Oleksandra Kukharska - 2020151174

# Cap. 1: Descrição do Problema

O objetivo deste trabalho é aplicar aprendizagem por transferência ao *dataset* **CoMNIST**, que consiste em imagens de letras do alfabeto cirílico organizadas em 34 classes. Modelos previamente treinados em grandes bases de dados serão utilizados para adaptar os modelos ao novo *dataset*, que contém menos dados de treino O modelo será otimizado para ser eficiente. A aplicação final do modelo será demonstrada por meio de uma web API, capaz de classificar imagens em tempo real.

As metas principais do projeto incluem:

1. **Seleção das Arquiteturas Pré-Treinadas**: Utilizar *VGG16*, *MobileNetV2*, *ResNet50* e *InceptionV3*;
2. **Otimização de Hiper parâmetros**: Explorar combinações de parâmetros, como unidades da camada densa, taxa de aprendizado e taxa de *dropout*;
3. **Avaliação de Desempenho**: Utilizar matrizes de confusão e métrica de *accuracy* para análise de acertos e erros, por classe;
4. **Implementação em Ambiente Real**: Construção de uma web API funcional para demonstração.

Os experimentos dos modelos pré-treinados são desenvolvidos em *Python*, utilizando o ambiente Anaconda, e as seguintes bibliotecas:

- ***TensorFlow/Keras***: Para criar e treinar os modelos CNN, incluindo modelos pré-treinados;
- ***Keras Tuner***: Para otimizar os Hiper parâmetros;
- ***NumPy* e *CSV***: Para manipulação de dados e registo dos resultados;
- ***Scikit-learn***: Para avaliação de desempenho, como validação cruzada e cálculo de métricas.

A web API é desenvolvida no ambiente Apidog para permitir a demonstração funcional da aplicação.

# Cap. 2: Descrição das Metodologias utilizadas

Neste Capítulo é descrito o *dataset*, a otimização de Hiper parâmetros e as arquiteturas pré-treinadas utilizadas.

## 2.1. Dataset

O *dataset* utilizado para o trabalho é o **CoMNIST**, retirado do Github, e constituído por imagens 34 classes de letras do alfabeto cirílico. O *dataset* já está balanceado, e dividido em 80% Treino e 20% Teste, tendo assim, cada classe contém 344 imagens de Treino e 87 de Teste. Para garantir uma avaliação robusta do modelo é utilizada a validação cruzada. Segue-se de seguida as 34 letras utilizadas para o reconhecimento:

*Tabela 1 Letras utilizadas para reconhecimento*

| | | | | | | |
|---|---|---|---|---|---|---|
| **А** | **В** | **С** | **Е** | **Н** | **І** | **К** |
| **М** | **О** | **Р** | **Т** | **Х** | **Б** | **Г** |
| **Д** | **Ё** | **Ж** | **З** | **И** | **Й** | **Л** |
| **П** | **У** | **Ф** | **Ц** | **Ч** | **Ш** | **Щ** |
| **Ъ** | **Ы** | **Ь** | **Э** | **Ю** | **Я** | |

O nome dado às classes consiste numa numeração de 1 a 34, a Figura 1, apresenta uma amostra aleatória por classe.

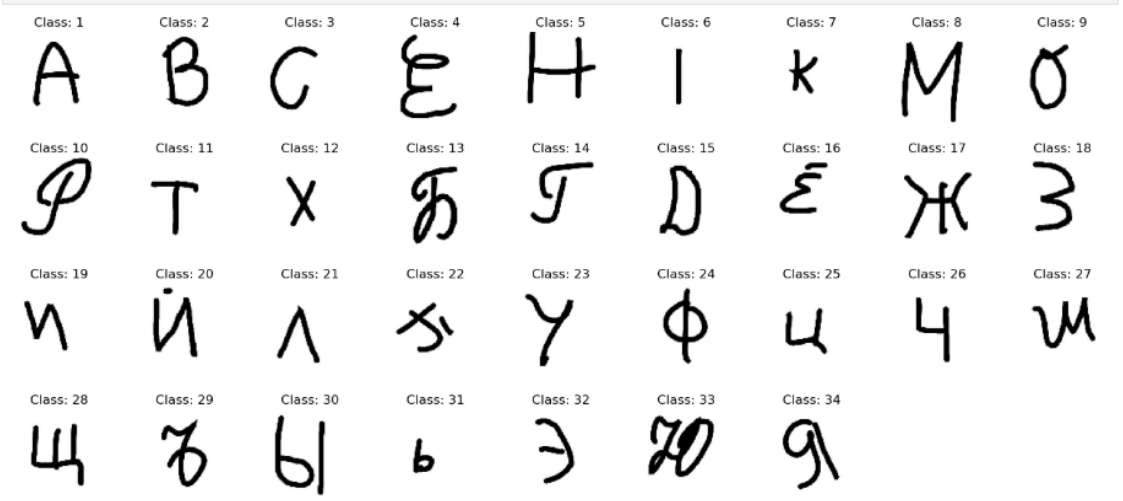


*Figura 1 Amostra random por classe*

## 2.2. Otimização de Hiper parâmetros

A otimização de Hiper parâmetros é feita com o método *Random Search*, que faz uma procura aleatória no espaço de 3 Hiper parâmetros nos seguintes intervalos:

- ***Units*:** [64, 512]

- ***Dropout*:** [0.2, 0.5]
- ***Learning rate*:** [1e-4, 1e-2]

O *Keras Tuner* foi utilizado para gerar combinações aleatórias e selecionar as melhores configurações, baseando-se nos resultados da validação.

## 2.3. Arquiteturas pré-treinadas

Em seguida, serão descritas as arquiteturas de redes neurais pré-treinadas que serão utilizadas no projeto:

- **VGG16**: *Very Deep Convolutional Networks* composta por 16 camadas treináveis;
- **ResNet-50**: *Residual Networks* com 50 camadas, útil para evitar problemas de gradientes desvanecentes;
- **MobileNetV2**: Rede otimizada para dispositivos móveis, leve e eficiente;
- **InceptionV3**: Arquitetura com múltiplos filtros de convolução de camada.

# Cap. 3: Apresentação da Arquitetura de Código

Neste capítulo, será apresentada a organização e a estrutura do código implementado para cada uma das arquiteturas pré-treinadas selecionadas. Será apresentada uma ilustração gráfica do diagrama de cada arquitetura utilizada, gerados utilizando a ferramenta Netron,

## 3.1. Arquitetura VGG16

A arquitetura VGG16 além das camadas originais já treinadas, possui 448 neurónios nas camadas densas do modelo, com uma taxa de *dropout* de 0.3.

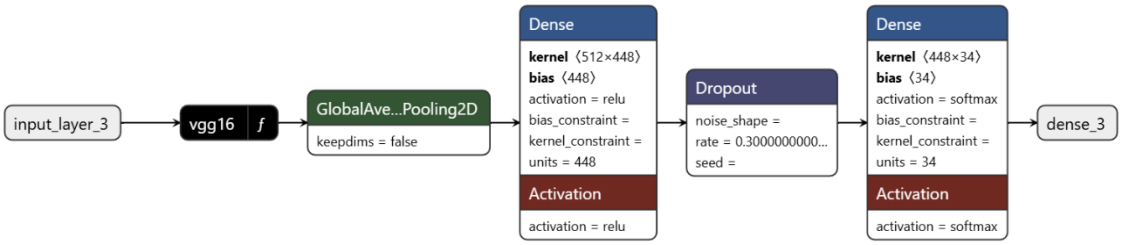


*Figura 2 Diagrama Netron da arquitetura VGG16*

## 3.2. Arquitetura ResNet50

A arquitetura ResNet50 além das camadas originais já treinadas, possui 512 neurónios nas camadas densas do modelo, com uma taxa de *dropout* de 0.2.

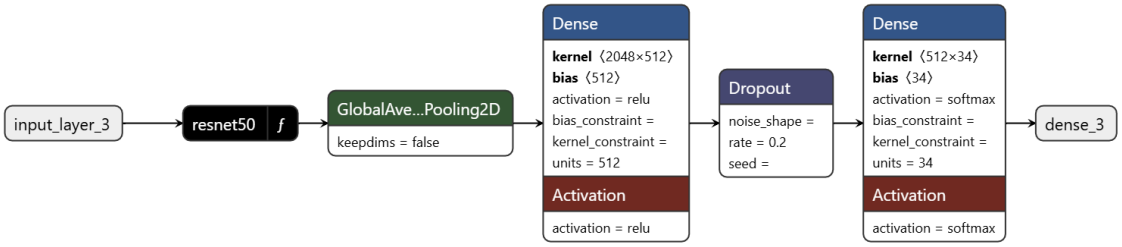


*Figura 3 Diagrama Netron da arquitetura ResNet50*

## 3.3. Arquitetura MobileNetV2

A arquitetura MobileNetV2 além das camadas originais já treinadas, possui 256 neurónios nas camadas densas do modelo, com uma taxa de *dropout* de 0.4.

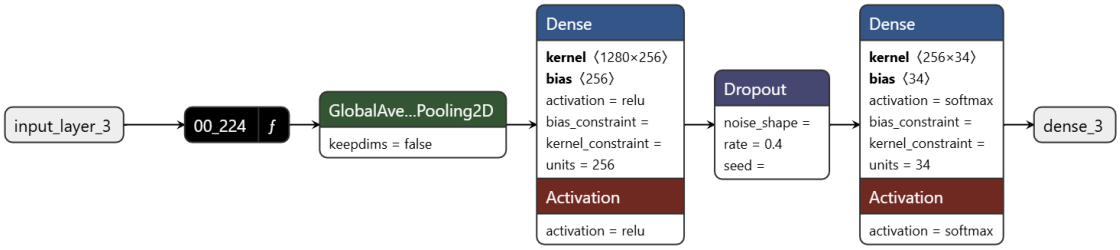


*Figura 4 Diagrama Netron da arquitetura MobileNetV2*

## 3.4. Arquitetura InceptionV3

A arquitetura InceptionV3 além das camadas originais já treinadas, possui 256 neurónios nas camadas densas do modelo, com uma taxa de *dropout* de 0.3.

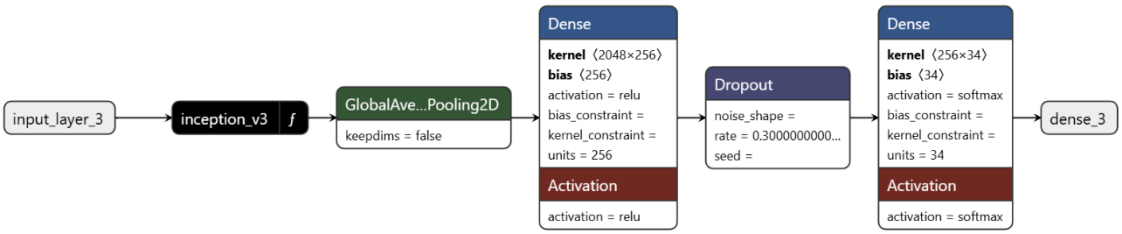


*Figura 5 Diagrama Netron da arquitetura InceptionV3*

# Cap. 4: Descrição da Implementação dos algoritmos

Neste Capítulo, é descrita a implementação prática dos algoritmos utilizados com ênfase no processo de otimização dos Hiper parâmetros, que é realizada através do método de *Random Search*, com o objetivo de encontrar a melhor combinação de parâmetros para cada modelo. O código foi estruturado para executar múltiplos *trials* em cada *fold* da validação cruzada, e os resultados obtidos em cada execução são registados em ficheiros CSV.

## 4.1. Configuração dos Modelos

Para as 4 arquiteturas, o parâmetro `trainable = False` é definido, o que implica o congelamento de todas as camadas da arquitetura. Isso evita o treino das camadas já aprendidas pelo modelo, garantindo que os pesos dessas camadas não sejam atualizados durante o treino. Apenas as camadas adicionadas ao modelo são treinadas. Essa abordagem visa aproveitar as características que as arquiteturas já aprenderam com o *ImageNet*, sem repetir o treino de todas as suas camadas.

## 4.2. Estrutura de Execução dos Trials e Folds

O processo de treino e validação foi configurado de modo a realizar 3 *trials* em cada *fold* da validação cruzada. Para cada trial, uma combinação aleatória de Hiper parâmetros foi selecionada e utilizada, através do *Random Search*. Ao término de cada execução, os dois melhores resultados de cada *fold* foram registados em ficheiros CSV para análise subsequente. A tabela a seguir ilustra a organização dos *trials* e *folds*:

*Tabela 2 Trials e Folds realizados por arquitetura*

| ***Fold*** | ***Trial 1*** | ***Trial 2*** | ***Trial 3*** |
|---|---|---|---|
| ***Fold 1*** | Hiper parâmetro 1 | Hiper parâmetro 2 | Hiper parâmetro 3 |
| ***Fold 2*** | Hiper parâmetro 1 | Hiper parâmetro 2 | Hiper parâmetro 3 |
| ***Fold 3*** | Hiper parâmetro 1 | Hiper parâmetro 2 | Hiper parâmetro 3 |
| ***Fold 4*** | Hiper parâmetro 1 | Hiper parâmetro 2 | Hiper parâmetro 3 |

Para cada arquitetura pré-treinada utilizada, o processamento foi realizado de forma independente, ou seja, cada modelo foi configurado e treinado separadamente, sem reutilização de funções ou classes entre elas.

## 4.3. Dimensão das imagens

As arquiteturas foram selecionadas com base na sua robustez e desempenho em tarefas de classificação de imagens. De maneira a reduzir recursos

computacionais, cada arquitetura foi configurada para receber a dimensão mínima permitida de entrada, conforme detalhado na tabela abaixo:

*Tabela 3 Dimensão utilizada por arquitetura*

| ***Arquitetura pré-treinada*** | ***Dimensão usada*** |
|---|---|
| *VGG16* | 32*32 |
| *MobileNetV2* | 32*32 |
| *ResNet50* | 32*32 |
| *InceptionV3* | 75*75 |

## 4.4. Deployment

Os modelos foram salvos em formato .h5, e na web API foi utilizado o modelo com melhor desempenho, conforme será abordado no Cap. 5. A implementação da API foi realizada no *Apidog*, que foi configurado para receber imagens, processá-las e retornar a classe prevista. Foram realizados testes para validar a usabilidade da API, incluindo exemplos de entradas e saídas para demonstrar seu funcionamento.

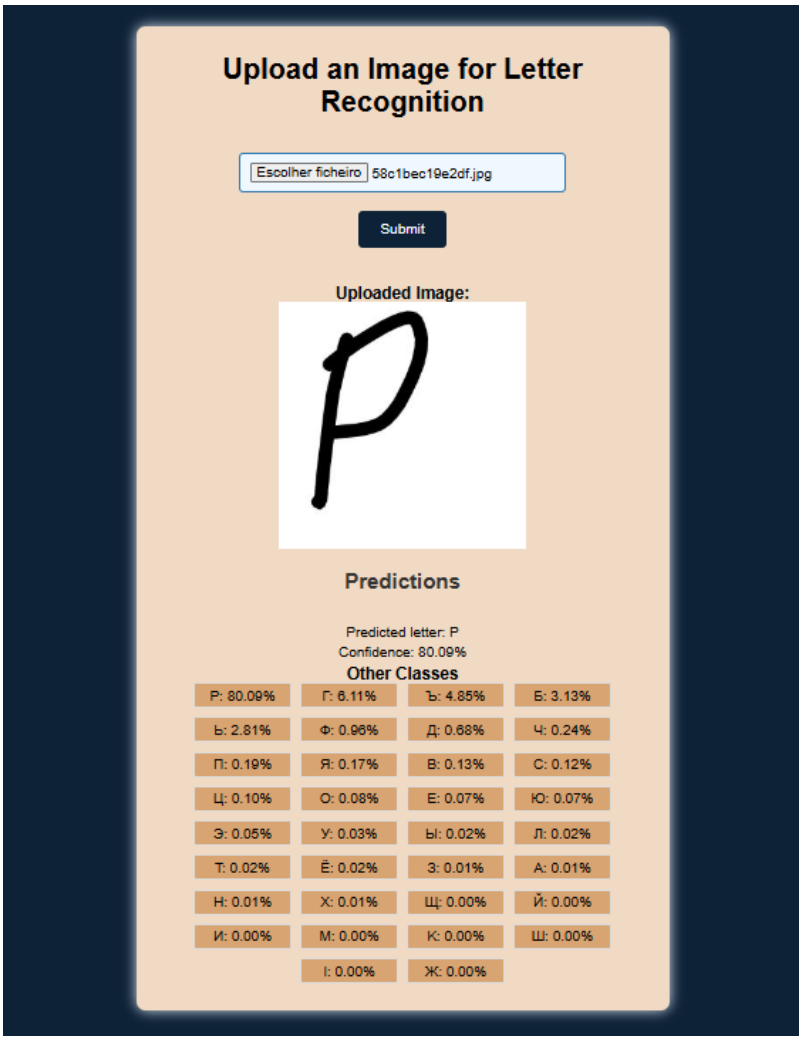


*Figura 6 Interface da API desenvolvida*

# Cap. 5: Análise de Resultados

Neste Capítulo, são apresentados e analisados os resultados obtidos durante os experimentos realizados com as 4 arquiteturas pré-treinadas. Após uma análise individual de cada arquitetura, será feita uma comparação entre as arquiteturas e a avaliação dos tempos de execução.

## 5.1. Análise do Desempenho de cada arquitetura

Para analisar o desempenho de cada arquitetura, são considerados os resultados de *val_accuracy* para cada *fold* e *trial*.

### 5.1.1. VGG16

A arquitetura VGG16 apresentou resultados não muito variados e altos ao longos dos *folds* e *trials*. Com base na validação cruzada, como se observa na Figura 7, a melhor combinação de Hiper parâmetros verifica-se no *fold* 1 e *trial* 1.

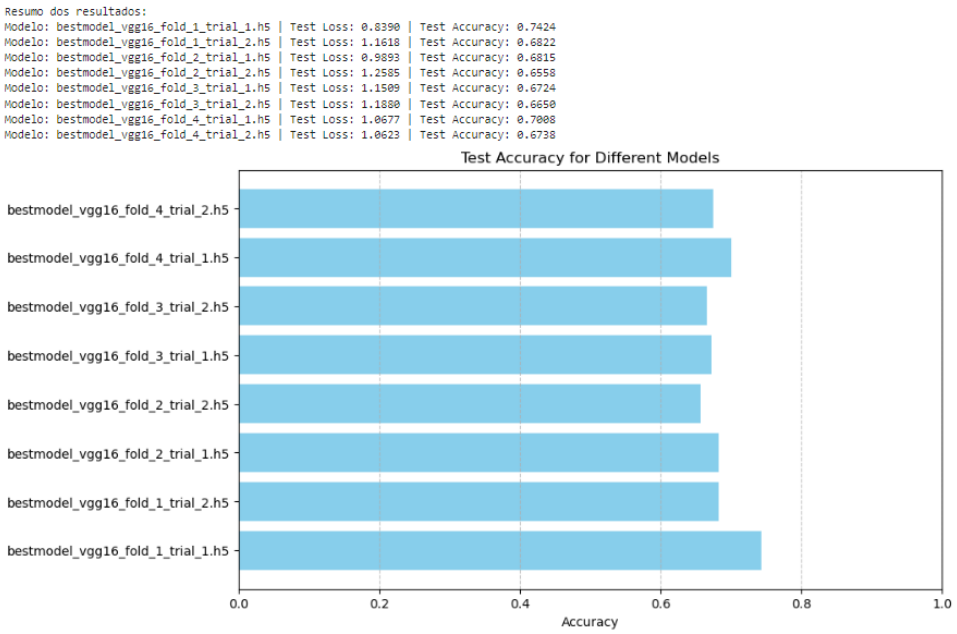


*Figura 7 Resumo dos resultados dos dados de treino, da arquitetura VGG16*

*Tabela 4 Hiper parâmetros testados com VGG16*

| fold | trial | units | dropout | learning_rate |
|---|---|---|---|---|
| 1 | 1 | 448 | 0.3 | 0.0009052494618776521 |
| 1 | 2 | 448 | 0.3 | 0.0009052494618776521 |
| 2 | 1 | 192 | 0.4 | 0.004299211781488068 |
| 2 | 2 | 128 | 0.3 | 0.00027363407710605464 |
| 3 | 1 | 512 | 0.3 | 0.00017241146658242794 |
| 3 | 2 | 448 | 0.3 | 0.00015814523764048707 |
| 4 | 1 | 384 | 0.4 | 0.0002686150982120402 |
| 4 | 2 | 64 | 0.4 | 0.0028309808141425185 |

Para os dados de teste, a *accuracy* atingiu 70,08%.

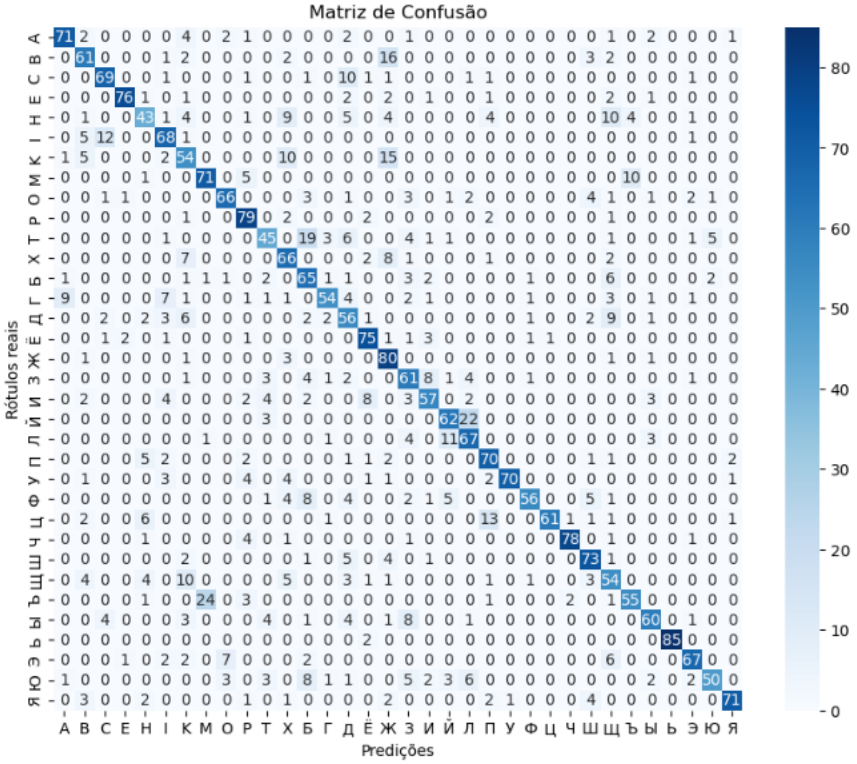


*Figura 8 Matriz de Confusão da arquitetura VGG16*

```
Test Loss: 1.0676963329315186
Test Accuracy: 0.7008113861083984
```

### 5.1.2. ResNet50

A arquitetura ResNet50 apresentou resultados baixos ao longos dos *folds* e *trials*, como se observa na Figura 9.

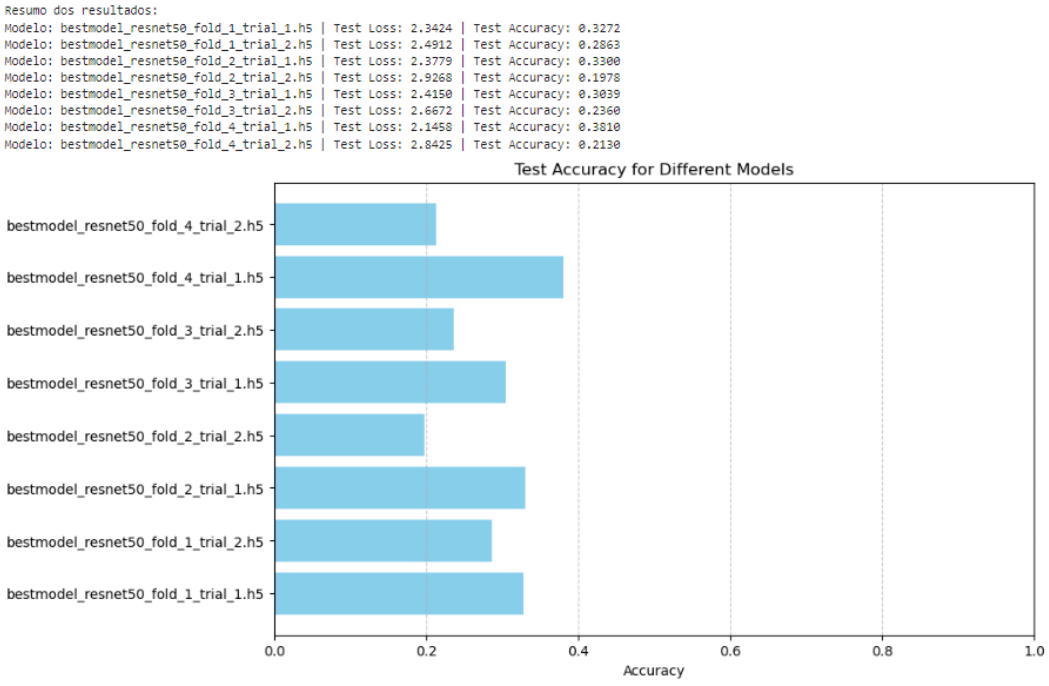


*Figura 9 Resumo dos resultados dos dados de treino com a arquitetura ResNet50*

Com os dados de teste, o valor da *accuracy* atingiu 38.10%, nenhum modelo é uma opção para o web API.

```
Test Loss: 2.145778179168701
Test Accuracy: 0.38100066781044006
```

### 5.1.3. MobileNetV2

A arquitetura MobileNetV2 também apresentou resultados baixos ao longos dos *folds* e *trials*, como se observa na Figura 10.

```
Resumo dos resultados:
Modelo: bestmodel_mobilenetv2_fold_1_trial_1.h5 | Test Loss: 2.4489 | Test Accuracy: 0.2924
Modelo: bestmodel_mobilenetv2_fold_1_trial_2.h5 | Test Loss: 2.4784 | Test Accuracy: 0.2836
Modelo: bestmodel_mobilenetv2_fold_2_trial_1.h5 | Test Loss: 2.4613 | Test Accuracy: 0.2823
Modelo: bestmodel_mobilenetv2_fold_2_trial_2.h5 | Test Loss: 2.5653 | Test Accuracy: 0.2701
Modelo: bestmodel_mobilenetv2_fold_3_trial_1.h5 | Test Loss: 2.4578 | Test Accuracy: 0.2789
Modelo: bestmodel_mobilenetv2_fold_3_trial_2.h5 | Test Loss: 2.5035 | Test Accuracy: 0.2792
Modelo: bestmodel_mobilenetv2_fold_4_trial_1.h5 | Test Loss: 2.6469 | Test Accuracy: 0.2529
Modelo: bestmodel_mobilenetv2_fold_4_trial_2.h5 | Test Loss: 2.6971 | Test Accuracy: 0.2492
```

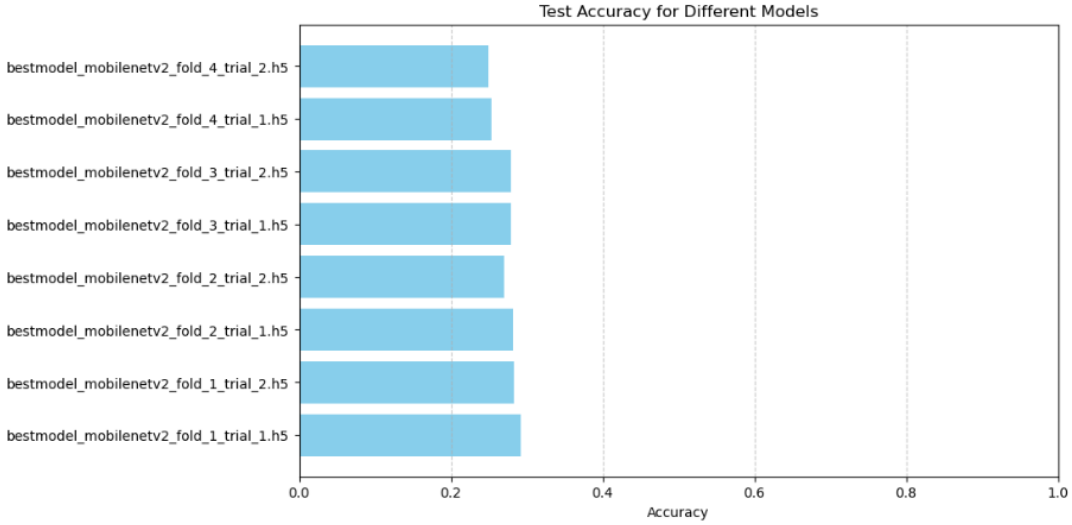


*Figura 10 Resumo dos resultados dos dados de treino com a arquitetura MobileNetV2*

Com os dados de teste, o valor da *accuracy* atingiu 25.28%, sendo a arquitetura com a pior performance registada, nenhum modelo é uma opção para o web API.

```
Test Loss: 2.6469132900238037
Test Accuracy: 0.25287356972694397
```

### 5.1.4. InceptionV3

A arquitetura InceptionV3 apresentou resultados não muito variados e os mais altos registados, ao longos dos *folds* e *trials*. Com base na validação cruzada, como se observa na Figura 11, a melhor combinação de Hiper parâmetros verifica-se no *fold* 4 e *trial* 1.

```
Resumo dos resultados:
Modelo: bestmodel_inceptionv3_fold_1_trial_1.h5 | Test Loss: 0.9209 | Test Accuracy: 0.7265
Modelo: bestmodel_inceptionv3_fold_1_trial_2.h5 | Test Loss: 0.9241 | Test Accuracy: 0.7170
Modelo: bestmodel_inceptionv3_fold_2_trial_1.h5 | Test Loss: 0.9510 | Test Accuracy: 0.7252
Modelo: bestmodel_inceptionv3_fold_2_trial_2.h5 | Test Loss: 0.9964 | Test Accuracy: 0.6853
Modelo: bestmodel_inceptionv3_fold_3_trial_1.h5 | Test Loss: 0.9877 | Test Accuracy: 0.7099
Modelo: bestmodel_inceptionv3_fold_3_trial_2.h5 | Test Loss: 1.0280 | Test Accuracy: 0.6924
Modelo: bestmodel_inceptionv3_fold_4_trial_1.h5 | Test Loss: 0.8854 | Test Accuracy: 0.7319
```

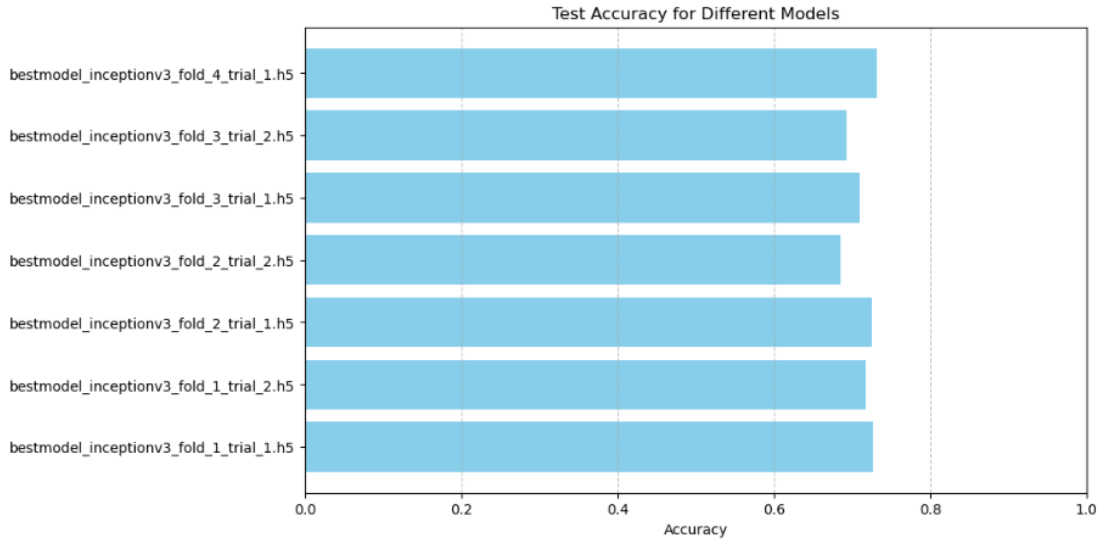


*Figura 11 Resumo dos resultados dos dados de treino com a arquitetura InceptionV3*

*Tabela 5 Hiper parâmetros testados com InceptionV3*

| fold | trial | units | dropout | learning_rate |
|---|---|---|---|---|
| **1** | **1** | 384 | 0.3 | 0.0002296904647915647 |
| **1** | **2** | 192 | 0.2 | 0.0005703556236244591 |
| **2** | **1** | 256 | 0.2 | 0.00024908165299495483 |
| **2** | **2** | 512 | 0.3 | 0.0020571513013852606 |
| **3** | **1** | 448 | 0.4 | 0.00015963096475777054 |
| **3** | **2** | 64 | 0.2 | 0.001264847732375066 |
| **4** | **1** | **256** | **0.3** | **0.00047875503773115347** |
| - | - | - | - | - |

Com os dados de teste, o valor da *accuracy* deu 73.19%, sendo a arquitetura a melhor performance testada, o modelo do *fold* 4 e *trial* 1 será utilizado no web API.

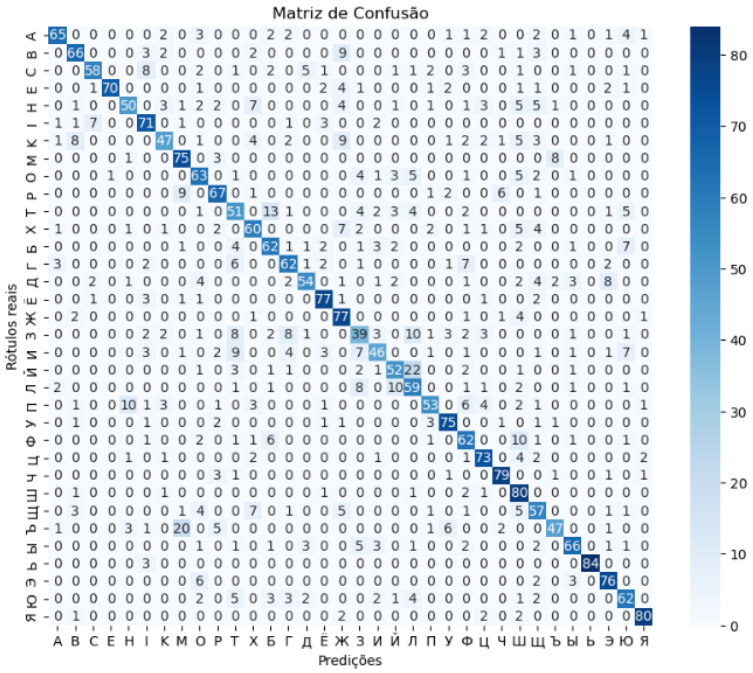


*Figura 12 Matriz de Confusão da arquitetura InceptionV3*

```
Test Loss: 0.8853622674942017
Test Accuracy: 0.7319134473800659
```

## 5.2. Tempo de execução

A Tabela 6 mostra o tempo de execução para 25 *epoch,* de cada arquitetura. A arquitetura MobileNetV2, tem o menor tempo de execução, no entanto apresenta os piores resultados com os dados de treino e teste. Por outro lado, a arquitetura InceptionV3 apresenta o maior tempo de execução, porém alcança o melhor resultado.

*Tabela 6 Tempo de Execução, por arquitetura*

| ***Arquitetura*** | ***Tempo*** |
|---|---|
| *VGG16* | Tempo total de execução: 11158.04 segundos |
| *ResNet50* | Tempo total de execução: 15686.70 segundos |
| *MobileNetV2* | Tempo total de execução: 10842.33 segundos |
| *InceptionV3* | Tempo total de execução: 24318.16 segundos |

## 5.3. Limitações

Os modelos foram treinados num tamanho de imagem reduzido para equilibrar o tempo de treino e o uso de recursos computacionais. As imagens foram redimensionadas para 32x32 pixels e 75x75 pixels, o que pode ter causado a perda de detalhes importantes. No entanto, uma resolução maior poderia melhorar a precisão, mas exigiria mais tempo e capacidade de processamento. Como o modelo foi treinado com imagens pequenas, ele pode ter dificuldade em generalizar bem para novas imagens com mais detalhes ou diferentes resoluções, o que pode impactar seu desempenho no reconhecimento de letras no web API.

# Cap. 6: Conclusões

O presente trabalho teve como objetivo aplicar aprendizagem por transferência ao *dataset* CoMNIST, utilizando imagens de letras do alfabeto cirílico organizadas em 34 classes. O projeto utilizou arquiteturas pré-treinadas, tais como VGG16, ResNet50, MobileNetV2 e InceptionV3, com foco na adaptação dessas redes para um *dataset* menor. A validação cruzada com 4 *folds* e o uso do método *Random Search* permitiram otimizar Hiper parâmetros, como número de unidades nas camadas densas, taxa de *dropout* e taxa de aprendizado.

Os experimentos revelaram diferenças significativas no desempenho entre as arquiteturas testadas:

- InceptionV3 apresentou os melhores resultados, alcançando uma *accuracy* de 73,19% no conjunto de teste. A sua performance superior a torna a mais adequada para reconhecimento de letras cirílicas no cenário proposto;
- VGG16 também obteve resultados satisfatórios, com 70,08% de *accuracy*, mostrando estabilidade nos diferentes *folds* e *trials*.
- Em contrapartida, ResNet50 e MobileNetV2 apresentaram desempenhos insatisfatórios, com 38,10% e 25,28% de *accuracy*, respetivamente, não sendo ideal para a aplicação final.

Embora MobileNetV2 tenha apresentado o menor tempo de execução, os seus resultados foram os piores. InceptionV3, apesar de um tempo de execução maior, mostrou um equilíbrio entre precisão e desempenho, destacando-se como a arquitetura mais eficiente do estudo.

Durante o projeto, o redimensionamento das imagens para 32x32 e 75x75 pixels foi necessário para otimizar recursos computacionais, mas isso pode ter resultado em perda de detalhes importantes, limitando a capacidade dos modelos de generalizar para novas imagens. O uso de resoluções maiores poderia potencialmente melhorar o desempenho, mas exigiria maior tempo de processamento e recursos computacionais.

# Bibliografia

Keras. (n.d.). *MobileNet, MobileNetV2, and MobileNetV3*. Keras Documentation. Retrieved December 12, 2024, from https://keras.io/api/applications/mobilenet/

Keras Team. (n.d.). *ResNet and ResNetV2*. Keras Applications API. Retrieved December 12, 2024, from https://keras.io/api/applications/resnet/

Keras Team. (n.d.). *VGG16 and VGG19*. Keras Applications API. Retrieved December 12, 2024, from https://keras.io/api/applications/vgg/

Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., & Wojna, Z. (2016). *Rethinking the inception architecture for computer vision*. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (pp. 2818-2826).