Deep Adaptive Optics:

A Machine Learning Method for Aberration Correction in Coordinate-Targeted Super-Resolution Microscopy

Work by:
Hope McGovern, Sc.B
Wiebke Jahr, PhD
Johann Danzl, PhD, MD

Authored by:
Hope McGovern, Sc.B

Table of Contents

Table of Contents (Cont.)

1. Abstract

In coordinate-targeted superresolution microscopy, aligning the light patterns responsible for switching molecules between the signalling and non-signalling states is often a tedious and painstaking process for the user. However, the process is necessary, as image resolution is largely determined by the quality of the intensity distributions -- especially the minima. To aid in this process, we propose a neural network able to detect aberrations in the “donut” PSF optically and transmit a corrective phase pattern from the predicted Zernike polynomials to a SLM in the beam path. We expand upon earlier work in this area [1,2] by creating a synthetic data generation pipeline, as well adding novel features such as the ability to train on cross-sections of the PSFs.

2. Introduction

Light microscopy can be used in combination with fluorescent labelling to distinguish molecular species with a high signal-to-noise (SNR) ratio. This, in addition to its ease of sample preparation, renders light microscopy immensely relevant for the life sciences. For several centuries, diffraction of light was believed to fundamentally limit the resolution of far-field light microscopy, but this historical limit has been circumvented by various super-resolution microscopies [3].

High light intensities and optical aberrations hinder the separation of fine features in a specimen, even though the resolution is theoretically unlimited [4–6]. In STED microscopy, a light pattern of alternating maxima and minima drives fluorescent molecules from the signalling excited state to the dark ground state. However, a tightly confined volume around intensity minima is not affected by this process and those fluorescent molecules remain “switched ON” [7]. The most widely used light patterns are created using a vortex phase mask, resulting in a "doughnut"-shaped focus to constrict the fluorescent volume in the image plane (called *xy*-STED here). Simulations that have this pattern are exquisitely sensitive to aberrations [8,9]. Specifically, aberrations "filling" the zero intensities of the STED patterns result in a decrease of signal, increase of state cycling and phototoxicity [10]. Since the image formation is strongly dominated by the intensity distribution in the STED beams [11], it is sufficient to correct aberrations in these [12]. The required corrections are determined using either sensor-less iterative adaptive optics attempting to optimize suitable image quality metrics [13], or sensor-based approaches directly measuring the aberrated wavefront [14].

It is possible to identify and correct for many aberrations through meticulous alignment of the microscope system before the start of an experiment. Many modern STED microscopes are

equipped with a spatial light modulator (SLM) displaying phase masks to create the STED intensity patterns [15], where it is straightforward to add aberration correction to the vortex or top-hat patterns [16] and adjust the overlay of the excitation and depletion beams [17]. Routine alignment is usually performed using non-bleaching scattering gold beads which directly visualize the STED intensities.

Aberration-induced changes of the point spread functions (PSF) may be subtle or easy to mistake for other misalignments [9], thus requiring an experienced operator. Even when aberrations are parametrized into orthogonal modes using, e.g. Zernike polynomials, the large number of free parameters makes alignment cumbersome and time-consuming.Therefore, a computational auto-alignment routine replacing, or at least aiding, the operator would be highly beneficial and could be easily established also in non-expert labs or microscopy facilities.

Increasingly, there is growing interest in incorporating machine learning techniques into (super-resolution) microscopy [18,19]. In one example, it has been used to find optimal acquisition parameters for STED imaging [20]. In another, the genetic algorithm has been proposed to correct both system- and sample-induced aberrations in STED, but requires tens of iterations to produce adequate compensations [21]. Another growing area of application for machine learning has been post-acquisition image restoration [22,23], whose goal is to restore low quality or undersampled images.

Our problem is fundamentally one of regression: estimating the underlying Zernike polynomials of a given image of an aberrated PSF. As such, some statistical method of estimation guided by a limiting cost function is required. The Convolutional Neural Network (CNN), a deep learning architecture, has revolutionized the field of computer vision and provided state-of-the-art results for tasks in which training data can be found in sufficient quantities [24]. The main advantage rendered by a CNN for this particular problem statement is that its convolutional structure leverages compositional (spatiotemporal) information in the input image. As the underlying phenomenon (aberrations as parameterized by Zernike modes) has a noticeable visual impact on the image, we hypothesize that a CNN will be able to “notice” the visual effects of coma, trefoil, etc. on the PSF and be able to quantify its aberrations precisely.

Our pipeline renders the creation of synthetic training data trivial, thus inviting the use of this simple yet powerful deep learning tool which often requires training data in the tens of thousands to be effective. We designed an automated, easy-to-use alignment suite for STED microscopy based on neural networks. All computationally intense operations are performed offline in order to ensure fast (on the order of seconds) online alignment of the microscope.

Training data were generated in-silico; the phase distribution in the back aperture of the objective lens was calculated by adding a vortex to a linear combination of Zernike polynomials (weighted randomly within a given range). The resulting aberrated STED PSF was calculated using vector diffraction [25]. For alignment on scattering gold beads, a training data pair for the neural net [26] consisted of a vector containing the random weights of the Zernike polynomials, and the aberrated STED PSFs. During operation of the microscope, experimental PSFs were acquired using bead samples and the weights required for correction were determined by the trained model. The linear combination of Zernike polynomials was calculated and added onto the phase mask currently displayed on the SLM in the STED beampath.

3. Simulation of Aberrated STED PSFs using Vector Diffraction Theory

The use of high numerical aperture (NA) lenses in the microscopy process renders the traditional scalar diffraction assumptions too simplistic -- it is necessary to account for the polarization of light and consequently represent the electric field at some distance from the aperture as a vector. We therefore make use of so-called vector diffraction theory, as described in [25]:

$$E(r_2, \phi_2, z_2) = iC \int_\Omega \int \sqrt{\cos\theta} \sin\theta A_1(\theta, \phi) V(\theta, \phi) \cdot \begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix} e^{ikn(z_2 \cos\theta + r_2 \sin\theta \cos(\phi - \phi_2))} d\theta d\phi$$

Where C is a constant scalar, A_1 is the input aperture function; in other words, the phase mask multiplied by an amplitude, which is constant in (θ, ϕ) for our use case. The vector $\overrightarrow{P}$ represents the polarisation of the light. The polarisation conversion matrix is given as:

$$V(\theta, \phi) = \begin{bmatrix} 1 + (\cos\theta - 1)\cos^2\phi & (\cos\theta - 1)\cos\phi\sin\phi & -\sin\theta\cos\phi \\ (\cos\theta - 1)\cos\phi\sin\phi & 1 + (\cos\theta - 1)\sin^2\phi & -\sin\theta\sin\phi \\ \sin\theta\cos\phi & \sin\theta\sin\phi & \cos\theta \end{bmatrix}$$

With this formula in mind, it is possible to calculate the focused PSF created by a given wavefront in the objective's back aperture. We made use of code written to implement these calculations in order to create synthetic data for the training of our machine learning models. Two kinds of focused PSFs were simulated: Emission PSFs and STED PSFs.

3.1. STED PSF

Fundamental to the process of simulating our data is the idea that aberrations can be parametrized by any set of orthogonal polynomials, but it is standard in the optics community to parametrize aberrations by a set of such orthogonal polynomials called Zernike polynomials [6,27]. By changing the scalar multiplier of each polynomial term, we are able to simulate a different "mix" of aberrations, from coma to spherical to trefoil, etc. We use these weights of the Zernike modes to characterize the input image, and it is the predictions of these weights that are the output of the model.

The full process for generating a synthetic data point is to (1) randomly generate a sequence of numbers, taken either from a uniform or a random distribution from a range determined to match a realistic setting, (2) use those numbers to generate a phase mask of Zernike modes. This is described [27] analytically as:

$$\left\{ \begin{array}{l} Z_n^m(\rho\phi) = R_n^m(\rho)\cos(m\phi) \\ Z_n^{-m}(\rho\phi) = R_n^m(\rho)\sin(m\phi) \end{array} \right\}$$

Where Z_n^m is an even-numbered Zernike mode, Z_n^{-m} is an odd-numbered Zernike mode, and R_n^m is the radial polynomial. Typically, we simulate phase masks of the 3rd-15th Zernike modes. It is important to note that the 0th-2nd Zernike modes are not considered, as the 0th mode is Piston, which is a constant offset with no effect on the PSF, and the 1st and 2nd modes are Tip and Tilt, respectively, which merely translate the PSF and do not affect its shape [9]. We also treat the 5th mode, Defocus, rather uniquely, as detailed in Section 7.

Step (3) is that the phase mask of the aberrations is added to the vortex beam phase mask (the phase mask for the ideal PSF) and finally (4) the combined phase mask is run through the vector diffraction code, which integrates it and evaluates the integral at a given distance from the lens. The integration process depends on many real world parameters from the microscope itself that are read from a python dictionary that is stored in a text file and read. To replicate our procedure with a different microscope system or any variation in back aperture size, wavelength, laser power, polarization of incoming light, pulse length, etc. those would need to be modified before creating synthetic data.

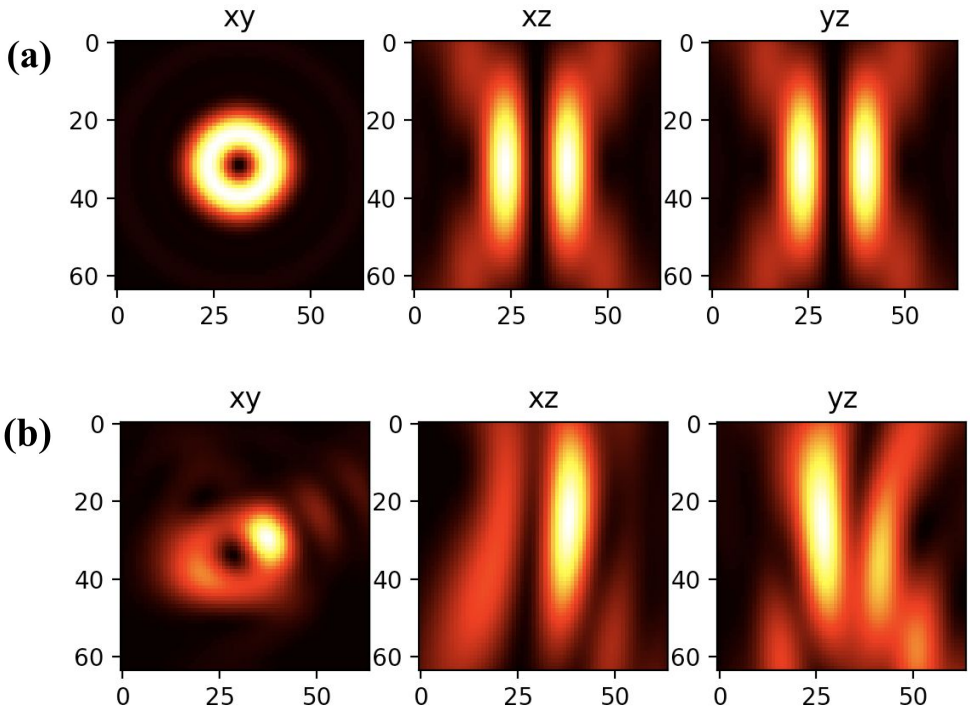


Figure 1: (a) the ideal PSF, three ortho-sections seen. **(b)** aberrated STED PSF with a Zernike weight label of **[-0.119, 0.156, -0.107, 0.152, 0.209, -0.3, -0.085, -0.156, 0.115, -0.02, 0.095]**

The PSF is a fully 3-dimensional object, but for convenience we tend to observe it through three "ortho-sections", which are essentially cross-sections of the full 3D volume at the respective center of each axis (see Fig. 1). [2] uses only the xy ortho-section in their model, whereas we use all three to better characterize the aberrations, similarly to the model in [28] for flower classification with multiple different views of the input. It would perhaps be possible to incorporate more of the cross-sections, eventually analyzing the whole volume of the PSF, although it remains to be seen whether this addition would provide relevant data to arrive at a higher accuracy model or if it would merely slow down the training without adding particularly unique information. It may also be that when you feed in a series of overlapping spatiotemporally continuous data that a time-series model, like a Long Short-Term Memory (LSTM) network [29], would be better suited to the problem statement. For now, we limit ourselves to the three ortho-sections, assuming that each renders unique information that assists the model in converging.

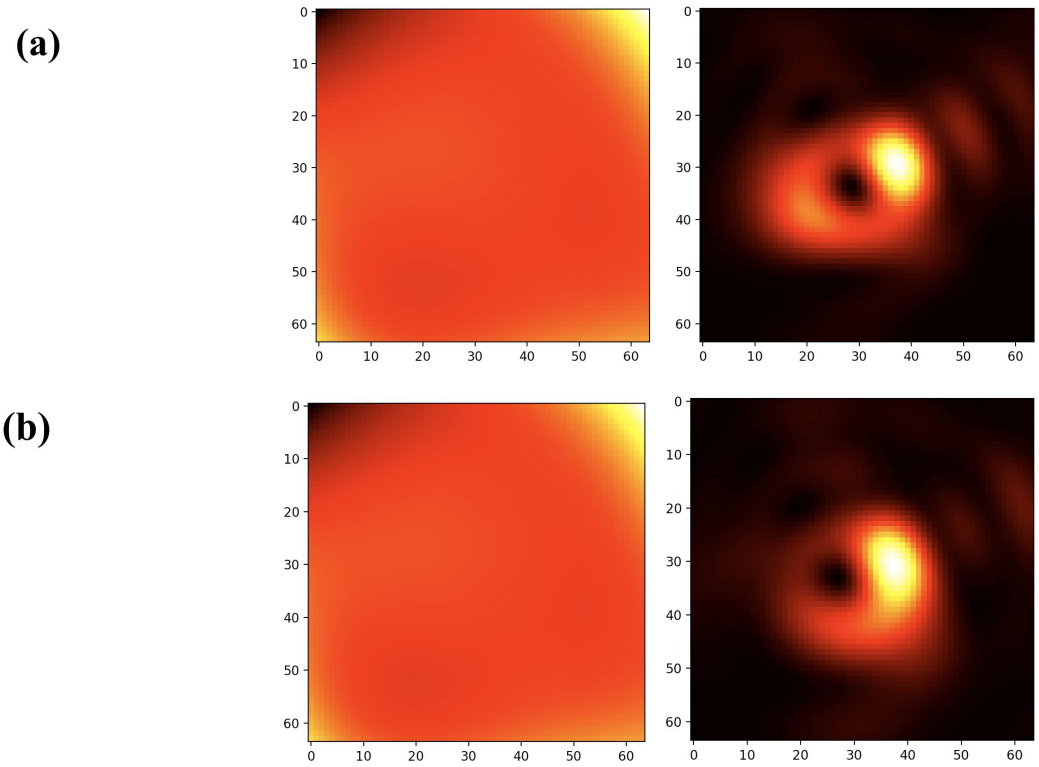


Figure 2: (a) a phase mask with no offset and its corresponding (aberrated) PSF **(b)** a phase mask shifted by the vector **[-0.117, 0.427]** and its corresponding PSF. The difference in result despite the small shift of the phase mask renders determination of the offset very relevant.

As an additional note, the phase mask is sometimes offset by a vector similarly drawn from a uniform distribution. For an illustration of the way in which a lateral shift of the phase mask changes the resulting STED PSF, observe Fig. 2. Because this shift affects the appearance of the PSF, it is important that we are able to distinguish it from regular Zernike aberrations. When this step is incorporated, the numbers are generated at the same time as those for the Zernike weights. Then, during Step (3), the phase mask is added to the vortex beam with the randomly generated offset from the center of the two images.

4. Convolutional Neural Networks

CNNs build upon previous neural network structures such as multi-layer perceptron models and have provided extraordinary performance gains for image classification and object detection [24]. This is because by maintaining the spatiotemporal continuity of the input (rather than flattening it), CNNs can leverage compositional relationships to abstract not only low level features like edges and gradients, but also higher level features like eyes or bicycles.

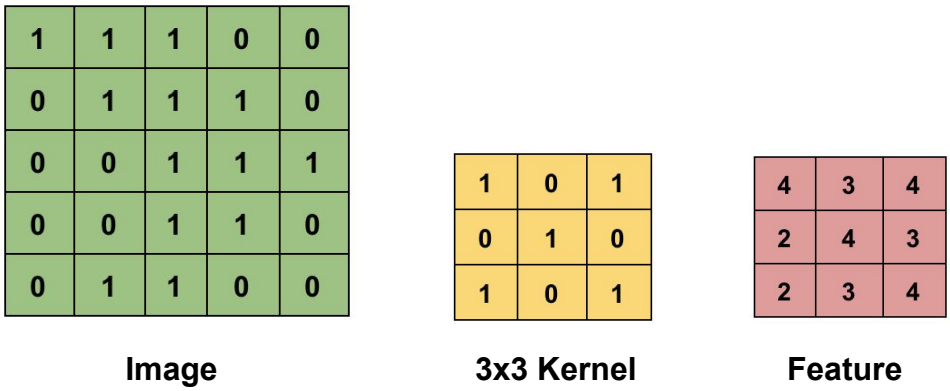
As a deep learning algorithm, CNNs transform an input into a list of distinctive and identifiable features that are present. The algorithm accomplishes this by passing the input through a series of filters -- this process of passing a filter over an input array is known as convolution [26]. CNNs are based on the pattern of neurons in the human brain, in which individual neurons respond to external stimuli in a restricted region, known as a receptive field, and multiple of these receptive fields overlap to cover the entire visual field. Replicating this mathematically, we represent some input, usually an image or a vectorization of text, as an array of numbers and multiply some subset of this array by a filter array, then slide the filter array over the input like a sliding window, and in this manner the most important features of the image are abstracted and used to characterize the image.

4.1. Components of CNN

4.1.1. Convolutional filter

A convolutional filter or convolution kernel is simply an array of values which are multiplied and summed over some subset of an input. The result of this is the extraction of distinguishing features from the input. Conventionally, the first convolutional layer is thought to extract low level features like edges or gradients [24], and every subsequent convolutional layer extracts higher level features (these may be more complex "objects" like eyes or dogs). After the filter is convolved with the first region of the input, it is moved like a "sliding window" at some stride until the entire input has been processed with overlapping fields. Sometimes the input is padded so that the convolved feature is not a reduced dimensionality compared to the input. See Fig. 3 for an illustration of the convolutional process.

(a)



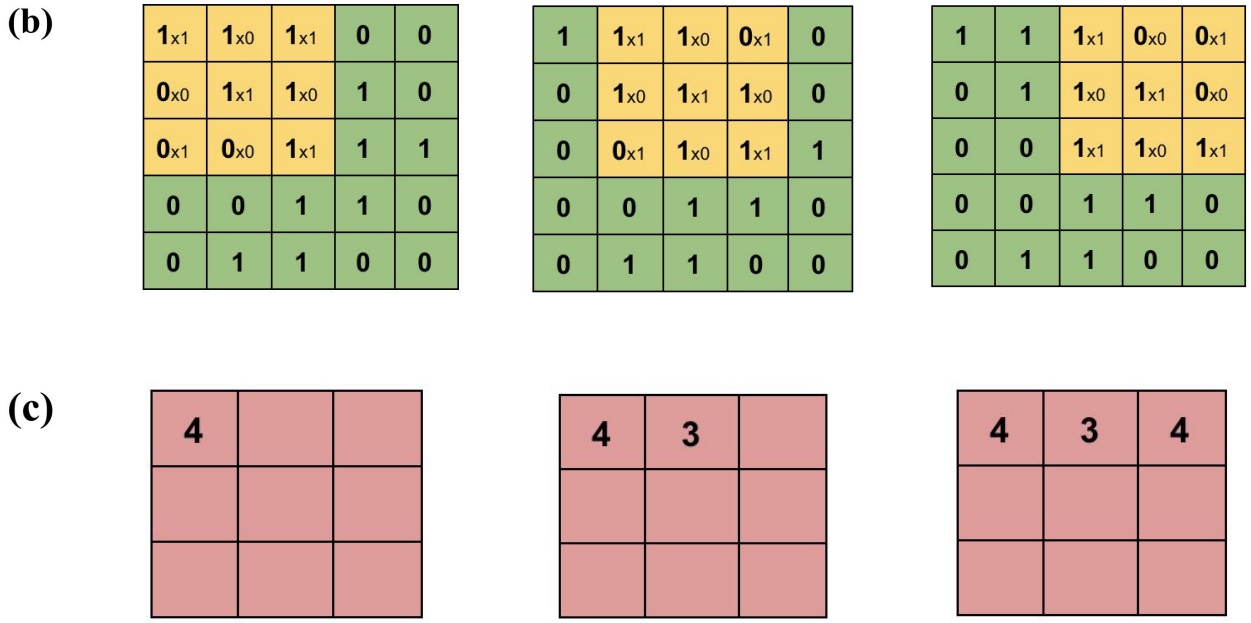


Figure 3: (a) Illustration of an image as a 5x5 array of integer pixel values, a 3x3 convolutional kernel with which to convolve the image, and the resulting 3x3 convolved feature. **(b)** shows the sliding window approach to convolution, the kernel convolves a subset of the pixel values before sliding a step to an overlapping region. This is what allows a CNN to take advantage of compositional information in the image. **(c)** shows the process of building up the values of a feature. Each element is the sum of the matrix multiplication applied to the pixel subset highlighted in (b). Example extracted from [30].

4.1.2. Activation function

An activation function makes it possible for neural networks to represent complicated data by adding nonlinearities to the computational process [30]. Without introducing any nonlinearities, passing information through a neural net would just be a series of linear combinations -- and linear combinations can always be represented by an equivalent, single combination. This would render a model unable to represent any non-linear relationship between input and output. The most commonly used activation function for CNNs is called the Rectified Linear Unit (ReLU), defined as:

$$f(x) = \max(0, x)$$

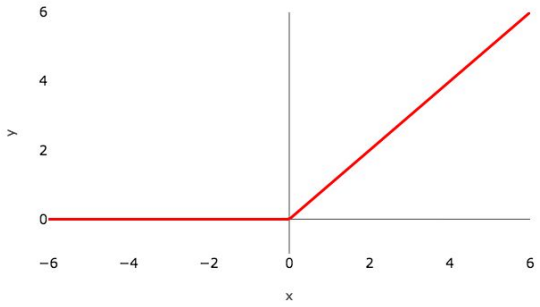


Figure 4: plot of the Rectified Linear Unit (ReLU) function

An activation function is typically applied to every convolutional kernel in a network with rare exceptions. It is typically not applied to fully connected layers [30,31].

4.1.3. Dropout

Dropout [32] is essentially a regularization technique: at specific points between layers, a percentage of connections between neurons are simply dropped. Sometimes users will choose to drop as much as 50% of the connections between neurons. This is done to prevent large neural network models from overfitting and forming irrelevant connections in the data. Following (Zhang et al. 2019), we apply a few dropout layers to our models, ranging between 10 and 20 percent.

4.1.4. Fully Connected Layer

The ultimate, or sometimes penultimate, step in the CNN's architecture definition is the fully connected (fc) layer. After all of the feature maps have been generated by the convolutional kernel, they are flattened to one dimension and passed to an fc layer. If the problem statement requires a classification algorithm, the fc layer will be activated by a softmax function such that the most likely class of the input is the output of the model. Our project is one of multiple regression, so rather than the class of the input; we want the model to predict the weights of the Zernike modes themselves. To do this, we pass the flattened feature maps through several fully connected layers with no activation function, paring the dimensionality down until the final layer is a single vector which corresponds to the Zernike mode weights we are trying to predict.

5. Model Design

5.1. Deep Learning Framework

Many programmers would like to avoid writing all backend code for dropout layers, activation functions, etc. and instead opt to use a deep learning framework to provide built-in functions to vastly decrease the time to get a model up and running. Some of the most popular such frameworks are: Tensorflow, PyTorch, Caffe, and Theano. We choose to use PyTorch [33] as the deep learning framework for its ease of use and particularly its default eager execution mode. PyTorch is qualitatively the most "pythonic" of the available frameworks and therefore has the lowest learning curve for prior python users. Tensorflow, the other main framework of interest, does not use eager execution mode but rather defines types statically, which makes the underlying data structure more opaque to the user and thus harder to debug. However, Tensorflow includes useful visualization features natively such as Tensorboard [34], a dashboard to visualize input data, batching, loss curve during training, etc (See Sec 5.3-5.5). Although less elegant than native use, it is possible to use the Tensorboard visualization tools with PyTorch, and we choose to do so to monitor the training process.

5.2. List of Model Architectures

The following is an exhaustive list of the models used for the project and their unique features. Many are largely similar with a minor but significant change. Where reasonable to do so, redundancies have been redacted from the report with ellipses and only changed lines are listed (made evident by the line number).

5.2.1. Net12

The following model definition was the first model that was created during the course of the project and was the prototype for all subsequent models. The architecture makes use of all the features of a typical CNN: convolutional layers, fully connected layers, a ReLU activation function, maximum pooling, and drop out. The number of layers and percentages of dropouts are recreated from [2]. The output vector has a dimension of 12 -- this accounts for the weights of the 3rd-15th Zernike modes.

```
1    import torch
2    import torch.nn as nn
3    import torch.nn.functional as F
4
5
6    class Net12(nn.Module):
```

```
7       """
8       A simple CNN based on AlexNet
9       Architecture followed from Zhang et al.,
10      "Machine learning based adaptive optics for doughnut-shaped beam" (2019)
11      """
12       def __init__(self):
13        super(Net, self).__init__()
14        self.conv1 = nn.Conv2d(1, 32, kernel_size=5, stride=1, padding=2)
15        self.conv2 = nn.Conv2d(32, 32, kernel_size=5, stride=1, padding=2)
16        self.conv3 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)
17        self.conv4 = nn.Conv2d(64, 64, kernel_size=3, stride=1, padding=1)
18        self.conv5 = nn.Conv2d(64, 64, kernel_size=3, stride=1, padding=1)
19
20        self.fc1 = nn.Linear(8 * 8 * 64, 512)  # 64 channels, final img size 8x8
21        self.fc2 = nn.Linear(512, 512)
22
23        self.fc3 = nn.Linear(512, 12)
24
25    def forward(self, x):
26
27        x = x.float()
28        x = F.dropout(F.max_pool2d(F.relu(self.conv1(x)), (2, 2)), p=0.1)
29        x = F.dropout(F.max_pool2d(F.relu(self.conv2(x)), (2, 2)), p=0.1)
30        x = F.relu(self.conv3(x))
31        x = F.relu(self.conv4(x))
32
33        x = F.max_pool2d(F.relu(self.conv5(x)), (2, 2))
34        # flatten
35        x = x.reshape(x.size(0), -1)
36        x = F.dropout(F.relu(self.fc1(x)), p=0.2)
37        x = F.dropout(F.relu(self.fc2(x)), p=0.2)
38        x = self.fc3(x)
39        return x
```

5.2.2. MultiNet12

The modification of this model from the previous is that we consider the xz and yz ortho-sections as well as the xy. Here, we simply stack the three ortho-sections before passing it as an input to the model, treating each ortho-section as a different color channel.

```
14        self.conv1 = nn.Conv2d(3, 32, 5, padding=2)
...
23        self.fc3 = nn.Linear(512, 12)
```

5.2.3. MultiOffsetNet14

This model is created to train for the 3rd-15th Zernike modes (including Defocus) and offset. In practice, none of the models which included defocus performed very well in real world testing, so this model has been deprecated.

```
14     self.conv1 = nn.Conv2d(3, 32, 5, padding=2)
...
23     self.fc3 = nn.Linear(512, 14)
```

5.2.4. Net11

The small but highly significant change to this model compared to the vanilla Net12 is that the Zernike mode describing Defocus (the 5th) has been removed from the training dataset and therefore is also removed from the output vector, so the output reduces to an 11-dimensional vector rather than a 12. This shift in process is detailed more in Section 8b. All subsequent models likewise drop this 5th Zernike mode, as we instead correct for it in situ rather than with the model.

```
14    self.conv1 = nn.Conv2d(1, 32, 5, padding=2)
...
23    self.fc3 = nn.Linear(512, 11)
```

5.2.5. OffsetNet13

This model predicts the offset and Zernike mode weights of an aberrated phase mask, so the output vector has a dimension of 13: 11 Zernike modes and 2 offset terms.

```
14    self.conv1 = nn.Conv2d(1, 32, kernel_size=5, stride=1, padding=2)
...
23    self.fc3 = nn.Linear(512, 13)
```

5.2.6. MultiNet11

This model also has an 11-dimensional output vector, but treats the ortho-sections as the different color channels of a single input, similar to MultiNet12.

```
14    self.conv1 = nn.Conv2d(3, 32, 5, padding=2)
...
23    self.fc3 = nn.Linear(512, 11)
```

5.2.7. MultiOffsetNet13

This model outputs a 13-dimensional vector: 11 Zernike modes and 2 offset terms, with the additional feature that it operates on all three ortho-sections.

```
14    self.conv1 = nn.Conv2d(3, 32, 5, padding=2)
...
23    self.fc3 = nn.Linear(512, 13)
```

5.2.8. OffsetNet2

This model is designed to just predict the offset of the phase mask; accordingly, it has a 2-dimensional output vector. It is trained on a smaller training dataset (10% the size of the typical dataset), in which the phase pattern for an ideal doughnut is shifted from center by some offset and no Zernike modes are present. We used this model to verify that the offset was able to be correctly predicted and that there was no interference of the offset prediction for the Zernike prediction.

```
14    self.conv1 = nn.Conv2d(1, 32, kernel_size=5, stride=1, padding=2)
...
23 self.fc3 = nn.Linear(512, 2)
```

5.2.9. MultiNetCat11

Following [28], we also create a model that treats each of the three ortho-sections as separate inputs rather than as three color channels of the same image. The *xy*-, *xz*-, and *yz*- ortho-sections are convolved with separate filters and at some point, concatenated prior to the fully connected (predictive) layer. To date, no visible accuracy gains during have been noticed using this model compared to the MultiNet11, but rigorous, quantitative comparison has not yet been conducted.

```
11      self.conv1 = nn.Conv2d(1, 32, 5, padding=2)
...
25      def forward(self, img):
26
27          x = img[:, 0].unsqueeze(1) # adding dim of 0 after batch dim
28          y = img[:, 1].unsqueeze(1)
29          z = img[:, 2].unsqueeze(1)
30
31          x = x.float()
32          x = F.dropout(F.max_pool2d(F.relu(self.conv1(x)), (2, 2)), p=0.1)
33          x = F.dropout(F.max_pool2d(F.relu(self.conv2(x)), (2, 2)), p=0.1)
34
35          y = y.float()
36          y = F.dropout(F.max_pool2d(F.relu(self.conv1(y)), (2, 2)), p=0.1)
37          y = F.dropout(F.max_pool2d(F.relu(self.conv2(y)), (2, 2)), p=0.1)
38
39
40          z = z.float()
41          z = F.dropout(F.max_pool2d(F.relu(self.conv1(z)), (2, 2)), p=0.1)
42          z = F.dropout(F.max_pool2d(F.relu(self.conv2(z)), (2, 2)), p=0.1)
43         # concatenating the ortho-sections
44          a = torch.cat((x, y, z), dim=1)
45          a = F.relu(self.conv3(a))
46          a = F.relu(self.conv4(a))
47
48          a = F.max_pool2d(F.relu(self.conv5(a)), (2, 2))
49          # flatten
50          a = a.reshape(a.size(0), -1)
51
52          a = F.dropout(F.relu(self.fc1(a)), p=0.2)
53          a = F.dropout(F.relu(self.fc2(a)), p=0.2)
54          a = self.fc3(a)
55          return a
```

5.3. Training parameters

5.3.1. Loss fn

While not technically considered a training parameter, the loss function is perhaps the most important choice to be made about the training process, as it is the determination of how the

model calculates its deviation from the ground truth label [30]. The cost function that we use is the mean squared error (MSE) between the ground truth Zernike mode weight labels and the vector that is the output of the model. This operation is done at the batch-level, and it is important to note that when calculating the loss, we first average along the 0th dimension before we sum across the vector. This way, minimising the error for each coefficient is privileged over minimising the overall loss of the prediction vector.

During the training loop, the inputs are passed through the model (the series of matrices with learned weights and non-linear activation functions defined in the network architecture), then the MSE of the predicted labels to the ground truth is calculated. The parameter gradients are zeroed, then the loss is propagated backwards.

```
# Run the forward pass
outputs = model(images) # e.g. [32, 12] = [batch_size, output_dim]
loss = criterion(outputs, labels) # MSE
# sum of averages for each coeff position
loss = torch.sum(torch.mean(loss, dim=0))

# zero the parameter gradients
optimizer.zero_grad()
# backward + optimize only in train
loss.backward()
optimizer.step()
```

5.3.2. Learning rate

Learning rate is considered a key training parameter. The learning rate determines how quickly the model adapts to the problem at hand. Sometimes a scheduler is used to monitor decreases in the learning rate over the course of the training (for example, after 3 epochs the learning rate decreases by a factor of 10). Presumably, this aids in zeroing in on a local minimum for the loss [30]. We use a learning rate of 1e-3, which is within the recommended range for the optimizer we are using. There is no single ideal learning rate for every problem; rather, there is an ideal operating range for every optimizer [35].

5.3.3. Optimizer

The general principle of training a machine learning model is iteratively updating randomly initialized parameters to minimize a given cost function. When a model performs this update after seeing just one sample, it is known as Stochastic Gradient Descent (SGD). SGD is potentially the method of training that converges the fastest by updating the model parameters

more frequently, but it consequently has a much higher variance in model parameters and may overshoot the minimum loss value if the learning rate is not incrementally reduced during training [36]. If instead the model weights are updated only after an entire batch (a subset of the data), it is known as Mini-Batch Gradient Descent [30].. Several other optimization techniques have been proposed (AdaGrad, RMSprop, Adam, AdaDelta, Momentum, Nesterov Accelerated Gradient) and is an active field of research. Different optimization algorithms all offer trade-offs in terms of robustness, computational power, reliability, and time to converge. For image classification and/or regression, the Adam, or Adaptive Moment Estimation [37], optimization method seems to be the industry standard [38,39], and we use this optimizer for all of our trainings.

5.3.4. Epochs

Epochs are, quite simply, the number of times that the training loop iterates through the entire dataset. The balance to find here is to expose the model to train to be robust while not allowing it to get too used to the data it has already seen. Our default was 15 epochs as, from empirical trials, the training curve seems to level out around that time.

5.3.5. Dataset class

To organize and store the synthetic data while it is being manipulated in python, we write a custom dataset as a subclass of PyTorch data class. This dataset is created from an h5py [40] file that was used as a repository for the synthetic data. It returns the data marked for train, validation, and test sets depending on a given 'mode' argument and has the ability to add transformations such as normalization and added noise to the sample. By adding these transforms to the __getitem__ function, it ensures they will happen when the dataset is iterated over, rather than when the constructor call is given. See below our class definition for the PSFDataset class.

```
1 import h5py
2 import numpy as np
3 from torch.utils import data
4 import torch
5 from torchvision import transforms
6
7 class PSFDataset(data.Dataset):
8   """ Point Spread Function h5py Dataset. """
9
10   def __init__(self, hdf5_path, mode, transform=None):
11       """
12       Args:
```

```
13          hdf5_path (str): Path to the hdf5 file
14      """
15      # Creates an h5py object from the given path
16      self.file = h5py.File(hdf5_path, "r")
17      self.transform = transform

18      # if training, loads the training and validation images and labels
19      if mode =='train':
20          self.images = self.file['train_img']
21          self.labels = self.file['train_labels']
22      elif mode == 'val':
23          self.images = self.file['val_img']
24          self.labels = self.file['val_labels']
25      # if testing, loads the test images and labels
26      elif mode == 'test':
27          self.images = self.file['test_img']
28          self.labels = self.file['test_labels']
29
30  def __len__(self):
31      return self.images.shape[0]
32
33  def __getitem__(self, idx):
34      sample = {'image': self.images[idx], 'label': self.labels[idx]}
35
36      if self.transform:
37          sample = self.transform(sample)
38
39      return sample
```

5.3.6. Batch Size, Data Loaders, and Batch Normalization

As only a subset of the whole training dataset is used at one time, PyTorch offers built-in classes called Dataloaders which only load in the examples contained in the relevant batch. We make use of these to minimize the amount of data needed to be stored in memory during the training loop.

At this stage, the data transformations described in the Batch Normalization and Adding Noise During Loading sections are applied to each data point as it is loaded into the training loop by the data loader.

Batch normalization is a method that is proven to help models converge to a solution more quickly without suffering a loss in accuracy [41]. The changes in the distribution of internal nodes of a network are known as internal covariate shift, and eliminating this shift leads to faster training. The motivating idea is that by removing irrelevant variations in the data, there is less need for careful parameter initialization and small learning rates. This process is also considered a form of regularization similar to dropout and may sometimes take the place of dropout in a network [30]. To implement this, we first iterate through the entire training set and calculate the mean and standard deviation, then as each batch is loaded, each training example in the batch is normalized, such that the mean of each training example is ~1 and the standard deviation of each training example is ~0. Below is our implementation:

```
class Normalize(object):
    """Given a mean and std with constructor call, it normalizes the input.
    Mean and std must be calculated first."""
    def __init__(self, mean, std):
        self.mean = mean
        self.std = std

    def __call__(self, sample):
        image, label = sample['image'], sample['label']

        for channel in range(image.size(0)):
            image[channel] = (image[channel] - self.mean[channel])/ self.std[channel]

        return {'image': image,
                'label': label}
```

5.3.7. Adding Noise During Loading

Adding noise to data should theoretically increase the robustness of the model [30]. However, it is not necessary to add noise to the training data itself, as it can be added when the batch is loaded into the training loop. We add two types of noise, background noise and Poisson noise, to the synthetic images to help them more realistically simulate real world data. Below, our implementation of the add_noise function can be seen. The default arguments of background

noise and Poisson amount were determined experimentally by the Danzl group member who contributed the function. We modified those slightly to better match visually the real world data we obtained from the microscope, but the amounts have not yet been subjected to rigorous testing to determine optimal amounts.

```
def add_noise(image, bgnoise_amount=1, poiss_amount=350):
    """A fn to add background and poisson noise to an image, contributed by
    Julia Lyudchik, PhD student in the Danzl Group"""
    _, x0,y0 = image.shape
    #Background noise
    Nb = np.random.normal(0, 0.001, [x0,y0])
    final_Nb = image + Nb*bgnoise_amount
    final_Nb = (final_Nb-np.amin(final_Nb))/(np.amax(final_Nb) - np.amin(final_Nb))
    #Poisson noise
    final_poiss = np.random.poisson(final_Nb / np.amax(final_Nb) * poiss_amount) /
poiss_amount * np.amax(final_Nb)
    return final_poiss
```

5.3.8. Saving the Model

At the end of the training loop, the model "checkpoints" are saved. In reality, what is saved are the learned weights of the convolutional filters. For the ability to continue the training of one model using the weights of another model, as described in 6.1.10, it is also necessary to save the optimizer state.

5.3.9. Warm Start

Rather than starting with random initialization of weights for a model, it is at times advantageous to begin with the weights of a different trained model [30]. This is known as a "warm start" and can be used for transfer learning techniques, wherein a model is trained on one task and evaluated on a different one. While we are not using transfer learning, it seemed useful to have a way to continue the training of one model if the standard number of epochs proved to not be long enough. Therefore, we added an option in the code the resume training from a previous model's final layer weights.

5.4. Training curve

We make use of Tensorboard to visualize the training process; specifically, we monitor the model's training curve. This curve is one of the most useful tools to qualitatively diagnose problems that occur during training. At each step of the training loop, the batch loss is calculated

and logged to a visualization graph. Presumably, if the model is "learning" this loss will show an exponential decrease as the training continues.

5.5. Validation curve

During the creation of the dataset, 20,000 individual data points were split into three portions in a 80/10/10 split: 80% of the data was marked to be used during training, 10% was marked to be used during testing (for the trained model), and 10% is marked for validation, which is a second loop that happens intermittently during the training process. After the model parameters have been updated and the optimizer has been incremented, the model enters eval mode and is evaluated on the validation dataset. This step is used to indicate a model's generalisability or its lack thereof [30]. If a model has a regular (exponential decay) training curve but a horrible validation curve, the likelihood is high that the model has simply memorized the training data rather than actually learning to extract important features from the input [30]. The ideal situation is exponential decay for both the training and validation curves. Often the lowest achieved loss for the validation data prediction is higher than for the training data, as models typically perform slightly less well on unseen data.

6. Evaluation on Synthetic Data

To provide a quick and simple test for the trained model, we test on that segment of the synthetic data set aside for testing. Each test sample is passed through the model and outputs a vector containing a numerical value for each Zernike mode coefficient and/or offset that it was trained to predict. We can describe the accuracy of this prediction in a few quantitative ways. One of these ways is by a simple mean squared error calculation between the prediction vector and the ground truth vector, similarly to what was done during training, just without the updating of the parameters based on this loss. A low mean squared error would imply a good prediction.

The other way we found it useful to evaluate the model was by using the same synthetic pipeline to visualize the output of the model. For example, using the Net11 model described above, the output of the model is an 11-dimensional vector where each number in the vector supposedly corresponds to the weight of a Zernike mode of aberration present in the given input image. We take the numbers output by the model and generate a phase mask with them, then add it onto the phase mask of an ideal PSF and use the vector diffraction code to simulate the overall resulting PSF. If the prediction is good, this reconstruction should be very similar to the input image. We can know this computationally by calculating the Pearson correlation coefficient [42] between the two images, as:

$$\rho_{XY} = \frac{cov(X,Y)}{\sigma_X \sigma_Y}$$

Where ρ_{XY} is the correlation coefficient between two samples, X and Y, $cov(X, Y)$ is the covariance of X and Y, and $\sigma_X \sigma_Y$ is the product of the standard deviations of X and Y.

7. In-Situ Data Collection

Although accuracy on synthetically created test data is a good sanity check to have, the most useful evaluation comes from the model's performance on data collected from the microscope in a real imaging session. Establishing this procedure was at times tedious, as it required a good understanding of the existing code provided by the microscope's developer as well as the hardware of the microscope itself.

The first step of the data collection is to mount a gold bead sample immersed in oil onto the microscope system, manually focus on one bead, and try to make sure it is relatively centered within the field of view. The process of actually correcting for aberrations present in the image happens through the Spatial Light Modulator (SLM) in the beam path. All gratings, offsets, and phase masks are loaded onto the SLM, which shifts the incoming light beam appropriately. The result is monitored electronically on the viewing station. The vector diffraction code simulates this transformation from phase mask to focused PSF that the imaging optics completes in the real microscope setup.

7.1. Handling Tip/Tilt

The second step of data collection is correcting for the aberration we have not trained the models to predict. The first two Zernike modes were not included in any of the trained models. This is because these modes, known as Tip and Tilt or X-Tilt and Y-Tilt, only translate the PSF along the X-/Y- axis, but do not affect its shape.

When imaging, we calculate the center of mass of the image and find its offset from the pixel-wise center of the image (for example, [32.5, 32.5] for a 64 x 64 pixel image). The offset from the center of mass is then scaled by other system parameters (namely, wavelength, back aperture, and focal length) and used to create a phase mask which is then loaded onto the SLM. Accordingly, the image shifts to be practically in the center of the field of view (there are some rounding errors present).

Theoretically, each Zernike mode is orthogonal to every other mode, meaning that they do not interfere with one another and can be predicted independently. This may not always be true in

practice, and we found that without this centering step, the performance of any model decreased significantly.

7.2. Handling Defocus

Defocus, the 5th Zernike mode also does not affect the shape of the 3D PSF, but translates it along the optical axis. Previous work using CNNs to predict Zernike modes has only been applied to the *xy* ortho-section, where the apparent PSF changes, but the sign of defocus is visually indeterminate. [1], to counteract this problem, applies the model predictions twice -- once with positive defocus and once with negative defocus -- and chooses the best option by comparing the resulting intensity of the image. By virtue of the fact that we incorporate all three ortho-sections of the gold bead (the *xy* view, the *yz*, and the *xz* view), we are able to analytically calculate defocus external to the model and center the measured PSF also along the optical axis. Subsequently, we are able to leave it out of the training.

The process of calculating defocus is similar to calculating tip or tilt; it is some scaling of the offset of the center of mass of a PSF ortho-section. This time, the scaled offset calculation is performed on both the *xz* and *yz* ortho-sections, and the result is averaged. Like before, once the coefficient of defocus has been determined, a phase mask is generated and passed to the SLM.

7.3. Quantitative Validation

To obtain the amount of data on the scale necessary to prove reproducibility, we must automate the data collection process. To do this, we write a loop that first zeros the SLM, acquires an image from the software, fits the center of mass and then centers the stage via hardware controls. This is to counteract the likelihood of a bead "walking off" the field of view after a period of time imaging it. After that, the regular process of correcting tip/tilt, correcting defocus, and then loading the phase mask generated by the model's prediction onto the SLM. This process, while it eventually is ended by the bead walking off, has allowed us to obtain a few hundred data points for several of the trained models.

8. Discussion

The project is in an on-going state and therefore a full analysis of our results is not provided. However, to briefly summarize, this project is the creation and implementation of an auto-alignment system for coordinate-targeted super-resolution microscope techniques to replace a tedious human alignment process. The determination of the aberrations present in an acquired image is done through a CNN, which leverages compositional information in the image to predict the weights of Zernike modes. The model is trained for this kind of prediction on a large

synthetic dataset generated by vector diffraction theory that simulates real-world acquired images.

We focus most of our energy on the xy-STED PSF use case in this report, but the main advantage of our approach is its transferability; once implemented, all that is required to generate data, train, and evaluate a model for a different use case is to change out the specific data generation function. For example, our pipeline would work with little or no modification for z-STED or for fluorescence (also known as effective PSF). Using this pipeline for the effective PSF would require slight modification because, rather than imaging one effective PSF at a time, many are present within the field of view and therefore would require some kind of averaging or segmentation script. Many other patterns would be possible as well; e.g. the overlay of xy/z STED or coherent hybrid STED.

One of the most interesting further use cases to us is the Fluorescence PSF, as in practice, our group acquires better images when using the Fluorescence PSF to align rather than the STED PSF. The process for generating a synthetic effective PSF, alternatively called a fluorescence PSF, is quite similar to the above except in that rather than the phase mask of the aberrations being added onto the phase mask describing the ideal donut shape, it is added onto the phase mask of a Gaussian beam. With our pipeline in place, this is trivial to do. In fact, it would be trivial to simulate any desired pattern of minima and maxima that could be used in STED microscopy.

9. Acknowledgements

This work would not have been possible without the support and guidance of many people. Firstly, I'd like to acknowledge the Fulbright-Marshall Plan donors, who graciously funded this project, as well as the Fulbright Austria Commission, particularly Darrah Lustig and Dune Johnson, who helped facilitate my time as a Fulbright scholar. Secondly, I'd like to thank my Principal Investigator, Dr. Johann Danzl, for his guidance and support for the duration of the project and for his willingness to be my institutional sponsor during the Fulbright application process. This work certainly would not have been successful on any scale if it weren't for the troubleshooting efforts, day-to-day guidance, and endless support of Dr. Wiebke Jahr, the postdoctoral researcher associated with this project. Beyond that, I'd like to thank the rest of the Danzl Lab: Dr. Sven Truckenbrodt, Dr. Philipp Velicky, Dr. Giulio Abagnale, Dr. Caroline Kreuzinger, Marek Suplata, Julia M. Michalska, Nathalie Agudelo Dueñas, Jakob Vorlaufer, Mojtaba Tavakoli, and Julia Lyudchik. Special thanks to Rishabh Sahu for his vector diffraction code, which we incorporated into our data generation pipeline, to Julia L. for the contribution of function to generate sample noise, and to Marek and Jakob for various technical and project input and hardware support.

10. References

1. Y. Jin, Y. Zhang, L. Hu, H. Huang, Q. Xu, X. Zhu, L. Huang, Y. Zheng, H.-L. Shen, W. Gong, and K. Si, "Machine learning guided rapid focusing with sensor-less aberration corrections," Opt. Express **26**, 30162–30171 (2018).
2. Y. Zhang, C. Wu, Y. Song, K. Si, Y. Zheng, L. Hu, J. Chen, L. Tang, and W. Gong, "Machine learning based adaptive optics for doughnut-shaped beam," Opt. Express **27**, 16871–16881 (2019).
3. S. W. Hell, S. J. Sahl, M. Bates, X. Zhuang, R. Heintzmann, M. J. Booth, J. Bewersdorf, G. Shtengel, H. Hess, P. Tinnefeld, A. Honigmann, S. Jakobs, I. Testa, L. Cognet, B. Lounis, H. Ewers, S. J. Davis, C. Eggeling, D. Klenerman, K. I. Willig, G. Vicidomini, M. Castello, A. Diaspro, and T. Cordes, "The 2015 super-resolution microscopy roadmap," J. Phys. D Appl. Phys. **48**, 443001 (2015).
4. M. J. Booth, "Adaptive optics in microscopy," Philos. Trans. A Math. Phys. Eng. Sci. **365**, 2829–2843 (2007).
5. J. A. Kubby, *Adaptive Optics for Biological Imaging* (CRC Press, 2013).
6. M. J. Booth, "Adaptive optical microscopy: the ongoing quest for a perfect image," Light: Science & Applications **3**, e165–e165 (2014).
7. S. W. Hell and J. Wichmann, "Breaking the diffraction resolution limit by stimulated emission: stimulated-emission-depletion fluorescence microscopy," Opt. Lett. **19**, 780–782 (1994).
8. S. Deng, L. Liu, Y. Cheng, R. Li, and Z. Xu, "Investigation of the influence of the aberration induced by a plane interface on STED microscopy," Opt. Express **17**, 1714–1725 (2009).
9. B. R. Patton, D. Burke, R. Vrees, and M. J. Booth, "Is phase-mask alignment aberrating your STED microscope?," Methods Appl Fluoresc **3**, 024002 (2015).
10. W. Jahr, P. Velicky, and J. G. Danzl, "Strategies to maximize performance in STimulated Emission Depletion (STED) nanoscopy of biological specimens," Methods **174**, 27–41 (2020).
11. B. Harke, J. Keller, C. K. Ullal, V. Westphal, A. Schönle, and S. W. Hell, "Resolution scaling in STED microscopy," Opt. Express **16**, 4154–4162 (2008).
12. M. Booth, D. Andrade, D. Burke, B. Patton, and M. Zurauskas, "Aberrations and adaptive optics in super-resolution microscopy," Microscopy **64**, 251–261 (2015).
13. T. J. Gould, J. R. Myers, and J. Bewersdorf, "Total internal reflection STED microscopy," Opt. Express **19**, 13351–13357 (2011).
14. M. G. M. Velasco, M. Zhang, J. Antonello, P. Yuan, E. S. Allgeyer, D. May, O. M'Saad, P. Kidd, A. E. S. Barentine, V. Greco, J. Grutzendler, M. J. Booth, and J. Bewersdorf, "3D super-resolution deep-tissue imaging in living mice," 790212 (2019).
15. E. Auksorius, B. R. Boruah, C. Dunsby, P. M. P. Lanigan, G. Kennedy, M. A. A. Neil, and P. M. W. French, "Stimulated emission depletion microscopy with a supercontinuum source and fluorescence lifetime imaging," Opt. Lett. **33**, 113–115 (2008).
16. M. O. Lenz, H. G. Sinclair, A. Savell, J. H. Clegg, A. C. N. Brown, D. M. Davis, C. Dunsby, M. A. A. Neil, and P. M. W. French, "3-D stimulated emission depletion microscopy with programmable aberration correction," J. Biophotonics **7**, 29–36 (2014).
17. T. J. Gould, E. B. Kromann, D. Burke, M. J. Booth, and J. Bewersdorf, "Auto-aligning stimulated emission depletion microscope using adaptive optics," Opt. Lett. **38**, 1860–1862 (2013).
18. C. Belthangady and L. A. Royer, "Applications, promises, and pitfalls of deep learning for fluorescence image reconstruction," Nat. Methods **16**, 1215–1225 (2019).
19. L. Möckl, A. R. Roy, P. N. Petrov, and W. E. Moerner, "Accurate and rapid background estimation

in single-molecule localization microscopy using the deep neural network BGnet," Proc. Natl. Acad. Sci. U. S. A. **117**, 60–67 (2020).
20. A. Durand, T. Wiesner, M.-A. Gardner, L.-É. Robitaille, A. Bilodeau, C. Gagné, P. De Koninck, and F. Lavoie-Cardinal, "A machine learning approach for online automated optimization of super-resolution optical microscopy," Nat. Commun. **9**, 5247 (2018).
21. L. Wang, W. Yan, R. Li, X. Weng, J. Zhang, Z. Yang, L. Liu, T. Ye, and J. Qu, "Aberration correction for improving the image quality in STED microscopy using the genetic algorithm," Nanophotonics **7**, 1971–1980 (2018).
22. M. Weigert, U. Schmidt, T. Boothe, A. Müller, A. Dibrov, A. Jain, B. Wilhelm, D. Schmidt, C. Broaddus, S. Culley, M. Rocha-Martins, F. Segovia-Miranda, C. Norden, R. Henriques, M. Zerial, M. Solimena, J. Rink, P. Tomancak, L. Royer, F. Jug, and E. W. Myers, "Content-aware image restoration: pushing the limits of fluorescence microscopy," Nature Methods **15**, 1090–1097 (2018).
23. W. Ouyang, A. Aristov, M. Lelek, X. Hao, and C. Zimmer, "Deep learning massively accelerates super-resolution localization microscopy," Nature Biotechnology **36**, 460–468 (2018).
24. A. Voulodimos, N. Doulamis, A. Doulamis, and E. Protopapadakis, "Deep Learning for Computer Vision: A Brief Review," Comput. Intell. Neurosci. **2018**, 7068349 (2018).
25. B. Richards, E. Wolf, and D. Gabor, "Electromagnetic diffraction in optical systems, II. Structure of the image field in an aplanatic system," Proc. R. Soc. Lond. A Math. Phys. Sci. **253**, 358–379 (1959).
26. A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," Communications of the ACM **60**, 84–90 (2017).
27. V. Lakshminarayanan and L. Srinivasa Varadharajan, "Zernike Polynomials," Special Functions for Optical Science and Engineering (n.d.).
28. K. I. Bae, J. Park, J. Lee, Y. Lee, and C. Lim, "Flower classification with modified multimodal convolutional neural networks," Expert Syst. Appl. **159**, 113455 (2020).
29. Y. Chen, "LSTM recurrent neural network prediction algorithm based on Zernike modal coefficients," Optik **203**, 163796 (2020).
30. S. Saha, "A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way," https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53.
31. V. Jain, "Everything you need to know about “Activation Functions” in Deep learning models," https://towardsdatascience.com/everything-you-need-to-know-about-activation-functions-in-deep-learning-models-84ba9f82c253.
32. S. Cai, Y. Shu, G. Chen, B. C. Ooi, W. Wang, and M. Zhang, "Effective and Efficient Dropout for Deep Convolutional Neural Networks," arXiv [cs.LG] (2019).
33. "PyTorch," https://pytorch.org/.
34. "TensorBoard," https://www.tensorflow.org/tensorboard.
35. D. Mack, "How to pick the best learning rate for your machine learning project," https://medium.com/octavian-ai/which-optimizer-and-learning-rate-should-i-use-for-deep-learning-5acb418f9b2.
36. S. Doshi, "Various Optimization Algorithms For Training Neural Network," https://towardsdatascience.com/optimizers-for-training-neural-network-59450d71caf6.
37. D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," arXiv [cs.LG] (2014).
38. N. S. Keskar and R. Socher, "Improving Generalization Performance by Switching from Adam to SGD," arXiv [cs.LG] (2017).
39. S. Bock, J. Goppold, and M. Weiß, "An improvement of the convergence proof of the ADAM-Optimizer," arXiv [cs.LG] (2018).
40. A. Collette and Others, "h5py: HDF5 for Python, 2018," URL www. h5py. org (n.d.).

41. S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," arXiv [cs.LG] (2015).
42. J. Benesty, J. Chen, Y. Huang, and I. Cohen, "Pearson Correlation Coefficient," in *Noise Reduction in Speech Processing*, I. Cohen, Y. Huang, J. Chen, and J. Benesty, eds. (Springer Berlin Heidelberg, 2009), pp. 1–4.