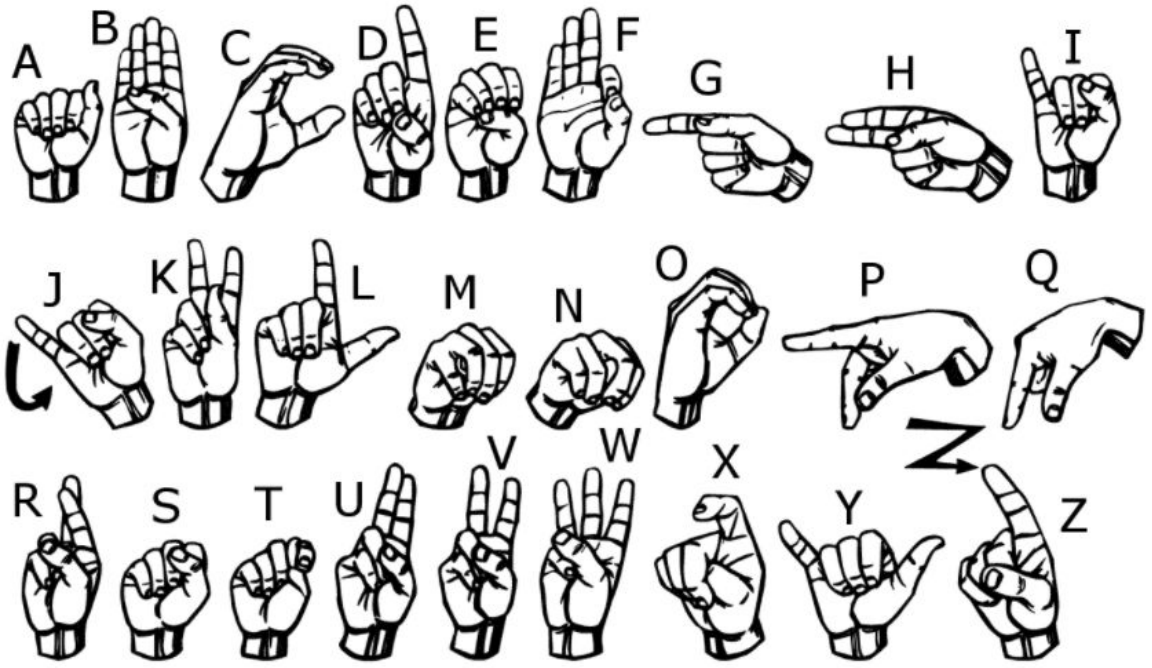
Relatório de Gráficos e Inteligentes

Reconhecimento de Hand Gesture Utilizando Dataset do Kaggle



Rio Grande

2019

Relatório escrito para as disciplinas de sistemas inteligentes e sistemas gráficos, ministradas pelos professores Silvia Botelho e Ricardo Nagel.

Alunos responsáveis:

- Carlos Alberto Nascimento - 85489.
- Willian Lemos - 87708.

Sumário

1 - Introdução e Informações Sobre o Dataset

1.1 - Conversão das imagens para CSV

O dataset de imagem MNIST original é uma referência popular para métodos de aprendizado de máquina baseados em imagem, mas os pesquisadores renovaram seus esforços para atualizá-lo e desenvolver substituições drop-in mais desafiadoras para a visão computacional e originais para aplicativos do mundo real. Um algoritmo robusto de reconhecimento visual pode fornecer não apenas novos benchmarks que desafiam os métodos modernos de aprendizado de máquina, como as Redes Neurais Convolucionais também poderiam ajudar pragmaticamente os surdos e com deficiência auditiva a se comunicarem melhor usando aplicativos de visão computacional.

O dataset de imagens original é composto por imagens de vários usuários repetindo gestos de mão com diferentes origens e pode ser encontrado **aqui**.

Para agilizar esta tarefa, essas imagens são convertidas para arquivos CSV com rótulos e valores de pixel em linhas únicas. Para criar novos dados, um pipeline de imagem foi usado com base no ImageMagick e incluiu recorte em apenas mãos, escala de cinza e redimensionamento, o que gerou um aumento significativo no tamanho e qualidade do dataset. O banco de dados de letras dos gestos das mãos da American Sign Language representa um problema de várias classes com 24 classes de letras (excluindo J e Z, que requerem movimento).

Figura 1: Dataset de Imagens de Cada Sinal de Libras

2 - Problema

Somos todos diferentes e o mundo fica melhor quando as pessoas entendem as outras em suas diferenças, mas isso é difícil.

Pessoas que utilizam da linguagem de sinais enfrentam problemas para expressar seus desejos a outros indivíduos que não possuem conhecimento a respeito dessa linguagem. Tivemos a inspiração portanto de utilizar um dataset de reconhecimento de gestos com a mão para auxiliar na inclusão e acesso desses indivíduos que demandam. Podemos citar que:

- Cerca de 80% dos surdos do mundo são analfabetos nas línguas escritas;
- A língua de sinais não é universal;
- Acessibilidade em Libras é obrigatória.

Portanto, esse trabalho tem como iniciativa solucionar esse empecilho de comunicação e pode ser utilizado inclusive para tradução da linguagem de sinais em qualquer outra linguagem desejada.

3 - Metodologia

3.1 - Importando as Bibliotecas Necessárias

```
import time
from time import perf_counter as timer
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import os
import keras
from keras.models import Sequential
from keras.layers import Dense, Conv2D, MaxPooling2D, Flatten, Dropout
```

Figura 2: importando de bibliotecas

3.2 - Importar as Imagens para Treino e Testes e Extração de dados

Nesta etapa, começamos a seleção e separação de dados para treinamento. Nesta fase, os dados dos pixels das imagens encontram-se em arquivos csv. Os dados de treinamento (27.455 casos) e os dados de teste (7172 casos) são aproximadamente metade do tamanho do MNIST padrão e são representados por cada linha de cabeçalho do rótulo. pixel1, pixel2 ... pixel784 representam uma única imagem de 28x28 pixels em escala de cinza com valores entre 0-255.

```
# Rota do dataset, usamos pelo próprio notebook da kaggle. Pra rodar localmente tem que mudar isso.
train = pd.read_csv('../input/sign-language-mnist/sign_mnist_train.csv')
test = pd.read_csv('../input/sign-language-mnist/sign_mnist_test.csv')

# Extraindo dados
labels = train['label'].values
unique_val = np.array(labels)
```

Figura 3: Importação de imagens e extração de dados

3.3 - Conversão de labels multi-classes para labels binárias

Como queremos predizer a que classe certa imagem pertence ou não, utilizamos a função LabelBinarizer(), a qual irá converter labels multi-classes para labels binárias (pertence ou não).

```
# Converte multi-class labels para labels binárias --> pertence ou não pertence a classe
from sklearn.preprocessing import LabelBinarizer
label_binrizer = LabelBinarizer()
labels = label_binrizer.fit_transform(labels)
```

Figura 4: Conversão de labels

3.4 - Preparação dos dados de imagem para que sejam treinados.

Nesta fase, começamos a fazer algumas transformações com nossos dados, para que possam ser usados com maior eficiência. Para isso, usamos a função da biblioteca pandas que seleciona um data frame dos nossos rótulos sem a parte selecionada, que foi removida. Após isso, normalizamos as imagens dividindo cada entrada por 255, pois os valores variam de 0 a 255.

```
# pandas.drop seleciona um dataframe de index/column labels sem a parte removida
# --> separa dados de treino para trabalhar com o resto dos dados
train.drop('label', axis = 1, inplace = True)
images = train.values
plt.style.use('grayscale')
images =  images/255 # Os valores variam de 0 a 255, então para normalizar, dividimos cada entrada por 255.
```

Figura 5: Preparação de dados para treinamento

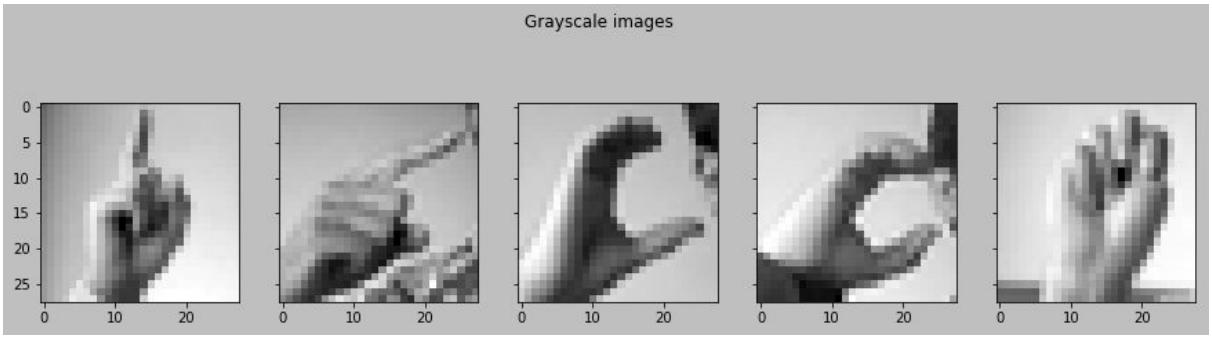


Figura 6: Imagens em Grayscale

3.5 - Separação de dados para treino e testes

Para validação durante o model fitting, precisamos dividir nosso conjunto de dados em duas partes, treino e teste. Com 70% dedicados para teste para que tenhamos uma boa acurácia mas que também tenhamos dados suficientes para validar nossos métodos.

```
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(images, labels, test_size = 0.3, stratify = labels, random_state = 7)

x_train = x_train.reshape(x_train.shape[0], 28, 28, 1)
x_test = x_test.reshape(x_test.shape[0], 28, 28, 1)
```

Figura 7: Separação de dados para treino e teste

3.6 - Definição da rede neural convolucional

Esta é a etapa onde definimos as propriedades da nossa rede neural convolucional. Neste caso utilizamos o modelo sequencial, onde os layers são empilhados linearmente. Após isso, adicionamos a primeira camada convolucional, com 64 filtros, modo de ativação relu e input baseado na quantidade de pixels de entrada, que no caso são 28*28.

O método de ativação Relu é muito utilizado nestes casos, por se tratar de uma função não linear. Também utilizamos Dropouts que desligam determinadas áreas da rede neural no processo de treinamento, o que reduz sua sensibilidade a pequenas alterações. Também utilizamos Pooling no final de cada layer para a redução da dimensionalidade/feature maps. Para o output layer utilizamos a função de ativação Softmax, já que possuímos 24 classes e o Softmax é uma boa alternativa para casos com distribuição multinomial.

```
model = Sequential() # Model onde os layers são empilhados sequencialmente.

# Layer com 64 _ e modo de ativação relu, que tem como entrada 28*28 pixeis da imagem.
model.add(Conv2D(64, kernel_size=(4,4), activation = 'relu', input_shape=(28, 28 ,1), padding='same' ))
model.add(Dropout(0.4)) # Descarta unidades aleatoriamente, ajuda a evitar overfitting
model.add(MaxPooling2D(pool_size = (2, 2)))

model.add(Conv2D(64, kernel_size = (4, 4), activation = 'relu', padding='same' ))
model.add(Dropout(0.4))
model.add(MaxPooling2D(pool_size = (2, 2)))

model.add(Conv2D(64, kernel_size = (3, 3), activation = 'relu'))
model.add(Dropout(0.4))
model.add(MaxPooling2D(pool_size = (2, 2)))

model.add(Flatten()) # Nivela a entrada
model.add(Dense(128, activation = 'relu')) # Camada NN regularmente conectada densamente.
model.add(Dense(num_classes, activation = 'softmax'))
model.compile(loss = keras.losses.categorical_crossentropy, optimizer='nadam',
              metrics=['accuracy'])
```

Figura 8: Definição de layers

3.7 - Ajuste de imagens

Nesta etapa é feito apenas alguns ajustes sobre as propriedades das imagens, preparando-as durante o model fitting.

```
# Aumentar a imagem durante o model fitting.
from keras.preprocessing.image import ImageDataGenerator
train_datagen = ImageDataGenerator(shear_range = 0.25,
                                   zoom_range = 0.15,
                                   rotation_range = 15,
                                   brightness_range = [0.15, 1.15],
                                   width_shift_range = [-2,-1, 0, +1, +2],
                                   height_shift_range = [ -1, 0, +1],
                                   fill_mode = 'reflect')
test_datagen = ImageDataGenerator()
```

Figura 9: Ajuste de imagens

3.8 - Preparação para validação

Vamos validar com os dados do teste. No início, ele deve ser pré-processado da mesma maneira que os nossos dados no model fitting. Isso significa que devemos remover sua coluna de label, dividir todos os valores por 225 e as linhas devem ser remodeladas como matrizes quadradas.

```
history = model.fit(x_train, y_train, validation_data = (x_test, y_test), epochs=epochs, batch_size=batch_size) #Rodando

# Validação dos dados.
test_labels = test['label']
test.drop('label', axis = 1, inplace = True)
test_images = test.values/255
test_images = np.array([np.reshape(i, (28, 28)) for i in test_images])
test_images = np.array([i.flatten() for i in test_images])
test_labels = label_binrizer.fit_transform(test_labels)
test_images = test_images.reshape(test_images.shape[0], 28, 28, 1)
test_images.shape
```

Figura 10: Preparação para validação

3.9 - Validação

Por fim realizamos a validação do nosso modelo. Para verificar se nossa rede 'aprendeu' e não 'apenas memorizou', a colocamos para predizer com a porção de imagens que não foi utilizada para treinamento, testando sua capacidade de predição com dados estrangeiros e validando a acurácia do nosso método.

```
# Acurácia de predição
y_pred = model.predict(test_images)
from sklearn.metrics import accuracy_score
y_pred = y_pred.round()
print("Acuracia media:" +str(accuracy_score(test_labels, y_pred)))
# accuracy_score(test_labels, y_pred)
```

Figura 11: Validação

4 - Resultados

Como pode ser visto na Figura 12 abaixo, conseguimos nos resultados uma precisão em média maior que 0.94, chegando a 1 em alguns epochs, o que indica que a nossa rede de treinamento foi eficiente.

```
Train on 19218 samples, validate on 8237 samples
Epoch 1/15
19218/19218 [==============================] - 51s 3ms/step - loss: 1.9725 - accuracy: 0.3841 - val_loss: 1.1246 - val_accuracy: 0.8321
Epoch 2/15
19218/19218 [==============================] - 49s 3ms/step - loss: 0.4065 - accuracy: 0.8558 - val_loss: 0.4553 - val_accuracy: 0.9745
Epoch 3/15
19218/19218 [==============================] - 48s 3ms/step - loss: 0.1510 - accuracy: 0.9493 - val_loss: 0.2838 - val_accuracy: 0.9798
Epoch 4/15
19218/19218 [==============================] - 49s 3ms/step - loss: 0.0880 - accuracy: 0.9717 - val_loss: 0.1294 - val_accuracy: 0.9998
Epoch 5/15
19218/19218 [==============================] - 50s 3ms/step - loss: 0.0535 - accuracy: 0.9820 - val_loss: 0.0997 - val_accuracy: 0.9931
Epoch 6/15
19218/19218 [==============================] - 50s 3ms/step - loss: 0.0477 - accuracy: 0.9842 - val_loss: 0.0620 - val_accuracy: 0.9994
Epoch 7/15
19218/19218 [==============================] - 50s 3ms/step - loss: 0.0387 - accuracy: 0.9867 - val_loss: 0.0660 - val_accuracy: 0.9944
Epoch 8/15
19218/19218 [==============================] - 49s 3ms/step - loss: 0.0331 - accuracy: 0.9890 - val_loss: 0.0518 - val_accuracy: 0.9994
Epoch 9/15
19218/19218 [==============================] - 50s 3ms/step - loss: 0.0313 - accuracy: 0.9898 - val_loss: 0.0328 - val_accuracy: 1.0000
Epoch 10/15
19218/19218 [==============================] - 50s 3ms/step - loss: 0.0235 - accuracy: 0.9920 - val_loss: 0.0332 - val_accuracy: 0.9994
Epoch 11/15
19218/19218 [==============================] - 50s 3ms/step - loss: 0.0232 - accuracy: 0.9926 - val_loss: 0.0243 - val_accuracy: 1.0000
Epoch 12/15
19218/19218 [==============================] - 50s 3ms/step - loss: 0.0208 - accuracy: 0.9933 - val_loss: 0.0410 - val_accuracy: 0.9981
Epoch 13/15
19218/19218 [==============================] - 49s 3ms/step - loss: 0.0229 - accuracy: 0.9922 - val_loss: 0.0268 - val_accuracy: 1.0000
Epoch 14/15
19218/19218 [==============================] - 49s 3ms/step - loss: 0.0260 - accuracy: 0.9920 - val_loss: 0.0183 - val_accuracy: 1.0000
Epoch 15/15
19218/19218 [==============================] - 50s 3ms/step - loss: 0.0256 - accuracy: 0.9913 - val_loss: 0.0326 - val_accuracy: 0.9941
Acuracia media:0.9411600669269381
```

Figura 12: Treinamento com as 15 epochs e valor de precisão