 08:45:31--  https://github.com/jeffheaton/data-mirror/release
s/download/v1/paperclips.zip
Resolving github.com (github.com)... 140.82.114.4
Connecting to github.com (github.com)|140.82.114.4|:443... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://objects.githubusercontent.com/github-production-release-
asset-2e65be/408419764/25830812-b9e6-4ddf-93b6-7932d9ef5982?X-Amz-Algorith
m=AWS4-HMAC-SHA256&X-Amz-Credential=AKIAIWNJYAX4CSVEH53A%2F20211128%2Fus-e
ast-1%2Fs3%2Faws4_request&X-Amz-Date=20211128T084531Z&X-Amz-Expires=300&X-
Amz-Signature=6db9f72de68b167bd44bfec7661073997b48ed04708a827f50189f622b03
95cd&X-Amz-SignedHeaders=host&actor_id=0&key_id=0&repo_id=408419764&respon
se-content-disposition=attachment%3B%20filename%3Dpaperclips.zip&response-
content-type=application%2Foctet-stream [following]
--2021-11-28 08:45:31--  https://objects.githubusercontent.com/github-prod
uction-release-asset-2e65be/408419764/25830812-b9e6-4ddf-93b6-7932d9ef598
2?X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Credential=AKIAIWNJYAX4CSVEH53A%2
F20211128%2Fus-east-1%2Fs3%2Faws4_request&X-Amz-Date=20211128T084531Z&X-Am
z-Expires=300&X-Amz-Signature=6db9f72de68b167bd44bfec7661073997b48ed04708a
827f50189f622b0395cd&X-Amz-SignedHeaders=host&actor_id=0&key_id=0&repo_id=
408419764&response-content-disposition=attachment%3B%20filename%3Dpapercli
ps.zip&response-content-type=application%2Foctet-stream
Resolving objects.githubusercontent.com (objects.githubusercontent.com)...
185.199.108.133, 185.199.109.133, 185.199.110.133, ...
Connecting to objects.githubusercontent.com (objects.githubusercontent.co
m)|185.199.108.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 163590691 (156M) [application/octet-stream]
Saving to: ‘/content/paperclips.zip’

/content/paperclips 100%[===================>] 156.01M  25.5MB/s    in 5.9
s

2021-11-28 08:45:37 (26.6 MB/s) - ‘/content/paperclips.zip’ saved [1635906
91/163590691]
```

The labels are contained in a CSV file named **train.csv** for the regression. This file has just two labels, **id** and **clip_count**. The ID specifies the filename; for example, row id 1 corresponds to the file **clips-1.jpg**. The following code loads the labels for the training set and creates a new column, named **filename**, that contains the filename of each image, based on the **id** column.

In [11]:

```
df_train = pd.read_csv(os.path.join(SOURCE, "train.csv"))
df_train['filename'] = "clips-" + df_train.id.astype(str) + ".jpg"
```

We want to use early stopping. To do this, we need a validation set. We will break the data into 80 percent test data and 20 validation. Do not confuse this validation data with the test set provided by Kaggle. This validation set is unique to your program and is for early stopping.

In [12]:

```
TRAIN_PCT = 0.9
TRAIN_CUT = int(len(df_train) * TRAIN_PCT)

df_train_cut = df_train[0:TRAIN_CUT]
df_validate_cut = df_train[TRAIN_CUT:]

print(f"Training size: {len(df_train_cut)}")
print(f"Validate size: {len(df_validate_cut)}")
```

```
Training size: 18000
Validate size: 2000
```

Next, we create the generators that will provide the images to the neural network during training. We normalize the images so that the RGB colors between 0-255 become ratios between 0 and 1. We also use the **flow_from_dataframe** generator to connect the Pandas dataframe to the actual image files. We see here a straightforward implementation; you might also wish to use some of the image transformations provided by the data generator.

The **HEIGHT** and **WIDTH** constants specify the dimensions to which the image will be scaled (or expanded). It is probably not a good idea to expand the images.

In [13]:

```
import tensorflow as tf
import keras_preprocessing
from keras_preprocessing import image
from keras_preprocessing.image import ImageDataGenerator

WIDTH = 256
HEIGHT = 256

training_datagen = ImageDataGenerator(
  rescale = 1./255,
  horizontal_flip=True,
  #vertical_flip=True,
  fill_mode='nearest')

train_generator = training_datagen.flow_from_dataframe(
        dataframe=df_train_cut,
        directory=SOURCE,
        x_col="filename",
        y_col="clip_count",
        target_size=(HEIGHT, WIDTH),
        # Keeping the training batch size small
        # USUALLY increases performance
        batch_size=32,
        class_mode='raw')

validation_datagen = ImageDataGenerator(rescale = 1./255)

val_generator = validation_datagen.flow_from_dataframe(
        dataframe=df_validate_cut,
        directory=SOURCE,
        x_col="filename",
        y_col="clip_count",
```

```
        target_size=(HEIGHT, WIDTH),
        # Make the validation batch size as large as you
        # have memory for
        batch_size=256,
        class_mode='raw')
```

```
Found 18000 validated image filenames.
Found 2000 validated image filenames.
```

We will now use a ResNet neural network as a basis for our neural network. We will redefine both the input shape and output of the ResNet model, so we will not transfer the weights. Since we redefine the input, the weights are of minimal value. We begin by loading, from Keras, the ResNet50 network. We specify **include_top** as False because we will change the input resolution. We also specify **weights** as false because we must retrain the network after changing the top input layers.

In [14]:

```
from tensorflow.keras.applications.resnet50 import ResNet50
from tensorflow.keras.layers import Input

input_tensor = Input(shape=(HEIGHT, WIDTH, 3))

base_model = ResNet50(
    include_top=False, weights=None, input_tensor=input_tensor,
    input_shape=None)
```

Now we must add a few layers to the end of the neural network so that it becomes a regression model.

In [15]:

```
from tensorflow.keras.layers import Dense, GlobalAveragePooling2D
from tensorflow.keras.models import Model

x=base_model.output
x=GlobalAveragePooling2D()(x)
x=Dense(1024,activation='relu')(x)
x=Dense(1024,activation='relu')(x)
model=Model(inputs=base_model.input,outputs=Dense(1)(x))
```

We train like before; the only difference is that we do not define the entire neural network here.

In [ ]:

```
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.metrics import RootMeanSquaredError

# Important, calculate a valid step size for the validation dataset
STEP_SIZE_VALID=val_generator.n//val_generator.batch_size

model.compile(loss = 'mean_squared_error', optimizer='adam',
              metrics=[RootMeanSquaredError(name="rmse")])
monitor = EarlyStopping(monitor='val_loss', min_delta=1e-3,
        patience=50, verbose=1, mode='auto',
        restore_best_weights=True)

history = model.fit(train_generator, epochs=100, steps_per_epoch=250,
                    validation_data = val_generator, callbacks=[monitor]
                    verbose = 1, validation_steps=STEP_SIZE_VALID)
```

```
Epoch 1/100
250/250 [==============================] - 70s 256ms/step - loss: 73.1411
- rmse: 8.5523 - val_loss: 701.4966 - val_rmse: 26.4858
Epoch 2/100
250/250 [==============================] - 61s 243ms/step - loss: 27.7530
- rmse: 5.2681 - val_loss: 365.0618 - val_rmse: 19.1066
Epoch 3/100
250/250 [==============================] - 61s 243ms/step - loss: 28.6821
- rmse: 5.3556 - val_loss: 130.9240 - val_rmse: 11.4422
Epoch 4/100
250/250 [==============================] - 61s 243ms/step - loss: 18.8626
- rmse: 4.3431 - val_loss: 55.8694 - val_rmse: 7.4746
Epoch 5/100
250/250 [==============================] - 61s 242ms/step - loss: 14.1957
- rmse: 3.7677 - val_loss: 554.3814 - val_rmse: 23.5453
Epoch 6/100
250/250 [==============================] - 61s 242ms/step - loss: 12.8428
- rmse: 3.5837 - val_loss: 79.6855 - val_rmse: 8.9267
Epoch 7/100
250/250 [==============================] - 61s 242ms/step - loss: 13.2751
- rmse: 3.6435 - val_loss: 316.9753 - val_rmse: 17.8038
Epoch 8/100
250/250 [==============================] - 61s 242ms/step - loss: 11.9826
- rmse: 3.4616 - val_loss: 466.4104 - val_rmse: 21.5965
Epoch 9/100
250/250 [==============================] - 61s 243ms/step - loss: 12.0956
- rmse: 3.4779 - val_loss: 4.5767 - val_rmse: 2.1393
Epoch 10/100
250/250 [==============================] - 60s 242ms/step - loss: 9.6629 -
rmse: 3.1085 - val_loss: 82.4498 - val_rmse: 9.0802
Epoch 11/100
250/250 [==============================] - 60s 242ms/step - loss: 6.0348 -
rmse: 2.4566 - val_loss: 134.9830 - val_rmse: 11.6182
Epoch 12/100
250/250 [==============================] - 61s 242ms/step - loss: 9.1004 -
rmse: 3.0167 - val_loss: 13.1667 - val_rmse: 3.6286
Epoch 13/100
250/250 [==============================] - 60s 242ms/step - loss: 9.2808 -
rmse: 3.0464 - val_loss: 372.9783 - val_rmse: 19.3126
Epoch 14/100
250/250 [==============================] - 61s 242ms/step - loss: 5.7128 -
rmse: 2.3901 - val_loss: 26.7188 - val_rmse: 5.1690
Epoch 15/100
250/250 [==============================] - 61s 242ms/step - loss: 5.8171 -
rmse: 2.4119 - val_loss: 15.2567 - val_rmse: 3.9060
Epoch 16/100
250/250 [==============================] - 61s 242ms/step - loss: 5.2777 -
rmse: 2.2973 - val_loss: 61.7677 - val_rmse: 7.8592
Epoch 17/100
250/250 [==============================] - 61s 242ms/step - loss: 8.9798 -
rmse: 2.9966 - val_loss: 116.6043 - val_rmse: 10.7983
Epoch 18/100
250/250 [==============================] - 60s 242ms/step - loss: 6.0367 -
rmse: 2.4570 - val_loss: 11.1855 - val_rmse: 3.3445
Epoch 19/100
250/250 [==============================] - 60s 241ms/step - loss: 7.0206 -
rmse: 2.6496 - val_loss: 157.4581 - val_rmse: 12.5482
Epoch 20/100
250/250 [==============================] - 60s 240ms/step - loss: 7.4522 -
rmse: 2.7299 - val_loss: 210.9105 - val_rmse: 14.5228
Epoch 21/100
250/250 [==============================] - 60s 241ms/step - loss: 10.0948
- rmse: 3.1772 - val_loss: 18.0399 - val_rmse: 4.2473
```

```
Epoch 22/100
250/250 [==============================] - 61s 242ms/step - loss: 5.0645 -
rmse: 2.2505 - val_loss: 21.2375 - val_rmse: 4.6084
Epoch 23/100
250/250 [==============================] - 60s 241ms/step - loss: 5.7177 -
rmse: 2.3912 - val_loss: 28.5047 - val_rmse: 5.3390
Epoch 24/100
250/250 [==============================] - 60s 241ms/step - loss: 5.1064 -
rmse: 2.2597 - val_loss: 47.6090 - val_rmse: 6.8999
Epoch 25/100
250/250 [==============================] - 60s 241ms/step - loss: 4.4656 -
rmse: 2.1132 - val_loss: 51.3690 - val_rmse: 7.1672
Epoch 26/100
250/250 [==============================] - 60s 240ms/step - loss: 3.6463 -
rmse: 1.9095 - val_loss: 93.4837 - val_rmse: 9.6687
Epoch 27/100
250/250 [==============================] - 60s 241ms/step - loss: 3.6216 -
rmse: 1.9031 - val_loss: 49.9860 - val_rmse: 7.0701
Epoch 28/100
250/250 [==============================] - 60s 241ms/step - loss: 3.9304 -
rmse: 1.9825 - val_loss: 4.8757 - val_rmse: 2.2081
Epoch 29/100
250/250 [==============================] - 60s 240ms/step - loss: 4.6160 -
rmse: 2.1485 - val_loss: 159.4939 - val_rmse: 12.6291
Epoch 30/100
250/250 [==============================] - 60s 240ms/step - loss: 5.9745 -
rmse: 2.4443 - val_loss: 31.3900 - val_rmse: 5.6027
Epoch 31/100
250/250 [==============================] - 60s 241ms/step - loss: 4.9073 -
rmse: 2.2152 - val_loss: 44.5920 - val_rmse: 6.6777
Epoch 32/100
250/250 [==============================] - 60s 241ms/step - loss: 4.4296 -
rmse: 2.1047 - val_loss: 6.8120 - val_rmse: 2.6100
Epoch 33/100
250/250 [==============================] - 60s 241ms/step - loss: 6.6059 -
rmse: 2.5702 - val_loss: 103.0320 - val_rmse: 10.1505
Epoch 34/100
250/250 [==============================] - 60s 242ms/step - loss: 3.9264 -
rmse: 1.9815 - val_loss: 318.6042 - val_rmse: 17.8495
Epoch 35/100
250/250 [==============================] - 61s 242ms/step - loss: 3.7293 -
rmse: 1.9311 - val_loss: 245.8616 - val_rmse: 15.6800
Epoch 36/100
250/250 [==============================] - 61s 244ms/step - loss: 3.5809 -
rmse: 1.8923 - val_loss: 3.9251 - val_rmse: 1.9812
Epoch 37/100
250/250 [==============================] - 61s 243ms/step - loss: 3.6419 -
rmse: 1.9084 - val_loss: 23.3965 - val_rmse: 4.8370
Epoch 38/100
250/250 [==============================] - 61s 243ms/step - loss: 3.6437 -
rmse: 1.9089 - val_loss: 22.4549 - val_rmse: 4.7387
Epoch 39/100
250/250 [==============================] - 61s 242ms/step - loss: 3.5197 -
rmse: 1.8761 - val_loss: 103.7435 - val_rmse: 10.1855
Epoch 40/100
250/250 [==============================] - 61s 242ms/step - loss: 5.8539 -
rmse: 2.4195 - val_loss: 272.6473 - val_rmse: 16.5120
Epoch 41/100
250/250 [==============================] - 61s 242ms/step - loss: 2.8808 -
rmse: 1.6973 - val_loss: 97.9878 - val_rmse: 9.8989
Epoch 42/100
250/250 [==============================] - 61s 242ms/step - loss: 3.9501 -
rmse: 1.9875 - val_loss: 237.1111 - val_rmse: 15.3984
```

```
Epoch 43/100
250/250 [==============================] - 61s 242ms/step - loss: 5.9793 -
rmse: 2.4453 - val_loss: 102.9308 - val_rmse: 10.1455
Epoch 44/100
250/250 [==============================] - 61s 242ms/step - loss: 3.2876 -
rmse: 1.8132 - val_loss: 13.3443 - val_rmse: 3.6530
Epoch 45/100
250/250 [==============================] - 61s 243ms/step - loss: 2.8473 -
rmse: 1.6874 - val_loss: 4.4881 - val_rmse: 2.1185
Epoch 46/100
250/250 [==============================] - 61s 243ms/step - loss: 2.9382 -
rmse: 1.7141 - val_loss: 13.9019 - val_rmse: 3.7285
Epoch 47/100
250/250 [==============================] - 61s 242ms/step - loss: 3.5568 -
rmse: 1.8860 - val_loss: 59.7056 - val_rmse: 7.7269
Epoch 48/100
250/250 [==============================] - 61s 243ms/step - loss: 3.0542 -
rmse: 1.7476 - val_loss: 48.3846 - val_rmse: 6.9559
Epoch 49/100
250/250 [==============================] - 61s 242ms/step - loss: 4.0696 -
rmse: 2.0173 - val_loss: 21.7313 - val_rmse: 4.6617
Epoch 50/100
250/250 [==============================] - 61s 242ms/step - loss: 3.7122 -
rmse: 1.9267 - val_loss: 118.0979 - val_rmse: 10.8673
Epoch 51/100
250/250 [==============================] - 61s 242ms/step - loss: 2.6829 -
rmse: 1.6379 - val_loss: 10.7841 - val_rmse: 3.2839
Epoch 52/100
250/250 [==============================] - 61s 243ms/step - loss: 2.8031 -
rmse: 1.6742 - val_loss: 13.8609 - val_rmse: 3.7230
Epoch 53/100
250/250 [==============================] - 61s 242ms/step - loss: 2.7726 -
rmse: 1.6651 - val_loss: 21.6723 - val_rmse: 4.6553
Epoch 54/100
250/250 [==============================] - 61s 243ms/step - loss: 3.4007 -
rmse: 1.8441 - val_loss: 77.9120 - val_rmse: 8.8268
Epoch 55/100
250/250 [==============================] - 61s 243ms/step - loss: 3.0227 -
rmse: 1.7386 - val_loss: 17.7745 - val_rmse: 4.2160
Epoch 56/100
250/250 [==============================] - 61s 243ms/step - loss: 4.1116 -
rmse: 2.0277 - val_loss: 20.5534 - val_rmse: 4.5336
Epoch 57/100
250/250 [==============================] - 61s 243ms/step - loss: 2.5196 -
rmse: 1.5873 - val_loss: 1.6131 - val_rmse: 1.2701
Epoch 58/100
250/250 [==============================] - 61s 243ms/step - loss: 3.0357 -
rmse: 1.7423 - val_loss: 72.6971 - val_rmse: 8.5263
Epoch 59/100
250/250 [==============================] - 61s 243ms/step - loss: 2.4410 -
rmse: 1.5624 - val_loss: 300.2112 - val_rmse: 17.3266
Epoch 60/100
250/250 [==============================] - 61s 242ms/step - loss: 2.2377 -
rmse: 1.4959 - val_loss: 4.8804 - val_rmse: 2.2092
Epoch 61/100
250/250 [==============================] - 61s 243ms/step - loss: 2.5355 -
rmse: 1.5923 - val_loss: 3.1464 - val_rmse: 1.7738
Epoch 62/100
250/250 [==============================] - 61s 243ms/step - loss: 2.4223 -
rmse: 1.5564 - val_loss: 149.8977 - val_rmse: 12.2433
Epoch 63/100
250/250 [==============================] - 61s 243ms/step - loss: 2.3303 -
rmse: 1.5265 - val_loss: 97.8213 - val_rmse: 9.8905
```

```
Epoch 64/100
250/250 [==============================] - 61s 242ms/step - loss: 2.6361 -
rmse: 1.6236 - val_loss: 7.0856 - val_rmse: 2.6619
Epoch 65/100
250/250 [==============================] - 61s 243ms/step - loss: 2.2068 -
rmse: 1.4855 - val_loss: 42.9824 - val_rmse: 6.5561
Epoch 66/100
250/250 [==============================] - 61s 243ms/step - loss: 2.2291 -
rmse: 1.4930 - val_loss: 27.3345 - val_rmse: 5.2282
Epoch 67/100
250/250 [==============================] - 61s 242ms/step - loss: 2.2970 -
rmse: 1.5156 - val_loss: 5.9973 - val_rmse: 2.4489
Epoch 68/100
250/250 [==============================] - 61s 243ms/step - loss: 2.8215 -
rmse: 1.6797 - val_loss: 12.4237 - val_rmse: 3.5247
Epoch 69/100
250/250 [==============================] - 61s 243ms/step - loss: 2.4158 -
rmse: 1.5543 - val_loss: 12.4950 - val_rmse: 3.5348
Epoch 70/100
250/250 [==============================] - 61s 243ms/step - loss: 2.4888 -
rmse: 1.5776 - val_loss: 27.5749 - val_rmse: 5.2512
Epoch 71/100
250/250 [==============================] - 61s 243ms/step - loss: 1.9211 -
rmse: 1.3860 - val_loss: 17.0489 - val_rmse: 4.1290
Epoch 72/100
250/250 [==============================] - 61s 243ms/step - loss: 2.3726 -
rmse: 1.5403 - val_loss: 167.8536 - val_rmse: 12.9558
```