Assignment-3: Saliency Map Prediction and Analysis for ASD and TD Fixation Maps

Samia Nusrat
Nafiz Anwar Sadi
Sojeeb Roy Partho

2517318050
2517372650
2517297650

samia.nusrat.251@northsouth.edu
nafiz.sadi.251@northsouth.edu
sojeeb.partho.251@northsouth.edu

March 28, 2025

1 Introduction

The study of visual saliency is fundamental in both cognitive science and medical research, offering valuable insights into human attention mechanisms. Saliency prediction models estimate regions within an image that are most likely to attract human gaze. While these models have demonstrated effectiveness in predicting attention patterns in Typically Developed (TD) individuals, their ability to model the visual behavior of individuals with Autism Spectrum Disorder (ASD) remains an open question. ASD is a neurodevelopmental condition characterized by atypical attention allocation, often resulting in distinct gaze patterns that significantly differ from those of TD individuals. This variation poses challenges for conventional saliency models, which are typically trained on datasets reflecting TD attention behaviors. This study investigates the capability of three deep-learning-based saliency prediction models—DeepGaze IIe, SalFBNet, and the Saliency Attentive Model—to approximate the gaze patterns of both TD and ASD individuals. Using images from the Saliency4ASD dataset, we generate saliency maps and compare them against ground-truth fixation maps for both groups. By evaluating the predicted saliency maps relative to real fixation data, we assess the models' effectiveness in capturing ASD-specific visual attention and contrast their performance with TD fixation behavior. To achieve this, we employ multiple quantitative evaluation metrics, including Normalized Scanpath Saliency (NSS), Area Under the Curve (AUC), Correlation Coefficient (CC), and Kullback-Leibler Divergence (KL-Div). These metrics provide a comprehensive assessment of how closely the predicted saliency maps align with actual fixation patterns.

2 Method

2.1 Dataset

The dataset used in this study is sourced from the "Saliency4ASD" dataset, which was introduced by Duan et al. (2019). The dataset is designed to facilitate saliency prediction tasks, particularly in the context of comparing attention focus between individuals with Autism Spectrum Disorder (ASD) and typically developing (TD) individuals. The Saliency4ASD dataset consists of five distinct folders, each containing different types of data. For the purposes of this project, we used only three of these folders:

- **Saliency4asd/Images**: This folder contains the original images of patients, which are central to the study. These images represent visual stimuli used to observe and predict where subjects with

ASD and TD individuals direct their visual attention. The goal is to generate saliency maps for each image, which will highlight the areas of the image that attract attention.

- **Saliency4asd/ASDFixMaps**: This folder contains fixation maps specifically for individuals with Autism Spectrum Disorder (ASD). The fixation maps represent the eye-tracking data of ASD subjects when viewing the corresponding images from the "Images" folder. A Gaussian blur filter is applied to these fixation maps, which detects the regions of frequent eye movements and attention patterns. These fixation maps will be used for comparison with the saliency maps generated by the model. The goal is to analyze how well the model-generated saliency maps align with the actual attention patterns of ASD subjects.

- **Saliency4asd/TDFixMaps**: Similar to the ASD fixation maps, this folder contains fixation maps for typically developing (TD) individuals. These maps are generated by applying a Gaussian blur filter on the eye-tracking data, highlighting regions where TD individuals focused their attention while viewing the corresponding images. Just like the ASD fixation maps, these fixation maps will be compared with the saliency maps generated by the model to assess the model's ability to predict human visual attention in TD individuals.

Each fixation map represents the eye-tracking data of individuals, providing valuable insight into the regions of an image where the subjects focused their attention. By comparing the fixation maps for both ASD and TD groups with the generated saliency maps, we aim to evaluate the model's effectiveness in predicting attention patterns across different groups. These comparisons will provide a better understanding of how attention differs between individuals with ASD and those with typical development, and whether the saliency prediction model can capture these differences.

The comparison between the generated saliency maps and the fixation maps is crucial for evaluating the model's accuracy and its potential applications in understanding visual attention in clinical settings, particularly for ASD individuals.

2.2 DeepGaze IIE

The DeepGaze2 model is a neural network architecture specifically designed for predicting human gaze or fixation patterns on images. This model has been developed to understand how humans visually attend to different regions within an image. It is based on a hierarchical feature extraction process, starting with a pretrained CNN backbone, followed by task-specific layers that generate saliency predictions. In addition, the model incorporates methods to evaluate inter- and intra-model complementarity. Core Components are:

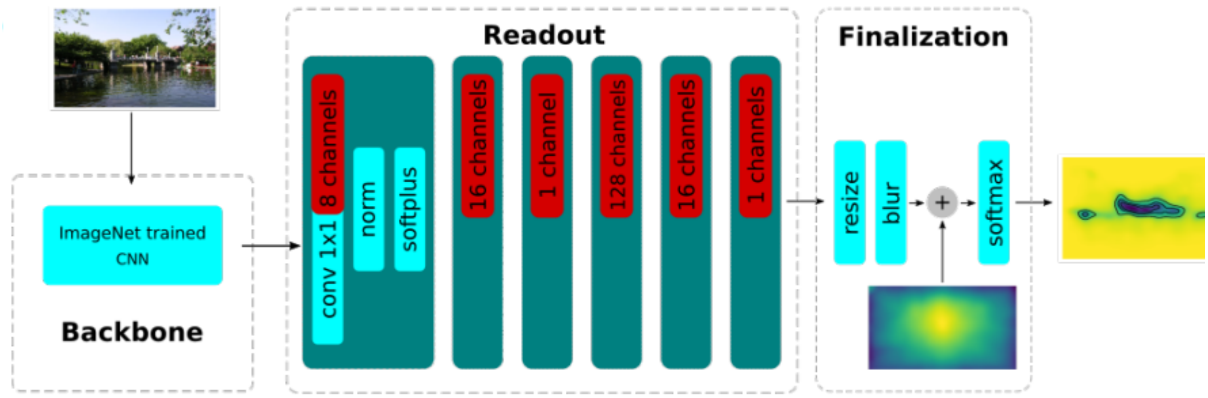

Figure 1: DeepGaze Operation

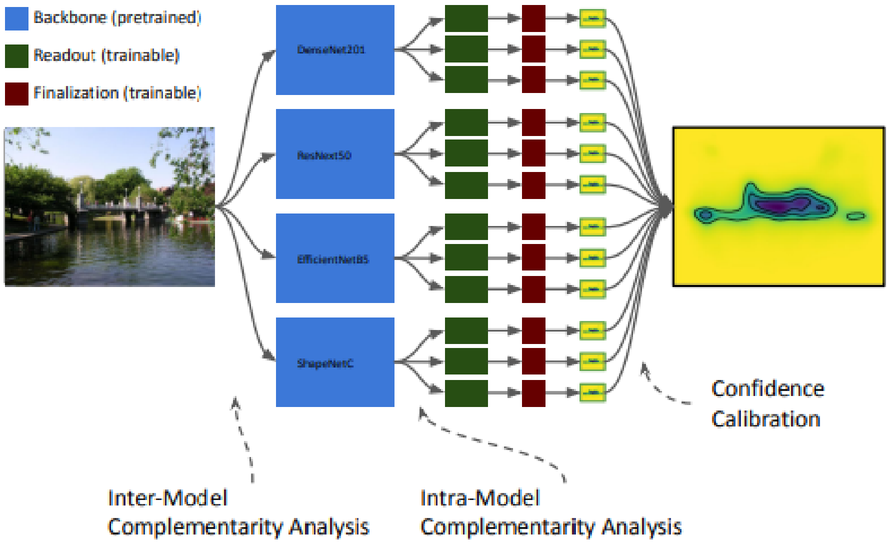

Figure 2: DeepGaze Operation

2.2.1 Backbone (Pretrained)

The backbone of DeepGaze2 typically uses a pretrained convolutional neural network (CNN) such as VGG-19. This component is responsible for extracting general visual features from the input image.

- The pretrained CNN captures general features like edges, textures, and shapes that are common in a wide variety of images.
- The output of this component is a set of feature maps, denoted as F.

The mathematical representation of the backbone is given by:

$$F = \text{CNN}(I),$$

where I is the input image, and F are the extracted feature maps from the CNN.

2.2.2 Readout (Trainable)

The readout component is responsible for learning task-specific mappings from the extracted features to saliency predictions. This part of the model is trainable and typically implemented as a series of 1x1 convolutional layers. The output is a saliency prediction, denoted S_r.

$$S_r = \sigma(W_r * F + b_r),$$

where:

- σ is an activation function (e.g., ReLU or sigmoid),
- W_r and b_r are learnable weights and biases,
- F is the feature map from the CNN.

The readout layer learns how to map the general features from the backbone to specific saliency values for the image.

2.2.3 Finalization (Trainable)

The finalization component refines the saliency predictions produced by the readout layer. This part of the model often includes additional convolutional layers and normalization steps to further improve the saliency map.

The final saliency prediction is represented as:

$$S_f = \text{Finalize}(S_r),$$

where S_f is the final saliency map that provides the predicted fixation regions of the image.

Inter-Model Complementarity Inter-model complementarity measures how different models complement each other in predicting saliency. It assesses whether multiple models capture different aspects of saliency in an image. The complementarity can be quantified using the correlation coefficient between the saliency maps predicted by different models.

$$C_{\text{inter}} = 1 - \rho(S_1, S_2),$$

where:

- ρ represents the correlation between the saliency maps S_1 and S_2,
- High complementarity indicates that the models capture different aspects of saliency.

Intra-Model Complementarity Intra-model complementarity analyzes how different layers within the same model complement each other in generating saliency predictions. This can be measured using the correlation between features at different layers.

$$C_{\text{intra}} = \frac{1}{N} \sum_{i \neq j} \left(1 - \rho(F_i, F_j)\right),$$

where:

- F_i and F_j represent features extracted from different layers in the model,
- N is the total number of layers,
- High intra-model complementarity suggests that the different layers provide unique, useful features for saliency prediction.

2.2.4 Center-Surround Mechanisms

The DeepGaze2 model is inspired by the biological vision systems that rely on center-surround mechanisms. In these systems, neurons respond strongly to stimuli in the center of their receptive field but are inhibited by stimuli in the surrounding area. This concept helps the model focus on important regions in an image that attract attention.

2.2.5 Feature Integration Theory

Feature Integration Theory (FIT) posits that visual attention is guided by the integration of low-level features such as color, intensity, and orientation. DeepGaze2 leverages this theory by combining these features into a final saliency map, where regions that attract attention are highlighted.

2.2.6 Deep Learning Principles

The DeepGaze2 model uses deep learning principles, particularly hierarchical feature learning. Early layers in the network detect simple, low-level features such as edges and textures, while later layers capture more complex and abstract patterns. This enables the model to learn sophisticated representations of the input image, which are crucial for saliency prediction.

2.2.7 Saliency Prediction

The saliency prediction can be represented mathematically as follows:

$$S(x, y) = \sum_{c} w_c \cdot F_c(x, y) + b,$$

where:

- $S(x, y)$ is the saliency value at the location (x, y),
- $F_c(x, y)$ are feature maps from different channels,
- w_c are learned weights,
- b is a bias term.

2.2.8 Loss Function and Training

The model is typically trained using a loss function that minimizes the difference between the true and predicted fixation distributions. One common loss function is the Kullback-Leibler (KL) divergence:

$$L = \mathrm{KL}(P_{\text{true}} || P_{\text{pred}}) + \lambda \cdot R(\theta),$$

where:

- P_{true} and P_{pred} are the true and predicted fixation distributions,
- λ is a regularization parameter,
- $R(\theta)$ is a regularization term that helps prevent overfitting.

2.3 SalFBNet

SalFBNet is a feedback-enhanced deep learning model designed for saliency prediction, addressing the limitations of traditional feedforward convolutional neural networks (CNNs). Unlike conventional models that process visual information in a purely feedforward manner, SalFBNet incorporates feedback mechanisms that iteratively refine saliency predictions. This feedback process allows the model to capture high-level contextual information and improve accuracy in predicting human visual attention.

Traditional CNN-based saliency models often struggle to incorporate long-range dependencies and contextual relationships due to their hierarchical nature. The introduction of feedback in SalFBNet helps bridge this gap by allowing high-level feature representations to influence lower-level feature maps, leading to more refined and context-aware saliency maps. This makes SalFBNet more effective in real-world scenarios where complex visual patterns and varying attention regions exist. SalFBNet is composed of three primary components:

1. **Encoder**: A hierarchical CNN-based feature extractor responsible for learning multi-scale spatial features from the input image.
2. **Feedback Module**: A recurrent mechanism that propagates high-level feature information back to earlier layers, enabling context-aware refinement.
3. **Decoder**: A multi-scale feature integration network that generates the final saliency map based on the refined representations.

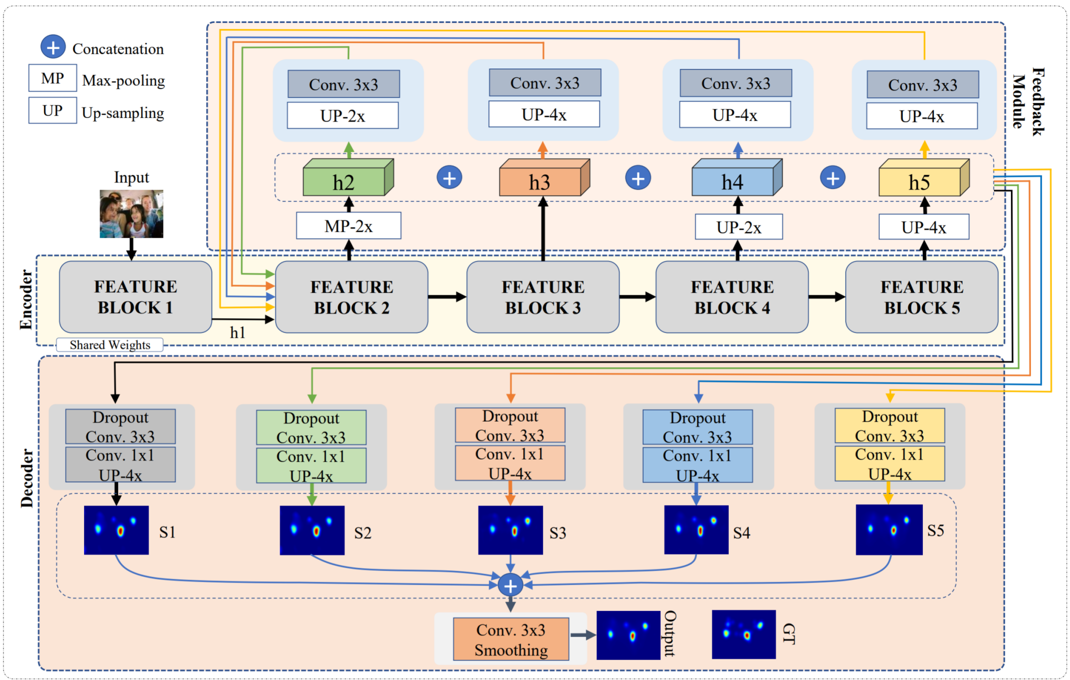

Figure 3: SalFBNet Operation

2.3.1 Encoder: Feature Extraction

The encoder extracts hierarchical features from the input image $I \in R^{H \times W \times 3}$, where H and W represent the image height and width, and the three channels correspond to RGB color components.

The feature extraction process consists of multiple convolutional layers, each capturing different levels of abstraction. The transformation in each layer is defined as:

$$H_i = f(W_i * H_{i-1} + b_i), \quad i \in \{1, 2, ..., 5\}$$

where:

- H_i represents the output of the i-th feature block,
- W_i and b_i denote the learnable convolutional weights and biases,
- $f(\cdot)$ is the activation function, typically ReLU (Rectified Linear Unit).

Each layer extracts increasingly complex features, with lower layers capturing fine-grained spatial details and higher layers capturing semantic information such as object shapes and boundaries.

2.3.2 Feedback Module: Contextual Refinement

The feedback module introduces a mechanism that integrates high-level contextual information back into lower-level layers, enabling better refinement of saliency predictions. Unlike standard CNNs that rely solely on feedforward computations, this feedback pathway allows the model to iteratively refine feature representations based on the broader scene context.

Mathematically, the refinement process is formulated as:

$$F_i = g(U_i(H_{i+1})) + H_i, \quad i \in \{2, 3, 4, 5\}$$

where:

- F_i is the refined feature representation at level i,
- $U_i(\cdot)$ represents an upsampling operation that brings high-level feature maps to a compatible resolution,
- $g(\cdot)$ is a convolutional layer followed by an activation function to process the upsampled information.

This feedback connection allows higher-level semantic information (e.g., object-level context) to enhance lower-level feature maps, improving the spatial coherence of saliency predictions.

2.3.3 Decoder: Saliency Map Generation

After the refinement process, the decoder combines multi-scale features to generate the final saliency map. Each refined feature F_i contributes to the saliency prediction as follows:

$$S_i = \sigma(W_d * F_i + b_d)$$

where:

- S_i represents an intermediate saliency map prediction,
- W_d and b_d are learnable parameters,
- $\sigma(\cdot)$ is the sigmoid activation function, ensuring that the output values range between 0 and 1.

To obtain the final saliency map, all intermediate predictions are combined using a weighted summation:

$$S_{final} = \sum_{i=1}^{5} \alpha_i S_i$$

where α_i are learnable weights that determine the contribution of each scale-level saliency prediction to the final output.

2.4 Saliency Attentive Model

This document describes a deep learning-based saliency prediction model that integrates a dilated convolutional network, an attentive ConvLSTM, and learned priors to generate saliency maps. The architecture consists of three major components:

- **Dilated Convolutional Network**: Extracts spatial features from the input image.
- **Attentive ConvLSTM**: Captures sequential dependencies and refines feature representations.
- **Learned Priors**: Guides saliency prediction using Gaussian-based priors.
- **Saliency Map Generation**: Produces the final saliency map, optimized via a loss function.

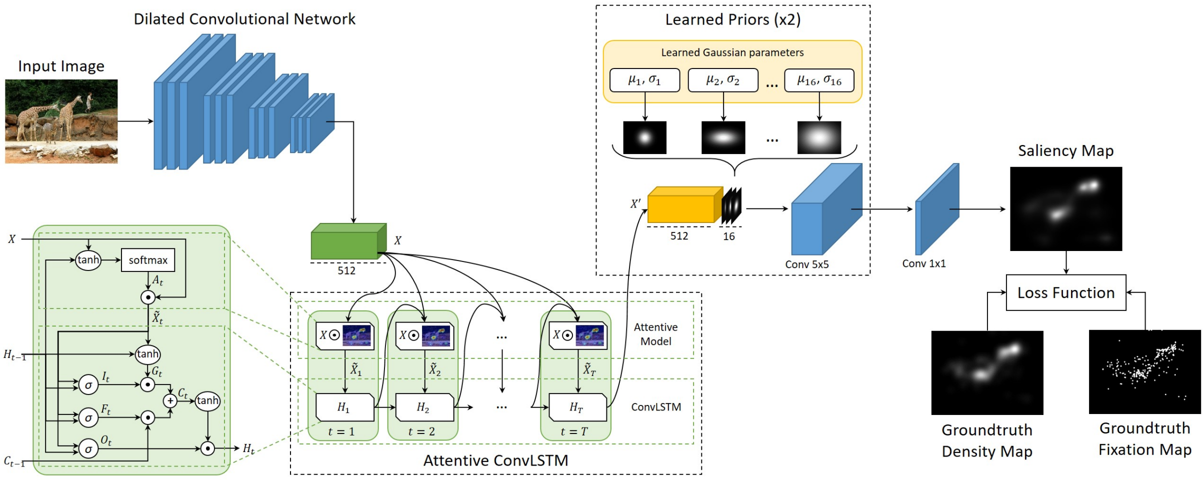

Figure 4: Saliency Attentive Model

2.4.1 Dilated Convolutional Network

Dilated convolutions capture multi-scale contextual information without increasing computation. The feature map X is computed as:

$$X = f(W_d * I + b) \tag{1}$$

where:

- I is the input image,
- W_d represents the dilated convolutional weights,
- $*$ denotes convolution,
- b is the bias,
- $f(\cdot)$ is the activation function (e.g., ReLU).

2.4.2 Attentive ConvLSTM

ConvLSTM extends traditional LSTMs by incorporating convolutional operations. For each time step t:

$$I_t = \sigma(W_i * X + U_i * H_{t-1} + b_i) \tag{2}$$
$$F_t = \sigma(W_f * X + U_f * H_{t-1} + b_f) \tag{3}$$
$$O_t = \sigma(W_o * X + U_o * H_{t-1} + b_o) \tag{4}$$
$$g_t = \tanh(W_g * X + U_g * H_{t-1} + b_g) \tag{5}$$
$$C_t = F_t \odot C_{t-1} + I_t \odot g_t \tag{6}$$
$$H_t = O_t \odot \tanh(C_t) \tag{7}$$

where:

- I_t, F_t, O_t are the input, forget, and output gates,
- g_t represents the cell update,
- W and U are learned weight matrices,
- σ is the sigmoid activation,
- tanh is the hyperbolic tangent function,
- C_t is the cell state,
- H_t is the hidden state.

The attention mechanism refines X using:

$$A_t = \text{softmax}(\tanh(W_a * X + b_a)) \tag{8}$$

$$\tilde{X}_t = A_t \odot X \tag{9}$$

2.4.3 Learned Priors

The model incorporates Gaussian priors to guide saliency prediction. Each prior is parameterized as:

$$P_i(x, y) = \frac{1}{2\pi\sigma_i^2} \exp\left(-\frac{(x - \mu_{i_x})^2 + (y - \mu_{i_y})^2}{2\sigma_i^2}\right) \tag{10}$$

where:

- (μ_{i_x}, μ_{i_y}) are the mean locations,
- σ_i is the standard deviation.

These priors are convolved with the learned feature representations for refinement.

2.4.4 Saliency Map Generation and Loss Function

The final saliency map S is computed as:

$$S = \sigma(W_s * X' + b_s) \tag{11}$$

where W_s are learned weights and X' is the refined feature representation.

The loss function is computed as:

$$\mathcal{L} = \lambda_1 \text{KL}(S, D) + \lambda_2 \text{BCE}(S, F) \tag{12}$$

where:

- $\text{KL}(S, D)$ is the Kullback-Leibler divergence between S and the ground truth density map D,
- $\text{BCE}(S, F)$ is the Binary Cross-Entropy loss between S and the ground truth fixation map F,
- λ_1 and λ_2 are weighting factors.

T his model integrates dilated convolutions, ConvLSTM for attention, and learned Gaussian priors for saliency prediction. The combined KL and BCE loss ensures alignment with human visual attention patterns.

2.5 Evaluation Metrics

The predicted saliency maps were compared with TD (Typical Development) and ASD (Autism Spectrum Disorder) fixation maps using various evaluation metrics. The following common saliency evaluation metrics were applied to assess the performance of the predicted saliency maps.

2.5.1 AUC (Area Under Curve)

AUC measures how well the predicted saliency map aligns with fixation points. The AUC score determines the ability of each model to rank actual fixation points higher than non-fixation points. The formula for AUC is:

$$\mathrm{AUC} = \int_0^1 \mathrm{TPR}(\mathrm{FPR})\, d(\mathrm{FPR}),$$

where:

- **TPR** (True Positive Rate) is the proportion of true fixation points correctly classified as fixations.
- **FPR** (False Positive Rate) is the proportion of non-fixation points incorrectly classified as fixations.

Some possible cases for AUC are:

- If AUC = 1.0, the saliency prediction is perfect.
- If AUC = 0.5, the model is performing randomly.
- If AUC ¡ 0.5, the model is worse than random, indicating incorrect behavior.

2.5.2 CC (Correlation Coefficient)

The Correlation Coefficient (CC) evaluates the linear correlation between predicted and ground-truth fixation maps. It quantifies how similar the maps are by comparing them pixel by pixel. A higher CC indicates a good correlation between the predicted saliency map and the fixation map. The formula is:

$$\mathrm{CC}(S, F) = \frac{\sum_i (S_i - \mu_S)(F_i - \mu_F)}{\sqrt{\sum_i (S_i - \mu_S)^2 \sum_i (F_i - \mu_F)^2}},$$

where:

- S_i and F_i are the pixel values at location i in the predicted saliency map S and the ground-truth fixation map F.
- μ_S and μ_F are the mean values of the predicted and ground-truth maps, respectively.

Some possible cases for CC:

- If CC = 1.0, there is a perfect positive correlation.
- If CC = 0.0, there is no correlation.
- If CC = -1.0, there is a perfect negative correlation.

2.5.3 MSE (Mean Squared Error)

MSE computes the pixel-wise error between the predicted saliency and fixation maps. It is a helpful indicator for evaluating how well a model predicts fixation densities. The formula for MSE is:

$$\mathrm{MSE}(S, F) = \frac{1}{N} \sum_{i=1}^{N} (S_i - F_i)^2,$$

where:

- N is the total number of pixels in the image.
- S_i is the predicted saliency value at pixel i.
- F_i is the ground-truth fixation value at pixel i.

Some possible cases for MSE:

- If MSE = 0, there is a perfect saliency prediction.
- If MSE is low, the predicted saliency map closely approximates the fixation map.
- If MSE is high, a large discrepancy exists between the predicted and actual fixation maps, indicating poor model performance.

2.5.4 EMD (Earth Mover's Distance)

EMD quantifies the dissimilarity between two distributions. Given two distributions, S (predicted saliency map) and F (ground-truth fixation map), represented as normalized probability distributions over an image grid, EMD is computed as:

$$\mathrm{EMD}(S, F) = \min_{f_{ij}} \sum_i \sum_j f_{ij} d_{ij},$$

subject to the constraints:

$$\sum_j f_{ij} \leq S_i, \quad \sum_i f_{ij} \leq F_j, \quad \sum_i \sum_j f_{ij} = \min\left(\sum_i S_i, \sum_j F_j\right),$$

where:

- f_{ij} is the flow of "mass" from pixel i in S to pixel j in F.
- d_{ij} is the ground distance between pixel i in S to pixel j in F, using the Euclidean distance formula.

Some possible cases for EMD:

- If EMD $= 0$, there is a perfect match between the saliency map and the fixation map.
- If EMD is low, the predicted saliency map closely resembles the ground truth, with minimal spatial displacement.
- If EMD is high, significant spatial mismatch between the predicted and actual saliency maps is present.

2.5.5 KLDiv (Kullback-Leibler Divergence)

KL divergence measures the divergence between predicted and fixation map distributions. A larger KLDiv value reveals notable differences, while a lower KLDiv value suggests that the predicted saliency map is closer to the ground truth fixation map. The formula is:

$$\mathrm{KL}(S \parallel F) = \sum_i F_i \log \frac{F_i}{S_i},$$

where:

- S_i is the predicted saliency value at pixel i, normalized to sum to 1.
- F_i is the ground-truth fixation value at pixel i, also normalized to sum to 1.

Some possible cases for KLDiv:

- If KLDiv $= 0$, the predicted saliency distribution perfectly matches the ground-truth fixation distribution.
- If KLDiv is low, the predicted saliency map closely approximates the ground-truth fixation map.
- If KLDiv is high, significant divergence occurs, indicating poor prediction.

2.5.6 NSS (Normalized Scanpath Saliency)

NSS measures how well high-saliency regions correspond to actual fixations. It evaluates how well the saliency map corresponds to human gaze patterns. The formula is:

$$\mathrm{NSS} = \frac{1}{N} \sum_{i=1}^{N} \frac{S(x_i, y_i) - \mu}{\sigma},$$

where:

- $S(x_i, y_i)$ is the saliency value at the fixation location (x_i, y_i),

- μ is the mean of the saliency map,
- σ is the standard deviation of the saliency map,
- N is the total number of fixations.

Some possible cases for NSS:

- If NSS is low, the performance is poor.
- If NSS is high, the performance is good.

2.5.7 InfoGain

Information Gain (InfoGain) is used to measure the effectiveness of a feature in predicting the saliency map. It quantifies the reduction in uncertainty about the fixation location when the saliency map is known. The formula for InfoGain is:

$$\text{InfoGain}(S, F) = H(F) - H(F|S),$$

where:

- $H(F)$ is the entropy of the ground-truth fixation map F,
- $H(F|S)$ is the conditional entropy of F given the predicted saliency map S.

The entropy $H(X)$ of a probability distribution X is given by:

$$H(X) = -\sum_i P(x_i) \log P(x_i),$$

where $P(x_i)$ is the probability of the event x_i occurring.

Some possible cases for InfoGain:

- If InfoGain is high, the saliency map provides significant information about the fixation map, indicating a strong predictive power.
- If InfoGain is low, the saliency map provides little information about the fixation map, indicating poor predictive power.

Each of these metrics provides different insights into the accuracy and quality of the saliency prediction. The combination of multiple metrics ensures a more comprehensive evaluation of the predicted saliency maps' performance, allowing for a thorough assessment of how well the model is capturing human gaze patterns.

3 Experiment

3.1 Dataset Description

The dataset used in this study is from Duan et al. (2019), containing eye movement fixation data from children with Autism Spectrum Disorder (ASD) and Typically Developed (TD) children.

3.1.1 Image Set

The dataset consists of a set of images stored in the `TrainingData/Images/` folder, which were used as stimuli during eye-tracking experiments.

3.1.2 Fixation Maps

The dataset provides fixation maps indicating where subjects focused their visual attention. These fixation maps serve as ground truth for saliency prediction models:

- `TrainingData/TD_FixMaps/`: Fixation maps for TD participants.
- `TrainingData/ASD_FixMaps/`: Fixation maps for ASD participants.

3.2 Selection of Saliency Prediction Models

The study utilizes three state-of-the-art saliency prediction models:

- **DeepGaze IIE**: A deep learning-based saliency model using a neural network trained on human gaze data.
- **SalFBNet**: A CNN-based model optimized for efficient saliency detection.
- **Saliency Attentive Model (SAM)**: Integrates attention mechanisms to enhance saliency prediction.

The models were selected from publicly available repositories and used without modification.

3.3 Saliency Map Prediction Process

For each image in the dataset, the following steps were performed:

1. **Input Preparation**: Each image was fed into the saliency models.
2. **Model Processing**: The models generated predicted saliency maps representing likely human attention areas.
3. **Output Generation**: The predicted maps were stored for further evaluation.

3.4 Evaluation Metrics and Performance Analysis

The predicted saliency maps were compared with ground truth fixation maps (TD and ASD) using nine evaluation metrics:

- AUC_Borji
- AUC_Judd
- AUC_Shuffled
- CC (Correlation Coefficient)
- MSE (Mean Squared Error)
- EMD (Earth Mover's Distance)
- KLdiv (Kullback-Leibler Divergence)
- NSS (Normalized Scanpath Saliency)
- Info Gain (Information Gain)

3.5 Results and Observations

The mean performance values for each model were recorded for TD vs. Saliency Models and ASD vs. Saliency Models.

3.5.1 TD vs. Saliency Models

Metric	SalFBNet	SAM	DeepGaze IIE
AUC_Borji	0.2329	0.6671	0.2702
AUC_Judd	0.3835	0.5410	0.2702
AUC_Shuffled	-0.2665	0.1706	0.5001
CC	-0.4499	0.2568	-0.2563
MSE	0.3827	0.2038	0.3706
EMD	0.1567	0.6469	0.0834
KLdiv	0.0964	4.5218	0.0394
NSS	-0.2092	0.1194	0.5470
Info Gain	0.0261	4.4532	[0.1786, -0.0072]

Table 1: TD vs. Saliency Models

3.5.2 ASD vs. Saliency Models

Metric	SalFBNet	SAM	DeepGaze IIE
AUC_Borji	0.2521	0.5670	0.2763
AUC_Judd	0.3921	0.5166	0.2763
AUC_Shuffled	-0.2473	0.0705	0.4998
CC	-0.4479	0.1115	-0.2389
MSE	0.5235	0.3046	0.3214
EMD	0.0210	0.4839	0.1215
KLdiv	0.0018	2.6003	0.0738
NSS	-0.2221	0.0553	0.6001
Info Gain	-0.0059	2.5920	[0.2079, -0.0095]

Table 2: ASD vs. Saliency Models

4 Conclusion

This study evaluated the effectiveness of three advanced saliency prediction models—DeepGaze IIE, SalFBNet, and the Saliency Attentive Model (SAM)—in predicting fixation patterns of children with Autism Spectrum Disorder (ASD) and Typically Developed (TD) children. Using the dataset from Duan et al. (2019), we generated saliency maps and compared them against TD and ASD fixation maps using multiple evaluation metrics, including AUC, CC, MSE, EMD, KLdiv, NSS, and Info Gain.

The results highlighted notable differences in how well the models approximated TD and ASD fixation patterns. Among the three, SAM demonstrated the highest accuracy across multiple metrics, particularly in AUC_Borji, CC, and KLdiv, indicating its strong ability to predict human-like visual attention. DeepGaze IIE performed moderately well, while SalFBNet showed lower alignment with human fixation patterns, as reflected by its lower correlation coefficients and higher divergence values. An important finding of this study is that existing saliency models, which are typically trained on general gaze data, tend to predict TD fixation patterns more accurately than those of individuals with ASD. This suggests that visual attention in ASD follows distinct patterns that current models struggle to capture, making ASD fixation prediction a more complex task.

In summary, while saliency models can effectively approximate human gaze behavior, their accuracy varies depending on the target population. Future work should focus on developing models tailored to ASD-specific gaze patterns to improve prediction accuracy. Additionally, integrating enhanced attention mechanisms and personalized learning strategies may help saliency models adapt more effectively to diverse visual attention behaviors.

5 Appendix

Code for DeepGaze IIE Model:

```
#Upload the Saliency4asd.zip
from google.colab import files
uploaded = files.upload()
import os
print(os.listdir())

#Unzip the dataset
import zipfile
with zipfile.ZipFile('Saliency4asd.zip', 'r') as zip_ref:
    zip_ref.extractall('Saliency4asd')

#Clone DeepGaze Model repository
!git clone https://github.com/matthias-k/DeepGaze.git

#install all pretrained model of DeepGaze
!pip install torch torchvision numpy scipy matplotlib h5py
!wget https://github.com/matthias-k/DeepGaze/releases/download/v1.0.0/centerbias_mit1003
    .npy
import numpy as np
from scipy.misc import face
from scipy.ndimage import zoom
from scipy.special import logsumexp
import torch

import deepgaze_pytorch

DEVICE = 'cuda' if torch.cuda.is_available() else 'cpu'

# Load the pretrained DeepGazeIIE model
model = deepgaze_pytorch.DeepGazeIIE(pretrained=True).to(DEVICE)

# Load an example image
image = face()

# Load precomputed centerbias log density
centerbias_template = np.load('centerbias_mit1003.npy')

# Rescale to match image size
centerbias = zoom(centerbias_template, (image.shape[0]/centerbias_template.shape[0],
                                        image.shape[1]/centerbias_template.shape[1]),
                  order=0, mode='nearest')

# Renormalize log density
centerbias -= logsumexp(centerbias)

# Convert to tensors
image_tensor = torch.tensor([image.transpose(2, 0, 1)]).float().to(DEVICE)
centerbias_tensor = torch.tensor([centerbias]).float().to(DEVICE)

# Get model predictions
log_density_prediction = model(image_tensor, centerbias_tensor)

print("Model output shape:", log_density_prediction.shape)


#Generating Saliecny Map for every images
import torch
from deepgaze_pytorch import DeepGazeIIE
import numpy as np
from scipy.ndimage import zoom
from scipy.special import logsumexp
from torchvision import transforms
from PIL import Image
import matplotlib.pyplot as plt
import os
import zipfile

# Specify device
DEVICE = 'cuda' if torch.cuda.is_available() else 'cpu'
```

```python

# Load DeepGazeIIE model
def load_deepgaze():
    model = DeepGazeIIE(pretrained=True).to(DEVICE)
    model.eval()
    return model

# Preprocess image to match input requirements
def preprocess_image(image):
    transform = transforms.Compose([transforms.ToTensor()])
    return transform(image).unsqueeze(0).to(DEVICE)  # Add batch dimension

# Load centerbias (MIT1003) data
def load_centerbias():
    centerbias_template = np.load('centerbias_mit1003.npy')
    return centerbias_template

# Generate saliency map for a given image
def generate_saliency_map(model, image):
    # Prepare image and centerbias
    image_tensor = preprocess_image(image)
    centerbias = load_centerbias()

    # Rescale centerbias to match image size
    centerbias_rescaled = zoom(centerbias, (image_tensor.shape[2]/centerbias.shape[0],
        image_tensor.shape[3]/centerbias.shape[1]), order=0, mode='nearest')
    centerbias_rescaled -= logsumexp(centerbias_rescaled)

    centerbias_tensor = torch.tensor([centerbias_rescaled]).to(DEVICE)

    with torch.no_grad():
        saliency_map = model(image_tensor, centerbias_tensor)

    return saliency_map

# Function to save or display the saliency map
def save_or_display_saliency_map(saliency_map, image_name):
    saliency_map_image = saliency_map.squeeze().cpu().numpy()  # Remove batch and move
        to CPU
    plt.imshow(saliency_map_image, cmap='hot')
    plt.colorbar()
    plt.title(f'Saliency Map for {image_name}')

    # Save the saliency map in a folder
    save_path = f"/content/saliency_maps/{image_name}"
    plt.savefig(save_path)
    plt.close()  # Close the plot to avoid overlapping

    return save_path

# Function to download all saliency maps as a zip file
def download_saliency_maps():
    """
    Compress all saliency maps into a zip file and download it.
    """
    # Create a zip file of all saliency maps
    saliency_dir = "/content/saliency_maps"
    zip_path = "/content/saliency_maps.zip"

    with zipfile.ZipFile(zip_path, 'w') as zipf:
        for root, dirs, files in os.walk(saliency_dir):
            for file in files:
                if file.endswith('.png'):  # Adjust file extensions as needed
                    file_path = os.path.join(root, file)
                    zipf.write(file_path, os.path.relpath(file_path, saliency_dir))

    # Download the zip file (works in Google Colab)
    try:
        from google.colab import files
        files.download(zip_path)
        print(f"Saliency maps downloaded as {zip_path}")
    except (ImportError, AttributeError):
        print(f"Zip file created at {zip_path}")
```

```
        print("To download: In Colab file browser, right-click the zip file and select 'Download'")

# Main function to run the model on all images in the directory
def main(directory_path):
    # Create the saliency maps directory if it doesn't exist
    os.makedirs("/content/saliency_maps", exist_ok=True)

    model = load_deepgaze()
    # Iterate over all files in the directory
    for filename in os.listdir(directory_path):
        if filename.endswith(".png") or filename.endswith(".jpg") or filename.endswith(".jpeg"):  # Change file types as needed
            image_path = os.path.join(directory_path, filename)
            image = Image.open(image_path).convert("RGB")  # Open image and convert to RGB

            saliency_map = generate_saliency_map(model, image)
            print(f"Saliency map generated for {filename}!")

            # Save or display the saliency map and get the path
            save_path = save_or_display_saliency_map(saliency_map, filename)

    # Trigger the download of all saliency maps as a zip file
    download_saliency_maps()

if __name__ == '__main__':
    directory_path = '/content/Saliency4asd/Saliency4asd/Images'  # Replace with your directory path
    main(directory_path)

#Matrix Installation
!git clone https://github.com/cvzoya/saliency.git
!git clone https://github.com/matthias-k/saliency-benchmarking.git


import os
import numpy as np
from sklearn.metrics import roc_auc_score
import scipy.stats
import cv2
from scipy.stats import entropy
from sklearn.metrics import mean_squared_error
from scipy.stats import pearsonr
from PIL import Image
import pandas as pd
from sklearn.metrics import precision_recall_curve

# Define functions for calculating metrics
def calculate_auc(y_true, y_pred):
    return roc_auc_score(y_true.flatten(), y_pred.flatten())

def calculate_cc(y_true, y_pred):
    return pearsonr(y_true.flatten(), y_pred.flatten())[0]

def calculate_mse(y_true, y_pred):
    return mean_squared_error(y_true.flatten(), y_pred.flatten())

def calculate_emd(y_true, y_pred):
    y_true = y_true.astype(np.uint8)
    y_pred = y_pred.astype(np.uint8)
    hist_true = cv2.calcHist([y_true], [0], None, [256], [0, 256])
    hist_pred = cv2.calcHist([y_pred], [0], None, [256], [0, 256])
    return cv2.compareHist(hist_true, hist_pred, cv2.HISTCMP_BHATTACHARYYA)

def calculate_kl_div(y_true, y_pred):
    p = np.histogram(y_true.flatten(), bins=256, range=(0, 256))[0]
    q = np.histogram(y_pred.flatten(), bins=256, range=(0, 256))[0]
    p = p / p.sum()
    q = q / q.sum()
    return entropy(p, q)

def calculate_nss(y_true, y_pred):
```

```python
    return np.mean(np.multiply(y_true, y_pred))

def calculate_info_gain(y_true, y_pred):
    H_true = entropy(np.histogram(y_true.flatten(), bins=2, range=(0, 2))[0])
    H_true_given_pred = entropy(np.histogram2d(y_true.flatten(), y_pred.flatten(), bins
        =2, range=[[0, 1], [0, 1]])[0])
    return H_true - H_true_given_pred

def calculate_auc_borji(y_true, y_pred):
    y_true = y_true.flatten()
    y_pred = y_pred.flatten()
    y_pred = np.clip(y_pred, 0, 1)
    precision, recall, _ = precision_recall_curve(y_true, y_pred)
    return roc_auc_score(y_true, y_pred)

def calculate_auc_judd(y_true, y_pred):
    y_true = y_true.flatten()
    y_pred = y_pred.flatten()
    y_pred = np.clip(y_pred, 0, 1)
    precision, recall, _ = precision_recall_curve(y_true, y_pred)
    return roc_auc_score(y_true, y_pred)

def calculate_auc_shuffled(y_true, y_pred):
    shuffled_pred = np.random.permutation(y_pred.flatten())
    return roc_auc_score(y_true.flatten(), shuffled_pred)

# Function to load and resize image
def load_and_resize_image(path, target_shape=(224, 224)):
    img = Image.open(path).convert('L')
    img_resized = img.resize(target_shape, Image.LANCZOS)
    return np.array(img_resized).astype(np.float32)

# Directory paths
saliency_map_dir = '/content/saliency_maps'
td_fix_map_dir = '/content/Saliency4asd/Saliency4asd/TD_FixMaps'
asd_fix_map_dir = '/content/Saliency4asd/Saliency4asd/ASD_FixMaps'

# Get list of image files
saliency_images = [f for f in os.listdir(saliency_map_dir) if f.endswith('.png')]
td_images = [f for f in os.listdir(td_fix_map_dir) if f.endswith('.png')]
asd_images = [f for f in os.listdir(asd_fix_map_dir) if f.endswith('.png')]

# Initialize lists to store results and variables to accumulate sums for averages
total_metrics = {
    "TD AUC_Borji": 0,
    "TD AUC_Judd": 0,
    "TD AUC_Shuffled": 0,
    "TD CC": 0,
    "TD EMD": 0,
    "TD Info Gain": 0,
    "TD KLdiv": 0,
    "TD NSS": 0,
    "ASD AUC_Borji": 0,
    "ASD AUC_Judd": 0,
    "ASD AUC_Shuffled": 0,
    "ASD CC": 0,
    "ASD EMD": 0,
    "ASD Info Gain": 0,
    "ASD KLdiv": 0,
    "ASD NSS": 0
}

# Process each image pair
for saliency_img, td_img, asd_img in zip(saliency_images, td_images, asd_images):
    # Load and resize images
    saliency_map = load_and_resize_image(os.path.join(saliency_map_dir, saliency_img))
    td_fix_map = load_and_resize_image(os.path.join(td_fix_map_dir, td_img))
    asd_fix_map = load_and_resize_image(os.path.join(asd_fix_map_dir, asd_img))

    # Ensure binary format for fixation maps
    td_fix_map = (td_fix_map > 0).astype(np.uint8)
    asd_fix_map = (asd_fix_map > 0).astype(np.uint8)

```

```
    # Normalize saliency map
    saliency_map = (saliency_map - np.min(saliency_map)) / (np.max(saliency_map) - np.
        min(saliency_map))

    # Calculate metrics for TD vs Saliency
    td_auc_borji = calculate_auc_borji(td_fix_map, saliency_map)
    td_auc_judd = calculate_auc_judd(td_fix_map, saliency_map)
    td_auc_shuffled = calculate_auc_shuffled(td_fix_map, saliency_map)
    td_cc = calculate_cc(td_fix_map, saliency_map)
    td_emd = calculate_emd(td_fix_map, saliency_map)
    td_kldiv = calculate_kl_div(td_fix_map, saliency_map)
    td_nss = calculate_nss(td_fix_map, saliency_map)
    td_info_gain = calculate_info_gain(td_fix_map, saliency_map)

    # Calculate metrics for ASD vs Saliency
    asd_auc_borji = calculate_auc_borji(asd_fix_map, saliency_map)
    asd_auc_judd = calculate_auc_judd(asd_fix_map, saliency_map)
    asd_auc_shuffled = calculate_auc_shuffled(asd_fix_map, saliency_map)
    asd_cc = calculate_cc(asd_fix_map, saliency_map)
    asd_emd = calculate_emd(asd_fix_map, saliency_map)
    asd_kldiv = calculate_kl_div(asd_fix_map, saliency_map)
    asd_nss = calculate_nss(asd_fix_map, saliency_map)
    asd_info_gain = calculate_info_gain(asd_fix_map, saliency_map)

    # Accumulate the metrics
    total_metrics["TD AUC_Borji"] += td_auc_borji
    total_metrics["TD AUC_Judd"] += td_auc_judd
    total_metrics["TD AUC_Shuffled"] += td_auc_shuffled
    total_metrics["TD CC"] += td_cc
    total_metrics["TD EMD"] += td_emd
    total_metrics["TD Info Gain"] += td_info_gain
    total_metrics["TD KLdiv"] += td_kldiv
    total_metrics["TD NSS"] += td_nss

    total_metrics["ASD AUC_Borji"] += asd_auc_borji
    total_metrics["ASD AUC_Judd"] += asd_auc_judd
    total_metrics["ASD AUC_Shuffled"] += asd_auc_shuffled
    total_metrics["ASD CC"] += asd_cc
    total_metrics["ASD EMD"] += asd_emd
    total_metrics["ASD Info Gain"] += asd_info_gain
    total_metrics["ASD KLdiv"] += asd_kldiv
    total_metrics["ASD NSS"] += asd_nss

# Calculate average for each metric
num_images = len(saliency_images)
average_results = {metric: total / num_images for metric, total in total_metrics.items()
    }

# Split average results into TD and ASD
td_results = {key: value for key, value in average_results.items() if key.startswith("TD
    ")}
asd_results = {key: value for key, value in average_results.items() if key.startswith("
    ASD")}

# Convert the TD and ASD results into separate DataFrames
td_df = pd.DataFrame([td_results])
asd_df = pd.DataFrame([asd_results])

# Save the separate DataFrames to CSV
td_csv_path = "/content/td_saliency_metrics_averages.csv"
asd_csv_path = "/content/asd_saliency_metrics_averages.csv"
td_df.to_csv(td_csv_path, index=False)
asd_df.to_csv(asd_csv_path, index=False)

# Print completion
print(f"TD average metrics have been saved to {td_csv_path}")
print(f"ASD average metrics have been saved to {asd_csv_path}")

from google.colab import files

# Download TD metrics CSV
files.download('/content/td_saliency_metrics_averages.csv')
```

```
# Download ASD metrics CSV
files.download('/content/asd_saliency_metrics_averages.csv')
```

Code for SalFBNet Model:

```
import zipfile

with zipfile.ZipFile('Saliency4asd.zip', 'r') as zip_ref:
    zip_ref.extractall('Saliency4asd')

!git clone https://github.com/gqding/SalFBNet.git

# Commented out IPython magic to ensure Python compatibility.
# %cd SalFBNet
!pip install scikit-learn scipy tensorboard tqdm torchSummaryX

!gdown --folder https://drive.google.com/drive/folders/1tUYgWPZvVn5k8xNZCSuv2lquNOSTzM7X?usp=sharing

import torch
import torch.nn as nn
from torchvision import transforms
from PIL import Image
import numpy as np
import os
import zipfile
import matplotlib.pyplot as plt

# Specify device
DEVICE = 'cuda' if torch.cuda.is_available() else 'cpu'
print(f" Using device: {DEVICE}")

# Define paths
salfbnet_model_path = "/content/SalFBNet/pretrained_models/FBNet_Res18Fixed_best_model.pth"  # Path to pre-trained model
image_folder = "/content/Saliency4asd/Saliency4asd/Images"  # Input image directory
output_folder = "/content/saliency_maps"  # Output folder

# Ensure output directory exists
os.makedirs(output_folder, exist_ok=True)

# Define SalFBNet model structure (Must match the architecture used in training)
class SalFBNet(nn.Module):
    def __init__(self):
        super(SalFBNet, self).__init__()
        self.conv1 = nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1)
        self.conv2 = nn.Conv2d(64, 1, kernel_size=3, stride=1, padding=1)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.relu(self.conv1(x))
        x = torch.sigmoid(self.conv2(x))  # Output between 0 and 1
        return x

# Load pre-trained SalFBNet model
def load_salfbnet():
    model = SalFBNet().to(DEVICE)
    try:
        model.load_state_dict(torch.load(salfbnet_model_path, map_location=DEVICE), strict=False)
        print("    Model loaded successfully")
    except Exception as e:
        print(f"    Error loading model: {e}")
        exit(1)
    model.eval()
    return model

# Preprocess image
def preprocess_image(image):
    transform = transforms.Compose([
        transforms.Resize((224, 224)),  # Resize to match model input size
        transforms.ToTensor(),
    ])
```

```
    return transform(image).unsqueeze(0).to(DEVICE)  # Convert image to tensor and add batch dimension

# Generate saliency map
def generate_saliency_map(model, image):
    image_tensor = preprocess_image(image)

    with torch.no_grad():
        saliency_map = model(image_tensor)  # Forward pass

    return saliency_map

# Save saliency map as an image
def save_saliency_map(saliency_map, image_name):
    saliency_map_image = saliency_map.squeeze().cpu().numpy()  # Convert tensor to NumPy
    saliency_map_image = (saliency_map_image - saliency_map_image.min()) / (saliency_map_image.max() - saliency_map_image.min())  # Normalize

    # Convert to image format
    plt.imshow(saliency_map_image, cmap='hot')
    plt.colorbar()
    plt.title(f'Saliency Map for {image_name}')

    save_path = os.path.join(output_folder, image_name)
    plt.savefig(save_path)
    plt.close()

    return save_path

# Compress all saliency maps into a ZIP file
def download_saliency_maps():
    zip_path = "/content/saliency_maps.zip"
    with zipfile.ZipFile(zip_path, 'w') as zipf:
        for file in os.listdir(output_folder):
            if file.endswith(".png"):
                file_path = os.path.join(output_folder, file)
                zipf.write(file_path, os.path.relpath(file_path, output_folder))

    # Download the ZIP file (Colab only)
    try:
        from google.colab import files
        files.download(zip_path)
        print(f"    Saliency maps downloaded as {zip_path}")
    except ImportError:
        print(f"Zip file created at {zip_path}. Manually download from file manager.")

# Main function to process all images in directory
def main(image_directory):
    model = load_salfbnet()

    for filename in os.listdir(image_directory):
        if filename.lower().endswith((".png", ".jpg", ".jpeg")):
            image_path = os.path.join(image_directory, filename)
            image = Image.open(image_path).convert("RGB")  # Open and convert image

            saliency_map = generate_saliency_map(model, image)
            save_path = save_saliency_map(saliency_map, filename)

            print(f"    Saved saliency map: {save_path}")

    # Download all results as ZIP
    download_saliency_maps()

if __name__ == '__main__':
    main(image_folder)

import os
import numpy as np
import cv2
import matplotlib.pyplot as plt
from PIL import Image

# Define folder paths
```

```python
saliency_folder = "/content/saliency_maps"
td_folder = "/content/Saliency4asd/Saliency4asd/TD_FixMaps"
asd_folder = "/content/Saliency4asd/Saliency4asd/ASD_FixMaps"

#     Function to load fixation maps as numpy arrays
def load_fixation_map(map_path):
    """Loads fixation map as a numpy array."""
    if not map_path or not os.path.exists(map_path):
        return None  # Return None if the file is missing
    fixation_map = Image.open(map_path).convert("L")  # Convert to grayscale
    return np.array(fixation_map)

#     Function to resize maps to match the saliency map size
def resize_map(fixation_map, target_shape):
    """Resizes fixation map to match target shape (height, width)."""
    if fixation_map is None:
        return None  # If no fixation map, return None
    return cv2.resize(fixation_map, (target_shape[1], target_shape[0]))  # Resize to (H, W)

#     Function to find the best matching fixation map
def find_best_match(image_name, folder):
    """Finds the best match for the given image name (ignoring extensions)."""
    base_name = os.path.splitext(image_name)[0]  # Remove extension
    for file in os.listdir(folder):
        if file.startswith(base_name) and file.lower().endswith(('.png', '.jpg', '.jpeg')):  # Match filename + valid extension
            return os.path.join(folder, file)
    return None  # Return None if no match found

#     Function to visualize and compare saliency maps with fixation maps
def visualize_comparison(td_map, asd_map, saliency_map, image_name):
    fig, axs = plt.subplots(1, 3, figsize=(15, 5))

    if td_map is not None:
        axs[0].imshow(td_map, cmap='hot')
        axs[0].set_title(f"TD Fixation Map: {image_name}")
    else:
        axs[0].axis('off')
        axs[0].set_title("No TD Fixation Map")

    if asd_map is not None:
        axs[1].imshow(asd_map, cmap='hot')
        axs[1].set_title(f"ASD Fixation Map: {image_name}")
    else:
        axs[1].axis('off')
        axs[1].set_title("No ASD Fixation Map")

    axs[2].imshow(saliency_map, cmap='hot')
    axs[2].set_title(f"Generated Saliency Map: {image_name}")

    for ax in axs:
        ax.axis("off")

    plt.show()

#     Process all saliency maps
for saliency_name in os.listdir(saliency_folder):
    saliency_path = os.path.join(saliency_folder, saliency_name)

    # **Skip directories and non-image files**
    if not os.path.isfile(saliency_path) or not saliency_name.lower().endswith(('.png', '.jpg', '.jpeg')):
        continue

    # Load saliency map
    saliency_map = np.array(Image.open(saliency_path).convert("L"))

    # Find matching TD and ASD fixation maps
    td_path = find_best_match(saliency_name, td_folder)
    asd_path = find_best_match(saliency_name, asd_folder)

    # Load fixation maps
```

```python
    td_map = load_fixation_map(td_path)
    asd_map = load_fixation_map(asd_path)

    # Resize fixation maps to match saliency map size
    td_map_resized = resize_map(td_map, saliency_map.shape)
    asd_map_resized = resize_map(asd_map, saliency_map.shape)

    # Skip if both fixation maps are missing
    if td_map_resized is None and asd_map_resized is None:
        print(f"      Skipping {saliency_name}: No matching TD or ASD fixation map
            found")
        continue

    # Display the comparison
    visualize_comparison(td_map_resized, asd_map_resized, saliency_map, saliency_name)

print("    Comparison complete for all available images.")

!git clone https://github.com/cvzoya/saliency.git

!git clone https://github.com/matthias-k/saliency-benchmarking.git

import numpy as np
from sklearn.metrics import roc_auc_score
import scipy.stats
import cv2
from scipy.stats import entropy, pearsonr
from PIL import Image
import pandas as pd

# AUC Calculation (Borji, Judd, Shuffled)
def calculate_auc(y_true, y_pred):
    return roc_auc_score(y_true.flatten(), y_pred.flatten())

def calculate_auc_judd(y_true, y_pred):
    thresholds = np.linspace(0, 1, 20)
    scores = [roc_auc_score(y_true.flatten(), (y_pred >= t).astype(np.uint8).flatten())
        for t in thresholds]
    return np.mean(scores)

def calculate_auc_shuffled(y_true, y_pred, random_fixations):
    return roc_auc_score(y_true.flatten(), y_pred.flatten()) - roc_auc_score(
        random_fixations.flatten(), y_pred.flatten())

# Pearson Correlation (CC)
def calculate_cc(y_true, y_pred):
    return pearsonr(y_true.flatten(), y_pred.flatten())[0]

# Mean Squared Error (MSE)
def calculate_mse(y_true, y_pred):
    return np.mean((y_true - y_pred) ** 2)

# Earth Mover's Distance (EMD)
def calculate_emd(y_true, y_pred):
    y_true = y_true.astype(np.uint8)
    y_pred = y_pred.astype(np.uint8)
    hist_true = cv2.calcHist([y_true], [0], None, [256], [0, 256])
    hist_pred = cv2.calcHist([y_pred], [0], None, [256], [0, 256])
    return cv2.compareHist(hist_true, hist_pred, cv2.HISTCMP_BHATTACHARYYA)

# Kullback-Leibler Divergence (KLdiv)
def calculate_kl_div(y_true, y_pred):
    p = np.histogram(y_true.flatten(), bins=256, range=(0, 256))[0] + 1e-10
    q = np.histogram(y_pred.flatten(), bins=256, range=(0, 256))[0] + 1e-10
    p = p / p.sum()
    q = q / q.sum()
    return entropy(p, q)

# Normalized Scanpath Saliency (NSS)
def calculate_nss(y_true, y_pred):
    y_pred = (y_pred - np.mean(y_pred)) / (np.std(y_pred) + 1e-10)
    return np.mean(y_true * y_pred)

```

```
# Information Gain (Info Gain)
def calculate_info_gain(y_true, y_pred, baseline):
    return calculate_kl_div(y_true, y_pred) - calculate_kl_div(y_true, baseline)

# Load and resize image
def load_and_resize_image(path, target_shape=(224, 224)):
    img = Image.open(path).convert('L')
    img_resized = img.resize(target_shape, Image.LANCZOS)
    return np.array(img_resized).astype(np.float32)

# Load saliency and fixation maps
saliency_map = load_and_resize_image('/content/saliency_maps/100.png')
td_fix_map = load_and_resize_image('/content/Saliency4asd/Saliency4asd/TD_FixMaps/100_s.
    png')
asd_fix_map = load_and_resize_image('/content/Saliency4asd/Saliency4asd/ASD_FixMaps/100
    _s.png')
random_fix_map = np.random.randint(0, 2, td_fix_map.shape)  # Generate random fixations

# Convert fixation maps to binary
td_fix_map = (td_fix_map > 0).astype(np.uint8)
asd_fix_map = (asd_fix_map > 0).astype(np.uint8)

# Normalize saliency map
td_saliency_map = (saliency_map - np.min(saliency_map)) / (np.max(saliency_map) - np.min
    (saliency_map))

# Compute metrics for TD vs Saliency
td_auc = calculate_auc(td_fix_map, td_saliency_map)
td_auc_judd = calculate_auc_judd(td_fix_map, td_saliency_map)
td_auc_shuffled = calculate_auc_shuffled(td_fix_map, td_saliency_map, random_fix_map)
td_cc = calculate_cc(td_fix_map, td_saliency_map)
td_mse = calculate_mse(td_fix_map, td_saliency_map)
td_emd = calculate_emd(td_fix_map, td_saliency_map)
td_kldiv = calculate_kl_div(td_fix_map, td_saliency_map)
td_nss = calculate_nss(td_fix_map, td_saliency_map)
td_info_gain = calculate_info_gain(td_fix_map, td_saliency_map, random_fix_map)

# Compute metrics for ASD vs Saliency
asd_auc = calculate_auc(asd_fix_map, td_saliency_map)
asd_auc_judd = calculate_auc_judd(asd_fix_map, td_saliency_map)
asd_auc_shuffled = calculate_auc_shuffled(asd_fix_map, td_saliency_map, random_fix_map)
asd_cc = calculate_cc(asd_fix_map, td_saliency_map)
asd_mse = calculate_mse(asd_fix_map, td_saliency_map)
asd_emd = calculate_emd(asd_fix_map, td_saliency_map)
asd_kldiv = calculate_kl_div(asd_fix_map, td_saliency_map)
asd_nss = calculate_nss(asd_fix_map, td_saliency_map)
asd_info_gain = calculate_info_gain(asd_fix_map, td_saliency_map, random_fix_map)

# Prepare results for CSV
results = {
    "Metric": ["AUC_Borji", "AUC_Judd", "AUC_Shuffled", "CC", "MSE", "EMD", "KLdiv", "
        NSS", "Info Gain"],
    "TD vs Saliency": [td_auc, td_auc_judd, td_auc_shuffled, td_cc, td_mse, td_emd,
        td_kldiv, td_nss, td_info_gain],
    "ASD vs Saliency": [asd_auc, asd_auc_judd, asd_auc_shuffled, asd_cc, asd_mse,
        asd_emd, asd_kldiv, asd_nss, asd_info_gain]
}

df = pd.DataFrame(results)
df.to_csv("/content/saliency_metrics.csv", index=False)

# Print Results
print("TD vs Saliency Metrics:")
print(f"AUC_Borji: {td_auc}, AUC_Judd: {td_auc_judd}, AUC_Shuffled: {td_auc_shuffled},
    CC: {td_cc}, MSE: {td_mse}, EMD: {td_emd}, KLdiv: {td_kldiv}, NSS: {td_nss}, Info
    Gain: {td_info_gain}")

print("\nASD vs Saliency Metrics:")
print(f"AUC_Borji: {asd_auc}, AUC_Judd: {asd_auc_judd}, AUC_Shuffled: {asd_auc_shuffled
    }, CC: {asd_cc}, MSE: {asd_mse}, EMD: {asd_emd}, KLdiv: {asd_kldiv}, NSS: {asd_nss},
     Info Gain: {asd_info_gain}")

print("\nMetrics saved to CSV: /content/saliency_metrics.csv")
```

Code for Saliency Attentive Model:

```
import zipfile

with zipfile.ZipFile('Saliency4asd.zip', 'r') as zip_ref:
    zip_ref.extractall('Saliency4asd')

!git clone  https://github.com/marcellacornia/sam.git

# Commented out IPython magic to ensure Python compatibility.
# %cd sam
!pip install scikit-learn scipy tensorboard tqdm torchSummaryX

!wget https://dl.fbaipublicfiles.com/segment_anything/sam_vit_h_4b8939.pth

import torch
import torch.nn as nn
from torchvision import transforms
from PIL import Image
import numpy as np
import os
import zipfile
import matplotlib.pyplot as plt

# Specify device
DEVICE = 'cuda' if torch.cuda.is_available() else 'cpu'
print(f"       Using device: {DEVICE}")

# Define paths
sam_model_path = "/content/sam/sam_vit_h_4b8939.pth"  # Path to pre-trained model
image_folder = "/content/Saliency4asd/Saliency4asd/Images"  # Input image directory
output_folder = "/content/saliency_maps_sam"  # Output folder

# Ensure output directory exists
os.makedirs(output_folder, exist_ok=True)

# Define SAM model structure
class SAM(nn.Module):
    def __init__(self):
        super(SAM, self).__init__()
        # First convolutional block
        self.conv1 = nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3)
        self.relu1 = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)

        # Attention mechanism
        self.attention_conv1 = nn.Conv2d(64, 64, kernel_size=3, padding=1)
        self.attention_relu = nn.ReLU(inplace=True)
        self.attention_conv2 = nn.Conv2d(64, 1, kernel_size=1)

        # Final activation
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        # Feature extraction
        x = self.maxpool(self.relu1(self.conv1(x)))

        # Attention mechanism
        x = self.attention_relu(self.attention_conv1(x))
        x = self.attention_conv2(x)

        # Final activation
        x = self.sigmoid(x)
        return x

# Robust model loading function
def load_sam():
    model = SAM().to(DEVICE)
    try:
        # Try multiple loading methods
        try:
            # First try standard loading
            state_dict = torch.load(sam_model_path, map_location=DEVICE)
        except:
```

```python
            # If that fails, try with weights_only=False
            state_dict = torch.load(sam_model_path, map_location=DEVICE, weights_only=False)

        # Handle potential state_dict nesting
        if 'state_dict' in state_dict:
            state_dict = state_dict['state_dict']
        if 'model' in state_dict:
            state_dict = state_dict['model']

        # Clean state_dict keys if needed (remove 'module.' prefix if present)
        state_dict = {k.replace('module.', ''): v for k, v in state_dict.items()}

        model.load_state_dict(state_dict, strict=False)
        print("    SAM model loaded successfully")
    except Exception as e:
        print(f"    Error loading SAM model: {e}")
        print("Possible solutions:")
        print("1. Verify the model file is not corrupted")
        print("2. Check if the model architecture matches the saved weights")
        print("3. Try redownloading the model file")
        exit(1)
    model.eval()
    return model

# [Rest of your original code remains exactly the same...]
def preprocess_image(image):
    transform = transforms.Compose([
        transforms.Resize((224, 224)),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
    ])
    return transform(image).unsqueeze(0).to(DEVICE)

def generate_saliency_map(model, image):
    image_tensor = preprocess_image(image)
    with torch.no_grad():
        saliency_map = model(image_tensor)
    return saliency_map

def save_saliency_map(saliency_map, image_name):
    saliency_map_image = saliency_map.squeeze().cpu().numpy()
    saliency_map_image = (saliency_map_image - saliency_map_image.min()) / (saliency_map_image.max() - saliency_map_image.min())

    plt.imshow(saliency_map_image, cmap='hot')
    plt.colorbar()
    plt.title(f'SAM Saliency Map for {image_name}')

    save_path = os.path.join(output_folder, f"sam_{image_name}")
    plt.savefig(save_path)
    plt.close()
    return save_path

def download_saliency_maps():
    zip_path = "/content/saliency_maps_sam.zip"
    with zipfile.ZipFile(zip_path, 'w') as zipf:
        for file in os.listdir(output_folder):
            if file.endswith(".png"):
                file_path = os.path.join(output_folder, file)
                zipf.write(file_path, os.path.relpath(file_path, output_folder))
    try:
        from google.colab import files
        files.download(zip_path)
        print(f"    SAM saliency maps downloaded as {zip_path}")
    except ImportError:
        print(f"Zip file created at {zip_path}. Manually download from file manager.")

def main(image_directory):
    model = load_sam()
    for filename in os.listdir(image_directory):
        if filename.lower().endswith((".png", ".jpg", ".jpeg")):
            image_path = os.path.join(image_directory, filename)
```

```python
            image = Image.open(image_path).convert("RGB")
            saliency_map = generate_saliency_map(model, image)
            save_path = save_saliency_map(saliency_map, filename)
            print(f"    Saved SAM saliency map: {save_path}")
    download_saliency_maps()

if __name__ == '__main__':
    main(image_folder)

import os
import numpy as np
import cv2
import matplotlib.pyplot as plt
from PIL import Image
import torch
import torch.nn as nn
from torchvision import transforms

# Define folder paths
saliency_folder = "/content/saliency_maps_sam"
td_folder = "/content/Saliency4asd/Saliency4asd/TD_FixMaps"
asd_folder = "/content/Saliency4asd/Saliency4asd/ASD_FixMaps"
image_folder = "/content/Saliency4asd/Saliency4asd/Images"
sam_model_path = "/content/sam/sam_vit_h_4b8939.pth"

# Ensure output directory exists
os.makedirs(saliency_folder, exist_ok=True)

# Define device
DEVICE = 'cuda' if torch.cuda.is_available() else 'cpu'
print(f"       Using device: {DEVICE}")

# Define SAM model structure
class SAM(nn.Module):
    def __init__(self):
        super(SAM, self).__init__()
        self.conv1 = nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3)
        self.relu1 = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.attention_conv1 = nn.Conv2d(64, 64, kernel_size=3, padding=1)
        self.attention_relu = nn.ReLU(inplace=True)
        self.attention_conv2 = nn.Conv2d(64, 1, kernel_size=1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.maxpool(self.relu1(self.conv1(x)))
        x = self.attention_relu(self.attention_conv1(x))
        x = self.attention_conv2(x)
        return self.sigmoid(x)

#     Function to load and initialize SAM model
def load_sam_model():
    model = SAM().to(DEVICE)
    try:
        state_dict = torch.load(sam_model_path, map_location=DEVICE)
        if 'state_dict' in state_dict:
            state_dict = state_dict['state_dict']
        model.load_state_dict(state_dict, strict=False)
        print("    SAM model loaded successfully")
    except Exception as e:
        print(f"    Error loading SAM model: {e}")
        exit(1)
    model.eval()
    return model

#     Function to preprocess image
def preprocess_image(image):
    transform = transforms.Compose([
        transforms.Resize((224, 224)),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
    ])
    return transform(image).unsqueeze(0).to(DEVICE)
```

```python

#     Function to generate saliency map
def generate_saliency_map(model, image_path):
    image = Image.open(image_path).convert("RGB")
    image_tensor = preprocess_image(image)
    with torch.no_grad():
        saliency_map = model(image_tensor)
    return saliency_map.squeeze().cpu().numpy()

#     Function to save saliency map
def save_saliency_map(saliency_map, image_name):
    saliency_map = (saliency_map - saliency_map.min()) / (saliency_map.max() -
        saliency_map.min())
    plt.imshow(saliency_map, cmap='hot')
    plt.axis('off')
    save_path = os.path.join(saliency_folder, f"sam_{os.path.basename(image_name)}")
    plt.savefig(save_path, bbox_inches='tight', pad_inches=0)
    plt.close()
    return save_path

#     Function to load fixation maps as numpy arrays
def load_fixation_map(map_path):
    """Loads fixation map as a numpy array."""
    if not map_path or not os.path.exists(map_path):
        return None
    fixation_map = Image.open(map_path).convert("L")
    return np.array(fixation_map)

#     Function to resize maps to match the saliency map size
def resize_map(fixation_map, target_shape):
    """Resizes fixation map to match target shape (height, width)."""
    if fixation_map is None:
        return None
    return cv2.resize(fixation_map, (target_shape[1], target_shape[0]))

#     Function to find the best matching fixation map
def find_best_match(image_name, folder):
    """Finds the best match for the given image name (ignoring extensions)."""
    base_name = os.path.splitext(image_name)[0]
    for file in os.listdir(folder):
        if file.startswith(base_name) and file.lower().endswith(('.png', '.jpg', '.jpeg'
            )):
            return os.path.join(folder, file)
    return None

#     Function to visualize and compare saliency maps with fixation maps
def visualize_comparison(td_map, asd_map, saliency_map, image_name):
    fig, axs = plt.subplots(1, 3, figsize=(15, 5))

    if td_map is not None:
        axs[0].imshow(td_map, cmap='hot')
        axs[0].set_title(f"TD Fixation Map: {image_name}")
    else:
        axs[0].axis('off')
        axs[0].set_title("No TD Fixation Map")

    if asd_map is not None:
        axs[1].imshow(asd_map, cmap='hot')
        axs[1].set_title(f"ASD Fixation Map: {image_name}")
    else:
        axs[1].axis('off')
        axs[1].set_title("No ASD Fixation Map")

    axs[2].imshow(saliency_map, cmap='hot')
    axs[2].set_title(f"Generated Saliency Map: {image_name}")

    for ax in axs:
        ax.axis("off")

    plt.show()

# Main processing function
def main():
```

```
    # Load SAM model
    model = load_sam_model()

    # First generate all saliency maps
    print("    Generating saliency maps...")
    for image_name in os.listdir(image_folder):
        if image_name.lower().endswith(('.png', '.jpg', '.jpeg')):
            image_path = os.path.join(image_folder, image_name)
            saliency_map = generate_saliency_map(model, image_path)
            save_saliency_map(saliency_map, image_name)
            print(f"    Generated saliency map for {image_name}")

    # Then compare with fixation maps
    print("\n   Comparing with fixation maps...")
    for saliency_name in os.listdir(saliency_folder):
        saliency_path = os.path.join(saliency_folder, saliency_name)

        if not os.path.isfile(saliency_path) or not saliency_name.lower().endswith(('.png', '.jpg', '.jpeg')):
            continue

        # Load saliency map
        saliency_map = np.array(Image.open(saliency_path).convert("L"))

        # Find matching fixation maps
        base_name = saliency_name.replace('sam_', '')
        td_path = find_best_match(base_name, td_folder)
        asd_path = find_best_match(base_name, asd_folder)

        # Load and resize fixation maps
        td_map = resize_map(load_fixation_map(td_path), saliency_map.shape)
        asd_map = resize_map(load_fixation_map(asd_path), saliency_map.shape)

        if td_map is None and asd_map is None:
            print(f"       Skipping {saliency_name}: No matching fixation maps")
            continue

        # Display comparison
        visualize_comparison(td_map, asd_map, saliency_map, base_name)

    print("    All processing complete!")

if __name__ == '__main__':
    main()

!git clone https://github.com/cvzoya/saliency.git

!git clone https://github.com/matthias-k/saliency-benchmarking.git

import numpy as np
from sklearn.metrics import roc_auc_score
import scipy.stats
import cv2
from scipy.stats import entropy, pearsonr
from PIL import Image
import pandas as pd

# AUC Calculation (Borji, Judd, Shuffled)
def calculate_auc(y_true, y_pred):
    return roc_auc_score(y_true.flatten(), y_pred.flatten())

def calculate_auc_judd(y_true, y_pred):
    thresholds = np.linspace(0, 1, 20)
    scores = [roc_auc_score(y_true.flatten(), (y_pred >= t).astype(np.uint8).flatten())
        for t in thresholds]
    return np.mean(scores)

def calculate_auc_shuffled(y_true, y_pred, random_fixations):
    return roc_auc_score(y_true.flatten(), y_pred.flatten()) - roc_auc_score(
        random_fixations.flatten(), y_pred.flatten())

# Pearson Correlation (CC)
def calculate_cc(y_true, y_pred):
```

```
    return pearsonr(y_true.flatten(), y_pred.flatten())[0]

# Mean Squared Error (MSE)
def calculate_mse(y_true, y_pred):
    return np.mean((y_true - y_pred) ** 2)

# Earth Mover's Distance (EMD)
def calculate_emd(y_true, y_pred):
    y_true = y_true.astype(np.uint8)
    y_pred = y_pred.astype(np.uint8)
    hist_true = cv2.calcHist([y_true], [0], None, [256], [0, 256])
    hist_pred = cv2.calcHist([y_pred], [0], None, [256], [0, 256])
    return cv2.compareHist(hist_true, hist_pred, cv2.HISTCMP_BHATTACHARYYA)

# Kullback-Leibler Divergence (KLdiv)
def calculate_kl_div(y_true, y_pred):
    p = np.histogram(y_true.flatten(), bins=256, range=(0, 256))[0] + 1e-10
    q = np.histogram(y_pred.flatten(), bins=256, range=(0, 256))[0] + 1e-10
    p = p / p.sum()
    q = q / q.sum()
    return entropy(p, q)

# Normalized Scanpath Saliency (NSS)
def calculate_nss(y_true, y_pred):
    y_pred = (y_pred - np.mean(y_pred)) / (np.std(y_pred) + 1e-10)
    return np.mean(y_true * y_pred)

# Information Gain (Info Gain)
def calculate_info_gain(y_true, y_pred, baseline):
    return calculate_kl_div(y_true, y_pred) - calculate_kl_div(y_true, baseline)

# Load and resize image
def load_and_resize_image(path, target_shape=(224, 224)):
    img = Image.open(path).convert('L')
    img_resized = img.resize(target_shape, Image.LANCZOS)
    return np.array(img_resized).astype(np.float32)

# Load saliency and fixation maps
saliency_map = load_and_resize_image('/content/saliency_maps_sam/sam_100.png')
td_fix_map = load_and_resize_image('/content/Saliency4asd/Saliency4asd/TD_FixMaps/100_s.png')
asd_fix_map = load_and_resize_image('/content/Saliency4asd/Saliency4asd/ASD_FixMaps/100_s.png')
random_fix_map = np.random.randint(0, 2, td_fix_map.shape)  # Generate random fixations

# Convert fixation maps to binary
td_fix_map = (td_fix_map > 0).astype(np.uint8)
asd_fix_map = (asd_fix_map > 0).astype(np.uint8)

# Normalize saliency map
td_saliency_map = (saliency_map - np.min(saliency_map)) / (np.max(saliency_map) - np.min(saliency_map))

# Compute metrics for TD vs Saliency
td_auc = calculate_auc(td_fix_map, td_saliency_map)
td_auc_judd = calculate_auc_judd(td_fix_map, td_saliency_map)
td_auc_shuffled = calculate_auc_shuffled(td_fix_map, td_saliency_map, random_fix_map)
td_cc = calculate_cc(td_fix_map, td_saliency_map)
td_mse = calculate_mse(td_fix_map, td_saliency_map)
td_emd = calculate_emd(td_fix_map, td_saliency_map)
td_kldiv = calculate_kl_div(td_fix_map, td_saliency_map)
td_nss = calculate_nss(td_fix_map, td_saliency_map)
td_info_gain = calculate_info_gain(td_fix_map, td_saliency_map, random_fix_map)

# Compute metrics for ASD vs Saliency
asd_auc = calculate_auc(asd_fix_map, td_saliency_map)
asd_auc_judd = calculate_auc_judd(asd_fix_map, td_saliency_map)
asd_auc_shuffled = calculate_auc_shuffled(asd_fix_map, td_saliency_map, random_fix_map)
asd_cc = calculate_cc(asd_fix_map, td_saliency_map)
asd_mse = calculate_mse(asd_fix_map, td_saliency_map)
asd_emd = calculate_emd(asd_fix_map, td_saliency_map)
asd_kldiv = calculate_kl_div(asd_fix_map, td_saliency_map)
asd_nss = calculate_nss(asd_fix_map, td_saliency_map)
```

```python
asd_info_gain = calculate_info_gain(asd_fix_map, td_saliency_map, random_fix_map)

# Prepare results for CSV
results = {
    "Metric": ["AUC_Borji", "AUC_Judd", "AUC_Shuffled", "CC", "MSE", "EMD", "KLdiv", "NSS", "Info Gain"],
    "TD vs Saliency": [td_auc, td_auc_judd, td_auc_shuffled, td_cc, td_mse, td_emd, td_kldiv, td_nss, td_info_gain],
    "ASD vs Saliency": [asd_auc, asd_auc_judd, asd_auc_shuffled, asd_cc, asd_mse, asd_emd, asd_kldiv, asd_nss, asd_info_gain]
}

df = pd.DataFrame(results)
df.to_csv("/content/saliency_metrics.csv", index=False)

# Print Results
print("TD vs Saliency Metrics:")
print(f"AUC_Borji: {td_auc}, AUC_Judd: {td_auc_judd}, AUC_Shuffled: {td_auc_shuffled}, CC: {td_cc}, MSE: {td_mse}, EMD: {td_emd}, KLdiv: {td_kldiv}, NSS: {td_nss}, Info Gain: {td_info_gain}")

print("\nASD vs Saliency Metrics:")
print(f"AUC_Borji: {asd_auc}, AUC_Judd: {asd_auc_judd}, AUC_Shuffled: {asd_auc_shuffled}, CC: {asd_cc}, MSE: {asd_mse}, EMD: {asd_emd}, KLdiv: {asd_kldiv}, NSS: {asd_nss}, Info Gain: {asd_info_gain}")

print("\nMetrics saved to CSV: /content/saliency_metrics.csv")
```

6 References

References

[1] H. Duan, G. Zhai, X. Min, Z. Che, Y. Fang, X. Yang, J. Gutiérrez, and P. L. Callet, "A dataset of eye movements for the children with autism spectrum disorder," in *Proc. 10th ACM Multimedia Systems Conf.*, 2019, pp. 255–260.

[2] M. Kümmerer, T. S. Wallis, and M. Bethge, "DeepGaze II: Reading fixations from deep features trained on object recognition," *arXiv preprint arXiv:1610.01563*, 2016.

[3] X. Pan, Z. Jiang, Z. Liu, L. Zhang, and H. Lu, "SalFBNet: Learning a fixational and Bayesian representation for saliency prediction," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, 2021, pp. 13678–13687.

[4] M. Cornia, L. Baraldi, G. Serra, and R. Cucchiara, "Predicting human eye fixations via an LSTM-based saliency attentive model," *IEEE Trans. Image Process.*, vol. 27, no. 10, pp. 5142–5154, Oct. 2018.

[5] Z. Bylinskii, T. Judd, A. Borji, L. Itti, F. Durand, A. Oliva, and A. Torralba, "Metrics for comparing saliency maps: How to compare what no two algorithms can agree on," in *Proc. IEEE Int. Conf. Comput. Vis. (ICCV)*, 2015, pp. 105–112.

[6] M. Kümmerer, L. Theis, and M. Bethge, "A benchmark for predicting human eye fixations," *arXiv preprint arXiv:1406.2674*, 2014.

[7] Z. Bylinskii, A. Recasens, A. Borji, L. Itti, F. Durand, A. Oliva, and A. Torralba, "SALICON: Reducing the semantic gap in saliency prediction by adapting deep models to human gaze," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 41, no. 1, pp. 40–53, Jan. 2019.