Lista 2 - Fundamentos em Redes Neurais e Aprendizagem Estatística

Lorran de Araújo Durães Soares* Welber Paraizo Ferreira†

2024

Introdução

Este documento refere-se à elaboração da segunda lista de Fundamentos de Redes Neurais e Aprendizagem Estatística, promovida pelo Laboratório Nacional de Computação Científica (LNCC) sob a orientação do professor Gilson Antonio Giraldi. A lista consiste em três questões, que serão apresentadas neste formato de artigo, detalhando o passo a passo necessário para a resolução de cada uma. Todas as questões foram implementadas na linguagem Python, utilizando notebooks do tipo iPynb e empregando bibliotecas como `Keras` (CHOLLET et al., 2024), `Pandas` (TEAM, 2024b), `Matplotlib` (TEAM, 2024a), `Numpy` (DEVELOPERS, 2024a), `Scikit-learn` (DEVELOPERS, 2024b), `OpenCV` (OpenCV Team,), `Seaborn` (WASKOM et al., 2024) e `PIL` (CONTRIBUTORS, 2024). As implementações das questões 1 e 3 serão disponibilizadas ao final da explicação de cada uma delas neste documento.

Questão 1

Considere uma base de dados e um problema de classificação. Aplique a eave-one-out multi-fold cross-validation explanada na seção 8.5 de (GIRALDI, 2024), com K=4, para um modelo MLP. Use as facilidades disponíveis nas biliotecas para implementação de redes neurais, como Keras, Tensor flow, Scikit-learn, Matlab, etc.
Obs: Para base de dados de imagens, converta cada imagem para a escala cinza e use o histograma da imagem para extração de features.
(a) Mostre o gráfico que representa a evolução do treino e os estágios de validação.
(b) Realize uma análise estatística da performance dos quatro modelos aplicados sobre o Dte.
(c) Analise a influência do otimizador de hiperparâmetros.

Para a realização desta questão, foi escolhida a base de dados Fashion MNIST presente na biblioteca `Keras` composta por 70.000 imagens de dimensão 28x28, já em escala cinza, classificadas em dez categorias de roupas e acessórios, como exemplos de algumas delas na figura 1.

As classes presentes neste conjunto de dados são as seguintes:

1. Camiseta
2. Calça

*lorranspbr@gmail.com
†wparaizo@posgrad.lncc.br

3. Suéter
4. Vestido
5. Casaco
6. Sandália
7. Camisa
8. Tênis
9. Bolsa
10. Bota cano curto

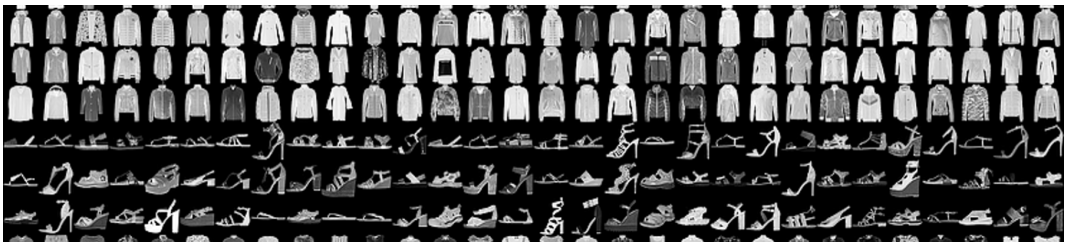

Figura 1 – Amostra do conjunto de dados

Esse exercício foi realizado de duas maneiras. Na primeira, a extração das features de cada imagem foi realizada através do cálculo do histograma de frequência de cada tom de cinza da respectiva imagem, na qual iremos denominar como Extração por Histograma. Já na segunda, foi considerado o tom de cinza de cada pixel para o treinamento da rede, que denominaremos como Extração por pixel.

0.0.1 Extração de Features por Histograma

Para a primeira maneira, inicialmente, foi realizada a extração das features pelo histograma por meio do uso da da função `cv2.calHist` da biblioteca `cv2`, tornando cada imagem com dimensão 256, relativa a frequência de cada tom de cinza. Foi então efetuada a normalização dividindo cada imagem por 784, que seria o número total máximo que um tom de cinza poderia obter, visto que a imagem possui resolução 28x28. Logo após, foi dividido o conjunto de dados através da função `train_test_split` da biblioteca `Sklearn`, de maneira aleatória em 80% para treinamento e 20% para teste. Após a divisão dos dados, foi gerado um histograma que mostra a distribuição do número de imagens por classificação no conjunto de testes. Este histograma permite avaliar se é necessário ajustar a quantidade de dados para cada classificação, identificando se alguma classe está desproporcionalmente representada e, se necessário, decidindo sobre a inclusão ou exclusão de dados. Como mostra a figura 2, não foi necessário então a adição ou remoção de dados nesse caso, pois havia uma quantidade de dados com pouca diferença entre as classes.

Para a determinação de alguns hiperparâmetros do modelo e do otimizador, foi realizado um grid search, que é um método de ajuste de hiperparâmetros, através da biblioteca `Sklearn` com a função `GridSearchCV`. Para isso, foi construido um modelo preliminar com a bilioteca `keras` usando o optimizador Adam, bastante recomendado no meio científico devido à sua eficácia e facilidade de uso em muitos tipos de problemas de machine learning. Além disso, o modelo contava com uma camada oculta de 128 neurônios e função de ativação relu, com acurácia como métrica de desempenho. Como o conjunto de classificação dos dados estão em números inteiros de 0 a 9 ao invés da forma binária, foi utilizado como função de perda a sparse_categorical_crossentropy. As funções de perda e de ativação

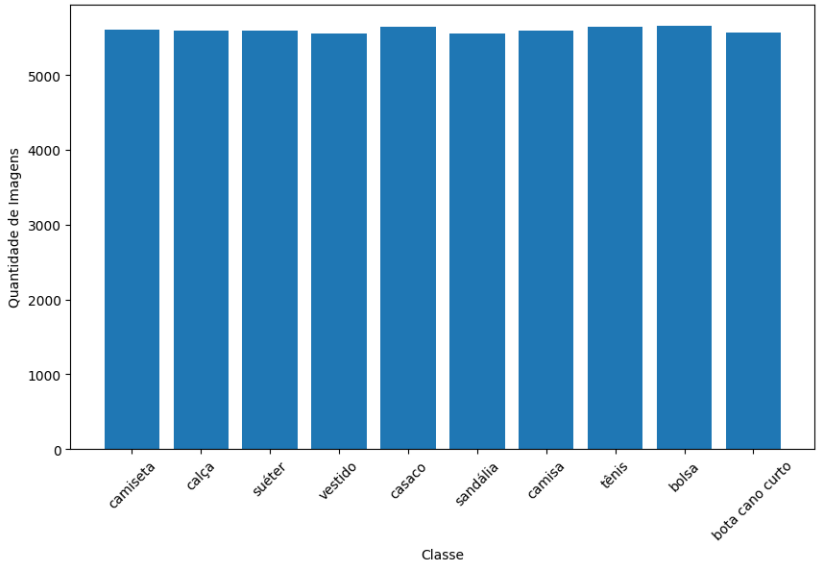

Figura 2 – Histograma da quantidade de imagens por classe no conjunto de treinamento

serão melhor descritas posteriormente na construção do modelo definitivo. Os hiperparâmetros do otimizador Adam analisado foram:

- **learning_rate**: A taxa de aprendizado controla o tamanho dos passos que o algoritmo de otimização dá ao ajustar os pesos.
- **beta_1**: O parâmetro β_1 controla a taxa de decaimento exponencial para a média dos gradientes passados (momento de primeira ordem).

  $$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

  Um benefício do β_1 é que ele promove uma convergência mais estável, ajudando a rede a escapar de mínimos locais.
- **beta_2**: O parâmetro β_2 controla a taxa de decaimento exponencial para a média dos quadrados dos gradientes passados (momento de segunda ordem):

  $$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$
- **epsilon**: O parâmetro ϵ é um pequeno valor adicionado ao denominador para evitar divisão por zero na atualização dos pesos:

  $$\theta_{t+1} = \theta_t - \eta \frac{m_t}{\sqrt{v_t} + \epsilon}$$
- **amsgrad**: O parâmetro *amsgrad* é uma variante do Adam que corrige um problema de convergência. Quando *amsgrad* é **True**, o algoritmo mantém um histórico do maior valor observado da segunda ordem do momento:

  $$\hat{v}_t = \max(\hat{v}_{t-1}, v_t)$$

  O parâmetro evita que o otimizador Adam tenha passos muito grandes, melhorando a robustez e a estabilidade do treinamento.

Hiperparâmetros	Valores Testados
Learning rate	0.001, 0.01
Beta 1	0.9, 0.95
Beta 2	0.999, 0.9999
epsilon	1e-7, 1e-8
amsgrad	False, True
Batch size	32,64,128,256
Épocas	20,30,40,50,100

Tabela 1 – Tabela de valores dos hiperparâmetros

Além disso, foram testados diferentes tamanhos para o batch size e para o número de épocas. A tabela 1 mostra todos os hiperparamêtros testados.

É notório que testar essa quantidade de hiperparâmetros é caro computacionalmente, mas havia a disponibilidade do uso de um computador com GPU, que tornou possível a realização deste processo.

A melhor combinação encontrada foi: learning rate de 0.001 que oferece um bom equilíbrio entre velocidade de convergência e estabilidade, beta_1 igual a 0.95 que dá mais peso aos gradientes recentes, suavizando as atualizações de peso, beta_2 igual a 0.9999 que estabiliza a aprendizagem em problemas com gradientes ruidosos ou esparsos. Um valor de epsilon igual a $1e-8$, proporcionando uma precisão numérica ligeiramente maior, útil em casos onde os gradientes são muito pequenos. Além disso, batch size igual a 32 e 20 épocas retornaram as melhores acurácias.

Partindo para a construção do modelo definitivo, que é formado pelos hiperparâmetros encontrados anteriormente, ele consiste de duas camadas ocultas, uma de 256 e outra de 128 neurônios, nas quais os números que foram determinados empiricamente, ou seja, foram experimentadas várias configurações de camadas ocultas e números de neurônios, baseadas em redes neurais existentes para classificação de imagens, a fim de encontrar a melhor combinação para o problema. Como função de ativação, foram utilizadas as funções descritas:

- A função de ativação **ReLu(Rectified Linear Unit)** foi utilizada nas duas camadas internas, devido às suas propriedades que facilitam o treinamento e a performance do modelo. Ela é definida como:

$$\text{ReLu}(x) = \max(0, x)$$

  Uma das vantagens de se utilizar a função ReLu é pelo motivo de ser muito simples de se calcular, pois para qualquer entrada x a função retorna o próprio valor x caso seja positivo, e 0 caso o valor da entrada seja negativo. Essa simplicidade resulta em uma computação rápida, o que é importante para o treinamento de redes neurais muito grandes. Como a ReLu define como zero todos os valores negativos, ela induz uma forma esparsa nas ativações da rede. Isso significa que, em média, apenas uma parte dos neurônios estará ativa, ou seja, terá uma saída diferente de zero para uma dada entrada. Essa forma esparça pode ajudar a tornar a rede mais eficiente em termos de armazenamento e processamento, além de introduzir uma forma de regularização implícita que pode melhorar a generalização. Em outras funções de ativação como a sigmoide e a tanh, os gradientes podem ficar muito pequenos em camadas profundas, levando a um treinamento lento ou estagnado durante o processo de aprendizagem.

- A função **Softmax** foi usada na última camada. Geralmente ela é utilizada para problemas de classificação multiclasse. Ela transforma os valores de entrada em uma distribuição de probabili-

dade, onde cada valor de saída é interpretado como a probabilidade da entrada pertencer a uma das classes. A função Softmax é definida como:

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

Uma das vantagens de se utilizar a função de ativação **softmax** é que ela converte as saídas da rede em probabilidades, com a soma de todas as probabilidades igual a 1. Isso é útil em problemas de classificação, onde queremos prever a classe à qual uma entrada pertence com base nas probabilidades.

A Softmax aponta a diferença entre os valores de entrada. Se uma das entradas for maiores que as outras, a função produzirá uma probabilidade muito alta para aquela entrada, tornando fácil identificar a classe correta. Ela é usada frequentemente em combinação com a função de perda *categorical_crossentropy* ou *sparse_categorical_crossentropy*. Essa combinação é bem estabelecida e eficaz para problemas de classificação, especialmente quando as classes são mutuamente exclusivas.

Para função de perda, foi utilizada a *sparse_categorical_crossentropy*, que é usada para problemas de classificação multiclasse. Ela mede a divergência entre a distribuição de probabilidade prevista pela rede (saída do softmax) e a verdadeira distribuição de rótulos.

A *sparse_categorical_crossentropy* é dada pela seguinte fórmula:

$$\text{Loss} = -\frac{1}{N}\sum_{i=1}^{N} \log(p_{i,y_i})$$

onde:

- N é o número de amostras no lote.
- p_{i,y_i} é a probabilidade prevista para a classe correta y_i da amostra i.

Ao contrário da função de perda *categorical_crossentropy*, onde os rótulos precisam ser representados como vetores one-hot, a *sparse_categorical_crossentropy* lida diretamente com inteiros, o que reduz a memória e o tempo de processamento.

A inicialização dos pesos e bias do modelo foi realizada de modo padrão pela biblioteca `keras` da seguinte forma:

- Seja $n_{entradas}$ o número de entradas para um neurônio e n_{saidas} o número de saídas, então os pesos são inicializados de maneira uniforme no intervalo: $\left(-\sqrt{\frac{6}{n_{\text{entradas}}+n_{\text{saídas}}}}, \sqrt{\frac{6}{n_{\text{entradas}}+n_{\text{saídas}}}}\right)$
- Os bias são inicializados com 0.

Devido à simplicidade da arquitetura da rede neural, composta por apenas duas camadas escondidas com 256 e 128 neurônios, não foi feito o uso de técnicas como dropout e batch normalization, uma vez que o risco de overfitting e instabilidade no treinamento é menor e o modelo pode ser treinado de forma eficiente sem a necessidade dessas técnicas de regularização. Além disso, não foi estipulado um critério de parada, afim de que ocorresse o trainamento durante todas as épocas e pudesse observar por mais tempo o comportamento da perda e da acurácia para os conjuntos de treinamento e validação. Como determinado pelo exercício, foi definido um K-fold igual a 4 para que fosse dividido o conjunto

de treinamento e validação, com a opção shuffle como True para que o conjunto de dados seja dividido de forma estocástica, de maneira a criar 4 modelos diferentes.

Após o treinamento, foram gerados os gráficos para acurácia e a loss ao longo das épocas, presentes nas figuras 3, 4, 5 e 6.

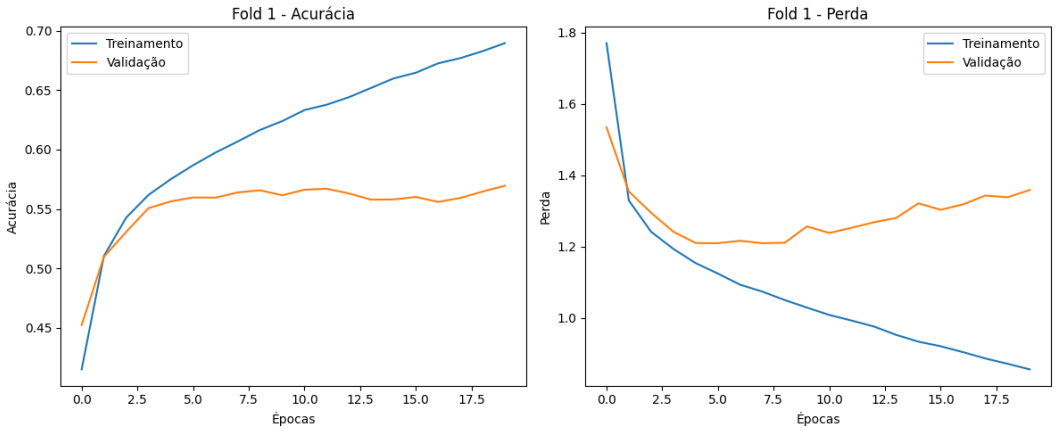

Figura 3 – Acurácia e Loss para o modelo com k=1 na Extração por Histograma

Como forma de performar uma analise estatística da performance do modelo, foi aplicado ao modelo o conjunto de testes e produzido a matriz de confusão resultante desta aplicação. Presente na figura 7. Com estes dados, baseado na seção 8.6 de (GIRALDI, 2024), foi calculado as métricas descritas, que são calculados de acordo com o número de acertos, falsos positivos e falsos negativos que cada classe apresentou em relação a cada outra classe, obtendo os valores para cada fold do modelo, descritos na tabela 2. A matriz de confusão e as métricas foram calculadas através da biblioteca `Scikit-learn`.

Como pode ser observado, o melhor modelo k-fold obteve apenas 56% de acurácia na classificação das imagens. Como forma alternativa de obter um melhor resultado nesta tarefa, foi então realizado o mesmo processo de treinamento na mesma rede, mas com uma maneira diferente de extrair as características, descrito na próxima subseção.

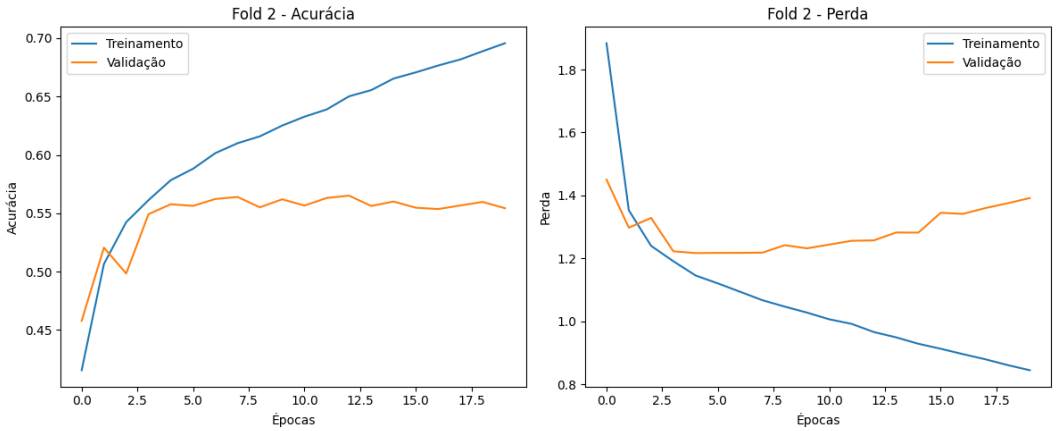

Figura 4 – Acurácia e Loss para o modelo com k=2 na Extração por Histograma

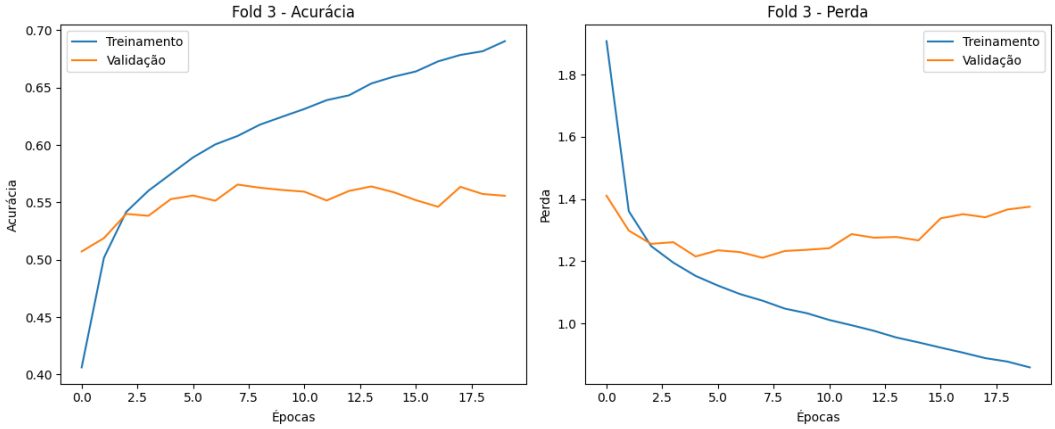

Figura 5 – Acurácia e Loss para o modelo com k=3 na Extração por Histograma

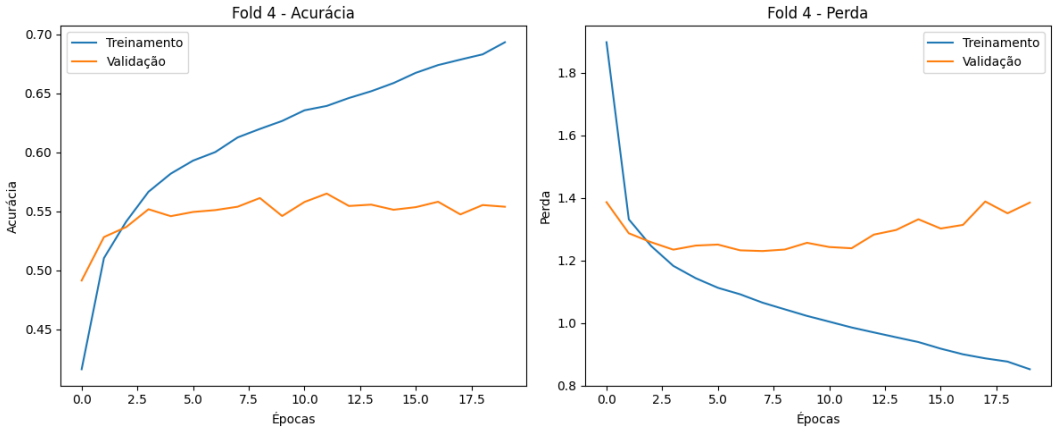

Figura 6 – Acurácia e Loss para o modelo com k=4 na Extração por Histograma

Métricas	K=1	K=2	K=3	K=4	Média
Macro Precision of the classifier	0.55	0.56	0.55	0.55	0.55
Macro Recal of the Classifier	0.55	0.55	0.56	0.55	0.55
Macro F1 score	0.55	0.55	0.55	0.54	0.55
Multi-Class Acuracy	0.55	0.56	0.56	0.55	0.55
Avanged Error Rate	0.45	0.44	0.44	0.45	0.44

Tabela 2 – Métricas de performance: Extração por Histograma

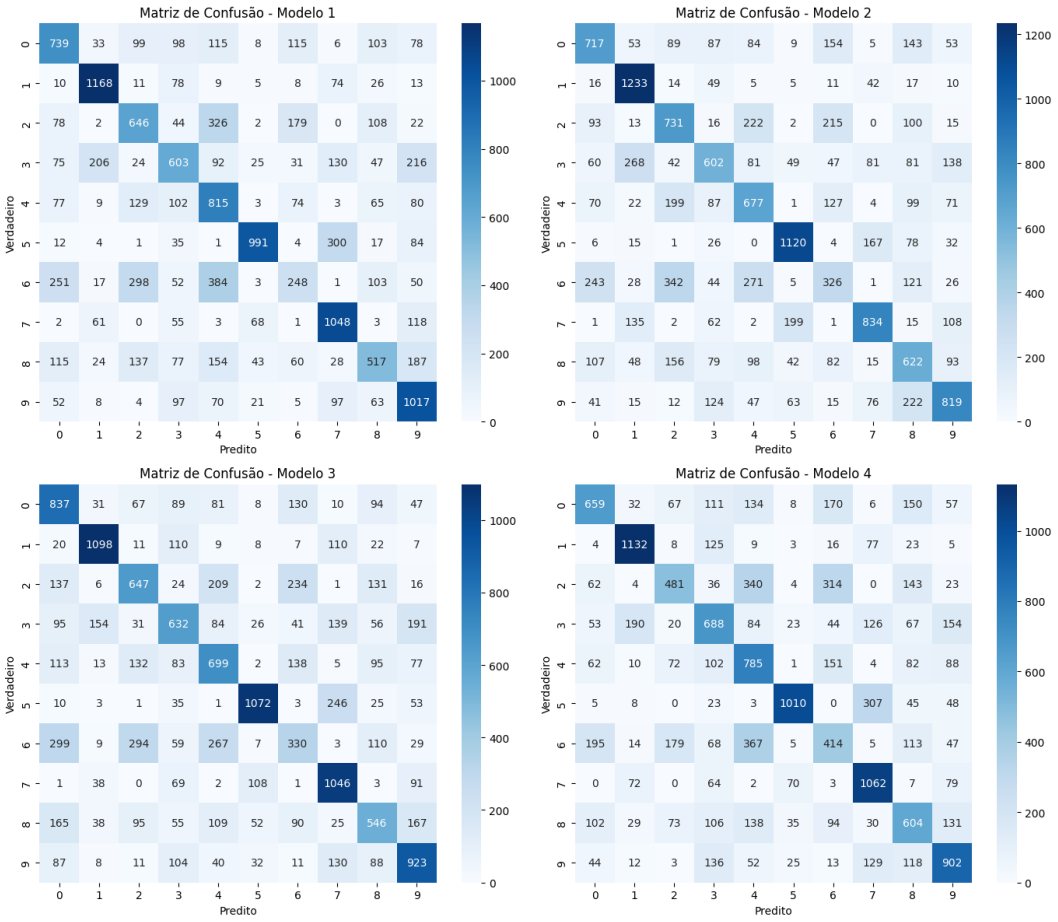

Figura 7 – Matriz de confusão para cada fold do modelo com Extração por Histograma

Métricas	K=1	K=2	K=3	K=4	Média
Macro Precision of the classifier	0.89	0.90	0.89	0.89	0.89
Macro Recal of the Classifier	0.89	0.90	0.89	0.89	0.89
Macro F1 score	0.89	0.90	0.89	0.89	0.89
Multi-Class Acuracy	0.89	0.90	0.89	0.89	0.89
Avanged Error Rate	0.11	0.10	0.11	0.11	0.11

Tabela 3 – Métricas de performance: Extração por Pixel

0.0.2 Extração de Features por Pixel

Como alternativa para uma melhorar a acurácia na classificação das imagens, foi então proposto o treinamento do mesmo modelo, mas agora, usando o grau de cinza de cada coordenada das imagens, e não a frequência relativa de cada tom de cinza como feito anteriormente. Desta maneira, o modelo pode ter mais capacidade para realizar a classificação devido a ter a referência da localização do cinza de cada pixel da imagem em questão. Logo foram utilizados os mesmos hiperparâmetros encontrados no grid search realizado anteriormente, mas alterando o número de neurônios para 512 na primeira camada oculta e 256 na segunda, mantendo o número de camadas escondidas. Esse aumento foi realizado pois, na extração por histograma, o banco de dados de imagens era de dimensão 70000 por 256, pois havia 70000 imagens e 256 tons de cinza. Mas, agora, ao trabalhar com cada pixel, como cada imagem tinha resolução 28x28, então o conjunto com as imagens vetorizadas passou a ter dimensão 70000 por 784, exigindo uma maior quantidade de neurônios, o que levou um maior custo computacional ao treinamento e à inferência.

Com esse intuito, foi realizada a vetorização das imagens do conjunto original e posteriormente normalizado esse conjunto dividindo todos os pixels por 255. Assim, dividiu se novamente o banco de dados em treinamento e teste, com uma proporção de treinamento de 80% e plotou-se o histograma de frequências de cada classificação para avaliar a necessidade de retirada ou adição de novas imagens, o que novamente não foi necessário. Portanto, foi realizado novamente o treinamento para esse conjunto de dados, apresentando a evolução da acurácia e da loss presentes nas figuras 8, 9, 10 e 11.

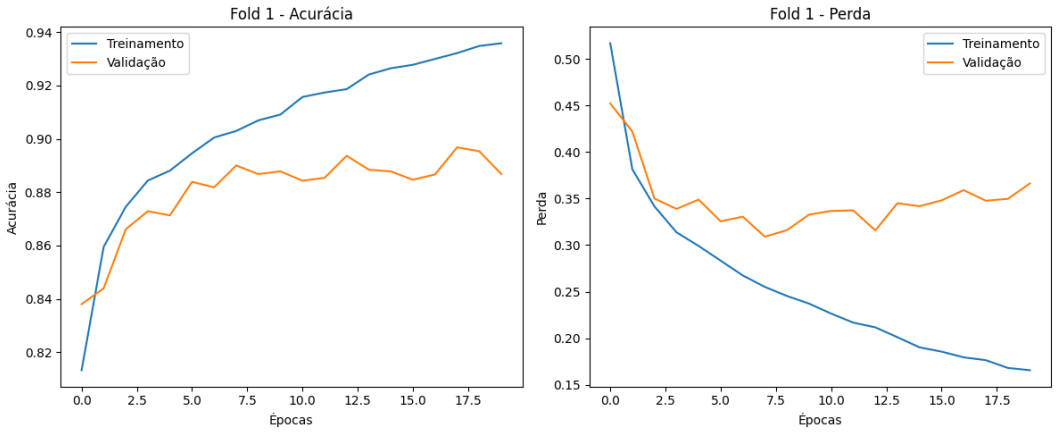

Figura 8 – Acurácia e Loss para o modelo com k=1 na Extração por Pixel

Novamente, para performar uma analise estatística do modelo, foi aplicado a ele o conjunto de testes e produzido a matriz de confusão resultante desta aplicação. Presente na figura 12. Além disso, a tabela 3 mostra o resultados das métricas de precisão neste caso.

Logo, ao observar a matriz de confusão 12 e a tabela 3, fica claro a maior efetividade que o

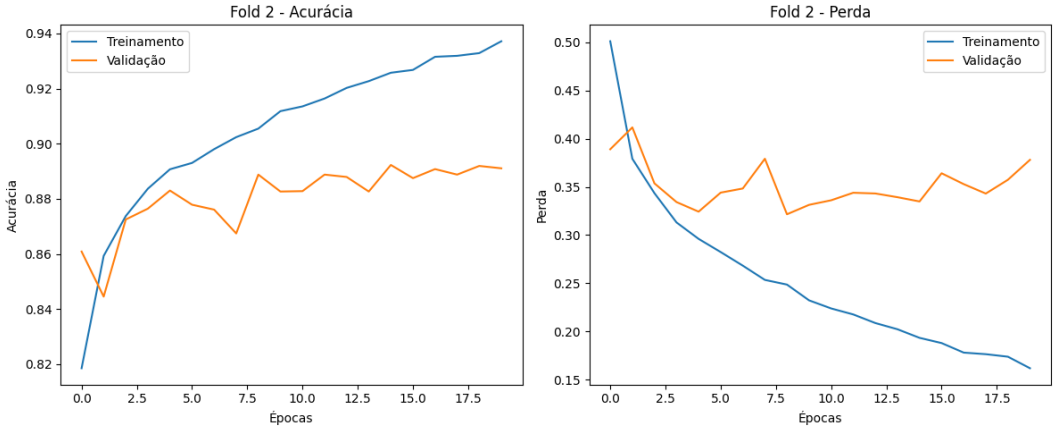

Figura 9 – Acurácia e Loss para o modelo com k=2 na Extração por Pixel

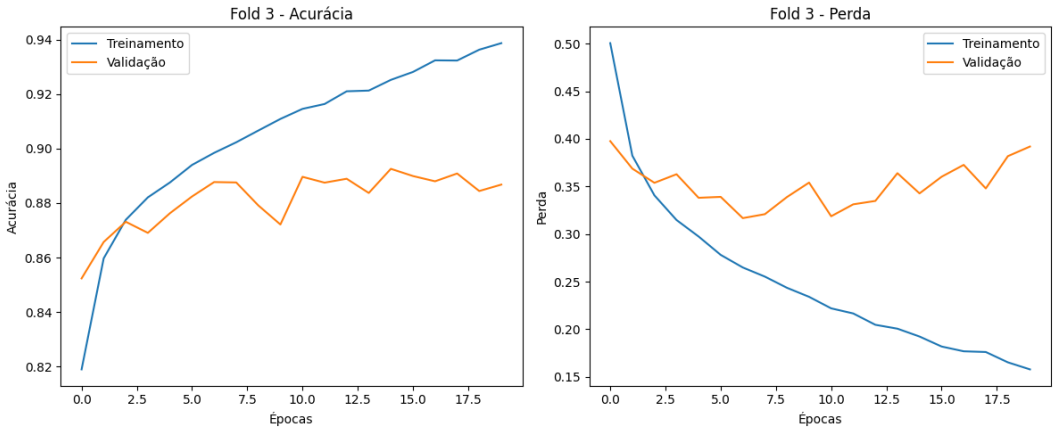

Figura 10 – Acurácia e Loss para o modelo com k=3 na Extração por Pixel

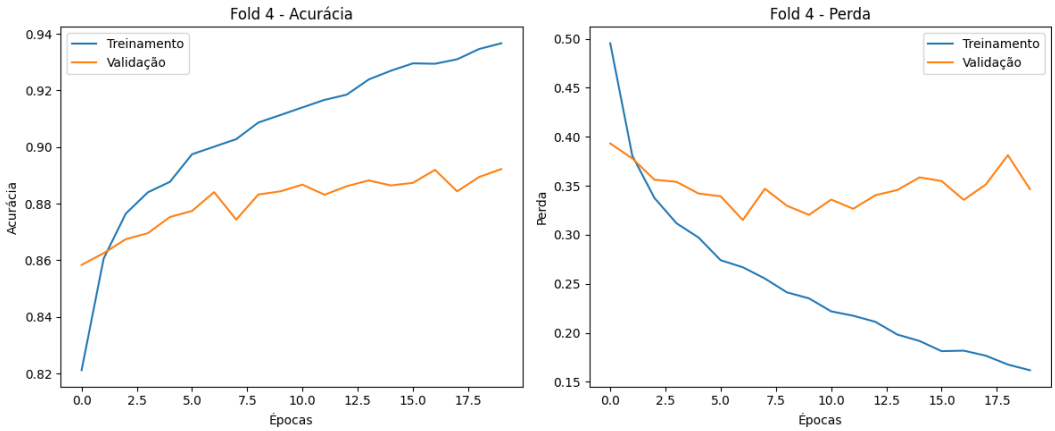

Figura 11 – Acurácia e Loss para o modelo com k=4 na Extração por Pixel

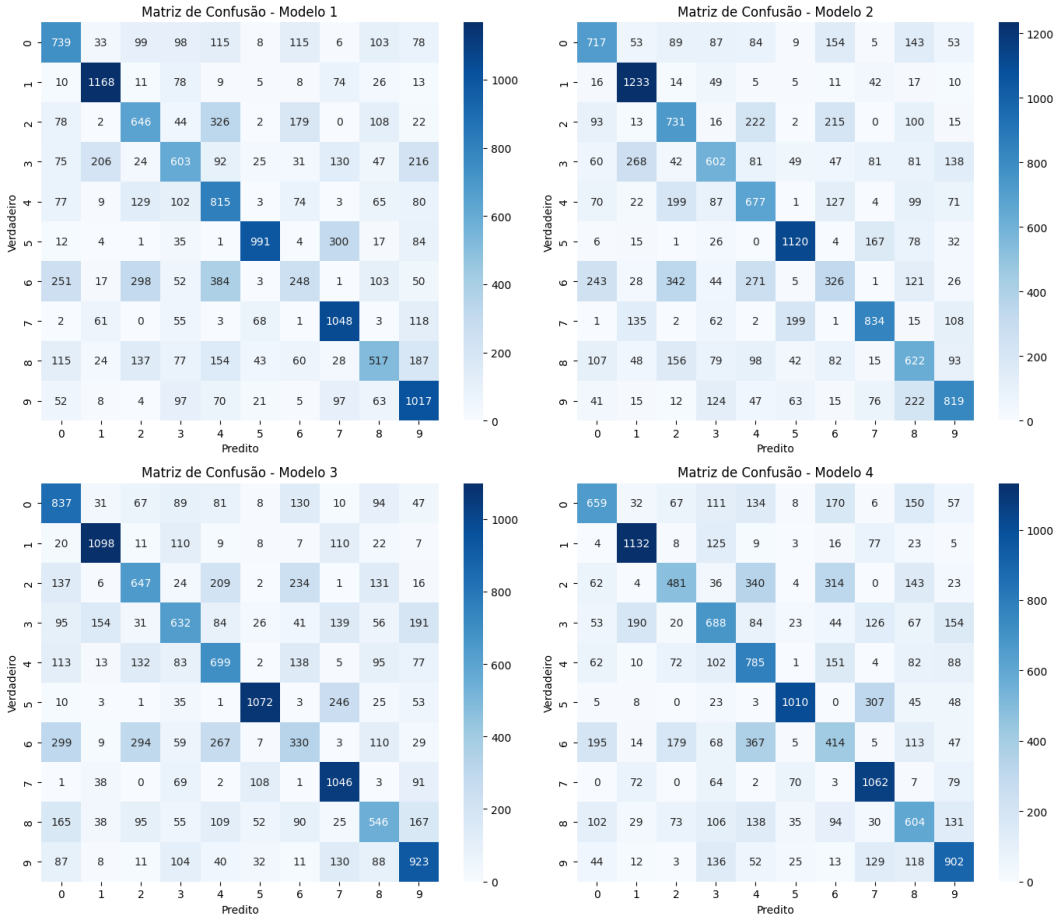

Figura 12 – Matriz de confusão para cada fold do modelo com Extração por Pixel

modelo obteve, tendo em média 89% nas métricas calculadas e chegando a apresentar 90% de acurácia para o k-fold igual a 2, que demonstrou que o modelo conseguiu classificar melhor as imagens deste modo de extração de features.

O código referente à realização do exercício descrito acima está disponível online na plataforma Github. Clique aqui para ver o código.

Questão 2

Dê a complexidade computacional de pior caso para o processo de treinamento do modelo MLP representado na Figura 9.6 da monografia (GIRALDI, 2024). Suponha uma função de ativação genérica e uma função de perda genérica.

Seja $D_{tr} = \{x_1, x_2, \ldots, x_m\}$ um conjunto de treinamento, e $R = M(x, z)$ um modelo de máquina representado por uma rede *multi-layer perceptron* (MLP) com L camadas ocultas e uma camada de saída com um único perceptron, ilustrada na Figura 13. O vetor de parâmetros z é composto pelos pesos e bias da rede neural, distribuídos em cada camada. Para uma rede com L camadas ocultas, z pode ser expresso como

$$z = \{W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)}, \ldots, W^{(L+1)}, b^{(L+1)}\},$$

onde $W^{(j)}$ e $b^{(j)}$ representam, respectivamente, os pesos e o bias da camada j. Aqui, $x \in \mathbb{R}^n$ é uma amostra do conjunto de treinamento e $z \in \mathbb{R}^q$ é um vetor de parâmetros de otimização internos da máquina. A entrada $x_k = \{x_{1,k}, x_{2,k}, \ldots, x_{n,k}\} \in D_{tr}$ possui $p^{(0)} = n$, onde p representa os perceptrons da máquina e $p(j)$ é o último perceptron da camada j. Logo, o conjunto de perceptrons da j-ésima camada interna é denotado como H^j, onde:

$$H^j = \{p^{j-1} + 1, p^{j-1} + 2, \ldots, p^j\}.$$

O conjunto de perceptrons da camada de entrada e da camada de saída é denotado como $H^{(0)}$ e $H^{(L+1)}$, respectivamente. O número de perceptrons de uma camada $H^{(j)}$, para $0 \leq j \leq L + 1$, denotado por n_j, é dado por:

$$n_j = p^j - p^{j-1}$$

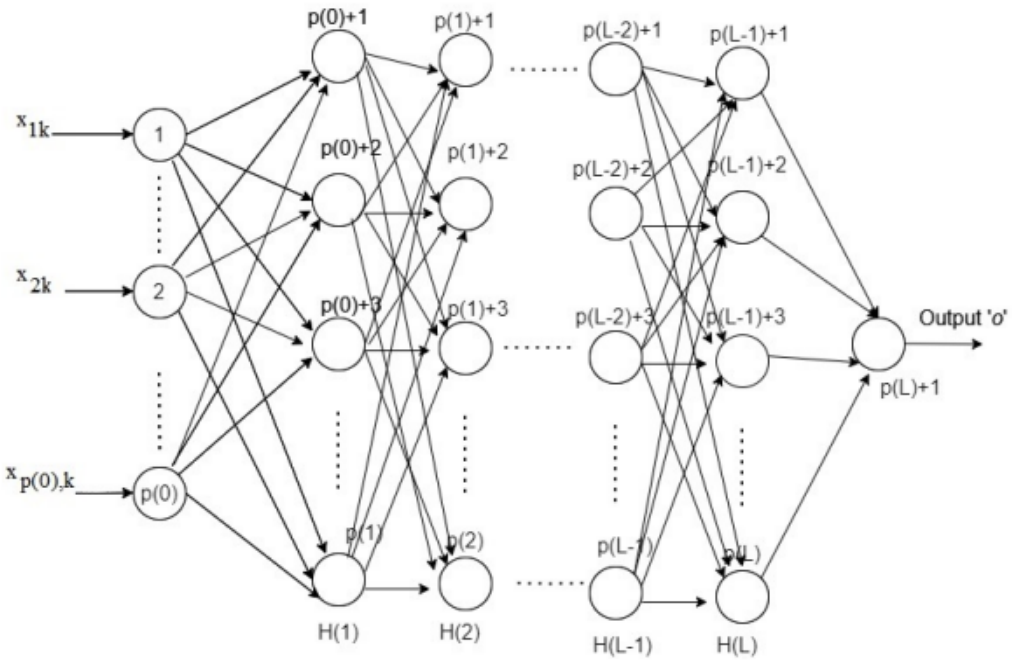

Figura 13 – Esquema de uma Deep MLP com entrada de exemplo x_k

Sejam também:

- $\mathcal{I}$ é o conjunto de todos os neurônios na camada de entrada e ;
- w_{ij} é o peso associado entre o neurônio j e o neurônio i da camada anterior;
- x_{ik} é o valor de entrada i para o exemplo k,
- b_j é o viés associado ao neurônio j na camada $H^{(1)}$.
- f é uma função de ativação genérica;
- n_0 é o número de entradas e n_j o número de neurônios na camada j.
- $o_i(k) = \sigma(\text{net}_i(k))$ é a saída do neurônio i para o exemplo k após a aplicação da função de ativação σ, onde i é a camada anterior.

Para resolver a questão sobre a complexidade computacional no pior caso para o processo de treinamento de um modelo MLP (Multi-Layer Perceptron), iremos considerar as seguintes hipóteses:

- O processo de treinamento será realizado utilizando o método Stochastic Gradient Descent com mini-batches;
- Vamos considerar que o treinamento acontecerá por todas as épocas.

Para calcular o custo computacional, levaremos em conta as seguintes etapas do treinamento de uma rede neural:

1. Foward pass
2. Backward pass
3. Atualização dos pesos

No entanto, durante o processo, ainda há o custo da inicialização dos pesos e do cálculo da Loss após o Forward Pass. No entanto, como ambos têm um custo constante, não os consideraremos, pois não influenciam na complexidade computacional do problema.

Assim, amos então calcular o custo computacional para cada etapa:

(a) Cálculo do custo computacional de pior caso do Forward Pass:

O Foward Pass é a etapa em que cada amostra do conjunto de treinamento é passada pela rede neural. A rede processa a entrada em cada camada até produzir uma saída final. Para a passagem dos dados de entrada em cada neurônio, teremos que:

1. Se $p \in H^{(1)}$ para o exemplo de entrada x_k, então será efetuada a seguinte operação:

$$\sigma(\text{net}_j(k)) = \sigma(\sum_{i \in \mathcal{I}} w_{ij} x_{ik} + b_j)$$

O número de operações efetuadas nesse caso será então:

- **Multiplicações:** n_0 multiplicações;

- **Somas:** $n_0 - 1$ somas para acumular os produtos e 1 soma para adicionar o bies;
- **Função de ativação:** custo computacional genérico para o cálculo de σ.

Assim, o número total de operações para cada neurônio p na camada $H^{(1)}$ é:

$$CC_{H^{(1)}} : 2n_0 + CC(\sigma)$$

Se houver n_1 neurônios na camada $H^{(1)}$, o custo computacional total para essa camada será:

$$CC_{H^{(1)}} = 2n_1 \cdot n_0 + CC(\sigma) \cdot n_1$$

2. Se $p \in H^{(j)}$ para o exemplo de entrada x_k, onde $2 \leq j \leq L+1$, teremos que efetuar a operação:

$$\sigma(\text{net}_j(k)) = \sigma(\sum_{i \in H^{(L-1)}} w_{ij} o_i(k) + b_j)$$

Assim, o número total de operações para cada neurônio p na camada $H^{(j)}$ é:
O número de operações efetuadas nesse caso será então:

- **Multiplicações:** n_{j-1} multiplicações;
- **Somas:** $n_{j-1} - 1$ somas para acumular os produtos e 1 soma para adicionar o bies;
- **Função de ativação:** custo computacional genérico para o cálculo de σ.

Logo:

$$CC_{H^{(1)}} : 2n_{j-1} + CC(\sigma)$$

Se houver n_j neurônios na camada $H^{(j)}$, o custo computacional total para essa camada será:

$$CC_{H^{(j)}} = 2n_j \cdot n_{j-1} + CC(\sigma) \cdot n_j$$

Então, o custo computacional total de todas as camadas será dado por:

$$CC_a : CC_{H^{(1)}} + \sum_{i=2}^{L+1} CC_{H^{(j)}}$$

$$CC_a : 2n_1 \cdot n_0 + CC(\sigma) \cdot n_1 + \sum_{i=2}^{L+1} (2n_i \cdot n_{i-1} + CC(\sigma) \cdot n_i)$$

$$CC_a : \sum_{i=1}^{L+1} (2n_i \cdot n_{i-1} + CC(\sigma) \cdot n_i)$$

Considerando a hipótese de que todas as camadas ocultas possuem l neurônios, temos então que:

$$CC_a : 2n_0 \cdot l + CC(\sigma) + (L-1)(2p^2 + CC(\sigma) \cdot l) + 2 \cdot l$$

Logo, esse é o custo computacional que a rede possui para que um dado de entrada x_k. Ao longo desta etapa, vamos considerar que a saída de cada neurônio está sendo salva para que esse valor possa ser usado posteriormente no calculo do gradiente da função Loss em relação aos pesos, mas vamos desconsiderar o custo computacional de armazenamento.

(b) Cálculo do custo computacional de pior caso do Backward:

Nesta fase, o erro calculado é propagado de volta à rede, camada por camada. Vamos considerar uma função de perda genérica:

$$L(w;B) \equiv \frac{1}{|B|}\sum_{k=1}^{|B|} l(R(\mathbf{p},\mathbf{x_k}),d_j)$$

onde sua derivada $\frac{\partial L}{\partial o_j(k)}$ terá um custo constante $CC_{Loss'}$. A derivada do erro em relação a cada peso ou bias será dada por:

$$\frac{\partial L}{\partial w_{ij}} = \sum_{k=1}^{|B|}\sum_{\xi=p(0)+1}^{p(L)+1} \frac{\partial L}{\partial \text{net}_\xi(k)}\frac{\partial \text{net}_\xi(k)}{\partial w_{ij}},$$

$$\frac{\partial \mathcal{L}}{\partial b_j} = \sum_{k=1}^{|\mathcal{B}|}\sum_{\xi=p(0)+1}^{p(L)+1} \frac{\partial \mathcal{L}}{\partial \text{net}_\xi(k)}\frac{\partial \text{net}_\xi}{\partial b_j}.$$

Logo, dividiremos esse momento em duas partes:

- Cálculo do delta para cada camada: $\delta_j(k) = \frac{\partial L}{\partial \text{net}_j(k)}$;
- Cálculo de $\frac{\partial \text{net}_\xi(k)}{\partial w_{ij}}$.

0.0.3 Cálculo dos deltas:

1. **Cálculo de $\delta_j(k)$ para $j \in$ Out:**

   Teremos que:

   $$\delta_j(k) = \frac{\partial L}{\partial o_j(k)} \cdot \frac{\partial o_j(k)}{\partial \text{net}_j(k)} = \frac{\partial L}{\partial o_j(k)} \cdot \sigma'(\text{net}_j(k)),$$

   Assim, para cada neurônio j na camada de saída, teremos que computar:

   - Cálculo da derivada $\sigma'(\text{net}_j(k))$ (Apenas uma vez);
   - Cálculo da derivada $\frac{\partial L}{\partial o_j(k)}$ (Apenas uma vez);
   - Uma multiplicação entre as derivadas.

   Portanto, como neste exemplo temos apenas um neurônio de saída, então o custo computacional total por neurônio j na camada de saída Out será:

   $$CC_{Out} = CC(\sigma') + CC(Loss') + 1$$

2. **Cálculo de $\delta_j(k)$ para $j \in H^{(L)}$**

   Teremos que:

   $$\delta_j(k) = \sigma'(\text{net}_j(k)) \left[\sum_{\xi\in \text{Out}} \frac{\partial L}{\partial \text{net}_\xi(k)} w_{j\xi}\right],$$

Mas como já calculamos no passo anterior as derivadas descritas, então teremos que efetuar as seguintes operações:

- Como há apenas um neurônio na camada de saída, será necessário apenas uma multiplicação entre o peso e a derivada dentro do somatório;
- Uma multiplicação do resultado do somatório sobre a derivada da função σ.

Logo, o custo computacional para um neurônio na camada L será dado por:

$$CC_{j\in H^{(L)}} = 2$$

Se a camada $H^{(L)}$ posssui n_L neurônios, teremos então que o custo computacional total para essa camada será dado por:

$$CC_{H^(L)} = 2n_L$$

3. **Cálculo de delta se $\delta_j(k)$ para $j \in H^{(n)}$, com $1 \leq n \leq L-1$:**

$$\delta_j(k) = \frac{\partial L}{\partial \text{net}_j(k)} = \sigma'(\text{net}_j(k))\frac{\partial L}{\partial o_j(k)} = \sigma'(\text{net}_j(k))\left[\sum_{\xi\in H^{(n+1)}} \frac{\partial L}{\partial \text{net}_\xi(k)}\frac{\partial \text{net}_\xi(k)}{\partial o_j(k)}\right].$$

Mas nesse caso, $\sigma'(\text{net}_j(k))$ já foi calculado anteriomente, cada derivada $\frac{\partial L}{\partial \text{net}_\xi(k)}$ foi calculada no passo anterior e cada derivada $\frac{\partial \text{net}_\xi(k)}{\partial o_j(k)}$ são os pesos que ligam esse neurônio à sua camada posterior. Logo, será necessário fazer as seguintes operações:

- (n_{n+1}) produtos dentro do somatório;
- $n_{n+1} - 1$ somas do somatório;
- $n_{n+1} - 1$ multipicações da derivada da função de ativação com cada item do somatório.

Logo, para um neurônio desta camada, teremos o seguinte custo computacional:

$$CC_{j\in H^(n)} = (n_{n+1}) + n_{n+1} - 1 + n_{n+1} - 1$$

$$CC_{j\in H^(n)} = 3n_{n+1} - 2$$

Se a camada n tem n_n neurônios, o custo computacional total para essa camada será dado então por:

$$CC_{H^(n)} = n_n(3n_{n+1} - 2)$$

Mas isso acontecerá para todas as outras camadas, logo o custo computacional do cálculo dos deltas para as camadas $1 \leq n \leq L-1$ será dado por:

$$CC_{H^{(n)}} = \sum_{n=1}^{L-1} n_n(3n_{n+1} - 2)$$

Então, o custo computacional total para o cálculo de todos os deltas será dado por:

$$CC_\delta = \sum_{l=1}^{L-1} [n_l(3n_{l+1}+1)] + 2n_L + CC(\sigma') + CC(Loss') + 1$$

Supondo que todas as camadas internas tenha o mesmo número l de neurônios, temos então o seguinte custo computacional:

$$CC_\delta = (L-1)[l(3l+1)] + 2l + CC(\sigma') + CC(Loss') + 1$$

$$CC_\delta = (L-1)[3l^2+l)] + 2l + CC(\sigma') + CC(Loss') + 1$$

0.0.4 Cálculo de $\frac{\partial \text{net}_\xi(k)}{\partial w_{ij}}$

- **Se** $j \in H^{(1)}$**:**

$$\frac{\partial \text{net}_j(k)}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}} \left(\sum_{\xi \in \mathcal{I}} w_{\xi j} x_{\xi k} + b_j \right) = x_{ik},$$

$$\frac{\partial \text{net}_j(k)}{\partial b_j} = 1.$$

- **Se** $j \in H^{(n)}$**, com** $2 \leq n \leq L$**:**

$$\frac{\partial \text{net}_j(k)}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}} \left(\sum_{\xi \in H^{(n-1)}} w_{\xi j} o_\xi(k) + b_j \right) = o_i(k), \quad i \in H^{(n-1)},$$

$$\frac{\partial \text{net}_j(k)}{\partial b_j} = 1.$$

- **Se** $j \in$ **Out:**

$$\frac{\partial \text{net}_j(k)}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}} \left(\sum_{\xi \in H^{(L)}} w_{\xi j} o_\xi(k) + b_j \right) = o_i(k), \quad i \in H^{(L)},$$

$$\frac{\partial \text{net}_j(k)}{\partial b_j} = 1.$$

Nesse 3 casos, já foram contabilizados as derivadas durante o processo de forward pass, pois os resultados das derivadas são as saidas dos neurônios e as entradas da rede.

Portanto, podemos agora calcular o custo da derivada da função Loss, em relação a cada peso.

1. Derivada em Relação aos Pesos w_{ij}

A expressão para a derivada em relação aos pesos é dada por:

$$\frac{\partial L}{\partial w_{ij}} = \sum_{k=1}^{|B|} \sum_{\xi=p(0)+1}^{p(L)+1} \frac{\partial L}{\partial \text{net}_\xi(k)} \frac{\partial \text{net}_\xi(k)}{\partial w_{ij}},$$

onde:

- $|B|$ é o número de amostras no lote;
- n_ξ é o número total de neurônios nas camadas subsequentes a j.
- A derivada $\frac{\partial L}{\partial \text{net}_\xi(k)}$ ja foi realizado o custo computacional anteriormente;
- Enquanto a derivada $\frac{\partial \text{net}_\xi(k)}{\partial w_{ij}}$ também ja calculada durante o processo de forward pass.

Análise de Custo

Supondo l neurônios por camada interna, então teremos no total de $p(0)+1$ a $p(L)+1$ $L \cdot l + 1$ neurônios. Logo, para cada amostra k, calcular o somatório envolve $L \cdot l + 1$ multiplicações e $L \cdot l$ somas. Portanto, o custo computacional para cada amostra é aproximadamente $(L \cdot l + 1) + L \cdot l$ operações.

O custo total é então dado por:

$$CC_{w_{ij}} = |B|(2L \cdot l + 1)$$

2. Derivada em Relação ao Bias b_j

A expressão para a derivada em relação ao bias é dada por:

$$\frac{\partial L}{\partial b_j} = \sum_{k=1}^{|B|} \sum_{\xi=p(0)+1}^{p(L)+1} \frac{\partial L}{\partial \text{net}_\xi(k)} \frac{\partial \text{net}_\xi(k)}{\partial b_j}.$$

Como $\frac{\partial \text{net}_\xi(k)}{\partial b_j} = 1$, a expressão se simplifica para:

$$\frac{\partial L}{\partial b_j} = \sum_{k=1}^{|B|} \sum_{\xi=p(0)+1}^{p(L)+1} \frac{\partial L}{\partial \text{net}_\xi(k)}.$$

O custo computacional para cada amostra é então $L+1$ somas, pois cada camada possui um bias. Multiplicando pelo número total de amostras $|B|$, temos:

$$CC_{\partial b_j} = |B|(L+1)$$

Logo, o custo computacional total para o Backward será dado por:

$$CC_{(b)} = CC_{\partial b_j} + CC_{\partial w_{ij}} + CC_\delta = |B|(L+1) + |B|(2L \cdot l + 1) + (L-1)[3l^2 + l)] + 2l + CC(\sigma') + CC(Loss') + 1$$

c) Cálculo do Custo Computacional da atualização dos pesos

Durante a atualização dos pesos, é realizado a seguinte operação:

$$w_{ij} \leftarrow w_{ij} + \eta \left(\frac{\partial L}{\partial w_{ij}} \right)$$

onde:

- w_{ij} é o peso entre os neurônios i e j,
- η é a taxa de aprendizado,
- $\frac{\partial L}{\partial w_{ij}}$ é a derivada da função de perda em relação ao peso w_{ij}, onde o custo também ja foi calculado durante o processo.

Para atualizar um peso w_{ij}, você precisa realizar:

- 1 multiplicação (para $\frac{\partial L}{\partial w_{ij}} \times \eta$);
- 1 adição (para atualizar w_{ij}).

Então o custo computacional para o cálculo da atualização de pesos é dada por:

$$CC_{w_{ij}} = 1\text{Multiplicação} + 1\text{Adição}$$

então:

$$CC_{w_{ij}} = 2$$

Uma rede com n_0 entradas, L camadas internas e l neurônios por camada terá $n_0 \cdot l + (L-1)l^2 + 1$ pesos no total. Logo, o custo para atualizar todos esses pesos será dado por:

$$CC_{w_{ij}} = 2 \; cdot(n_0 \cdot l + (L-1)l^2 + 1)$$

Para a atualização dos bias, teremos que:

$$b_j \leftarrow b_j + \eta \frac{\partial \mathcal{L}}{\partial b_j}$$

Então:

$$CC_{b_j} = 2$$

Uma rede com L camadas escondidas terá $L + 1$ bias. Logo, o custo total para atualizar os bies será dado por:

$$CC_{b_j} = 2(L + 1)$$

Logo, teremos que o custo computacional total da atualização de pesos será dada por:

$$CC_{(c)} : 2 \cdot (n_0 \cdot l + (L-1)l^2 + 1) + 2(L + 1)$$

Conclusão

O Forward Pass será realizado para cada dado k dentro de um mini-batch. Tanto o Forward quanto o Backward Pass serão realizados para cada um dos b mini-batches. Finalmente, esse processo será repetido por e épocas. Assim, o custo total será dado por:

$$CC_{total} = ((k \cdot CC_{(a)}) + CC_{(b)} + CC_{(c)}) \cdot b \cdot e$$

$CC_{total} = ((k \cdot (2n_0 \cdot l + CC(\sigma) \cdot l + (L-1)(2l^2 + CC(\sigma) \cdot l) + 2 \cdot l + CC(\sigma)) + k(L+1) + k(2L \cdot l + 1) + (L-1)[3l^2 + l)] + 2l + CC(\sigma') + CC(Loss') + 1 + 2 \cdot (n_0 \cdot l + (L-1)l^2 + 1) + 2(L+1)) \cdot b \cdot e$

A fim de realizar algumas simplificações, devido ao grande número de constantes envolvidas no custo computacional do problema. Vamos reescrever a equação do custo computacional, da seguinte forma:

$$CC_{total} = a_1 \cdot bekLl^2 + a_2 \cdot bekLl + a_3 \cdot bekl + a_4 \cdot bel + a_5 \cdot beL + a_6 \cdot bel^2 + a_7 \cdot be$$

Temos então que $CC_{total} = \Theta(L \cdot l^2 \cdot k \cdot b \cdot e)$, onde:

- L é o número de camadas internas;
- l é o número de neurônios por camada interna;
- k é o número de dados em um mini-batch;
- b é o número de mini-batchs;
- e é o número de épocas que ocorrerá o treinamento.

De fato:
A notação $\Theta(f(n))$ significa que a função $g(n)$, onde $n \in \mathbb{N}^n$, pertence à classe $\Theta(f(n))$ se existir $c_1 > 0$, $c_2 > 0$ e n_0 tal que:

$$c_1 \cdot f(n) \leq g(n) \leq c_2 \cdot f(n) \quad \text{para todo } n_i \geq n_{0i}.$$

Primeiro, vamos mostrar que:

$$a_1 \cdot bekLl^2 + a_2 \cdot bekLl + a_3 \cdot bekl + a_4 \cdot bel + a_5 \cdot beL + a_6 \cdot bel^2 + a_7 \cdot be \leq C_2 \cdot L \cdot l^2 \cdot k \cdot b \cdot e$$

para alguma constante $C_2 > 0$.

Observamos que cada termo individualmente é uma combinação de L, l^2, k, b e e. O termo $a_1 \cdot bekLl^2$ já está na forma $L \cdot l^2 \cdot k \cdot b \cdot e$, enquanto os outros termos têm fatores menores em L, l e k, o que significa que:

$$a_1 \cdot bekLl^2 + a_2 \cdot bekLl + a_3 \cdot bekl + a_4 \cdot bel + a_5 \cdot beL + a_6 \cdot bel^2 + a_7 \cdot be \leq (a_1 + a_2 + a_3 + a_4 + a_5 + a_6 + a_7) \cdot L \cdot l^2 \cdot k \cdot b \cdot e$$

Podemos escolher $C_2 = a_1 + a_2 + a_3 + a_4 + a_5 + a_6 + a_7$ para obter o limite superior.

Agora, vamos mostrar que:

$$C_1 \cdot L \cdot l^2 \cdot k \cdot b \cdot e \leq a_1 \cdot bekLl^2 + a_2 \cdot bekLl + a_3 \cdot bekl + a_4 \cdot bel + a_5 \cdot beL + a_6 \cdot bel^2 + a_7 \cdot be$$

para alguma constante $C_1 > 0$.

Como $a_1 \cdot bekLl^2$ é o termo de maior ordem e todos os coeficientes a_i são positivos, temos:

$$a_1 \cdot bekLl^2 \leq a_1 \cdot bekLl^2 + a_2 \cdot bekLl + a_3 \cdot bekl + a_4 \cdot bel + a_5 \cdot beL + a_6 \cdot bel^2 + a_7 \cdot be$$

Logo, podemos escolher $C_1 = a_1$ para o limite inferior:

$$C_1 \cdot L \cdot l^2 \cdot k \cdot b \cdot e \leq a_1 \cdot bekLl^2 + a_2 \cdot bekLl + a_3 \cdot bekl + a_4 \cdot bel + a_5 \cdot beL + a_6 \cdot bel^2 + a_7 \cdot be$$

Dessa forma, mostramos que existem $C_1 > 0$ e $C_2 > 0$ tais que, para qualquer $n = (L, l, k, b, e) \in \mathbb{N}$, teremos que:

$$C_1 \cdot L \cdot l^2 \cdot k \cdot b \cdot e \leq CC_{total} \leq C_2 \cdot L \cdot l^2 \cdot k \cdot b \cdot e$$

Portanto, concluímos que:

$$CC_{total} = \Theta(L \cdot l^2 \cdot k \cdot b \cdot e)$$

dado que $k \cdot b$ é o número de amostras de treinamento, faremos então:

$$k \cdot b = N_{amostras}$$

então, teremos que Θ do custo computacional do treinamento de uma rede MLP é dado por:

$$CC_{total} = \Theta(L \cdot l^2 \cdot N_{amostras} \cdot e)$$

Questão 3

Estude a teoria do PCA para problemas de pequeno tamanho de amostra, onde o número de dados é menor do que a dimensão do espaço de dados. Escolha um banco de dados de imagens, converta as imagens para escala de cinza e aplique a teoria de "PCA para problemas de pequeno tamanho de amostra"para redução de dimensionalidade.

(a) *Se $\bar{x}$ é a média amostral (centroide do conjunto de dados) e p_1 é o componente principal, visualize o resultado da expressão:*

$$x = \bar{x} + \alpha p_1,$$

onde $\alpha \in \{-\beta\lambda_1, 0, \beta\lambda_1\}$ com λ_1 sendo o autovalor associado a p_1 e β um fator escalar.

(b) *Estude o espectro da matriz $X^T X$ para realizar a redução de dimensionalidade. Visualize algumas imagens no espaço de dimensão reduzida.*

(c) *Construa um gerador de imagens usando os d componentes principais escolhidos no item (b).*

Para realizar a redução de dimensionalidade usando Análise de Componentes Principais (PCA) em um conjunto de dados onde o número de amostras é inferior à dimensão do espaço dos dados, foi escolhida a base de imagens 'frontalimages_spatiallynormalized' da FEI Face Database (THOMAZ, 2024). Esta base é composta por 400 imagens faciais frontais, já em escala de cinza e normalizadas, de homens e mulheres, sérios e sorridentes, com resolução de 260x360 pixels. Todos os plots de imagens e gráficos foram realizados através da biblioteca `matplotlib` do Numpy. Para o carregamento dessas imagens, obtidas em (THOMAZ, 2024), foram utilizadas as bibliotecas `OS` e `PIL`, que também realizaram a conversão das imagens para arrays numpy, o formato desejado para os cálculos subsequentes.

Como este é um problema com poucas amostras em relação à dimensionalidade dos dados, para o cálculo da matriz P_{PCA}, com o objetivo de minimizar a perda de informação das imagens ao reduzir sua dimensionalidade, seguimos os passos descritos em (MIRANDA, 2023). Aqui, X representa a matriz de dados já vetorizada:

1. Calculo da matriz $\tilde{X}$ de dados centralizados pela expressão

$$\tilde{X} = \begin{bmatrix} \tilde{x}_1^T \\ \tilde{x}_2^T \\ \vdots \\ \tilde{x}_N^T \end{bmatrix} \in \mathbb{R}^{N \times n},$$

   onde N é o número de amostras e n é a dimensão do espaço de dados (neste caso, $360 \cdot 260 = 93600$) e cada $\tilde{x}_i$ é centralizado da forma $\tilde{x}_i = x - \bar{x}$, onde $\bar{x}$ é a média dos dados.

2. Resolução do problema de autovalores e autovetores para a matriz $\frac{1}{N}\tilde{X}\tilde{X}^T \in \mathbb{R}^{N \times N}$:

$$\frac{1}{N}\tilde{X}\tilde{X}^T v_i = \lambda_i v_i,$$

3. Calculo dos vetores $\omega_i = \tilde{X}^T v_i, \quad i = 1, 2, \ldots, N.$

4. Normalização dos vetores ω_i para obter os autovetores da matriz de covariância $S = \frac{1}{N}\tilde{X}^T\tilde{X}$:

$$a_i = \frac{\omega_i}{\|\omega_i\|} = \frac{\tilde{X}^T v_i}{\|\tilde{X}^T v_i\|}.$$

Os autovetores obtidos ao final desse processo, ordenados em ordem decrescente em relação aos respectivos autovalores, são as colunas da matriz P_{PCA} que queríamos.

O cálculo dos autovetores e autovalores da matriz foi realizada através da função `np.linalg` e a multiplicação de matrizes foi calculada por meio da função `np.dot`, ambas presentes na biblioteca científica `Numpy`.

Para o item a), o resultado da expressão $\mathbf{x} = \bar{\mathbf{x}} + \alpha \mathbf{p_1}$, com $\alpha \in \{-\beta\sqrt{\lambda_1}, 0, \beta\sqrt{\lambda_1}\}$, com $\beta = 0.8$, resultou nas imagens presentes na figura 14, onde λ_1 é o autovalor correspondente à componente principal $\mathbf{p_1}$. A legenda coeficiente 1 se trata do valor $-\beta\sqrt{\lambda_1}$, o coeficiente 2 é 0 e o coeficiente 3 é $\beta\sqrt{\lambda_1}$. As imagens demonstraram que a componente principal ficou mais responsável pelo fundo da imagem, que era um fator bem comum a todas as fotos.

Figura 14 – Imagens geradas para o item (a)

Para o item (b), ao calcular os autovalores da matriz $S = \frac{1}{N}\tilde{X}^T\tilde{X}$ e os ordenar em ordem descrescente, foi plotado o gráfico da variância explicada individual e acumulada dos autovalores, com o objetivo de analisar a contribuição de cada um na representação do conjunto de dados e identificar as principais componentes. A partir do gráfico 15, foi então decidido aplicar a redução para as 150 primeiras componentes , foi decidido aplicar a redução para as 150 primeiras componentes, pois estas já representavam mais de 90% da energia total dos autovalores.

Foi então aplicado o truncamento multiplicando a matriz P_{PCA} por I_{150}, onde I tem dimensão 400×400, que é o tamanho das colunas da matriz P_{PCA}, mas tem até os primeiros 150 números da diagonal preenchido por 1, e todo o resto por 0. Para visualizar então como ficaram as imagens após essa redução, foi realizado o cálculo $X \cdot P_{PCA} \cdot P_{PCA}^T$, obtendo então as imagens presentes na 16 com a comparação com suas originais. Nas imagens, é notada a redução de qualidade e nitidez em relação à imagem original, mas de forma que as faces ainda mantiveram seus traços e podem ser reconhecidas.

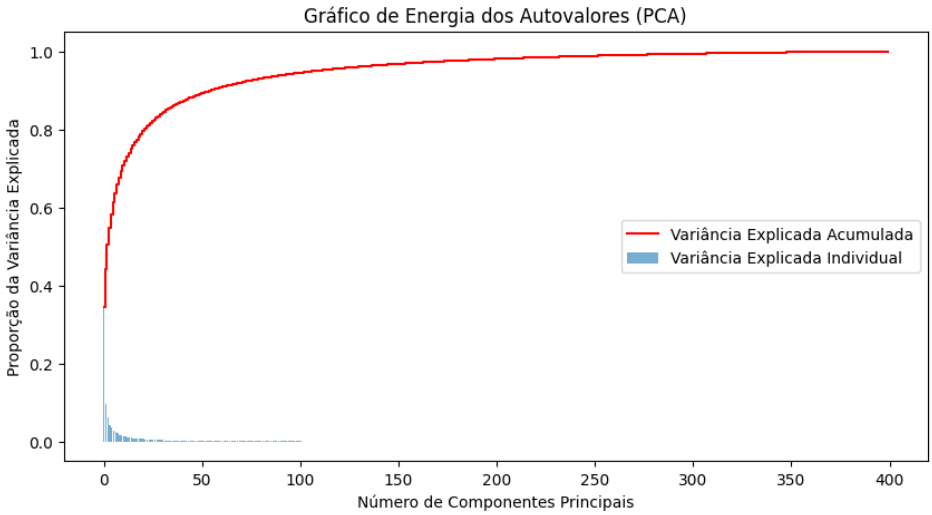

Figura 15 – Gráfico de energia dos autovalores

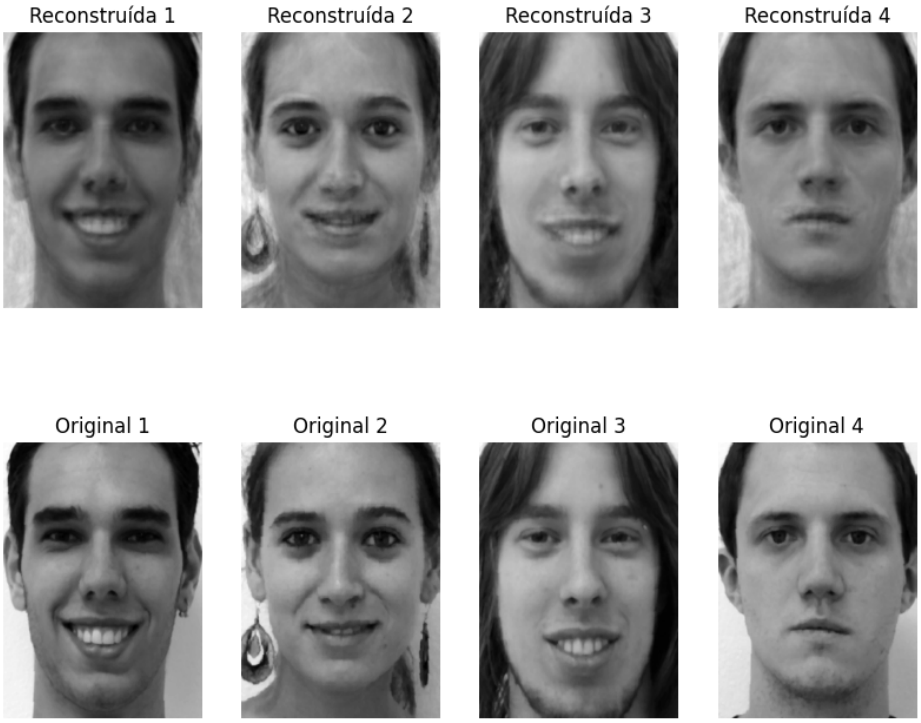

Figura 16 – Comparação reconstruída e original

Para o item (c), na construção do gerador de imagens, foi realizada a operação de maneira analoga ao item (a), só que agora, ao invés de usar apenas a primeira componente principal, foi considerado as 150 primeiras componentes principais. Para a seleção dos coeficientes β, através da biblioteca `numpy`, foi realizado o sorteio do número entre 0 e 1, com o uso de uma distribuição de probabilidade uniforme disponibilizada pela biblioteca denominada `np.random.uniform` , resultando, por exemplo, nas quatro imagens de novas faces, dada em 17. Ao realizar várias gerações, observou se faces com características bem diferentes, como lábios, sombrancelha e narizes. Em um dos testes realizados, no qual a imagem não consta aqui nestre trabalho, chegou a ser gerada uma pessoa de óculos e outras sem.

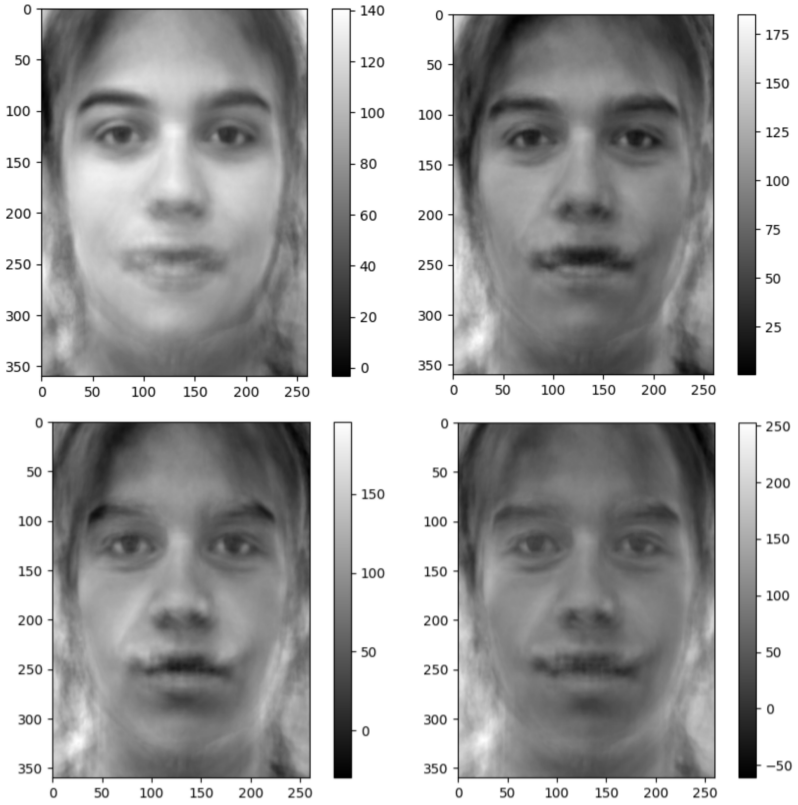

Figura 17 – Imagem gerada pelas 150 primeiras componentes

O código referente à realização do exercício descrito acima está disponível online na plataforma github. Clique aqui para ver o código.

Referências

CHOLLET, F. et al. *Keras documentation*. 2024. <https://keras.io/>. Accessed: 2024-08-22. Citado na página 1.

CONTRIBUTORS, P. *Pillow documentation*. 2024. <https://pillow.readthedocs.io/>. Accessed: 2024-08-22. Citado na página 1.

DEVELOPERS, N. *NumPy documentation*. 2024. <https://numpy.org/doc/>. Accessed: 2024-08-22. Citado na página 1.

DEVELOPERS, S. learn. *Scikit-learn documentation*. 2024. <https://scikit-learn.org/stable/>. Accessed: 2024-08-22. Citado na página 1.

GIRALDI, G. A. Fundamentals of neural networks and statistical learning. 2024. Citado 3 vezes nas páginas 1, 6 e 12.

MIRANDA, L. C. P. *Abordagens computacionais para calcular componentes principais ponderadas com aplicações em análise de imagens de faces humanas*. 2023. Citado na página 22.

OpenCV Team. *OpenCV documentation*. <https://docs.opencv.org/>. Accessed: 2024-08-22. Citado na página 1.

TEAM, M. D. *Matplotlib documentation*. 2024. <https://matplotlib.org/stable/contents.html>. Accessed: 2024-08-22. Citado na página 1.

TEAM, P. D. *Pandas documentation*. 2024. <https://pandas.pydata.org/docs/>. Accessed: 2024-08-22. Citado na página 1.

THOMAZ, C. E. *FEI Face Database*. 2024. <https://fei.edu.br/~cet/facedatabase.html>. Accessed: 2024-08-14. Citado na página 22.

WASKOM, M. et al. *Seaborn: statistical data visualization*. 2024. <https://seaborn.pydata.org/>. Accessed: 2024-08-22. Citado na página 1.