Senior Thesis in Mathematics

Appearance-based Eye-tracking using Convolutional Neural Networks

Author:
Zihao Xu

Advisor:
Dr. Alexandra Papoutsaki

Submitted to Pomona College in Partial Fulfillment
of the Degree of Bachelor of Arts

August 25, 2019

Abstract

In this thesis, we explore the intersection of the field of eye-tracking and the field of computer vision to build an efficient appearance-based eye-tracker using Convolutional Neural Networks (CNNs). We first build the mathematical framework of Artificial Neural Network (ANN) and transition to its variation, Convolutional Neural Networks (CNN), in addition to their training methods. Then, we provide a holistic overview of the field of eye-tracking and several distinguished existing eye-tracking tools, laying the ground work for our model. Finally, we present two CNN-based eye tracking models using the MobileNet(5) and the ResNet50(4) architectures trained on dataset obtained from the WebGazer(13) project. Through experimental results, we demonstrate that our models offer a 25% and 10% performance improvements measure by average euclidean distance for PC and Mac users, respectively. In addition, we show that our CNN-based models have the potential to serve as blink detectors.

Contents

1 Introduction

2 Artificial Neural Network (ANN)
- 2.1 The Perceptron Algorithm . . .
- 2.2 Artificial Neural Networks (ANNs) . . .
- 2.3 Backpropagation . . .
  - 2.3.1 Loss Function . . .
  - 2.3.2 Calculate Gradients Using Computational Graphs . . .
  - 2.3.3 Gradient Descent . . .
- 2.4 Model Architecture . . .

3 Convolutional Neural Networks (CNNs)
- 3.1 History of CNNs . . .
- 3.2 Representation of Images . . .
- 3.3 Convolutional Neural Networks . . .
  - 3.3.1 Convolution (CONV) Layer . . .
  - 3.3.2 Max-pooling layer . . .
- 3.4 Model Architecture . . .

4 Eye-tracking
- 4.1 Types of Estimation Models . . .
  - 4.1.1 2D Models . . .
  - 4.1.2 3D Models . . .
  - 4.1.3 Appearance-based Models . . .
- 4.2 Types of Evaluation Methods . . .
  - 4.2.1 Actual Distance . . .
  - 4.2.2 Angular Resolution . . .
  - 4.2.3 Block-based Classification Accuracy . . .

4.2.4 Cross-subject and Cross-dataset Validation . . . . . .
4.3 Distinguished Work . . . . . .
4.3.1 GazeNet . . . . . .
4.3.2 iTraker . . . . . .
4.3.3 Block-based CNN . . . . . .
4.3.4 WebGazer . . . . . .

5 Methods
5.1 Dataset Description . . . . . .
5.1.1 Demographics . . . . . .
5.2 Pre-processing . . . . . .
5.2.1 Label Creation . . . . . .
5.2.2 Facial Landmark Detection . . . . . .
5.2.3 Image Pre-processing . . . . . .
5.3 Selected Model Architectures . . . . . .

6 Experimental Results
6.1 Validation Set Setup . . . . . .
6.2 Model Performance . . . . . .
6.3 Comparison with WebGazer . . . . . .
6.4 Side-effect - Blink Detection . . . . . .

7 Conclusion

Chapter 1

Introduction

Eye-tracking is the process of monitoring the eye movements of an individual and calculating the direction of their gaze(14). In this thesis, we define eye tracking as the mapping of the gaze direction to the coordinate system of a computer screen. There are two main categories of eye trackers: i) dedicated external devices with specialized hardware and ii) software solutions. Here, we focus on the second category and investigate the process of gaze prediction by mapping webcam images of the user's face and eyes to screen coordinates. With accurate information on users' gaze locations and patterns, researchers employing eye-tracking technologies can better understand behavioral habits and gain insights into strategic placements of web content for enhanced user experience or commercial purposes. At the same time, this technology allows traditional eye-tracking researchers to conduct their studies remotely while having participants at the convenience of their homes (14). Due to a wide range of applications, the field of eye-tracking has attracted a lot of attention from both the academia and the industry (12).

Setting aside the appeal of eye-tracking, developing an accurate eye-tracker can pose a lot of practical problems and algorithmic challenges. Although there have been several devoted attempts to build eye-tracking tools and software, the end products usually suffer from several drawbacks: i) high cost of dedicated eye-tracking devices, ii) excessive external constraints, such as limited head positioning and usage of infrared lights, and iii) inaccuracy and variability in gaze predictions. To address the first and second problem, researchers developed eye-tracking algorithms leveraging the power of webcams, which are readily available to most laptop and desktop users.

Lightweight machine learning models take video streams coming from webcams as input and make real-time gaze predictions after a short phase of calibration. One such example is WebGazer, an eye-tracking JavaScript library developed by Papoutsaki et al. (13). However, due to model constraints and the limited computing resources, there is still room for improvement for WebGazer, both in terms of prediction accuracy and stability, by reducing its prediction variance.

Parallel to the popularity of eye-tracking, the field of computer vision has witnessed a series of breakthroughs, initiated by the renowned AlexNet in 2012 (9). AlexNet is the first dedicated effort and successful deployment of what is know as Convolutional Neural Networks (CNNs) to classify images depicting various objects. Subsequently, researches began to widely adopt CNNs for a variety of tasks in computer vision, including object detection, image captioning, and even image synthesis (6).

Due to their effectiveness, CNN models are also adopted by researchers of eye-tracking and have yielded successful results. Krafka et al. (8) introduced iTracker, a CNN-based model that predicts gaze locations on Apple tablets and cellphones with reasonable accuracy.

In light of such circumstances, this thesis explores the effectiveness of applying CNN-based models on PC and Mac users within the EOTT dataset (13) collected as part of the WebGazer project. To facilitate comparison, we will build several models using two different architectures, MobileNet (5) and ResNet50 (4). Besides, we will also critically evaluate the model performances and compare them with WebGazer's performance.

The following chapters will be organized as follows: first, I will introduce Artificial Neural Networks (ANNs) in Chapter 2, and, immediately following that, Convolutional Neural Networks (CNNs) in Chapter 3. I will discuss the history and underpinning mathematical formulation of ANNs and CNNs, laying the theoretical groundwork for this thesis. Then, in Chapter 4, I will discuss in detail previous attempts at using CNN models for eye-tracking and gaze estimation. Following the background information, I will document the methodology in Chapter 5 and experimental results in chapter 6. Finally, I will conclude the thesis and propose several future directions of this research in Chapter 7.

Chapter 2

Artificial Neural Network (ANN)

Aritifical Neural Networks (ANNs) are the underpinning model in the field of "deep learning". ANNs, alongside their variations, are widely used in the field of Artificial Intelligence and Machine Learning, manifesting their power in a broad range of applications. To understand ANN models, we need to start with the perceptron algorithm (15) proposed by Rosenblatt in 1958. We will cover the standard perceptron algorithm, its modern variation used in ANNs, and how we can build ANNs with layers of perceptrons. This chapter will be concluded by the concept of backpropagation (10), the method used to efficiently train ANN models.

2.1 The Perceptron Algorithm

The perceptron algorithm represents a single unit within a standard neural network. To fully understand this algorithm, we will start introducing a naive version used in binary classification. Essentially, this class of perceptrons has a bias term, b, and a vector of weight terms, W, whose entries are associated with different input channels. Presented with an input vector, X_i, with matching number of channels, the perceptron first computes the dot product between the input vector, X_i, and the weight vector, W, and then sums up the dot product and the bias term, b. After that, the perceptron tests if the sum passes a certain threshold value T. If the threshold is surpassed, the output of the perceptron is 1; otherwise the output is -1. Assuming that the

number of channels is C, the above calculation can be summarized as:

$$output = \begin{cases} 1, & \text{if } \sum_{k=1}^{C} w_k X_{ik} + b > T \\ -1, & \text{otherwise} \end{cases} \tag{2.1}$$

where X_{ik} refers to the k^{th} feature of the input X_i. To simplify this setup, note that we could move the threshold term T to the left of the equation, to get $\sum_{k=1}^{C} w_k X_{ik} + (b - T) > 0$. Thus, we could encode both the threshold and the bias by assigning to the bias term, b, the value of $b - T$ and the perceptron would simply test if the quantity $\sum_{k=1}^{C} w_k X_{ik} + (b - T)$ is positive.

To train a perceptron, initially, we randomly initialize the weights and bias terms of the perceptron to indicate no prior knowledge. Then, training examples are iteratively passed to the percetron one at a time, and we adjust the weights for the perceptron if the prediction using the current weights does not match the class for each new input. Note that this process is also known as *online learning*.

As an example, assume that we have an input $X_i = (-1, 1)$ whose label is $Y_i = 1$, a weight vector $W = (1, 0)$, and a bias term $b = 0$. The perceptron calculates $\sum_{k=1}^{C} w_k X_{ik} + b = -1 + 0 = -1 < 0$ and thus classifies X_i as -1. This disagrees with the true label of X_i and thus, the perceptron will adjust its weights so that the prediction for that data point becomes 1 (some plausible new weights are $(0, 1)$ and $(-1, 1)$).

A more rigorous formalization of this type of perceptron can be found in algorithm 1 below. Note that in this algorithm, the update[1] we performed on W and b is just one of many methods to change the weights. We can also add in a discount factor, α, for each of the quantities updated to make the change less abrupt.

However, observe the following example with 4 observations, $X_1 = (-1, -1)$, $X_2 = (1, 1)$, $X_3 = (-1, 1)$, $X_4 = (1, -1)$ with corresponding labels $Y_1 = 1$, $Y_2 = 1$, $Y_3 = -1$, $Y_4 = -1$ (the XOR relationship), which are not linearly separable. In such a case, the perceptron, a linear classifier, is unable to correctly classify all four points. In other words, the algorithm is not flexible enough to capture such more complicated relationships between the inputs and their labels.

[1] To demonstrate its correctness, going back to the example we have above, we update the weights $W = (1, 0) + (-1, 1) \cdot 1 = (0, 1)$ and the bias $b = 0 + 1 = 1$. After this update, the new $\sum_{k=1}^{C} w_k X_{ik} + b = 1 + 1 = 2 > 0$, so the prediction becomes 1 and is equal to the actual label.

Algorithm 1 Perceptron Algorithm

INPUT:

$X_1, X_2, \ldots, X_n$: observations $iter$: number of iterations

$Y_1, Y_2, \ldots, Y_n$: class labels for observations

OUTPUT:

W: the trained weight vector b: the trained bias term

function TRAIN_PERCEPTRON($\{X_1, X_2, ..., X_n\}, \{Y_1, Y_2, ..., Y_n\}, iter$)
  $W, b \leftarrow$ randomly initialize
  for i in 1:$iter$ **do**
    pred $\leftarrow \sum_{k=1}^{C} w_k X_{ik} + b$
    // If the prediction disagrees with the actual class
    if $pred * Y_i \leq 0$ **then**
      $W \leftarrow W + X_i \cdot Y_i$
      $b \leftarrow b + Y_i$
    end if
  end for
  return W, b
end function

This problem of the standard perceptron algorithm led to an extension that replaces the threshold, T, with a non-linear transformation, $f(\cdot)$. In other words, the output value of the perceptron now becomes $f(\sum_{k=1}^{C} w_k X_{ik} + b)$. Some common choices for the non-linearity include:

$$sigmoid(x) = \frac{1}{1 + e^{-x}}$$

$$tanh(x) = \frac{2}{1 + e^{-2x}} - 1$$

$$ReLu(x) = max(x, 0)$$

These functions are also known as "activation functions". It is interesting to observe that the above "threshold" method can also be encoded as the following activation function:

$$\text{hard\_threshold}(x, T) = \begin{cases} 1, & \text{if } x > T \\ -1, & \text{otherwise} \end{cases} \tag{2.2}$$

We will revisit these functions in Chapter 3.

With the help of these activation functions, we can adapt the perceptrons to learn more complex spaces formed by the input observations and, more importantly, their mapping to the labels. Even better, we are able to generalize the perceptron algorithm, which is strictly used in a classification setting, to both regression and classification settings. In the next section, we will discuss how individual perceptrons can be organized into layers to form ANN models.

2.2 Artificial Neural Networks (ANNs)

In this section, I will discuss how individual perceptrons can form feed-forward, multi-layer Artificial Neural Networks.

In essence, ANNs are formed by three types of layers: an input layer, several hidden layers, and an output layer. The input layer is similar to the "input channels" we described in the previous section. Each node in the input layer takes in a value corresponding to a single feature, X_{ik}, from the input observation, X_i. This means the number of nodes in the input layer is the same as the number of input features. The hidden layers are each formed by a number of perceptrons, whose inputs include **all** the values coming from the most immediate previous layer. Output of the hidden layers depends on type of the next layer: if the next layer is another hidden layer, outputs of the perceptrons within the current hidden layer are fed to **all** perceptrons in the next layer; if the next layer is the output layer, outputs of the perceptrons in the last hidden layer will be linearly combined to compute inputs to nodes of the output layer. Finally, the output layer simply yields its inputs as the final predictions.

It is important to note that, the number of nodes in the output layer depends on the problem we want to solve. In a regression setting, the output layer has one node that produces the numeric prediction. However, in a classification setting, the output layer typically has the same number of nodes as the number of classes, and the outputs are transformed by the $softmax$ function to represent discrete probabilities of each class. Assuming that we have K classes of inputs and we use z_j to denote the j^{th} value before the $softmax$ function, then the j^{th} prediction of the output layer is calculated by:

$$softmax(z_j) = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$

With these three types of layers, input observations come in from the input layer, pass through the dot products and non-linear transformations of the hidden layers, and finally output from the output layer. Since this process involves "feeding" values in the forward direction from the input layer to the output layer, we refer to this type of ANNs as "feed-forward" ANN models. A graphical representation of a four-layer, feed-forward ANN is shown in Figure 2.1:

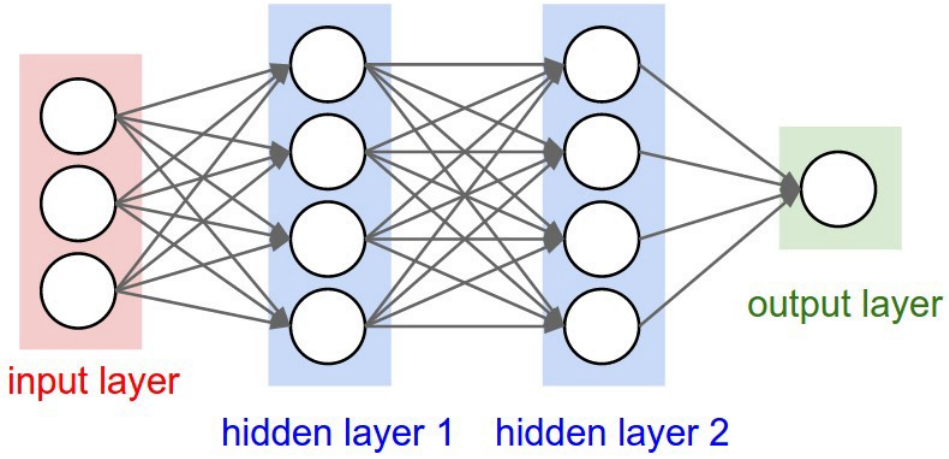


Figure 2.1: This figure show a sample two-layer ANN model.

Note that this ANN has two hidden layers, each with 4 perceptrons. If we put more perceptrons per layer or add more hidden layers to the network, the model would become more sophisticated and will be able to model more complex input spaces.

Similar to the perceptron, ANN models require training before they can make a reasonable prediction on new input data. In the next section, we will discuss backpropagation, the method that efficiently trains an ANN model.

2.3 Backpropagation

Backpropagation (10), proposed by Lecun et al., is the method used to train a neural network. Its primary aim is to minimize a defined loss function so

as to maximize the performance of the model.

2.3.1 Loss Function

The process of backpropogation starts by defining a loss function. For classification tasks, the most commonly used loss function is *categorical cross-entropy*, or CE. Given a sample of size N, a vector of labels y, and a vector of predictions ($\hat{y}$), we can calculate CE by:

$$CE(\hat{y}, y) = -\frac{1}{N}(\sum_{1}^{N} y_i \cdot log(\hat{y}_i))$$

For regression tasks, a common loss function is the Mean Squared Error, or MSE. Given a sample of size N, a vector of labels y, and a vector of predictions ($\hat{y}$), we can calculate MSE by:

$$MSE(\hat{y}, y) = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2$$

After a loss function is specified, we will use a technique called *gradient descent* to optimize the model with regard to the loss function.

2.3.2 Calculate Gradients Using Computational Graphs

Before discussing the actual gradient descent algorithm, we need to examine how to calculate the gradients by computing the partial derivatives of the loss function with respect to each of the input variables. To achieve this, we need to define what is called a "computational graph".

Essentially, a computational graph is a connected and directed graph, where its nodes correspond to variables or operations. Variables feed their values to operation nodes, while operation nodes perform specified operations and output values to their neighbors (other operations down the line).

Figure 2.2 illustrates a simple example of a computational graph. Nodes x and y feed their values into the $+$ node, which adds the values up and outputs the sum, q. Then the output q and the value from node z are sent to the $*$ node, where their product is calculated to form the final output f. Note that this computational graph can also be represented as a function, $f(x, y, z) = z(x + y)$.

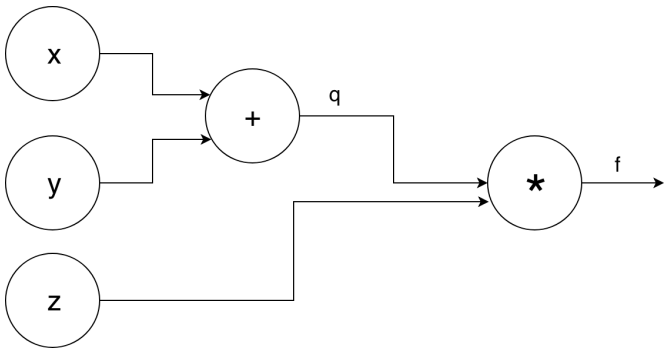


Figure 2.2: This figure show a simple computational graph that calculates $z(x + y)$.

Then, we calculate the partial derivative of f with respect to each of the inputs, x, y, and z. For this particular example, we first observe the sub-functions that make up f:

$$q = x + y$$

$$f = qz$$

The derivatives of these two functions can be computed as:

$$\frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1$$

$$\frac{\partial f}{\partial q} = z, \quad \frac{\partial f}{\partial z} = q$$

Then, using the chain rule, we obtain:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \cdot \frac{\partial q}{\partial x} = z \cdot 1 = z$$

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q} \cdot \frac{\partial q}{\partial y} = z \cdot 1 = z$$

$$\frac{\partial f}{\partial z} = q = x + y$$

More generally, we can apply the chain rule to solve extremely complicated, nested, but differentiable operations. This is achieved by only looking

at the local gradient and propagate it back to previous nodes. In Figure 2.3, we pass two inputs, x and y, to the function, f, to obtain the output z. After one forward pass of the entire computation is completed, we evaluate some chosen loss function to obtain the loss value, L. Assuming that the upstream gradient $\frac{\partial L}{\partial z}$ is provided, we simply need to calculate the local gradient of z with respect to x and y - $\frac{\partial z}{\partial x}$ and $\frac{\partial z}{\partial y}$ - and apply the chain rule to obtain the partial derivatives of L with respect to both x and y.

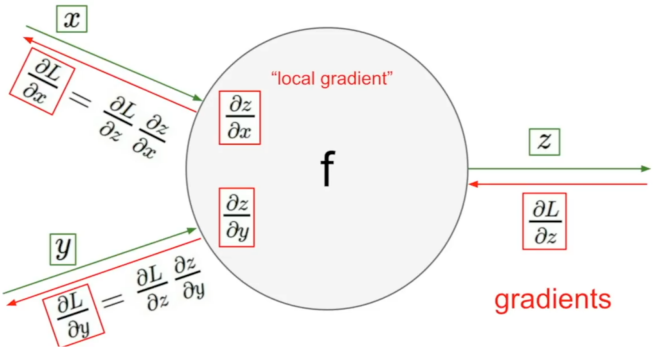


Figure 2.3: This figure illustrates back propagation of gradients using local gradients.

This local structure can be universally applied to more complex networks with arbitrarily complex functions, as long as the function can be represented as differentiable components in a computational graph. The function f can be as simple as addition, multiplication, or more complicated functions. Note that complex functions can oftentimes be expressed using computational graphs with simple operations. We could also simplify the amount of computation by pre-calculating the analytical derivatives of these functions.

Take the *sigmoid* function as an example. It is given by:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

And its analytical derivative is calculated as:

$$\frac{d\sigma(x)}{x} = \frac{e^{-x}}{(1 + e^{-x})^2} = \frac{1 + e^{-x} - 1}{1 + e^{-x}} \frac{1}{1 + e^{-x}} = (1 - \sigma(x))\sigma(x)$$

Combining the power of local gradients and such analytical derivations, we are able to gauge the magnitude of contribution from each input node

to the loss function, which could guide the training process of weights and biases within the network.

2.3.3 Gradient Descent

With an understanding of how gradients can be calculated, we are ready to discuss the actual gradient descent algorithm. The essential idea behind gradient descent is that we iteratively take steps toward the deepest gradient to walk down the loss terrain until we minimize the loss. Computational graphs provide us with a powerful means to back-propagate and distribute the final loss value to each of the inputs. Note that we only know the direction and, to some degree, the magnitude that weights in the perceptrons should adjust, but it is still unclear exactly how we should update the weights. Additionally, it is also unclear when to stop the descending process. This leads us to a discussion of the specific methods that allow us to perform the actual "descent" step and to train the network efficiently.

Below, we will introduce the vanilla version of gradient descent and how it can be modified to form the *Stochastic Gradient Descent* method.

Vanilla Gradient Descent

In the vanilla version of gradient descent algorithm, we assume that we have a function, *calculate_gradient*, that takes the data, *data*, the loss function, *loss_fn*, and weights, *W*, and returns the gradients of the loss function with respect to each of the input variables. Additionally, we have a tuning parameter, *learning_rate*, that scales the vector of gradients before we update the weights themselves.

With these two components, we can form a iterative process of: i) calculating the gradients and ii) updating the weights using the gradients and the specified *learning_rate*, until a certain stopping condition is met. Candidates for the stopping condition abound. Some examples include a fixed number of iterations (i.e. the two-step update procedure will stop when a fixed number of iteration is performed), or, if we use a validation-set approach to monitor the training process, when the validation loss stops decreasing (i.e. the model starts to over-fit the training data). Algorithm 2 formalizes the vanilla gradient descent algorithm:

Algorithm 2 Vanilla Gradient Descent Algorithm

INPUT:

data: the training data, *iter*: number of iterations
learning_rate: training rate, $loss\_fn$: the loss function

OUTPUT:

W: the optimized weights for the ANN model

function VANILLA_GRADIENT_DESCENT($data, loss\_fn, iter, learning\_rate$)
  $W \leftarrow$ randomly initialize
  for i in 1:$iter$ **do**
    $weights\_grad \leftarrow$ calculate_gradient($data, loss\_fn, W$)
    $W \leftarrow W - learning\_rate \times weights\_grad$
  end for
  return W
end function

Note that this version of the algorithm uses a fixed number of iterations. We can also replace the for loop with a $while$ loop and not terminate the training process until, for example, the validation loss stops decreasing.

However, the vanilla gradient descent algorithm takes in the entire data when calculating the gradients, which is oftentimes impractical given the large sizes of datasets found in practice.

Stochastic Gradient Descent (SGD)

Stochastic Gradient Descent (SGD) (7), also known as Mini-batch Gradient Descent, is a variation of the vanilla gradient descent invented by Kiefer and Wolfowitz in 1952. The SGD algorithm is well-adapted to handle large-scale datasets. Different from the vanilla gradient descent which calculates the gradients of input variables using the entire dataset, which is time consuming and computationally inhibitive, SGD loads in sizable batches of data (e.g., batches of 32, 64, or 128 input observations) to compute the gradients. The size of the mini-batches is a parameter of the model that is usually determined by memory constraints. Also, the sizes are powers of 2 in practice because many vectorized operation implementations work faster when their inputs are sized in powers of 2 (6). With this modification, the SGD algorithm can be formalized as follows:

Algorithm 3 Stochastic Gradient Descent Algorithm

INPUT:

data: the training data, *iter*: number of iterations
learning_rate: training rate, $loss\_fn$: the loss function
generator: mini-batch generator, *batch_size*: size of mini-batch

OUTPUT:

W: the optimized weights for the ANN model

function SGD($data, loss\_fn, iter, learning\_rate, batch\_size$)
  $W \leftarrow$ randomly initialize
  for i in 1:$iter$ **do**
    for j in 1:$size(data)//batch\_size$ **do**
      $mini\_batch_j \leftarrow$ generator($data, batch\_size$)
      $weights\_grad \leftarrow$ calculate_gradient($mini\_batch_j, loss\_fn, W$)
      $W \leftarrow W - learning\_rate \times weights\_grad$
    end for
  end for
  return W
end function

Note that the *generator* in the above algorithm outputs batches of data of size *batch_size*, and within each iteration, we still loop through the entire dataset (until the *generator* exhausts). SGD is preferred to the vanilla gradient descent due to its practicality.

2.4 Model Architecture

Another important factor of ANN models is the model architecture, which refers to the number of hidden layers, number of perceptrons in each layer, and activation function for each perceptron. A general rule of thumb is that more hidden layers with more neurons would increase the capability of the models to generalize to more complex problems. However, unnecessarily complex models suffer from long training time and overfitting. We will come back to the topic of model architecture in the next chapter.

Chapter 3

Convolutional Neural Networks (CNNs)

Convolutional Neural Networks are a set of complicated models that build on top of Artificial Neural Networks. In this chpater, I will first briefly go over the history of Convolutional Neural Networks and how they are applied in several computer vision tasks. Then, I will move on to discussing the backbone of this model by going over the two additional operations – convolution and max-pooling – that enable neural networks to perform computations on images. I will conclude this section with several well-known CNN architectures.

3.1 History of CNNs

Convolutional Neural Networks achieved their first success in digit recognition in the seminal work of Lecun et al. (11). In their paper, Lecun et al. surveyed a wide range of methods used for hand-written digit recognition and concluded that their model, LeNet, which is a fully-connected multi-layer neural networks, achieved the state-of-the-art performance of 0.7% - 1.7% test error rate (depending on the nuance of the model architecture and the optional application of boosting). After that, due to the limitation of both the computational power and the amount of labeled data, CNNs did not gain large scholarly attention until the success of AlexNet (9), which is a large and deep[1] CNN model trained to classify 1.2 million images from the

[1] the model is consisted of 8 layers and have 60 million parameters and 650,000 neurons

ImageNet dataset into 1000 classes. This work represents the first successful effort in using modern computation power, largely attributed to the usage of multiple Graphical Processing Units (GPUs), to train CNN models that yield unprecedented accuracy.

Subsequently, researchers have developed deeper and deeper CNN architectures that continuously break the record, culminating with the success of ResNet (4) by He et al., a CNN model that consisted of 152 layers and has surpassed human level perception. Figure 3.1 illustrates the relationship between the depth of CNN models and the accuracy they achieve:

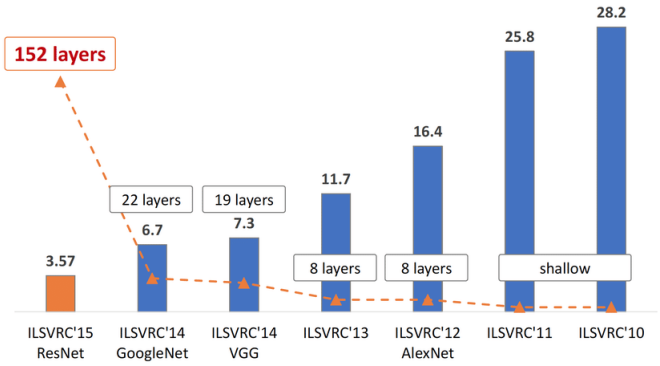


Figure 3.1: This figure shows the evolution of CNN models - over the years, the complexity and number of layers of models increased significantly while model performance kept imporving.

Beyond image classification, CNN models are widely applied in other tasks in computer vision, such as object localization, image captioning, medical image analysis, and synthetic image generation (6).

3.2 Representation of Images

Before diving into the details of CNN models, we will discuss how images are represented in memory.

In modern day computers, each image is represented by a collection of pixels. For monochromatic (black-and-white) images, each pixel has a numeric value ranging from 0 to 255 that represents the level of "whiteness" of that pixel, where 0 represents the complete black and 255 represents the complete white. For images with color, each pixel is a mixture of values ranging

from 0 to 255 of 3 color channels - red, green, and blue. Mathematically, we could treat monochromatic images as 2-D arrays of real numbers, and colored images as 3-D arrays of real numbers. This representation is essential to the following discussion on numerical operations within CNN models.

3.3 Convolutional Neural Networks

With the understanding of image representation, we are ready to discuss the operations within Convolutional Neural Networks (CNNs). In this section, we will introduce two types of numerical operations - convolution and max-pooling - that are unique to CNN models.

3.3.1 Convolution (CONV) Layer

A variation of multi-layer perceptrons, Convolutional Neural Networks (CNNs) are characterized by their usage of convolutional (CONV) layers which account for most of the computation and are the core building blocks of CNN models.

It is worth noting that the convolution operation in CNNs differs in meaning from the mathematical definition of convolution in signal processing. In CNN models, a typical CONV layer consists of a number of cubic (or square for images with only 1 color channel) filters, each of size $k \times k \times 3$. Here, k is a hyperparameter called "receptive field" that determines the size, or the field of vision, of each filter [2], while the last dimension, 3, matches the number of color channels of input images. Given an input image of size[3] $m \times m \times 3$, we slide, or *convolve*, each filter across the width and height of the input volume and compute dot products between the entries of the filter and the input at any position. Then, we would sum up the dot products to obtain a single value in the output map. Assuming a *stride*, or step size, of 1, we would get an output of size $(m - k + 1) \times (m - k + 1)$. Note that the third dimension of the output collapses because we sum up the $k \times k \times 3$ products across the three color channels. Additionally, we could use a stride of 2, which means

[2]Empirically, 3 and 5 are both suitable candidate values

[3]Note that the first two dimensions of the image are the same. This is because by convention, during the pre-processing process, researchers would resize the images into a square shape for the ease of computation. Besides, the last dimension, 3, refers to the three color channels, *RGB*, and matches the last dimension of filters within CONV layers.

the filter will slide for 2 pixels after each calculation. This will shrink the output size even further.

As illustrated in Figure 3.2, for a $6 \times 6 \times 3$ image and a $3 \times 3 \times 3$ filter, we start by putting this 3-dimensional filter at location $(1, 1, 1)$, or the upper left and front corner of the image. Then, we compute and sum up the products of $3 \times 3 \times 3 = 27$ parameters of the filter and the 27 color values of the image to form the first output value. Then, we slide the filter across the image horizontally and vertically to calculate values in the entire output of size 4×4. The resulting output is a 2-dimensional activation map that gives the responses of that filter at every spatial position of the input image.

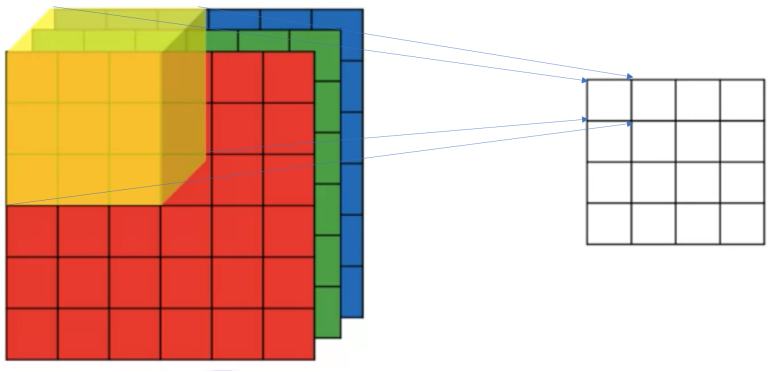


Figure 3.2: This figure show the convolution operation.

Before training begins, we would randomly initialize the weights within each filter in each CONV layer. After being trained on a number of input images with corresponding labels, CNN models should be able to adjust weights of each filter to perform a specific task. Such tasks range in difficulty from detecting a single color to detecting complex shapes and patterns such as a dog's nose. The matrix below represents a filter that is able to perform edge detection:

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

To illustrate its usage, we present three 2-D matrices that present monochromatic images:

$$\begin{bmatrix} 10 & 10 & 10 \\ 10 & 10 & 10 \\ 10 & 10 & 10 \end{bmatrix} \begin{bmatrix} 10 & 20 & 20 \\ 10 & 20 & 20 \\ 10 & 20 & 20 \end{bmatrix} \begin{bmatrix} 10 & 20 & 20 \\ 10 & 10 & 20 \\ 10 & 10 & 10 \end{bmatrix}$$

Note that the first matrix represents a homogeneous color patch; the second image represents a vertical edge; the third image represents a diagonal edge. For the three images, the activation values (sum of dot products) are respectively:

$$10 * (-1) * 8 + 8 * 10 = 0$$

$$10 * (-1) * 3 + (20) * (-1) * 5 + 8 * (20) = 30$$

$$10 * (-1) * 5 + (20) * (-1) * 3 + 8 * 10 = -30$$

Thus, this filter activates when there is an edge present in the image, but does not activate when the pixel values are homogeneous.

With proper training, filters in the earlier layers of a CNN model tend to perform simpler tasks such as edge detection, while filters in later layers tend to perform more complex tasks such as detecting more complex shapes and patterns. This is because the inputs to filters in later layers represent more high-level abstractions of the images due to the effect of the max-pooling layers, which we will discuss below.

3.3.2 Max-pooling layer

A max-pooling layer generally follows a CONV layer so as to form higher-level abstractions of the inputs images. It does so by shrinking, or down-sampling, squares within the input images to make the images smaller.

As an example, Figure 3.3 shows how a 2×2 images, represented as a 2-D array, is shrank to a single value by taking the maximum value across the 4 pixels:

$$\begin{bmatrix} 10 & 20 \\ 5 & 7 \end{bmatrix} \Rightarrow \begin{bmatrix} 20 \end{bmatrix}$$

Figure 3.3: A max-pooling operation shrinks a 2×2 image to a single value.

The max-pooling operation shown above is performed on a 2×2 image. We can adapt this operation to larger images by performing max-pooling

across the width and height of inputs images in a similar fashion as the convolution operation. The final output would be a down-sampled image with half the width and height as the input image.

3.4 Model Architecture

After talking about the unique characteristics of CNN models, it is also important to cover their model architecture.

Different from ANN models, we have two different layers in a CNN model, CONV and max-pooling layers. Thus, how we interleave these two types of layers can affect model performance. However, similar to that of ANN models, the number of layers within a CNN model plays a large role in model performance as well.

As a more concrete example, LeNet, designed by LeCun et al., (11) is one of the first successful attempts at using CNN models to perform digit recognition. Its architecture is shown in Figure 3.4.

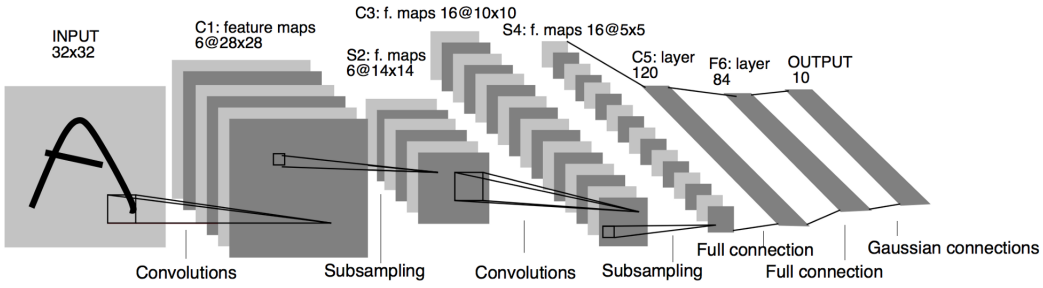


Figure 3.4: This figure shows the model architecture of LeNet by LeCun et al.

In this CNN architecture, we observe that the input image is fed to two blocks of CONV layer and max-pooling layer, followed by two fully-connected layers and the output layer. The fully-connected layers shown in the graph are exactly the same as the hidden layers within a standard ANN model.

Beyond LeNet, recent advancements in computational power have led to deeper and more complex CNN architectures. One of the most prominent CNN architectures is called ResNet (4), designed by He et al. This CNN architecture has four variations, each with 34, 50, 101, and 152 layers. Figure 3.5 shows its 34-layer variation.

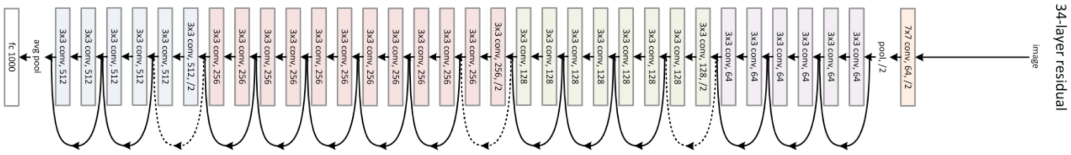


Figure 3.5: This figure shows the model architecture of the 34-layer ResNet.

There are two main observations from this architecture. First, there is no max-pooling layer in this network. This is because He et al. decided to use CONV layers with a stride of 2 to down-sample the inputs to replace the max-pooling operation. Also, notice that there are so-called "skip connections" that run in between block of CONV layers. These skip connections essentially pass the inputs to preivous CONV layer to a later CONV layer in conjunction with outputs of the immediately previous layer. The benefit of doing so is to combat the vanishing gradient problem, which refers to the phenomenon that when performing back-propagation on very deep CNN models, gradients of layers closer to the input will shrink to 0, and to enhance the gradient flow in ultra-deep networks for the model to converge faster. Models like ResNet have achieved astounding results in the field of computer vision, as demonstrated in Figure 3.1.

Chapter 4

Eye-tracking

As defined in the introduction, eye-tracking, also known as gaze estimation, is the process of predicting the screen coordinates at which a subject is fixating on at a certain time. In this chapter, I will discuss the process of eye-tracking, different models for this task, a number of evaluation metrics along with their advantages and disadvantages, and several distinguished works on the topic.

4.1 Types of Estimation Models

We start by describing the three major methods of eye-tracking, including 2D models, 3D models (model-based), and appearance-based models.

4.1.1 2D Models

2D models use a polynomial transformation function for mapping the vector between pupil center and corneal glint (shown in Figure 4.1) to the corresponding gaze coordinates on the screen (12). More specifically, researches use assistant devices such as cameras and light sources to mark pupil or reflection on cornea called glint (2).

After the vector is obtained, methods such as geometric transformation or Artificial Neural Networks are applied to obtain the mapping to gaze location on the screen.

(a) pupil-glint vector (b) pupil-corner vector

Figure 4.1: This figure shows the vector between pupil center and corneal glint.

4.1.2 3D Models

According to Lemley et al.(12), 3D model-based methods typically use a geometrical model of the human eye to estimate the center of the cornea, and the optical and visual axes of the eye. Gaze coordinates are estimated as points of intersection of the visual axes with the scene. In (18), Wu et al. present a 3-D model that consists of eyeballs, iris contours, and eyelids, to describe the geometrical properties and the movements of eyes. Despite their high accuracy, 3-D based models require elaborate system setups and knowledge about geometric relations between system components like LEDs, monitors, and cameras.

4.1.3 Appearance-based Models

Appearance-based models use cropped images of the subject's eyes or face to train machine learning models that predict the gaze location or gaze vector. This thesis mainly focuses on this set of models.

The essential work flow of appearance-based model goes as follows:

1. Gather input images from the user through a camera (typically a webcam).

2. Use facial landmark detection tools to detect and extract images of the users' eyes.

3. Train a model that maps eye images to gaze predictions.

Note that the gaze prediction could take various forms, which will be covered more in section 4.2 below. With the help of computer vision models such as Convolutional Neural Networks, there has been a huge growth in appearance-based models that have achieved noted success.

4.2 Types of Evaluation Methods

Due to the difference in the datasets, measurement of accuracy also differs in previous research. In this section, we will cover some commonly adopted measures of model performance, how they can be calculated, their pros and cons, and what implications they have on the models.

4.2.1 Actual Distance

The most direct and easy to understand measure of performance is the euclidean distance between the point of gaze prediction and the actual gaze location. This method is adopted by Krafka et al. in (8).

Despite its interpretability, this method suffers from a major drawback - that it is hard to compare cross-device performance. For example, sizes of iPhones have evolved over recent years, which means a $1cm$ difference in prediction error could imply different levels of performance for an iPhone X Max versus for an iPhone 6. The same problem also applies to monitors of desktops, which display a wide range of dimensions.

4.2.2 Angular Resolution

Another commonly used measure is the angular resolution in degrees of the predicted gaze vector and actual gaze vector. Such a method is adopted by (19), (20), (21), and (12).

To calculate the angular resolution, we need to normalize the images and head-pose space into a polar-coordinate angle space. More specifically, the normalization is done by rescaling and rotating the images so that the point-of-view, or the camera, looks at the mid-point between the edges of the subject's eyes from the fixed distance d, while the subject's head coordinate system is horizontally aligned. Then, eye images are cropped at a fixed resolution with a constant focal length. At the same time, ground-truth gaze positions are also converted using the exact same normalization procedure subject to each image to produce 2D gaze angle (yaw and pitch) vectors. An example a set of 2D gaze angle vector can be seen in Figure 4.2 (the x and y vectors shown in the graph).

After this pre-processing procedure, gaze prediction models are trained using normalized eye images and their corresponding labels. Then, the trained models can be used to predict gaze vectors of test images, usually

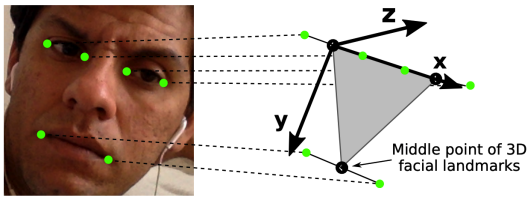


Figure 4.2: Definition of the head coordinate system defined based on the triangle connecting three midpoints of the eyes and mouth. The x-axis goes through the midpoints of both while the y-axis is perpendicular to the x-axis inside the triangle plane. The z-axis is perpendicular to this triangle plane.

of unseen subjects. The angular differences between the true gaze vectors and the predicted gaze vectors are calculated and averaged to form a model assessment.

The main benefit of this method is that the differences in angular resolution can be compared across subjects, across devices, and across datasets, since images, alongside the true gaze vectors, are normalized to the same dimension, using the same camera distance and focal length, in the normalized camera space. However, this method requires significant pre-processing steps, which means the gaze prediction models need to spend significant time in normalizing the input images, which increases time for prediction. Besides, although angular resolution can make model comparison device-independent, it is hard to directly interpret that 1 degree of angular distance, as it can translate into drastically different performance for subjects closer or farther away from the devices, and for small or large devices.

4.2.3 Block-based Classification Accuracy

Another way of measuring model performance is block-based classification accuracy. This method is tied to a completely different way of problem formulation, which is to divide up the screen into blocks and collect images while users fixate on these blocks. Essentially, this formulation transforms a regression problem, in which the predictions are the gaze coordinates or angular resolutions, to a classification problem, where the predictions are boxes that contain the gaze coordinates. This measure of accuracy was adopted by (17).

Similar to the euclidean distance, the method is also easy to interpret - a good model is one that predicts boxes that correctly contain the actual gaze locations. However, this measure also fails to consider the differences in device shapes and sizes and a model trained on one screen size cannot be easily adapted to another monitor.

4.2.4 Cross-subject and Cross-dataset Validation

Aside from specific measures of model performance, two other meta-techniques used in the literature are cross-subject and cross-dataset validation. We use the term "meta" to refer to these two methods in that they are not direct measures of accuracy, but rather techniques designed to ensure model genelizability.

More specifically, cross-subject validation is performed by repeatedly leaving out images of one or multiple subjects while training the model only on images of the other subjects, and then validating the trained model using images of the subjects left out. This method ensures that the model is not tested directly on subjects that appear in during training time, which is more reflective of the model's actual task, i.e. predicting gaze locations of unseen subjects.

Cross-dataset validation, as the name suggests, is the process of validating the model across datasets. We start by training the model using images exclusively from one dataset and then validate model performance on images from a different dataset. This technique ensures the robustness of trained models because datasets often differ significantly in terms of subject appearance and the environment. If a model is able to perform well on a dataset different from what it was exposed to, then this model will probably generalize well to other realistic scenarios.

4.3 Distinguished Work

In this section, we synthesize information in the previous sections by talking about four appearance based models. Specifically, we discuss their approaches, performance, and how they could be improved.

4.3.1 GazeNet

GazeNet (21), introduced by Zhang et al, is the first deep neural network using the VGG architecture that achieve 10.8 degrees cross-dataset validation. In their work, Zhang et al. performed several experiments, including adjusting the input image resolution, using only one eye to train the model, effects of illumination conditions and facial appearance variation. Their results show that image resolution and the use of both eyes affect gaze estimation performance, while head pose and pupil centre information are less informative.

4.3.2 iTraker

Krafka et al. presented iTraker in (8). This paper has two major contributions: GazeCapture, the first large scale data set for eye-tracking with 2.5 million images on 1450 subjects, and iTracker, a CNN and SVR-based model that is able to predict gaze location on tablets and cell-phones accurately. Their model is able to do predictions in real time (10–15 fps), and achieves a prediction error of 1.71cm and 2.53cm without calibration on mobile phones and tablets respectively. With calibration, the errors are reduced to 1.34cm and 2.12cm. It's interesting to note that the dataset, GazeCapture, is collected using crowd-sourcing, which is the first time that this method is explored in this field. Cross-dataset validation and the importance of the number of training samples are also explored. They demonstrate that the model is robust enough to generalize well to other datasets.

4.3.3 Block-based CNN

In (17), Wu et al. present a Block-based CNN model. This CNN model predicts the gaze location of subjects in terms of the block he or she is fixating on at a given time. Both a 6-block model and a 54-block model are trained and tested, achieving high classification accuracy. This same methodology is then applied to previous datasets such as the MPIIGaze data, but the performance is not as ideal, demonstrating the difficulty (mainly due to the lab environment of this work's dataset) of cross-dataset validation.

4.3.4 WebGazer

Different from other work, the WebGazer project focuses on building an accurate eye-tracker on commercial websites and thus democratize the technology of eye-tracking, which can often be expensive (13). This paper presents the model called WebGazer, a gaze predictor that employs user interactions (mouse clicks to be specific) to calibrate the model and outputs predictions in real-time. The model depends on *clmtrackr*, a JavaScript library for facial landmark detection, to first capture the face and pupil and then transforms the pupil pixels into a 120-D feature vector. Then, dimensionality reduction technique is used before the features are passed onto a ridge regression model to predict the x,y coordinates of the gaze location. The paper also presents an in-person study and online study to evaluate model performance. This model is implemented in JavaScript and can be used in real time.

Chapter 5

Methods

After the brief survey of the eye-tracking field, we will now dive into the specifics of model training. Specifically, we will explore The Eye Of The Typer Dataset (EOTT) dataset used for model training, the image preprocessing steps, and the set of model architectures employed in this thesis.

5.1 Dataset Description

The Eye Of The Typer Dataset *(EOTT) (13)* dataset is used by Papoutsaki et al. to build an extension of the WebGazer model mentioned in section 4.3.4. This dataset contains video footage and demographic information collected from 51 participants that participated in an eye tracking study. The data include user input data (such as mouse and cursor logs), screen recordings, webcam videos of the participants' faces, eye-gaze locations as predicted by a Tobii Pro X3-120 eye tracker, demographic information, and information about the lighting conditions. Participants completed pointing tasks including a Fitts' Law study, as well as reading, Web search, and typing tasks. The study was conducted on a desktop PC and Macbook Pro laptop based on the participant's preference. Below, we provide some additional dataset exploration.

5.1.1 Demographics

In machine learning tasks, it is critical to inspect diversity of datasets to ensure that there is no or little inherent imbalance in representation that

might introduce bias to the models to be contructed. Thus, we shall explore the composition of the EOTT dataset through several dimensions. To begin with, we will explore the demographics of the subjects of this study.

In terms of gender representation (Table 5.1), this dataset is well balanced with 23 male and 22 female participants. At the same time, number of images of each group is also well balanced. The distribution of *setting*, or the device that subjects used, is also balanced in terms of number of subjects and number of images.

	Male	**Female**	**PC**	**Laptop**
Num. Subjects	23	22	23	22
Percentage	51.1%	48.9%	51.1%	48.9%
Num. Images	226565	267322	273695	220192
Percentage	45.9%	54.1%	55.4%	44.6%

Table 5.1: Number of subjects and number of images for each gender and each device are both roughly balanced.

Figure 5.1 below shows the composition of the subjects' racial background and vision. We notice that the racial distribution is biased towards Asian and Caucasian and representation of people of color is somewhat lacking. However, the dataset is pretty well balanced in terms of vision situations.

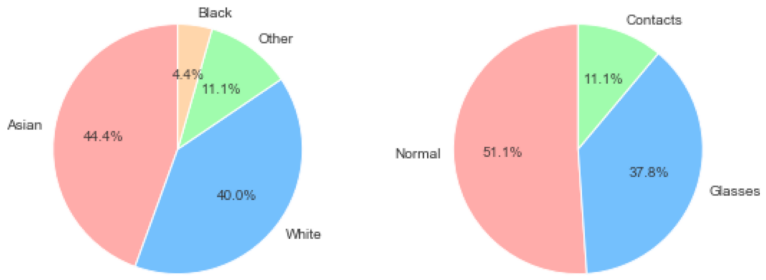


Figure 5.1: This figure shows the distribution of race and vision of the subjects.

On another account, as shown in Figure 5.2, participants' eye color is strongly biased towards dark brown to brown. Representation from other color groups are somewhat lacking.

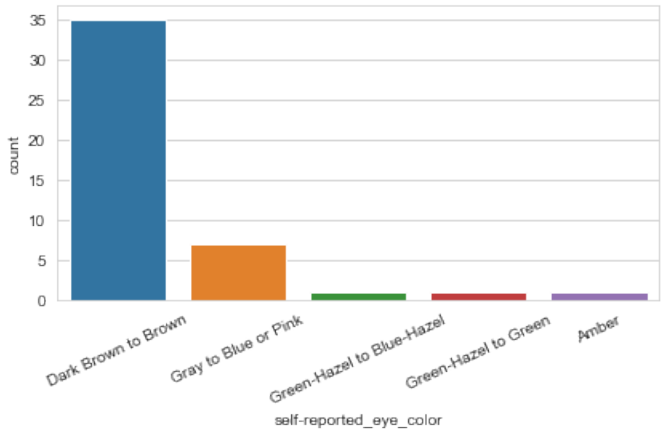


Figure 5.2: Distribution of eye color is strongly skewed towards dark brown to brown.

Another important aspect of the dataset is the number of images per participant – we need to carefully construct the training and validation dataset so that participants selected for each set are roughly representative of the data. We will cover more details of the training and validation set in Chapter 6. Figure 5.3 shows the number of images per participant, with participants colored by the device they are using.

5.2 Pre-processing

Following the general dataset information, we will now cover the details of the pre-processing steps used in this thesis, including image label creation, facial landmark detection, and image pre-processing.

5.2.1 Label Creation

The most crucial step of any supervised machine learning task is to establish the "underground truth". In this thesis, we use a script written by Papoutsaki et al. to dissect the videos into a total of 702027 frame images and match the time frame of each image with the closest prediction in time given by the Tobii Pro X3-120 eye tracker as the "true label". This way, we have established a one-to-one pairing of images and labels.

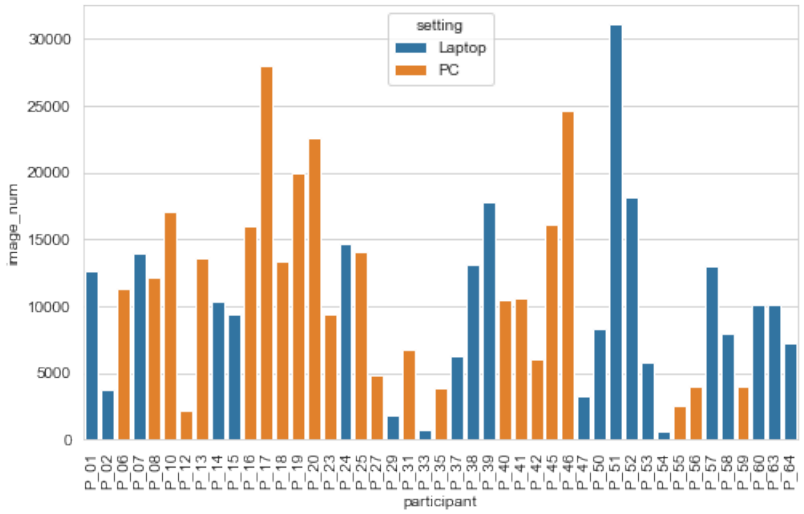


Figure 5.3: A count plot showing the number of images per participant, with participants colored by the device they used during the study.

More specifically, for each image, the Tobii eye tracker produces a set of four coordinates: *LeftX*, *LeftY*, *RightX*, *RightY*, representing the predicted X (along the width of screen) and Y (along the height of screen) gaze locations of the left and right eye respectively. Upon examination, we found that 31.84% of the images have missing predictions for the right eye and 32.07% of the images have missing predictions for the left eye. However, only 29.65% of images have missing predictions for both the left and the right eye. Based on this information, we made two assumptions. First, that gaze predictions for the two eyes are trying to capture the same on-screen location, and second, if the gaze prediction is only available for one eye, that prediction is accurate. With these two assumptions, we condense the sets of four predictions for each frame image into one coordinate, in x and y directions, using the following rule:

$$\text{coord\_x} = \begin{cases} \frac{LeftX + RightX}{2} & \text{if both are present} \\ LeftX \text{ or } RightX & \text{if only one is present} \\ \text{Undefined} & \text{otherwise} \end{cases}$$

$$\text{coord\_y} = \begin{cases} \frac{LeftY+RightY}{2} & \text{if both are present} \\ LeftY \text{ or } RightY & \text{if only one is present} \\ \text{Undefined} & \text{otherwise} \end{cases}$$

Additionally, the gaze predictions from Tobii follow what is called the "Active Display Coordinate System (ADCS)" (Tobii.com), a 2D coordinate system that aligns with the active display area of the device used. When using an eye tracker with a monitor, the active display area is the display area excluding the monitor frame. The origin of the ADCS is the upper left corner of the active display area. In other words, the point (0, 0) denotes the upper left corner of the active display area while (1, 1) denotes the lower right corner of it.

There are two points worth mentioning. First, under such a setup, the eye tracker does not differentiate between PC versus Mac, and thus would predict a value of (0, 0) to the upper left corner of the active display area regardless of the screen size or position. This fact has profound implications for model training in that if we do not separate the frame image coming from PC versus Mac, the model might be confused by the mis-labelling; second, it is important to note that gaze locations are not bounded in between 0 and 1 since the Tobii eye-tracker is able to capture it when subjects look outside the screen. In such cases, we decided to cap the gaze locations to be strictly in between 0 and 1 so that we could still map gaze predictions onto the screen in a reasonable way. In Figure 5.4, we present the scatter plot of normalized gaze coordinates for all the frame images colored by the device used. We observe that the distribution of gaze locations is indeed different for those from the PC subset and those from the laptop subset. At the same time, gaze locations are not strictly bounded between 0 and 1.

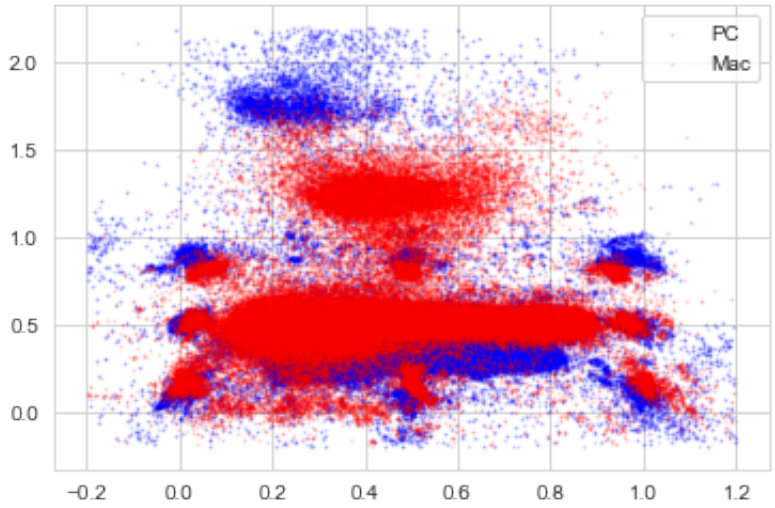


Figure 5.4: Distribution of normalized gaze locations shows a significant difference between images from the PC subset and those from the laptop subset.

Further, Figure 5.5 below plots the marginal distribution of normalized gaze locations, i.e. the x and y coordinates of gaze location. These plots reveal that the distribution of gaze coordinates is different for images from the two device groups. The differences is more apparent in the y direction than in the x direction.

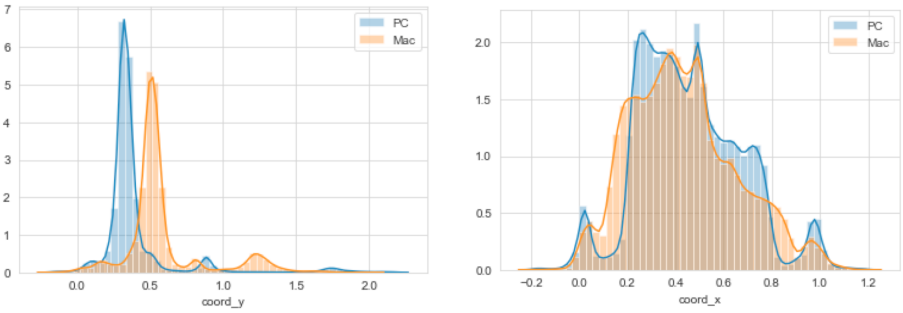


Figure 5.5: Distribution of x and y coordinates of gaze locations are significantly different for images from the two settings. Difference in the y direction is particularly conspicuous.

5.2.2 Facial Landmark Detection

As demonstrated in literature (14), images of the eye encode the most prevalent information regarding a person's gaze location. Consequently, we extract what is known as "facial landmarks", or a strategic set of coordinates on a person's face, using an open-source software called OpenFace (1). The purpose of this thesis does not concern the mechanisms of facial landmark detection. Figure 5.6 below shows the set of coordinates.

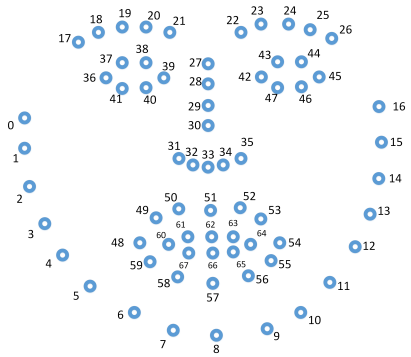


Figure 5.6: This figure show the set of facial landmarks detected by OpenFace.

Note that, for our purposes, we are only interested in extracting the coordinates 36 to **47**, which depicts the locations of a person's left and right eye.

Using command line tools in conjunction with OpenFace, we extracted the complete set of facial landmarks of all the frame images within the EOTT dataset. Note that, we also instructed OpenFace to output a confidence score which describes how confident the tracker was in each landmark detection estimate. We exclude the images with a confidence lower than 0.9, a cutoff value decided by exploratory analysis of images with different confidence scores. Figure 5.7 shows a sample framed imaged marked with detected facial landmarks.

Out of the 702027 frame images, there are a total of 642719 images with a confidence score larger than 0.9. And out of the 642719 images, we identified 493887 images with valid labels, using the procedure outlined in the previous sub-section.

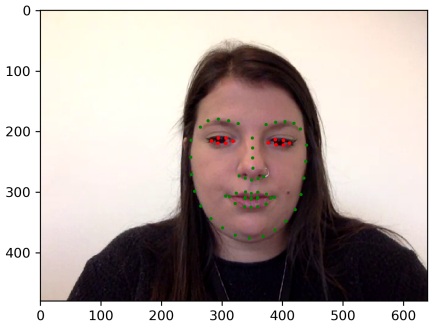


Figure 5.7: A sample frame image marked with detected facial landmark. The landmarks around the eye are colored in red, while the rest are colored in green.

5.2.3 Image Pre-processing

Now, we have 493887 images marked with facial landmarks and labeled with x and y screen coordinates. Next, we prepare the images for model training using two pre-processing steps – bounding box extraction and histogram equalization.

For each eye, we extract a bounding box whose top boundary is defined by the maximum y value of the top four coordinates, whose bottom boundary is defined by the minimum y value of the bottom four coordinates, whose left boundary is defined by the x coordinate of the left-most coordinate, and whose right boundary is defined by the x coordinate of the right-most coordinate. Take the left eye shown in Figure 5.8 as an example. The top of the bounding box is equal to $max\{y_{36}, y_{37}, y_{38}, y_{39}\}$; its bottom is equal to $min\{y_{36}, y_{41}, y_{40}, y_{39}\}$; its left is equal to x_{36}; and its right is equal to x_{39}.

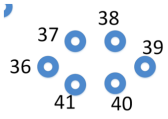


Figure 5.8: The figure shows the six landmark coordinates surrounding the left eye as detected by OpenFace.

On top of this, closer examination of the resulting images reveals that

sometimes the eyes are not entirely captured within the boxes defined above. Thus, we have decided to enlarge each eye box by a factor of 1.2 both horizontally and vertically so that the eyes are completely included. For each frame image, we stitch together the bounding boxes of the left and right eye to form a single image so as to ease the computational process.

After the bounding box extraction, we performed another image transformation called histogram equalization. The mathematics of this transformation are beyond the scope of this thesis, but in essence, histogram equalization is a process that stretches the black and white pixels of an image so that the cumulative distribution function of the image intensity becomes approximately linear. The end effect of this transformation is that the contrast of each image is maximized to an equal amount. Figure 5.9 below demonstrates the process of histogram equalization. The top left image is the original images while the top right depicts the probability density function (pdf) and cumulative distribution function (cdf) of its intensity. The bottom left image shows the histogram equalized image while to the right is its corresponding cdf and cdf. As we can see, the transformed image displays greater contrast while the distribution of its intensity is more spread out towards the extremes. At the same time, its cdf is now roughly linear. With the same transformation, images will not only become clearer to view, they will also become more "standardized" as their cdfs are roughly the same.

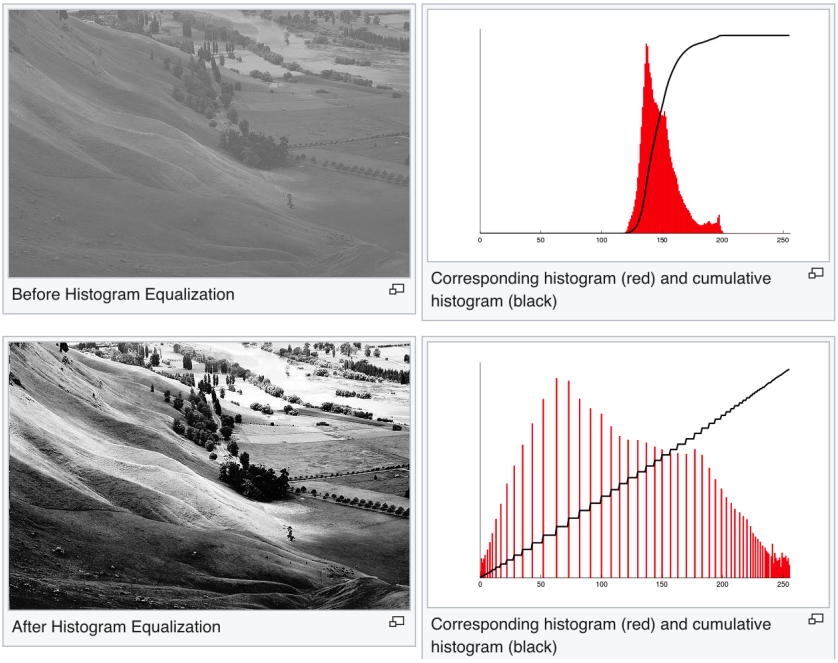


Figure 5.9: This figure shows the original image with the corresponding *pdf* and *cdf* of its intensity, and the histogram equalized image with the corresponding *pdf* and *cdf* of its intensity.

For each stitched image, we perform bounding box extraction and histogram equalization, followed by a resizing step which transforms the image dimensions to 20×100. This choice of dimension considers a balance between the space that the entire dataset takes and the resolution (which directly affects the amount of information encoded). Figure 5.10 below shows several examples of the final image before they are passed for training.

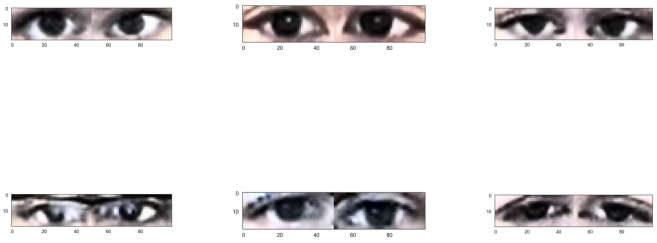


Figure 5.10: This figure shows some sample images that are passed onto the CNN model.

5.3 Selected Model Architectures

After the pre-processing steps, we are ready to build our CNN-based eye-trackers. In this thesis, we use CNN models to map from stitched eye images to predicted x and y coordinates on the active display area of a device. We adopt two model architectures - MobileNet (5), an efficient and fast CNN model introduced by Howard et al, and ResNet50 (4), a deep and accuracy CNN model developed by He et al. Due to the aforementioned difference in gaze distributions of subjects from PC and Mac groups, we build separate models for the two groups, resulting in a total of 4 different models. In the next chapter, we will discuss the performance of each model and compare it against that achieved by WebGazer.

Chapter 6

Experimental Results

In this section, I will cover the set-up of the datasets I used for my model training and validation and the experimental results obtained using my models. Specifically, I will cover how the training set and validation set are created so as to ensure a candid representation of unseen data. Then, I will discuss test results for a number of models and compare their performances with those of WebGazer. As we will demonstrate, the best CNN model offers a 25% reduction in terms of test average euclidean distance in the PC group and a 10% reduction in the Mac group.

6.1 Validation Set Setup

As we have mentioned in section 5.1.1, due to discrepancies in the recording length of each task of each participant, the numbers of images vary a lot for each participant. In addition, we have also mentioned in section 5.2.1 that the distribution of gaze locations differ for participants using PC to complete the study and those that use Mac laptop. Due to these two reasons, we decide to split up the entire dataset into two training sets, one for PC and one for Mac, and two validation sets, one for PC and one for Mac. We strategically picked 4 participants from the PC group and 4 from the Mac group as the respective validation sets, and ensured that the number of images within each validation set is about 20% of the total number of images for each group. This way, we have ensured that the validation set is large enough to be representative.

An aspect of this setup is that instead of randomly dividing all the images belonging to each group into 80% and 20%, we use the cross-subject valida-

tion technique mentioned in Chapter 4 which requires that the training set and validation set include different subjects. According to past work (14), models tend to perform better if they are exposed to images of test subjects, whereas they tend to perform worse on unseen subjects in test time. However, the second approach, i.e., cross-subject validation, is a more realistic representation as the models are to be applied to new subjects.

Finally, in this thesis, we do not employ the traditional three-dataset split approach which divides the dataset into training, validation, and testing sets. This is because in our experiments, we are not tuning hyperparameters of any model; rather, we are just using the validation set to assess our model performance. Thus, our validation set is the de facto test set in the traditional machine learning sense.

6.2 Model Performance

In our experimentation, we trained a number of CNN models with two architectures – MobileNet(5) and ResNet50(4). MobileNet is chosen as it is light-weight which enables fast prediction and deployment on mobile devices, while ResNet50 is chosen due to its demonstrated superior performance on the ImageNet dataset(4). For all training and testing we used the Keras(3) implementation of the two models.

The loss function we use to assess each model is the mean absolution error (MAE). MAE is a metric used to assess the average absolute distance between actual values and predictions. Assuming we have a vector of n true labels, y, and a vector of n predictions, $\hat{y}$, MAE can be calculated by:

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

In our particular case, since each on-screen coordinate has two coordinates, x and y, we have to output two predictions for every image. Thus, we will also have two MAE measures for each model - MAE_x and MAE_y. To represent model performance using a single metric, we also calculate the MAE across both x and y direction, MAE_{avg}, and the average euclidean distance between the actual and predicted coordinates, $EuDist$, calculated by

$$EuDist = \frac{1}{n}\sum_{i=1}^{n}\sqrt{(x_i - \hat{x_i})^2 + (y_i - \hat{y}_i)^2}$$

In table 6.1, below, we present the performances of four CNN models.

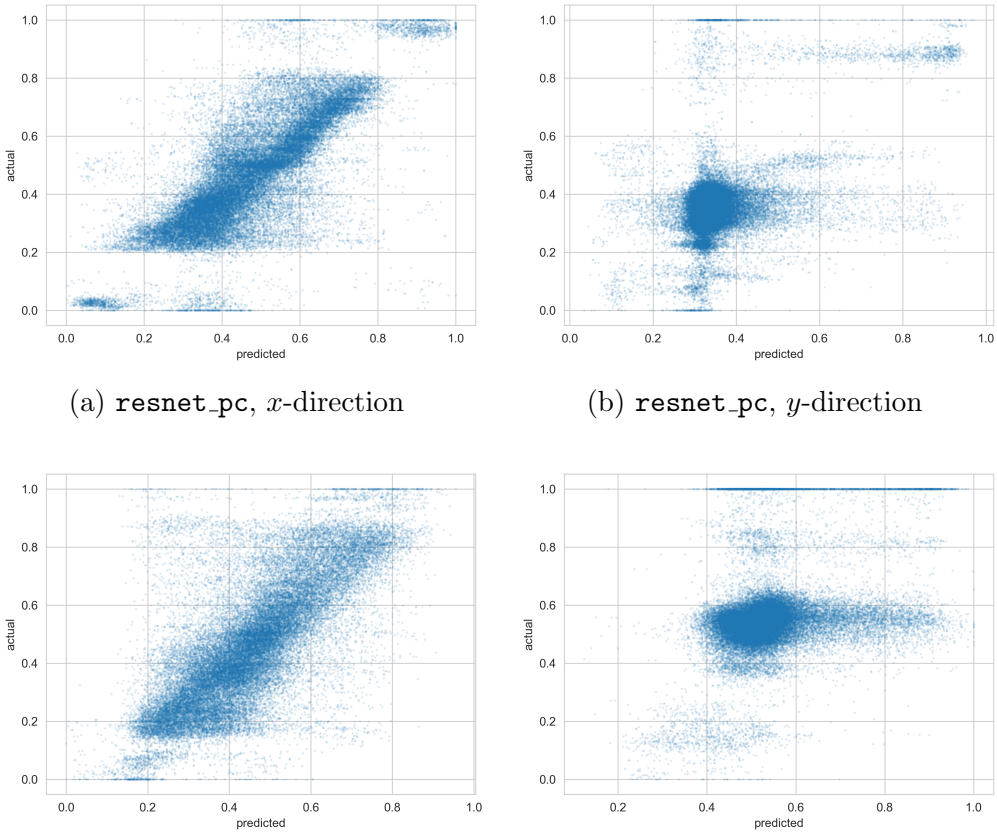
	mobile_pc	resnet_pc	WebGazer_pc	mobile_mac	resnet_mac	WebGazer_mac
MAE_x	0.0972	**0.0833**	0.1161	0.1171	**0.1045**	0.1130
MAE_y	0.0887	**0.0735**	0.0930	**0.0937**	0.1038	0.1197
MAE_{avg}	0.0930	**0.0784**	0.1045	0.1054	**0.1041**	0.1163
$EuDist$	0.1488	**0.1255**	0.1668	0.1722	**0.1685**	0.1866

Table 6.1: Model Performances

We can observe from the table that for the PC group, both MobileNet and ResNet yield better results compared to those of WebGazer. For the Mac group, MobileNet yields results comparable with those of WebGazer (MobileNet's predictions are better in the y-direction but worse in the x-direction) but ResNet's performance strictly dominates that of WebGazer in both directions. This finding illustrates the effectiveness of using CNN models for the eye-tracking task. In addition, we observe that performance metrics of ResNet, a deeper and more complex model, are better than those of MobileNet, a shallower and more light-weight model, in all but one case (y-direction of the Mac group). This finding suggests that deeper and more complex models tend to yield superior results.

Besides measuring MAE and $EuDist$, we also plotted the actual against predicted x and y coordinates for each of the models in order to better understand the model performance. Here, we display the plots corresponding to the two best models, `resnet_pc` and `resnet_mac`, in Figure 6.1.

It is useful to compare the scatter plot with the 45-degree line which represents $actual = predicted$ so as to assess how far away the predictions are from the actual values. We observe that, for both models, the actual and predicted x coordinates generally align with the 45-degree line, while the actual and predicted y values deviate from the 45-degree line. This suggests that the model is better at predicting the x-coordinates of gaze locations (left and right) than predicting the y-coordinates of gaze locations (up and down). This suggests that images of the eye are more generalizable and predictive of the left-and-right direction than the up-and-down direction. Additionally, from this set of figures, we can examine whether the difference between actual and predicted values vary as value of actual coordinates changes. One observation is that, for both models, smaller x-coordinates (closer to 0) tend to be overestimated while larger x-coordinates (closer to 1) tend to be underestimated. Similarly, the predictions for the y-coordinates are largely



(a) `resnet_pc`, x-direction

(b) `resnet_pc`, y-direction

(c) `resnet_mac`, x-direction

(d) `resnet_mac`, y-direction

Figure 6.1: True against predicted value plots

concentrated around the respective average y value for both groups. In other words, the predictions display the phenomenon known as "regression toward the mean". To counter this systematic bias, we could consider adding in additional information other than images of the eye, such as head position and subject's distance to the screen. These topics are left for future exploration.

6.3 Comparison with WebGazer

As mentioned in section 4.3.4, WebGazer (13) is an eye-tracking tool that uses ridge regression as the underlying algorithm. As shown in table 6.1,

we can calculate that, for the PC group, the CNN model decreases the Euclidean distance between the actual and predicted coordinates by $(0.1668 - 0.1255)/0.1668 = 24.76\%$; for the Mac group, the CNN model decreases the Euclidean distance between the actual and predicted coordinates by $(0.1866 - 0.1685)/0.1866 = 9.69\%$. While the CNN models for the two group both improved prediction quality, we notice a discrepancy in terms of the amount of improvement - the PC group experienced a larger drop in the loss metric. Though we do not know the exact reason for such a difference, a hypothesis is that the validation set for Mac group possesses some inherent bias that renders itself to be significantly different from the corresponding training group. If this hypothesis is true, we should be able to solve this problem by increasing the sizes of both the training and validation sets for this group, making both sets more representative of the general population.

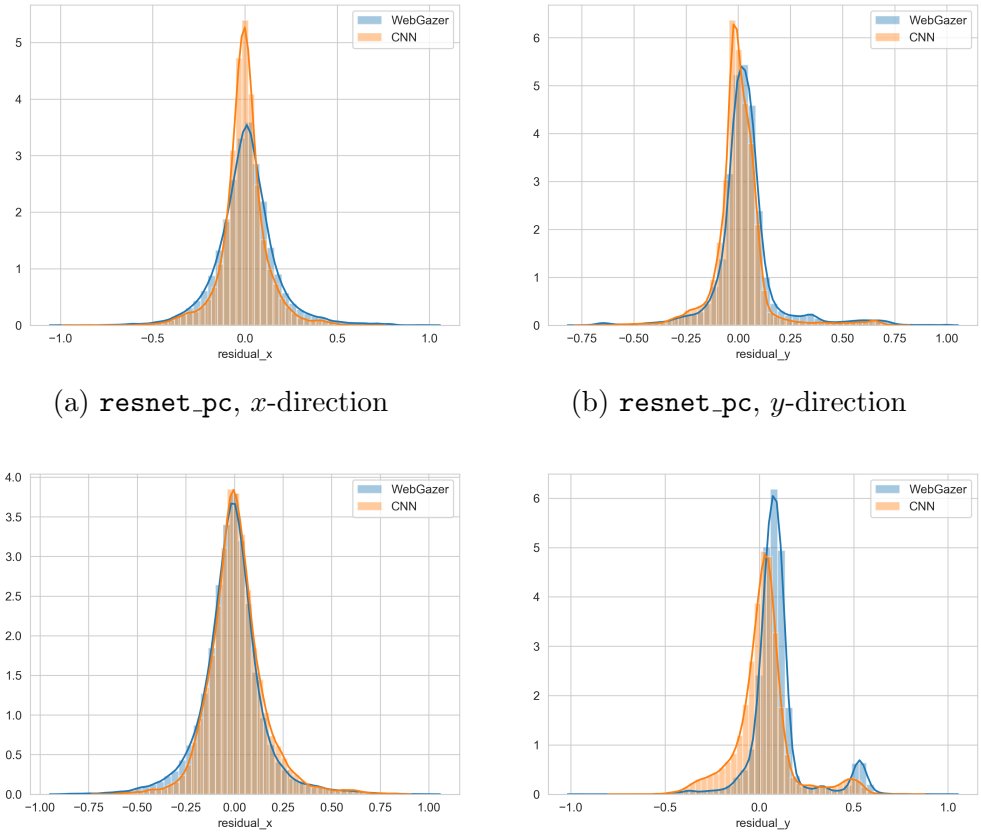
To illustrate the difference in performance, we also plot the distribution of residuals in both x and y directions for the CNN models and WebGazer within both PC group and Mac group, as shown in Figure 6.2.

We observe that, for the PC group, the distributions of residuals for the CNN models strictly dominate those of WebGazer in both x and y directions; the difference in distribution of residuals for the Mac group is not as stark. This reaffirms our earlier finding that the improvements in performance for the PC group is significant while that of the Mac group is not as noticeable. For the Mac group, however, we do notice an observable improvement in the y-direction, especially the disappearance of the concentration of residuals at around $residual\_y = 0.5$ for the WebGazer's pdf, which correspond to the out-of-screen predictions produced by WebGazer.

6.4 Side-effect - Blink Detection

To showcase the effectiveness of the new CNN models, we write a Python script to map the gaze predictions back onto the computer screen. We used `resnet_pc` model due to its superior performance and apply it to participant 10's recording of the "the benefit of social networking" task. It is worth noting that participant 10 is within the validation set of subjects, which means the model was not exposed the images of this subject during training time.

In the video, we use the color red to denote the underground truth, or the gaze coordinates predicted by the Tobii eye-tracker. We use green to



(a) `resnet_pc`, x-direction

(b) `resnet_pc`, y-direction

(c) `resnet_mac`, x-direction

(d) `resnet_mac`, y-direction

Figure 6.2: Distribution of residuals - CNN models versus Webgazer

denote the CNN predictions and blue to denote the WebGazer predictions. In addition, we also use transparency to denote the goodness of a prediction - the closer (better) a prediction is to its actual gaze coordinates, the less transparent it will be. In other words, predictions that are too far off the actual truth will be almost completely transparent. The video output can be found here: `https://www.youtube.com/watch?v=hi36JKSD9-8`.

We observe that, in this video, the CNN predictions closely follows the actual coordinates and are pretty stable, while the WebGazer predictions are further away and display more variation. One interesting observation is that, there are certain times (in the video, see 17s, 18s, 26s, and etc.) when

the CNN predictions fluctuates wildly up-and-down. To investigate into the cause of such fluctuation, we plotted the actual and predicted $y-$coordinates against the frame number (which we can treat as time), as shown in Figure 6.3 below:

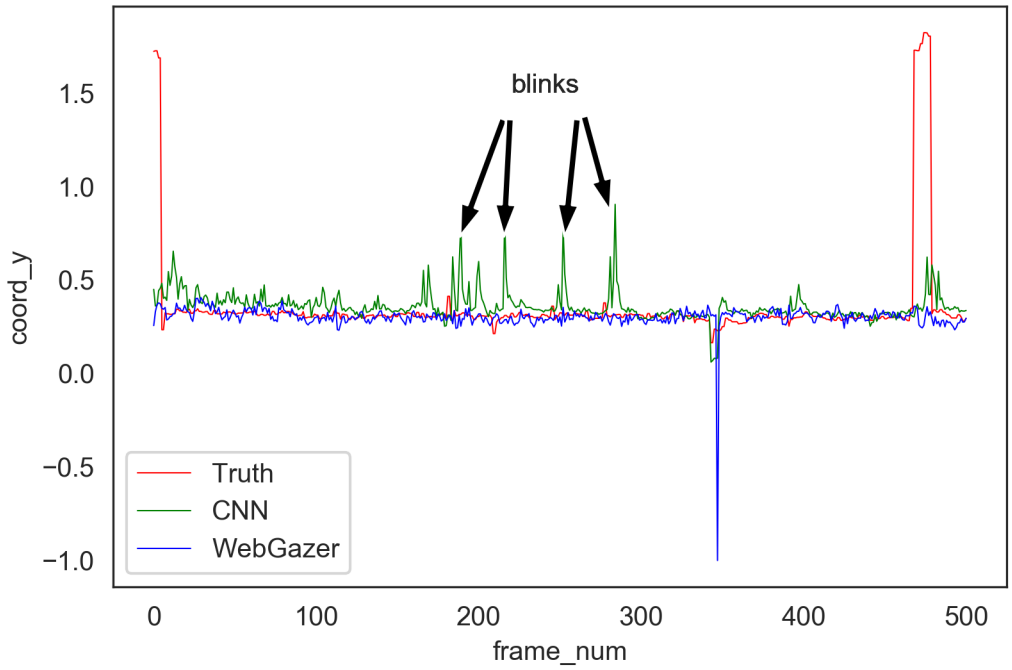


Figure 6.3: This figure shows actual and prediction $y-$coordinates against the frame number. We observe several fluctuations in the CNN predictions, which, through close examinations of the video footage, cooresponds to subject's blinks.

Observe the four arrows pointing at several peaks of CNN predictions in the y direction. To find out the cause of such behavior, we outputted a slower version of the same video corresponding to the related frame numbers (200-300), available through this link: `https://www.youtube.com/watch?v=89aMJDFggz4`. Close examination of the video reveals that these peaks correspond to times when the subject is blinking. This finding has two important implications. First, observe that the y-coordinates predicted by the Tobii eye

tracker at those time did not fluctuate as much as the CNN predictions. This means that if we do not include these blink moments in model evaluation, our model performance will be even better than currently recorded. Second, out model has the potential to serve as a "blink-detector", an interesting application in addition to its accurate gaze predictions.

Chapter 7

Conclusion

In this thesis, we built the mathematical framework of Artificial Neural Networks (ANN) and its variations in the field of computer vision, Convolutional Neural Networks (CNN). We covered the mechanisms of back propagation as well as the Stochastic Gradient Algorithms used to efficiently train CNN models. Then, we provided a holistic overview of the field of eye-tracking, covering different types of eye-tracking models, various evaluation metrics, and several distinguished appearance-based eye-tracking tools devised by past work. Finally, we built four CNN-based eye tracking models using the dataset obtained from the WebGazer(13) project, and demonstrated that our best models offer a 25% and 10% performance improvements measure by average euclidean distance for the PC and Mac groups, respectively. In addition, we showed through video demonstrations that our CNN-based models have the potential to serve as blink detectors.

However, the current model has a lot room for further improvements. One potential aspect is to improve the amount of data in terms of number of subjects and diversity of background. The current dataset only includes merely 45 participants, which means it is far from being representative of all subjects that could use such technology. The dataset is especially lacking in terms of racial diversity, as there are only 2 black subjects in this study. Also, the studies were done exclusively in a controlled lab environment, which means the frame images are not representative of the diverse background color and lighting in the actual daily life. Thus, it is reasonable to believe that additional and more diverse data could contribute to the generalizability of the dataset and thus make the models more robust in application. Furthermore, we could consider using other available inputs

to enhance model performance. The current implementation only uses the cropped eye images as input, while ignoring other available information such as the relative position of subjects' heads within the frame images, and other detected facial landmarks by OpenFace. These additional information has the potential to mitigate the ineffectiveness of the current models trained using eye images alone.

Acknowledgement

First, I would like to thank Professor Alexandra Papoutsaki at the Pomona Computer Science Department for her generous support throughout my thesis journey. She provided me with access to the Pomona HCI lab and its high-performance machines without which my thesis would not have been possible. At the same time, her expertise in the field of eye-tracking and her weekly meetings with me contributed significantly to this project's success. Under the pressure of teaching 3 classes and advising 20+ students, Professor Papoutsaki showed me love and emotional support which I appreciate whole-heartedly. I wish her best of luck in the upcoming review.

Additionally, I would like to thank Dr. Hardin, Professor Chandler, Professor Sarkis, and Dr. Radunskaya at the Pomona Math Department for their teaching. I am glad that I have received one of the best math educations within the nation that instilled in me a sense of mathematical maturity that I am sure will benefit my whole life. I will never forget the amount of rigor and FUN (credit to Dr. Rad) of each class and the wonderful time I spent inside Millikan, a place I would call home.

Further, I would like to thank my parents for their financial and emotional support through my four years at Pomona College. Their upbringing and education shaped me into the humble and diligent student I am today and are essential to my success at Pomona. Recently, I was deeply touched by a Chinese soap opera, *All is Well*, and realized that my level of caring for my parents is far from their love for me. I will shoulder my responsibilities as a child to his parents.

Another important person in my college life is my girlfriend, Jie Wang. My classmate since kindergarten, Jie is my most important friend of life. An upright, kind-hearted, and sometime sentimental girl, Jie played a crucial role in my four years of college. I will not forget the hundreds of travels between Pomona and UCLA and the Fridays when we got to meet that I call "hope". Together, we have been through the ups and downs in both of our life, kept each other company, and made each other's college years more meaningful and memorable. I hope in the upcoming years, when I start my master's degree, my job, and eventually a family, we will continue to support and look out for one another and live a happy life.

Finally, I would like to thank all my friends in college, "Haojiyou" DMQ, "Shiyou for 3 years" Edward, "Laoban & Professor" Lio, Nina, Alex, Olivia, and Sitong. They filled my college life with happiness and laughter. I wish

them all the best in their post-graduate endeavours, and I hope we will remain life-long friends.

Bibliography

[1] Amos, B., Ludwiczuk, B., and Satyanarayanan, M. (2016). Openface: A general-purpose face recognition library with mobile applications. Technical report, CMU-CS-16-118, CMU School of Computer Science.

[2] Blignaut, P. (2014). Mapping the pupil-glint vector to gaze coordinates in a simple video-based eye tracker. *Journal of Eye Movement Research*, 7.

[3] Chollet, F. et al. (2015). Keras. `https://keras.io`.

[4] He, K., Zhang, X., Ren, S., and Sun, J. (2015). Deep residual learning for image recognition. *CoRR*, abs/1512.03385.

[5] Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H. (2017). Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861.

[6] Joshson, J. (2018). Cs231n: Convolutional neural networks for visual recognition. `http://cs231n.github.io/`. Accessed: 2010-09-30.

[7] Kiefer, J. and Wolfowitz, J. (1952). Stochastic estimation of the maximum of a regression function. *Ann. Math. Statist.*, 23(3):462–466.

[8] Krafka, K., Khosla, A., Kellnhofer, P., Kannan, H., Bhandarkar, S., Matusik, W., and Torralba, A. (2016). Eye tracking for everyone. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2176–2184.

[9] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Proceedings of the*

25th International Conference on Neural Information Processing Systems - Volume 1, NIPS'12, pages 1097–1105, USA. Curran Associates Inc.

[10] Lecun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.

[11] Lecun, Y., Larry, J., L, B., A, B., Corinna, C., John, D., Harris, D., Isabelle, G., Urs, M., E, S., Patrice, S., and Vapnik (1999). Comparison of learning algorithms for handwritten digit recognition. *International Conference on Artificial Neural Networks*.

[12] Lemley, J., Kar, A., Drimbarean, A., and Corcoran, P. (2018). Efficient cnn implementation for eye-gaze estimation on low-power/low-quality consumer imaging systems. *CoRR*, abs/1806.10890.

[13] Papoutsaki, A., Sangkloy, P., Laskey, J., Daskalova, N., Huang, J., and Hays, J. (2016). Webgazer: Scalable webcam eye tracking using user interactions. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 3839–3845. AAAI.

[14] Punde, P. A., Jadhav, M. E., and Manza, R. R. (2017). A study of eye tracking technology and its applications. In *2017 1st International Conference on Intelligent Systems and Information Management (ICISIM)*, pages 86–90.

[15] Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, pages 65–386.

[Tobii.com] Tobii.com. Tobii pro sdk documentation.

[17] Wu, X., Li, J., Wu, Q., and Sun, J. (2017). Appearance-based gaze block estimation via cnn classification. In *2017 IEEE 19th International Workshop on Multimedia Signal Processing (MMSP)*, pages 1–5.

[18] Wu, Y.-L., Yeh, C.-T., Hung, W.-C., and Tang, C.-Y. (2014). Gaze direction estimation using support vector machine with active appearance model. *Multimedia Tools and Applications*, 70(3):2037–2062.

[19] Zhang, X., Sugano, Y., Fritz, M., and Bulling, A. (2015). Appearance-based gaze estimation in the wild. In *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4511–4520.

[20] Zhang, X., Sugano, Y., Fritz, M., and Bulling, A. (2016). It's written all over your face: Full-face appearance-based gaze estimation. *CoRR*, abs/1611.08860.

[21] Zhang, X., Sugano, Y., Fritz, M., and Bulling, A. (2017). Mpiigaze: Real-world dataset and deep appearance-based gaze estimation. *CoRR*, abs/1711.09017.