NOISE ROBUST REAL-TIME FOCUS DETECTION WITH DEEP LEARNING FOR ULTRA-FAST LASER MICROMACHINING

by

Can Polat

B.S., Physics Engineering, Hacettepe University, 2019

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Physics

Boğaziçi University

2022

ACKNOWLEDGEMENTS

I would like to start by thanking Assistant Professor Parviz Elahi, who was my supervisor during this research. His guidance and expertise enabled me to establish both confidence and the scientific thinking required to make this research reality. I cannot thank him enough for his contributions to my scientific development.

Furthermore, I would like to thank Gizem Nuran Yapıcı, Sepehr Elahi, and Aysu Ay for their fellowships and collaborations during this research. I thank my dear friend Esat Erdem Eygi for his fellowship and Dr. Onur Külçe for his advice and guidance. Additionally, I sincerely thank all of the optical-and-optomechanical design team of Aselsan A.Ş. and the hardware design team of Arçelik Electronics Plant for their support and friendship.

Last but not least, I would like to thank my beloved sister Canan, my mother Nazmiye, my father Abdullah, my brother Ahmet Eren Sırmacı, and my love Elif Nisa Güler for their unyielding and unconditional support during this process and the beyond.

ABSTRACT

NOISE ROBUST REAL-TIME FOCUS DETECTION WITH DEEP LEARNING FOR ULTRA-FAST LASER MICROMACHINING

In this thesis, different types of machine learning models are provided for ultra-fast laser micromachining system to actively control the focusing of light on the processing material by detecting the reflected light. These different types of models are tested for both experimental and theoretical approaches. For the experimental approach, four different machine learning models are explored. These models were tested for mirror, silicon, steel, and copper samples. The proposed machine learning models offer real-time control with over 90% accuracy. For the simulation, noise at the material surface and the detection system are considered. The noise simulation, including the laser micromachining system, is done using Fourier optics and signal processing. Noise levels at the material surface are determined by laser scanning microscope measurements of experimental samples, and the commercial detection camera noises are considered for the detection noise. Convolutional neural network models are used for focus control in the simulation. Depending on the noise level, the proposed model achieves above 95% accuracy.

ÖZET

ULTRA-HIZLI LASER MİKROİŞLEME İÇİN DERİN ÖĞRENMEYE DAYALI GÜRÜLTÜYE DAYANIKLI GERÇEK ZAMANLI ODAK BELİRLEME SİSTEMİ

Bu tezde, ultra-hızlı laser kaynağının ürettiği laser ışınınını materyal üzerinden yansıması dedekte edilerek ışının ilgili malzemenin üzerine doğru bir şekilde odaklanabilmesi için birden fazla makine öğrenmesi metodu geliştirilmiştir. Geliştirilen bu modeller hem deneysel hem de kuramsal olarak test edilmiştir. Deneysel olarak ayna, silikon, bakır ve demir örnekleri kullanılıp geliştirilen modeller teste tabii tutulmuştur. Önerilen bu modeller anlık kontrol sağlayıp %90 üzerinde tahmin başarısına sahiptir. Kuramsal çalışma için hem materyal yüzeyinde olan gürültü hem de dedektör sistemi üzerinde olan gürültü değerlendirmeye tutulmuştur. Kuramsal çalışmada bulunan gürültülerin simülasyonu için Fourier optiği ve sinyal işleme teknikleri kullanılmıştır. Materyal üzerindeki gürültü seviyelerini belirlemek için lazer taramalı mikroskop kullanılıp, dedektör sistemi üzerindeki gürültü için de piyasadaki dedektör gürültüleri baz alınmıştır. Evrişimsel Sinir Ağları modeliyle odak kontrolü yapılmıştır. Bu kontrol, gürültü seviyesine göre %95 üzerinde tahmin başarısına sahiptir.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

D_L	Distance from the beam splitter to the detection lens
D_Z	Distance from the beam splitter to the focusing lens
f_1	Focusing lens focal length
f_2	Detection lens focal length
$f/\#$	F-number of an optical system
L_1	Focusing lens
L_2	Detection lens
Z_R	Rayleigh length
λ	Wavelength
W_0	Half of the beam waist

LIST OF ACRONYMS/ABBREVIATIONS

1D	One Dimension
2D	Two Dimensions
AVG	Average
AI	Artificial Intelligence
BB	Beam Blocker
BS	Beam Splitter
CNN	Convolutional Neural Network
CPU	Central Processing Unit
D	Diameter of the Beam
FFT	Fast Fourier Transformation
FOV	Field of View
LSM	Laser Scanning Microscope
LSTM	Long Short-Term Memory
ML	Machine Learning
NA	Numerical Aperture
ROI	Region of Interest
SLR	Single-lens Reflex
STD	Standard Deviation
USB	Universal Serial Bus

1. INTRODUCTION

Lasers have found many different places since the time of their invention. These places include many different applications such as cutting [2], cleaning [3], welding [4], engraving [5], ablation [6], and material processing [7]. It is being applied to many materials in material processing, and the applications span from automotive to medical industries [8].

All these possible applications and high practicality come with a price, which is in the form of focusing the laser light onto the processing material. When an unfocused beam is sent onto the material, there can be two possible results. The first is that the light does not affect the sample at all, and the second, which is highly unwanted, damages the sample. Both of these situations would cause excess time and money for the user. Moreover, properly focusing the light onto the sample is no ordinary problem to solve. This problem is mainly due to the non-linear interaction of light and atoms.

Previously, many different solutions have been offered regarding this crucial problem. These approaches include scanning the distance of the material from the focusing lens [9], a lens group that is cylindrical [10], or with a contrast-detection algorithm [11].

At first, these existing methods for focus detection might seem enough; however, they are limited by different parameters of their system. The limitations can be in the resolution of the detection camera, algorithms, the non-linear interaction between the light and atom, the surface roughness of the processed material, and the aberrations in the optical system. Different approaches can solve some problems, like a camera with higher resolution or low aberration optical elements. Nonetheless, this would increase the cost of the system and will turn the system specifications only to the processing of a specific material. On the other hand, most of these approaches do not offer real-time control, and not having real-time control would slow the processing, which would turn into low production.

Recently, Xu et al. [12] used a machine-vision-based approach for focus detection. This approach records the light reflected from the sample. The reflected light has different patterns depending on the position of the sample. These different positions are the result of the diffraction patterns. This approach offers a Gaussian curve fitting method for detecting proper focus. Their method relies on an objective with a high numerical aperture and does not offer real-time system control.

Furthermore, the effect of surface roughness is being ignored. Since every material has its own surface topology, one must consider the surface roughness of the material for precise focusing. In addition, researchers also ignore the effect of detection camera noise. These noise effects would cause a lousy accuracy and lower the precision of the micromachining. This low precision again would have turned into a time and money-consuming effect.

In this thesis, the power of machine learning is being used for focus detection. As they are being used in many different areas of natural science such as biology [13], condensed-matter physics [14], and microscopy [15]. Convolutional neural networks (CNN) [16] have shown huge success in classification and prediction problem ranging from object recognition [17] to agriculture [18]. They also have found place in focus detection [19–23]. CNNs are already used for digital microscopy, cameras, and holography and are optimized for accuracy and speed.

With a machine learning approach, this thesis goes beyond what is achieved in [12] and offers a real-time focus detection approach for a sample with a rough surface and a detection camera with noise. The approach presented in this thesis also excels in terms of cost with a low numerical aperture aspherical lens with a long focal length and a simple USB camera attached to a cheap computer.

First, this thesis will discuss previously done focusing methods for different applications. This will be followed by explaining the experimental setup used in this thesis, data acquisition, and surface roughness measurements using a laser scanning microscope (LSM). Then will progress onto a simulation of a similar laser machining

setup with surface roughness and detection camera noise. Lastly, this work will explain the machine learning models used in this thesis and give details of machine learning models, structure, and training methods. After this step, work results will be shared and concluded.

The codes required for the calculation of LSM standard deviations (std) and averages (avg) are given in Appendix A. The code required for machine learning models applied to the experimental data is given in Appendix B and the codes required for a similar laser micromachining setup and the related machine learning model are given in Appendix C and Appendix D, respectively.

The experimental part, experimental setup specifications in Table 3.1, machine learning model given in Table 5.1, machine learning results of experimental data, which is given in Table 5.2, experimentally observed diffraction patterns for different materials in Figure 3.4, and the confusion matrices in Figure 5.3 are adapted and used from a work [1] previously published by the author of this thesis as the first author. This re-usage is satisfied by the guidance of the publisher's reproducibility and re-usage of the author's work. Further information about the re-usage can be obtained from the publisher's website. These figures and tables are cited accordingly.

2. FOCUS DETECTION

In this chapter, conventional focus detection methods will be discussed with a few examples then it will be followed by the focus detection experiment done in this thesis.

2.1. Focus Detection and Control

Focus detection is not a unique problem for laser material processing alone. Usually, whenever there is light and its detection, a precise focus detection and correction system is required.

One of the most fundamental imaging systems is the eye. Light comes towards the eye and gets focused on photoreceptors by a lens. This lens's focal length can be changed by the movements of muscles in the eye. Therefore different light source positions can be focused on with the eye. The scene or image will be blurred if the light is not focused well enough. If the muscle movements of the eyes are not enough for this correction of blurring, one can use glasses to help the lens in the eye to focus better.

However, it might seem easy to solve when compared to the eye but having a focal length changing lens and a high processing power (the brain) is not cheap and easy to get. So alternate and similar approaches are needed.

2.1.1. Mechanical Focus Control

Most of the current optical systems used in mobile phones or single-lens reflex (SLR) cameras are zoom-based [24] systems. This is due to the need for different fields of view in a constant aperture. Zoom-based camera systems usually consists of three or four elements. Objective group, variator, compensator, and a relay group. Objective groups are usually fixed in position; however, variator and compensator groups move

and cause the change of the system's focal length. This change of the focal length affects the F-number ($f/\#$) of the system and therefore changes the field of view. A schematic of a zoom-based optical system is given in Figure 2.1.

F-number of an optical system can be given as

$$f/\# = \frac{f_s}{EPD} \tag{2.1}$$

where f_s is the system's focal length and EPD is the entrance pupil diameter. When the focal length increases for a constant EPD, $f/\#$ also increases, decreasing the field of view and vice versa.

In Figure 2.1, the system's focal length changes if the variator changes its position. This change would cause the focus on the detector to be changed, and one would observe a blurred image. In order to correct this blurring, the compensator moves as well. This movement of the two elements can be done with either an electric motor or by the hand of the user.

In addition, there is also defocusing on the detector by the thermal expansions and the tolerancing of the optical elements. Another group, the relay group, can solve this blurring. In modern systems, relay groups can be moved like the compensator and variator groups. The movement is usually done with the help of an electrical motor. All of the changes in FOV are done by the meaning of mechanical control. The moving lens groups have predetermined positions and are moved into these positions depending on the FOV setting. However, if one increases the configurations on the optical system, this zoom-based system can be called a continuous zoom-based system. However, this time the precise location of the lenses cannot be easily predetermined due to low step sizes and the high number of configurations. Therefore, the help of software is needed for precise control.

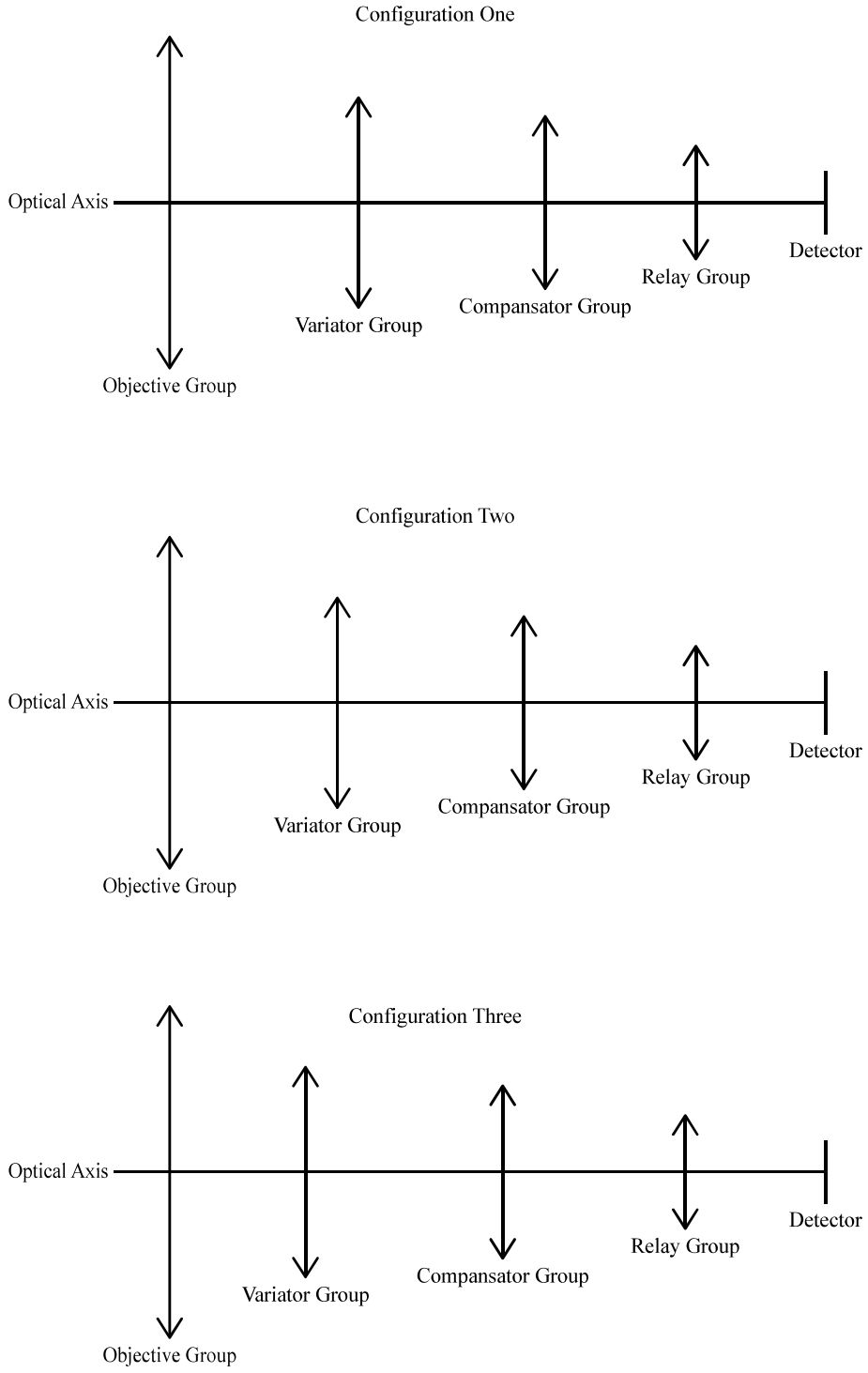

Figure 2.1. Three different focal length configurations are given. While the objective group stays still, the variator group changes the system's focal length. The compensator group corrects the blurring caused by the change of focal length, and the relay group corrects tolerancing and thermal effects.

2.1.2. Contrast Based Focus Controlling

One of the most common ways to control focus in continuous zoom cameras is contrast-based focus controlling. In this method, an algorithm tries to find the correct positions of the moving lenses. This type of algorithm is based on a maximizing of a focus measure [25] for a given region of interest (ROI).

Let $f(x, y)$ be the grayscale at pixel (x, y) of an image with the size M $\times$ N. The squared gradient focus measure, $\Phi(p)$, for an image detected can be given as

$$\Phi(p) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-2} [f(x, y+1) - f(x, y)]^2 \tag{2.2}$$

where the lens position is p. Depending on the image taken, this focus measure would have different values. If the scene is in-focus the value would be high since the pixels have a high difference between their values, i.e., sharper image. But in a blurred scene, pixels would have values close to each other so that the $\Phi(p)$ would be lower. A scene with different $\Phi(p)$ values can be seen in Figure 2.2.

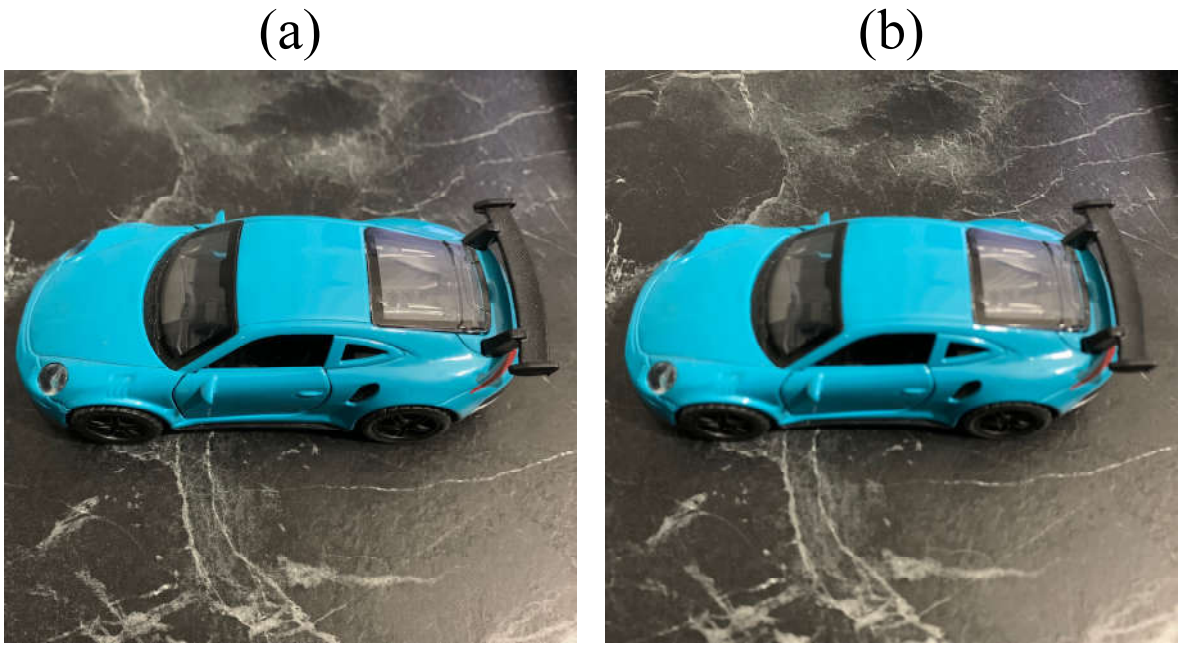

Figure 2.2. Scenes with two different focus measures. (a) When the car is focused properly. (b) is when the car is blurred.

There is a point that needs to be mentioned here. The ROI is also a problem to

be determined. Focusing is done depending on ROI. For example, in Figure 2.2, ROI is the car. This ROI can be the whole scene or can be determined manually, just like tapping on the object on a smartphone screen.

2.1.3. Distance Measurement for Focus Detection

Another possible method to determine the focus is by measuring the distance of the object which needs to be focused. A simple range-finder can be used to determine this distance. However, using other hardware makes the system more expensive and open to threats from outside, such as getting damaged.

Range-finders benefit from time-of-flight [26] measurements. When the range-finder is pointed to a target at a distance, the source beam produced by the range-finder hits the target and returns to the range-finder. By calculating the time of flight of the source beam, the distance to the targets can be found. A schematic of this method can be found in Figure 2.3.

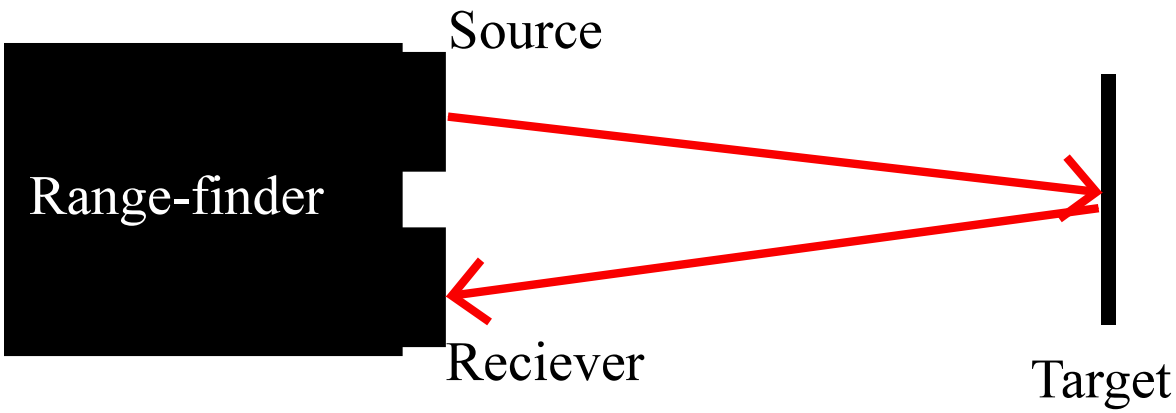

Figure 2.3. Range-finder schematic. A source beam goes to a target and reflects from it. Once the receiver detects the source beam, the flight time can be calculated.

This application can help eliminate the processing time of contrast-based algorithms and yield precise focusing distance to move the lenses. Nonetheless, when the object is not flat, the system performs poorly in focusing the object properly. The car in Figure 2.2 is not a smooth object, but if the ROI is the platform on which the car

stays, the range-finder is a good but expensive choice to use.

2.1.4. Convolutional Neural Networks for Focus Detection

After the massive success of machine learning in most scientific fields, they have also started to be used for focus detection and control. Depending on the nature of the application, different models are proposed for detection and control of focus.

As discussed in previous chapters, for an optical system with a high number of FOVs, a contrast-based algorithm was used to determine the position of lenses in order to have the correct focus of the scene. However, as mentioned, this method also has problems such as computation requirements and the fact that continuously moving lenses for focusing and computing for each position is time-consuming and confusing.

In order to solve these problems, a deep learning approach [27] has been offered. In this approach, for each position of the lens, an ROI was captured for a focused position. Once this scene capturing for different lens locations is done, data is fed to a convolutional neural network model. The model consists of six convolution layers and long short-term memory (LSTM) [28] layer. This model then predicts by looking at a scene to determine the step size of the lens to have the ROI in focus.

With this method, both computational requirements, extra hardware, and time-consuming problems are solved. Furthermore, this approach is applied to a whole scene instead of just an ROI. Details of both machine learning and its tool are given in Chapter 5.

The approaches mentioned above are merely the tip of the iceberg for focus detection. Many approaches are specific to a single problem and cannot be easily generalized. However, the machine learning models can be applied to many problems with minor changes. Therefore, applying these models to many different problems is convenient and goes beyond what has been achieved with conventional models.

3. LASER MICROMACHINING EXPERIMENT

This chapter will start by explaining the experimental setup. Once this explanation is done, it will be followed by the data acquisition method and LSM measurements done for the surface roughness std and avg values of the samples.

3.1. Experimental Setup

The experimental setup consists of many different optical elements. A laser source, a 50/50 beam splitter (BS), two lenses with different focal lengths, a beam blocker (BB), an adjustable stage, a sample holder, samples, and a detection camera. The experimental setup is illustrated in Figure 3.1. An overview of the element's specifications can be found in Table 3.1. A picture of the optical setup consisting of elements in the experiment with a mirror as the sample is given in Figure 3.2 and a version of the sample holder with copper, silicon, and steel is given in Figure 3.3.

Table 3.1. Important parameters of the laser and the optical elements in the experiment. Adapted from [1].

Element	**Specification**
Laser source	0.976 µm
Beam diameter (D)	1 mm
Beam waist	6.84 µm
Beam splitter	50/50 at 45 degrees
Focusing lens focal length	11 mm
Detection lens focal length	35 mm
Detection camera pixel size	3.6 µm

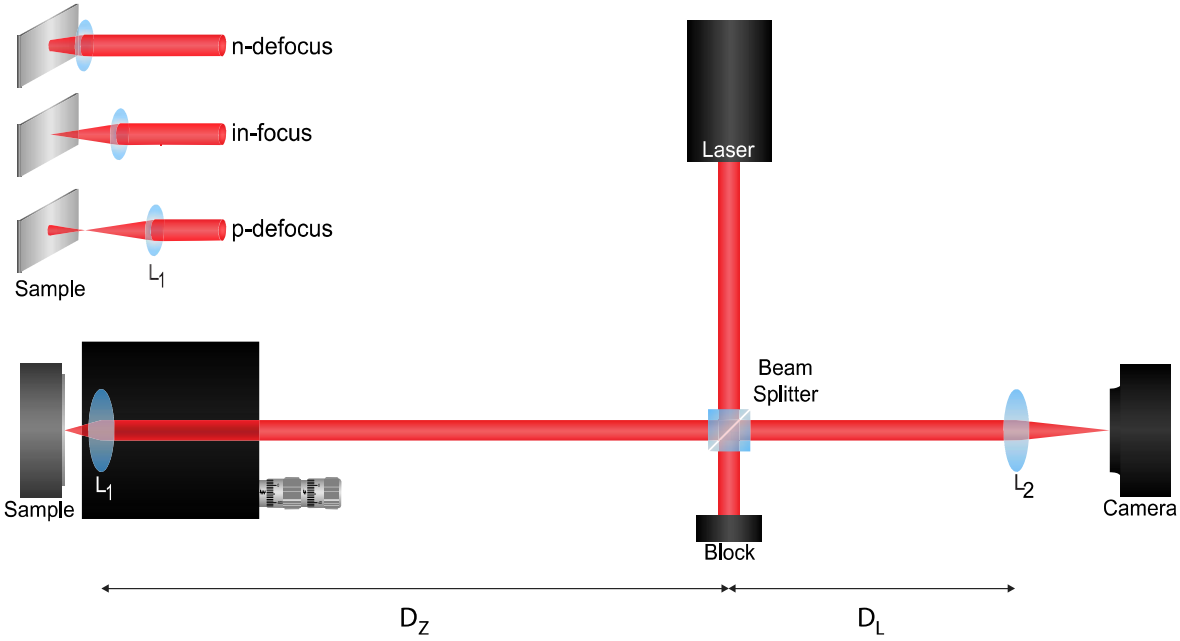

Figure 3.1. Experimental setup illustration consisting of a 0.976 µm laser, a non-polarizing 50/50 beam splitter, an aspheric lens with $f_1 = 11$ mm (denoted as L_1) on a stage with a resolution of 1 µm, a sample, a plano-convex lens with $f_2 = 35$ mm (denoted as L_2), and a USB camera. The distance from the beam splitter to L_1 is given as D_Z and to L_2 is given as D_L. The incoming beam to L_1 and the names of the corresponding defocus positions are given in the upper left image with an exaggeration of the distance.

As it is a must in optical setups, the alignment of the elements is also essential for laser micromachining, which is as follows:

- First, the BS should be aligned with the laser source. This alignment can be done by putting a mirror in the place of the sample holder. The mirror will reflect the light. The reflected light at the place of the detection camera should be in the same direction with equal intensity compared to the incoming one. Once the beams are equal, this would ensure the BS is at a 45-degree angle with respect to the laser source and operating correctly.
- Second, L_1 can be put in its place and match the incoming and reflected beam by fine-tuning the distance between the mirror and the focusing lens.
- Third, L_1 and the camera can be put in their place. Fine-tuning of the distance can be done until the highest intensity to area ratio beam can be observed.

- Lastly, the mirror can be changed with the sample when the steps are done correctly.
- Therefore, the machining setup is well aligned with the sample in-focus position.

Now that the alignment is done, one can trace the path of the light until it reaches the detection camera as follows:

- A Laser source with a 976 nm wavelength emits a coherent light towards a 50/50 beam splitter.
- The BS splits the light in two. One goes to the beam blocker, and the other propagates to the focusing lens L_1.
- L_1 focuses the light onto sample.
- A part of the light gets reflected by the sample and returns to L_1.
- This time, L_1 collimates the reflected light and passes it over to BS.
- The BS again splits the beam. One goes back to the source, and another propagates toward detection lens L_2.
- L_2 focuses the beam onto the detection camera.

If the sample is in-focus position, which is when the sample is at the focus position of L_1, a Gaussian-shaped beam can be observed with the highest intensity to area ratio. This beam has the shape of Gaussian since the laser is a Gaussian source. Co-centric diffraction rings can be observed if the sample is between L_1's focus position and the lens itself. This position will be called n-defocus. Furthermore, suppose the sample is away from the focus position of L_1 in the opposite direction of L_1, a Gaussian spread beam with lower intensity to area ratio is observed. This position will be named p-defocus.

Once the laser micromachining system is properly aligned and the expected defocus positions are observed, data acquisition can be made in order to train and test the machine learning models.

3.1.1. Data Acquisition

Acquisition of the experimental data in order to train and test proposed machine learning models consists of a few critical parameters. The first is being Rayleigh length [29] of the laser micromachining setup. This length is an essential parameter since it defines the depth of focus. Therefore, the range for data acquisition will be based on the experimental setup's Rayleigh length.

Rayleigh length can be calculated using the specifications of the laser source and L_1. The Rayleigh length calculation starts by calculating the beam waist. Beam waist for a focused light by a lens can be given as

$$2W_0 = \frac{4\lambda}{\pi}\frac{f_1}{D} \tag{3.1}$$

where $2W_0$ is the beam waist, and D is the diameter of the beam. Given that $2W_0$ is the beam waist, Rayleigh length Z_R can be obtained as

$$Z_R = \frac{\pi W_0^2}{\lambda}. \tag{3.2}$$

Putting the values given in Table 3.1, the Rayleigh length of the designed micromachining system in this work can be calculated as 150 µm.

Now that the Rayleigh length for the experimental setup is found, data acquisition can start. In this work, 10 µm step sizes are chosen in the both n-defocus and p-defocus directions in the range of ±150 µm. This step size corresponds to about 7% of the Rayleigh length which is enough for having a comparable resolution for the machining. If the step size is too small, the detected image cannot be distinguished from a consecutive position due to random noise in the system. This random noise in the experiment will be discussed in the following chapter.

Thanks to the alignment process, the laser starts at the in-focus position when the images are wanted to be taken. For the training of machine learning models, 4000 frames videos are taken for each sample with defocus ranging from -150 µm to +150 µm with an increment of 10 µm. This would correspond to having 31 videos for each sample. Experimentally detected defocus at ±150 µm and in-focus positions for the

mirror, silicon, steel, and copper can be found in Figure 3.4.

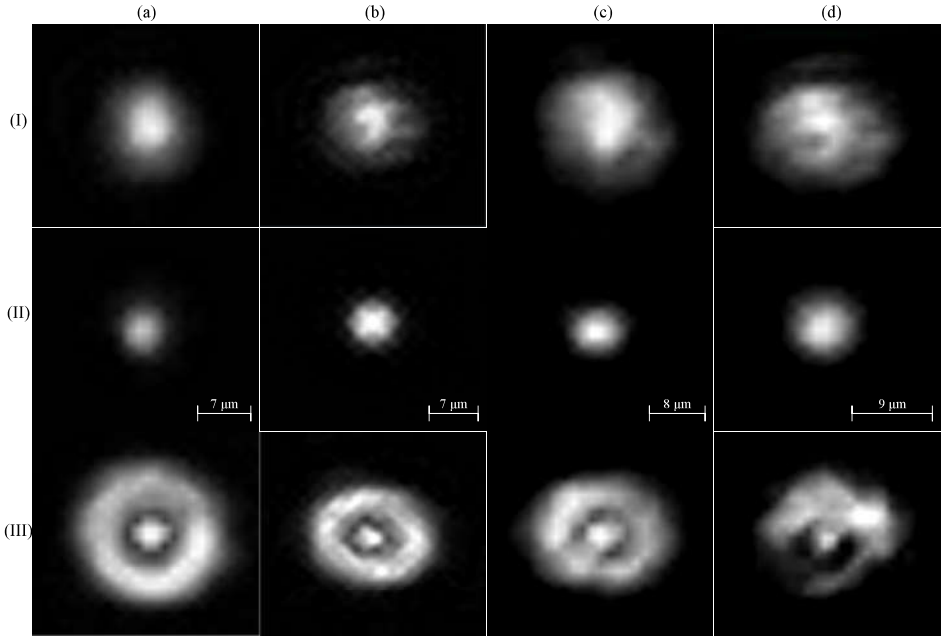

Figure 3.4. Observed diffraction patterns for different defocus positions. (I) p-defocus position at $z = 150$ µm, (II) focus position (III) n-defocus position at $z = -150$ µm for (a) mirror sample, (b) silicon, (c) steel, (d) copper sample. Adapted from [1].

Detected experimental positions in Figure 3.4 opens up a new discussion: the surface roughness of the material. Materials with different surface roughness cause detected light to lose its circular symmetry in different amounts. As it can be seen from the figure, going from column (a) to (d) in the same row, the circularity of the beam changes. After a certain amount of roughness level, it is even hard to say there is symmetry after all. To model this surface roughness within a reasonable range, LSM measurements of the samples are taken.

3.1.2. LSM Measurements

Since the shape of the detected light is highly governed by surface roughness, LSM measurement of the samples is required to correctly model the effect. Three different measurements are done for silicon, steel, and copper. Measurements are given in the format of heatmaps and values of height with respect to the position in Figure 3.5.

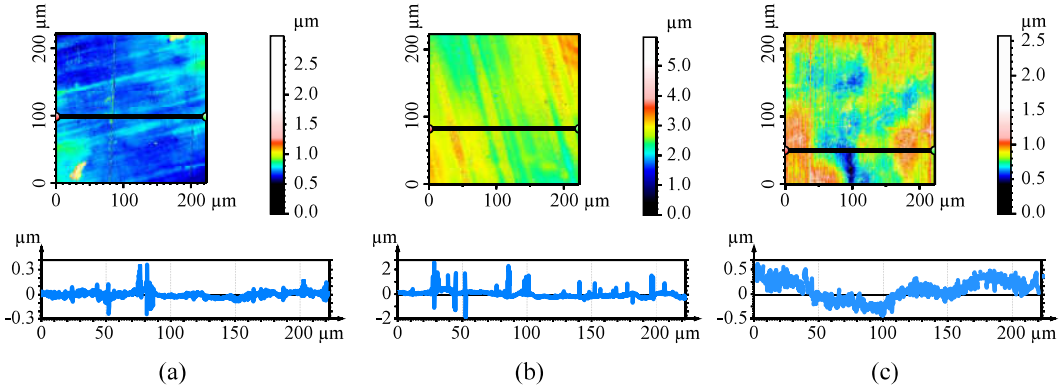

Figure 3.5. LSM measurements. (a) Silicon sample with roughness height std. about 0.050 µm and correlation length about 120 µm. (b) Steel sample with roughness height std. about 0.300 µm and correlation length about 120 µm. (c) Copper sample with roughness height std. about 150 µm and correlation length about 85 µm.

In order to obtain the std and avg values of the roughness, an easy Python implementation can be made to calculate the values. Calculated standard deviation values are given in Table 3.2. The code required for this calculation is given in Appendix A.

Table 3.2. Standard deviation values for different LSM measurements of different samples. All have 0 mean.

Measurement	**Standard Deviation Values**
Copper 1	150 nm
Copper 2	219 nm
Copper 3	177 nm
Silicon 1	202 nm
Silicon 2	49.3 nm
Silicon 3	41.5 nm
Steel 1	338 nm
Steel 2	304 nm
Steel 3	558 nm

Now that all the experimental data and the LSM measurements are obtained, Fourier optics and signal processing tools can simulate a similar setup.

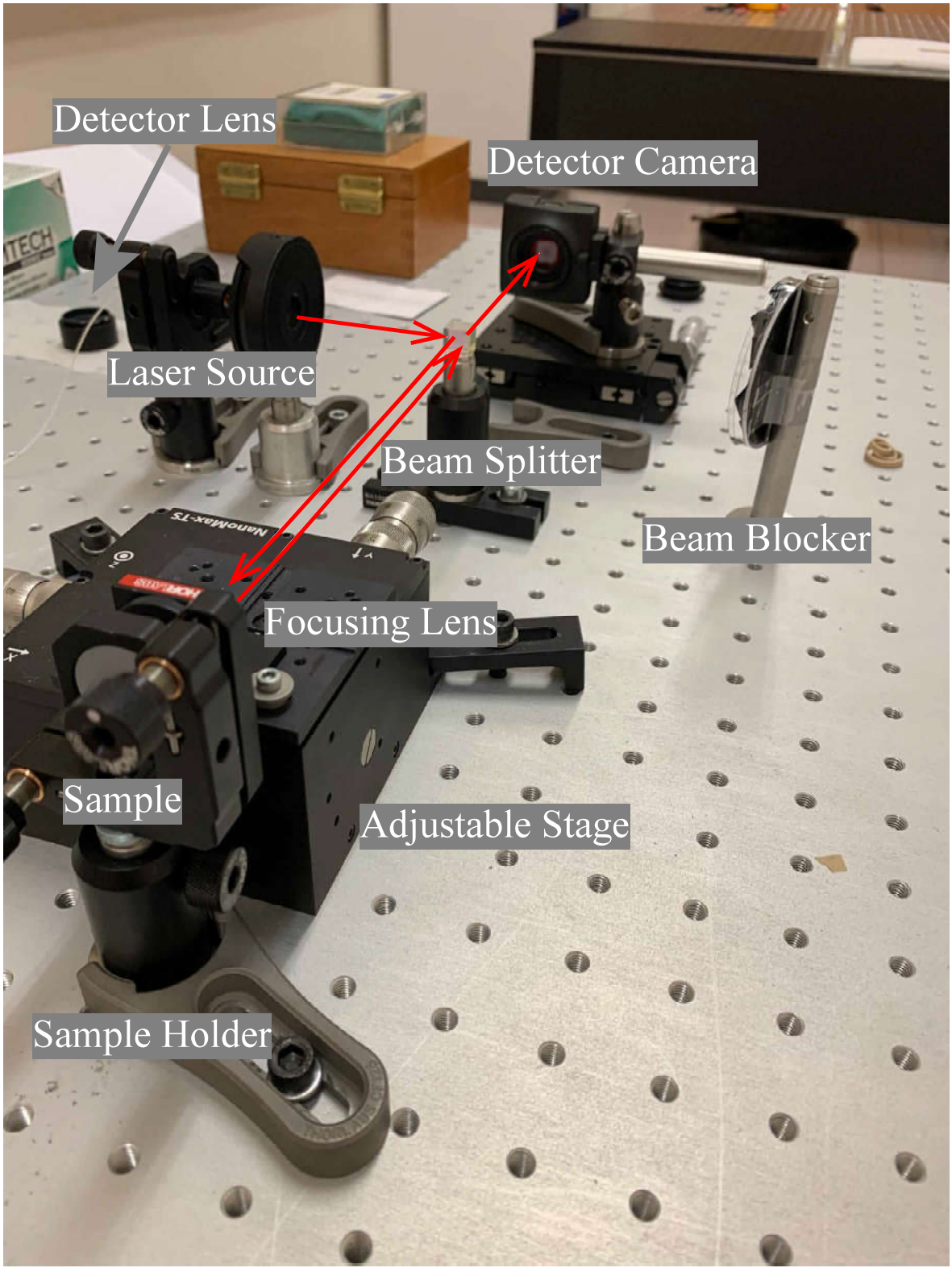

Figure 3.2. Experimental setup consisting of a laser source, 50/50 beam splitter, an aspheric lens on an adjustable stage, a sample holder, mirror as a sample, a plano-convex lens, and a USB camera.

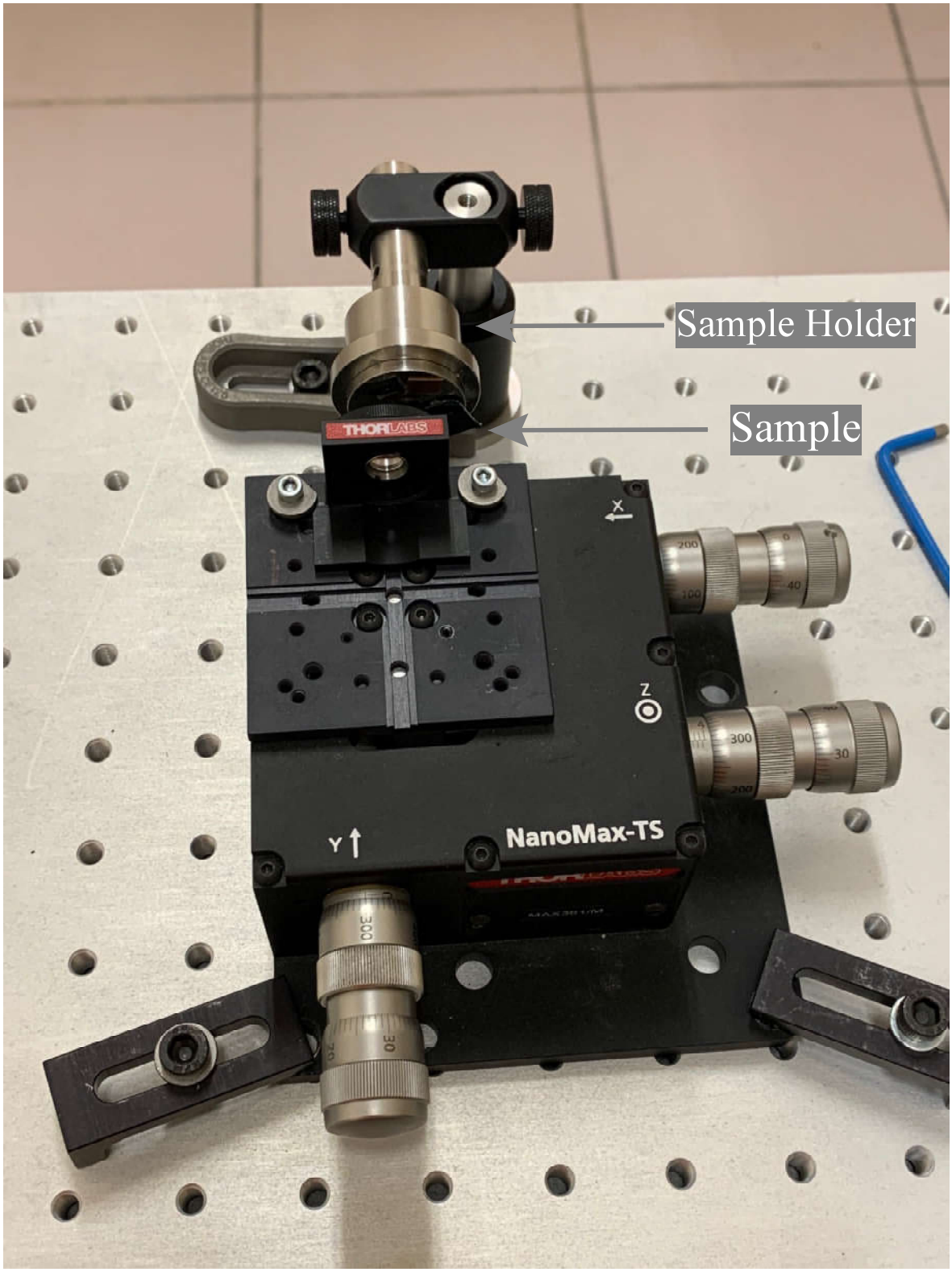

Figure 3.3. Sample holder with copper, silicon, and steel. The holder rotates on the z-axis without losing the alignment.

4. SIMULATION OF A SIMILAR LASER MACHINING SETUP

This chapter will start by explaining the theoretical background and then simulating a similar machining setup using Fourier optics [30] and signal processing. Lastly, it will explain the methods to simulate the surface roughness and the detection camera noise. Simulations will be made in Python. Scientific libraries of it will be used such as NumPy [31], scikit-learn [32], diffractio [33], Matplotlib [34], and SciPy [35]. The related code is given in Appendix C.

4.1. Laser Machining Simulation

The laser micromachining system simulation is built around wave propagation and modulation. These operations are fundamental elements of Fourier optics. In order to simulate the effect of wave propagation, Fourier integrals are needed. For that matter, a brief explanation of Fourier integrals will be given, and it will be followed by the simulation of a similar laser micromachining system.

4.1.1. Fourier Integrals

In one dimension (1D), a Fourier integral of an arbitrary function $f(t)$ can be written [36] as

$$F(w) = \frac{1}{2\pi} \int_{-\infty}^{\infty} f(t) e^{j\omega t} dt \tag{4.1}$$

which holds for every t. The function $f(t)$ in Equation (4.1) can be given as

$$f(t) = \int_{-\infty}^{\infty} F(\omega) e^{-j\omega t} d\omega. \tag{4.2}$$

The Equation (4.1) is named as Fourier transform of $f(t)$ and the Equation (4.2) named as inverse Fourier transform. These equations above can also be expanded into two

dimensions (2D), and the Fourier transform of function g, a function of two independent variables, can be written [30] as

$$\mathcal{F}\{g\} = \iint_{-\infty}^{+\infty} g(x,y)e^{-j2\pi(f_X x + f_Y y)}dxdy \tag{4.3}$$

and the inverse Fourier transformation can be given as

$$\mathcal{F}^{-1}\{G\} = \iint_{-\infty}^{+\infty} G(f_X, f_Y)e^{j2\pi(f_X x + f_Y y)}df_X df_Y. \tag{4.4}$$

The function $g(x,y)$ can be real or complex valued function. The function $G(f_X, f_Y)$ is the Fourier transform of the function $g(x,y)$ and in order to have the existence of $g(x,y)$ following three conditions (Dirichlet) [37] are sufficient and must hold.

- |**g**| integrable over infinite (x,y) plane.
- **g** has a finite number of extrema and discontinuities in any finite rectangle.
- **g** has no infinite discontinuity.

Now that an essential tool in Fourier optics is stated, further discussion of wave propagation can be made by starting from the diffraction theory.

4.1.2. Theory of Diffraction of Light

Consider an aperture at $z = 0$, illuminated by the electric field distribution $E(x', y', z = 0)$ within the aperture itself. For a point lying in another plane such that $z > 0$, the net field at this point is the contribution of each point emitting wavelets from the aperture. An illustration of this is given in Figure 4.1. Given that each wavelet is spherical, the mathematical formulation [38] of the net field can be given as

$$E(x,y,z) = -\frac{i}{\lambda}\iint_{aperture} E(x', y', z = 0)\frac{e^{ikR}}{R}dx'dy' \tag{4.5}$$

where λ is the wavelength of the light and $R = \sqrt{(x-x')^2 + (y-y') + z^2}$. The Equation (4.5) is called as Huygens-Fresnel diffraction formula. This equation can be solved for the light $x, y = 0$. This light is called on-axis. However, a numerical approach is needed for the off-axis calculation. Nonetheless, this is the usual case for most wave

propagation applications. Now it is time to discuss of propagation of light in free space.

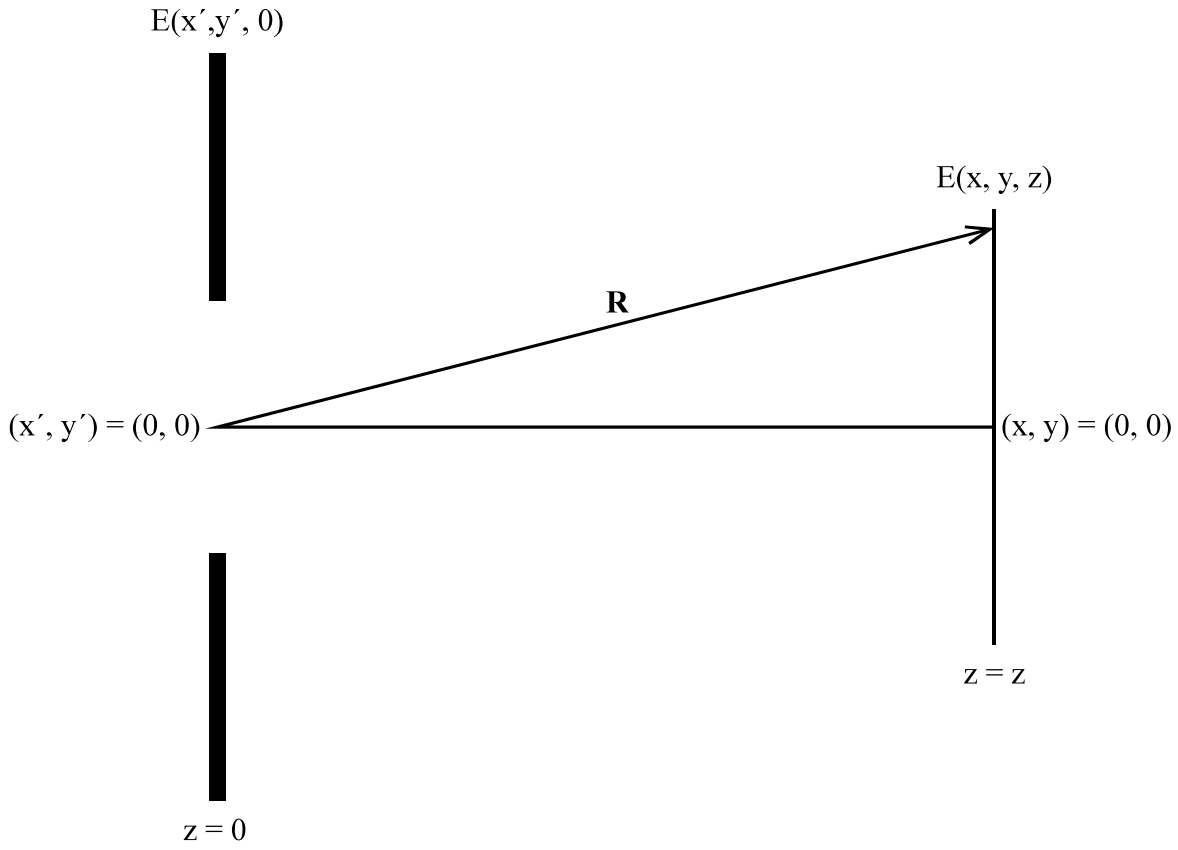

Figure 4.1. Electric field distribution $E(x\prime, y\prime, 0)$ at $z = 0$ which illuminates the plane at $z = z$. Therefore creating an electric field distribution at $z = z$ as $E(x, y, z)$.

4.1.3. Helmholtz Equation and Approximations

A light field can be written as

$$\mathbf{E}(\mathbf{r},t) = \mathbf{E}(\mathbf{r})e^{-i\omega t} \tag{4.6}$$

where ω is the frequency. This light field is a solution to the equation

$$\nabla^2\mathbf{E} - \epsilon_0\frac{\partial^2\mathbf{E}}{\partial t^2} = 0. \tag{4.7}$$

Taking the derivatives of the light field in Equation (4.6) and putting it in Equation (4.7), one can obtain

$$\nabla^2\mathbf{E}(\mathbf{r}) + k^2\mathbf{E}(\mathbf{r}) = 0. \tag{4.8}$$

Equation (4.8) is called Helmholtz equation. This equation is a special case, single frequency, for the wave equation. Now, the next step is to make the scalar approximation and assume $\mathbf{E}(\mathbf{r})$ is a scalar as $E(\mathbf{r})$. With this approximation, the Helmholtz equation becomes

$$\nabla^2 E(\mathbf{r}) + k^2 E(\mathbf{r}) = 0 \tag{4.9}$$

which can also be written as

$$[\nabla^2 + k^2] E(\mathbf{r}) = 0. \tag{4.10}$$

This is a valid approximation for spherical waves under the condition of $kr \gg 1$. The Helmholtz equation became time-independent by using the light field given in Equation (4.6) and the light field can be written as

$$\mathbf{E}(\mathbf{r}) = \mathbf{A}(\mathbf{r}) e^{-ikz}. \tag{4.11}$$

One last approximation is needed for the wave propagation simulation to be easily calculated with numerical methods.

4.1.4. Fresnel Approximation

The Fresnel approximation is when the R in the denominator is approximated by z and expanded under the assumption of $z^2 \gg (x - x')^2 + (y - y')^2$. This is a valid approximation when restricted to small angles. The approximation can be given as

$$R = z\sqrt{1 + \frac{(x - x')^2 + (y - y')^2}{z^2}} \cong z\left[1 + \frac{(x - x')^2 + y - y')^2}{2z^2} + ...\right]. \tag{4.12}$$

If one puts the approximation into the Equation (4.5), will obtain the following approximation equation as

$$E(x, y, z) \cong -\frac{ie^{ikz} e^{i\frac{k}{2z}(x^2+y^2)}}{\lambda z} \iint\limits_{aperture} E(x', y', 0) e^{i\frac{k}{2z}(x'^2+y'^2)} e^{-i\frac{k}{z}(xx'+yy')} dx' dy'. \tag{4.13}$$

This integral in Equation (4.13) somewhat is easier to solve analytically compared to the Equation (4.5). Solving this integral will yield the propagated light field at a distance z for the light source given in Equation (4.11).

One crucial point here needed to be mentioned. A Gaussian beam is not a solution to Equation (4.5). However, it is a solution to the paraxial Helmholtz equation. The paraxial Helmholtz equation is another approximation for the source light in Equation (4.11). Assuming the amplitude of the light source in Equation (4.11) changes slowly in z compared to the wavelength, which can be written as

$$\delta A = \frac{\partial A}{\partial z}\delta z \ll A, \text{ where } \delta z \sim \lambda \tag{4.14}$$

which means

$$\frac{\partial A}{\partial z} = \frac{A}{\lambda} \sim kA. \tag{4.15}$$

This would make the second derivative of the amplitude of the light source in Equation (4.15) equal to

$$\frac{\partial^2 A}{\partial z^2} \ll k\frac{\partial A}{\partial z} \ll k^2 A. \tag{4.16}$$

Therefore, Laplacian in Equation (4.10) can be expanded as its longitudinal and transverse components as

$$\nabla^2 = [\nabla_\perp^2 + \partial_z^2]. \tag{4.17}$$

Now, Laplacian in the Equation (4.10) can be applied to the light source in the Equation (4.11) and the equation

$$\nabla_\perp^2[A(\mathbf{r})e^{-ikz}] + [\partial_z^2 A(\mathbf{r}) - 2ik\partial_z A(\mathbf{r}) - k^2 A(\mathbf{r})]e^{-ikz} + k^2 A(\mathbf{r})e^{-ikz} = 0 \tag{4.18}$$

is obtained. When the terms are canceled, one can obtain the paraxial Helmholtz equation as

$$\nabla_\perp^2 A(\mathbf{r}) - 2ik\partial_z A(\mathbf{r}) = 0. \tag{4.19}$$

Before going into the simulation of a similar laser micromachining setup, a few fundamental simulation examples can be given. Therefore, the next section will consist of the propagation and focusing of waves.

4.1.5. Gaussian Beam and its Propagation

A Gaussian beam [39], which is at $z = 0$ can be written as

$$E(x, y, z = 0) = E_0 e^{-\frac{r^2}{W_0^2}} \tag{4.20}$$

where W_0 is the half of beam waist and $r = \sqrt{x^2 + y^2 + 0^2}$. The beam can be plotted for its intensity in 1D. The plot is given in Figure 4.2.

The beam's phase and intensity with respect to z can be given in Figure 4.3. If the beam passes through a plane with a refractive index of 1.5 which lies at $z = -25$ µm, the position of the beam waist would change. Therefore, the beam's new intensity and phase plots can be found in Figure 4.4. Changes in the phase and the position of the beam waist can be easily seen when Figure 4.4 compared to Figure 4.3.

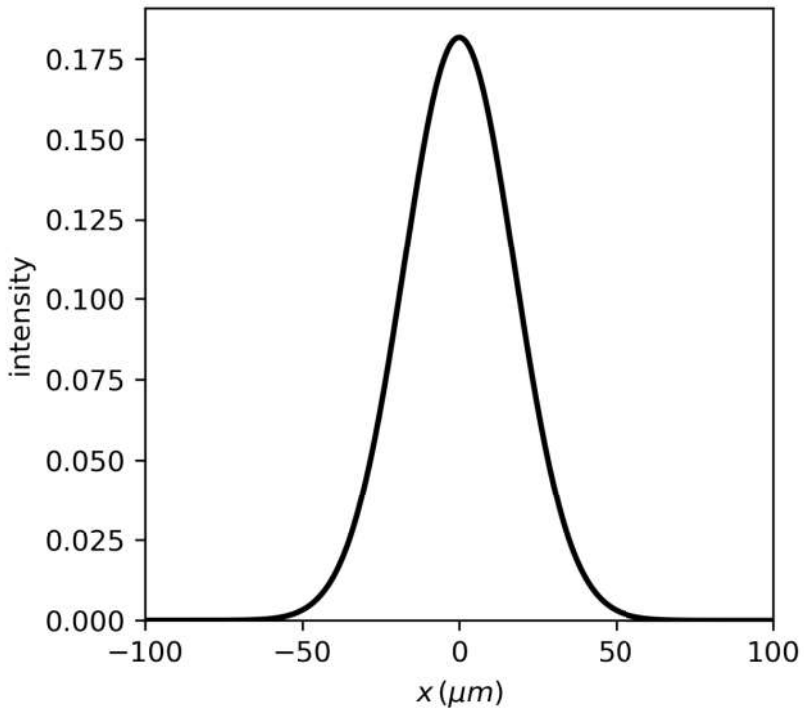

Figure 4.2. Gaussian beam intensity plot with respect to its position.

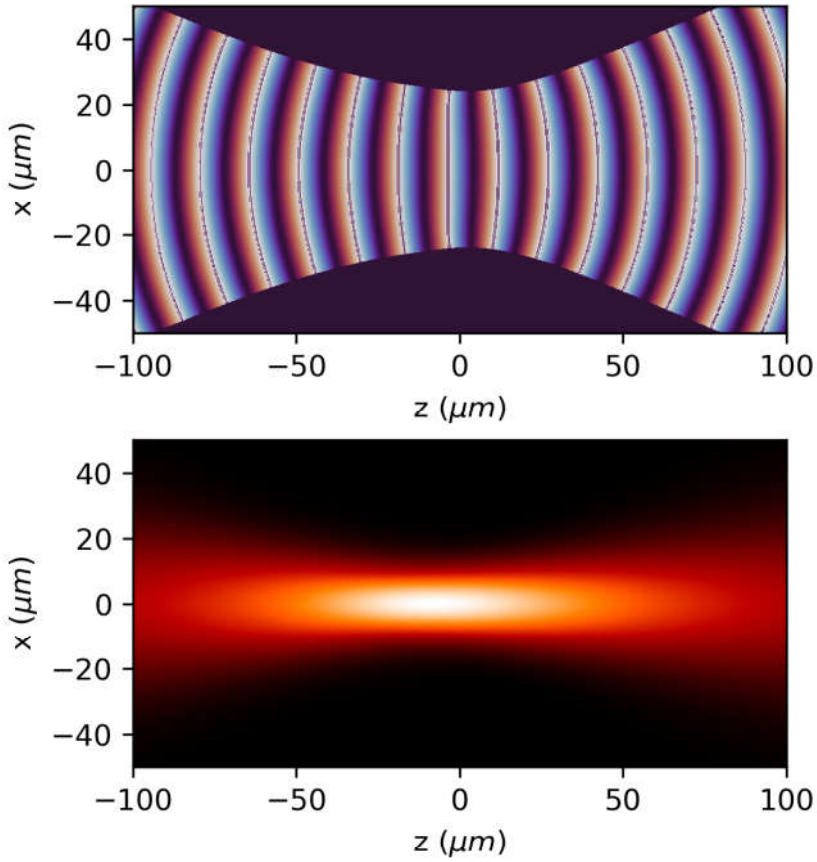

Figure 4.3. Gaussian beam phase (top) and intensity (bottom) plots with respect to its position in z.

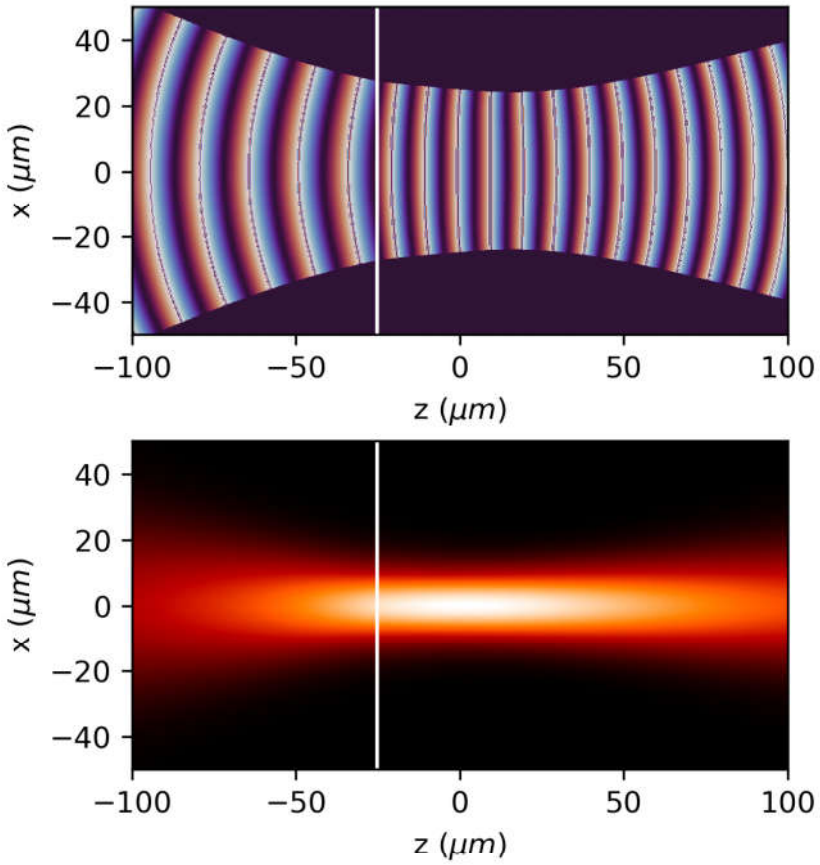

Figure 4.4. Gaussian beam phase (top) and intensity (bottom) plot with respect to its position in z when the beam passed through a plane at $z = -25$ µm with a refractive index of 1.5.

4.1.6. Focusing of a Wave

In order to simulate the focusing of a wave, it would be more informative to show it with a plane wave. A simple illustration of focusing a plane wave is shown in Figure 4.5. The intensity distribution of a plane wave, $F(\boldsymbol{x}, t)$, with respect to position is given in Figure 4.6.

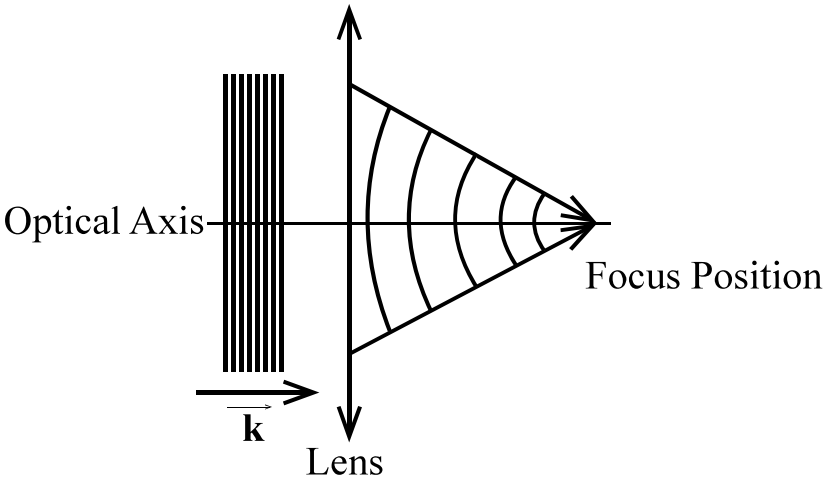

Figure 4.5. An illustration of focusing of a plane wave with a thin lens.

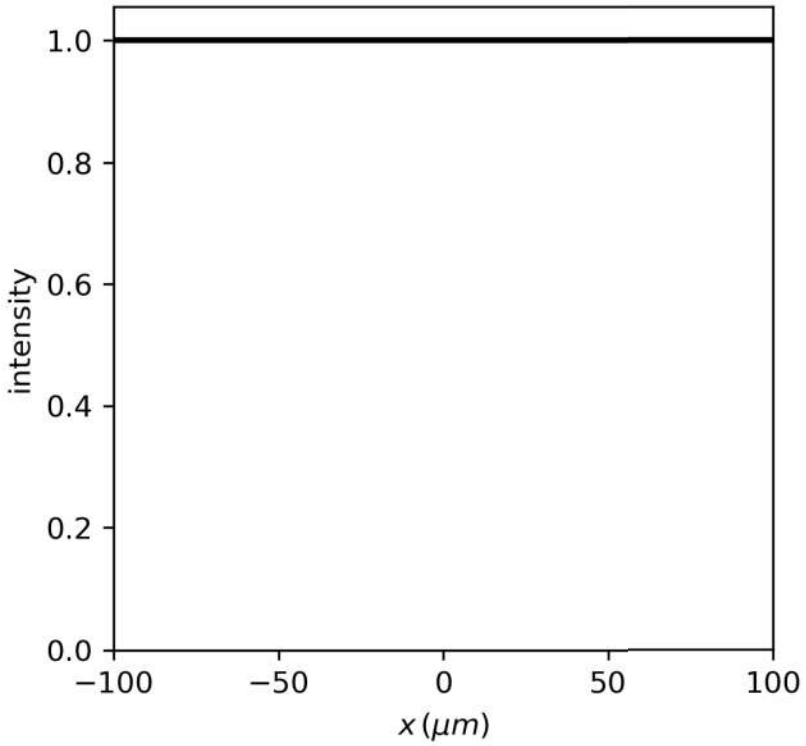

Figure 4.6. Intensity distribution of a plane wave.

If this plane wave supposes to pass through a thin lens with a focal length of $f_0 = 11$ mm, the beam would be modulated by the following function

$$L_0(x) = e^{-ik\frac{(x^2)}{2f_0}}. \tag{4.21}$$

Therefore, the wave after the lens would have the form of

$$F_1 = FL_0. \tag{4.22}$$

If this wave would be propagated distance of 11 mm using Equation (4.13) with the parameters related to this propagation and observed its intensity, one would obtain the result given in Figure 4.7.

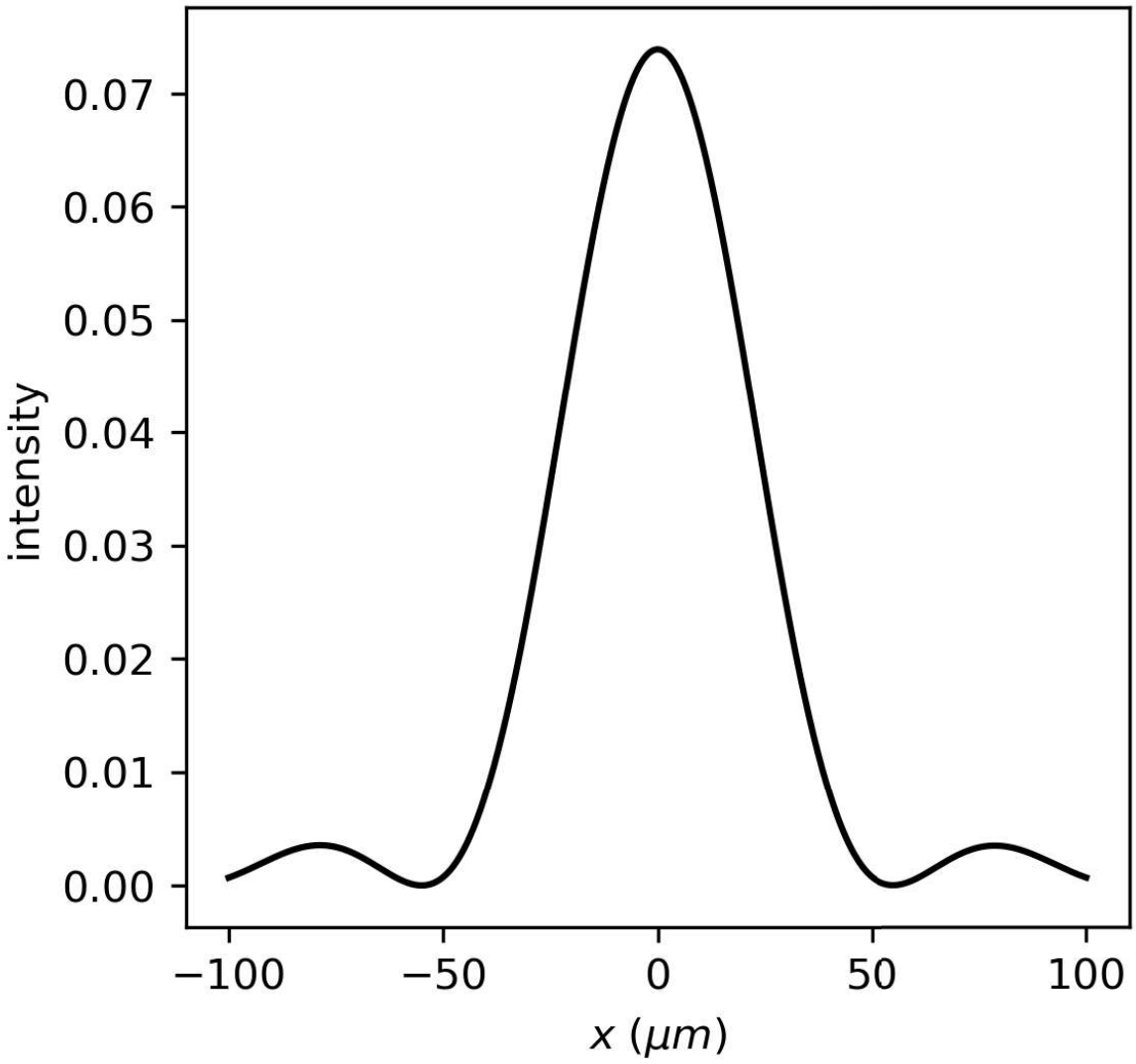

Figure 4.7. An intensity plot of a plane wave focused by an 11 mm lens.

One exciting investigation can be made by checking the positions different than the focus. This is particularly interesting since it governs the physics in this thesis to make focus detection. The observed intensities when the detection made ±10 mm of the focus position are given in Figure 4.8.

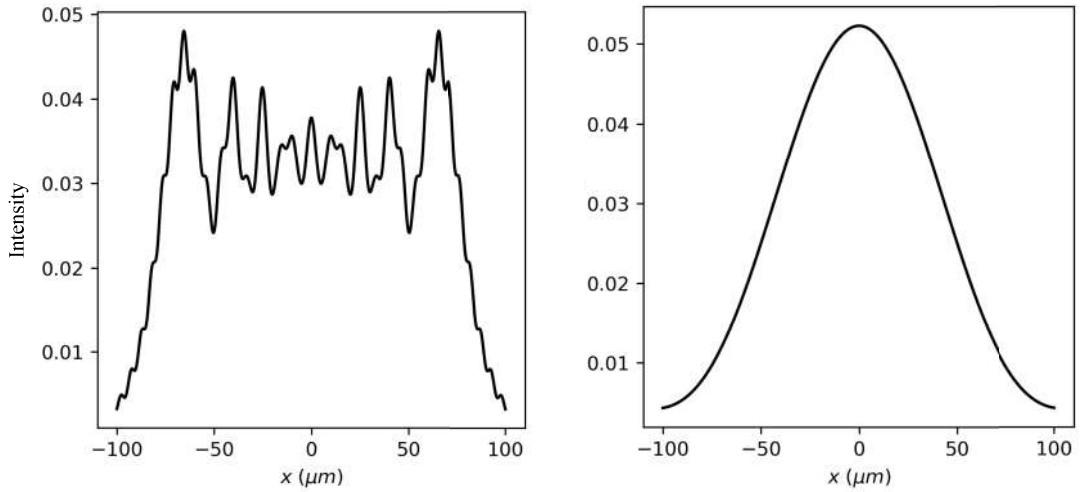

Figure 4.8. An intensity plot of a plane wave defocused. Observed intensity 10 mm before the focus position (left). Observed intensity 10 mm after the focus position (right).

In Figure 4.8, the effect of defocusing can be seen easily. When the detection is made before the focus position of the lens, diffraction ring patterns appear. Moreover, if the detection is made after the focus position, a wide wave compared to the focus position will appear. This result is used for the focus detection in the experiment and the simulation.

4.1.7. Laser Machining Simulation

A similar approach of Xu et al. [12] is used in this work with a vital difference. The reflecting light in this simulation is considered a Gaussian wave instead of a spherical wave. This is mainly due to having a Gaussian laser beam source.

A Gaussian beam, which is the solution to the paraxial Helmholtz equation in 3D, can be written as

$$U_0(x, y, z = 0) = E_0 e^{-\frac{r^2}{w_0^2}}. \tag{4.23}$$

The wave's propagation can be calculated using the Fresnel approximation in Equa-

tion (4.13) for L instead of z. The propagation of Equation (4.23) for L distance can be given as

$$U_1(x,y,L) = -i\frac{e^{ikL}e^{\frac{ik(x^2+y^2)}{2L}}}{\lambda L}\int\int U_0(x',y',0) \\ e^{ik\frac{x'^2+y'^2}{2L}}e^{-ik\frac{xx'+yy'}{L}}\,dx'dy'. \tag{4.24}$$

After this propagation, the reflected wave arrives at the front surface of L_1. The modulation [40] of the incoming wave by a lens with focal length f_1 can be given as

$$l_1(x,y) = e^{-ik\frac{(x^2+y^2)}{2f_1}}. \tag{4.25}$$

Therefore, the wave just after the L_1 in the form of

$$U_2(x,y) = U_1(x,y)l_1(x,y). \tag{4.26}$$

Another wave propagation is needed towards L_2. This propagating can be obtained by changing L to D in Equation (4.24) and E_0 to U_2. The calculated wave just before L_2 is U_3. Again, modulate the wave with L_2. Since L_2 have focal length of f_2. So, changing f_1 into f_2 in the Equation (4.25) and call this lens modulation l_2. Now, the initial wave is in the form of

$$U_4(x,y) = U_3(x,y)l_2(x,y). \tag{4.27}$$

One more propagation is needed to make the wave reach the detection camera by propagating it in the free space for f_2 distance using Equation (4.24) and naming it U_5. To obtain the images on the detector screen, pixel values are needed to be calculated. This would correspond to the intensity of the wave. Hence, the intensity distribution at the camera is as

$$I(x,y) = |U_5|^2. \tag{4.28}$$

Designed simulation's length for two consecutive data points corresponds to 0.909 µm. For this setup, the simulation has less than a λ for each consecutive data point. Furthermore, the generated images include more than 99% of the beam's energy. An illustration of these propagations and lens modulations is given in Figure 4.9.

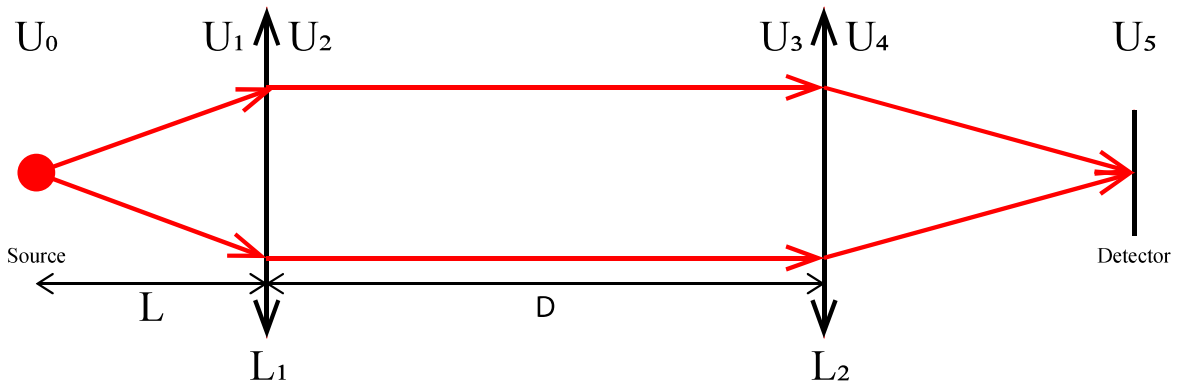

Figure 4.9. Illustration of the simulation consists of two lenses for wave modulation, three parts for wave propagation, and a detector. The source beam, U_0, propagates three times and gets modulated by each lens before reaching the detector.

4.2. Surface Roughness Simulation

There are two essential parameters for defining a rough surface: standard deviation and correlation length [41, 42]. Surface roughness (input noise) with these two parameters can be modeled with many different approaches, including the fast Fourier Transform (FFT) method [43], Mandelbrot-Weierstrass Function [44], and neural networks with single or multiple inputs [45]. These approaches are mainly based on adding random heights to the material's surface. In this thesis, moving average processes [46] will be used to simulate the surface roughness. This way, the effect of surface roughness can be observed for the simulation of a similar laser machining setup.

The method of moving average processes starts by generating a random Gaussian noise with zero-mean and root-mean-square (RMS) value σ height distribution $h_0(x, y)$. The height distribution h_0 does not have any correlation between the points. To correlate the points, one needs to multiply this distribution with a weight function to obtain a Gaussian distribution having the same mean and RMS value but now correlated points. Therefore, the process can continue by defining the autocorrelation function, which is a Gaussian, as

$$C(\boldsymbol{R}) = \frac{\langle h(\boldsymbol{r})h(\boldsymbol{r}+\boldsymbol{R})\rangle}{\sigma^2} = e^{\frac{-\boldsymbol{R}^2}{\lambda_0^2}}. \tag{4.29}$$

The correlation length is when the autocorrelation function falls to its e^{-1} value which is λ_0. Now, the Fourier transform of Equation (4.29) needs to be taken to obtain the

power spectrum function. This calculation is given as

$$P(\boldsymbol{k}) = \frac{1}{2\pi} \int e^{\frac{-\boldsymbol{R}^2}{\lambda_0^2}} e^{i\boldsymbol{k}\cdot\boldsymbol{R}} \, d\boldsymbol{R}. \tag{4.30}$$

If the inverse Fourier transform of square rooted Equation (4.30) is taken, one obtains the weight distribution, w(x,y), to turn our uncorrelated Gaussian distribution into a correlated one. Therefore, multiplying uncorrelated distribution $h_0(x, y)$ with w(x,y) yields the new distribution equation

$$h(x, y) = h_0(x, y)\mathrm{w}(x, y). \tag{4.31}$$

As a result, a Gaussian distributed surface roughness with the std of σ, zero-mean, and the correlation length λ_0, $h(x, y)$, in Equation (4.31) is obtained. Now that the surface roughness function is found, one can weight the initial beam in Equation (4.23) with Equation (4.31) and obtain the initial beam as

$$U_0'(x, y) = U_0(x, y)h(x, y). \tag{4.32}$$

The effect of surface roughness can be assumed as retarding the wave in phase [47, 48]. So the phase of surface roughness simulations can be found in Figure 4.10.

The phase distributions in Figure 4.10 also clearly explains the importance of correlation length. An increase of the std has a similar effect as the decrease in correlation length. So one must consider the surface roughness not just by its std but also by its correlation length for this application. A further discussion will be made in conclusion with the results of machine learning models.

For this simulation, surface roughness standard deviation levels are based on the LSM measurements of our copper, steel, and silicon samples. LSM Measurement can be seen in Figure 3.5. The effect of correlation length can be observed experimentally as well from Figure 3.4. Even though the copper sample has a lower std than the steel, it still has the worse form of the recorded light.

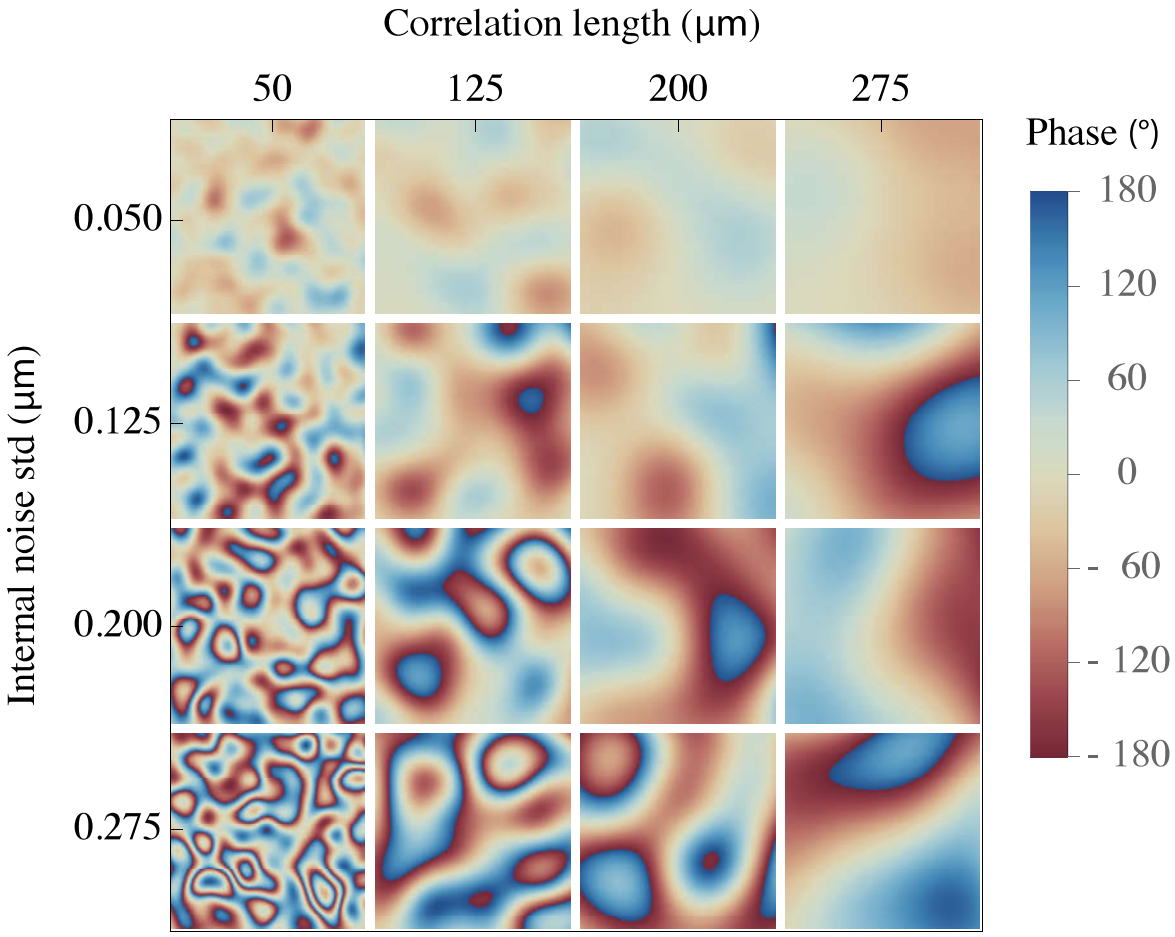

Figure 4.10. Phase values of roughness simulation for different noise stds and correlation lengths.

4.3. Detector Noise Simulation

Since the machining setup is not isolated from outside effects and the detector itself consists of noise, it is necessary to consider the possible effects of those noises on the detection system. These noises can be considered Gaussian [49], and the output (detection) noise level can be compatible with a cheap camera's detection noise. Thus, given that $I(x, y)$ is the image intensity at point (x, y), the noisy image intensity is given by

$$I'(x, y) = I(x, y) + \eta_{\text{out}} \tag{4.33}$$

where $\eta_{\text{out}} \sim \mathcal{N}(0, \sigma^2_{\text{out}})$. Generated images that include both input and output noise for focus position can be found in Figure 4.11.

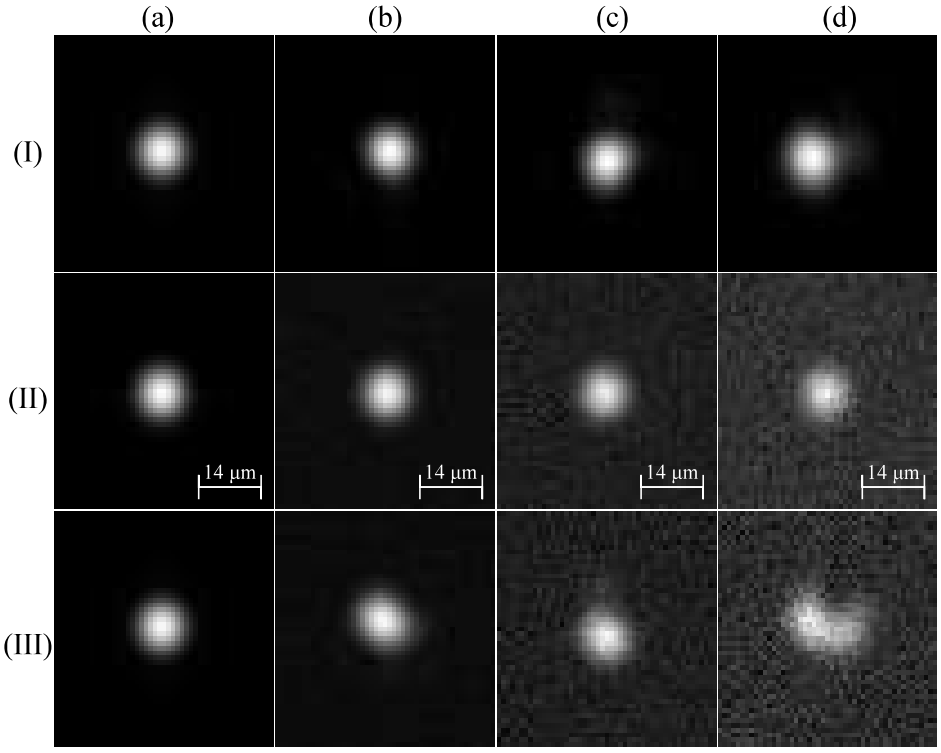

Figure 4.11. Simulated images when different noise types applied to the in-focus position (a): (I) Only input noise with constant correlation length applied for (b) 250 nm std, (c) 500 nm std, (d) 750 nm std. (II) Only output noise applied for (b) 0.01 std, (c) 0.03 std, (d) 0.05 std, (III) when both of the noises above applied.

4.4. Data Generation for Simulation

Details of simulation parameters can be found in Table 4.1. Using the parameters in Table 4.1, one can generate images for each sample position with surface roughness and detector noise. Images are generated in .h5 format for flexibility, organization, and practicality for carrying extensive complex data.

The number of images generated is 200 per defocus position with a specific value in both surface roughness value and detector noise. Eleven different defocus positions, ten different surface roughness standard deviations, ten different surface roughness correlation lengths, and ten different detector noise standard deviations.

For example, in defocus position -150 µm there are 200 images with surface roughness correlation length 50 µm, surface roughness standard deviation 0 µm, and detection camera noise standard deviation 0. Two hundred more for detection camera noise standard deviation 0.006 and the rest is the same. This goes on for each value of detection camera noise standard deviations, surface roughness correlation lengths, and standard deviations for one defocus position. Then this process is repeated for each defocus position. In total, two million two hundred thousand images are generated to be used by the machine learning model. Each image is square and has a size of (50, 50) pixels. The generation of the images is time-consuming due to wave propagation and correlation length generation. However, using the power of multiprocessing with a decent computer drastically reduces the computation time.

Table 4.1. Parameters and their values used in Python code for simulation of a similar laser machining setup with noise included.

Simulation Parameters	**Values**
Laser wavelength	0.976 µm
Rayleigh length of the system	150 µm
Beam waist	13.7 µm
Focusing lens focal length	11 mm
Detection lens focal length	150 mm
Distance between the lenses	300 cm
Defocus distances	$\{-150, -140, \ldots, 0, \ldots, 140, 150\}$ µm
Detection camera noise std.	10 equally spaced numbers from 0 to 0.05
Surface roughness correlation length	10 equally spaced numbers from 50 µm to 500 µm
Surface roughness height std.	10 equally spaced numbers from 0 µm to 0.5 µm
Generated image size	(50, 50) pixels

5. MACHINE LEARNING FOR FOCUS DETECTION

This chapter will start by broadly explaining machine learning methods and then share the results of four different machine learning methods for the experimental data. Finally, will apply the machine learning model with the best result to the simulated data and share the prediction outcome.

5.1. Machine Learning

Computer systems or machines must make a decision depending on their situation or the process they are used in. In order to make decisions, they are equipped with artificial intelligence, a simulation of human intelligence in machines or systems. This intelligence can be embodied by just using hard-coded if statements or a more computational approach, machine learning [50].

Machine learning consists of many approaches, such as supervised, unsupervised, and reinforcement learning. Depending on the problem at hand, different methods can be chosen.

In supervised learning models, a learner receives training data, which are labeled, and makes a prediction depending on what it learned from the data. This learning process depends on the data and the problem. There are many different supervised learning models in the literature, and each can be better than the others depending on the problem that needs to be solved. For example, for image processing problems, CNNs are the state-of-the-art model. However, another approach, like a simple linear regression [51] can be much more helpful for a time series problem.

The use of machine learning models has found application in many different areas of physical sciences [52]. This is due to their vast potential and power in solving complex problems. These methods also found a place in laser material processing to improve the process accuracy and speed [53]. Furthermore, researchers [54] used the application

of CNN for single-axis beam translation detection for the monitoring of the machining process.

Machine learning can solve the problem addressed in this thesis efficiently and accurately. Since the data are labeled, i.e., defocus distances, the focusing problem is solvable with supervised learning. First, four supervised learning models will be considered for the data obtained by the experiment. These models can be split into two groups: non-CNN-based and CNN-based models. Non-CNN-based approaches are logistic regression and multi-class support vector machines. CNN-based are CNN classifier and CNN predictor. The best-performing model, the CNN predictor, will then be used for the simulated data.

5.2. Machine Learning Models for Experiment

This section will explain the structure of experimental data and ML models used for the data.

5.2.1. Data Structure

Two recordings with 4000 frames for each material (copper, steel, silicon) and defocus distances in the range of ± 150 µm with 10 µm step sizes as an 8-bit grayscale image with a 40 by 32 pixels resolution. One of the recordings is saved for testing, and the other one is used for training and validation. Split of $85 - 15\%$ used for training/validation. This would correspond to 3400 images per material for each defocus distance for training and 600 images for testing.

5.2.2. Non-CNN Models

Processing starts by flattening each grayscale 40x32 image to a vector. This is followed by principal component analysis [55] (PCA) to the splitted training and validation data. This process reduces the vector's correlated components into its uncorrelated principal components. PCA assumes linear separability and works well while

using it to reduce dimensionality. This process also conserves the essential features. Once the principal components are obtained from the training data, validation data is projected to a subspace of the principal components. The library scikit-learn is used for the implementation of PCA and the machine learning models.

First, logistic regression is used to classify each frame's focus distance. The input to the logistic regression classifier is the PCA projected vector. This model used L_2 regularization with the strength of 1.0.

Then, one-versus-rest multi-class support vector machine (SVM) is used. The kernel is RBF and again L_2 regularization with the strength of 1.0.

5.2.3. CNN Models

Two different CNN models are trained. One is for classification and the other for predicting the defocus distance. Both models hold the same architecture, while the final layer is the only difference. These networks are coded with PyTorch [56] and trained with the batch size of 16. The architecture model are given in Table 5.1.

CNN pretictor model tries to predict a real number from the image it is fed in the $[0, 1]$ range. This would mean that 0 is -150 µm and 1 is $+150$ µm. Each predicted output is mapped to the closest defocus distance class for accuracy computing. For example, 126 is being mapped into 130. For the robustness of the model, the training was done for 20 µm step sizes. So the training defocus distance range is {-150, -130,..., -10 , 0, +10,. . . , +130, +150} µm. However, the model is tested and validated on all of the data.

CNN classifier tries to classify the image it is fed. The model is trained for 10 epochs. The weights of epoch with the highest validation accuracy are saved for classifying.

Table 5.1. The architecture of the CNN is used for classification and prediction, excluding the final layer. Adapted from [1].

Layer	**Parameters**
Input	Size: 1x40x32
Conv	Out channels: 32 Kernel size: (3,3)
Max-pool	Kernel size: (2,2) Stride: (2,2)
Conv	Out channels: 64 Kernel size: (3,3)
Max-pool	Kernel size: (2,2) Stride: (2,2)
Flatten	Output size: 1x3072
Dense	Output size: 512
Dense	Output size: 256

5.2.4. Results of Machine Learning Models for Experiment

Four designed models are tested on previously unseen data, and the results are given in Table 5.2. These results also include the inference speed of each model since the speed is essential for real-time applications.

Table 5.2. Accuracy and CPU inference speed of each model when tested on the testing dataset. Adapted from [1].

ML Model	Accuracy of Copper (%)	Accuracy of Silicon (%)	Accuracy of Steel (%)	Inference speed on CPU (Hz)
SVM	94	92	95	47
LogReg	98	98	99	7580
CNN-P	99	96	91	1408
CNN-C	100	100	100	1170

As mentioned before, CNNs are the state-of-the-art architecture for image classification problems. Therefore, it is not surprising to see the best performance came from CNN models. CNN-C (CNN for classification) performs best among the two designed CNN models. The result for CNN-P (CNN for prediction) is impressive because it only trained on half of the defocus training data. CNN-P outperformed the SVM model, which trained on all the defocus training data.

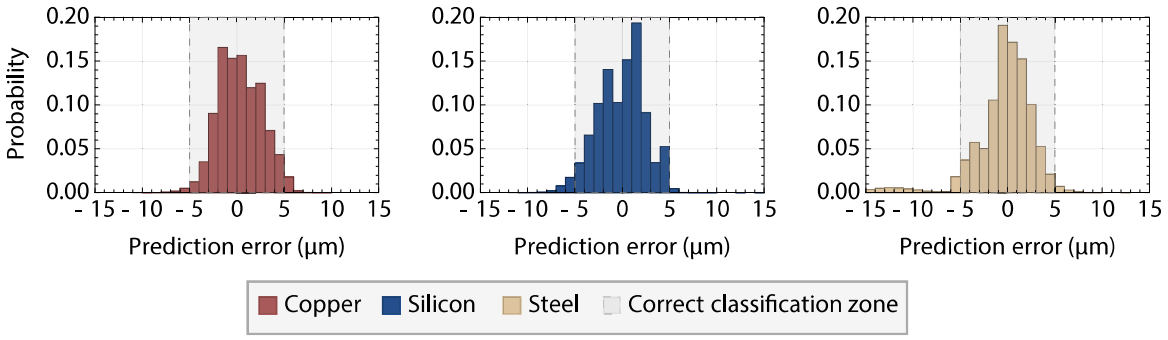

Figure 5.1. Histogram of the testing prediction errors for each material. The correct classification zone indicates the tolerable range of prediction error in which a prediction results in correct classification.

When it comes to processing speed, the logistic regression model is the best. This is an expected result since it is a smaller neural network model compared to CNN-C and CNN-P. SVM is the worst, this is due to testing of 31 different SVM models for each given image with a one-versus-rest strategy. Moreover, CNN models perform similarly to each other when it comes to speed. Testing on GPU does not increase the inference speed significantly. This means the models are already well equipped with a CPU (Central Process Unit) and do not require expensive processing power.

The performance of CNN-P requires more close look. The close look starts with its confusion matrices for three different surfaces. These matrices are given in Figure 5.3. The predictor model outperforms logistic regression and SVM models and only loses to CNN-C just by 1% for copper. The confusion appears on 0 µm, where around 30% of images were incorrectly classified to be at -10 µm. It performs similarly to silicon and only incorrect classification for 140 µm. Steel is the worse performing among the

three. Defocus distances of 0 µm and −20 µm were never correctly classified.

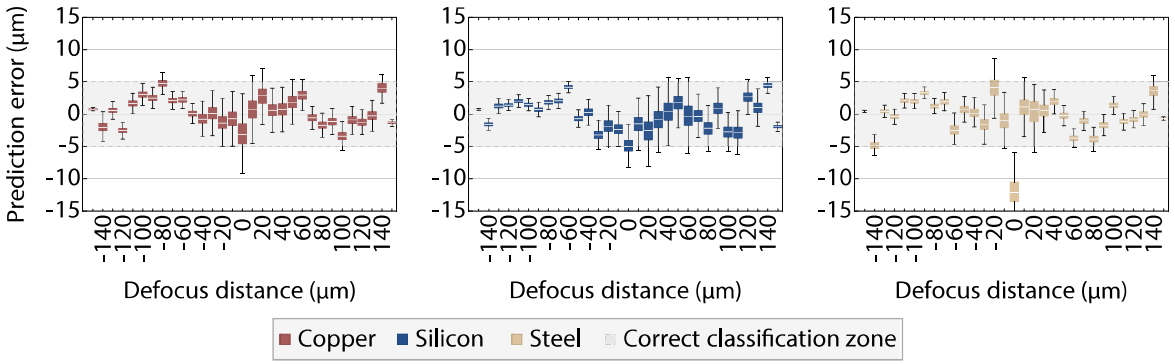

Figure 5.2. Box and whisker plots of testing prediction errors for each material. Each plot's lower and upper fences represent the minimum and maximum prediction error with outliers excluded. The correct classification zone indicates the tolerable range of prediction error in which a prediction results in correct classification.

Furthermore, box and whisker plots, which are given in Figure 5.2, can be investigated for each material. The similar behavior of Copper and silicon prediction can be observed again. Maximum prediction error is about 10 µm. However, steel has a higher prediction error band compared to the other two materials.

One last investigation can be made by obtaining the probability distribution of each possible prediction error. This investigation is given in Figure 5.1. Copper and silicon do not have a bigger prediction error probability than steel. The peak probability of making a prediction error is lower for the copper sample.

5.3. Deep Learning Model for Focus Prediction in Simulation

Now that the results of different models are seen, the performance of CNN models is better than SVM and logistic regression. Therefore, in this section, CNN models will be used to determine the position of the sample for correct focusing. More specifically, CNN-P will be used due to its ability to predict unseen defocus distances.

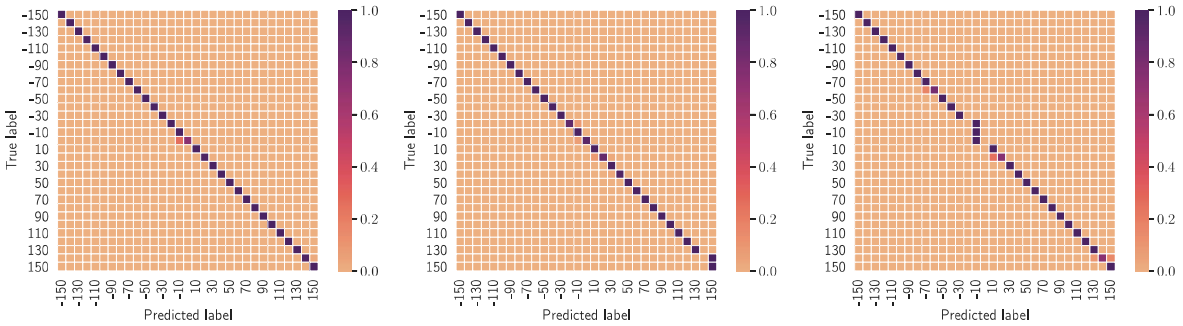

Figure 5.3. Confusion matrices of CNN-P when tested on the testing datasets of copper, silicon, and steel, respectively. Adapted from [1].

5.3.1. Data Structure

200 images per defocus position ranging from ±150 µm with 10 µm increments and with a specific value for surface roughness values and detection camera noise. The data structure is as explained in Section 4.4 when the generation of the images is made, and the parameters to simulate this data are given in Table 4.1.

However, as mentioned before only defocus distances {-150, -130, ..., 0, ..., +130, +150} µm used for training and the testing and validation are made on all the data generated.

5.3.2. Model Parameters

The same CNN-P model was used for syntactic images as well. The network structure is given in Table 5.1. The model is trained with a batch size of 16.

Model is predicting a real number in the range of $[0, 1]$ which corresponds to defocus distances from -150 to 150 µm. The predicted outputs are mapped to the nearest defocus distance class. For example, 112.2 is mapped to 110.

5.3.3. Prediction Results

Once the training is done, the model performance testing can start. First, the model's prediction accuracy can be investigated when keeping two of the three parameters fixed and while changing the third. The result of this investigation can be given in Figure 5.4.

When the correlation length and external noise are constant, prediction accuracy decreases while the internal noise std increases. This is an expected result also due to higher noise std causing the surface to be rougher.

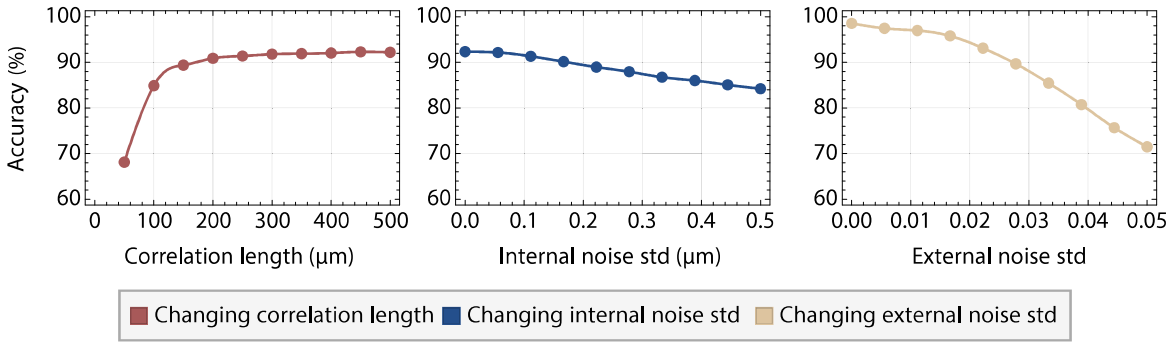

Figure 5.4. Plots of the testing accuracy for changing correlation length, internal noise std, and external noise std. In each plot, the model is tested on simulated images whose parameters match the parameter on the x-axis.

When the internal noise std and external noise std is constant, the accuracy increases as the correlation length increase. This is an expected result since the higher the correlation length, the smoother the surface. Therefore, minor deformation of the reflected light.

Lastly, when the parameters of surface roughness are constant, the accuracy gets worse and worse with the increase of external noise std. This is mainly due to having massive noise in the detection system and making the observed light too noisy to determine a reasonable outcome.

In order to further investigate the effect of the noises on prediction accuracy, one can check the results of the accuracy heatmap given in Figure 5.5. The model performs exceptionally well for high correlation length and low external and internal noise std values. The prediction values go above 90% for this case.

However, the model performs poorly as expected for high external and internal noise std and low correlation length. Even if the result may seem low and unsatisfying, the values used for the noise are way beyond what is experimentally measured with LSM. Therefore, it is reasonable to say the model is making reasonable decisions. The predictions it make are still have higher success, more than ten times, than just random guessing.

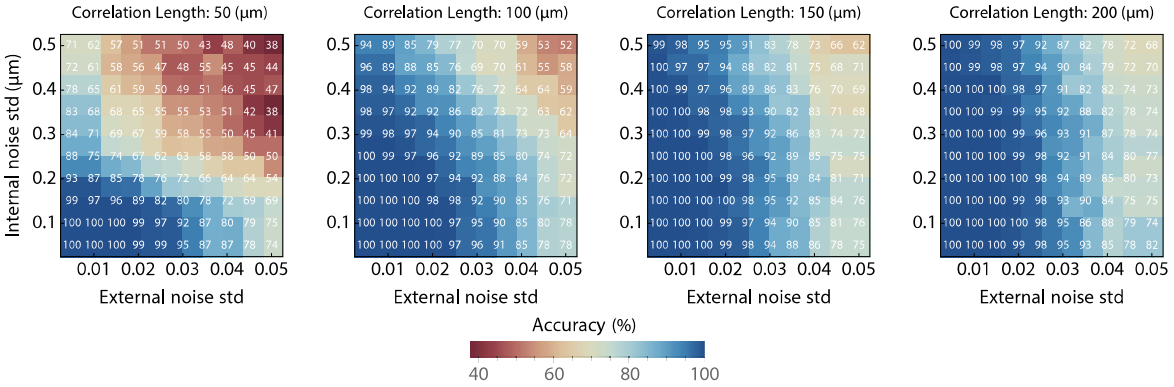

Figure 5.5. Heatmap of the average testing accuracy. A cell of a heatmap with internal and external noise std i and j, and correlation length k represents the classification accuracy of the predictor model when classifying testing data that have internal noise std, external noise std, and correlation length i, j, and k, respectively.

6. CONCLUSION

This thesis offers four possible machine learning models for real-time focus detection and controlling of ultra-fast laser micromachining in a cheap, affordable, and easily applicable way.

First, an experimental setup is built for detecting reflected light from the processing material. This experimental setup is given in Figure 3.1. Depending on the material's position compared to focusing lens focus, a tight Gaussian beam, a wide Gaussian beam, or diffraction rings appeared. These positions are called in-focus, p-defocus, and n-defocus, respectively. This setup consists of a long focal length aspheric lens and a cheap camera, making the setup cheaper and easily buildable compared to previous works.

Once the experimental setup is established, recordings are taken for different defocus positions ranging from ± 150 µm with 10 µm step size. Then these recordings are used to train, validate, and test four different machine learning models.

These four machine learning models are logistic regression, support vector machines, a convolutional neural network for classification, and a convolutional neural network for prediction. All of these models

The prediction accuracy and inference speed of all four models are given in Table 5.2. SVM has the worse performance overall. It is both slow and has terrible prediction accuracy compared to other models. A change in the one-versus-rest strategy might increase its performance. Among the others, logistic regression is best with speed since it is a shrunk version of CNN-C and CNN-P architectures.

If the comparison is made just by looking at the accuracy numbers between the CNN models, CNN-C takes the crown. Nonetheless, it only does classification and can only classify the defocus distances it has trained on. However, CNN-P is trained on

fewer data and can predict the defocus distances it has not trained on. This situation puts the CNN-P in a unique position. The fact that it can perform at a similar level and beyond with fewer data suggest that the prediction model is better than the classification for this type of problem.

Furthermore, the inference speed of CNN-P also makes it applicable for real-time use. This is a necessary trait to have if the focusing time is high, then the micromachining speed is reduced therefore costing time and money.

The machine learning approach's efficiency can also be realized by comparing the beam waist and detector pixel size, which is given in Table 3.1. This comparison of 6.84 µm beam waist and 3.6 µm detector pixel size shows the low-resolution setting and machine learning methods can efficiently make focus detection in small step sizes. This trait also allows for having a cheap setup with high performance.

When micromachining is considered, the material which is to be processed includes surface roughness. For this matter it should be considered as well. Moreover, since this experiment consists of a detection camera, one must also consider the noise in the detection camera as well.

In order to consider the surface roughness, Fourier optics simulation is done for a similar laser micromachining setup. The moving average process is used to generate surface roughness, and a simple Gaussian noise model is used for the detection of camera noise.

Effect of each noise is seen clearly in Figure 5.4. When the correlation length increases, the surface becomes smoother, and the CNN-P model can make better predictions. However, after a certain level of increase, the accuracy does not get the benefit too much, and the accuracy saturates. The model already starts from a good accuracy level. and it increases more than 20% when the surface roughness correlation length is increased about 10 times.

The surface roughness std negatively affects the accuracy when the std is increased. The surface becomes rougher with higher maximum and lower minimum. However, increasing from 0 to 500 nm, the accuracy stays still above 80%.

External noise has to most impact on the accuracy. This is an expected outcome, as discussed before. However, the noise level is about 5% the accuracy is still above 70%.

Finally, heatmaps in Figure 5.5 adds another layer of information to this work. Depending on the noise levels, the model can make prediction accuracies differentiating from 100% to 36%. Nonetheless, the prediction is below 70% when the noise levels are above what is measured in real life by LSM.

In this thesis, a cheap and effective focus detection system based on machine learning models has been offered for ultra-fast laser micromachining in real-time application. The model offered tested in both experiment and simulation and achieved high accuracy and robustness to noise in the system. The model can be applicable to many materials such as copper, steel, and silicon with the same architecture and can run on a cheap computer without a GPU.

The preliminary results of this thesis have been presented and published. For the experimental part, please refer to Polat et al. [1] and please refer to Elahi et al. [57] and Polat et al. [58] for the simulation part.

REFERENCES

1. Polat, C., G. N. Yapici, S. Elahi and P. Elahi, "Machine Learning-Based High-Precision and Real-Time Focus Detection for Laser Material Processing Systems", *Optics, Photonics and Digital Technologies for Imaging Applications VII*, Vol. 12138, pp. 7–13, Strasbourg, 2022.

2. Choudhury, I. A. and S. Shirley, "Laser Cutting of Polymeric Materials: An Experimental Investigation", *Optics & Laser Technology*, Vol. 42, No. 3, pp. 503–508, 2010.

3. Rauh, B., S. Kreling, M. Kolb, M. Geistbeck, S. Boujenfa, M. Suess and K. Dilger, "UV-Laser Cleaning and Surface Characterization of an Aerospace Carbon Fibre Reinforced Polymer", *International Journal of Adhesion and Adhesives*, Vol. 82, No. 1, pp. 50–59, 2018.

4. Akman, E., A. Demir, T. Canel and T. Sınmazçelik, "Laser Welding of Ti6Al4V Titanium Alloys", *Journal of Materials Processing Technology*, Vol. 209, No. 8, pp. 3705–3713, 2009.

5. Lahoz, R., G. F. de la Fuente, J. M. Pedra and J. B. Carda, "Laser Engraving of Ceramic Tiles", *International Journal of Applied Ceramic Technology*, Vol. 8, No. 5, pp. 1208–1217, 2011.

6. Ravi-Kumar, S., B. Lies, X. Zhang, H. Lyu and H. Qin, "Laser Ablation of Polymers: A Review", *Polymer International*, Vol. 68, No. 8, pp. 1391–1401, 2019.

7. Hung, O.-N. and C.-W. Kan, "Effect of CO2 Laser Treatment on the Fabric Hand of Cotton and Cotton/Polyester Blended Fabric", *Polymers*, Vol. 9, No. 11, p. 609, 2017.

8. Padmanabham, G. and R. Bathe, "Laser Materials Processing for Industrial Ap-

plications", *Proceedings of the National Academy of Sciences, India Section A: Physical Sciences*, Vol. 88, No. 1, pp. 359–374, 2018.

9. Chen, T.-H., R. Fardel and C. B. Arnold, "Ultrafast Z-Scanning for High-Efficiency Laser Micro-Machining", *Light: Science & Applications*, Vol. 7, No. 4, pp. 17181–17181, 2018.

10. Bai, Z. and J. Wei, "Focusing Error Detection Based on Astigmatic Method with a Double Cylindrical Lens Group", *Optics & Laser Technology*, Vol. 106, No. 1, pp. 145–151, 2018.

11. Subbarao, M., T.-S. Choi and A. Nikzad, "Focusing Techniques", *Optical Engineering*, Vol. 32, No. 11, pp. 2824–2836, 1993.

12. Xu, S.-J., Y.-Z. Duan, Y.-H. Yu, Z.-N. Tian and Q.-D. Chen, "Machine Vision-Based High-Precision and Robust Focus Detection for Femtosecond Laser Machining", *Optics Express*, Vol. 29, No. 19, pp. 30952–30960, 2021.

13. Tarca, A. L., V. J. Carey, X.-w. Chen, R. Romero and S. Drăghici, "Machine Learning and its Applications to Biology", *PLoS Computational Biology*, Vol. 3, No. 6, p. e116, 2007.

14. Bedolla, E., L. C. Padierna and R. Castañeda-Priego, "Machine Learning for Condensed Matter Physics", *Journal of Physics: Condensed Matter*, Vol. 33, No. 5, p. 53001, 2020.

15. Xing, F., Y. Xie, H. Su, F. Liu and L. Yang, "Deep Learning in Microscopy Image Analysis: A Survey", *IEEE Transactions on Neural Networks and Learning Systems*, Vol. 29, No. 10, pp. 4550–4568, 2017.

16. LeCun, Y., L. Bottou, Y. Bengio and P. Haffner, "Gradient-Based Learning Applied to Document Recognition", *Proceedings of the IEEE*, Vol. 86, No. 11, pp. 2278–2324, 1998.

17. Gandarias, J. M., A. J. Garcia-Cerezo and J. M. Gomez-de Gabriel, "CNN-Based Methods for Object Recognition with High-Resolution Tactile Sensors", *IEEE Sensors Journal*, Vol. 19, No. 16, pp. 6872–6882, 2019.

18. Al-Amin, M., D. Z. Karim and T. A. Bushra, "Prediction of Rice Disease from Leaves Using Deep Convolution Neural Network Towards a Digital Agricultural System", *2019 22nd International Conference on Computer and Information Technology (ICCIT)*, pp. 1–5, Dhaka, 2019.

19. Ren, Z., Z. Xu and E. Y. Lam, "Autofocusing in Digital Holography Using Deep Learning", *Three-Dimensional and Multidimensional Microscopy: Image Acquisition and Processing XXV*, Vol. 10499, p. 104991V, San Francisco, 2018.

20. Pinkard, H., Z. Phillips, A. Babakhani, D. A. Fletcher and L. Waller, "Deep Learning for Single-Shot Autofocus Microscopy", *Optica*, Vol. 6, No. 6, pp. 794–797, 2019.

21. Nguyen, T., A. Thai, P. Adwani and G. Nehmetallah, "Autofocusing of Fluorescent Microscopic Images through Deep Learning Convolutional Neural Networks", *Digital Holography and Three-Dimensional Imaging*, pp. W3A–32, Bordeaux, 2019.

22. Wang, C., Q. Huang, M. Cheng, Z. Ma and D. J. Brady, "Deep Learning for Camera Autofocus", *IEEE Transactions on Computational Imaging*, Vol. 7, No. 1, pp. 258–271, 2021.

23. Li, F. and H. Jin, "A Fast Auto Focusing Method for Digital Still Camera", *2005 International Conference on Machine Learning and Cybernetics*, Vol. 8, pp. 5001–5005, Guangzhou, 2005.

24. Youngworth, R. N. and E. I. Betensky, "Fundamental Considerations for Zoom Lens Design (Tutorial)", *Zoom Lenses IV*, Vol. 8488, pp. 63 – 71, San Diego, 2012.

25. Santos, A., C. Ortiz de Solórzano, J. J. Vaquero, J. M. Pena, N. Malpica and F. del

Pozo, "Evaluation of Autofocus Functions in Molecular Cytogenetic Analysis", *Journal of Microscopy*, Vol. 188, No. 3, pp. 264–272, 1997.

26. Hansard, M., S. Lee, O. Choi and R. P. Horaud, *Time-of-Flight Cameras: Principles, Methods and Applications*, Springer Science & Business Media, London, 2012.

27. Wang, C., Q. Huang, M. Cheng, Z. Ma and D. J. Brady, "Deep Learning for Camera Autofocus", *IEEE Transactions on Computational Imaging*, Vol. 7, pp. 258–271, 2021.

28. Hochreiter, S. and J. Schmidhuber, "Long Short-Term Memory", *Neural Computation*, Vol. 9, No. 8, p. 1735–1780, 1997.

29. Siegman, A., *Lasers*, University Science Books, Sausalito, 1986.

30. Goodman, J. W., *Introduction to Fourier Optics*, McGraw-Hill, San Francisco, 1968.

31. Harris, C. R., K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke and T. E. Oliphant, "Array Programming with NumPy", *Nature*, Vol. 585, No. 7825, pp. 357–362, 2020.

32. Pedregosa, F., G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot and E. Duchesnay, "Scikit-Learn: Machine Learning in Python", *Journal of Machine Learning Research*, Vol. 12, No. 85, pp. 2825–2830, 2011.

33. Brea, L. S., "Diffractio, Python Module for Diffraction and Interference Optics",

2019, `https://pypi.org/project/diffractio/`, accessed on July 21, 2022.

34. Hunter, J. D., "Matplotlib: A 2D Graphics Environment", *Computing in Science & Engineering*, Vol. 9, No. 3, pp. 90–95, 2007.

35. Virtanen, P., R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt and SciPy 1.0 Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python", *Nature Methods*, Vol. 17, No. 3, pp. 261–272, 2020.

36. Papoulis, A., *The Fourier Integral and its Applications*, McGraw-Hill, New York, 1962.

37. Weber, H. J. and G. B. Arfken, *Essential Mathematical Methods for Physicists, ISE*, Elsevier, San Diego, 2003.

38. Peatross, J. and M. Ware, *Physics of Light and Optics*, Brigham Young University, Department of Physics Brigham, Provo, 2011.

39. Yariv, A. and P. Yeh, *Optical Waves in Crystals*, John Wiley & Sons, New York, 1984.

40. Saleh, B. E. and M. C. Teich, *Fundamentals of Photonics*, John Wiley & Sons, New York, 1991.

41. Ogilvy, J. A., "Wave Scattering from Rough Surfaces", *Reports on Progress in Physics*, Vol. 50, No. 12, pp. 1553–1608, 1987.

42. Ogilvy, J. and J. Foster, "Rough Surfaces: Gaussian or Exponential Statistics?", *Journal of Physics D: Applied Physics*, Vol. 22, No. 9, p. 1243, 1989.

43. Wu, J.-J., "Simulation of Rough Surfaces with FFT", *Tribology International*, Vol. 33, No. 1, pp. 47–58, 2000.

44. Lai, L. and E. Irene, "Area Evaluation of Microscopically Rough Surfaces", *Journal of Vacuum Science & Technology B: Microelectronics and Nanometer Structures Processing, Measurement, and Phenomena*, Vol. 17, No. 1, pp. 33–39, 1999.

45. Patrikar, R. M., "Modeling and Simulation of Surface Roughness", *Applied Surface Science*, Vol. 228, No. 1, pp. 213–220, 2004.

46. Ogilvy, J., "Computer Simulation of Acoustic Wave Scattering from Rough Surfaces", *Journal of Physics D: Applied Physics*, Vol. 21, No. 2, p. 260, 1988.

47. Thornton, B., "Effect of Surface Roughness on the Phase Change at Reflection in Interferometers", *JOSA*, Vol. 49, No. 5, pp. 476–479, 1959.

48. Pinel, N., C. Bourlier and J. Saillard, "Degree of Roughness of Rough Layers: Extensions of the Rayleigh Roughness Criterion and Some Applications", *Progress in Electromagnetics Research B*, Vol. 19, pp. 41–63, 2010.

49. Gonzalez, R. C., *Digital Image Processing*, Addison-Wesley, Reading, 1992.

50. Mohri, M., A. Rostamizadeh and A. Talwalkar, *Foundations of Machine Learning*, MIT Press, 2012.

51. McCullagh, P. and J. A. Nelder, *Generalized Linear Models*, Chapman & Hall, London, 1983.

52. Carleo, G., I. Cirac, K. Cranmer, L. Daudet, M. Schuld, N. Tishby, L. Vogt-Maranto and L. Zdeborová, "Machine Learning and the Physical Sciences", *Reviews of Modern Physics*, Vol. 91, No. 4, p. 045002, 2019.

53. Mills, B. and J. A. Grant-Jacob, "Lasers that Learn: The Interface of Laser Machining and Machine Learning", *IET Optoelectronics*, Vol. 15, No. 5, pp. 207–224,

2021.

54. Xie, Y., D. J. Heath, J. A. Grant-Jacob, B. S. Mackay, M. D. McDonnell, M. Praeger, R. W. Eason and B. Mills, "Deep Learning for the Monitoring and Process Control of Femtosecond Laser Machining", *Journal of Physics: Photonics*, Vol. 1, No. 3, p. 035002, 2019.

55. Pearson, K., "LIII. On lines and Planes of Closest Fit to Systems of Points in Space", *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, Vol. 2, No. 11, pp. 559–572, 1901.

56. Paszke, A., S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai and S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library", *Advances in Neural Information Processing Systems*, pp. 8024–8035, Vancouver, 2019.

57. Elahi, S., C. Polat, O. Safarzadeh and P. Elahi, "Noise Robust Focal Distance Detection in Laser Material Processing Using CNNs and Gaussian Processes", *Optics, Photonics and Digital Technologies for Imaging Applications VII*, Vol. 12138, p. 1213802, Strasbourg, 2022.

58. Polat, C., G. N. Yapıcı, S. Elahi and P. Elahi, "Noise Robust High Precision and Real-Time Focus Detection for Laser Micromanaging", *CLEO: Applications and Technology*, pp. AM3I–4, San Jose, 2022.

APPENDIX A: CALCULATION OF STANDARD DEVIATION CALCULATIONS FOR LSM MEASUREMENS

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

data = pd.read_csv('Copper 1.txt', sep="\t", header=None)
data.columns = ["distance", "value"]
copper1 = data["value"]
copper1=copper1.dropna()

copper1Std= copper1.std()
copper1stdValue = str(round(copper1Std,3))
copper1mean= copper1.mean()
copper1meanValue = str(round(copper1mean,3))

data = pd.read_csv('Copper 2.txt', sep="\t", header=None)
data.columns = ["distance", "value"]
copper2 = data["value"]
copper2=copper2.dropna()

copper2Std= copper2.std()
copper2stdValue = str(round(copper2Std,3))
copper2mean= copper2.mean()
copper2meanValue = str(round(copper2mean,3))

data = pd.read_csv('Copper 3.txt', sep="\t", header=None)
data.columns = ["distance", "value"]
copper3 = data["value"]
copper3=copper3.dropna()

copper3Std= copper3.std()
copper3stdValue = str(round(copper3Std,3))
copper3mean= copper3.mean()
copper3meanValue = str(round(copper3mean,3))
```

```
data = pd.read_csv('Silicon 1.txt', sep="\t", header=None)
data.columns = ["distance", "value"]
silicon1 = data["value"]
silicon1=silicon1.dropna()

silicon1Std= silicon1.std()
silicon1stdValue = str(round(silicon1Std,3))
silicon1mean= silicon1.mean()
silicon1meanValue = str(round(silicon1mean,3))

data = pd.read_csv('Silicon 2.txt', sep="\t", header=None)
data.columns = ["distance", "value"]
silicon2 = data["value"]
silicon2=silicon2.dropna()

silicon2Std= silicon2.std()
silicon2stdValue = str(round(silicon2Std,3))
silicon2mean= silicon2.mean()
silicon2meanValue = str(round(silicon2mean,3))

data = pd.read_csv('Silicon 3.txt', sep="\t", header=None)
data.columns = ["distance", "value"]
silicon3 = data["value"]
silicon3=silicon3.dropna()

silicon3Std= silicon3.std()
silicon3stdValue = str(round(silicon3Std,3))
silicon3mean= silicon3.mean()
silicon3meanValue = str(round(silicon3mean,3))

data = pd.read_csv('stainlessSteel 1.txt', sep="\t", header=None)
data.columns = ["distance", "value"]
steel1 = data["value"]
steel1=steel1.dropna()

steel1Std= steel1.std()
steel1stdValue = str(round(steel1Std,3))
```

```
steel1mean= steel1.mean()
steel1meanValue = str(round(steel1mean,3))

data = pd.read_csv('stainlessSteel 2.txt', sep="\t", header=None)
data.columns = ["distance", "value"]
steel2 = data["value"]
steel2=steel2.dropna()

steel2Std= steel2.std()
steel2stdValue = str(round(steel2Std,3))
steel2mean= steel2.mean()
steel2meanValue = str(round(steel2mean,3))

data = pd.read_csv('stainlessSteel 3.txt', sep="\t", header=None)
data.columns = ["distance", "value"]
steel3 = data["value"]
steel3=steel3.dropna()

steel3Std= steel3.std()
steel3stdValue = str(round(steel3Std,3))
steel3mean= steel3.mean()
steel3meanValue = str(round(steel3mean,3))
```

APPENDIX B: IMPLEMENTATION ON EXPERIMENTAL DATA

B.1. Required Python Libraries

```
numpy = 1.20.2
onnx = 1.11.0
torch = 1.10.2
torchvision = 0.11.3
onnx2pytorch = 0.4.1
torchinfo = 1.6.3
tqdm = 4.62.3
fastai = 2.5.3
pandas = 1.2.4
scikit-learn = 1.0.2
seaborn = 0.11.1
matplotlib = 3.3.4
Pillow = 9.0.1
joblib = 1.1.0
diffractio = 0.0.13
```

B.2. Code for Turning Experimental Videos Into Datasets

```
import torchvision.io
from torch.utils.data import Dataset, DataLoader

class VideoDataset(Dataset):
    def __init__(self, video_path, label, is_grayscale=True):
        self.label = label
        self.video_path = video_path
        self.video_tensor, _, _ = torchvision.io.read_video(video_path
    )
        if is_grayscale:
            self.video_tensor = self.video_tensor[:, :, :, 0].
    unsqueeze(1)
        self.video_tensor = 1.0 * self.video_tensor
        self.video_tensor = (self.video_tensor - self.video_tensor.min
```

```
    ()) / (
                        self.video_tensor.max() - self.video_tensor.min())

    def __len__(self):
        return len(self.video_tensor)

    def __getitem__(self, item):
        return self.video_tensor[item], self.label
```

B.3. Machine Learning Models Used for Experimental Data

```
import os
import numpy as np
import onnx
import torch.cuda
import torchvision.io
from onnx2pytorch import ConvertModel
from torch import nn, optim
from torchinfo import summary
from torch.utils.data import *
from tqdm import tqdm
from VideoDataset import VideoDataset
from model import FocusClassifier, FocusPredictor
import onnxruntime as ort

device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu'
    )
val_percentage = 0.15
test_percentage = 0.10
seed = 242344
batch_size = 16

def load_all_datasets(dataset_dir, predictor=False):
    video_files = os.listdir(dataset_dir)
    label_names = np.asarray([int(video_file[:video_file.find(".")])
   for video_file in video_files])
    if predictor:
        min_label, max_label = np.min(label_names), np.max(label_names
```

```
    )

    sorted_indices = np.argsort(label_names)
    label_names = label_names[sorted_indices]
    video_files = np.asarray(video_files)[sorted_indices]

    datasets = []
    for video_file in video_files:
        label = np.where(label_names == int(video_file[:video_file.
    find(".")]))[0].item()
        label = (label_names[label] - min_label) / (max_label -
    min_label) if predictor else label
        dataset = VideoDataset(os.path.join(dataset_dir, video_file),
    label)
        datasets.append(dataset)

    return label_names, datasets

def split_train_val(datasets, label_names, training_labels=None, seed=
    None):
    gen = torch.Generator().manual_seed(seed) if seed is not None else
    torch.Generator()

    # for training, only keep odd distances (-15, -13, ..., 13, 15)
    training_dsets = []
    other_dsets = []
    training_labels = label_names if training_labels is None else
    training_labels
    for i, label in enumerate(label_names):
        if label in training_labels:
            training_dsets.append(datasets[i])
        else:
            other_dsets.append(datasets[i])

    training_dsets = ConcatDataset(training_dsets)
    val_number = int(val_percentage * len(training_dsets))
    test_number = int(test_percentage * len(training_dsets))
    train_number = len(training_dsets) - val_number - test_number
```

```
    train_dset, val_dset, test_dset = random_split(training_dsets, [
train_number, val_number, test_number], gen)

    # split other datasets into val and test
    if len(other_dsets) > 0:
        other_dsets = ConcatDataset(other_dsets)
        valtest_percentage = val_percentage / (val_percentage +
test_percentage)
        val_number = int(valtest_percentage * len(other_dsets))
        test_number = len(other_dsets) - val_number
        val_dset_extra, test_dset_extra = random_split(other_dsets, [
val_number, test_number], gen)
        val_dset = ConcatDataset([val_dset, val_dset_extra])
        test_dset = ConcatDataset([test_dset, test_dset_extra])

    return train_dset, val_dset, test_dset

def get_onnx_session():
    # onnx_model = onnx.load("predictorNet.onnx")
    # onnx.checker.check_model(onnx_model)
    #
    # inputs = onnx_model.graph.input
    # name_to_input = {}
    # for input in inputs:
    #     name_to_input[input.name] = input
    #
    # for initializer in onnx_model.graph.initializer:
    #     if initializer.name in name_to_input:
    #         inputs.remove(name_to_input[initializer.name])
    #
    # onnx.save(onnx_model, "netPredictorV2.onnx")

    ort_sess = ort.InferenceSession('predictorNetV2.onnx', providers=[
"CUDAExecutionProvider"])

    return ort_sess

def test_mathematica_model():
```

```
    ort_sess = get_onnx_session()

    tensor, _, _ = torchvision.io.read_video("videos/02_03_videos/
Steel (10 micron)/0.avi")
    tensor = tensor[:, :, :, 0].unsqueeze(1) * 1.0
    tensor = (tensor - tensor.min()) / (tensor.max() - tensor.min())
    img = tensor[0].unsqueeze(0).numpy()

    input_name = ort_sess.get_inputs()[0].name
    output = ort_sess.run(None, {input_name: img})
    print(output)

def classifier_train():
    label_names, datasets = load_all_datasets("videos/02_03_videos/
Steel (10 micron)")
    training_labels = [label_names[i] for i in range(0, len(
label_names), 2)]
    print(f"Training using {training_labels}")

    train_dset, val_dset, test_dset = split_train_val(datasets,
label_names, label_names[0:1], seed)

    train_loader = DataLoader(train_dset, batch_size=batch_size,
shuffle=True, num_workers=4)
    val_loader = DataLoader(val_dset, batch_size=batch_size, shuffle=
False, num_workers=2)

    criterion = nn.CrossEntropyLoss()
    net = FocusClassifier(len(training_labels)).to(device)
    optimizer = optim.SGD(net.parameters(), lr=0.0005, momentum=0.93,
weight_decay=0.001)

    num_epochs = 10
    val_acc_arr = np.zeros(num_epochs)
    for epoch in tqdm(range(num_epochs)):
        train_model_one_epoch(net, train_loader, criterion, optimizer)

        correct = 0
```

```
        total = 0
        with torch.no_grad():
            for inputs, labels in val_loader:
                inputs = inputs.to(device)
                labels = labels.to(device)

                # calculate outputs by running images through the
    network
                outputs = net(inputs)

                # the class with the highest energy is what we choose
    as prediction
                _, predicted = torch.max(outputs.data, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()
            val_acc_arr[epoch] = correct / total
            # print(val_acc_arr[epoch])

    print(val_acc_arr)

def train_model_one_epoch(net, train_loader, criterion, optimizer):
    for inputs, labels in train_loader:
        inputs = inputs.to(device)
        labels = labels.to(device)
        if labels.dtype == torch.float64:
            labels = labels.float()
            labels = labels.unsqueeze(1)

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

if __name__ == '__main__':
```

```
 # test_mathematica_model()
 label_names, datasets = load_all_datasets("videos/02_03_videos/
Steel (10 micron)", predictor=True)
 max_label = np.max(label_names)
 min_label = np.min(label_names)
 training_labels = [label_names[i] for i in range(0, len(
label_names), 2)]
 print(f"Training using {training_labels}")

 train_dset, val_dset, test_dset = split_train_val(datasets,
label_names, training_labels, seed)

 train_loader = DataLoader(train_dset, batch_size=batch_size,
shuffle=True, num_workers=4)
 val_loader = DataLoader(val_dset, batch_size=1, shuffle=False,
num_workers=2)

 label_names = torch.from_numpy(label_names).to(device)

 ort_sess = get_onnx_session()
 total = correct = 0
 with torch.no_grad():
     for inputs, labels in tqdm(val_loader):
         # inputs = inputs.to(device)
         labels = labels.to(device)

         # calculate outputs by running images through the network
         outputs = torch.from_numpy(ort_sess.run(None, {"Input":
inputs.numpy()})[0]).to(device)

         # outputs = outputs * (max_label - min_label) + min_label
         pred_indices = torch.abs(outputs - label_names).argmin(dim
=1)
         pred_labels = label_names[pred_indices]

         labels = labels * (max_label - min_label) + min_label

         # the class with the highest energy is what we choose as
```

```
prediction
            total += pred_labels.size(0)
            correct += (pred_labels == labels).sum().item()
        acc = correct / total
    print(acc)

    criterion = nn.MSELoss()
    net = FocusPredictor().to(device)
    # optimizer = optim.SGD(net.parameters(), lr=0.0005, momentum
=0.93, weight_decay=0.001)
    optimizer = optim.Adam(net.parameters(), lr=0.0005)

    num_epochs = 10
    val_acc_arr = np.zeros(num_epochs)
    for epoch in tqdm(range(num_epochs)):
        train_model_one_epoch(net, train_loader, criterion, optimizer)
        correct = 0
        total = 0
        with torch.no_grad():
            for inputs, labels in val_loader:
                inputs = inputs.to(device)
                labels = labels.to(device)

                # calculate outputs by running images through the
network
                outputs = net(inputs)

                # rescale outputs
                outputs = outputs * (max_label - min_label) +
min_label
                pred_indices = torch.abs(outputs - label_names).argmin
(dim=1)
                pred_labels = label_names[pred_indices]

                labels = labels * (max_label - min_label) + min_label

                # the class with the highest energy is what we choose
as prediction
```

```
                total += pred_labels.size(0)
                correct += (pred_labels == labels).sum().item()
            val_acc_arr[epoch] = correct / total
            # print(val_acc_arr[epoch])

    print(val_acc_arr)
    # onnx_model = onnx.load("predictorNet.onnx")
    # pytorch_model = ConvertModel(onnx_model)
    # summary(pytorch_model)
    # print("sdfds")
```

B.4. Training of CNN Model on Experimental Data

```
import argparse
import pickle
from fastai.data.core import DataLoaders
from fastai.data.load import DataLoader
from model import FocusPredictor, FocusClassifier
from utils import *
directories = [os.path.join("training_videos", x) for x in ["Copper",
    "Steel", "Silicon"]]
parser = argparse.ArgumentParser(description='Train CNN to predict
    images on different sets of videos')
parser.add_argument('--bs', type=int, default=16, help='Batch size')
parser.add_argument('--num_epochs', type=int, default=5, help='Number
    of epochs')
parser.add_argument('--num_epochs', type=int, default=30, help='Number
     of epochs')
parser.add_argument('--val_p', type=float, default=0.15, help='
    Percentage of frames to use for validation')
parser.add_argument('--seed', type=int, default=423132, help='Seed
    used for splitting frames into train/val/test')
parser.add_argument('--directories', default=directories, nargs='+',
    help='Directories where video files are located. Each '
                                                                    '
    directory should contain videos at different focal '
                                                                    '
    distances and the name of each video file should be '
```

```
                                                                    'its
    focal distance (e.g., 150.avi)')
parser.add_argument('--results_dir', default="results_classifier",
    help='Directory where results will be saved')
parser.add_argument('--train_classifier', action='store_true')
parser.set_defaults(train_classifier=True)
args = parser.parse_args()
val_percentage = args.val_p
test_percentage = 0  # test on separate data
seed = args.seed
batch_size = args.bs
num_epochs = args.num_epochs
use_predictor = not args.train_classifier
if __name__ == '__main__':
    pred_str = "predictors" if use_predictor else "classifiers"
    print(f"Training different CNN {pred_str} for videos in {args.
    directories}")
    for directory in args.directories:
        result_dir = os.path.join(args.results_dir, os.path.split(
    directory)[-1])
        os.makedirs(result_dir, exist_ok=True)
        print(f"Loading videos in {directory}...")
        label_names_np, datasets = load_all_datasets(directory,
    predictor=use_predictor)
        max_label = np.max(label_names_np)
        min_label = np.min(label_names_np)
        # only train on every other label including beginning and end
        if use_predictor:
            train_label_indices = [0] + list(range(2, len(
    label_names_np) - 1, 2)) + [-1]
            training_labels = label_names_np[train_label_indices]
        else:
            training_labels = label_names_np
        print(f"Training using videos {training_labels}...")
        train_dset, val_dset, _ = split_train_val(datasets,
    label_names_np, val_percentage, test_percentage, training_labels,
    seed)
        label_names = torch.from_numpy(label_names_np).long().to(
```

```
    device)
        train_loader = DataLoader(train_dset, batch_size=batch_size,
    shuffle=True, num_workers=4, pin_memory=True)
        val_loader = DataLoader(val_dset, batch_size=512, shuffle=
    False, num_workers=2, pin_memory=True)
        dls = DataLoaders(train_loader, val_loader)
        if use_predictor:
            net = FocusPredictor()
            train_predictor(net, dls, label_names, num_epochs,
    result_dir)
        else:
            net = FocusClassifier(len(label_names_np))
            train_classifier(net, dls, num_epochs, result_dir)
        pickle.dump(label_names_np, open(os.path.join(result_dir, "
    label_names.pkl"), "wb"))
        # test on testing data
        # test_preds, test_targets = predict_labels(net, test_loader,
    label_names, use_predictor)
        # pickle.dump((test_preds, test_targets, label_names_np), open
    (os.path.join(result_dir, "test_pred_targets.pkl"), "wb"))
```

B.5. Training of Non-CNN Models on Experimental Data

```
import argparse
import pickle
import sklearn.svm
from fastai.data.core import DataLoaders
from fastai.data.load import DataLoader
from sklearn.decomposition import PCA
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from model import FocusPredictor, FocusClassifier
from utils import *

directories = [os.path.join("training_videos", x) for x in ["Copper",
```

```
    "Steel", "Silicon"]]
parser = argparse.ArgumentParser(description='Train CNN to predict
    images on different sets of videos')
parser.add_argument('--val_p', type=float, default=0.15, help='
    Percentage of frames to use for validation')
parser.add_argument('--seed', type=int, default=423132, help='Seed
    used for splitting frames into train/val/test')
parser.add_argument('--pca_var_explained', type=float, default=0.90,
    help='Percentage of variance to be explained by PCA components.')

parser.add_argument('--directories', default=directories, nargs='+',
                    help='Directories where video files are located.
    Each '
                         'directory should contain videos at different
     focal '
                         'distances and the name of each video file
    should be '
                         'its focal distance (e.g., 150.avi)')
parser.add_argument('--results_dir', default="results_scikit", help='
    Directory where results will be saved')

models = [LogisticRegression(max_iter=1000), SVC()]
model_names = ["lr", "svc"]

args = parser.parse_args()

val_percentage = args.val_p
test_percentage = 0  # test on separate data
seed = args.seed

if __name__ == '__main__':
    print(f"Training different {model_names} classifiers for videos in
    {args.directories}")
    for directory in args.directories:
        result_dir = os.path.join(args.results_dir, os.path.split(
    directory)[-1])
        trained_models_dir = os.path.join(result_dir, "models")
        os.makedirs(trained_models_dir, exist_ok=True)
```

```
        print(f"Loading videos in {directory}...")
        label_names, datasets = load_all_datasets(directory, predictor
    =False)
        max_label = np.max(label_names)
        min_label = np.min(label_names)

        train_dset, val_dset, _ = split_train_val(datasets,
    label_names, val_percentage, test_percentage,
                                                  label_names, seed)

        for model, model_name in zip(models, model_names):
            print(f"Training {model}...")
            pca = PCA(n_components=args.pca_var_explained)
            scaler = StandardScaler()
            pipeline = Pipeline(steps=[("scaler", scaler), ("pca", pca
    ), ("model", model)], memory="scikit_cache")

            x_arr = np.asarray([x[0].numpy() for x in train_dset]).
    reshape(len(train_dset), -1)
            y_arr = label_names[np.asarray([x[1] for x in train_dset])
    ]
            pipeline.fit(x_arr, y_arr)
            print(f"Using {pipeline['pca'].n_components_} for pca")
            pickle.dump(pipeline, open(os.path.join(trained_models_dir
    , f"{model_name}.pkl"), "wb"))

        pickle.dump(label_names, open(os.path.join(result_dir, "
    label_names.pkl"), "wb"))
```

B.6. Testing of CNN Model on Experimental Data

```
import argparse
import pickle
from glob import glob
import matplotlib.pyplot as plt
import pandas as pd
import sklearn.metrics
```

```
from fastai.data.block import DataBlock, RegressionBlock,
    CategoryBlock
from fastai.data.core import DataLoaders
from fastai.data.load import DataLoader
from fastai.data.transforms import get_image_files, RandomSplitter,
    IntToFloatTensor
from fastai.vision.core import PILImageBW
from fastai.vision.data import ImageBlock
from matplotlib.ticker import StrMethodFormatter
from sklearn.metrics import *
from model import FocusClassifier
from test_cnn import plot_cm_matrix
from utils import *

parser = argparse.ArgumentParser(description='Plot confusion matrices
    and report accuracies of trained models on test '
                                                 'data. train_cnn_on_synth
    .py must be run before')
parser.add_argument('--synth_imgs_dir', default="synth_images_testing"
    ,
                    help='Directory where generated synthetic images
    for '
                         'testing from gen_synth_images.py are located
    .')
parser.add_argument('--results_dir', default="results_synth_classifier
    ", help='Directory where results will be saved')

args = parser.parse_args()

def get_label_from_fname(fname):
    return float(fname.parts[-2].split("_")[0])

def get_df_from_synth_img_dir(img_dir):
    """
    Returns a pandas DataFrame with containing focus and noise info of
    each img folder
    """
    folder_paths = glob(os.path.join(img_dir, "*/"), recursive=False)
```

```
    row_list = []
    for folder_path in folder_paths:
        folder_name = os.path.split(folder_path)[0].split("/")[-1]
        row = [float(x) for x in folder_name.split("_")[:-1]] + [
    folder_name]
        row_list.append(row)

    return pd.DataFrame(row_list, columns=["focus", "int_noise", "
    ext_noise", "path"])

def plot_noise_acc(acc_mat, int_noise_arr, ext_noise_arr, file_name):
    fig = plt.figure(figsize=(6.4 * 1.1, 4.8 * 1.1))
    # disp.plot(include_values=False, ax=fig.gca())
    int_noise_arr = np.round(int_noise_arr * 100) / 100.0  # hacky way
    to format .2f
    ext_noise_arr = np.round(ext_noise_arr * 100) / 100.0
    df_cm = pd.DataFrame(acc_mat, index=int_noise_arr, columns=
    ext_noise_arr)
    sns.heatmap(df_cm, cmap="flare", linewidths=.5, square=True, vmin
    =0, vmax=1, annot=True, fmt=".2f")
    plt.xlabel("Output noise std.")
    plt.ylabel("Input noise std.")
    plt.tight_layout()
    if file_name is not None:
        plt.savefig(file_name, bbox_inches='tight', pad_inches=0.02)

if __name__ == '__main__':
    result_dir = args.results_dir
    os.makedirs(result_dir, exist_ok=True)

    df = get_df_from_synth_img_dir(args.synth_imgs_dir)

    int_noise_arr = np.unique(df["int_noise"].values)
    ext_noise_arr = np.unique(df["ext_noise"].values)

    num_classes = len(np.unique(df["focus"].values))

    net = FocusClassifier(num_classes, img_size=(48, 48)).to(device)
```

```
 net.load_state_dict(torch.load(os.path.join(result_dir, "models/
trainedModel.pth")))

 acc_mat = np.zeros((len(int_noise_arr), len(ext_noise_arr)))
 for i, int_noise in enumerate(int_noise_arr):
     for j, ext_noise in enumerate(ext_noise_arr):
         folders = list(
             df[np.logical_and(df["int_noise"] == int_noise, df["
ext_noise"] == ext_noise)]["path"].values)
         get_items_fun = lambda path: get_image_files(path, folders
=folders)
         dblock = DataBlock(blocks=(ImageBlock(cls=PILImageBW),
CategoryBlock),
                            get_items=get_items_fun,
                            get_y=get_label_from_fname,
                            item_tfms=IntToFloatTensor(),
                            splitter=RandomSplitter(valid_pct=0))
         test_loader = dblock.dataloaders(args.synth_imgs_dir,
batch_size=64, shuffle=True, num_workers=4,
                                          pin_memory=True, device=
device)[0]
         label_names = torch.tensor(test_loader.vocab.items).to(
device)
         label_names_np = label_names.cpu().numpy().astype(int)
         test_preds, test_targets, avg_time_taken =
predict_labels_net(net, test_loader, label_names, False)
         acc = accuracy_score(test_targets, test_preds)
         acc_mat[i, j] = acc

 plot_noise_acc(acc_mat, int_noise_arr, ext_noise_arr, os.path.join
(result_dir, "noise_acc.pdf"))

 dblock = DataBlock(blocks=(ImageBlock(cls=PILImageBW),
CategoryBlock),
                    get_items=get_image_files,
                    get_y=get_label_from_fname,
                    item_tfms=IntToFloatTensor(),
                    splitter=RandomSplitter(valid_pct=0))
```

```
    test_loader = dblock.dataloaders(args.synth_imgs_dir, batch_size
    =256, shuffle=True, num_workers=4, pin_memory=True,
                                        device=device)[0]
    label_names = torch.tensor(test_loader.vocab.items).to(device)
    label_names_np = label_names.cpu().numpy().astype(int)
    print(f"Loaded focal distances (classes) are {label_names_np}")
    temp_batch = next(iter(test_loader))
    net = FocusClassifier(len(test_loader.vocab), img_size=temp_batch
    [0].size()[-2:]).to(device)
    net.load_state_dict(torch.load(os.path.join(result_dir, "models/
    trainedModel.pth")))

    test_preds, test_targets, avg_time_taken = predict_labels_net(net,
    test_loader, label_names, False)
    class_report_txt = classification_report(test_targets, test_preds,
    target_names=label_names_np.astype(str))
    print(class_report_txt)
    with open(os.path.join(result_dir, "testing_report.txt"), 'w') as
    f:
        f.write(class_report_txt)

    class_report_dict = classification_report(test_targets, test_preds
    , target_names=label_names_np.astype(str),
                                              output_dict=True)
    with open(os.path.join(result_dir, "class_report_dict.pkl"), 'wb')
    as f:
        pickle.dump(class_report_dict, f)

    plot_cm_matrix(test_targets, test_preds, label_names_np, os.path.
    join(result_dir, "testing_cm.pdf"))
    plt.show()
```

B.7. Testing of Non-CNN Models on Experimental Data

```
import argparse
import os
import pickle
import time
```

```
import pandas as pd
import sklearn.metrics
from fastai.data.load import DataLoader
from matplotlib import pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay,
    classification_report
from utils import *
from model import FocusClassifier, FocusPredictor
sns.set_theme(style="white")

tex_fonts = {
    # Use LaTeX to write all text
    "text.usetex": True,
    "font.serif": 'Times New Roman',
    # # Use 10pt font in plots, to match 10pt font in document
    "axes.labelsize": 16,
    "font.size": 16,
    # # Make the legend/label fonts a little smaller
    # "legend.fontsize": 11,
    "legend.fontsize": 8,
    "xtick.labelsize": 16,
    "ytick.labelsize": 16
}
plt.rcParams.update(tex_fonts)
folder_naems = ["Copper", "Steel", "Silicon"]
directories = [os.path.join("testing_videos", x) for x in folder_naems
    ]
results_directories = [os.path.join("results_scikit", x) for x in
    folder_naems]

parser = argparse.ArgumentParser(description='Plot confusion matrices
    and report accuracies of trained models on test '
                                                'data. train_cnn.py must
    be run before')
parser.add_argument('--results_directories', nargs='+', default=
    results_directories,
                    help='Directory where results are saved')
```

```
parser.add_argument('--directories', default=directories, nargs='+',
                    help='Directories where video files are located
    for testing. Each '
                         'directory should contain videos at different
     focal '
                         'distances and the name of each video file
    should be '
                         'its focal distance (e.g., 150.avi)')

args = parser.parse_args()
model_names = ["lr", "svc"]

def plot_cm_matrix(targets, predictions, label_names, file_name=None):
     cm = confusion_matrix(targets, predictions, labels=label_names,
    normalize='true')
     disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=
    label_names)

     fig = plt.figure(figsize=(6.4 * 1.1, 4.8 * 1.1))
     # disp.plot(include_values=False, ax=fig.gca())
     df_cm = pd.DataFrame(cm, index=label_names, columns=label_names)
     sns.heatmap(df_cm, cmap="flare", linewidths=.5, square=True)
     plt.xlabel("Predicted label")
     plt.ylabel("True label")
     plt.tight_layout()
     if file_name is not None:
          plt.savefig(file_name, bbox_inches='tight', pad_inches=0.02)

if __name__ == '__main__':
     results_directories = args.results_directories
     testing_directories = args.directories
     for directory, result_dir in zip(testing_directories,
    results_directories):
          label_names = pickle.load(open(os.path.join(result_dir, "
    label_names.pkl"), "rb"))

          print(f"Loading videos in {directory}...")
          label_names_testing, datasets = load_all_datasets(directory,
```

```
predictor=False)
    assert np.all(label_names_testing == label_names)

    testing_dset = ConcatDataset(datasets)
    x_arr = np.asarray([x[0].numpy() for x in testing_dset]).
reshape(len(testing_dset), -1)
    test_targets = label_names[np.asarray([x[1] for x in
testing_dset])]
    for model_name in model_names:
        print(f"\nTesting {model_name}...")
        scikit_model = pickle.load(open(os.path.join(result_dir, "
models", f"{model_name}.pkl"), "rb"))
        starting_time = time.time()
        test_preds = scikit_model.predict(x_arr)
        time_taken = (time.time() - starting_time) / len(x_arr)
        print(f"{model_name} processed {1 / time_taken:.2f} images
/sec when using {device}")
        class_report_txt = classification_report(test_targets,
test_preds, target_names=label_names.astype(str))
        print(class_report_txt)
        with open(os.path.join(result_dir, f"{model_name}
_testing_report.txt"), 'w') as f:
            f.write(class_report_txt)

        class_report_dict = classification_report(test_targets,
test_preds, target_names=label_names.astype(str),
                                                  output_dict=True
)
        with open(os.path.join(result_dir, f"{model_name}
_class_report_dict.pkl"), 'wb') as f:
            pickle.dump(class_report_dict, f)

        plot_cm_matrix(test_targets, test_preds, label_names,
                       os.path.join(result_dir, f"{model_name}
_testing_cm.pdf"))
```

APPENDIX C: SIMULATION OF LASER MACHINING SETUP

```
import copy
import itertools
import multiprocessing
import os.path
import sys
import PIL.Image
import numpy as np
from PIL import Image

# temporarily block printing when importing diffractio
sys.stdout = open(os.devnull, 'w')
from diffractio import degrees, mm, um
from diffractio.scalar_masks_XY import Scalar_mask_XY
from diffractio.scalar_sources_XY import Scalar_source_XY
sys.stdout = sys.__stdout__

from joblib import Parallel, delayed
from tqdm import tqdm
import h5py

"""
This script generates images with given internal and external noise
    stds and resizes and saves them in the given
save directory. It uses multiple cores/threads in parallel to speed up
    the simulations.
"""

saved_img_size = (50, 50)
num_samples = 1100
save_dir = "synth_images_11_mm"
num_imgs_per_config = 200
num_procs_to_use = 12

x0 = np.linspace(-500 * um, 500 * um, num_samples)
```

```
y0 = np.linspace(-500 * um, 500 * um, num_samples)

focal = 11 * mm
focal2 = 150 * mm
diameter = 5.5 * mm
dia2 = 1 * mm
d = 300 * mm
raylen = 150 * um
rayleigh_arr = np.arange(-raylen, raylen + 0.00001, raylen / 5)  #
    Modify the stepsize by this
wavelength = 0.976 * um

og_u0 = Scalar_source_XY(x=x0, y=y0, wavelength=wavelength)
og_u0.gauss_beam(A=1, r0=(0, 0), z0=0 * mm, w0=6.85 * um, theta=0 *
    degrees)

# initialize lenses
t0 = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength)
t0.lens(r0=(0.0, 0.0), radius=diameter / 2, focal=focal, mask=True)
t1 = Scalar_mask_XY(x=x0, y=y0, wavelength=wavelength)
t1.lens(r0=(0.0, 0.0), radius=dia2 / 2, focal=focal2, mask=True)
t3 = Scalar_mask_XY(x0, y0, wavelength)

# corr_len_arr = np.asarray([50, 125, 250, 500])
# internal_noise_stds = np.asarray([0, 0.25, 0.5, 0.75])
# external_noise_stds = np.asarray([0, 0.01, 0.03, 0.05])
corr_len_arr = np.linspace(50, 500, 10)
internal_noise_stds = np.linspace(0, 0.50, 10)
external_noise_stds = np.linspace(0, 0.05, 10)

def get_folder_path(rayleigh, corr_len, int_noise, ext_noise):
    return os.path.join(save_dir, f"{round(rayleigh, 1)}_{corr_len}_{
    int_noise}_{ext_noise}_imgs")

def gen_imgs():
    """
    Generates an ndarray containing image data after simulation for
    each rayleigh length, int. noise, correlation length,
```

```
 and ext. noise.
 :return: Image ndarray of size (len(rayleigh_arr), len(
internal_noise_stds), len(corr_len_arr), len(external_noise_stds))
 + saved_img_size
 """

 imgs_arr = np.zeros(
     (len(rayleigh_arr), len(internal_noise_stds), len(corr_len_arr
), len(external_noise_stds)) + saved_img_size,
     np.uint8)
 for i, rayleigh in enumerate(rayleigh_arr):
     for j, int_noise in enumerate(internal_noise_stds):
         for k, corr_len in enumerate(corr_len_arr):
             u0 = copy.deepcopy(og_u0)
             t3.roughness(t=(corr_len, corr_len), s=int_noise)
             u0 = u0 * t3

             z0 = focal + rayleigh  # Initial wave moving towards
the first lens
             u0 = u0.RS(z=z0, verbose=False, new_field=True)

             u1 = u0 * t0  # After first lens

             u2 = u1.RS(z=d, verbose=False, new_field=True)  #
Moving d distance

             u4 = u2 * t1  # After second lens
             u5 = u4.RS(z=focal2, verbose=False, new_field=True)  #
 After being focused on the camera

             intensity = np.abs(u5.u) ** 2  # Here is where we take
 the abs square of the electric field (u5.u) in

             for l, ext_noise in enumerate(external_noise_stds):
                 ext_noise_arr = np.random.normal(0, ext_noise, (
num_samples, num_samples))

                 i_w_ext_noise = (intensity - intensity.min()) / (
```

```
    intensity.max() - intensity.min()) + ext_noise_arr
                    img_data = ((i_w_ext_noise - i_w_ext_noise.min())
    / (
                                i_w_ext_noise.max() - i_w_ext_noise.min())
     * 255).astype(np.uint8)
                    img = Image.fromarray(img_data, "L")
                    img_data_resized = np.asarray(img.resize(
    saved_img_size, PIL.Image.NEAREST))
                    imgs_arr[i, j, k, l] = img_data_resized

    return imgs_arr

if __name__ == '__main__':
    os.makedirs(save_dir, exist_ok=True)
    info_text = (f"Focus distances: {rayleigh_arr}\n"
                 f"Correlation lengths: {corr_len_arr}\n"
                 f"Internal noises: {internal_noise_stds}\n"
                 f"External noises: {external_noise_stds}")
    print(info_text)
    with open(os.path.join(save_dir, "info.txt"), 'w') as f:
        f.write(info_text)

    print("Creating HDF5 datasets for each Rayleigh length...")
    for rayleigh in tqdm(rayleigh_arr):
        f_list = []
        for rayleigh in rayleigh_arr:
            f = h5py.File(os.path.join(save_dir, f"rayleigh_{rayleigh
    :.0f}.hdf5"), "w")
            f_list.append(f)

            # create HDF5 dataset for each correlation, int. noise,
    ext. noise triplet
            for i, int_noise in enumerate(internal_noise_stds):
                for j, corr_len in enumerate(corr_len_arr):
                    for k, ext_noise in enumerate(external_noise_stds)
    :
                        dset = f.create_dataset(f"{i}_{j}_{k}", (
    num_imgs_per_config,) + saved_img_size, np.uint8)
```

```
                    dset.attrs["rayleigh"] = rayleigh
                    dset.attrs["corr_len"] = corr_len
                    dset.attrs["int_noise"] = int_noise
                    dset.attrs["ext_noise"] = ext_noise

 total_num_imgs = len(rayleigh_arr) * len(internal_noise_stds) *
len(corr_len_arr) * len(external_noise_stds) \
                  * num_imgs_per_config

 print(
     f"Generating {num_imgs_per_config} images per config ({
total_num_imgs} images in total) using {num_procs_to_use} parallel
 processes...")

 img_idx_arr = np.arange(num_imgs_per_config)
 img_idx_chunks = [img_idx_arr[i:i + num_procs_to_use] for i in
range(0, len(img_idx_arr), num_procs_to_use)]
 try:
     with Parallel(n_jobs=num_procs_to_use) as parallel:
         with tqdm(total=num_imgs_per_config) as pbar:
             for img_idx_chunk in img_idx_chunks:
                 # use multi-processing to generate images
                 imgs_arr_list = parallel(delayed(gen_imgs)() for _
 in img_idx_chunk)
                 # write to H5 dataset
                 for imgs_arr, img_idx in zip(imgs_arr_list,
img_idx_chunk):
                     for i in range(len(rayleigh_arr)):
                         for j in range(len(internal_noise_stds)):
                             for k in range(len(corr_len_arr)):
                                 for l in range(len(
external_noise_stds)):
                                     f_list[i][f"{j}_{k}_{l}"][
img_idx] = imgs_arr[i, j, k, l]
                 pbar.update(len(img_idx_chunk))
 finally:
     for f in f_list:
         f.close()
```

APPENDIX D: IMPLEMENTATION ON SIMULATED DATA

D.1. Train and Test Splitting of Generated Data

```
import glob
import os.path
import numpy as np
import h5py
from tqdm import tqdm

h5_files_dir = "synth_images_11_mm"
val_split = 0.15
test_split = 0.15
seed = 542646

if __name__ == '__main__':

    rng = np.random.default_rng(seed)
    train_dir = os.path.join(h5_files_dir, "train")
    val_dir = os.path.join(h5_files_dir, "val")
    test_dir = os.path.join(h5_files_dir, "test")
    for dir in [train_dir, val_dir, test_dir]:
        os.makedirs(dir, exist_ok=True)

    for h5_path in tqdm(glob.glob(os.path.join(h5_files_dir, "*.hdf5")
   )):
        h5_file_name = os.path.split(h5_path)[-1]
        f = h5py.File(h5_path, "r")
        train_f = h5py.File(os.path.join(train_dir, h5_file_name), "w"
   )
        val_f = h5py.File(os.path.join(val_dir, h5_file_name), "w")
        test_f = h5py.File(os.path.join(test_dir, h5_file_name), "w")

        for key in f:
            # calculate train,val,test indices
            num_val, num_test = round(val_split * len(f[key])), round(
```

```
    test_split * len(f[key]))
            num_train = len(f[key]) - num_test - num_val

            indices = set(range(len(f[key])))
            val_indices = rng.choice(len(f[key]), num_val, replace=
    False)
            test_indices = rng.choice(list(indices - set(val_indices))
    , num_test, replace=False)
            train_indices = list(indices - set(val_indices) - set(
    test_indices))

            # add data to train,val,test h5
            train_dset = train_f.create_dataset(key, dtype=np.uint8,
    data=f[key][:][train_indices])
            val_dset = val_f.create_dataset(key, dtype=np.uint8, data=
    f[key][:][val_indices])
            test_dset = test_f.create_dataset(key, dtype=np.uint8,
    data=f[key][:][test_indices])

            for dset in [train_dset, val_dset, test_dset]:
                dset.attrs.update(f[key].attrs)

        for h5file in [f, train_f, val_f, test_f]:
            h5file.close()
     print("Done")
```

D.2. Machine Learning Models for Simulation

```
import os
import numpy as np
import onnx
import torch.cuda
import torchvision.io
from onnx2pytorch import ConvertModel
from torch import nn, optim
from torchinfo import summary
from torch.utils.data import *
from tqdm import tqdm
```

```
from VideoDataset import VideoDataset
from model import FocusClassifier, FocusPredictor
import onnxruntime as ort

device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu'
    )
val_percentage = 0.15
test_percentage = 0.10
seed = 242344
batch_size = 16

def load_all_datasets(dataset_dir, predictor=False):
    video_files = os.listdir(dataset_dir)
    label_names = np.asarray([int(video_file[:video_file.find(".")])
    for video_file in video_files])
    if predictor:
        min_label, max_label = np.min(label_names), np.max(label_names
    )

    sorted_indices = np.argsort(label_names)
    label_names = label_names[sorted_indices]
    video_files = np.asarray(video_files)[sorted_indices]

    datasets = []
    for video_file in video_files:
        label = np.where(label_names == int(video_file[:video_file.
    find(".")]))[0].item()
        label = (label_names[label] - min_label) / (max_label -
    min_label) if predictor else label
        dataset = VideoDataset(os.path.join(dataset_dir, video_file),
    label)
        datasets.append(dataset)

    return label_names, datasets

def split_train_val(datasets, label_names, training_labels=None, seed=
    None):
    gen = torch.Generator().manual_seed(seed) if seed is not None else
```

```
    torch.Generator()

    # for training, only keep odd distances (-15, -13, ..., 13, 15)
    training_dsets = []
    other_dsets = []
    training_labels = label_names if training_labels is None else
training_labels
    for i, label in enumerate(label_names):
        if label in training_labels:
            training_dsets.append(datasets[i])
        else:
            other_dsets.append(datasets[i])

    training_dsets = ConcatDataset(training_dsets)
    val_number = int(val_percentage * len(training_dsets))
    test_number = int(test_percentage * len(training_dsets))
    train_number = len(training_dsets) - val_number - test_number
    train_dset, val_dset, test_dset = random_split(training_dsets, [
train_number, val_number, test_number], gen)

    # split other datasets into val and test
    if len(other_dsets) > 0:
        other_dsets = ConcatDataset(other_dsets)
        valtest_percentage = val_percentage / (val_percentage +
test_percentage)
        val_number = int(valtest_percentage * len(other_dsets))
        test_number = len(other_dsets) - val_number
        val_dset_extra, test_dset_extra = random_split(other_dsets, [
val_number, test_number], gen)
        val_dset = ConcatDataset([val_dset, val_dset_extra])
        test_dset = ConcatDataset([test_dset, test_dset_extra])

    return train_dset, val_dset, test_dset

def get_onnx_session():
    # onnx_model = onnx.load("predictorNet.onnx")
    # onnx.checker.check_model(onnx_model)
    #
```

```
    # inputs = onnx_model.graph.input
    # name_to_input = {}
    # for input in inputs:
    #     name_to_input[input.name] = input
    #
    # for initializer in onnx_model.graph.initializer:
    #     if initializer.name in name_to_input:
    #         inputs.remove(name_to_input[initializer.name])
    #
    # onnx.save(onnx_model, "netPredictorV2.onnx")

    ort_sess = ort.InferenceSession('predictorNetV2.onnx', providers=[
"CUDAExecutionProvider"])

    return ort_sess

def test_mathematica_model():
    ort_sess = get_onnx_session()

    tensor, _, _ = torchvision.io.read_video("videos/02_03_videos/
Steel (10 micron)/0.avi")
    tensor = tensor[:, :, :, 0].unsqueeze(1) * 1.0
    tensor = (tensor - tensor.min()) / (tensor.max() - tensor.min())
    img = tensor[0].unsqueeze(0).numpy()

    input_name = ort_sess.get_inputs()[0].name
    output = ort_sess.run(None, {input_name: img})
    print(output)

def classifier_train():
    label_names, datasets = load_all_datasets("videos/02_03_videos/
Steel (10 micron)")
    training_labels = [label_names[i] for i in range(0, len(
label_names), 2)]
    print(f"Training using {training_labels}")

    train_dset, val_dset, test_dset = split_train_val(datasets,
label_names, label_names[0:1], seed)
```

```
    train_loader = DataLoader(train_dset, batch_size=batch_size,
shuffle=True, num_workers=4)
    val_loader = DataLoader(val_dset, batch_size=batch_size, shuffle=
False, num_workers=2)

    criterion = nn.CrossEntropyLoss()
    net = FocusClassifier(len(training_labels)).to(device)
    optimizer = optim.SGD(net.parameters(), lr=0.0005, momentum=0.93,
weight_decay=0.001)

    num_epochs = 10
    val_acc_arr = np.zeros(num_epochs)
    for epoch in tqdm(range(num_epochs)):
        train_model_one_epoch(net, train_loader, criterion, optimizer)

        correct = 0
        total = 0
        with torch.no_grad():
            for inputs, labels in val_loader:
                inputs = inputs.to(device)
                labels = labels.to(device)

                # calculate outputs by running images through the
network
                outputs = net(inputs)

                # the class with the highest energy is what we choose
as prediction
                _, predicted = torch.max(outputs.data, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()
            val_acc_arr[epoch] = correct / total
            # print(val_acc_arr[epoch])

    print(val_acc_arr)

def train_model_one_epoch(net, train_loader, criterion, optimizer):
```

```
    for inputs, labels in train_loader:
        inputs = inputs.to(device)
        labels = labels.to(device)
        if labels.dtype == torch.float64:
            labels = labels.float()
            labels = labels.unsqueeze(1)

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

if __name__ == '__main__':
    # test_mathematica_model()
    label_names, datasets = load_all_datasets("videos/02_03_videos/
    Steel (10 micron)", predictor=True)
    max_label = np.max(label_names)
    min_label = np.min(label_names)
    training_labels = [label_names[i] for i in range(0, len(
    label_names), 2)]
    print(f"Training using {training_labels}")

    train_dset, val_dset, test_dset = split_train_val(datasets,
    label_names, training_labels, seed)

    train_loader = DataLoader(train_dset, batch_size=batch_size,
    shuffle=True, num_workers=4)
    val_loader = DataLoader(val_dset, batch_size=1, shuffle=False,
    num_workers=2)

    label_names = torch.from_numpy(label_names).to(device)

    ort_sess = get_onnx_session()
    total = correct = 0
```

```
 with torch.no_grad():
     for inputs, labels in tqdm(val_loader):
         # inputs = inputs.to(device)
         labels = labels.to(device)

         # calculate outputs by running images through the network
         outputs = torch.from_numpy(ort_sess.run(None, {"Input":
inputs.numpy()})[0]).to(device)

         # outputs = outputs * (max_label - min_label) + min_label
         pred_indices = torch.abs(outputs - label_names).argmin(dim
=1)
         pred_labels = label_names[pred_indices]

         labels = labels * (max_label - min_label) + min_label

         # the class with the highest energy is what we choose as
prediction
         total += pred_labels.size(0)
         correct += (pred_labels == labels).sum().item()
     acc = correct / total
 print(acc)

 criterion = nn.MSELoss()
 net = FocusPredictor().to(device)
 # optimizer = optim.SGD(net.parameters(), lr=0.0005, momentum
=0.93, weight_decay=0.001)
 optimizer = optim.Adam(net.parameters(), lr=0.0005)

 num_epochs = 10
 val_acc_arr = np.zeros(num_epochs)
 for epoch in tqdm(range(num_epochs)):
     train_model_one_epoch(net, train_loader, criterion, optimizer)
     correct = 0
     total = 0
     with torch.no_grad():
         for inputs, labels in val_loader:
             inputs = inputs.to(device)
```

```
                labels = labels.to(device)

                # calculate outputs by running images through the
    network
                outputs = net(inputs)

                # rescale outputs
                outputs = outputs * (max_label - min_label) +
    min_label
                pred_indices = torch.abs(outputs - label_names).argmin
    (dim=1)
                pred_labels = label_names[pred_indices]

                labels = labels * (max_label - min_label) + min_label

                # the class with the highest energy is what we choose
    as prediction
                total += pred_labels.size(0)
                correct += (pred_labels == labels).sum().item()
            val_acc_arr[epoch] = correct / total
            # print(val_acc_arr[epoch])

    print(val_acc_arr)
    # onnx_model = onnx.load("predictorNet.onnx")
    # pytorch_model = ConvertModel(onnx_model)
    # summary(pytorch_model)
    # print("sdfds")
```

D.3. Training of CNN Model for Simulation

```
import argparse
import os
import pickle
import pandas as pd
import sklearn.metrics
from fastai.data.load import DataLoader
import seaborn as sns
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay,
```

```
    classification_report
from utils import *

from model import FocusClassifier, FocusPredictor

folder_names = ["Copper", "Steel", "Silicon"]
directories = [os.path.join("testing_videos", x) for x in folder_names
    ]
results_directories = [os.path.join("results_predictor", x) for x in
    folder_names]

parser = argparse.ArgumentParser(description='Plot confusion matrices
    and report accuracies of trained models on test '
                                             'data. train_cnn.py must
    be run before')
parser.add_argument('--results_directories', nargs='+', default=
    results_directories,
                    help='Directory where results are saved')
parser.add_argument('--directories', default=directories, nargs='+',
                    help='Directories where video files are located
    for testing. Each '
                         'directory should contain videos at different
     focal '
                         'distances and the name of each video file
    should be '
                         'its focal distance (e.g., 150.avi)')
parser.add_argument('--test_classifier', action='store_true')
parser.add_argument('--save_unscaled_out', action='store_true')

parser.set_defaults(test_classifier=False)
parser.set_defaults(save_unscaled_out=True)

args = parser.parse_args()

use_classifier = args.test_classifier

def plot_cm_matrix(targets, predictions, label_names, file_name=None):
    cm = confusion_matrix(targets, predictions, labels=label_names,
```

```
    normalize='true')
     disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=
    label_names)

     fig = plt.figure(figsize=(6.4 * 1.1, 4.8 * 1.1))
     # disp.plot(include_values=False, ax=fig.gca())
     df_cm = pd.DataFrame(cm, index=label_names, columns=label_names)
     sns.heatmap(df_cm, cmap="flare", linewidths=.5, square=True, vmin
    =0, vmax=1)
     plt.xlabel("Predicted label")
     plt.ylabel("True label")
     plt.tight_layout()
     if file_name is not None:
         plt.savefig(file_name, bbox_inches='tight', pad_inches=0.02)

if __name__ == '__main__':
     results_directories = args.results_directories
     testing_directories = args.directories
     time_taken_arr = []
     for directory, result_dir in zip(testing_directories,
    results_directories):
         label_names_np = pickle.load(open(os.path.join(result_dir, "
    label_names.pkl"), "rb"))
         net = FocusClassifier(len(label_names_np)).to(device) if
    use_classifier else FocusPredictor().to(device)
         net.load_state_dict(torch.load(os.path.join(result_dir, "
    models/trainedModel.pth")))

         print(f"Loading videos in {directory}...")
         label_names_testing, datasets = load_all_datasets(directory,
    predictor=not use_classifier)
         assert np.all(label_names_testing == label_names_np)

         label_names = torch.from_numpy(label_names_np).long().to(
    device)

         testing_dl = DataLoader(ConcatDataset(datasets), batch_size
    =256, shuffle=False, num_workers=4, pin_memory=True)
```

```
        if args.save_unscaled_out:
            net_output, test_targets, _ = get_net_outout_and_time(net,
 testing_dl, label_names)
            net_output.astype(np.float32).tofile(os.path.join(
result_dir, "net_out.dat"))
            test_targets.astype(int).tofile(os.path.join(result_dir, "
targets.dat"))

        test_preds, test_targets, avg_time_taken = predict_labels_net(
net, testing_dl, label_names, not use_classifier)
        time_taken_arr.append(avg_time_taken)

        class_report_txt = classification_report(test_targets,
test_preds, target_names=label_names_np.astype(str))
        print(class_report_txt)
        with open(os.path.join(result_dir, "testing_report.txt"), 'w')
 as f:
            f.write(class_report_txt)

        class_report_dict = classification_report(test_targets,
test_preds, target_names=label_names_np.astype(str),
                                                  output_dict=True)
        with open(os.path.join(result_dir, "class_report_dict.pkl"), '
wb') as f:
            pickle.dump(class_report_dict, f)

        plot_cm_matrix(test_targets, test_preds, label_names_np, os.
path.join(result_dir, "testing_cm.pdf"))

    print(f"CNN processed {1 / np.mean(time_taken_arr):.2f} images/sec
    when using {device}")
```

D.4. Testing of CNN Model for Simulation

```
import argparse
import pickle
```

```
from fastai.data.core import DataLoaders
from fastai.data.load import DataLoader

from model import FocusPredictor, FocusClassifier
from utils import *

directories = [os.path.join("training_videos", x) for x in ["Copper",
    "Steel", "Silicon"]]
parser = argparse.ArgumentParser(description='Train CNN to predict
    images on different sets of videos')
parser.add_argument('--bs', type=int, default=16, help='Batch size')
parser.add_argument('--num_epochs', type=int, default=100, help='
    Number of epochs')
parser.add_argument('--val_p', type=float, default=0.15, help='
    Percentage of frames to use for validation')
parser.add_argument('--seed', type=int, default=423132, help='Seed
    used for splitting frames into train/val/test')
parser.add_argument('--directories', default=directories, nargs='+',
    help='Directories where video files are located. Each '
                                                                    '
    directory should contain videos at different focal '
                                                                    '
    distances and the name of each video file should be '
                                                                    'its
    focal distance (e.g., 150.avi)')
parser.add_argument('--results_dir', default="results_predictor", help
    ='Directory where results will be saved')
parser.add_argument('--train_classifier', action='store_true')
parser.set_defaults(train_classifier=False)

args = parser.parse_args()

val_percentage = args.val_p
test_percentage = 0  # test on separate data
seed = args.seed
batch_size = args.bs
num_epochs = args.num_epochs
```

```
use_predictor = not args.train_classifier

if __name__ == '__main__':
    pred_str = "predictors" if use_predictor else "classifiers"
    print(f"Training different CNN {pred_str} for videos in {args.
    directories}")
    for directory in args.directories:
        result_dir = os.path.join(args.results_dir, os.path.split(
    directory)[-1])
        os.makedirs(result_dir, exist_ok=True)

        print(f"Loading videos in {directory}...")
        label_names_np, datasets = load_all_datasets(directory,
    predictor=use_predictor)
        max_label = np.max(label_names_np)
        min_label = np.min(label_names_np)

        # only train on every other label including beginning and end
        if use_predictor:
            train_label_indices = [0] + list(range(2, len(
    label_names_np) - 1, 2)) + [-1]
            training_labels = label_names_np[train_label_indices]
        else:
            training_labels = label_names_np
        print(f"Training using videos {training_labels}...")

        train_dset, val_dset, _ = split_train_val(datasets,
    label_names_np, val_percentage, test_percentage, training_labels,
    seed)
        label_names = torch.from_numpy(label_names_np).long().to(
    device)

        train_loader = DataLoader(train_dset, batch_size=batch_size,
    shuffle=True, num_workers=4, pin_memory=True)
        val_loader = DataLoader(val_dset, batch_size=512, shuffle=
    False, num_workers=2, pin_memory=True)

        dls = DataLoaders(train_loader, val_loader)
```

```
    if use_predictor:
        net = FocusPredictor()
        train_predictor(net, dls, label_names, num_epochs,
result_dir)
    else:
        net = FocusClassifier(len(label_names_np))
        train_classifier(net, dls, num_epochs, result_dir)

    pickle.dump(label_names_np, open(os.path.join(result_dir, "
label_names.pkl"), "wb"))

    # test on testing data
    # test_preds, test_targets = predict_labels(net, test_loader,
label_names, use_predictor)
    # pickle.dump((test_preds, test_targets, label_names_np), open
(os.path.join(result_dir, "test_pred_targets.pkl"), "wb"))
```