Universidade de São Paulo
Instituto de Matemática e Estatística
Bachalerado em Ciência da Computação

Matheus Santos Conceição

# Rastreador Ocular Baseado em Redes Neurais de Convolução

São Paulo
Dezembro de 2021

# Rastreador Ocular Baseado em Redes Neurais de Convolução

Monografia final da disciplina
MAC0499 – Trabalho de Formatura Supervisionado.

Supervisor: Prof. Dr. Carlos Hitoshi Morimoto

São Paulo
Dezembro de 2021

# Resumo

Matheus Santos Conceição. **Rastreador Ocular Baseado em Redes Neurais de Convolução**. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2021.

O uso de rastreadores do olhar (eye tracker), que inferem a localização de foco do olho numa tela a partir de imagens do próprio olho, não é novidade, entretanto eye trackers tradicionais exigem algum tipo de hardware especializado, câmeras de alta resolução ou infravermelhas. Isso torna o processo de coleta de dados mais complicado, tendo a necessidade de ser realizada em laboratório.

Dado o elevado número de aparelhos eletrônicos com câmeras presentes atualmente, esse trabalho propõem explorar a difusão de câmeras em aparelhos eletrônicos como notebooks, para o desenvolvimento de um modelo de rastreador ocular de baixo custo.

Este trabalho está divido em 2 partes, a primeira parte consiste em um experimento de coleta dados para desenvolvimento de um rastreador ocular, ou seja, um dispositivo capaz de estimar o ponto sobre o monitor sendo observado pelo usuário. Na segunda parte, os dados extraídos da face serão utilizados para treinar uma rede neural de convolução usando a arquitetura *AFF-NET* [1], que servirá como modelo de estimação do olhar a ser utilizado no rastreador ocular.

**Palavras-chave:** rastreador-ocular, redes-neurais, estimador-do-olhar, convolução, CNN.

# Abstract

Matheus Santos Conceição. **Eye tracker based on Convolutional Neural Networks**. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2021.

The use of eye trackers, which infer the focus location of the eye on a screen from images of the eye itself is nothing new, however traditional eye trackers require certain specialized hardware, high-resolution or infrared cameras. This makes the data collection process more complicated, having the need to be performed in the laboratory.

Given the high number of electronic devices with cameras currently, this work proposes to explore the diffusion of cameras in electronic devices such as laptops, for the development of an eye tracker model of low cost.

This study is divided into 2 parts, the first part consists of an experiment for collecting data to develop the eye tracker, which is, a device capable of estimating the point on the monitor being observed by the user. In the second part, the data extracted from the face, will be used to train a convolutional neural network using an architecture *AFF-NET* [1] which will be the eye gaze estimator to be used in the eye tracker.

**Keywords:** eye-tracker, neural-network, gaze-estimation, convolution, CNN.

# Conteúdo

# Lista de Figuras

# Lista de Tabelas

# Capítulo 1

# Introdução

## 1.1 Contextualização

A linguagem corporal desempenha papel fundamental na comunicação humana. Em especial, o olhar tem um poder comunicativo poderoso, sendo uma das manifestações mais expressivas de comunicação não verbal. O contato visual contém uma rica variedade de informações a respeito da atenção e intenção humanas que vem ajudando pesquisadores da área da psicologia cognitiva a desvendar os mecanismos do comportamento e da cognição humanas [2]. O uso da aparência do olho como indicador do alvo de interesse visual tem ajudado muito no desenvolvimento de modelos de estimação do olhar. Que com isso vem ganhando espaço dentre os diversos outros modelos de desenvolvimento de rastreadores oculares.

Rastreamento ocular se refere ao processo de rastrear o movimento dos olhos, mais especificamente, o ponto absoluto do olhar, isto é, a localização do ponto em que a pessoa está focando na cena visual em um dado momento. Existem diversas técnicas de rastreamento ocular, algumas delas não são baseadas exclusivamente em imagens do olho. Isso torna o campo de estudo da visão, extremamente rico, cujas aplicações se espalham numa ampla área de atuações, que vai desde psicologia comportamental da interação humano-máquina até diagnósticos médicos na medicina de precisão [8].

O uso de rastreadores oculares *eye trackers*, que inferem a localização de foco do olho sobre a cena visual a partir de imagens do próprio olho, não é novidade, entretanto rastreadores oculares tradicionais exigem algum tipo de hardware especializado, câmeras de alta resolução ou infravermelhas, além de serem extremamente invasivos e desconfortáveis. Avanços tecnológicos na década de 90, iniciou o desenvolvimento e uso de aparelhos não invasivos baseados em fotografia da luz refletida pela córnea. O desenvolvimento de sistemas baseados em câmeras em conjunto com o aumento do poder computacional vem permitindo

a coleta de imagens do olho em tempo real, possibilitando o uso do olhar como método de controle para interfaces não convencionais, especialmente, dedicadas a usuários com algum tipo de deficiência que os impeçam de usar efetivamente interfaces de controle tradicionais, como por exemplo o *mouse* ou *touchscreen* de aparelhos eletrônicos.

Visto que a estimação do olhar envolve uma complexa relação entre o estímulo visual, a tarefa sendo realizada, o movimento dos olhos durante a realização da tarefa e o tempo de fixação [10], o processo de coleta de dados torna-se uma tarefa extremamente complicada. Por isso a necessidade de ser, geralmente, realizada em laboratório, onde se pode controlar variáveis ambientais, como a iluminação, movimento e posição da cabeça.

## 1.2 Objetivos

Dado o elevado número de dispositivos eletrônicos com câmeras atualmente, notebooks, celulares, etc, é natural querer explorar esses recursos, já amplamente difundidos, para facilitar a coleta e torná-la mais acessível. Portanto, o objetivo desse projeto é desenvolver um rastreador ocular *eye tracker* de baixo custo, que utiliza apenas a webcam comum e navegador com acesso à *internet*, sem exigir condições rígidas e especiais como posição do usuário em relação à tela e iluminação adequada. Idealmente passando por todos os estágio necessários desde a modelagem do experimento, passando pela coleta de dados até o treinamento de uma rede neural de convolução usando a arquitetura *AFF-NET* [1] para a estimação do olhar.

## 1.3 Organização do trabalho

O projeto ao todo consiste em 2 etapas (coleta de dados e modelagem do estimador). A primeira etapa do projeto é o desenvolvimento de uma aplicação de coleta, que consistirá em um jogo de navegador, cujo objetivo é mapear imagens do olho para posições em duas dimensões na tela, e assim fornecer os dados de entrada para o treinamento de uma rede neural que devolverá um estimador do olhar. A segunda etapa será o treinamento da rede neural de convolução que fornecerá o modelo do estimador do olhar.

Quanto à monografia, o **capítulo 2** apresenta o experimento de coleta com os desafios de projetá-lo considerando detalhes da anatomia do olho e seu padrão de movimento.

O **capítulo 3** apresenta conceitos importantes para o desenvolvimento e implementação do estimador do olhar, assim como o background de algumas técnicas de aprendizado de máquina utilizadas.

O **capítulo 4** apresenta os principais métodos na modelagem de estimadores do olhar, assim como apresenta o modelo que foi implementado. Nesse capítulo temos também os detalhes de implementação do modelo usado no rastreador ocular.

O **capítulo 5** apresenta o experimento de treinamento do modelo junto com os resultados obtidos. No último capítulo (**capítulo 6**) temos a conclusão do trabalho.

# Capítulo 2

# Experimento de Coleta de Dados

## 2.1 Desafios do experimento

No dia a dia os nossos olhos buscam constantemente informações do ambiente. A cada segundo estamos tomando decisões inconscientes sobre onde olhar, para garantir quais fótons irão estimular a nossa retina de maneira a sermos capazes de concluir nossas tarefas diárias.

Esse rico fluxo de dados visuais vem com o preço de um mecanismo de processamento em tempo real complexo e dispendioso. Processos cognitivos de alto nível tais como reconhecimento de objetos ou interpretação da cena visual, dependem de dados que foram transformados de forma a serem computacionalmente tratáveis pelo cérebro. Um dos principais mecanismos para tornar os dados visuais que recebemos mais fáceis de serem processados está relacionado à atenção visual, cujo cerne está na ideia de um mecanismo prático de seleção e uma noção de relevância, prioridade. Em seres humanos, a atenção é facilitada por uma retina que desenvolveu uma alta resolução da fóvea central e uma periferia de baixa resolução. Enquanto a atenção visual guia esta estrutura anatômica para as partes importantes da cena de maneira a reunir informações mais detalhadas do ambiente[2].

Alguns desses movimentos oculares controlados pelo mecanismo da atenção visual podem ser decisivos em questões básicas como navegar pelo ambiente, ou procurar um objeto.

Assim a questão principal para o desenvolvimento desse experimento é como garantir que alguém esteja olhando para onde queremos que ela olhe, de maneira precisa? Logo desenvolver o experimento para captar o olhar dos usuários foi em essência tentar responder a pergunta acima. A primeira coisa que fizemos foi reduzir o nível de incerteza das posições de fixação do olhar, reduzindo a tela do computador a um conjunto de alvos posicionados equidistantes uns dos outros de forma que nosso problema de onde a pessoa está olhando na tela se reduzisse a qual dos alvos da tela a pessoa está olhando. Assim podemos explorar os

tipos de movimentos dos olhos e o conhecimento que temos deles sobre sua contribuição na visão para coletar imagens dos olhos nos momentos certos, nos momentos de fixação.

Os dados que o experimento pretende coletar são a posição do alvo em exibição atualmente e a imagem da pessoa com o olhar fixo no alvo (olhar de vergência). Como os alvos são apresentados em sequência era preciso estabelecer um período em que tivéssemos alguma confiança de que a pessoa estaria de fato olhando para o alvo, seja acompanhando o estímulo em movimento ou após a fixação no alvo. Mas o experimento não tem como garantir isso, pois é realizado individualmente pelos participantes com seus próprios computadores sem a supervisão dos pesquisadores. A melhor coisa que se podia então fazer era criar algum tipo de estímulo que induzisse o participante à olhar para o alvo.

Enfim, o experimento desenvolvido consiste em uma tarefa simples: mover com o *mouse* um objeto para dentro de um alvo exibido. No entanto, isso ainda não garante a fixação do olhar após conclusão da tarefa. Por isso fazemos o alvo piscar durante alguns milissegundos para mobilizar a atenção visual da pessoa a focalizar uma determinada posição da tela, para que tenhamos as imagens da pessoa olhando para o alvo durante esse intervalo.

## 2.2 Tipos de movimentos dos olhos

Um dos principais tipos de movimento dos olhos envolvido no processamento de informações é conhecido como olhar de sacada, ou olhar sacádico. Esses movimentos sacádicos são os deslocamentos, com mudanças de direções abruptas, que os olhos realizam, a cada segundo para a realização de uma tarefa onde seja necessária o controle ocular fino. Como esses movimentos são extremamente rápidos chegando a 500 graus por segundo, a sensibilidade visual durante os movimentos fica bastante reduzida, o que é chamado de supressão sacádica.

Assim, cada sacada é intercalada por períodos de fixação de aproximadamente $200 - 300$ms [10], e são esses próprios movimentos sacádicos que ligam todas as fixações oculares entre si, possibilitando a execução de tarefas complexas que requerem uma enorme carga de processamento visual em tempo real, como a leitura, escrita ou a busca de objetos no ambiente. Caso contrário perceberíamos apenas um borrão, já que novas informações não são adquiridas durante os movimentos.

Cada tarefa entretanto difere no tempo mínimo de fixação, a depender da complexidade da própria tarefa.

A velocidade de sacada é uma função monotônica com respeito a distância percorrida pelo olho. Ela aumenta rapidamente no início do movimento atingindo um máximo um

| Tarefa | Média de fixação (ms) | Média da largura do olhar de sacada (graus) |
|---|---|---|
| Leitura | 225 | 2 |
| Leitura em voz alta | 275 | 1.5 |
| Busca visual | 275 | 3 |
| Percepção da cena | 330 | 4 |
| Leitura musical | 375 | 1 |
| Digitação | 400 | 1 |

**Tabela 2.1:** *Tabela com tempo mínimo de fixação para cada tipo de tarefa [10]*

pouco antes do ponto médio entre a ponto de partida e o ponto chegada (alvo) e diminui gradualmente até atingir a localização do alvo.

Existem outros tipos de olhares além dos sacádicos. O movimento de busca suave (*Pursuit eye movements*) descreve um tipo de movimento ocular em que os olhos permanecem fixo num objeto em movimento, sua velocidade é menor em relação ao movimento sacádico, mas se o objeto estiver se movendo muito rápido temos uma combinação de movimentos sacádicos e de busca suave. Esse tipo de movimento é voluntário já que o observador escolhe ou não rastrear o estímulo em movimento.

A vergência é um outro tipo de movimento voluntário dos olhos. É observado quando ocorre movimento simultâneo de ambos os olhos em direções opostas para obter ou manter uma visão binocular única de maneira a alinhar a fóvea de cada olho com os alvos localizados a distâncias diferentes do observador.

Ao contrário de outros tipos de movimentos oculares nos quais os dois olhos se movem na mesma direção (movimentos oculares conjugados), os movimentos de vergência são desconjugados (ou disjuntivos); eles envolvem uma convergência ou divergência das linhas de visão de cada olho para ver um objeto que está mais próximo ou mais distante. A convergência é uma das três respostas visuais reflexivas eliciadas pelo interesse em um objeto próximo. Os outros componentes da chamada tríade de reflexo próximo são a acomodação da lente, que focaliza o objeto, e a constrição pupilar, que aumenta a profundidade de campo e torna a imagem mais nítida na retina [9].

Então a estratégia para o experimento se torna a seguinte. Dois objetos são apresentados na tela. Inicialmente temos os movimentos sacádicos que rastreiam a cena para encontrar a posição dos objetos e determinar o pontos de partida (a bola) e de chegada (alvo). Feito isso, enquanto o participante estiver movendo a bola do ponto de partida até o alvo, ocorre o movimento de busca suave em conjunto com os movimentos sacádicos que conferem constantemente a posição de chegada mesmo que esta seja fixa. Ao atingir a localização do alvo com a bola temos o movimento de vergência em que o participante irá focalizar o ponto do alvo com os dois olhos por alguns milissegundos em quanto este piscará para evitar que movimentos sacádicos possam causar distração e diminuir a confiabilidade dos dados, durante

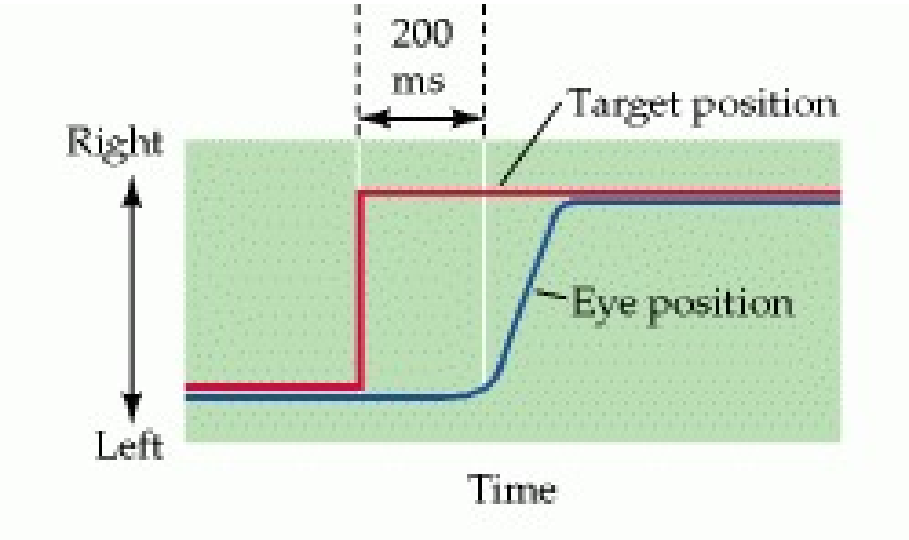


**Figura 2.1:** *Métricas do movimento sacádico, a linha vermelha indica a posição de fixação do alvo a linha azul a posição da fóvea*

esse intervalo são capturadas as imagens dos olhos que serão utilizadas no treinamento do estimador do olhar.

## 2.3 Descrição do experimento

O experimento, em princípio, é bem simples, e foi pensado para exigir o mínimo de esforço do usuário para completá-lo. Assim, o experimento consiste em um jogo de clicar e arrastar bolas para dentro de caixas com o mouse, enquanto a face do usuário é gravada pela sua webcam.

Durante o experimento será coletado o vídeo da face do usuário e a posição das bolas ao serem inseridas dentro dos alvos. Os alvos possuem posições relativas a tela pré estabelecidas, mas a sequência com que os alvos aparecerão é determinada aleatoriamente.

No total, são 35 bolas e 35 alvos exibidos na tela em sequência. Quando o usuário arrastar a bola para dentro do alvo, é pedido para que a pessoa fixe o olhar na bola que irá piscar por 1500 milissegundos, essa é a parte em que a face do usuário é utilizada para estimar a posição do olhar na tela, os frames desse intervalo de tempo serão mapeados para a posição normalizada do alvo na tela. A bola seguinte sempre aparecerá na posição do alvo anterior, e então o próximo alvo da sequência é exibido, e assim por diante.

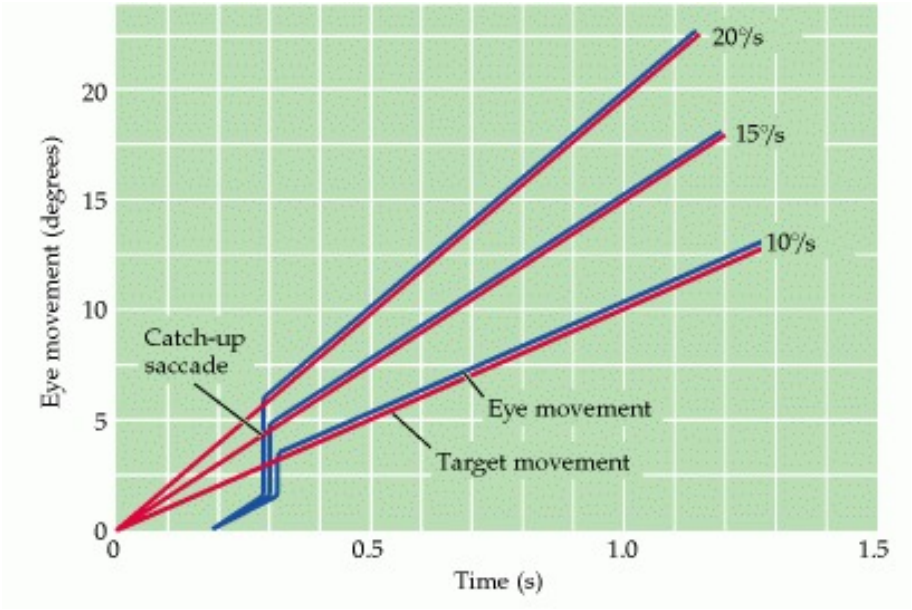


**Figura 2.2:** *Métricas do movimento de busca suave, as linhas azuis mostram o movimento dos olhos acompanhando um estímulo em movimento, as linhas vermelhas mostram a velocidade do movimento do estímulo*

## 2.4 Resultados do experimento

Participaram do experimento 28 pessoas resultando em um conjunto de dados de 15,151 imagens. Para cada pessoa foram selecionados aleatoriamente até 15 *frames* no intervalo em que o alvo estava sendo focalizado, isso foi realizado para os 35 alvos. As imagens foram pré-processadas com descarte de imagens em que o rosto ou os olhos não foram reconhecidos pelo modelo de *face mesh* da biblioteca *mediapipe*. Usando este modelo, para cada *frame* coletado foram produzidas 3 imagens: um recorte de cada olho e um recorte da face. As 3 imagens mais as coordenadas do retângulo das bordas de cada imagem foram usadas para alimentar a rede neural de convolução.

O conjunto de dados resultante do experimento foi construído de maneira similar ao conjunto de dados usado pelos autores do artigo [1], seguindo as mesmas orientações descritas no artigo. Ou seja, o conjunto de dados usado no artigo poderia ser fácilmente utilizado no modelo implementado por este trabalho.

Assim, o conjunto de dados utilizado no treinamento consiste em 3 imagens coloridas RGB mais os retângulos das bordas de cada imagem. As imagens dos olhos foram redimensionadas para $(112 * 112 * 3)$, e a da face para $(224 * 224 * 3)$, enquanto os retângulos das bordas mantêm as informações de posição na imagem original.

A única diferença no conjunto de dados usado neste trabalho e o usado no artigo está na variedade de tamanhos das telas, já que no artigo os dados foram coletados em *iphones* e *ipads*, e não houve mistura entre conjunto de dados com dispositivos de tamanhos de telas

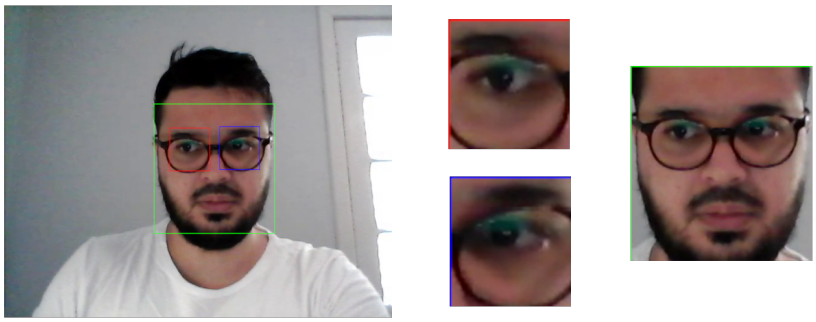

**Figura 2.3:** *Exemplo de um input para a rede neural*

diferentes, enquanto que neste trabalho foram utilizados *desktops* na coleta, que possuem uma variedade de tamanho de telas muito maior.


**Figura 2.4:** *Telas do experimento*

# Capítulo 3

# Conceitos

## 3.1 Redes neurais tradicionais

### 3.1.1 Introdução

Uma rede neural é um modelo computacional inspirado nas redes biológicas do sistema nervoso central, principalmente do cérebro, que é a melhor máquina que se conhece quando o assunto é o reconhecimento de padrões. Numa rede neural temos um conjunto de dados de entrada e um conjunto de dados de saída e, para treinar a rede, vamos alimentando-na com dados e ajustando seus parâmetros de acordo com a diferença entre o resultado produzido e o resultado esperado.

Para essa diferença atribuímos um custo que reflete o quão próximo o resultado obtido se aproxima do esperado. Assim podemos imaginar o aprendizado da rede neural como um processo de minimização desta função de erro. É nessa parte do treinamento que ocorre o aprendizado, usando técnicas de otimização, como retropropagação, para tentar minimizar a função de erro.

### 3.1.2 Retropropagação (*Backpropagation*)

A técnica de retropropagação consiste em comparar valores de saída com a resposta correta para computar uma função de erro; esse erro é alimentado na rede em caminho inverso de maneira a ir ajustando os parâmetros de cada camada da rede até chegar no início. Quando novos parâmetros forem encontrados, repete-se a primeira fase que é a da alimentação da rede com os dados de entrada, mas agora a rede tem novos parâmetros o que resultará em novos valores de saída, estes agora serão comparados com a resposta correta, e

assim por diante, até que os valores de saída estejam suficientemente próximos dos valores corretos, dado um erro admissível.

### 3.1.3 Não linearidade

A camada de não linearidade, tem como objetivo introduzir um componente não linear à rede neural, para que a rede seja capaz de aproximar funções mais complexas. Uma das operações mais utilizadas nessa camada, a função de ativação ReLU (*Rectified Linear Unit*) substitui valores negativos por zero ReLU(x) = max(0,x). Também podemos usar a função de ativação *LeakyReLU* que é capaz de reter alguns valores negativos dependendo do nível de ativação, o que torna o modelo que a utiliza um pouco mais flexível. E por último temos a função de ativação sigmoide que devolve valores entre 0 e 1.

### 3.1.4 Otimizador Adam

O algoritmo de otimização Adam é uma extensão do algoritmo gradiente descendente estocástico que vem sendo amplamente utilizado em aplicações de computação visual. Diferentemente do algoritmo gradiente descendente clássico, o algoritmo Adam atualiza os parâmetros da rede iterativamente com relação aos dados de treinamento com a taxa de aprendizado sendo adaptada com o decorrer do treinamento. Os resultados mostram que o algoritmo Adam converge mais rapidamente e alcança valores menores para a função de perda [3].

### 3.1.5 Validação cruzada

Validação cruzada é uma técnica para avaliar modelos de aprendizado de máquina por meio do treinamento de vários modelos em subconjuntos de dados de entrada disponíveis e avaliação deles no subconjunto complementar dos dados [5]. A validação cruzada é usada para detectar sobreajuste, ou seja, a não generalização de um padrão, principalmente, em conjuntos de dados pequenos, além de obter uma estimativa de erro mais confiável.

Na validação cruzada k-fold, divide-se os dados de entrada em k subconjuntos. Então treina-se o modelo em todos os subconjuntos, menos em um (k-1) dos conjuntos de dados e, em seguida, avalia-se o modelo no conjunto de dados que não foi usado para o treinamento. Esse processo é repetido k vezes, com um subconjunto diferente reservado para avaliação (e excluído do treinamento) a cada vez.

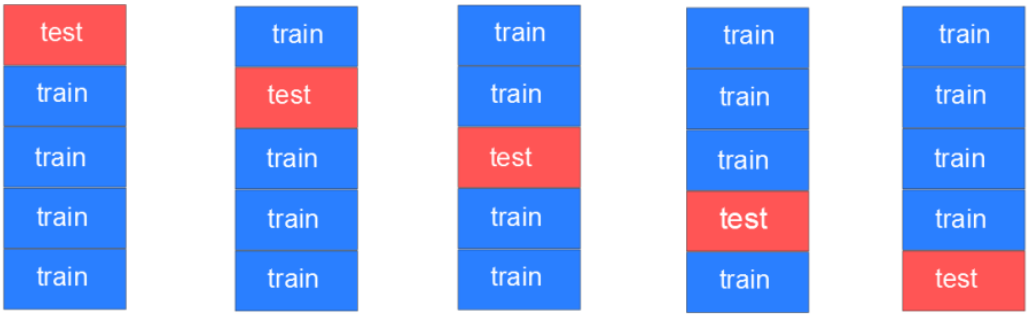


**Figura 3.1:** *Exemplo de validação cruzada usando 5-fold*

## 3.2 Redes neurais de convolução

### 3.2.1 Introdução

O uso de camadas lineares (*linear layer*) vem sendo progressivamente substituído por camadas de convolução (*Convolutional Layer*) em redes neurais no campo de processamento de imagens, tanto para tarefas de classificação como de regressão. As redes neurais de convolução têm sua inspiração no funcionamento do córtex visual humano, especialmente na sua maneira particular de ativação de diferentes áreas de acordo com os padrões visuais a que está sendo exposto.

### 3.2.2 Definição de convolução

Convolução é uma operação de grande importância na matemática aplicada tendo especial destaque na área de processamento de sinais. A ideia por trás da convolução no contexto do processamento de imagens está relacionada aos núcleos de convolução (*kernels*). Esses núcleos são matrizes que atuam como filtros que aplicam algum efeito à imagem.

Cada imagem são coleções de pixeis que são representados por um ou mais valores numéricos a depender do número de canais ou da profundidade da imagem, por exemplo em uma imagem em escala de cinzas, um pixel é representado por um valor numérico, enquanto em uma imagem colorida é representado por três ou quatro valores numéricos. A convolução de uma imagem dado um núcleo de dimensões especificadas (em geral $3x3$) é obtida colocando se o núcleo sobreposto à imagem, alinhando-se o valor central do núcleo à primeira posição do pixel da imagem. Assim, deslizando a área do núcleo para cada pixel da imagem e calculando para cada pixel a soma dos pixeis vizinhos que estão também sobrepostos pelo núcleo, ponderada pelos valores do núcleo.

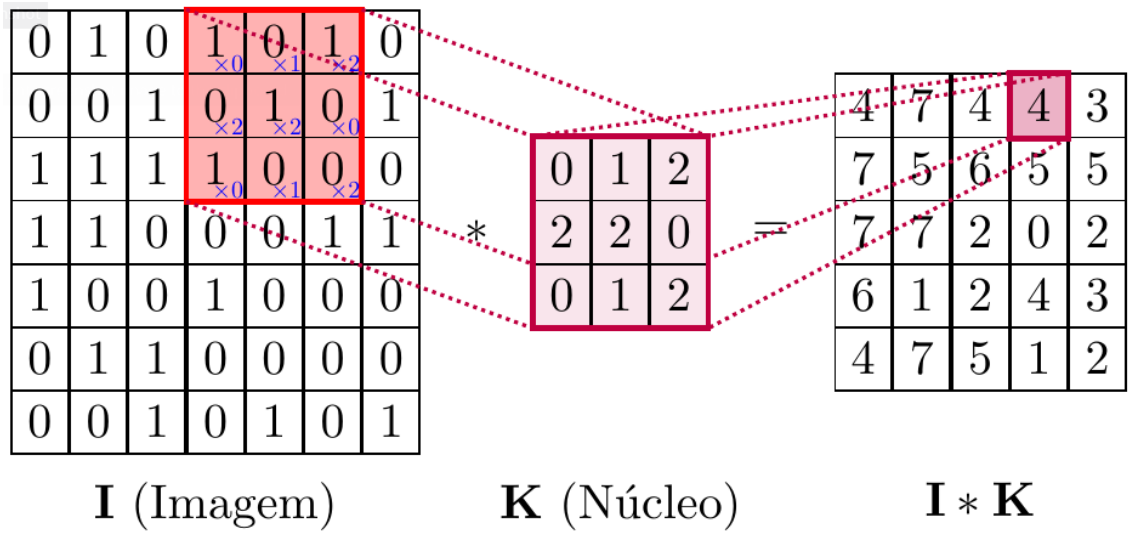


**Figura 3.2:** *Convolução de uma imagem binária (7 x 7) por um núcleo (3 x 3)*

### 3.2.3 Convolução em três dimensões

No contexto das *CNN*s geralmente se usa convoluções tri-dimensionais com altura e largura representando as dimensões da imagem e a profundidade o número de canais necessários para representar a cor do pixel, por exemplo três (RGB). Em geral em uma camada de convolução não se aplica apenas um filtro, mas vários que resultam em mapas de características (*feature maps*) distintos , o que aumenta o número de canais do resultado da convolução. Os mapas de características são empilhados e reunidos para formar a saída da camada com dimensões proporcionais ao número de filtros aplicados e suas dimensões originais. É comum aumentar o número de canais enquanto se diminui as dimensões (altura x largura) do bloco, o que resulta em um ganho computacional.

### 3.2.4 Agrupamento

Depois da operação de convolução se executa alguma operação de agrupamento (*pooling*) que mais uma vez reduz as dimensões do resultado da camada de convolução, sem alterar o número de canais. Camadas de agrupamento (*Pooling Layer*) diminuem a altura e largura dos mapas de características mas deixam o número de canais inalterados. O tipo mais comum é o agrupamento de valor máximo (*max pooling*), que mantém o valor máximo dentro de uma janela de agrupamento.

## 3.3 Camadas de compressão e excitação

Outra técnica que vem sendo usada em *CNN*s em conjunto com as camadas de convolução, são as camadas de compressão e excitação *SE* (*Squeeze and Excitation layer*), essa técnica é usada para melhorar a interdependência entre canais dos blocos de convolução, ou seja, melhorar a interdependência entre os mapas de características resultantes da operação de convolução, de maneira a ajustar adaptativamente os pesos de cada mapa de características. [6].

Os filtros de convolução extraem informações hierarquizadas das imagens, de forma que camadas inferiores extraem informações mais simples como bordas e outras informações espaciais, enquanto que camadas superiores, ou seja, filtros aplicados depois no bloco de convolução, extraem padrões visuais mais complexos como formas geométricas, que são codificados ao longo dos canais do bloco. Geralmente, a rede neural atribuiria a cada canal um mesmo peso, mas o diferencial da camada *SE* é adicionar parâmetros a cada canal, com o intuito de ajustar quais filtros, ou quais mapas de características desempenham um papel mais relevante para determinada tarefa.

$$\begin{cases} W_{weight} = \sigma(L(GAP(f_{in}))), & (3.1) \\ f_{out} = F_{scale}(W_{weight}, f_{in}) & (3.2) \end{cases}$$

$f_{in}$ refere-se ao bloco de entrada da *SE layer*. $GAP()$ refere-se *Global Average Pooling layer*. $L()$ refere-se a camada linear e $\sigma(.)$ refere-se a função de ativação sigmoide. $Fscale()$ refere-se a multiplicação canal a canal do bloco de entrada. $W_{weight}$ são os pesos aplicados ao bloco de entrada com resultado denotado por $f_{out}$.

A camada *SE* funciona da seguinte maneira: Primeiro a camada recebe como *input* um bloco de convolução, o número de canais e uma razão. Em seguida tira-se a média dos valores de cada canal individualmente com uma camada de agrupamento (*pooling*), para que cada canal do bloco seja reduzido a um único valor numérico. Então usamos uma camada linear com função de ativação $ReLU$ e reduzimos o número de canais dividindo-no pela razão informada. Adiciona-se à saída uma outra camada linear mas dessa vez com uma função de ativação sigmoide que suaviza os valores entre 0 e 1. Por último multiplica-se os pesos obtidos pelo bloco de convolução inicial.

Vamos definir as camadas que serão usadas na implementação da *SE Layer*. Primeiro definimos uma camada de agrupamento global médio *Global Avarege Pooling layer (GAP)*

```
GAP = nn.AdaptiveAvgPool2d(1)
```

Depois definimos uma camada linear totalmente conectada com função de ativação *ReLU*.

```
L = nn.Sequential(
    nn.Linear(ch, ch // ratio, bias=True),
    nn.ReLU(),
    nn.Linear(ch // ratio, ch, bias=True),
)
```

E por fim uma camada com função de ativação sigmoide.

```
sigmoid = nn.Sigmoid()
```

Agora implementamos as equações (3.1) e (3.2) definidas no artigo [1].

```
batch_size, num_channels, _, _ = f_in.shape
W = sigmoid(L(GAP(f_in).view(f_in.size(0), -1)))
f_out = torch.mul(W.view(batch_size, num_channels, 1, 1), f_in)
```

**Trecho do código para da implementação da camada SE**

```
class SELayer(nn.Module):
    def __init__(self, ch, ratio):
        super(SELayer, self).__init__()
        # Global Average Pooling layer
        self.gap = nn.AdaptiveAvgPool2d(1)
        # Fully Conected layer L(.)
        self.L = nn.Sequential(
            nn.Linear(ch, ch // ratio, bias=True),
            nn.ReLU(),
            nn.Linear(ch // ratio, ch, bias=True),
        )
        # Sigmoid function
        self.sigmoid = nn.Sigmoid()

    def forward(self, f_in):
        batch_size, num_channels, _, _ = f_in.shape
        # Formula from the paper equation (1)
        W = self.sigmoid(self.L(self.gap(f_in).view(f_in.size(0), -1)))
        return torch.mul(W.view(batch_size, num_channels, 1, 1), f_in)
```

## 3.4 Normalização Adaptativa em Grupo

Uma maneira de combinar as diferentes características (*features*) faciais para os dois olhos, de maneira a dependerem menos de informações ambientais como orientação e ilumi-

nação, é com o uso de uma normalização adaptativa em grupo (*adaptive group Normalization AdaGN*) [1]. A camada AdaGN recebe características faciais assim como os retângulos que compõem as bordas das regiões da face e dos olhos na imagem original. E calcula a escala e o deslocamento para recalibrar a extração de características dos olhos.

$$\begin{cases} [W_{shift}, W_{scale}] = LeakyReLU(L(f_{rects}, f_{face})), & (3.3) \\ f_{out} = W_{scale} * GN(f_{in}) + W_{shift} & (3.4) \end{cases}$$

Onde L(·) refere-se a camada linear, GN (·) refere-se a *normal Group Normalization*. $f_{in}$ refere-se ao mapa de características original, $f_{rects}$ representa as características extraídas dos retângulos das imagens que contém os olhos e a face, enquanto $f_{face}$ representa as características extraídas diretamente da imagem da face. $W_{scale}$ e $W_{shift}$ são os parâmetros de escala e deslocamento para cada canal de $f_{in}$. E $f_{out}$ é o resultado da camada *AdaGN*, após aplicação dos parâmetros de escala e deslocamento no mapa de características original normalizado.

**Trecho de código da implementação da camada *AdaGN***

```
class AdaGN(nn.Module):
    def __init__(self, sz, ch):
        super(AdaGN, self).__init__()
        self.FC = nn.Linear(sz, 2 * ch)
        self.LeakyRelU = nn.LeakyReLU()

    def forward(self, f_in, block, ada_in):
        batch_size, num_channels, _, _ = f_in.shape
        # Scale and shift parameters for each channel as for equation (2)
            from the paper
        shape = self.LeakyRelU(self.FC(ada_in))
        shape = shape.view([batch_size, 2, num_channels, 1, 1])
        scale = shape[:, 0, :, :, :]
        shift = shape[:, 1, :, :, :]
        f_gn = f_in.view(batch_size * block, -1)
        # Get mean and std of the batch
        std, mean = torch.std_mean(input=f_gn, dim=1, keepdim=True)
        # Normalize
        f_gn = (f_gn - mean) / (std + 1e-8)
        # Back to the original shape but with normal Group Normalization
        f_gn = f_gn.view(f_in.shape)
        f_out = (scale) * f_gn + scale
        return f_out
```

# Capítulo 4

# Estimação do olhar usando uma rede neural AFF

## 4.1 Introdução

Existe uma grande quantidade de pesquisas que propõem soluções para a tarefa de estimação do olhar, essas soluções entretanto podem ser divididas, grosso modo, em duas categorias: métodos baseados em modelos dos olhos e métodos baseados na aparência dos olhos.

## 4.2 Métodos baseados em modelo

Essa categoria utiliza modelos da geometria dos olhos para inferir a direção do olhar. A partir do formato dos olhos, considerando a localização da pupila e as bordas da íris, se cria um modelo capaz de regredir essas informações em uma direção do olhar. A desvantagem desses métodos está na necessidade de imagens de alta qualidade, tendendo a sofrer com imagens de baixas resoluções encontradas nas *webcams* comuns. Métodos baseados em modelo são mais adequados para experimentos realizados em laboratório em que se pode utilizar equipamentos de captura de melhor resolução e onde se pode controlar melhor os fatores ambientais como iluminação e posição da câmera.

## 4.3 Métodos baseados em aparência

Métodos baseados na aparência dos olhos estimam o olhar mapeando diretamente imagens dos olhos para uma estimação (seja posição de foco na cena ou a direção do olhar). Esses métodos requerem configurações de captura mais modestos, geralmente, utiliza-se uma única câmera sem a necessidade do experimento ser supervisionado, ou ter regras de controle ambiental muito rígidas. A tarefa de mapeamento pode utilizar vários métodos diferentes de regressão como, redes neurais, interpolação local, regressão gaussiana. No entanto, métodos baseados em redes neurais de convolução já demostraram bons resultados em tarefas de regressão, com precisões maiores do que outros tipos de regressões para métodos baseados na aparência do olho.

## 4.4 Método utilizado (*Adaptive feature fusion network)*

### 4.4.1 Visão geral

A *AFF-Net* é a arquitetura da rede neural que foi utilizada para treinar os dados coletados e inferir a posição do olhar [1], a implementação foi feita inteiramente do zero em *pytorch*, seguindo as equações apresentadas no artigo assim como as orientações e detalhes de implementação támbem fornecidos em Bao et al [1]. Sua estrutura é mostrada na Figura 4.1. Essa arquitetura recebe como entrada as imagens da face, do olho esquerdo, do olho direito e os retângulos que compõem as bordas dessas regiões. O objetivo é aproveitar a similaridade dos dois olhos para extrair características complexas de maneira adaptativa. Mapas de características dos dois olhos são empilhados em um bloco único, depois de terem passado por uma série de camadas de convolução. As camadas *SE* são usadas para ponderar quais mapas de características são mais importantes e os retângulos com as bordas da região serão usados para recalibrar a extração das características dos olhos, o que se dá na camada *AdaGN*.

### 4.4.2 Implementação

O modelo da rede $AFF$ foi inteiramente implementado usando a biblioteca pytorch, seguindo as fórmulas e detalhes apresentadas em Bao et al [1].

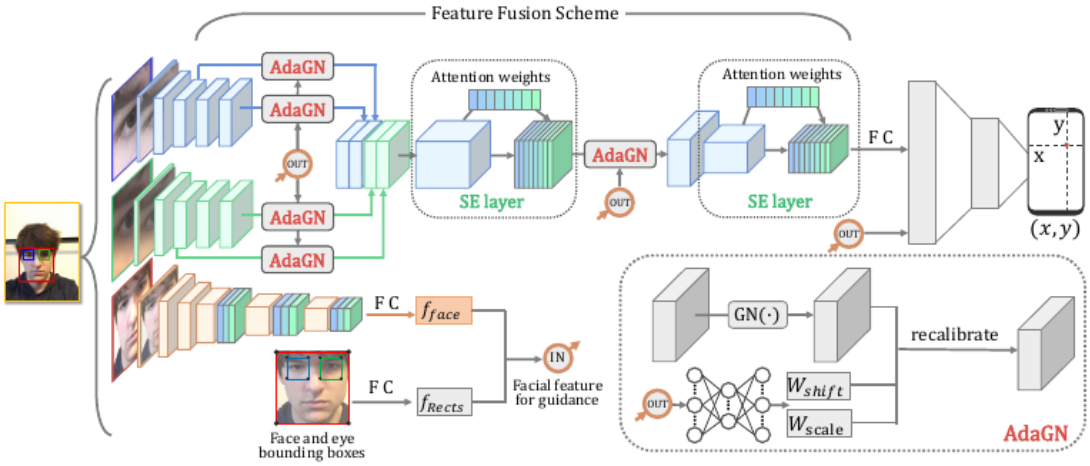


**Figura 4.1:** *Arquitetura do modelo AFF-NET, imagem retirada de Bao et al [1]*

### Modelo da Face

O modelo da face possui 6 camadas de convolução. Com uma camada *max pooling* depois da segunda e quinta camada de convolução. Duas camadas lineares são usadas para comprimir as características faciais em um vetor de 64 dimensões. Os retângulos são processados por 4 camadas lineares com canais de saída respectivamente (64, 96, 128, 64).

```
class FaceModel(nn.Module):
    def __init__(self):
        super(FaceModel, self).__init__()
        self.conv_block = nn.Sequential(
            nn.Conv2d(3, 48, kernel_size=5, stride=2, padding=0),
            nn.GroupNorm(6, 48),
            nn.ReLU(),
            nn.Conv2d(48, 96, kernel_size=5, stride=1, padding=0),
            nn.GroupNorm(12, 96),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(96, 128, kernel_size=5, stride=1, padding=2),
            nn.GroupNorm(16, 128),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(128, 192, kernel_size=3, stride=1, padding=1),
            nn.GroupNorm(16, 192),
            nn.ReLU(),
            SELayer(192, 16),
            nn.Conv2d(192, 128, kernel_size=3, stride=2, padding=0),
            nn.GroupNorm(16, 128),
            nn.ReLU(),
            SELayer(128, 16),
            nn.Conv2d(128, 64, kernel_size=3, stride=2, padding=0),
```

```
            nn.GroupNorm(8, 64),
            nn.ReLU(),
            SELayer(64, 16),
        )

        self.linear1 = nn.Linear(5 * 5 * 64, 128)
        self.linear2 = nn.Linear(128, 64)
        self.lrelu = nn.LeakyReLU()
```

Depois de definido o bloco de convolução acima, o aplicamos nos dados de entrada $x$ que são imagens recortadas da face. Em seguida aplicamos duas vezes uma camada linear com função de ativação *LeakyReLU*.

```
    def forward(self, x):
        x = self.conv_block(x).view(x.size(0), -1)
        x = self.linear1(x)
        x = self.lrelu(x)
        x = self.linear2(x)
        return self.lrelu(x)
```

### Modelo do Olho

O modelo do olho, que é o mesmo para o olho esquerdo quanto o direito, possui 5 camadas de convolução, com camadas *max pooling* na segunda e na quinta camada. Os mapas de características resultantes são fundidos pela camada *AdaGN* depois da terceira e da quinta camadas de convolução. Temos uma camada linear que comprime o mapa de características dos dois olhos em um único vetor de 128 dimensões.

```
class EyeModel(nn.Module):
    def __init__(self):
        super(EyeModel, self).__init__()

        self.pool = nn.MaxPool2d(kernel_size=3, stride=2)
        self.relu = nn.ReLU()
        self.grup = nn.GroupNorm(3, 24)
        self.se1 = SELayer(48, 16)
        self.se2 = SELayer(128, 16)
        self.conv1 = nn.Conv2d(3, 24, kernel_size=5, stride=2, padding=0)
        self.conv2 = nn.Conv2d(24, 48, kernel_size=5, stride=1, padding=5)
        self.conv3 = nn.Conv2d(48, 64, kernel_size=5, stride=1, padding=1)
```

```
        self.conv4 = nn.Conv2d(64, 128, kernel_size=3, stride=1, padding
            =1)
        self.conv5 = nn.Conv2d(128, 64, kernel_size=3, stride=1, padding
            =1)
        self.agn1 = AdaGN(128, 48)
        self.agn2 = AdaGN(128, 64)
        self.agn3 = AdaGN(128, 128)
        self.agn4 = AdaGN(128, 64)
```

Segue a aplicação das camadas definidas acima para o modelo do olho. $ada_{in}$ refere-se ao fator de recalibragem extraído dos retângulos das bordas das regiões dos olhos e face na imagem original, resultante da camada *AdaGN*.

```
    def forward(self, x, ada_in):

        x1 = self.conv1(x)
        x1 = self.grup(x1)
        x1 = self.relu(x1)
        x1 = self.conv2(x1)
        x1 = self.relu(x1)
        x1 = self.agn1(x1, 6, ada_in)
        x1 = self.pool(x1)
        x1 = self.se1(x1)
        x1 = self.conv3(x1)
        x1 = self.agn2(x1, 8, ada_in)
        x1 = self.relu(x1)
        x1 = self.pool(x1)

        x2 = self.conv4(x1)
        x2 = self.agn3(x2, 16, ada_in)
        x2 = self.relu(x2)
        x2 = self.se2(x2)
        x2 = self.conv5(x2)
        x2 = self.agn4(x2, 8, ada_in)
        x2 = self.relu(x2)

        return torch.cat((x1, x2), 1)
```

Os resultados finais de cada modelo para face, olhos e retângulos são concatenados e alimentados como entrada de duas camadas lineares que devolvem uma coordenada bidimensional correspondendo à posição normalizada do olhar na tela.

```
def forward(self, eyesLeft, eyesRight, faces, rects):

    # Calibration step
    outFace = self.faceModel(faces)
    outRect = self.rectsFC(rects)
    ada_in = torch.cat((outFace, outRect), 1)

    # Apply the eye model to the left eye
    outEyeL = self.eyeModel(eyesLeft, ada_in)
    # Apply the eye model to the left eye
    outEyeR = self.eyeModel(eyesRight, ada_in)
    # Concat both eyes
    outEyes = torch.cat((outEyeL, outEyeR), 1)

    # Apply SELayer
    outEyes = self.se1(outEyes)
```

```
        # Apply a convolutional layer
        outEyes = self.conv(outEyes)
        # Apply AdaGN
        outEyes = self.agn(outEyes, 8, ada_in)
        outEyes = self.relu(outEyes)
        outEyes = self.se2(outEyes)
        outEyes = outEyes.view(outEyes.size(0), -1)
        # Apply Fully Conected layer
        outEyes = self.eyesFC(outEyes)

        # Apply eyes, face and rects features
        f_out = torch.cat((outEyes, outFace, outRect), 1)
        return self.FCLayer(f_out)
```

### Modelo das bordas das regiões dos olhos e face

Nesse modelo aplicamos três vezes uma camada linear com função de ativação *Leaky-ReLU*, para produzir um *output* de dimensões 128 x 64;

```
self.rectsFC = nn.Sequential(
    nn.Linear(12, 64),
    nn.LeakyReLU(),
    nn.Linear(64, 96),
    nn.LeakyReLU(),
    nn.Linear(96, 128),
    nn.LeakyReLU(),
    nn.Linear(128, 64),
    nn.LeakyReLU(),
)
```

### Recalibragem

Nessa parte pegamos o resultado do modelo da face e o resultado do modelo das bordas das regiões dos olhos e face, e concatenados seus *outputs*, para ser usado como fator de recalibragem na extração de características no modelo do olho.

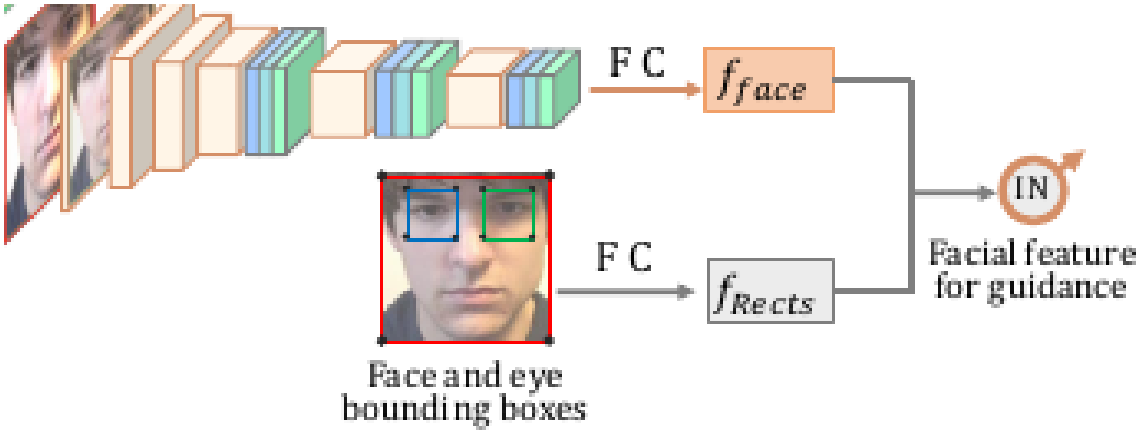


**Figura 4.2:** *Imagem retirada de Bao et al [1]*

```
outFace = self.faceModel(faces)
outRect = self.rectsFC(rects)
ada_in = torch.cat((outFace, outRect), 1)
```

### Modelo da rede AFF

Combinando os modelos definidos acima temos o seguinte esquema

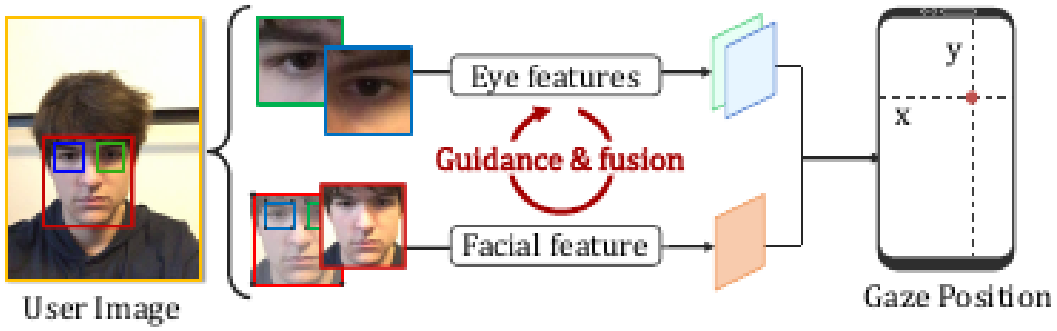


**Figura 4.3:** *Imagem retirada de Bao et al [1]*

Trecho de código do modelo completo da rede AFF, utilizando os modelos definidos acima.

```
class Model(nn.Module):
    ''' Implement the main structure for the Adaptive Feature Fusion
    network.
    '''
```

```
    def __init__(self):
        super(Model, self).__init__()

        # Eyes and face model
        self.eyeModel = EyeModel()
        self.faceModel = FaceModel()

        self.conv = nn.Conv2d(256, 64, kernel_size=3, stride=2, padding=1)
        self.se1 =SELayer(256, 16)
        self.se2 = SELayer(64, 16)
        self.relu = nn.ReLU()
        self.agn = AdaGN(128, 64)

        # Fully conected layers for eyes and face and rects
        self.FCLayer = nn.Sequential(
            nn.Linear(128 + 64 + 64, 128),
            nn.LeakyReLU(),
            nn.Linear(128, 2),
        )
        self.eyesFC = nn.Sequential(
            nn.Linear(3136, 128),
            nn.LeakyReLU(),
        )
        self.rectsFC = nn.Sequential(
            nn.Linear(12, 64),
            nn.LeakyReLU(),
            nn.Linear(64, 96),
            nn.LeakyReLU(),
            nn.Linear(96, 128),
            nn.LeakyReLU(),
            nn.Linear(128, 64),
            nn.LeakyReLU(),
        )

        self.loss_fn = nn.SmoothL1Loss().cuda()

    def training_step(self, data, device):
        output = self(data['leftEye'].to(device),
                      data['rightEye'].to(device),
                      data['face'].to(device),
                      data['rects'].to(device))

        loss = self.loss_fn(output, data['label'].to(device)) * 4
        return loss

    def validation_step(self, data, device):
        output = self(data['leftEye'].to(device),
                      data['rightEye'].to(device),
```

```
                    data['face'].to(device),
                    data['rects'].to(device))

        # Labels are the score which are the normalized position on screen
        labels = data['label'].to(device)
        # Resolution of the screen
        resolution = data['resolution'].to(device)
        return output, labels, resolution

    def forward(self, eyesLeft, eyesRight, faces, rects):

        # Calibration step
        outFace = self.faceModel(faces)
        outRect = self.rectsFC(rects)
        ada_in = torch.cat((outFace, outRect), 1)

        # Apply the eye model to the left eye
        outEyeL = self.eyeModel(eyesLeft, ada_in)
        # Apply the eye model to the left eye
        outEyeR = self.eyeModel(eyesRight, ada_in)
        # Concat both eyes
        outEyes = torch.cat((outEyeL, outEyeR), 1)

        # Apply SELayer
        outEyes = self.se1(outEyes)
        # Apply a convolutional layer
        outEyes = self.conv(outEyes)
        # Apply AdaGN
        outEyes = self.agn(outEyes, 8, ada_in)
        outEyes = self.relu(outEyes)
        outEyes = self.se2(outEyes)
        outEyes = outEyes.view(outEyes.size(0), -1)
        # Apply Fully Conected layer
        outEyes = self.eyesFC(outEyes)

        # Apply eyes, face and rects features
        f_out = torch.cat((outEyes, outFace, outRect), 1)
        return self.FCLayer(f_out)
```

# Capítulo 5

# Treinamento

## 5.1 Experimento

Para o treinamento foi usado a técnica de validação cruzada k-fold [5], dada a natureza do conjunto de dados. Com isso espera-se evitar sobreajuste, ou seja, evitar que o modelo se ajuste muito bem ao conjunto de dados anteriormente observado, mas se mostre ineficaz em conjuntos de dados não observados.

A seguir são exibidos os parâmetros que foram usados no treinamento.

```
params = {
          number_of_epochs: 100,
          batch_size: 64,
          learning_rate: 0.0001,
          optimizer: adam,
          loss_function: SmoothL1Loss,
         }
```

O modelo foi treinado por 100 épocas usando o conjunto de dados de 15,151 imagens coletado previamente. Foi usada a *Smooth L1* como função perda. O treinamento levou aproximadamente 26 horas e foi realizado numa máquina com GPU de 16 gigas de memória dedicada. A taxa de aprendizado inicial foi de 0.0001, sendo diminuída para 0.00001 após 25 épocas e depois novamente para 0.000002 após 50 épocas. O tamanho do lote de treinamento (*batch size*) usado foi de 64. Esse treinamento foi repetido 5 vezes sempre iniciando um novo modelo a cada nova iteração da validação cruzada. Os hiperparâmetros foram escolhidos por tentativa e error até que a função de perda convergisse de maneira satisfatória usando como base os hiperparâmetros fornecidos em Bao et al [1].

Segue o trecho de código usado no treinamento.

```
for fold, (train_ids, test_ids) in enumerate(kfold.split(dataset)):

    train_subsampler = SubsetRandomSampler(train_ids)
    test_subsampler = SubsetRandomSampler(test_ids)

    trainLoader = DataLoader(dataset, batch_size=batch_size, shuffle=False
        , sampler=train_subsampler)
    testLoader = DataLoader(dataset, batch_size=batch_size, shuffle=False,
         sampler=test_subsampler)

    # Create new model
    net = Model()
    # Learning rate
    lr = 0.0001
    # Train
    net.train()
    # Load data into gpu
    net.to(device)
    # Create optimizer
    optimizer = torch.optim.Adam(net.parameters(), lr, weight_decay
        =0.0005)

    # Save the model in this path
    savepath = os.path.join(abspath,f'checkpoint/checkpoint_{fold}-fold.
        tar')

    print(f'Starting training {fold} fold...')
    results = {}
    results[f'fold-{fold}'] = {'mean_error': 0, 'loss': []}

    for epoch in range(num_epochs):
        if epoch > 25:
            lr = 0.00001
        elif epoch > 50:
            lr = 0.000002

        print(f'Fold: {fold}, Epoch: {epoch}, lr: {lr}')
        results[f'fold-{fold}']['loss'].append([])
        for i, data in enumerate(trainLoader):
            # Execute a training step
            loss = net.training_step(data, device)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            # Register the loss along the training process
            results[f'fold-{fold}']['loss'][epoch].append(loss.item())
            print(f'loss: {loss.item()}', end='\r')
```

```
        if epoch % 10 == 0:
            torch.save({
                'epoch': epoch,
                'model_state_dict': net.state_dict(),
                'optimizer_state_dict': optimizer.state_dict(),
                'loss': loss,
                'lr': lr,
            }, savepath)
    # Training process is complete.
    print('Training process has finished. Saving trained model.')

    torch.save({
        'epoch': epoch,
        'model_state_dict': net.state_dict(),
        'optimizer_state_dict': optimizer.state_dict(),
        'loss': loss,
        'lr': lr,
    }, savepath)
```

Segue o trecho de código usado na validação do modelo

```
    print(f'Starting testing for {fold}-fold...')
    # Evaluation for this fold
    with torch.no_grad():
        errors = []
        # Iterate over the test data and generate predictions
        for i, data in enumerate(testLoader, 0):
            # Generate gaze outputs
            output, labels, resolution = net.validation_step(data, device)

            for k, gaze in enumerate(output):
                gaze = gaze.cpu().detach()
                xyGaze = [gaze[0]*resolution[k][0], gaze[1]*resolution[k
                    ][1]]
                xyTrue = [labels[k][0]*resolution[k][0], labels[k][1]*
                    resolution[k][1]]
                errors.append(dist(xyGaze, xyTrue))

        mean_error = np.mean(np.asarray(errors)) / 38 # convert from
            pixels to cm
        results[f'fold-{fold}']['mean_error'] = mean_error
        print(f'fold: {fold}, mean error: {mean_error}')
```

## 5.2 Resultados

Abaixo seguem os gráficos do treinamento com validação cruzada *5-fold*. Percebe-se que o modelo converge rapidamente em 20 épocas e depois a taxa de convergência reduz drasticamente. O modelo foi treinado com um conjunto de dados de tamanho $15,151$ imagens, o que é consideravelmente menor que conjuntos de dados já prontos como *MPIIFaceGaze* com $213,659$ imagens e *GazeCapture* com $2,445,504$ imagens.

Seguem os gráficos da função de perda durante o treinamento junto com erro médio obtido durante a validação para cada *fold*.

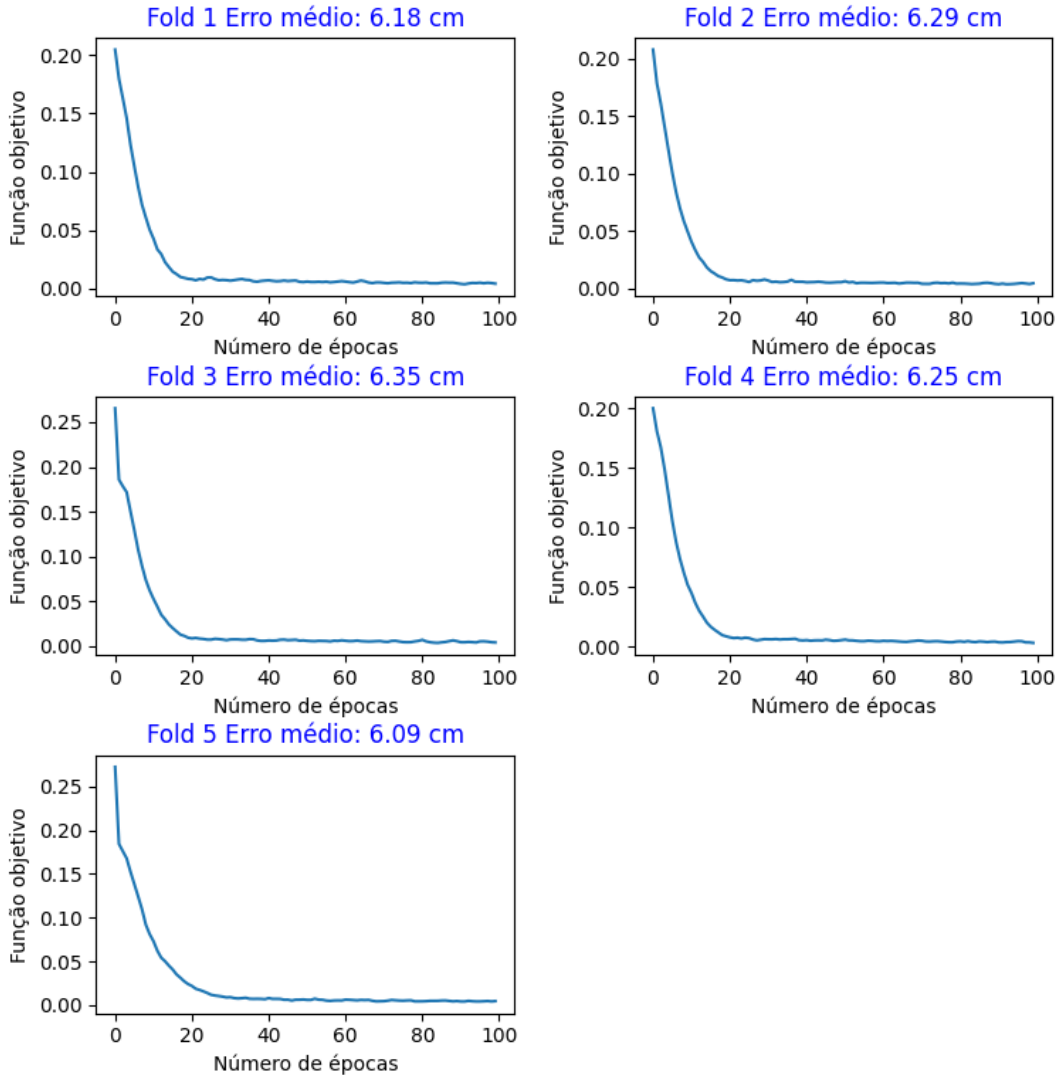


**Figura 5.1:** *O erro médio da validação cruzada é 6.23 cm*

Comparando o resultado deste trabalho com o experimento artigo do *Adaptive Feature Fusion Network for Gaze Tracking in Mobile Tablets* vemos que mesmo para um conjunto de dados pequeno o modelo *AFF-NET* é superior a outros modelos propostos anteriormente.

Experimento usando *MPIIFaceGaze* apresentado no artigo do modelo *AFF-NET* [1].

| Método | Erro médio da estimação da posição 2D do olhar (cm) |
|---|---|
| iTracker [7] | 5.46 |
| Spatial weights CNN [11] | 4.2 |
| RT-GENE [4] | 4.2 |
| AFF-NET [1] | 3.9 |

# Capítulo 6

# Conclusão

Neste trabalho empenhamos em seguir um *pipeline* completo de um trabalho de pesquisa na área de visão computacional, desde o desenho de um experimento de coleta até o treinamento de uma rede neural de convolução. O rastreador ocular baseado na aparência do olho utilizando o modelo *AFF-NET* [1] é bastante poderoso para generalizar estimadores do olhar, mesmo para conjuntos de dados pequenos. A dificuldade principal encontra-se em desenvolver um experimento de coleta eficiente e executá-lo de maneira a obter um conjunto de dados grande e confiável.

Apesar de que o tamanho do conjunto de dados usado no treinamento tenha sido pequeno, o trabalho apresentou um passo importante na direção do desenvolvimento de interfaces de controle não convencionais, e de como elas podem ter um baixo custo de produção e serem extremamente acessíveis.

Web: www.linux.ime.usp.br/~mattconce/mac0499/.

# Bibliografia


[1] Yiwei Bao, Yihua Cheng, Yunfei Liu, and Feng Lu. Adaptive feature fusion network for gaze tracking in mobile tablets. In *2020 25th International Conference on Pattern Recognition (ICPR)*, pages 9936–9943, 2021. i, iii, vii, 2, 9, 18, 19, 22, 23, 28, 31, 35, 37

[2] A. Borji and L. Itti. State of the art in visual attention modeling. *IEEE Trans Pattern Anal Mach Inte*, 35:185–207, 2013. 1, 5

[3] Kingma Diederik, Ba Jimmy, et al. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, pages 273–297, 2014. 14

[4] Tobias Fischer, Hyung Jin Chang, and Yiannis Demiris. Rt-gene: Real-time eye gaze estimation in natural environments. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 334–352, 2018. 35

[5] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction.* Springer, 2009. 14, 31

[6] Jie Hu, Li Shen, Samuel Albanie, Gang Sun, and Enhua Wu. *Squeeze-and-Excitation Networks.* PhD thesis, Department of Computer Science, Cornell University, Estados Unidos, Setembro 2017. 17

[7] Kyle Krafka, Aditya Khosla, Petr Kellnhofer, Harini Kannan, Suchendra Bhandarkar, Wojciech Matusik, and Antonio Torralba. Eye tracking for everyone. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2176–2184, 2016. 35

[8] P. Majaranta and A. Bulling. *In Advances in Physiological Computing.* Springer, 2014. 1

[9] Dale Purves and Stephen Mark Williams. *Neuroscience.* Sinauer Associates, 2 edition, 2001. 7

[10] Keith Rayner. *Eye Movements in Reading and Information Processing: 20 Years of Research.* PhD thesis, Department of Psychology, University of Massachusetts Amherst, Estados Unidos, Setembro 2017. ix, 2, 6, 7

[11] Xucong Zhang, Yusuke Sugano, Mario Fritz, and Andreas Bulling. It's written all over your face: Full-face appearance-based gaze estimation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 51–60, 2017. 35