Electrical and Electronics Engineering

EE 677 Report - Dec. 27, 2022

Examining the application of modern neural networks to computer image precompensation for refractive error correction.

ADEREMI, Adeola.

Instructor:
Prof. Dr. EMIN ANARIM

Abstract

Wavefront aberrations, which occur when light is not perfectly focused on the retina of the eye, can be classified as low or high order aberrations. Low order aberrations, also known as refractive errors, are the most common type and include myopia, hyperopia, and astigmatism. These errors often cause blurry vision and can be corrected using deep learning methods to estimate a precompensated image that appears clear to individuals with refractive errors when viewed on a computer. This project aims to correct refractive errors and improve image clarity for individuals with this type of vision impairment by correcting the image from the source using a Deep Learning method. We have been able to get relatively clear color images for different blur levels.

Keywords

Here your keywords: eg. **Deep Learning; Detection and Estimation Theory; Bogazici University; Blur; Wavefront aberrations**

Contents

1. Introduction

Wavefront aberrations is the general term used to describe the phenomenon whereby light rays coming from an object is not perfectly focused on the retina of the eye of the observing human. They are generally classified as low order or high order aberrations. Low order aberrations are the most common type of aberrations (making up about 85% of observed wavefront aberrations [1]) and are commonly known as refractive errors. The most common type of refractive errors in humans include mypoia, hyperopia and astigmatism. Refractive errors usually manifest themselves as blurry visions in humans. This means that when a person with refractive errors look at an image, whether in real life or on a digital screen, what they see is a blurred version of the true image. In this project, we have tried to correct this while viewing images on computers by using deep learning methods to estimate a precompensated image which would give a true (i.e unblurred) image when viewed by a user with refractive errors.

2. Related Work

Refractive errors are usually solved by wearing prescription glasses. Although disputed, the use of glasses for correcting refractive errors like myopia can lead to a worsening of the situation [2]. Additionally, wearing glasses can come with other challenges like cost, broken lenses, and people's fear that lenses actually ruin their eyes [3]. Our proposed solution, that is precompensating images from the source, solves this. Both [4] and [5] have also proposed different methods of precompensating images at the source as a solution to the blurring effect of refractive errors. In [4], the precompensated image was calculated using a minimum mean squared error wiener filter. They reported results on grayscale images which, although did recover the true images, produced images with lower contrast than the actual image. In our work, we have estimated the same precompensated image by using Deep Neural Networks which minimized the mean of the square error between what the output from an eye with refractive errors would see and the true image. We have also worked exclusively on color images to reflect more what users would typically interact with.

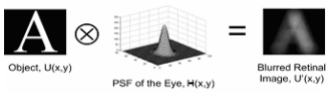


Figure 1: The eye as an LTI system performing convolution

3. Solution

Our estimation problem was solved using a deep learning method. Our deep learning model estimates and outputs the precompensated image that we desire. The eye can be modeled as an LTI system that performs a convolution on whatever is input to it [4]. For this reason, to model the behaviour of eyes with refractive errors, we have convolved our deep learning model output with a Gaussian Blur. The foregoing can be represented mathematically as follows:

$$output = f(g(x)).$$

Where both the f and the g are convolutions and x is the input image. f is the blur introduced by the eyes of the observer and the g is the precompensated image estimated with our Deep Neural Network.

3.1. Deep Learning Overview

Machine learning techniques have been used to automate detection and estimation tasks. Say one wants to classify dogs vs cats from their images, several pictures could be obtained and then one could do feature extraction to get useful features or representations from the images. These features can now be fed to a linear classifier which then classifies a test pattern as a cat or a dog. Deep learning methods simplify this classification process by removing the need to manually perform feature extraction. They automate this by using different layers to learn increasingly complicated features which can then be used to classify the test pattern into the correct class [6]. A typical deep neural network with linear layers is shown in Figure 2. The learning is done by using gradient descent to modify the parameters of the deep learning networks such that some sort of objective function is minimized. Not only can they be used for detection, they can also be used for estimation, which is was we have done here. We have used our deep learning network to estimate the precompensated image such that the mean squared error between the true image and the model output, after it is blurred, is minimized.

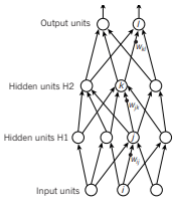


Figure 2: DNN architecture with linear layers.

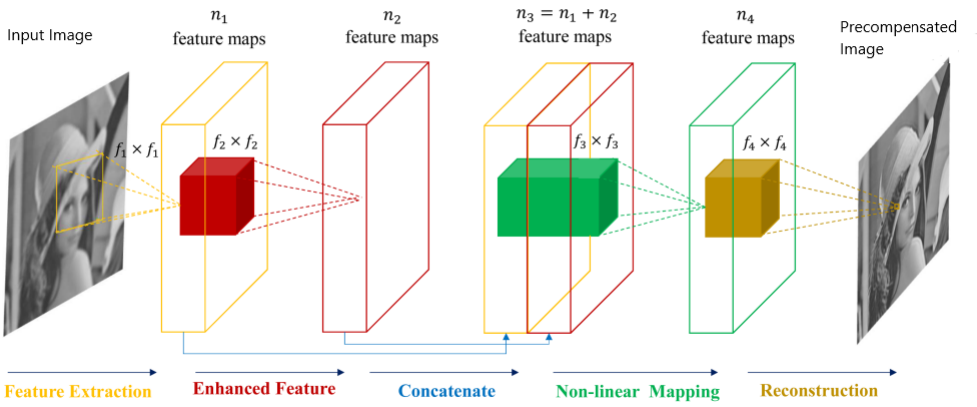


Figure 3: Architecture for our proposed solution.

3.2. Architecture and Description

The basic idea is that given an input image $\mathbf{x}$, applying the blur operation on the the model output $F(\mathbf{x})$ leads us to obtain the same image as the input, i.e $\mathbf{x}$. Here, the blur serves to be a model of the eye and the model output is the precompensated image such that when seen by a person with refractive index errors the correct image is seen. Our model is largely inspired by [7], who used it to perform image deblurring and superresolution, and is shown in figure 3.

The deep neural network here uses only four hidden layer, one of which is just a concatenation operation and hence does not have learnable parameters. Each layer applies a 2D convolutional neural network with padding. Padding the input ensures that the height and width of the input is the same as that of the output which allows to perform concatenation later on. The operation

of this network can be summarised as follows:

$$F_i(\mathbf{x}) = max(0, W_i * F_{i-1}(\mathbf{x}) + b_i), i \in 1, 2, 4$$

$$F_3(\mathbf{x}) = concat(F_1(\mathbf{x}), F_2(\mathbf{x}))$$

$$F(\mathbf{x}) = W_5 * F_4(\mathbf{x}) + b_5$$

Where the $max(0, f)$ operator is known as the ReLU operator which serves to provide nonlinearities in each layer since convolution itself is a linear operator but the functions we are looking to learn are, in general, not linear. The $*$ operator represents convolution, the $W's$ are the learnable parameters(weights) and the $b's$ are the biases.

Like was done in [7], we used 32 $9X9$ filters to generate 32 feature maps. From these using 32 $5X5$ filters we generate another set of 32 feature maps. These were concatenated to form 64 feature maps then the fourth layer maps these into another set of 32 feature maps using a filters of size 1. This is then mapped back into our 3 channel color image.

4. Results and Discussion

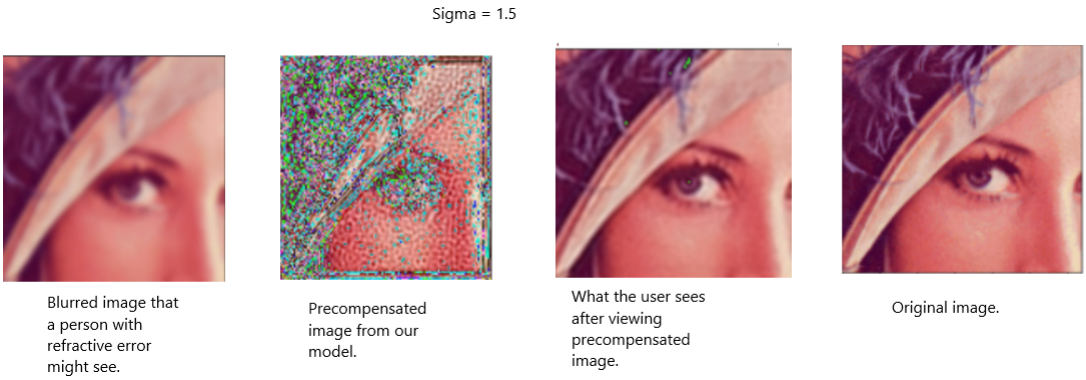
4.1. Experimentation protocol

The model was trained on the set14 dataset which is a dataset used for testing image superresolution models[1] using the PyTorch Lightning Deep Learning framework in Python. And it was tested on the set5 from the same dataset. The dataset was obtained from [8]. Some of the results from the model with and without blur and the original inputs are shown in figures 4 and 5. It can be seen that the obtained final image is very similar to the input in most cases. The Mean Squared Error (MSE) is also shown in the table 1.

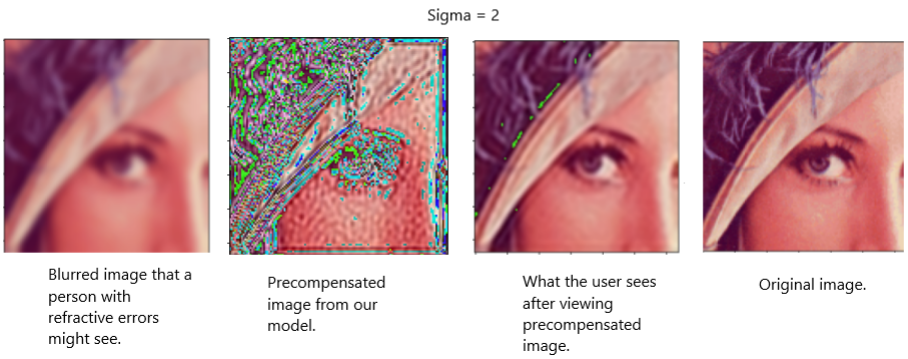
Mean Squared Error		
Data	$\sigma = 1.5$	$\sigma = 2$
Train	0.00177	0.00259
Test	0.0008	0.0013

Table 1: Mean Squared Error on Train and Test data.

[1] Paperswithcode



: Figure 4
Results for blur level $\sigma = 1.5$



: Figure 5
Results for blur level $\sigma = 2$

5. Discussion and Conclusion

The discussion section focuses on the main challenges/issues you had to overcome during the project. Outline what your approach does better than the ones you mentioned in your related work, and explain why. Do the same with issues where other solutions outperform your own. Are there limitations to your approach? If so, what would you recommend towards removing/mitigating them? Given the experience you've gathered working on this project, are there other approaches that you feel are worth exploring?

We have been able to apply precompensation to color images using a deep neural network and to obtain reconstructed images that look very much like the input. We have been able to obtain clear color images for different blur levels. While they used a particular blurring (PSF) in their calculations, our method can be trained to work on the different PSFs, as shown by the blur levels employed. We have currently used a Gaussian Blur to simulate the PSF, in future works we can try different types of blurring techniques such as the motion blur and others that might better approximate what a human Point Spread Function (PSF) might look like and see how well our method works in those cases. A popular neural network architecture, called an Autoencoder, was also tested while experimenting but it did not work as well as the architecture we have settled for. Code for this and for our final architecture are included in the appendix.

References

[1] M. Resan, M. Vukosavljević, and M. Milivojević, "Wavefront aberrations, advances in ophthalmology," *InTech*, 2012. [Online]. Available: http://www.intechopen.com/books/advances-in-ophthalmology/wavefront-aberrations

[2] A. Medina, "The progression of corrected myopia," *Graefes Arch Clin Exp Ophthamol*, August 2015.

[3] A. A. Ayanniyi, F. G. Adepoju, R. O. Ayanniyi, and R. E. Morgan, "Challenges, attitudes and practices of the spectacle wearers in a resource-limited economy," *Middle East Afr J Ophthalmol*, January 2010.

[4] M. A. Jr., A. Barreto, and J. G. Cremades, "Image pre-compensation to facilitate computer access for users with refractive errors," *Behaviour and Information Technology*, vol. 24, no. 3, pp. 161–173, January 2005.

[5] O. Keleş and E. Anarim, "Adjustment of digital screens to compensate the eye refractive errors via deconvolution," in *2019 Ninth International Conference on Image Processing Theory, Tools and Applications (IPTA)*, 2019, pp. 1–6.

[6] Y. LeCun, Y. Bengio, and H. G., "Deep learning," *Nature*, vol. 521, pp. 436 – 444, May 2015.

[7] F. Albluwi, V. A. Krylov, and R. Dahyot, "Image deblurring and super-resolution using deep convolutional neural networks," in *2018 IEEE 28th International Workshop on Machine Learning for Signal Processing (MLSP)*, 2018, pp. 1–6.

[8] J.-B. Huang, A. Singh, and N. Ahuja, "Single image super-resolution from transformed self-exemplars," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 5197–5206.

A. Appendix A: Our Architecture. Python Implementation

```
class LitEncoder(pl.LightningModule):
  def __init__(self):
   super().__init__()
   #changed values from 1 to 3 here for color images as well
   self.low = nn.Conv2d(3, 32, 9, padding=4)
   self.high = nn.Conv2d(32, 32, 5, padding=2)
   self.second_order = nn.Conv2d(64, 32, 1)
   self.decoder = nn.Conv2d(32, 3, 5, padding = 2)
   self.blur = transforms.GaussianBlur(121, 2)

  def forward(self, x):
    low_feats = nn.ReLU()(self.low(x))
    high_feats = nn.ReLU()(self.high(low_feats))
    concat = torch.cat((low_feats, high_feats), 1)

    second_order_feat = nn.ReLU()(self.second_order(concat))

    embedding = self.decoder(second_order_feat)

    return embedding

  def configure_optimizers(self):
    optimizer = torch.optim.Adam(self.parameters(), lr=2e-3)
    scheduler = torch.optim.lr_scheduler.CosineAnnealingWarmRestarts(optimizer, T
    return {
        "optimizer": optimizer,
        "lr_scheduler": {
            "scheduler": scheduler
        }
    }

  def training_step(self, train_batch, batch_idx):
    y, x = train_batch
    #x = x.view(x.size(0), -1)
    x_hat = self.blur(self(x))
    loss = F.mse_loss(x_hat, y)
    self.log('train_loss', loss)
    return loss

  def validation_step(self, val_batch, batch_idx):
    y, x = val_batch
    #x = x.view(x.size(0), -1)
    x_hat = self.blur(self(x))
    loss = F.mse_loss(x_hat, y)
    self.log('val_loss', loss)
```

B. Appendix B: Autoencoder Implementation

```
class LitAutoEncoder(pl.LightningModule):
        def __init__(self):
                super().__init__()
                self.encoder = nn.Sequential(
      nn.Conv2d(1, 32, 3, padding=1),
      nn.MaxPool2d(2),
      nn.ReLU(),
      nn.Conv2d(32, 64, 3, padding=1),
      nn.MaxPool2d(2),
      nn.ReLU(),
      nn.Conv2d(64, 128, 3, padding=1),
      nn.MaxPool2d(2),
      nn.ReLU(),
      nn.Conv2d(128, 256, 3, padding=1),
      nn.MaxPool2d(4),
      nn.ReLU(),
      nn.Flatten(),
      nn.Linear(4096, 256),
      nn.ReLU(),
      nn.Linear(256, 64))
                self.decoder = nn.Sequential(
      nn.Linear(64, 256),
      nn.ReLU(),
      nn.Linear(256, 4096),
      nn.ReLU(),
      nn.Unflatten(1, (16, 16, 16)),
      nn.Upsample(scale_factor=2),
      nn.Conv2d(16, 32, 3, padding=1),
      nn.ReLU(),
      nn.Upsample(scale_factor=2),
      nn.Conv2d(32, 32, 3, padding=1),
      nn.ReLU(),
      nn.Upsample(scale_factor=2),
      nn.Conv2d(32, 1, 3, padding=1)
      )

        def forward(self, x):
                embedding = self.encoder(x)
                return embedding

        def configure_optimizers(self):
                optimizer = torch.optim.Adam(self.parameters(), lr=1e-4)
                return optimizer

        def training_step(self, train_batch, batch_idx):
                y, x = train_batch
                #x = x.view(x.size(0), -1)
                z = self.encoder(x)
                x_hat = self.decoder(z)
```

```
        loss = F.mse_loss(x_hat, y)
        self.log('train_loss', loss)
        return loss

    def validation_step(self, val_batch, batch_idx):
        y, x = val_batch
        #x = x.view(x.size(0), -1)
        z = self.encoder(x)
        x_hat = self.decoder(z)
        loss = F.mse_loss(x_hat, y)
        self.log('val_loss', loss)
```