Codificación No Lineal para el Análisis de Imágenes en PySpark

October 29, 2024

Godinez Bravo Diego

Centro de Investigación en Matemáticas

Maestría en Cómputo Estadístico

[2]:
```
import pyspark
import numpy as np
import matplotlib.pyplot as plt
from pyspark.sql import SparkSession
from pyspark.ml.linalg import Vectors, VectorUDT
from pyspark.sql import functions as F
from pyspark.sql.functions import udf
from tensorflow.keras import datasets, layers, models
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
from pyspark.ml.classification import MultilayerPerceptronClassifier
from pyspark.sql.types import StructField, StructType, ArrayType, IntegerType,␣
↪FloatType # import libraries
```

El conjunto de datos **CIFAR-10** consta de $60,000$ imágenes a color de 32×32 píxeles, distribuidas en 10 clases, con $6,000$ imágenes por clase. El conjunto se divide en $50,000$ imágenes de entrenamiento y $10,000$ imágenes de prueba.

[3]:
```
from tensorflow.keras.datasets import cifar10 # load the CIFAR-10 dataset from␣
↪Keras API
```

[4]:
```
spark = SparkSession.builder\
        .appName('MLP-CIFAR10')\
        .config('spark.driver.memory', '12g')\
        .getOrCreate() # create a Spark session
```

[5]:
```
spark
```

[5]:
```
<pyspark.sql.session.SparkSession at 0x7fb603e72620>
```

[6]:
```
(X_train, y_train), (X_test, y_test) = datasets.cifar10.load_data() # download␣
↪CIFAR10 data
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170498071/170498071             5s
0us/step
```

Cada imagen se representa como un arreglo tridimensional de tamaño $32 \times 32 \times 3$, donde la 3ra dimensión corresponde a los canales RGB (Rojo, Verde y Azul), los cuales codifican la información de color de cada píxel (i.e., cada píxel tiene tres valores, uno para cada canal de color).

[7]:
```
X_train.shape # each image has 32 by 32 pixels and 3 RGB channels; 50,000␣
↪training images
```

[7]:
```
(50000, 32, 32, 3)
```

[8]:
```
X_test.shape # each image has 32 by 32 pixels and 3 RGB channels; 10,000 test␣
↪images
```

[8]:
```
(10000, 32, 32, 3)
```

Es crucial verificar la presencia de valores nulos en el conjunto de datos, ya que pueden provocar fallas en la implementación de modelos de machine learning, dado que estos no pueden procesar datos con valores nulos.

[9]:
```
print("NaN values in training set:", np.isnan(X_train).any() or np.
↪isnan(y_train).any()) # check for NaN values in training set
```

```
NaN values in training set: False
```

[10]:
```
print("NaN values in test set:", np.isnan(X_test).any() or np.isnan(y_test).
↪any()) # check for NaN values in test set
```

```
NaN values in test set: False
```

[11]:
```
type(X_train) # data stored in a np array
```

[11]:
```
numpy.ndarray
```

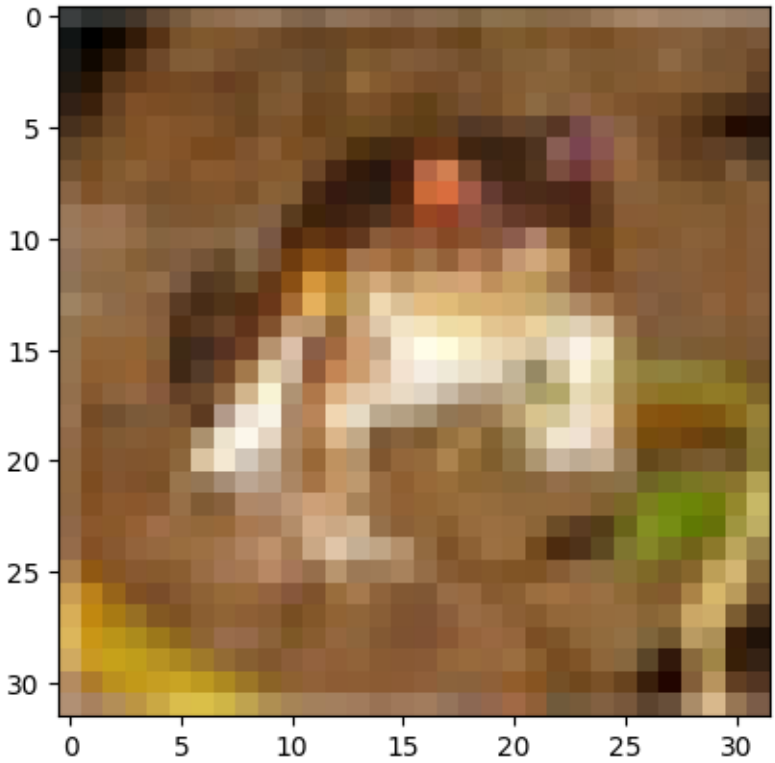
Valores de los pixeles se encuentran en el rango de $[0, 255]$.

[12]:
```
X_train[0] # numpy array (32, 32, 3)
```

[12]:
```
array([[[ 59,  62,  63],
        [ 43,  46,  45],
        [ 50,  48,  43],
        …,
        [158, 132, 108],
        [152, 125, 102],
        [148, 124, 103]],

       [[ 16,  20,  20],
        [  0,   0,   0],
        [ 18,   8,   0],
```

```
       …,
       [123,  88,  55],
       [119,  83,  50],
       [122,  87,  57]],

      [[ 25,  24,  21],
       [ 16,   7,   0],
       [ 49,  27,   8],
       …,
       [118,  84,  50],
       [120,  84,  50],
       [109,  73,  42]],

      …,

      [[208, 170,  96],
       [201, 153,  34],
       [198, 161,  26],
       …,
       [160, 133,  70],
       [ 56,  31,   7],
       [ 53,  34,  20]],

      [[180, 139,  96],
       [173, 123,  42],
       [186, 144,  30],
       …,
       [184, 148,  94],
       [ 97,  62,  34],
       [ 83,  53,  34]],

      [[177, 144, 116],
       [168, 129,  94],
       [179, 142,  87],
       …,
       [216, 184, 140],
       [151, 118,  84],
       [123,  92,  72]]], dtype=uint8)
```

[13]:
```
img = plt.imshow(X_train[0])
```



[14]:
```
y_train[0] # according to Keras documentation the label 6 correspond to frog␣
↪category
```

[14]:
```
array([6], dtype=uint8)
```

Para evitar problemas de saturación de la memoria RAM, se decidió reducir el número de imágenes, ya que la ejecución del modelo se interrumpía al utilizar el conjunto de datos completo.

[15]:
```
X_train = X_train[:30000]
y_train = y_train[:30000]
```

[16]:
```
X_train.shape # verify the reduction of the no. of training images
```

[16]:
```
(30000, 32, 32, 3)
```

[17]:
```
X_test = X_test[:5000]
y_test = y_test[:5000]
```

[18]:
```
X_test.shape # verify the reduction of the no. of testing images
```

[18]:
```
(5000, 32, 32, 3)
```

Dado que las funciones de activación (e.g., sigmoide, softmax, etc.) generan valores en el rango de $[0, 1]$, normalizamos los valores de los píxeles en este rango para garantizar que las entradas de estas

funciones estén en una escala compatible. Esto mejora el proceso de convergencia de los algoritmos de optimización y evita la saturación de las funciones de activación.

[19]:
```
X_train = X_train.astype("float32") / 255
X_test = X_test.astype("float32") / 255 # normalize the pixel values dividing␣
↪by 255
```

[20]:
```
X_train[0][0]
```

[20]:
```
array([[0.23137255, 0.24313726, 0.24705882],
       [0.16862746, 0.18039216, 0.1764706 ],
       [0.19607843, 0.1882353 , 0.16862746],
       [0.26666668, 0.21176471, 0.16470589],
       [0.38431373, 0.28627452, 0.20392157],
       [0.46666667, 0.35686275, 0.24705882],
       [0.54509807, 0.41960785, 0.29411766],
       [0.5686275 , 0.43137255, 0.3137255 ],
       [0.58431375, 0.45882353, 0.34901962],
       [0.58431375, 0.47058824, 0.3647059 ],
       [0.5137255 , 0.40392157, 0.3019608 ],
       [0.49019608, 0.3882353 , 0.29803923],
       [0.5568628 , 0.4509804 , 0.35686275],
       [0.5647059 , 0.4392157 , 0.3372549 ],
       [0.5372549 , 0.4117647 , 0.30980393],
       [0.5058824 , 0.38039216, 0.2784314 ],
       [0.5372549 , 0.41568628, 0.30980393],
       [0.5254902 , 0.41568628, 0.29803923],
       [0.4862745 , 0.38039216, 0.2509804 ],
       [0.54509807, 0.44313726, 0.30588236],
       [0.54509807, 0.4392157 , 0.29411766],
       [0.52156866, 0.4117647 , 0.27058825],
       [0.53333336, 0.4117647 , 0.2901961 ],
       [0.54509807, 0.42352942, 0.3019608 ],
       [0.59607846, 0.47058824, 0.34901962],
       [0.6392157 , 0.5137255 , 0.39215687],
       [0.65882355, 0.53333336, 0.42352942],
       [0.62352943, 0.5058824 , 0.4       ],
       [0.61960787, 0.50980395, 0.40784314],
       [0.61960787, 0.5176471 , 0.42352942],
       [0.59607846, 0.49019608, 0.4       ],
       [0.5803922 , 0.4862745 , 0.40392157]], dtype=float32)
```

Para mantener la consistencia de los datos, se define el esquema de manera explícita. Esto se hace para evitar comportamientos inesperados por parte de Spark.

[21]:
```
schema = StructType([
    StructField("label", FloatType(), False),
    StructField("features", ArrayType(FloatType()), False)
```

```
]) # to maintain data consistency, the schema is explicitly defined
```

Definimos una función para procesar el conjunto de datos por *batches* con el objetivo de reducir el uso de memoria, evitando almacenar todo el conjunto de datos inmediatamente. Durante el procesamiento, solo un batch es almacenado en la memoria a la vez, una vez que es procesado se descarta, liberando así el uso de memoria para el siguiente batch.

Este enfoque reduce de manera significativa el uso general de memoria durante todo el procesamiento.

[1]:
```
def process_data(X_data, y_data, batch_size = 10000): # features, labels, and␣
↪batch size as input parameters
    no_images = X_data.shape[0] # total no. of images
    for i in range(0, no_images, batch_size): # iterate through the entire␣
↪dataset in increments defined by the batch size
        X_batch = X_data[i:i + batch_size]
        y_batch = y_data[i:i + batch_size]

        X_batch_flattened = X_batch.reshape((X_batch.shape[0], -1)) # convert␣
↪the 3-dimensional data into a 1-dimensional array keeping the 32*32*3 pixel␣
↪values

        labels_and_features = [(float(y), [float(val) for val in x]) for x, y␣
↪in zip(X_batch_flattened, y_batch.flatten())] # list of tuples containing␣
↪both labels and features
        spark_df = spark.createDataFrame(labels_and_features, schema = schema) ␣
↪# specify schema explicitly

        yield spark_df # return each spark dataframe immediately
```

Utilizamos `yield` cuando queremos iterar sobre una secuencia, pero no queremos almacenar toda la secuencia en memoria. Devuelve cada batch inmediatamente a medida que se obtiene, en lugar de esperar a que se procesen todos los batches para devolverlos en una sola colección grande. Cada vez que se llama a `yield` se devuelve el batch actual y luego se pausa. Cuando la función se reanuda para la siguiente iteración, sobrescribe el batch anterior con el nuevo.

Esto mantiene el uso de memoria bajo al almacenar solo un batch a la vez en memoria y descartarlo una vez que se ha procesado, siendo reemplazado por el siguiente batch en la iteración subsiguiente.

0.0.1 Procesamiento del Conjunto de Entrenamiento

[22]:
```
batches = [] # empty list to batches
for batch_df in process_data(X_train, y_train, batch_size = 10000): # iterate␣
↪through the batches produced by the function
    print(batch_df) # total no. of batches processed
    batches.append(batch_df) # combine all batches into a single list
```

```
DataFrame[label: float, features: array<float>]
DataFrame[label: float, features: array<float>]
```

```
DataFrame[label: float, features: array<float>]
```

[23]:
```
train_df = batches[0]
for spark_dataframe in batches[1:]:
    train_df = train_df.union(spark_dataframe) # combines all the spark␣
 ↪dataframes (batches) into a single spark dataframe
```

[24]:
```
type(train_df) # spark dataframe
```

[24]:
```
pyspark.sql.dataframe.DataFrame
```

[25]:
```
train_df.printSchema() # schema of the dataframe
```

```
root
 |-- label: float (nullable = false)
 |-- features: array (nullable = false)
 |    |-- element: float (containsNull = true)
```

Al verificar el esquema del *DataFrame* generado, observamos que la columna **features** se define como un arreglo de valores de tipo *float*. Sin embargo, los pipelines de **MLlib** requieren que las características estén en formato de tipo *vector*. Esto es fundamental para la correcta integración de los procesos de preprocesamiento, transformación y ajuste de modelos dentro de Spark.

[26]:
```
array_to_vector = udf(lambda array: Vectors.dense(array), VectorUDT()) # create␣
 ↪a user defined function to convert arrays into spark DenseVector objects
```

[28]:
```
train_df = train_df.withColumn("features",␣
 ↪array_to_vector(train_df["features"])) # apply the UDF to features column
```

[29]:
```
train_df.printSchema() # schema updated
```

```
root
 |-- label: float (nullable = false)
 |-- features: vector (nullable = true)
```

[30]:
```
train_df.show(2) # data preview
```

```
+-----+--------------------+
|label|            features|
+-----+--------------------+
|  6.0|[0.23137255012989…|
|  9.0|[0.60392159223556…|
+-----+--------------------+
only showing top 2 rows
```

0.0.2 Procesamiento del Conjunto de Prueba

[31]:
```
batches = [] # empty list to batches
for batch_df in process_data(X_test, y_test, batch_size = 2500): # iterate␣
 ↪through the batches produced by the function
    print(batch_df) # total no. of batches processed
    batches.append(batch_df) # combine all batches into a single list
```

```
DataFrame[label: float, features: array<float>]
DataFrame[label: float, features: array<float>]
```

[32]:
```
test_df = batches[0]
for spark_dataframe in batches[1:]:
    test_df = test_df.union(spark_dataframe) # combines all the spark␣
 ↪dataframes (batches) into a single spark dataframe
```

[33]:
```
type(test_df) # spark dataframe
```

[33]:
```
pyspark.sql.dataframe.DataFrame
```

[34]:
```
test_df.printSchema() # schema of the dataframe
```

```
root
 |-- label: float (nullable = false)
 |-- features: array (nullable = false)
 |    |-- element: float (containsNull = true)
```

[35]:
```
test_df = test_df.withColumn("features", array_to_vector(test_df["features"]))␣
 ↪# apply the UDF to features column
```

[36]:
```
test_df.printSchema() # schema updated
```

```
root
 |-- label: float (nullable = false)
 |-- features: vector (nullable = true)
```

[37]:
```
test_df.show(2) # data preview
```

```
+-----+--------------------+
|label|            features|
+-----+--------------------+
|  3.0|[0.61960786581039…|
|  8.0|[0.92156863212585…|
+-----+--------------------+
only showing top 2 rows
```

0.1 Clasificador Perceptrón Multicapa

El clasificador perceptrón multicapa es un clasificador basado en la red neuronal *feedforward*, un tipo de red neuronal artificial utilizado en aplicaciones de ML y reconocimiento de patrones. Su arquitectura se compone de multiples capas de nodos, tambien llamados *hidden units* o neuronas, las cuales estan completamente conectadas dentro de la red. Es decir, cada nodo en una capa anterior se encuentra conectado a cada nodo en la capa siguiente. Las salidas de cada nodo en una capa anterior L funcioanan como entrada para los nodos en la capa $L + 1$.

El clasificador perceptrón multicapa es un modelo basado en una red neuronal *feedforward*, un tipo de red neuronal artificial ampliamente utilizado en aplicaciones de ML y reconocimiento de patrones. Su arquitectura consta de múltiples capas de nodos, conocidos también como unidades ocultas o neuronas, que están completamente conectados dentro de la red. Esto significa que cada nodo en una capa anterior se conecta a cada nodo en la capa siguiente, de modo que las salidas de los nodos en una capa L funcionan como entradas para los nodos en la capa $L + 1$.

Cada nodo en las capas ocultas transforma los valores de la capa anterior mediante una combinacion lineal:

$$w_1x_1 + w_2x_2 + ... + w_mx_m$$

seguida de una función de activación no lineal $g()$.

Finalmente, la capa de salida recibe los valores de la última capa oculta y los convierte en los valores de salida.

Considerando un modelo con $K + 1$ capas, este puede expresarse como:

$$y(x) = f_k(...f_2(w_2^T f_1(w_1^T x + b_1) + b_2)... + b_k)$$

donde los nodos en las capas intermedias utilizan la función sigmoide definida como:

$$f(z_i) = \frac{1}{1 + e^{-z_i}},$$

y los nodos en la capa de salida utilizan la función *softmax*:

$$f(z_i) = \frac{e^{z_i}}{\sum_{k=1}^{N} e^{z_k}}$$

[42]:
```
train_df.persist() # persisting a dataframe can significantly speed up␣
↪processing time for iterative algorithms or when multiple actions need to be␣
↪performed on the same dataframe
```

[42]:
```
DataFrame[label: float, features: vector]
```

En este caso, se define una capa de entrada con $32 \times 32 \times 3 = 3072$ nodos, dos capas ocultas con 512 y 128 nodos, respectivamente, y una capa de salida con 10 valores de salida.

[39]:
```
layers = [3072, 512, 128, 10] # no. of pixel values for input + two hidden␣
↪layers + 10 output classes
```

[41]:
```
mlp = MultilayerPerceptronClassifier(labelCol = "label", featuresCol =␣
↪"features",
                                     maxIter = 100, layers = layers, blockSize␣
↪= 64, seed = 614) # defining a multilayer perceptron classifier
fitted_model = mlp.fit(train_df) # fit the mlp model
```

0.1.1 Validación y Prueba del Modelo

Calcular las predicciones utilizando los parámetros aprendidos durante el proceso de entrenamiento del modelo.

[46]:
```
predictions = fitted_model.transform(test_df) # calculate predictions
```

[47]:
```
predictions.printSchema() # dataframe contains the same columns as before plus␣
↪additional prediction and probability columns
```

```
root
 |-- label: float (nullable = false)
 |-- features: vector (nullable = true)
 |-- rawPrediction: vector (nullable = true)
 |-- probability: vector (nullable = true)
 |-- prediction: double (nullable = false)
```

[48]:
```
predictions.show(2)
```

```
+-----+--------------------+--------------------+--------------------+----------
+
|label|            features|       rawPrediction|
probability|prediction|
+-----+--------------------+--------------------+--------------------+----------
+
|  3.0|[0.61960786581039…|[-0.0711078736062…|[0.04019910159770…|
3.0|
|  8.0|[0.92156863212585…|[1.83922448354245…|[0.07755381276072…|
8.0|
+-----+--------------------+--------------------+--------------------+----------
+
only showing top 2 rows
```

[51]:
```
predictions_labels = predictions.select("prediction", "label") # get the␣
↪predicted class for each image and the actual label
```

[56]:
```
predictions_labels.show(2)
```

```
+----------+-----+
|prediction|label|
+----------+-----+
|       3.0|  3.0|
|       8.0|  8.0|
+----------+-----+
only showing top 2 rows
```

Precisión del Modelo

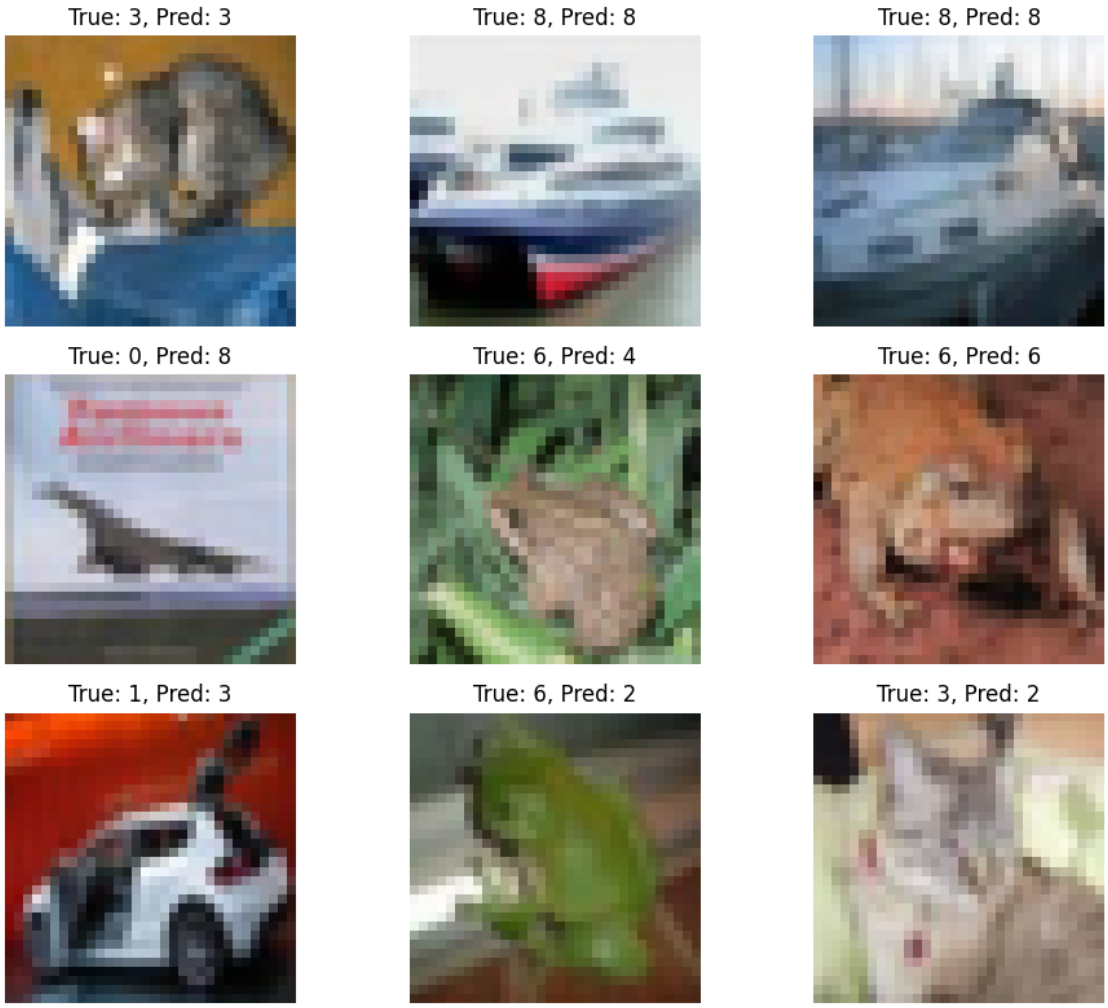
[58]:
```
evaluator = MulticlassClassificationEvaluator(metricName = "accuracy")
print("Test set accuracy = " + str(evaluator.evaluate(predictions_labels)))
```

```
Test set accuracy = 0.4384
```

0.1.2 Visualización de los Resultados

[ ]:
```
no_images = predictions_labels.limit(9).collect() # no. of images to visualize

images = []
labels = []
predictions = [] # empty lists to store images, labels and predictions
i = 0 # index variable to retrieve the images from the test set

for image in no_images:
    label = int(image.label) # true label
    predicted = int(image.prediction) # predicted label
    images.append(X_test[i]) # get the corresponding image from test set

    labels.append(label) # store label
    predictions.append(predicted) # store predicted label
    i += 1 # update the index
```

[91]:
```
fig, axes = plt.subplots(3, 3, figsize = (10, 8)) # subplot with 3 rows, 3␣
 ↪columns
axes = axes.flatten()

for ax, img, label, prediction in zip(axes, images, labels, predictions):
    ax.imshow(img) # show the image
    ax.set_title(f'True: {label}, Pred: {prediction}') # title with true label␣
 ↪and predicted label
    ax.axis('off')

for ax in axes[len(images):]:
    ax.axis('off') # hide any unused axes

plt.tight_layout()
plt.show()
```



[89]:
```
spark.stop() # stop spark session
```

0.2 Conclusión

Dado el valor de la precisión obtenido, podemos concluir que, en este caso, un clasificador perceptrón multicapa (MLP) no es adecuado para realizar la clasificación de estos datos. Si bien el modelo identifica ciertos patrones de manera eficaz, como en el caso de los barcos, puede pasar por alto patrones y características relevantes para las distintas clases dentro del conjunto de datos.

En este contexto, podríamos implementar una red neuronal convolucional (CNN) con el objetivo de detectar características más complejas, como bordes y texturas, que podrían ser ignoradas por el modelo MLP.

La importancia y aplicación de las transformaciones de datos en un espacio de representación de forma no lineal son fundamentales para mejorar la capacidad del modelo para capturar relaciones complejas. Estas transformaciones permiten que los modelos de machine learning (ML) sean más efectivos al abordar problemas que involucran características no lineales. La implementación de

modelos de ML en Spark a través de la biblioteca `MLlib`, facilita el procesamiento de grandes volúmenes de datos, optimizando el rendimiento y facilitando la escalabilidad de los modelos.

Maestría en Cómputo Estadístico
Noviembre 2024