Natural Frequency and Damping Identification using CNN

By

Nishant Dey

Roll No: 002111201198

Registration Number:158587 of 2021-22

Under the Guidance of

Dr. Arghya Nandi

Department of Mechanical Engineering

Faculty of Engineering and Technology

Jadavpur University

Kolkata 70032

September 2023-November 2023

Acknowledgements

I would like to express gratitude to my project guide, Prof. Arghya Nandy, for his guidance and support throughout the project. His expertise and encouragement helped me to overcome challenges and solve the problem. I am particularly grateful for their patience and willingness to provide me with feedback whenever I seemed to run out of ideas.

I would also like to thank Jyotishman Sarkar and Aryan Paul of the CSE department for their assistance in implementing some of the problems in implementation.

I certify that this project is entirely my own work and that I have not plagiarized any material from any source.

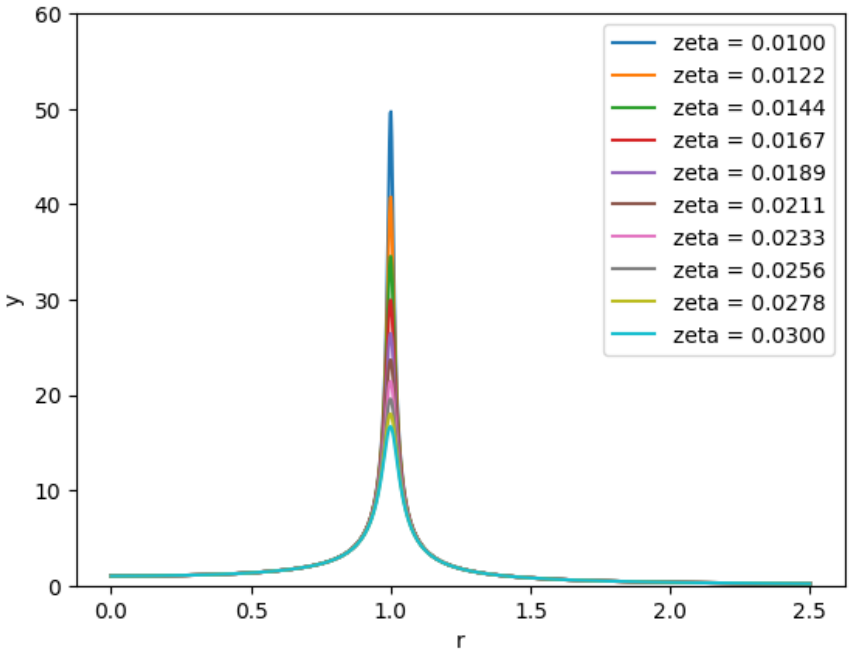
Table Of Contents

Link to Code

https://colab.research.google.com/drive/1DLtTqLNhmeY7W-A88cUS7C1PYiaLvCLr?usp=sharing

Problem Statement

The graph of harmonic forced vibration; x axis be $\frac{X}{X_{static}}$; $y\ axis\ be\ \frac{\omega}{\omega_n}$. The graph changes based on value of damping coefficient ξ. For $\frac{\omega}{\omega_n} = 1$, the value of $\frac{X}{X_{static}}$ increases

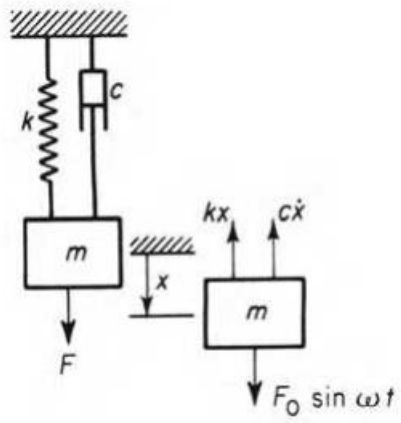


exponentially due to resonance. As the value of ξ increases, $\frac{X}{X_{static}}$ decreases. The graph is shown here.

When we obtain the graph experimentally, we need to calculate the value of damping coefficient mathematically. Here in this project, we try to provide a faster process to find ξ. We try to train a ML model that can predict the value of ξ by looking at the graph.

The ML model is built by using Convolutional Neural Network (CNN) that is widely used in the field of image Processing.

Forced Harmonic Vibration



Here we consider a single DOF system with viscous damping, excited by a harmonic force $F_0 sin\ (\omega t)$. The differential equation of the free-body diagram

$$m\ddot{x} + c\dot{x} + kx = \text{Fo}\ sin\ (\omega t)$$

The solution of the differential equation id a steady state oscillation of the same frequency as the excitation, which is of the form

$x = Xsin(\omega t - \phi)$,

Where , $X = \frac{Fo}{\sqrt{(k-m\omega^2)^2+(c\omega)^2}}$, $\phi = tan^{-1}\left(\frac{c\omega}{k-m\omega^2}\right)$

This equation can further be expressed in:

$$\frac{Xk}{F_0} = \frac{1}{\sqrt{[1-(\frac{\omega}{\omega_n})^2]^2 + \left[2\xi\left(\frac{\omega}{\omega_n}\right)\right]^2}}$$

$$tan\ \phi = \frac{2\xi\frac{\omega}{\omega_n}}{1-(\frac{\omega}{\omega_n})^2}$$

where $\omega_n = \sqrt{\frac{k}{m}} = natural\ frequency$ of undamped oscillation

$C_c = 2m\omega_n = critical\ damping$

$\xi = \frac{c}{C_c} = damping\ factor$

$\frac{c\omega}{k} = 2\xi\frac{\omega}{\omega_n}$

Why Deep Learning and CNN?

This problem, by default is an image processing problem. So, we need to use deep learning to solve the problem. Deep Learning has mainly 3 class of neural networks, they are:

1. Multi-layer Perceptrons (MLPs)
2. Convolutional Neural Networks (CNNs)
3. Recurrent Neural Networks (RNNs)

Each type is suitable for a particular type of problem where they give the best output. We choose CNNs because they are designed to map image data to an output variable. They are able to maintain an internal representation of a 2D image. We can train the model to learn about the position in variant structures in data which are important in working with images.

Development of Model

The following steps need to be followed if we want to train a model with good accuracy.

1. Data Collection
2. Data Pre-processing
3. Model Selection
4. Model Hyperparameter Tuning
5. Model Training
6. Model evaluation

Data Collection

We know that $\frac{X}{X_{st}} = \frac{1}{\sqrt{[1-(\frac{\omega}{\omega_n})^2]^2+\left[2\xi\left(\frac{\omega}{\omega_n}\right)\right]^2}}$.

We plot this equation using matplotlib for different value of ξ. We know that the peak of $\frac{X}{X_{st}}$occurs at $\frac{\omega}{\omega_n} = 1$. Lower the value of ξ,higher the peak of $\frac{X}{X_{st}}$. The peak falls exponentially as seen earlier in the graph. The value of $\frac{X}{X_{st}}$ is around 50 for ξ =0.1 and it is less than 20 for ξ=0.3. So it is important to safely capture the change in value in the model.

We generate 100 images for ξ in the range of 0.1 to 0.3. We choose to keep the range small to capture the fast variation in peak. We can extend the model as needed but in that case we need to change things accordingly and try and see what happens.

```
graph_dir = "my_graphs"
if not os.path.exists(graph_dir):
    os.makedirs(graph_dir)

# y=x/xst
# r=w/wn
r = np.linspace(0, 2.5, 1000)
zeta = np.linspace(0.01, 0.03, 100)
np.random.shuffle(zeta)
labels = ["zeta = {:.4f}".format(z) for z in zeta]
```

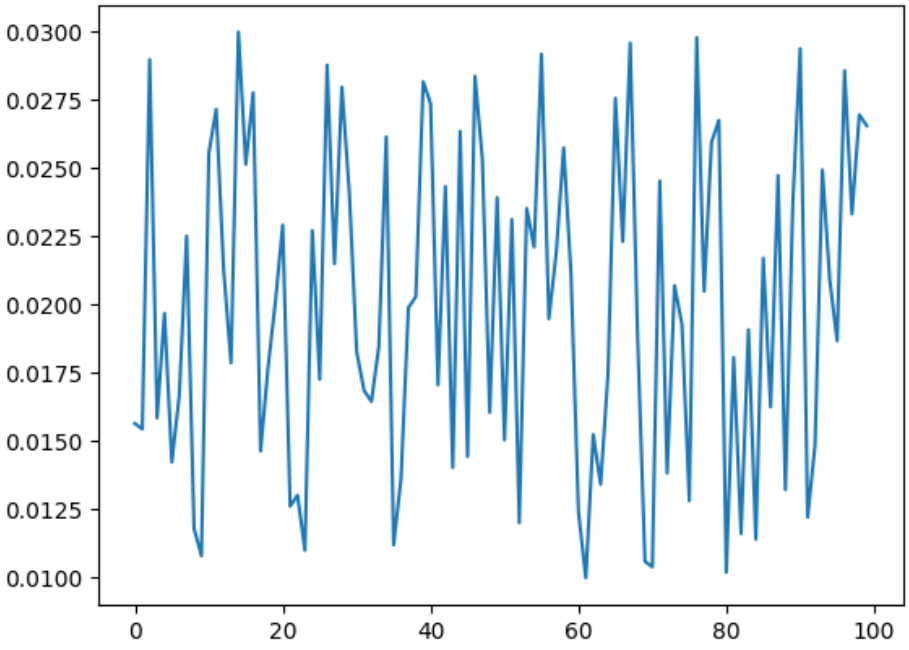
```
i = 0
for zeta_value in zeta:
    file_name = os.path.join(graph_dir,
f"zeta={zeta_value:.4f}.png")
    plt.plot(r, 1/(np.sqrt(np.square(1-
np.square(r))+np.square(2*zeta_value*r))), label=labels[i])
    plt.xlabel("r")
    plt.ylabel("y")
    plt.ylim([0,60])
    plt.savefig(file_name)
    plt.show()
    i += 1

plt.close()
```

Explanation

1. All the graphs generated will be stored in directory called “my_graphs”.
2. The y axis of the graph will range from [0-60] to maintain uniformity in the scales of all the graphs generated. Unnecessary details like legend, title, x axis, y-axis should be avoided to avoid un-necessary complications in reading images.
3. np.random.shuffle(zeta) was done because I do not want the model to learn the linear pattern in which the data was generated. By learning diverse images, it improves generalisation, reduces over fitting and encourages the model to learn the features in the image. Here is a sample image of variation in data.

```
Zeta=[0.01565657 ,0.01545455, 0.0289899, ……. 0.0269697, 0.02656566]
```

The model will first train for ξ=0.01565657, then for ξ=0.01545455 and so on.



Data Pre-Processing

There are a number of steps that we take before images are fed to the model.

Step 1:

We convert all the images to grayscale images

```
graph_dir = "my_grayscale_graphs"
if not os.path.exists(graph_dir):
    os.makedirs(graph_dir)

existing_graphs="my_graphs"
labels = ["zeta = {:.4f}".format(z) for z in zeta]
files = os.listdir(existing_graphs)

for file in files:
    file_path = os.path.join(existing_graphs, file)
    image = cv2.imread(file_path)
    gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    cv2_imshow(gray_image)
    grayscale_image_path = os.path.join(graph_dir, "grayscale_" +
file)
    cv2.imwrite(grayscale_image_path, gray_image)
```

Explanation

1. We create a new directory “my_grayscale_graphs” to store the grayscale graphs.
2. Then we read the graphs that we generated in previous code snippet using os.listdir(). Then we iterate over each and every file and read it with python library cv2. We convert it to grayscale using cv2.cvtColor() and save the file un the above created directory.

This was done because colour images have three channels (Red, Green, Blue) and so it will take three times the time to process a RGB image than a grayscale image.

Step 2:

We resize the grayscale images to size (256, 256)

```
from PIL import Image

graph_dir = "my_resized_grayscale_graphs"
if not os.path.exists(graph_dir):
    os.makedirs(graph_dir)

existing_graphs="my_grayscale_graphs"
files = os.listdir(existing_graphs)
for file in files:
    file_path = os.path.join(existing_graphs, file)
    image = Image.open(file_path)
```

```
    resized_image = image.resize((256, 256))
    image_path = os.path.join(graph_dir, "RESIZED_" + file)
    resized_image.save(image_path)
```

Explanation

1. We create a new directory to store the resized images.
2. We read the existing images and resize them using image.resize() and store it in the new directory that we just created.

Resizing the images will reduce time required to process images.

Step 3:

Now we want to convert these images to arrays. Every image is actually a grid of pixels with each pixel having a specific colour. Grayscale images have pixels which are either black or white in colour. So we can map it as follows: if a particular pixel is white it is represented as 0 in the array and vice-versa.

So, our image of 256×256 will be mapped into a 2D array of same size.

```
image_files =
os.listdir('/content/my_resized_grayscale_graphs')
x_train = []

for image_file in image_files:
    image =
cv2.imread(f'/content/my_resized_grayscale_graphs/{image_file}',0)
    image_array = np.array(image)
    x_train.append(image_array)

x_train = np.array(x_train)
x_train=x_train/255
```

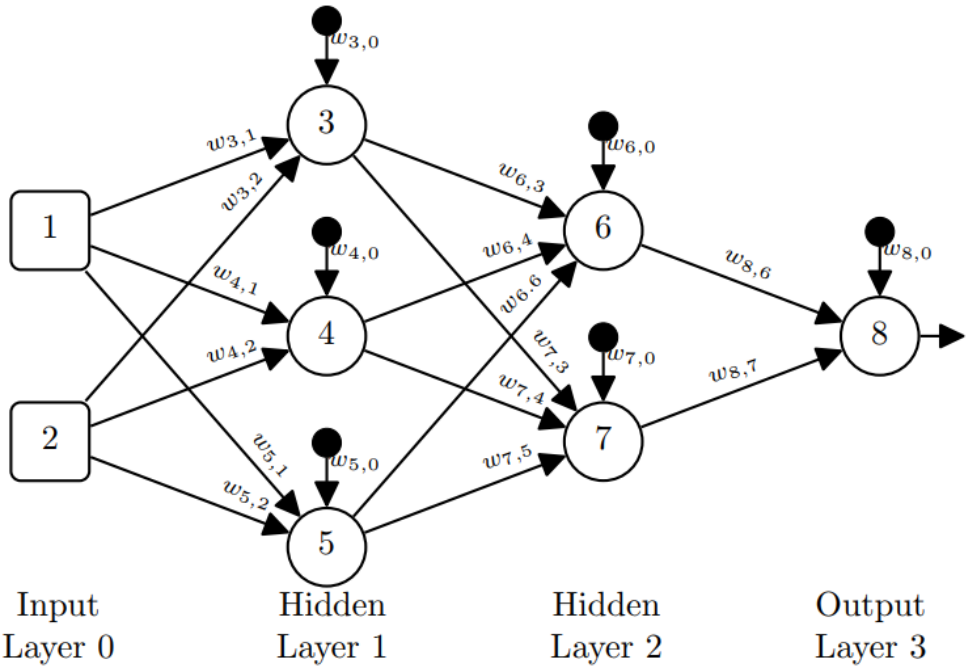
Explanation

1. We read all the images into the variable image_files. the function np.array() is able to do the mapping of images as described earlier. So image_array is essentially has the shape (256,256) . We then append it to array x_train.
2. All images are iteratively appended to x_train. So x_train has the shape (100,256,256), since we have generated 100 images.
3. The last line was added because np.array(image) converts the image into array whose elements have the range [0,255]. To convert it to range [0,1] we divide it by 255.

This completes pre-processing of data. We can convert feed it to the model.

Model Definition

The picture shows a basic idea of a CNN architecture. The first layer is a input layer which will receive the array we created in the previous step. There are some hidden layers in between which are responsible for extracting relevant features from the input image



.

Model Selection is a difficult process. There is a lot of options to choose from. It may seem that model selection is governed by precise mathematical metrics. However, every problem has a unique set of challenges, and optimal model is best decided by iterative process of exploration, experience and continuous refinement.

The problem that we are solving today is regression problem, and not a classification problem. So, we take inspiration from most common regression problems already solved, for example- prediction of human age from photograph of human faces.

The model that we build here is a result of multiple iterations to reduce the MSE (Mean Squared Error) and reduce the value of loss function as far as possible. There can exist better models for this problem. We want the model to be generalised and not prone to over fitting while training.

```
model = Sequential()
model.add(Conv2D(16, (3, 3), activation='relu',
kernel_initializer='he_uniform', padding='same', input_shape=(256,
256, 1)))
model.add(MaxPooling2D((2, 2)))
model.add(Flatten())
model.add(Dense(64, activation='relu',
kernel_initializer='he_uniform'))
model.add(Dense(32, activation='relu'))
model.add(Dense(1, activation='linear'))
```

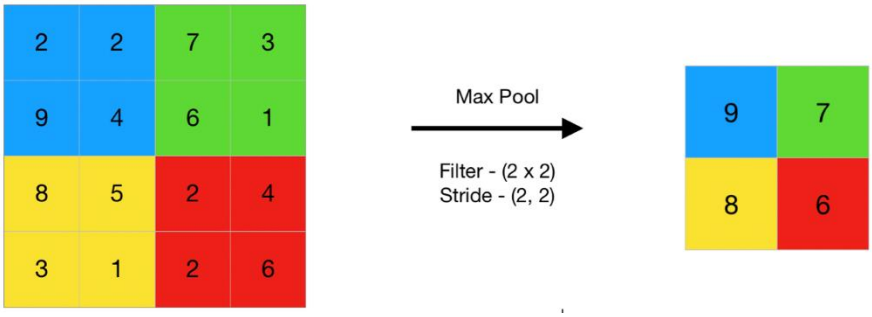
Explanation:

1. Sequential model is the simplest and straight forward neural architecture to implement. It is a plain stack of layers. Each layer has only one input tensor and only one output tensor. It is the most appropriate model if:
    a. There is only one input and only one output in this system.
    b. Output of previous layer is the input of the next layer, i.e., we do not want any layer sharing.
2. We add a single layer of convolutional layer. A convolutional layer contains a set of filters or kernels that is applied to the image.
    a. We add 16 filters each having kernel size (3,3). A filter is like a small matrix that slides across the input image. The model learns about only about the filter at a time. It is responsible for detecting patterns in the image.
    A large kernel size allows to capture more details but a smaller size focuses on smaller details.
    We decided 16 filters because it is the most standard in problems. We tried adding 32 filters but MSE and loss turns to be NAN. The reason is unknown.
    b. Activation helps us to include non-linearity in the model. It allows networks to model representations and functions that are not possible through simple linear models.
    How it works? Activation function computes the weighted sum of inputs and biases, and decide whether the neuron should be activated or not.
    Different activation functions are commonly used. For example-
        i. **Sigmoid:** It is mostly used for classification problems. The neurons having sigmoid activation takes input in (-∞,∞) and give output in [0,1] and so it mimics probability problems.
        ii. **Tanh:** It is similar to sigmoid, but gives output in range [-1,1]. It has a larger maximum gradient in comparison to sigmoid function.
        iii. **ReLU** (Rectified Linear Unit): Its output depends on the function:
        $$rectifier(z) = max(0, z)$$
        where z=input to the function.
        iv. **Maxout:** It is a piecewise linear function that returns the maximum of inputs. It is generally used when a dropout regularisation technique is used.
        v. **Softmax**: It calculates the probability distribution of event over all possibilities.

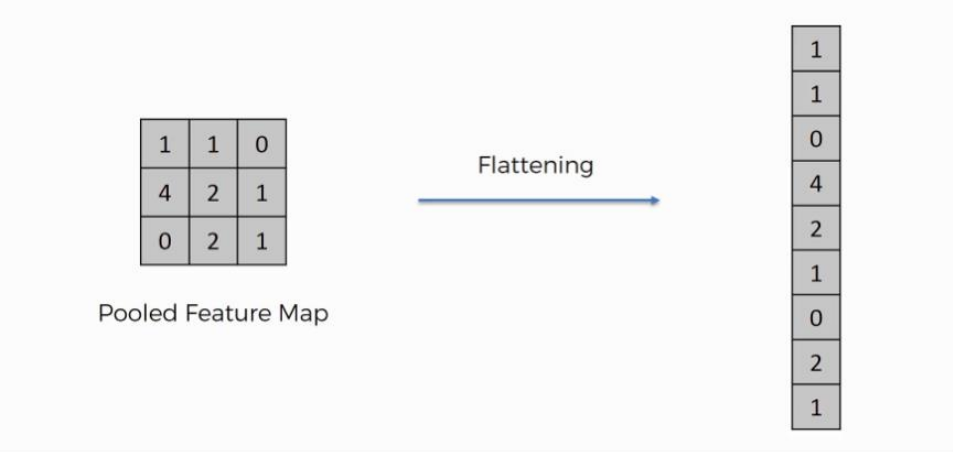
    We have used the ReLU in the convolutional layer because it was found to give the best results in the given problem. We tried using softmax function, but it gives a range of probabilities, which is not required in our case.

    c. The input shape is (256,256,1) because each image is a 2D array is of size (256,256) and it is a grayscale image so number of channels is 1.
3. Pooling Layer summarises the features lying in each channel of feature map and summarising features lying within the region covered by the filter. Max pooling selects the maximum element from the region of feature map covered by filter. As a result it reduces the dimensions of the feature map while retaining the most important

information. We choose kernel size of (2,2). Here is a example of how Pooling layer works.



4. Next comes the flatten layer. Idea behind flattening operation is to covert data to a 1D array before feeding to the next layer. The output of the CNN layers is 2D, and we flattened it into a single long array before feeding it to the dense layers.



5. In a dense layer, all neurons of the layer is connected to every neuron of the preceding layer. By using dense layer, we want to gradually reduce the dimensionality of the feature and finally get a single output.
6. The choice of activation functions in the output layer is matter of experience and experimentation. We have decided upon these after sufficient number of trials.

Model Training

Training of model is easy. We used the commands.

```
model.compile(
    optimizer=SGD(learning_rate=0.001, momentum=0.9),
    loss='mse',
    metrics=[RootMeanSquaredError()]
)
history=model.fit(x_train,zeta,epochs=15,batch_size=15,verbose=1)
```

Explanation:

1. Stochastic Gradient Descent (SGD) is a optimisation algorithm. It decides hyper-parameters based on which model is trained, i.e, the weights and other parameters of the neural network is decided.

2. Learning rate decides the step or rate at which algorithm converges to a solution, and achieves the minimum loss function. The value of this function lies in the range [0,1].
3. Loss function compares the target and the predicted output values. It measures how well the neural network models the training data. While training, we aim to minimise this loss between the predicted and target outputs.
4. We choose loss function to be mse which stands for mean squared error.

The results of the loss function and how well the training happens, varies for each runtime. It will give different losses each time the training happens from scratch. Here is a typical output of the training process.

```
Epoch 1/15
7/7 [==============================] - 3s 304ms/step - loss:
168.3000 - root_mean_squared_error: 12.9731
Epoch 2/15
7/7 [==============================] - 2s 291ms/step - loss: 0.0039
- root_mean_squared_error: 0.0624
Epoch 3/15
7/7 [==============================] - 2s 285ms/step - loss: 0.0014
- root_mean_squared_error: 0.0373
Epoch 4/15
7/7 [==============================] - 2s 328ms/step - loss:
7.2356e-04 - root_mean_squared_error: 0.0269
Epoch 5/15
7/7 [==============================] - 3s 393ms/step - loss:
4.3162e-04 - root_mean_squared_error: 0.0208
Epoch 6/15
7/7 [==============================] - 2s 289ms/step - loss:
3.4643e-04 - root_mean_squared_error: 0.0186
Epoch 7/15
7/7 [==============================] - 2s 291ms/step - loss:
1.6640e-04 - root_mean_squared_error: 0.0129
Epoch 8/15
7/7 [==============================] - 2s 294ms/step - loss:
1.5819e-04 - root_mean_squared_error: 0.0126
Epoch 9/15
7/7 [==============================] - 2s 291ms/step - loss:
1.2363e-04 - root_mean_squared_error: 0.0111
Epoch 10/15
7/7 [==============================] - 3s 386ms/step - loss:
1.3194e-04 - root_mean_squared_error: 0.0115
Epoch 11/15
7/7 [==============================] - 3s 342ms/step - loss:
9.2731e-05 - root_mean_squared_error: 0.0096
Epoch 12/15
7/7 [==============================] - 2s 287ms/step - loss:
6.3296e-05 - root_mean_squared_error: 0.0080
```

```
Epoch 13/15
7/7 [==============================] - 2s 289ms/step - loss:
5.1507e-05 - root_mean_squared_error: 0.0072
Epoch 14/15
7/7 [==============================] - 2s 295ms/step - loss:
8.0097e-05 - root_mean_squared_error: 0.0089
Epoch 15/15
7/7 [==============================] - 2s 293ms/step - loss:
7.1639e-05 - root_mean_squared_error: 0.0085
```

We find that loss and the root mean squared error decreases in each epoch. It finally converges.

Here is summary of the model parameters.

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 256, 256, 16)      160

 max_pooling2d (MaxPooling2  (None, 128, 128, 16)      0
 D)

 flatten (Flatten)           (None, 262144)            0

 dense (Dense)               (None, 64)                16777280

 dense_1 (Dense)             (None, 32)                2080

 dense_2 (Dense)             (None, 1)                 33

=================================================================
Total params: 16779553 (64.01 MB)
Trainable params: 16779553 (64.01 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Model Evaluation

To evaluate the model, we need to prepare the data as we described earlier. We tested the model for 30 values of zeta in range [0.01,0.03]. We make sure to keep the range of zeta same as the values of zeta on which it was trained.

To use the model to test its accuracy, we use

```
loss, accuracy = model.evaluate(x_test, zeta)
```

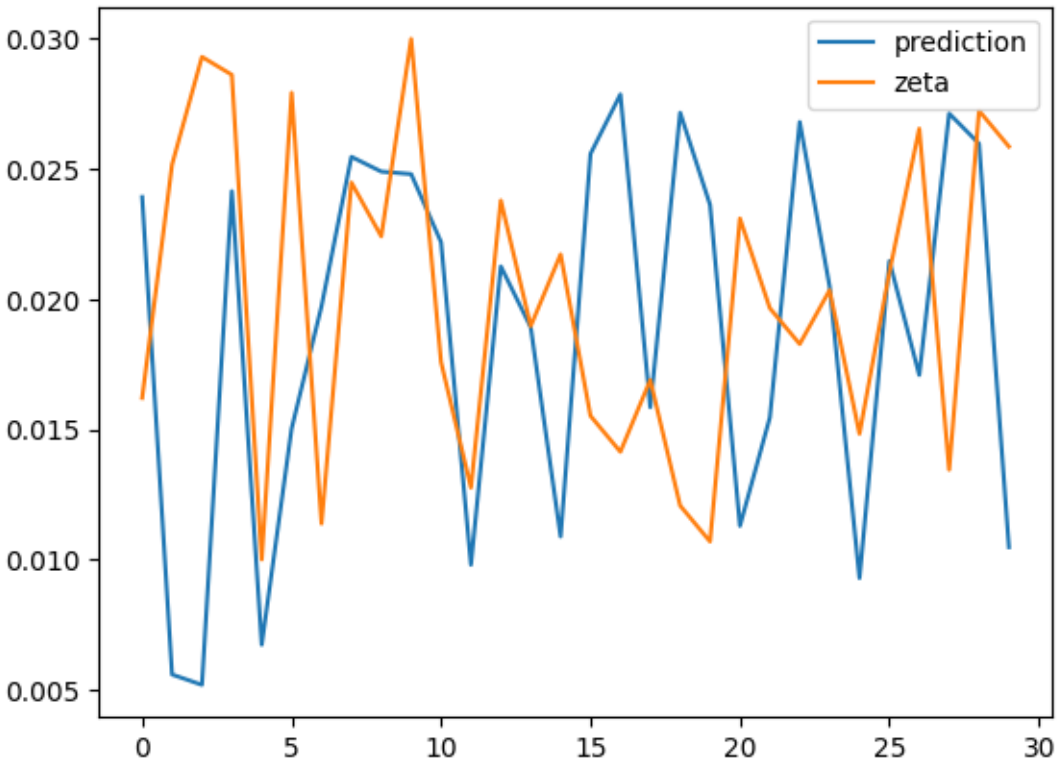
The typical output of this command looks like this. It will depend on the runtime. It will vary each time the model is run

```
1/1 [==============================] - 1s 647ms/step - loss:
1.1906e-04 - root_mean_squared_error: 0.0109
```

To use the model to make prediction, we use,

```
prediction=model.predict(x_test)
```

We plot the prediction and the original zeta values on the same graph, we find the following result.

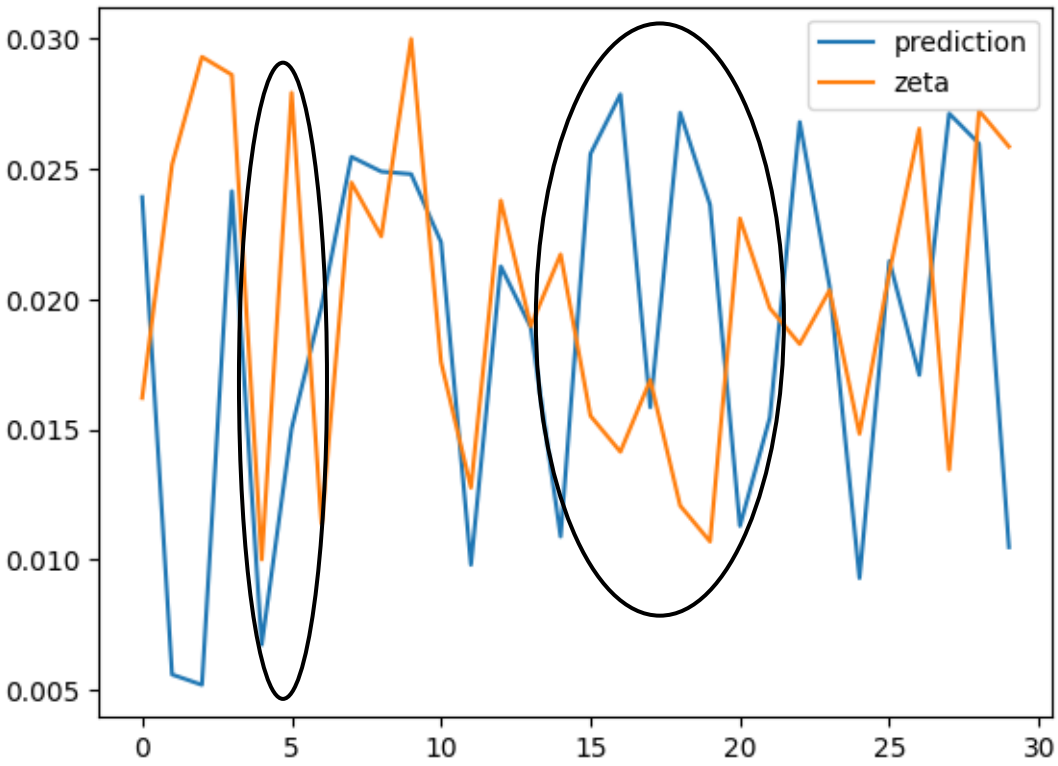


Comments on the result

The results are pretty interesting but there are few visible problems in it. I will point few of them.

1. The first circle from the left. We find that the actual zeta value moves upward, but it has predicted a downward trend. This problem has occurred in quite a few points. And I find this to be a serious problem.

I have highlighted this problem with red circles in the figure below.



Scope of Improvement

There is definitely a lot of improvement that can be made to the model.

1. First of all the model is very erratic. On the same set of data, using the same parameters, the model gives wide variety of results. The training of the model somehow does not happen properly. We want the model to be consistent enough to be used in a real life application.
2. We can try training the model on a wider value of zeta and try and see what happens. The model somehow appears to be underfitting the data.
3. Further trial and error can be done with the model to try and improve it.

References

1. https://towardsdatascience.com/loss-functions-and-their-use-in-neural-networks-a470e703f1e9
2. https://machinelearningmastery.com/regression-tutorial-keras-deep-learning-library-python/
3. https://stackoverflow.com/questions/46680481/activation-function-for-output-layer-for-regression-models-in-neural-networks
4. https://stackoverflow.com/questions/56133503/which-settings-to-use-in-last-layer-of-cnn-for-regression
5. https://machinelearningmastery.com/regression-tutorial-keras-deep-learning-library-python/
6. https://machinelearningmastery.com/choose-an-activation-function-for-deep-learning/
7. https://www.analyticsvidhya.com/blog/2022/03/introduction-to-densenets-dense-cnn/
8. https://stats.stackexchange.com/questions/222883/why-are-neural-networks-becoming-deeper-but-not-wider
9. https://datascience.stackexchange.com/questions/66597/how-to-use-cnn-to-deal-with-a-2d-regression-problem
10. https://datascience.stackexchange.com/questions/72347/how-to-determine-number-of-neurons-setup-in-convolutional-neural-networks
11. https://datascience.stackexchange.com/questions/44124/when-to-use-dense-conv1-2d-dropout-flatten-and-all-the-other-layers
12. https://stackoverflow.com/questions/65260309/still-confused-about-model-train
13. https://stackoverflow.com/questions/70653364/model-predict-return-an-array-instead-of-a-number-label
14. https://machinelearningmastery.com/using-activation-functions-in-neural-networks/
15. https://machinelearningmastery.com/how-to-develop-a-cnn-from-scratch-for-cifar-10-photo-classification/
16. https://machinelearningmastery.com/using-cnn-for-financial-time-series-prediction/
17. https://towardsdatascience.com/top-10-cnn-architectures-every-machine-learning-engineer-should-know-68e2b0e07201
18. https://datascience.stackexchange.com/questions/36238/what-does-the-output-of-model-predict-function-from-keras-mean
19. https://machinelearningmastery.com/how-to-develop-a-convolutional-neural-network-from-scratch-for-mnist-handwritten-digit-classification/

20. https://machinelearningmastery.com/how-to-read-write-display-images-in-opencv-and-converting-color-spaces/?__s=s5zox4mbrudfmuqgsfx3&utm_source=drip&utm_medium=email&utm_campaign=A+gentle+introduction+to+OpenCV&utm_content=A+gentle+introduction+to+OpenCV
21. https://medium.com/@muhammadshoaibali/flattening-cnn-layers-for-neural-network-694a232eda6a
22. Understanding Deep Learning, Application in Rare Event Prediction, Chitta Ranjan.
23. Fundamentals of Machine Learning For Predictive Data Analytics, John. D. Kelleher.