Composite OAM Beam Decomposition Via Convolutional Neural Network

A Project Report

submitted by

RAHUL R

in partial fulfilment of the requirements
for the award of the degree of

MASTER OF TECHNOLOGY

DEPARTMENT OF ELECTRICAL ENGINEERING.
INDIAN INSTITUTE OF TECHNOLOGY MADRAS.

MAY 2022

THESIS CERTIFICATE

This is to certify that the thesis titled **Composite OAM Beam Decomposition Via Convolutional Neural Network**, submitted by **RAHUL R**, to the Indian Institute of Technology, Madras, for the award of the degree of **Masters of Technology**, is a bonafide record of the research work done by him under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Prof. Dr. Ananth Krishnan
Research Guide
Professor
Dept. of Electrical Engineering
IIT-Madras, 600 036

Date: 27th May 2022

Place: Chennai

ACKNOWLEDGEMENTS

Firstly I would like to thank my thesis advisor Dr. Ananth Krishnan for guiding me throughout this project and motivating me. He has been very encouraging for working on new ideas and I am grateful for the support which he gave me whenever I was in needed. The door to Prof. Dr. Ananth krishnan office was open whenever I ran into trouble spot or had a question about my research work.

I am thankful to faculties of Electrical department for the support through out my M.tech post graduate degree.

Finally, I am thankful to my lab partner Nirjhar for helping me whenever necessary and guiding me throughout my project.

ABSTRACT

KEYWORDS: O.A.M; L.G.Beams.

The Vortex beam carrying Orbital Angular Momentum (O.A.M) has attracted great attention in optical communication field. Here, decomposition of composite O.A.M beams into its mode weights and mode phase of L.G Beams was achieved through two separate Alex-net Architecture. Negative charge was included in generating composite beam, but there seems to exists another L.G Beams mode set which produces same intensity pattern. New algorithm was developed in order to over come this issue. Radial charge was also introduced here, At most two L.G beams was combined to produce the composite beam and was able to accurately reconstruct the image,even when the noise was added to the original composite beam.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS i

ABSTRACT iii

LIST OF FIGURES viii

ABBREVIATIONS ix

NOTATION xi

1 INTRODUCTION 1

1.1 **Laguerre-Gaussian(LG) modes.** 1

1.2 Generation of O.A.M beams. 3

1.2.1 **Fork grating.** 3

1.2.2 **Optical element:** 3

2 Composite Beam Generation 5

3 Mirror Charges Correction. 7

4 Dataset Generation and Noise. 11

4.1 **Different types of noise** 11

4.1.1 **Translation:** 11

4.1.2 **Yaw rotation:** 11

4.1.3 **Pitch rotation:** 11

4.1.4 **Shot Noise:** 11

4.1.5 **Pepper Noise:** 12

4.1.6 **Gaussian Blur Noise:** 12

4.2 **Dataset Generation:** 14

5 Normalisation: 15

6 CNN Architecture **17**

6.1 Convolution Neural Network Theory 17

6.2 C.N.N Hyper parameter Tuning using Wand b: 18

6.3 Alex-Net: 20

7 Results and Discussion: **23**

8 CODE **29**

8.1 Imports of Libraries: 29

8.2 Dataset Generation. 30

8.2.1 Global variables. 30

8.2.2 Pure Mode Field profile. 30

8.2.3 Mode Selection Functions. 31

8.2.4 Noise Adding Functions. 33

8.2.5 Expected Output. 38

8.3 Data-loader for Training and validation Set. 38

8.3.1 Training dataset. 39

8.3.2 Validation dataset. 40

8.4 C.N.N Training. 41

8.4.1 C.N.N Architecture 41

8.4.2 Loss and Optimiser Functions 44

8.4.3 Defining Training and validation Functions. 45

8.4.4 Training Network. 48

8.4.5 Wand-b 51

LIST OF FIGURES

1.1 Intensity of LG_{00}. 2
1.2 Phase of LG_{00}. 2
1.3 Intensity of LG_{03}. 2
1.4 Phase of LG_{03}. 2
1.5 Intensity of LG_{23}. 2
1.6 Phase of LG_{23}. 2
1.7 Fork grating. 3
1.8 intensity of T.C =1. 3
1.9 Phase of T.C=1. 3

3.1 Composite intensity profile of set1. 8
3.2 Composite intensity profile of set2. 8
3.3 Mirror Charges Case 1. 10
3.4 Mirror Charges Case 2. 10
3.5 Mirror Charges Case 3. 10

4.1 Composite beam. 13
4.2 Translation of $t_x = t_y = 30$. 13
4.3 Composite beam. 13
4.4 yaw rotation of 45^0. 13
4.5 Composite beam. 13
4.6 pitch rotation of 45^0. 13
4.7 Composite beam. 13
4.8 Shot noise image. 13
4.9 Composite beam. 14
4.10 pepper noise image. 14
4.11 Composite beam. 14
4.12 Blue noisy image. 14

5.1 Without Normalisation. 16

5.2 With Normalisation. 16

6.1 Batch 01 hyper parameter tuning.. 19
6.2 Batch 02 hyper parameter tuning.. 19
6.3 Batch 03 hyperparameter tuning.. 20
6.4 Tuned Alex-net Architecture. 21
6.5 Loss curve for weights. 21
6.6 Loss curve for phase. 21

7.1 Input Composite beam to CNN 1. 24
7.2 Reconstructed image 1. 24
7.3 Weights prediction 1. 24
7.4 Phase prediction 1. 24
7.5 Input Composite beam to CNN 2. 25
7.6 Reconstructed image 2. 25
7.7 Weights prediction 2. 25
7.8 Phase prediction 2. 25
7.9 Input Composite beam to CNN 3. 26
7.10 Reconstructed Image 3. 26
7.11 Weights Prediction 3. 26
7.12 Phase Prediction 3. 26
7.13 Input Composite beam to CNN 4. 27
7.14 Reconstructed Image 4. 27
7.15 Weights Prediction 4. 27
7.16 Phase Prediction 4. 27
7.17 Input Composite beam to CNN 5. 28
7.18 Reconstructed Image 5. 28
7.19 Weights Prediction 5. 28
7.20 Phase Prediction 5. 28

ABBREVIATIONS

O.A.M	Orbital Angular Momentum
C.N.N	Convolution Neural Network.
G.P.U	Graphical Processing Unit .
fc1-sz	Fully connected layer 1 size.
fc2-sz	Fully connected layer 2 size.
S.D.G	Stochastic Gradient Descent.
conv1-sz	Convolution layer 1 Number of filters.
conv5n2-sz	Convolution layer 5 and 2 Number of filters.
conv4n3-sz	Convolution layer 4 and 3 Number of filters.
F.C	Fully connected layer .
C.C.D	Charged Couple Device .

NOTATION

μ	Mean value of the dataset.
σ	Standard Deviation of the dataset.
l	Topological charge
p	Radial Charge
σ_b	Gaussian Blur sigma value.

CHAPTER 1

INTRODUCTION

In,1992 Allen recognised the light beams with carry azimuthal phase dependence of $\exp(il\phi)$ carry an orbital angular momentum where l can be a integer value, positive or negative, Such beams has helical phase fronts. O.A.M has phase singularity running along the center of the beam. The O.A.M has l azimuthal index goes from $-\infty$ to ∞ and radial charge p from 0 to ∞ in steps of 1. Each O.A.M mode consists of l and p value. This O.A.M modes are orthogonal to each other.

1.1 Laguerre-Gaussian(LG) modes.

The complete basis set of orthogonal modes, for O.A.M carrying beams is given by Laguerre-Gaussian (LG) Modes set, these mode have amplitude distribution given by LG_{pl}

$$LG_{pl}(r,\theta,\phi) = \sqrt{\frac{2p!}{\pi(p+|l|)!}}\frac{1}{w(z)}\left[\frac{r\sqrt{2}}{w(z)}\right]^{|l|}\exp\left[\frac{-r^2}{w^2(z)}\right]L_p^{|l|}\left(\frac{-2r^2}{w^2(z)}\right)\exp[il\phi]$$

$$\exp\left[\frac{ik_o r^2 z}{2(z^2+z_R^2)}\right]\exp\left[-i(2p+|l|+1)tan^{-1}\left(\frac{z}{z_R}\right)\right] \tag{1.1}$$

where the $\frac{1}{e}$ radius of the Gaussian term is given by $w(z) = w(0)[\frac{(z^2+z_R^2)}{z_R^2})]^{0.5}$ with $w(0)$ being the beam waist , z_R the Rayleigh range and $\left[-i(2p+|l|+1)tan^{-1}\left(\frac{z}{z_R}\right)\right]$ the Gouy phase.

where $L_p^{|l|}(x)$ is an associated Laguerre polynomial , obtained from the more familiar Laguerre polynomial by,

$$L_p^{|l|(x)} = (-1)^{|l|}\frac{d^{|l|}}{dx^{|l|}}L_{p+|l|}(x)$$

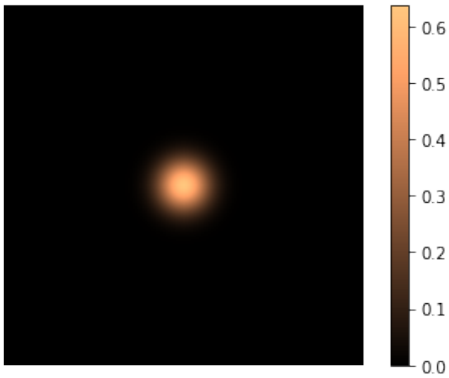

Figure 1.1: Intensity of LG_{00}.

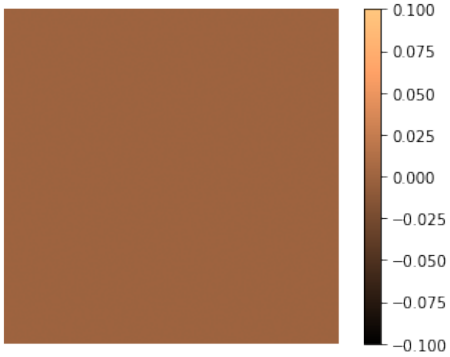

Figure 1.2: Phase of LG_{00}.

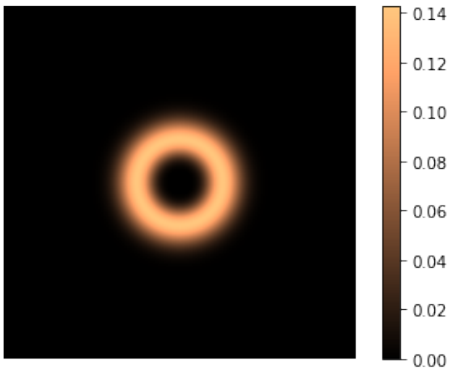

Figure 1.3: Intensity of LG_{03}.

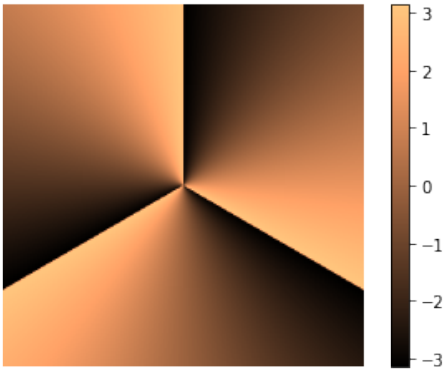

Figure 1.4: Phase of LG_{03}.

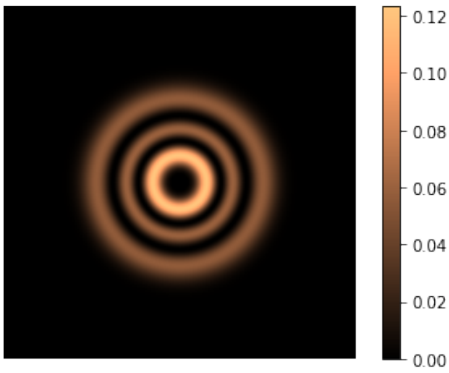

Figure 1.5: Intensity of LG_{23}.

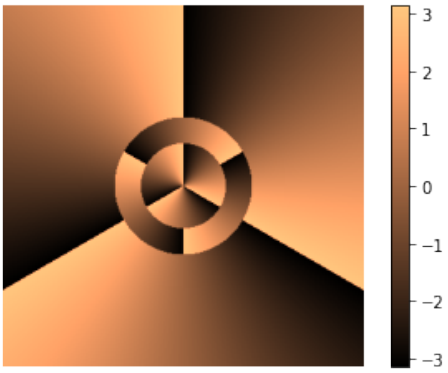

Figure 1.6: Phase of LG_{23}.

. where l is the azimuthal index and p is the number of radial nodes in the intensity distribution.

Intensity and phase profile of different O.A.M beams is shown above.

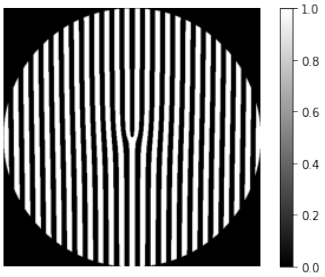

Figure 1.7: Fork grating.

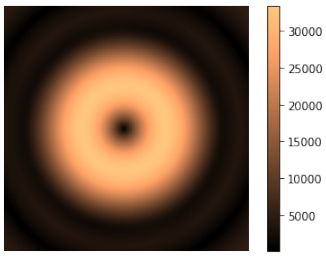

Figure 1.8: intensity of T.C =1.

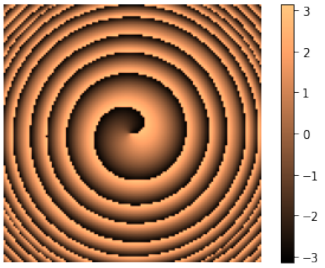

Figure 1.9: Phase of T.C=1.

1.2 Generation of O.A.M beams.

1.2.1 Fork grating.

Helical phased beams carrying O.A.M is widely generated using forked diffraction. When a plane wave Gaussian Beam is illuminated onto the fork grating, helically phased beam is produced in the first diffraction order.

1.2.2 Optical element:

O.A.M beam can be generated by passing through an optical element with helical surface. Optical thickness of the component increases with azimuthal position accordingly.

$$\frac{l\lambda\theta}{2\pi(n-1)}$$

Where n is the refractive index of the medium. But by this method requires extreme precision in the pitch of the helical path.

CHAPTER 2

Composite Beam Generation

The superimposed Optical field of different Orthogonal O.A.M Beams can be expressed as

$$U(r,\theta,\phi) = \sum_{n=1}^{N} a_n LG_{pl}^{n}(r,\theta,\phi)e^{(i\theta_n)} \tag{2.1}$$

where N is the number of modes, $LG_{pl}^{n}(r,\theta,\phi)$ is the n^{th} LG beam eigen mode, a_n are the amplitude and θ_n are phase of each eigen mode, respectively. a_n^2 is the proportion of the nth eigen mode in the superimposed optical field and satisfies this expression $\sum_{n=1}^{N} a_n^2 = 1$ which is called as mode weight and θ_n is called the mode Phase. Weights to the CNN output is given by

$$[a_1, a_2, a_3, ..., a_n]$$

and Phase is given by

$$[\theta_1, \theta_1, \theta_1, ..., \theta_n]$$

Here θ values was a angle from $[0, 2\pi]$, so by dividing the values of phase by 2π we are scaling it to $[0, 1]$, which is the output given to the CNN to train the network.

The intensity value of the O.A.M Beam is obtained by $I(x, y) = |U(r,\theta,\phi)|^2$

CHAPTER 3

Mirror Charges Correction.

So, Here we have included negative charges to create the composite beam, but it was found out that there were Same composite beam intensity profile for different charges. Detailed analysis showed that , if we invert the sign of the Topological charge(l) keeping the radial charge (p) same ,it was observed that same composite beam was produced and composite beam intensity pattern seems to change as we change the weights for the same eigen modes.

Example:

If a composite beam is formed by eigen modes of $[l, p]$,

$$[-2, 2], [-5, 1], [-1, 0]$$

The mirror charges are

$$[2, 2], [5, 1], [1, 0]$$

So, Here in this figure we have listed all the possible cases. Maroon and yellow color signifies the Mirror Charges Set, where Red color signifies where two mirror charges of different set coinciding. In the case 2 the mirror charges set is itself. So, since there are two charges set for same composite beam intensity pattern. We need to choose one charge set out of two and feed it to C.N.N. In order to choose the one we implemented a rule, choosing the set which obeys the **Rule 01**:

$$max(l - p)$$

Algorithm 01 :

1. For the given set of eigen modes. Mirror charges sets are found out.
2. Rule 01 was applied to all the charges in both of the eigen mode sets.
3. The eigen mode set is chosen, which contains highest value from Rule 01.

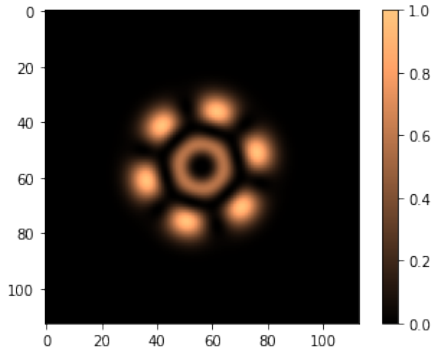

Figure 3.1: Composite intensity profile of set1.

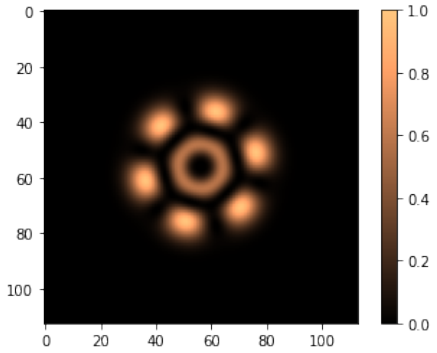

Figure 3.2: Composite intensity profile of set2.

Above Figure 3.1 and Figure 3.2 shows the same intensity profile for different charges.

For the **Figure 3.1**, the intensity pattern was formed by $[l, p]$,

$$[-8, 0], [-2, 1]$$

Weights and Phase value of Set 1 is given by.

$$[0.4500, 0.5500], [0.7600, 0.0000]$$

For the **Figure 3.2**, the intensity pattern was formed by $[l, p]$,

$$[8, 0], [2, 1]$$

Weights and Phase value of Set 2 is given by.

$$[0.4500, 0.5500], [-0.7600, 0.0000]$$

Since there is large variation of phase in highest $|l|$ charge, So we tried to maximise it and as p charges increases the discontinuity in phase profile increase so negative sign was implemented in order to reduce it in Rule 01.

There seems to be a problem in this case.

Example: Mirror charges set 1 is

$$[-2, 0], [-4, 0], [-5, 0], [6, 0]$$

Mirror charges set 2 is given by

$$[2,0],[4,0],[5,0],[-6,0]$$

Out of this two according to Rule 01 , it will choose **Mirror Charge Set 1**,Since weights are used in image reconstruction, that seems to cause a problem. Since,weights are chosen randomly, let the weights assigned in the above case be

$$[0.4,0.3,0.29.0.01]$$

Here the last charge $[6,0] and [-6,0]$ is given low weights. if we reconstruct the image there was only slight difference in composite beam reconstruction from

$$[-2,0],[-4,0],[-5,0],[6,0]$$

and

$$[-2,0],[-4,0],[-5,0]$$

If we can ignore in that case that [6,0] charge is not present then

$$[2,0],[4,0],[5,0]$$

set is chosen, which is opposite of

$$[-2,0],[-4,0],[-5,0],[6,0]$$

which was chosen. Where both intensity profile looks same. So, in order to get rid of this we have to include weight as well. the formula becomes (**Rule 02:**).

$$max(l-p+k*weights)$$

Here k was taken as 100 by trail and error method. By this formula, we can remove that issue. The mode which gives highest value from the Rule 02 that mode is called as **Principle Mode**.

p\l	-4	-3	-2	-1	0	1	2	3	4
0	0	6	12	18	24	30	36	42	48
1	1	7	13	19	25	31	37	43	49
2	2	8	14	20	26	32	38	44	50
3	3	9	15	21	27	33	39	45	51
4	4	10	16	22	28	34	40	46	52
5	5	11	17	23	29	35	41	47	53

Figure 3.3: Mirror Charges Case 1.

p\l	-4	-3	-2	-1	0	1	2	3	4
0	0	6	12	18	24	30	36	42	48
1	1	7	13	19	25	31	37	43	49
2	2	8	14	20	26	32	38	44	50
3	3	9	15	21	27	33	39	45	51
4	4	10	16	22	28	34	40	46	52
5	5	11	17	23	29	35	41	47	53

Figure 3.4: Mirror Charges Case 2.

Algorithm 02:

1. For the given set of eigen modes, Rule 02 was applied to all the charges.
2. The highest weight is called as principle mode, if the principle mode is positive we retain that mode set, else invert all the l signs.
3. If the Principle Mode has the value T.C(l)=0 then the Principle Mode is chosen as the next lowest value and the process is repeated.
4. If all the mode contains the value l=0, then the same mode is retained, and highest value from the Rule 02, is chosen as Principle Mode.

p\l	-4	-3	-2	-1	0	1	2	3	4
0	0	6	12	18	24	30	36	42	48
1	1	7	13	19	25	31	37	43	49
2	2	8	14	20	26	32	38	44	50
3	3	9	15	21	27	33	39	45	51
4	4	10	16	22	28	34	40	46	52
5	5	11	17	23	29	35	41	47	53

Figure 3.5: Mirror Charges Case 3.

CHAPTER 4

Dataset Generation and Noise.

4.1 Different types of noise

In the below example, composite beam was formed by the combination of LG_{01} and LG_{07} with proportional weights of 0.2467 and 0.7533 respectively and phase value was taken as 0.0 and $0.1788 * 2\pi$.

4.1.1 Translation:

For the original Image we added translation of $t_x = t_y = 30$ means from the center of the image , original image was translated to 30 pixels to right in x direction and y direction.

4.1.2 Yaw rotation:

If we consider the Cartesian coordinate system, where z axis is out of plane.Then rotation along the y axis rotation of image is yaw direction. so, here for the composite beam at yaw angle of 45^0 degree, Original Image was rotated.

4.1.3 Pitch rotation:

If we consider the Cartesian coordinate system, where z axis is out of plane.Then rotation along the x axis rotation of image is pitch direction. so, here for the composite beam at pitch angle of 45^0 degree,Original Image was rotated.

4.1.4 Shot Noise:

Shot noise basically arises due to the discrete nature of photons,when signal strength is low. So,usually obey Poisson distribution. Here we have added the Poisson distribution

noise,from creating the random numbers from from this distribution where mean value was taken as 0.2.

4.1.5 Pepper Noise:

In some case , Charged couple device (C.C.D) has some dead pixels. where that pixels all give dark image irrespective whatever the intensity falls on it, that to was considered here. So here we have taken cut off value as 0.9. For each pixel random number was generated if that number exceeds the cut-off value 0.9, we assign the value 0 to the pixel image. So, in this case if cut-off value is 0, then whole image becomes black, but when cut-off value is equal to 1 then the original image is retained.

4.1.6 Gaussian Blur Noise:

Blur is one such noise which is encountered, so we have added Gaussian noise of $\sigma_b = 2$, So from σ_b value we can calculate kernel size as $(6\sigma_b + 1) \times 6\sigma_b + 1)$. In this case it was 13×13 Gaussian kernel which was convolved on to the image in order to create blur.

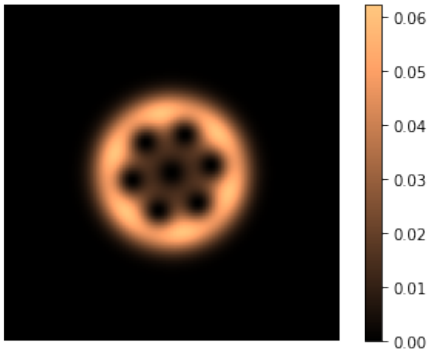

Figure 4.1: Composite beam.

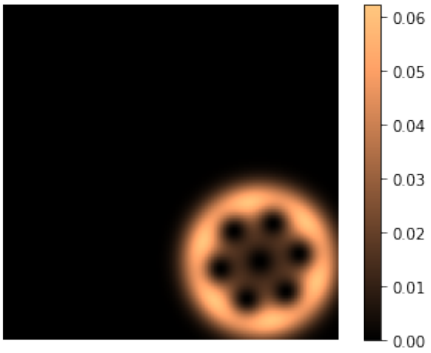

Figure 4.2: Translation of $t_x = t_y = 30$.

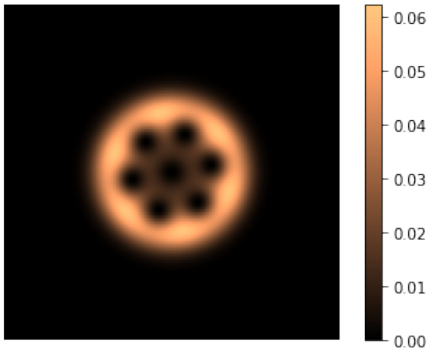

Figure 4.3: Composite beam.

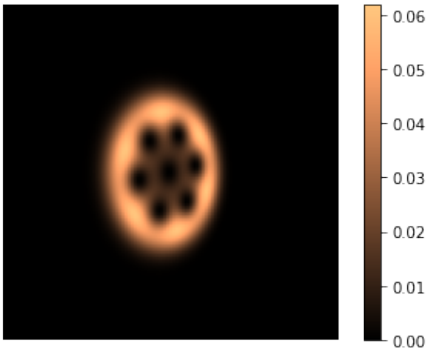

Figure 4.4: yaw rotation of 45^0.

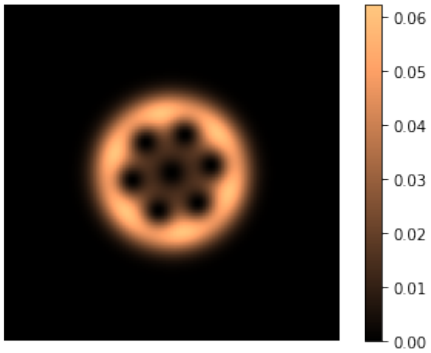

Figure 4.5: Composite beam.

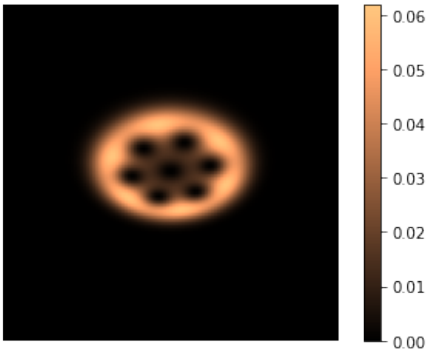

Figure 4.6: pitch rotation of 45^0.

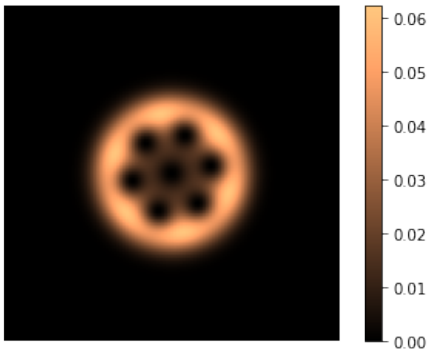

Figure 4.7: Composite beam.

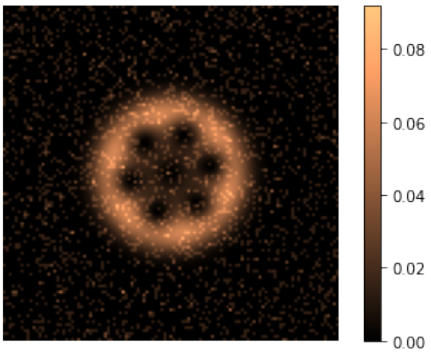

Figure 4.8: Shot noise image.

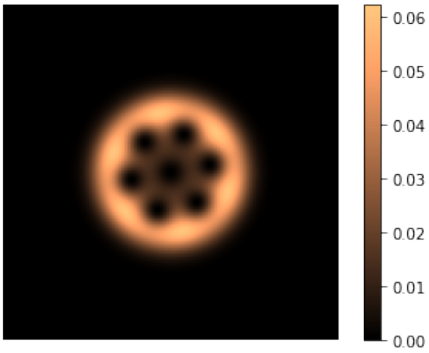

Figure 4.9: Composite beam.

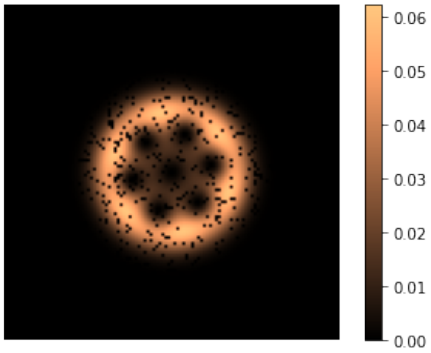

Figure 4.10: pepper noise image.

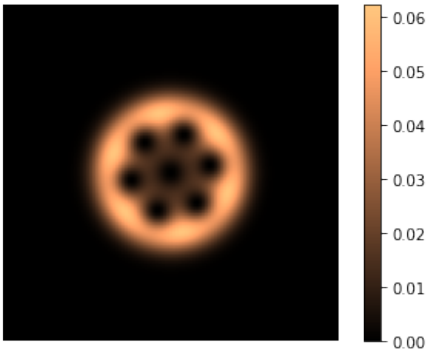

Figure 4.11: Composite beam.

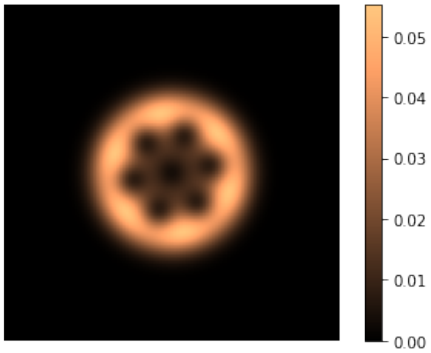

Figure 4.12: Blue noisy image.

4.2 Dataset Generation:

Generation of Dataset was done for Topological Charges (l) ranging from -9 to +9. and Radial Charge (p) ranging from 0 to 1. Random eigen modes were selected, No of eigen modes selected were less than or equal 2. Mirror charges correction algorithm was applied. For the selected eigen modes, Composite Beam Intensity profile was generated.

1. Size of the image was taken as 113×113.
2. Wavelength of λ =632.8 nm.
3. Beam waist size of $1mm$.
4. The image size was considered as $10mm \times 10mm$.

Totally 100000 (1 lakh) Training dataset was used and 10000 (10 Thousand) was used for validation and Testing dataset. For the each composite beam random only one noise was added to some images and for others there was no noise added. While generating composite beam the image was scaled to maximum value of 1 by dividing the each image by the maximum pixel value of that image.

CHAPTER 5

Normalisation:

Since the dataset of composite images generated was having different pixels range. we need to normalisation. The normalisation process was carried out as the mean(μ) and standard deviation(σ) of all the pixel value of all images was found out. Then each pixel value was found as $\frac{x-\mu}{\sigma}$ where here x is the pixel value , this was applied for all the pixel in the image for all the dataset. The figure given below the composite beam which is given to the CNN. The UN-Normalised images looks saturated, Patterns are not clearly visible. whereas the Normalised images we can see that pattern are clearly visible.

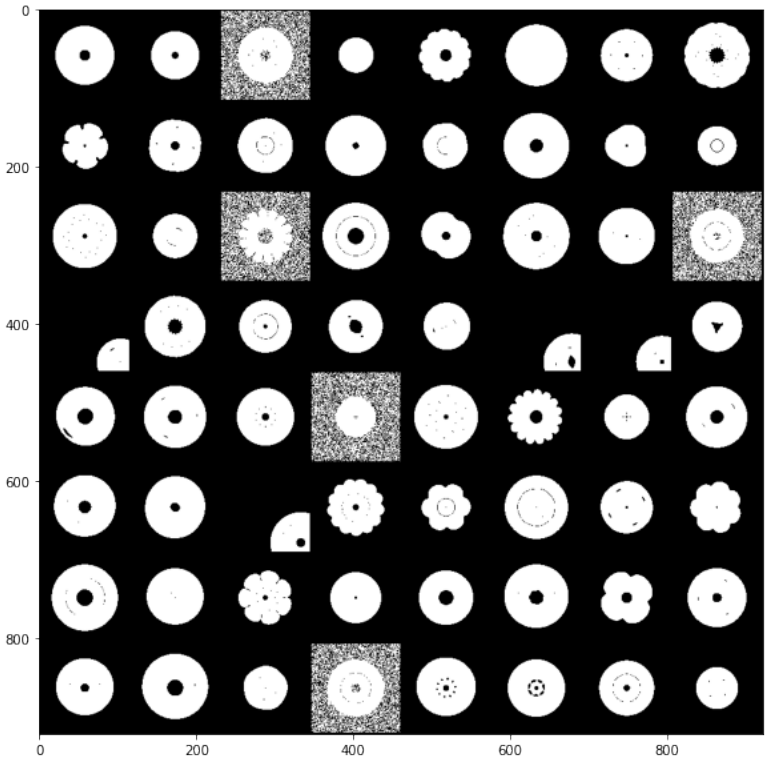

Figure 5.1: Without Normalisation.

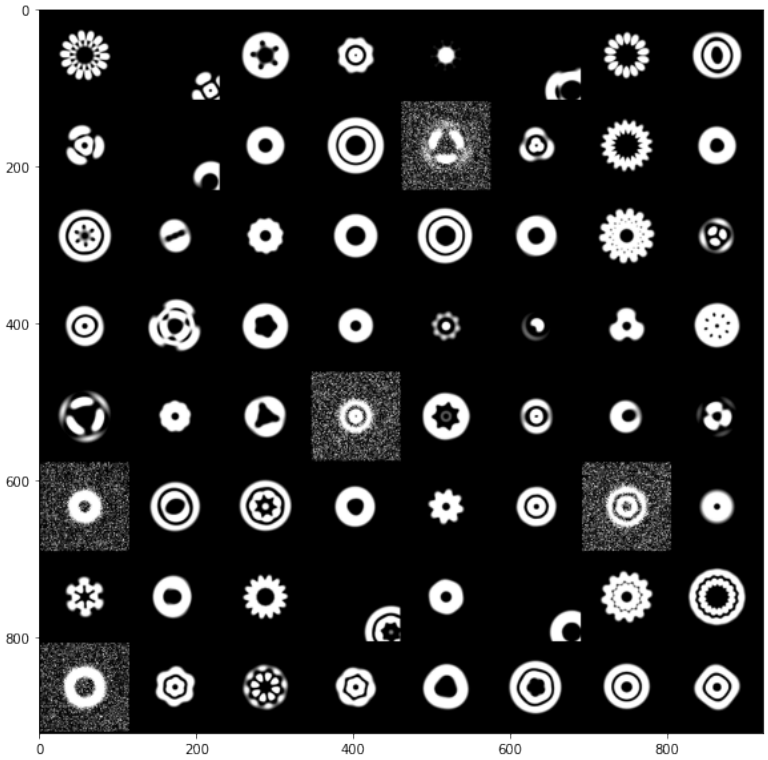

Figure 5.2: With Normalisation.

CHAPTER 6

CNN Architecture

6.1 Convolution Neural Network Theory

Convolution Neural network consists of multiple block of convolution layers, batch normalisation, max pooling, fully connected network. C.N.N has ability to extract features and update the weights through back propagation algorithm.

A tremendous interest has emerged in deep learning in recent years. The most established algorithm used in computer vision in order to classify the images has been C.N.N. Convolution and max pooling layer usually extracts the features from the image and F.C network maps the extracted features to find the output.

In order to train CNN we need lot of data, with is computationally expensive, which results in the use of G.P.U(Graphical precessing units.)

Usually how C.N.N works is image is passed to the convolution neural network, where initial layers like convolution, max-pooling extracts the important features from the image. This features is further feed to F.C layers , where it maps to find the output this process is called forward propagation. By using loss function we will back propagate in order to update the weights such that loss value is minimised.

Convolution for input image is a small matrix of size k is convolved at each pixel of the image, element wise product was carried out and summed to obtain the output value for the corresponding position in output matrix.Typically kernel size is around 3×3, 5×5 and 7×7. The distance between two consecutive kernel positions is called as stride. Usually stride value is kept as 1. The process of training the convolution

network is to identify the kernel value for the given training dataset. So, Number of kernels, kernel size, stride, padding are the hyper parameters.

Activation function defined as where output of the linear function like convolution passed to non linear activation function like Relu, sigmoid , elu,tanh etc.

Maxpooling defined as where from feature maps, a patches are chosen , where the maximum value of that patch is chosen. usually patches are taken as size of 2×2 and stride of 2.

6.2 C.N.N Hyper parameter Tuning using Wand b:

Hyper parameter tuning is requires in Neural network because they control the overall behaviour of the Machine learning model. There are many hyper parameters which was tuned in our architecture, like Learning rate, weight decay, batch size, No of kernel filters, Size of the kernel filters, Optimiser, number of Fully connected layers , loss function , Activation functions, Stochastic Gradient descent momentum value, Drop out value. All the hyper parameters was tuned which would take long time since there are many possible combinations, So it was divide into 3 batches. For each batch best hyper parameter value was chosen thus completing all the hyper parameters mentioned above. Here val accuracy is defined as if the predicted outputs like weights and phase is accurately classified, if mean output from the CNN is less than other equal to 0.001. Below 3 plots of hyper parameter tuning is called as **Parallel Coordinate Graph** each line represents the Trained CNN and for each of this hyper parameters tuning, training dataset was taken as 30000 , 6000 for validation and testing dataset. Same dataset was used for all the batch tuning. Wand-b was used into order tune the hyper parameter of Alex-net.
For batch 01 , Initial hyper parameters are drop-out value = **0.2**, fc1-sz=**4096**, fc2-sz=**4096**, Activation function = **Relu**, loss function = **MAE(Mean absolute error)**.

Batch 01:

1. Max number of epochs = [80,70,60,50,40]

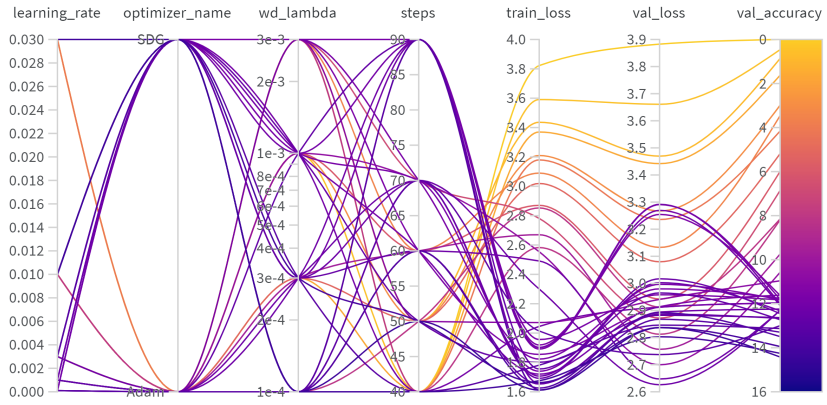

Figure 6.1: Batch 01 hyper parameter tuning..

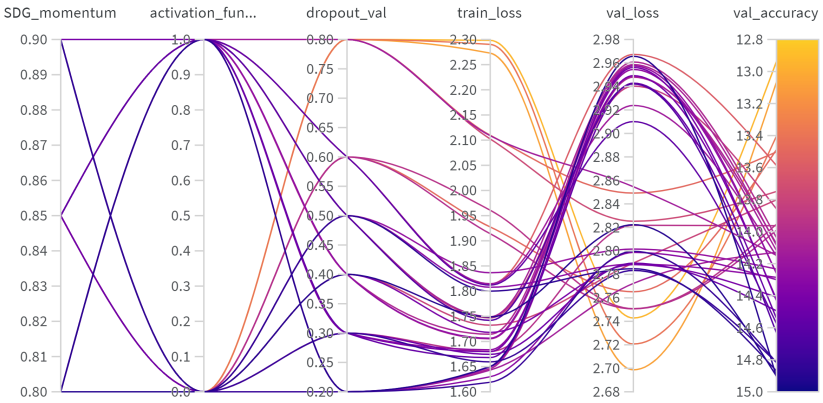

Figure 6.2: Batch 02 hyper parameter tuning..

2. Learning rate = $[10^{-4}, 3 * 10^{-3}, 10^{-3}, 3 * 10^{-2}, 10^{-2}]$
3. weight decay = $[3 * 10^{-4}, 110^{-4}, 3 * 10^{-3}, 1 * 10^{-3}]$
4. optimiser = ['Adam','S.D.G']

From Batch 01 it was found out that optimal one was optimiser = **S.D.G(Stochastic Gradient descent)**, learning rate = $1e-4$, weight decay= 10^{-4}

Batch 02:

1. drop out = [0.2,0.3,0.4,0.5,0.6,0.8]
2. S.D.G momentum = [0.9,0.8,0.85]
3. activation function = [Relu, Elu]

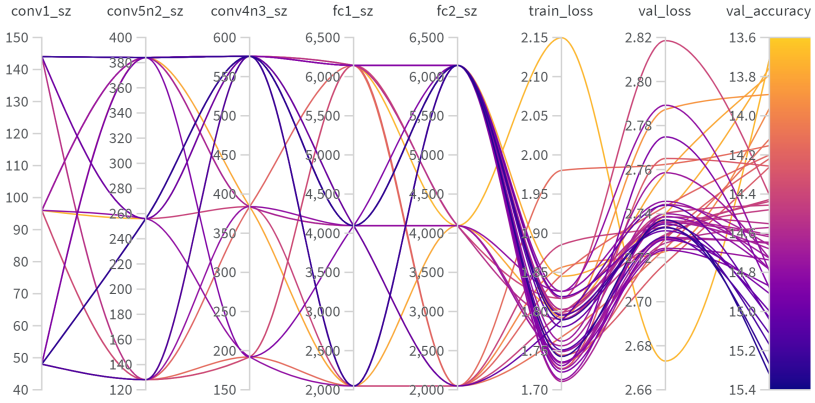

Figure 6.3: Batch 03 hyperparameter tuning..

All the optimal hyper parameters found from the batch 01 was retained, for Batch 02 hyper parameter tuned and optimal values found out as drop out = **0.5**, S.D.G momentum = **0.9**, activation function = **Relu**.

Hyper parameters from batch 01 and batch 02 was kept same but in batch 03, hyper parameter tuning was done for Number of kernels in CNN, No of units in F.C network.

Batch 03:

1. conv1-sz = [48, 96, 144]
2. conv5n2-sz = [128, 256, 384]
3. conv4n3-sz = [192, 384, 576]
4. fc1-sz = [2048, 4096, 6144]
5. fc2-sz = [2048, 4096, 6144]

Final Optimal values of hyper parameters are dropout = **0.5**, fc1-sz = **2048**, fc2-sz = **6144**, activation function = **Relu**, optimizer = **SDG** ,learning rate= 10^{-4},weight decay =10^{-4},SDG momentum = **0.9**, loss function = **MAE** , conv1-sz = **144**, conv5n2-sz = **384**, conv4n3-sz = **576**. The kernel size at each convolution was separately tuned and best was chosen, the values are mentioned below.

6.3 Alex-Net:

Hyper parameters tuning was done, optimal values was found out, here Alex-net was used to predict weights and phase of the composite beam. The weights which was given

```
Net(
  (conv1): Conv2d(1, 144, kernel_size=(5, 5), stride=(2, 2))
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): Conv2d(144, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv3): Conv2d(384, 576, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv4): Conv2d(576, 576, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv5): Conv2d(576, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (fc1): Linear(in_features=13824, out_features=2048, bias=True)
  (fc2): Linear(in_features=2048, out_features=6144, bias=True)
  (fc3): Linear(in_features=6144, out_features=38, bias=True)
  (bn1): BatchNorm2d(144, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (bn2): BatchNorm2d(384, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (bn3): BatchNorm2d(576, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (bn4): BatchNorm2d(576, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (bn5): BatchNorm2d(384, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (fc_bn1): BatchNorm1d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (fc_bn2): BatchNorm1d(6144, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (fc_bn3): BatchNorm1d(38, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (dropout): Dropout(p=0.5, inplace=False)
)
```

Figure 6.4: Tuned Alex-net Architecture.

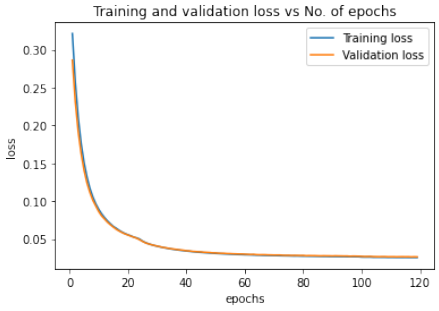

Figure 6.5: Loss curve for weights.

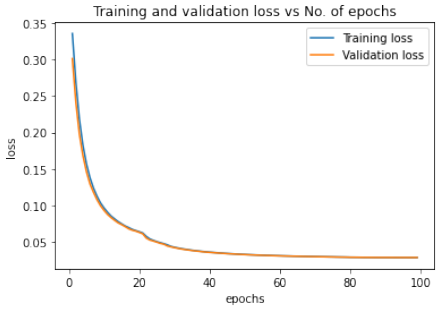

Figure 6.6: Loss curve for phase.

to the CNN is $[a_1, a_2, a_3, ...a_n]$. Since the phase is relative we need to choose a reference so the **Principle Mode** was chosen as reference and 0 value was assigned to the phase value of principle mode. The error in the prediction of the phase is higher than the weights prediction. Two separate Alex-net with same hyper parameters was used to predict weights and phase. Sigmoid activation function is used at the output layer for both weights and phase predictions.

CHAPTER 7

Results and Discussion:

Here we have totally 38 eigen modes for creating composite beam l value ranging from -9 to 9 and p value ranging from 0 to 1. Order of the composite beam was less than or equal to 2, only 2 eigen modes were combined at most to generate the composite beam. This model is able to predict order of composite beam less than or equal to 2. Different value of the noise was introduced in input image in training and testing dataset. Translation value was taken as $t_x = t_y = 45$, yaw and pitch angle = 10, Poisson mean value=0.05 ,pepper noise cut-off value= 0.9, σ_b=1. As we try to increase the No. of eigen modes above 50, Adding of the more noise and increasing the order of the composite beam above two, in all this cases there was increase in error in the prediction of weights and phase output. Further, As the radial charge increases more the $p = 1$ it was found that complex intensity pattern was formed it becomes difficult for CNN to extract features. All the above dataset was trained to VGG-16 Neural Network as well, both of them showed same results. Google colab pro + was used to train the C.N.N, GPU of (Tesla V100-SXM2-16GB). The future scope of this work includes the increasing in the order of the composite beam, Increasing the no. of eigen modes, Increasing the in Radial charges p and increase in the noise level in the image.

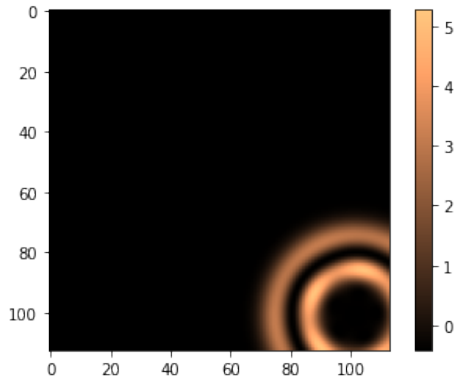

Figure 7.1: Input Composite beam to CNN 1.

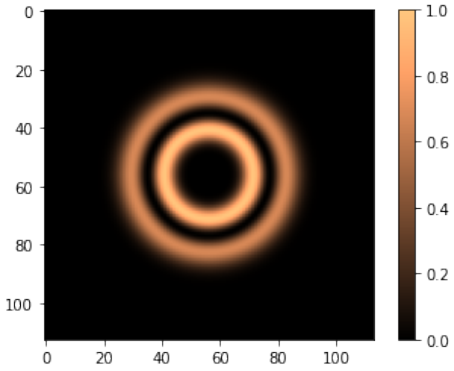

Figure 7.2: Reconstructed image 1.

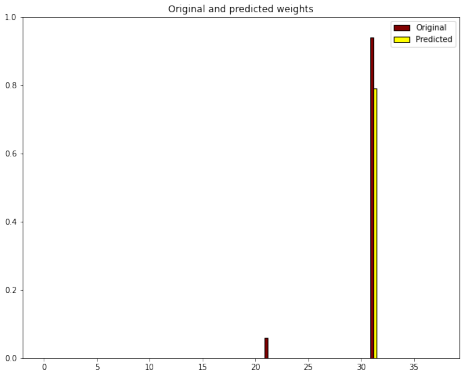

Figure 7.3: Weights prediction 1.

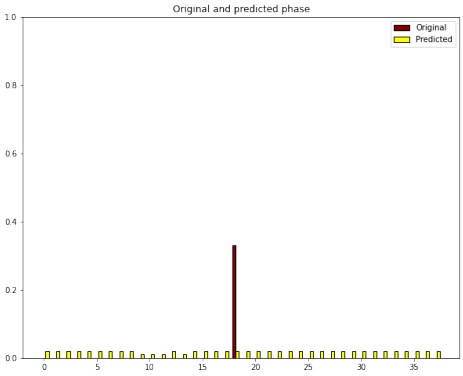

Figure 7.4: Phase prediction 1.

Observation 01: Input Image contains, Noise of Translation of tx=ty=45. Most part of the signal is missing but our CNN is able to retrieve the Image back.

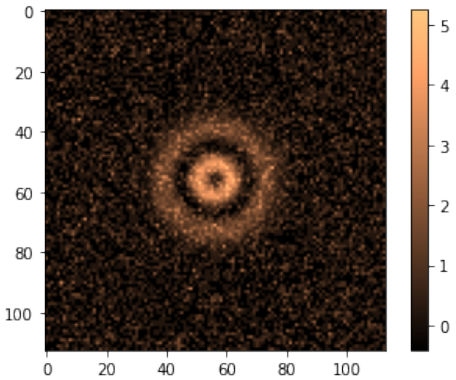

Figure 7.5: Input Composite beam to CNN 2.

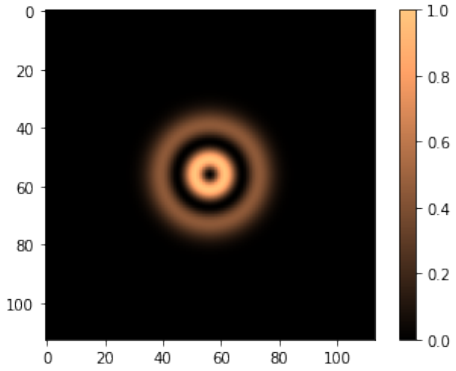

Figure 7.6: Reconstructed image 2.

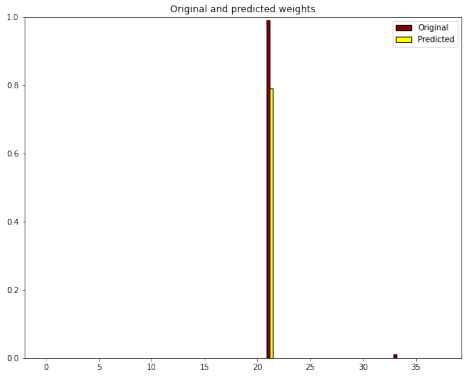

Figure 7.7: Weights prediction 2.

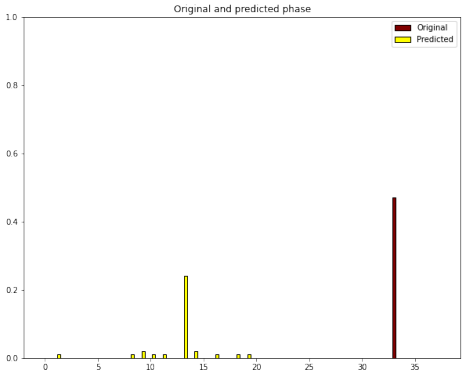

Figure 7.8: Phase prediction 2.

Observation 02: Input Image contains, Shot Noise of Poisson mean value =0.05.

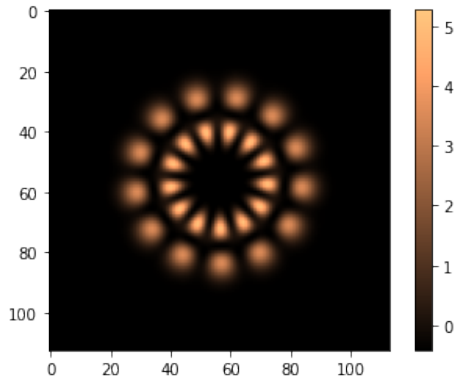

Figure 7.9: Input Composite beam to CNN 3.

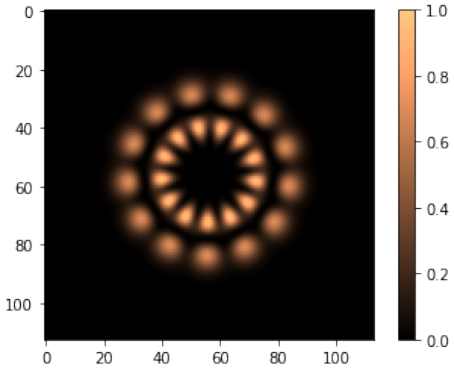

Figure 7.10: Reconstructed Image 3.

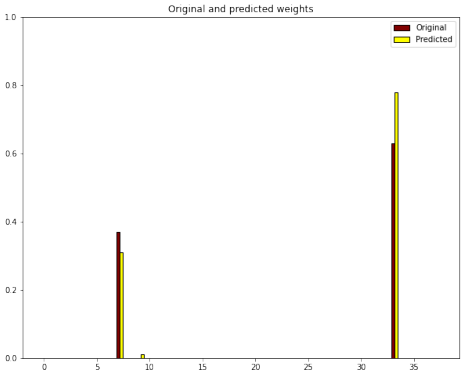

Figure 7.11: Weights Prediction 3.

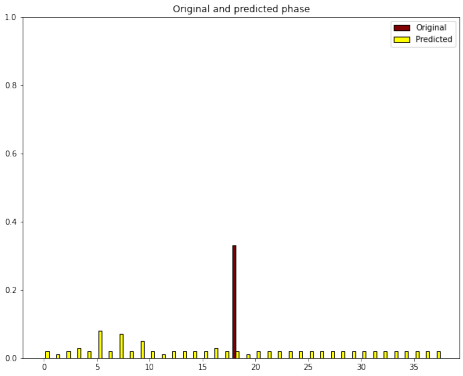

Figure 7.12: Phase Prediction 3.

Observation 03: Input Image contains No Noise.

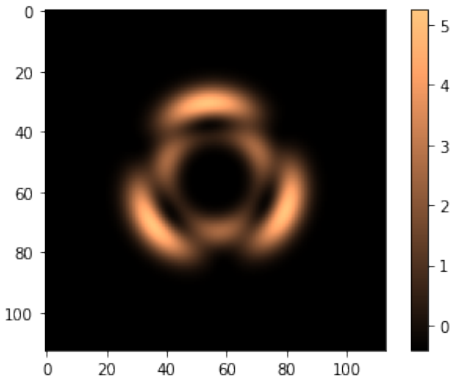

Figure 7.13: Input Composite beam to CNN 4.

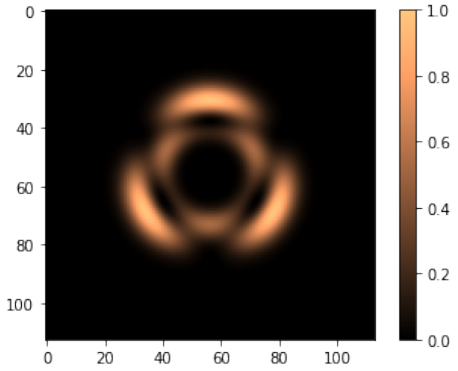

Figure 7.14: Reconstructed Image 4.

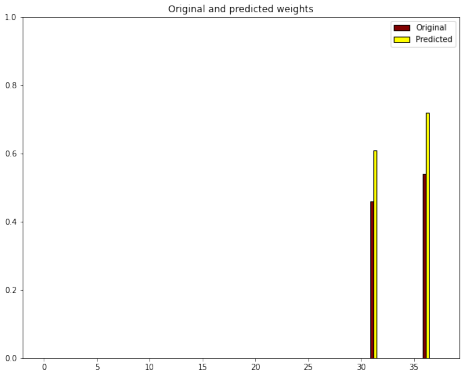

Figure 7.15: Weights Prediction 4.

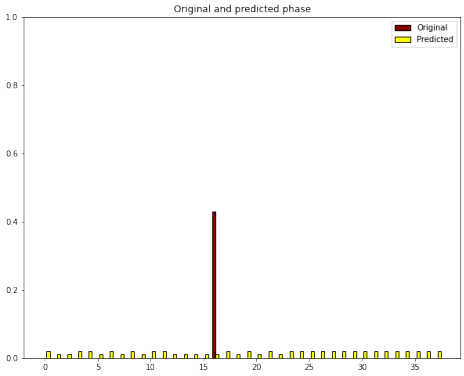

Figure 7.16: Phase Prediction 4.

Observation 04: Input Image contains, Gaussian blur of σ_b=1 , kernel size of 7×7.

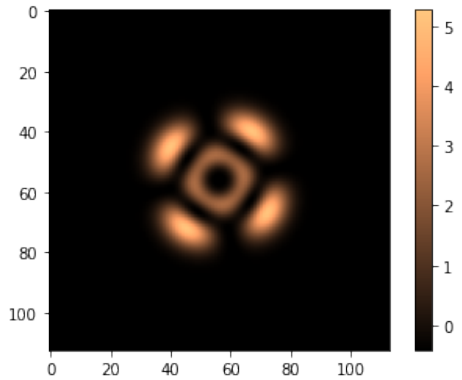

Figure 7.17: Input Composite beam to CNN 5.

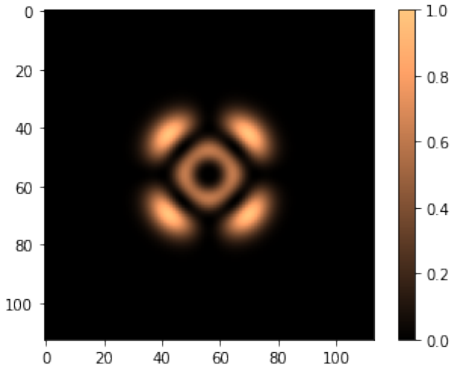

Figure 7.18: Reconstructed Image 5.

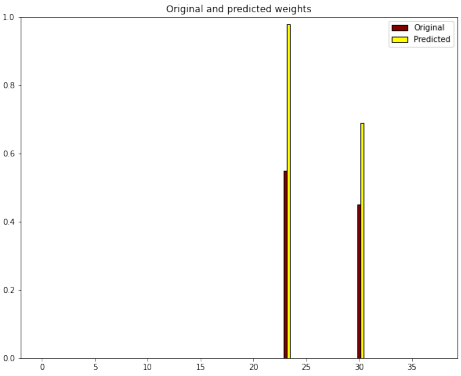

Figure 7.19: Weights Prediction 5.

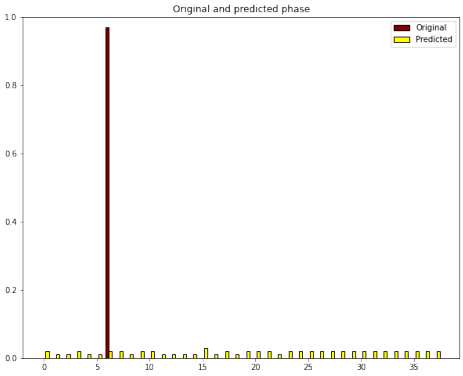

Figure 7.20: Phase Prediction 5.

Observation 05: Input Image contains No Noise.

CHAPTER 8

CODE

8.1 Imports of Libraries:

Listing 8.1: Importing libraries 1

```
import torch, os, os.path as osp
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from tqdm import tqdm
import numpy as np
import cv2
from sklearn.metrics import mean_squared_error
import scipy.stats
from scipy.special import comb, factorial, iv, eval_genlaguerre
```

Listing 8.2: Importing libraries 2

```
pip install photutils
from photutils.datasets import make_noise_image
import numpy as np
import math
from PIL import Image
import imageio as mp
import cv2
import matplotlib
import matplotlib.pyplot as plt
%matplotlib inline

import torch.nn as nn
import torch.nn.functional as F
import torchvision.models as models
from skimage.util import random_noise
```

8.2 Dataset Generation.

8.2.1 Global variables.

Listing 8.3: Global variables values.

```
Nx=113
Ny=113
Max_l=9
Min_l=-9
Max_p=1
modes_set_size=(Max_l-Min_l+1)*(Max_p+1)
operation=["translation","pitch","yaw","shot_noise","pepper_noise","Gaussian_blur"]
operation_index=[0,1,2,3,4,5]
```

8.2.2 Pure Mode Field profile.

Listing 8.4: Code to generate LG beams.

```
def LG_Beams(p = 0,l = 1,w0 = 2e-3,lamda=632.8e-9,z=0.0
             ,s_hsz=5e-3,Nx=113,Ny=113):
  # Setup the cartesian grid for the plot at plane z
  xx, yy = torch.meshgrid(torch.linspace(-s_hsz, s_hsz,Nx)
  , torch.linspace(-s_hsz, s_hsz,Ny));
  C=np.sqrt(2*factorial(p)/(torch.pi*factorial(p+abs(l))))
  # Calculate the cylindrical coordinates
  r_sq = (xx**2 + yy**2);
  phi = torch.arctan2(yy, xx);

  k = 2*torch.pi/lamda;    # Wavenumber of light
  zR = k*w0**2.0/2;        # Calculate the Rayleigh range
  #print(zR)
  A=1;
  w = w0 * np.abs(1.0 + 1j*z/zR);
  zeta = 2*r_sq/w**2
  R_inv= z/(z**2+zR**2)
  #print(R_inv)
  Psi = (2*p + np.abs(l) + 1)*np.arctan(z/zR)
```

```
    return C*A*(w0/w)*(zeta**abs(l/2))*torch.exp(-zeta/2)
    *np.exp(-1j*(k*z+0.5*k*r_sq*R_inv))*np.exp(1j*Psi)
    *eval_genlaguerre(p, abs(l), zeta)*torch.exp(-1j*l*phi)
```

Listing 8.5: Repository of all the pure mode field profiles.

```
LG_Beams_dict=torch.zeros([modes_set_size,Nx,Ny],dtype = torch.complex128)
count=0
for ll in range(Min_l,Max_l+1):
    for r in range(Max_p+1):
        LG_Beams_dict[count]= LG_Beams(p = r,l = ll,w0 = 1e-3)
        count+=1
```

8.2.3 Mode Selection Functions.

Listing 8.6: This code is used to select the random mode indices.

```
def choose_mode_indices(order_lessOrEqual):
    #initializing variables
    composite_beam_mode_index=torch.Tensor()
    #unique mode set not possible
    if (order_lessOrEqual>modes_set_size):
        return composite_beam_mode_index

    order_of_cb=torch.randint(low=2, high=order_lessOrEqual+1
                                    , size=(1,) )
    while(order_of_cb>composite_beam_mode_index.size()[0]):
        composite_beam_mode_index=torch.cat((composite_beam_mode_index
                    ,torch.randint(low=0, high=modes_set_size,
                size=(order_of_cb-composite_beam_mode_index.size()[0],) )))
        composite_beam_mode_index = torch.unique(composite_beam_mode_index)

    return composite_beam_mode_index
```

Listing 8.7: This code is used to assign weigts to the random modes.

```
def assign_mode_weights(order, weight_cutoff=0.005):
  weights=torch.square( torch.rand(order) )
  normalized_weights=weights/torch.sum(weights)
  if len(normalized_weights[normalized_weights<=weight_cutoff])==0:
    return normalized_weights
  else:
    return assign_mode_weights(order, weight_cutoff)
```

Listing 8.8: Finding mode priority using Rule 02

```
def assign_mode_priority(mode_indices, mode_weights, k_value=100):
  l=torch.floor(mode_indices/ (Max_p+1))+Min_l
  p=torch.remainder(mode_indices, Max_p+1)
  return torch.abs(l)-p+(k_value*mode_weights)
```

Listing 8.9: Finding principle mode indices.

```
def find_principle_mode_index(mode_indices, mp, N=1):
    pm_index=np.where(mp == mp.max())[0][0]
    l=np.floor(mode_indices[pm_index]/ (Max_p+1))+Min_l
    if l==0 and N<=mp.shape[0] :
       mp[pm_index]=mp.min()-10
       return find_principle_mode_index(mode_indices, mp, N+1)
    return pm_index
```

Listing 8.10: Correcting mirror charges by Algorithm 2.

```
def correcting_mirror_modes(mode_indices, principle_mode_index):
  l=torch.floor(mode_indices[principle_mode_index]/ (Max_p+1))+Min_l
  if l>=0:
    return mode_indices
  else :
    l=torch.floor(mode_indices/ (Max_p+1))+Min_l
    p=torch.remainder(mode_indices, Max_p+1)
    return ((-l-Min_l)*(Max_p+1)+p).type(torch.int32)
```

Listing 8.11: Assigning Mode phase.

```
def assign_mode_phase(principle_mode_index,order):
  phase=torch.rand(order)
  phase[principle_mode_index]=0
  return phase
```

Listing 8.12: Combining all the above functions to one.

```
def final_mode_indices_weights_and_phase(order_lessOrEqual,k_value):
mode_indices = choose_mode_indices(order_lessOrEqual)
mode_weights = assign_mode_weights(order=mode_indices.size()[0])
mode_priority = assign_mode_priority(mode_indices,mode_weights,k_value)
principle_mode_index = find_principle_mode_index(mode_indices,mode_priority)
mode_indices=correcting_mirror_modes(mode_indices,principle_mode_index)
mode_phase=assign_mode_phase(principle_mode_index,order=mode_indices.size()[0])

return mode_indices,mode_weights.round(decimals=2),mode_phase.round(decimals=2)
```

8.2.4 Noise Adding Functions.

Listing 8.13: Yaw and pitch noise function.

```
def yaw_pitch(max_pitch,max_yaw,image,yaw=0,pitch=0,op=0):
  if yaw == 0 and pitch == 0:
    p_angle=np.random.uniform(0,1)*max_pitch
    y_angle=np.random.uniform(0,1)*max_yaw
  elif op == 1 and (yaw != 0 or pitch != 0):
    p_angle=pitch
    y_angle=0
  elif op == 2 and (yaw != 0 or pitch != 0):
    p_angle=0
    y_angle=yaw

  gamma=0
  rtheta=math.radians(p_angle)
  rphi=math.radians(y_angle)
  rgamma=math.radians(gamma)
```

```
height=image.shape[0]
width=image.shape[1]
d = np.sqrt(height**2 + width**2)
focal = d / (2 * np.sin(rgamma) if np.sin(rgamma) != 0 else 1)
dz=focal

h=image.shape[0]
w=image.shape[1]
f=focal
dx=0
dy=0
# Projection 2D -> 3D matrix
A1 = np.array([[1, 0, -w/2],[0, 1, -h/2],[0, 0, 1],[0, 0, 1]])

# Rotation matrices around the X, Y, and Z axis
RX = np.array([[1, 0, 0, 0],[0, np.cos(rtheta), -np.sin(rtheta), 0]
                ,[0, np.sin(rtheta), np.cos(rtheta), 0],[0, 0, 0, 1]])

RY = np.array([[np.cos(rphi), 0, -np.sin(rphi), 0],[0, 1, 0, 0]
                ,[np.sin(rphi), 0, np.cos(rphi), 0],[0, 0, 0, 1]])

RZ = np.array([[np.cos(rgamma), -np.sin(rgamma), 0, 0],
        [np.sin(rgamma), np.cos(rgamma), 0, 0],[0, 0, 1, 0],[0, 0, 0, 1]])

# Composed rotation matrix with (RX, RY, RZ)
R = np.dot(np.dot(RX, RY), RZ)

# Translation matrix
T = np.array([[1, 0, 0, dx],[0, 1, 0, dy],[0, 0, 1, dz],[0, 0, 0, 1]])

# Projection 3D -> 2D matrix
A2 = np.array([[f, 0, w/2, 0],[0, f, h/2, 0],[0, 0, 1, 0]])

# Final transformation matrix
homograhy= np.dot(A2, np.dot(T, np.dot(R, A1)))
final_imageyp=cv2.warpPerspective(image, homograhy , (width, height))

return final_imageyp
```

Listing 8.14: Gaussian kernel matrix function.

```
#Gaussian kernel matrix function.
def gaussian_kernel(sigma):
  # this code used to find the Gaussian kernel for the given
  # sigma value of size âŇĹ6*sigma+1]*âŇĹ6*sigma+1âŇĽ.
  if sigma==0:
      n=math.ceil(6*sigma+1)
      x=int((n-1)/2)
      hmn=np.array([0])
  else:
      n=math.ceil(6*sigma+1)
      if n%2==0:
          n=n+1
      else:
          n=n
      x=int((n-1)/2)
      hmn=[]
      for a in range(-x,x+1):
        for b in range(-x,x+1):
        fn=((2*math.pi*(sigma**2))**-1)*math.exp(-0.5*((a**2+b**2)/(sigma**2)))
        hmn.append(fn)
      hmn=np.array(hmn)
      hmn=hmn.reshape(n,n)

  return hmn
```

Listing 8.15: Image translation noise function.

```
def image_translation(intensity_profile,max_shifx,max_shify,shifx=0,shify=0):
  if shifx == 0 and shify == 0:
    Trans = np.float32([[1,0,int(np.random.uniform(-1,1)*max_shifx)],
                        [0,1,int(np.random.uniform(-1,1)*max_shify)]])
  else:
    Trans = np.float32([[1,0,shifx],[0,1,shify]])

  return cv2.warpAffine(intensity_profile,Trans,(Ny,Nx))
```

Listing 8.16: Shot noise function.

```
def shot_noise(intensity_profile,max_poisson_mean_value,poisson_mean_value=0):
  if poisson_mean_value == 0:
    pmv = np.random.uniform(0,1)*max_poisson_mean_value
  else:
    pmv = poisson_mean_value
  shot_noise_value=make_noise_image((Nx,Ny), distribution='poisson', mean = pmv)
  shot_noise_value=shot_noise_value/(np.max(shot_noise_value))
  shot_noise_value=shot_noise_value*np.max(intensity_profile)
  return intensity_profile+shot_noise_value
```

Listing 8.17: Gaussian blur noise function.

```
def Gaussian_blur(max_sigma,intensity_profile,sigma=0):
  if sigma ==0 :
    sigma=np.random.randint(low=1, high=max_sigma+1, size=1)
    kernel=gaussian_kernel(sigma)
  else:
    kernel=gaussian_kernel(sigma)
  return cv2.filter2D(src=intensity_profile, ddepth=-1, kernel=kernel)
```

Listing 8.18: pepper noise function.

```
def pepper_noise(intensity_profile,max_pepper_l_value,pepper_l_value=0):
  if pepper_l_value == 0:
    plv = np.random.uniform(0,1)*max_pepper_l_value
  else:
    plv = pepper_l_value

  for i in range(Nx):
    for j in range(Ny):
      probs=np.random.uniform(0,1)
      if probs >= plv:
          intensity_profile[i,j]=0
      else:
          intensity_profile[i,j]=intensity_profile[i,j]

    return intensity_profile
```

Listing 8.19: Code used to add noise to generated image.

```
def add_noise(intensity_profile1 ,max_sigma,max_shifx,max_shify,
      max_pepper_l_value,max_poisson_mean_value,max_pitch_angle,max_yaw_angle,
    no_of_operations_lessOrEqual=7,shifx=0,shify=0,yaw=0,pitch=0,
    poisson_mean_value=0,pepper_l_value=0,sigma=0):
  np.random.shuffle(operation_index)
  intensity_profile = intensity_profile1.numpy()
  operation_indices_values = operation_index[0:int(np.random.randint(low=0,
                                high=no_of_operations_lessOrEqual+1, size=1))]
  operation_indices_values=np.sort(np.array(operation_indices_values))
  for i in operation_indices_values:
    if i==0:
      intensity_profile = image_translation(intensity_profile,max_shifx,
                                            max_shify,shifx,shify)
    elif i==1:
      intensity_profile = yaw_pitch(max_pitch=0,max_yaw=max_yaw_angle,
                            image=intensity_profile,yaw=yaw,pitch=pitch,op=i)
    elif i==2:
      intensity_profile = yaw_pitch(max_pitch=max_pitch_angle,max_yaw=0,
                            image=intensity_profile,yaw=yaw,pitch=pitch,op=i)
    elif i==3:
      intensity_profile = shot_noise(intensity_profile,max_poisson_mean_value,
                                     poisson_mean_value)
    elif i==4:
      intensity_profile = pepper_noise(intensity_profile,max_pepper_l_value,
                                       pepper_l_value)
    elif i==5:
      intensity_profile = Gaussian_blur(max_sigma,intensity_profile,sigma)

  Zero_matrix = np.zeros(intensity_profile.shape)
  intensity_profile = cv2.normalize(intensity_profile, Zero_matrix, 0, 255,
                                    cv2.NORM_MINMAX)
  intensity_profile = intensity_profile.astype(np.uint8)

return torch.tensor(intensity_profile,dtype=torch.float),operation_indices_values
```

8.2.5 Expected Output.

Listing 8.20: Calculating the outputs to CNN.

```
def calculate_expected_output(mode_indices,mode_weights,mode_phase):
    CNN_output_given_weight=torch.zeros(modes_set_size)
    CNN_output_given_phase=torch.zeros(modes_set_size)
    CNN_output_given_weight[mode_indices.numpy()] = mode_weights
    CNN_output_given_phase[mode_indices.numpy()] = mode_phase
    return CNN_output_given_weight,CNN_output_given_phase
```

Listing 8.21: Code used to generate intensity profile.

```
def generate_intensity_profile(mode_indices,mode_weights,mode_phase):
  intensity_profile =  torch.square(
      torch.abs(torch.sum(LG_Beams_dict[mode_indices.numpy()]  *
              ((mode_weights*torch.exp(1j*2*np.pi*mode_phase)).view
               (mode_indices.size()[0],1,1).repeat(1,Nx,Ny))  ,0) ))
  return intensity_profile/intensity_profile.max()
```

8.3 Data-loader for Training and validation Set.

Listing 8.22: Code to find the standard deviation and mean of dataset.

```
def get_mean_and_std(dataloader):
    channels_sum, channels_squared_sum, num_batches = 0, 0, 0
    for data in dataloader:
        # Mean over batch, height and width, but not over the channels
        channels_sum += torch.mean(data[0], dim=[0,2,3])
        channels_squared_sum += torch.mean(data[0]**2, dim=[0,2,3])
        num_batches += 1

    mean = channels_sum / num_batches

    # std = sqrt(E[X^2] - (E[X])^2)
    std = (channels_squared_sum / num_batches - mean ** 2) ** 0.5

    return mean, std
```

Listing 8.23: API to load the saved dataset file.

```
def get_dataloader_from_pth(contents, batch_size=4):
    dataset = torch.utils.data.TensorDataset(contents['x'])
    dataloader = torch.utils.data.DataLoader(dataset, batch_size=batch_size,
                                             shuffle=True, num_workers=2)

    #changed by Nirjhar
    mean,std=get_mean_and_std(dataloader)
    print('mean = ',mean,' std = ',std)
    transform = transforms.Compose([transforms.Normalize([mean],[std])])

    dataset = torch.utils.data.TensorDataset(transform(contents['x']),
                                        contents['w'], contents['p'], contents['m'])
    dataloader = torch.utils.data.DataLoader(dataset, batch_size=batch_size,
                                             shuffle=True, num_workers=2)


    print("{} data in loader".format(contents['x'].size()[0]))
    return dataloader
```

8.3.1 Training dataset.

Listing 8.24: Code used to generate the dataset

```
def db_train(training_dataset_length=200):
  training_weights_expected_output=torch.zeros(
      (training_dataset_length,modes_set_size))
  training_phase_expected_output=torch.zeros(
      (training_dataset_length,modes_set_size))
  training_images=torch.zeros([training_dataset_length,1,Nx,Ny])
  for count in range(training_dataset_length):
    mode_indices,mode_weights,mode_phase=final_mode_indices_weights_and_phase
    (order_lessOrEqual=2,k_value=100)
    img1=generate_intensity_profile(mode_indices,mode_weights,mode_phase)
    training_images[count],operation_indices_values = add_noise
    (img1,max_sigma=1,max_shifx=45,max_shify=45,max_pepper_l_value=0.9,
     max_poisson_mean_value=0.05,max_pitch_angle=10,max_yaw_angle=10,
     no_of_operations_lessOrEqual=1,shifx=45,shify=45,yaw=10,pitch=10,
```

```
        poisson_mean_value=0.9,pepper_1_value=0.05,sigma=1)
    training_weights_expected_output[count],
    training_phase_expected_output[count]=calculate_expected_output
    (mode_indices,mode_weights,mode_phase)
  return {'x': training_images, 'w': torch.sqrt(training_weights_expected_output
                    ),'p':training_phase_expected_output, 'm':
          (training_weights_expected_output>0).type(torch.float)}
```

Listing 8.25: Generating training dataset.

```
trainloader = get_dataloader_from_pth(db_train(1000), batch_size=64)
torch.cuda.empty_cache()
mem_report()
```

8.3.2 Validation dataset.

Listing 8.26: Generating validation dataset.

```
valloader = get_dataloader_from_pth(db_train(10000), batch_size=64)
```

Listing 8.27: This code is used to find out the image seen by the CNN.

```
def imshow(inp, title=None):
    """Imshow for Tensor."""
    inp = inp.numpy().transpose((1, 2, 0))
    # mean = np.array([0.0059])
    # std = np.array([0.0171])
    #inp = std * inp + mean
    #inp = np.clip(inp, 0, 1)
    plt.figure(figsize=(10, 10))
    plt.imshow(inp,cmap='copper')
    if title is not None:
        plt.title(title)
    plt.pause(0.001)  # pause a bit so that plots are updated


# Get a batch of training data
```

```
inputs, labels_w, labels_p,labels_m = next(iter(trainloader))

# Make a grid from batch
out = torchvision.utils.make_grid(inputs)

#imshow(out, title=[class_names[x] for x in labels_w])
title=[np.where(x > 0.005)[0] for x in labels_w]
title=[x[np.where(x > 0.005)[0]] for x in labels_w]
#[print(x[x > 0.005]) for x in labels_w]
#print(title)
imshow(out)
```

8.4 C.N.N Training.

8.4.1 C.N.N Architecture

Listing 8.28: Alexnet Architecture to find the weights.

```
class Net(nn.Module):
    def __init__(self,dropout_val,fc1_sz,fc2_sz,activation_func,conv1_sz,
                 conv5n2_sz,conv4n3_sz):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=1, out_channels= conv1_sz,
                               kernel_size= 5, stride=2, padding=0 )
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2)
        self.conv2 = nn.Conv2d(in_channels=conv1_sz, out_channels=conv5n2_sz,
                               kernel_size=3, stride= 1, padding= 1)
        self.conv3 = nn.Conv2d(in_channels=conv5n2_sz, out_channels=conv4n3_sz,
                               kernel_size=3, stride= 1, padding= 1)
        self.conv4 = nn.Conv2d(in_channels=conv4n3_sz, out_channels=conv4n3_sz,
                               kernel_size=3, stride=1, padding=1)
        self.conv5 = nn.Conv2d(in_channels=conv4n3_sz, out_channels=conv5n2_sz,
                               kernel_size=3, stride=1, padding=1)
        self.fc1   = nn.Linear(in_features= conv5n2_sz*36, out_features= fc1_sz)
        self.fc2   = nn.Linear(in_features= fc1_sz, out_features= fc2_sz)
        self.fc3 = nn.Linear(in_features=fc2_sz , out_features=modes_set_size)
```

```
        self.bn1 = nn.BatchNorm2d(num_features=conv1_sz)
        self.bn2 = nn.BatchNorm2d(num_features=conv5n2_sz)
        self.bn3 = nn.BatchNorm2d(num_features=conv4n3_sz)
        self.bn4 = nn.BatchNorm2d(num_features=conv4n3_sz)
        self.bn5 = nn.BatchNorm2d(num_features=conv5n2_sz)

        self.fc_bn1 = nn.BatchNorm1d(num_features=fc1_sz)
        self.fc_bn2 = nn.BatchNorm1d(num_features=fc2_sz)
        self.fc_bn3 = nn.BatchNorm1d(num_features=modes_set_size)

        # Define proportion or neurons to dropout
        self.dropout = nn.Dropout(dropout_val)
        self.activation_func = activation_func

    def forward(self,x):
        x = self.activation_func(self.bn1(self.conv1(x)))
        x = self.maxpool(x)
        x = self.activation_func(self.bn2(self.conv2(x)))
        x = self.maxpool(x)
        x = self.activation_func(self.bn3(self.conv3(x)))
        x = self.activation_func(self.bn4(self.conv4(x)))
        x = self.activation_func(self.bn5(self.conv5(x)))
        x = self.maxpool(x)
        x = x.reshape(x.shape[0], -1)
        x = F.leaky_relu(self.fc_bn1(self.fc1(x)))
        x = self.dropout(x)
        x = F.leaky_relu(self.fc_bn2(self.fc2(x)))
        x = self.dropout(x)
        x = self.fc_bn3(self.fc3(x))
        x = torch.sigmoid(x)
        return x
```

Listing 8.29: Alexnet Architecture to find the phase.

```
class NetP(nn.Module):
    def __init__(self,dropout_val,fc1_sz,fc2_sz,activation_func,conv1_sz,
                 conv5n2_sz,conv4n3_sz):
        super(NetP, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=1, out_channels= conv1_sz,
                               kernel_size= 5, stride=2, padding=0 )
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2)
        self.conv2 = nn.Conv2d(in_channels=conv1_sz, out_channels=conv5n2_sz,
                               kernel_size=3, stride= 1, padding= 1)
        self.conv3 = nn.Conv2d(in_channels=conv5n2_sz, out_channels=conv4n3_sz,
                               kernel_size=3, stride= 1, padding= 1)
        self.conv4 = nn.Conv2d(in_channels=conv4n3_sz, out_channels=conv4n3_sz,
                               kernel_size=3, stride=1, padding=1)
        self.conv5 = nn.Conv2d(in_channels=conv4n3_sz, out_channels=conv5n2_sz,
                               kernel_size=3, stride=1, padding=1)
        self.fc1  = nn.Linear(in_features= conv5n2_sz*36, out_features= fc1_sz)
        self.fc2  = nn.Linear(in_features= fc1_sz, out_features= fc2_sz)
        self.fc3 = nn.Linear(in_features=fc2_sz , out_features=modes_set_size)

        self.bn1 = nn.BatchNorm2d(num_features=conv1_sz)
        self.bn2 = nn.BatchNorm2d(num_features=conv5n2_sz)
        self.bn3 = nn.BatchNorm2d(num_features=conv4n3_sz)
        self.bn4 = nn.BatchNorm2d(num_features=conv4n3_sz)
        self.bn5 = nn.BatchNorm2d(num_features=conv5n2_sz)

        self.fc_bn1 = nn.BatchNorm1d(num_features=fc1_sz)
        self.fc_bn2 = nn.BatchNorm1d(num_features=fc2_sz)
        self.fc_bn3 = nn.BatchNorm1d(num_features=modes_set_size)

        # Define proportion or neurons to dropout
        self.dropout = nn.Dropout(dropout_val)
        self.activation_func = activation_func

    def forward(self,x):
        x = self.activation_func(self.bn1(self.conv1(x)))
        x = self.maxpool(x)
        x = self.activation_func(self.bn2(self.conv2(x)))
        x = self.maxpool(x)
        x = self.activation_func(self.bn3(self.conv3(x)))
```

```
        x = self.activation_func(self.bn4(self.conv4(x)))
        x = self.activation_func(self.bn5(self.conv5(x)))
        x = self.maxpool(x)
        x = x.reshape(x.shape[0], -1)
        x = self.activation_func(self.fc_bn1(self.fc1(x)))
        x = self.dropout(x)
        x = self.activation_func(self.fc_bn2(self.fc2(x)))
        x = self.dropout(x)
        x = self.fc_bn3(self.fc3(x))
        x=torch.sigmoid(x)
        return x
```

8.4.2 Loss and Optimiser Functions

Listing 8.30: Loss functions for weight.

```
def L1_Loss_w(outputs,labels_w,power_value=None):
    criterion = nn.L1Loss()
    return criterion(outputs, labels_w)
```

Listing 8.31: Loss functions for phase.

```
def L1_Loss_p(outputs,labels_p,power_value=None):
    criterion = nn.L1Loss()
    return criterion(outputs, labels_p)
```

Listing 8.32: Accuracy calculating function.

```
def accuracy_func(outputs,labels_w,power_value=1):
   error = (labels_w - outputs.round(decimals=2)).abs()**power_value
   #is_correct_prediction=(torch.sum(error,1)<0.001)
   is_correct_prediction=(torch.mean(error,1)<=0.002)

   return torch.mean(is_correct_prediction.type(torch.DoubleTensor))
```

8.4.3 Defining Training and validation Functions.

Listing 8.33: Training the network weights.

```
def train(net,epoch, trainloader, optimizer, loss_func,loss_power_value):
    running_loss = 0.0
    for i, data in enumerate(tqdm(trainloader,disable=True), 0):
        # get the inputs
        inputs,labels_w,labels_p,labels_m =data

        if torch.cuda.is_available():
            inputs,labels_w, labels_p,labels_m = inputs.cuda(),labels_w.cuda()
            , labels_p.cuda(),labels_m.cuda()

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = loss_func(outputs,labels_w,loss_power_value)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()

    running_loss = running_loss / (len(trainloader))
    #print('\nepoch %d training loss: %.3f' %
            #(epoch + 1, 1000*running_lossA[epoch]))
    return running_loss
```

Listing 8.34: Code to performs on the val dataset weights.

```
def val(net,epoch, valloader,loss_func,loss_power_value):

    with torch.no_grad():
      running_loss = 0.0
      running_accuracy = 0.0
      for i, data in enumerate(tqdm(valloader,disable=True), 0):
          # get the inputs
```

```
            inputs,labels_w,labels_p  ,labels_m=data

            if torch.cuda.is_available():
                inputs,labels_w, labels_p,labels_m = inputs.cuda(),
                labels_w.cuda(), labels_p.cuda(),labels_m.cuda()

            outputs = net(inputs)
            loss = loss_func(outputs,labels_w,loss_power_value)
            accuracy = accuracy_func(torch.softmax(outputs,dim=1),labels_w)
            # print statistics
            running_loss += loss.item()
            running_accuracy += accuracy.item()

    running_loss = running_loss / (len(valloader))
    running_accuracy = running_accuracy / (len(valloader))
    #print('\nepoch %d validation loss: %.3f' %
            #(epoch + 1, 1000*Vrunning_lossA[epoch]))
    return running_loss,running_accuracy*100
```

Listing 8.35: Training the network Phase.

```
def trainP(net,epoch, trainloader, optimizer, loss_func,loss_power_value):
    running_loss = 0.0
    for i, data in enumerate(tqdm(trainloader,disable=True), 0):
        # get the inputs
        inputs,labels_w,labels_p,labels_m =data

        if torch.cuda.is_available():
            inputs,labels_w, labels_p,labels_m = inputs.cuda(),
            labels_w.cuda(), labels_p.cuda(),labels_m.cuda()

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = loss_func(outputs,labels_p,loss_power_value)
        loss.backward()
        optimizer.step()
```

```
        # print statistics
        running_loss += loss.item()

    running_loss = running_loss / (len(trainloader))
    #print('\nepoch %d training loss: %.3f' %
            #(epoch + 1, 1000*running_lossA[epoch]))
    return running_loss
```

Listing 8.36: Code to performs on the val dataset phase.

```
def valP(net,epoch, valloader,loss_func,loss_power_value):

    with torch.no_grad():
      running_loss = 0.0
      running_accuracy = 0.0
      for i, data in enumerate(tqdm(valloader,disable=True), 0):
          # get the inputs
          inputs,labels_w,labels_p ,labels_m=data

          if torch.cuda.is_available():
              inputs,labels_w, labels_p,labels_m = inputs.cuda(),
              labels_w.cuda(), labels_p.cuda(),labels_m.cuda()

          outputs = net(inputs)
          loss = loss_func(outputs,labels_p,loss_power_value)
          accuracy = accuracy_func(outputs,labels_p)
          # print statistics
          running_loss += loss.item()
          running_accuracy += accuracy.item()

    running_loss = running_loss / (len(valloader))
    running_accuracy = running_accuracy / (len(valloader))
    #print('\nepoch %d validation loss: %.3f' %
            #(epoch + 1, 1000*Vrunning_lossA[epoch]))
    return running_loss,running_accuracy*100
```

8.4.4 Training Network.

Listing 8.37: Training of the Weights.

```
loss_func_list = [L1_Loss_w]
activation_func_list = [F.relu, F.elu]

Max_num_epochs=120
train_lossA= np.empty(Max_num_epochs+1,float)
val_lossA= np.empty(Max_num_epochs+1,float)
val_accuracyA=np.empty(Max_num_epochs+1,float)

def start_training(dropout_val,fc1_sz,fc2_sz,activation_func_code,
                   optimizer_name,learning_rate,wd_lambda,
                   SDG_momentum,loss_func_code,loss_power_value,
                   conv1_sz,conv5n2_sz,conv4n3_sz):


  loss_func = loss_func_list[loss_func_code]
  activation_func = activation_func_list[activation_func_code]

  net = Net(dropout_val,fc1_sz,fc2_sz,activation_func,conv1_sz,
            conv5n2_sz,conv4n3_sz)
  # transfer the model to GPU
  if torch.cuda.is_available():
      net = net.cuda()

  if (optimizer_name == 'Adam'):
    optimizer = optim.Adam(net.parameters(), lr=learning_rate,
                           weight_decay=wd_lambda)
  elif (optimizer_name == 'SDG'):
    optimizer = optim.SGD(net.parameters(), lr=learning_rate,
                          momentum=SDG_momentum, weight_decay=wd_lambda)


  lr_half=2
  for epoch in range(Max_num_epochs+1):
    if epoch >=100:
      if epoch%4==0:
        lr_half=2*lr_half
        optimizer = optim.SGD(net.parameters(), lr=learning_rate/lr_half,
```

```
                                momentum=SDG_momentum, weight_decay=wd_lambda)
    else:
      optimizer = optim.SGD(net.parameters(), lr=learning_rate/lr_half,
                            momentum=SDG_momentum, weight_decay=wd_lambda)

    train_loss = train(net,epoch, trainloader, optimizer, loss_func,
                       loss_power_value)
    val_loss,val_accuracy = val(net,epoch, valloader,loss_func,
                                loss_power_value)
    train_lossA[epoch] = train_loss
    val_lossA[epoch] = val_loss
    val_accuracyA[epoch] = val_accuracy
    wandb.log({"val_accuracy": val_accuracy,"val_loss": val_loss,
               "train_loss": train_loss,"steps":epoch})
    if (epoch % 2 == 0):
      print("\n steps{}: train_loss {}, val_loss {},val_accuracy {} "
      .format(epoch,train_loss,val_loss,val_accuracy))

  return net
```

Listing 8.38: Training of the Phase.

```
loss_func_list = [L1_Loss_p]
activation_func_list = [F.relu, F.elu]

Max_num_epochs=100
train_lossA= np.empty(Max_num_epochs+1,float)
val_lossA= np.empty(Max_num_epochs+1,float)
val_accuracyA=np.empty(Max_num_epochs+1,float)

def start_trainingp(dropout_val,fc1_sz,fc2_sz,activation_func_code,
                    optimizer_name,learning_rate,wd_lambda,SDG_momentum,
                    loss_func_code,loss_power_value,conv1_sz,
                    conv5n2_sz,conv4n3_sz):

  #This fucntion max epochs depends on the learning rate.
  #Max_num_epochs=int(np.round(np.log10(learning_rate),1)*-20+10)
  #Max_num_epochs=40
  loss_func = loss_func_list[loss_func_code]
```

```
activation_func = activation_func_list[activation_func_code]

netp = NetP(dropout_val,fc1_sz,fc2_sz,activation_func,conv1_sz,
            conv5n2_sz,conv4n3_sz)
# transfer the model to GPU
if torch.cuda.is_available():
    netp = netp.cuda()

if (optimizer_name == 'Adam'):
  optimizer = optim.Adam(netp.parameters(), lr=learning_rate,
                         weight_decay=wd_lambda)
elif (optimizer_name == 'SDG'):
  optimizer = optim.SGD(netp.parameters(), lr=learning_rate,
                        momentum=SDG_momentum, weight_decay=wd_lambda)


lr_half=2
for epoch in range(Max_num_epochs+1):
  if epoch >=80:
    if epoch%4==0:
      lr_half=2*lr_half
      optimizer = optim.SGD(netp.parameters(), lr=learning_rate/lr_half,
                            momentum=SDG_momentum, weight_decay=wd_lambda)
  else:
    optimizer = optim.SGD(netp.parameters(), lr=learning_rate/lr_half,
                          momentum=SDG_momentum, weight_decay=wd_lambda)

  train_loss = trainP(netp,epoch, trainloader, optimizer, loss_func,
                      loss_power_value)
  val_loss,val_accuracy = valP(netp,epoch, valloader,loss_func,
                               loss_power_value)
  train_lossA[epoch] = train_loss
  val_lossA[epoch] = val_loss
  val_accuracyA[epoch] = val_accuracy
  #wandb.log({"val_accuracy": val_accuracy,"val_loss": val_loss,
  "train_loss": train_loss,"steps":epoch})
  if (epoch % 2 == 0):
    print("\n steps{}: train_loss {}, val_loss {},val_accuracy {} "
    .format(epoch,train_loss,val_loss,val_accuracy))
```

```
  return netp
```

8.4.5 Wand-b

Listing 8.39: Initialising Wand-b

```
%pip install wandb -q
import wandb
wandb.login()
entity_name="prg"
project_name="Batch1_tuneNoOfK_run2"
```

Listing 8.40: Sweep Configurations.

```
#to add:  batch size,
sweep_config = {
    'method': 'grid', #grid, random, bayes
    'metric': {
      'name': 'val_loss',
      'goal': 'minimize'
    },
    'parameters': {
        'dropout_val': {
            'values': [0.2]
        },
        'fc1_sz': {
            'values': [4096]
        },
        'fc2_sz': {
            'values': [4096]
        },
        'activation_func_code': {
            'values': [0]
        },
        'optimizer_name': {
            'values': ['Adam','SDG']
        },
```

```
        'learning_rate': {
            'values': [1e-4, 3e-3, 1e-3, 3e-2, 1e-2]
        },
        'wd_lambda': {
            'values': [3e-4, 1e-4, 3e-3, 1e-3]
        },
        'SDG_momentum': {
            'values': [0.9]
        },
        'loss_func_code': {
            'values': [0]
        },
        'loss_power_value': {
            'values': [2]
        }
    }
}
sweep_id = wandb.sweep(sweep_config, entity="prg", project="Batch3_tune1")
```

Listing 8.41: Wand b Agent Function.

```
def wandb_agent_function():
    steps = 0
    # Default values for hyper-parameters we're going to sweep over
    config_defaults = {
        'dropout_val': 0.2,
        'fc1_sz': 4096,
        'fc2_sz': 4096,
        'activation_func_code': 0,
        'optimizer_name': "Adam",
        'learning_rate': 1e-3,
        'wd_lambda': 1e-4,
        'SDG_momentum': 0.9,
        'loss_func_code': 0,
        'loss_power_value': 2,
        'conv1_sz': 96,
        'conv5n2_sz': 256,
        'conv4n3_sz': 384
    }
```

```
    # Initialize a new wandb run
    wandb.init(entity=entity_name, project=project_name,config=config_defaults)


    # Config is a variable that holds and saves hyperparameters and inputs
    config = wandb.config
    dropout_val = config.dropout_val
    fc1_sz = config.fc1_sz
    fc2_sz = config.fc2_sz
    activation_func_code = config.activation_func_code
    optimizer_name = config.optimizer_name
    learning_rate = config.learning_rate
    wd_lambda = config.wd_lambda
    SDG_momentum = config.SDG_momentum
    loss_func_code = config.loss_func_code
    loss_power_value = config.loss_power_value
    conv1_sz = config.conv1_sz
    conv5n2_sz = config.conv5n2_sz
    conv4n3_sz = config.conv4n3_sz

    # Model training here
    start_training(dropout_val,fc1_sz,fc2_sz,activation_func_code,
                optimizer_name,learning_rate,wd_lambda,SDG_momentum,
                loss_func_code,loss_power_value,conv1_sz,conv5n2_sz,conv4n3_sz)
```

Listing 8.42: Running Wand b.

```
wandb.agent(sweep_id, wandb_agent_function)
```

LIST OF PAPERS BASED ON THESIS

1. Yuan, X.; Xu, Y.; Zhao, R.; Hong, X.; Lu, R.; Feng, X.; Chen, Y.; Zou, J.; Zhang, C.; Qin, Y.; Zhu, Y. Dual-Output Mode Analysis of Multimode Laguerre-Gaussian Beams via Deep Learning. Optics 2021, 2, 87-95.

2. Z. Wang et al., "Efficient Recognition of the Propagated Orbital Angular Momentum Modes in Turbulences With the Convolutional Neural Network," in IEEE Photonics Journal, vol. 11, no. 3, pp. 1-14, June 2019, Art no. 7903614, doi: 10.1109/JPHOT.2019.2916207.

3. Orbital angular momentum: origins, behavior and applications Alison M. Yao1 and Miles J. Padgett2

4. Convolutional neural networks: an overview and application in radiology Rikiya Yamashita1,2 ; Mizuho Nishio1,3 ; Richard Kinh Gian Do2 ; Kaori Togashi1

5. Timothy Doster and Abbie T. Watnik, "Machine learning approach to OAM beam demultiplexing via convolutional neural networks," Appl. Opt. 56, 3386-3396 (2017)

6. B. S. Freitas, C. J. R. Runge, J. Portugheis, I. de Oliveira and U. Dias, "Optimized OAM Laguerre-Gauss Alphabets for Demodulation using Machine Learning," 2020 IEEE 8th International Conference on Photonics (ICP), 2020, pp. 24-25, doi: 10.1109/ICP46580.2020.9206470.

7. Chenda Lu, Qinghua Tian, Xiangjun Xin, Bo Liu, Qi Zhang, Yongjun Wang, Feng Tian, Leijing Yang, and Ran Gao, "Jointly recognizing OAM mode and compensating wavefront distortion using one convolutional neural network," Opt. Express 28, 37936-37945 (2020).

8. W. Xiong et al., "Convolutional Neural Network Assisted Optical Orbital Angular Momentum Identification of Vortex Beams," in IEEE Access, vol. 8, pp. 193801-193812, 2020, doi: 10.1109/ACCESS.2020.3029139.

9. Identifying orbital angular momentum modes in turbulence with high accuracy via machine learning RiDong Sun1, Lixin Guo1, Mingjian Cheng1,2, Jiangting Li1 and Xu Yan1,Published 12 June 2019