UNIVERSIDAD NACIONAL DE SAN ANTONIO ABAD DEL CUSCO

ESCUELA PROFESIONAL INGENIERÍA INFORMÁTICA Y DE SISTEMAS

IF653AIN - Mineria de Datos

Red Neuronal para Clasificación de Imágenes de Números Escritos a Mano con PySpark

Docente:
MONTOYA-CUBAS-CARLOS FERNANDO

Autores:
Choque Bueno Fiorella Silvia
Condori Ccarhuarupay Bruno Moises
Huaman Guevara Alexander Javier
Huaraya Chara Bladimir
Luna Ccasani Charlie Joel
Tacusi Larota Jhon Edwin

13 de febrero de 2022

Índice

Índice de figuras

1. EXPLICACIÓN DEL MÉTODO

Para poder realizar el presente proyecto primeramente definimos la arquitectura de nuestra Red Neuronal por ejemplo, el número de capas ocultas, el número de neuronas por capa, la función de activación, la función de pérdida. Para evitar matemáticas complejas o una implementación modular compleja, Nuestro proyecto propone una red neuronal de 3 capas.

Nuestro modelo se muestra de la siguiente manera:

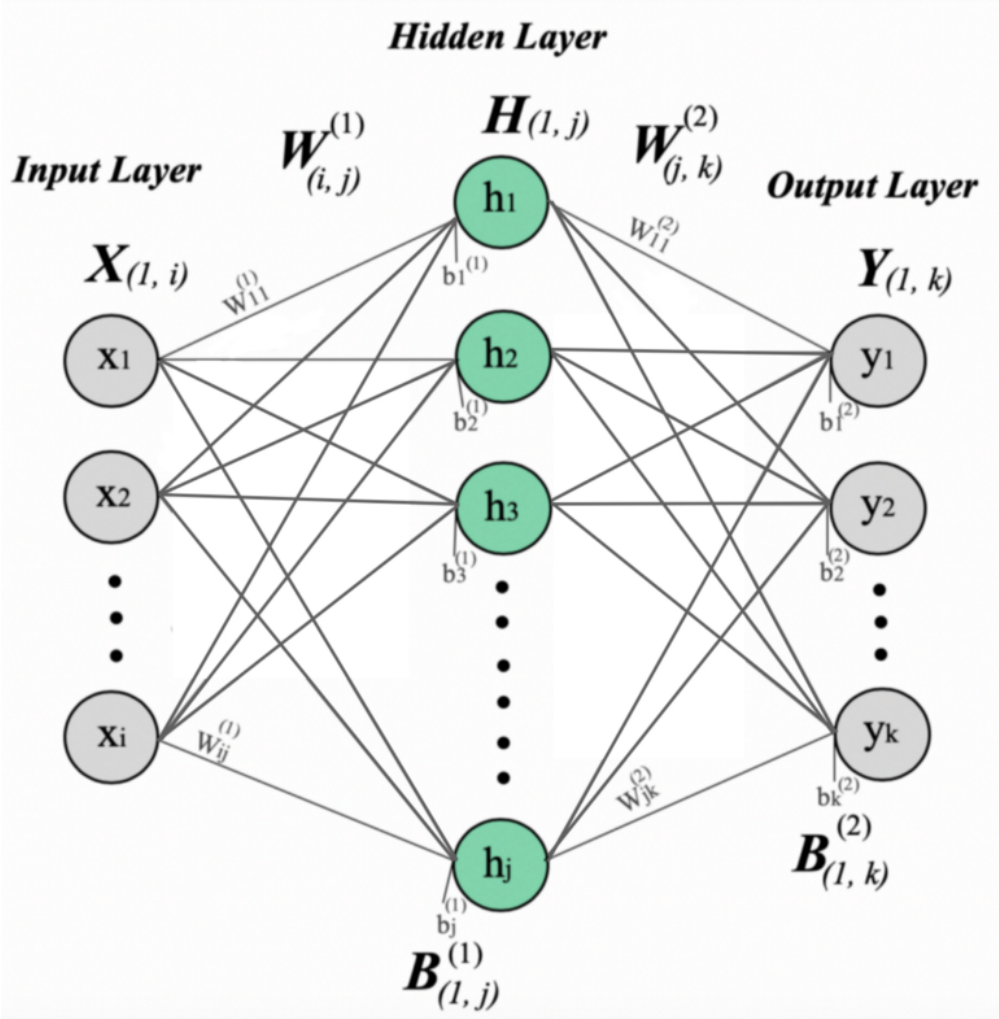


Figura 1: Arquitectura del modelo

Donde "X" es la capa de entrada con i neuronas, "H" es la capa oculta con j neuronas e "Y" es la capa de salida (con k neuronas). Este modelo tiene 4 parámetros: W(1), W(2), B(1), B(2).

Cómo usamos las imágenes de la base de datos MNIST como datos de entrada, tenemos que $i = 784$ imágenes (28x28), luego j el número de neuronas ocultas es una elección arbitraria, aquí podemos usar $j = 64$ y finalmente el número de neuronas de salida $k = 2$ ya que tenemos 2 números posibles para predecir 0 y 1.

función sigmoide

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Para activar las neuronas ocultas y de salida, utilizaremos la función de activación sigmoidea.

Una de las razones para utilizar la función sigmoide es por sus propiedades matemáticas, en nuestro caso, sus derivadas. Cuando más adelante la red neuronal haga backpropagation para aprender y actualizar los pesos, haremos uso de su derivada. En esta función puede ser expresada como productos de f y $1 - f$. Entonces $f'(t) = f(t)(1–f(t))$. Por ejemplo la función tangente y su derivada arcotangente se utilizan normalizadas, donde su pendiente en el origen es 1 y cumplen las propiedades.

$$MSE = \frac{2}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

Y finalmente, para calcular nuestro error de entrenamiento sobre un lote de datos de entrenamiento, usaremos una función de pérdida llamada error cuadrático medio MSE.

2. APLICACIÓN EN BIG DATA

2.1. *PySpark*

Spark es una solución poderosa para procesar grandes cantidades de datos. Permite distribuir el cómputo en una red de computadoras (a menudo llamada clúster). Spark facilita la implementación de algoritmos iterativos que analizan un conjunto de datos varias veces en un bucle. Spark es ampliamente utilizado en proyectos de aprendizaje automático.

Las bibliotecas famosas como *TensorFlow* o *Pytorch* generalmente se usan para construir redes neuronales. Uno de los beneficios de usar estas bibliotecas es el cómputo de GPU que acelera el entrenamiento al permitir el cómputo paralelo.

Las últimas versiones de *Spark* también permiten el uso de GPU, pero en este artículo, solo nos centraremos en el cálculo de la CPU (como la mayoría de las implementaciones temporales de redes neuronales) para mantenerlo simple. Este artículo propone una implementación con fines de aprendizaje que no se ajusta a las necesidades industriales.

2.2. Entrenamiento de redes neuronales con RDD

Aplicar las siguientes ecuaciones usando solo Numpy en la forma habitual de Python es bastante simple, solo requiere operaciones vectorizadas de Numpy, pero cuando se trata de distribuir el cálculo a varias máquinas, ya no podemos usar esta lógica.

2.3. Conjunto de datos distribuido resistente (RDD)

Los conjuntos de datos que utiliza Spark se almacenan mediante RDD (conjunto de datos distribuido resistente). Los RDD son çolecciones"de elementos particionados y distribuidos en los nodos del clúster. Con RDD, Spark realiza tareas iterativas e interactivas mientras mantiene la escalabilidad y la tolerancia a fallas del clúster.

Nuestro conjunto de datos MNIST generalmente se carga como RDD (para conjuntos de entrenamiento y prueba). Los RDD se estructuran mediante pares clave-valor.

2.4. MapReduce

Estos RDD se tratan mediante 2 operaciones principales: operaciones Map y Reduce.

2.4.1. Operaciones MAP

La operación Map desglosa el cálculo por lotes. El controlador del clúster enviará cada uno de estos lotes a diferentes computadoras en el clúster (según los recursos y la configuración del clúster). De esta manera, cada computadora en el clúster podrá calcular la propagación hacia adelante y hacia atrás en su porción de datos asignada.

2.4.2. Operaciones Reduce

Una vez que se ha calculado la propagación hacia adelante y hacia atrás en el clúster, es hora de agregar los resultados para obtener el costo promedio, la precisión promedio y los gradientes promedio del costo sobre los parámetros. Esta agregación se realiza en lotes de datos de entrenamiento y da como resultado una sola tupla de salida.

2.5. Dataset MNIST

El dataset MNIST (Modified National Institute of Standards and Technology) es el considerado "Hello World"de la visión artificial. Contiene un conjunto de entrenamiento de 60.000 imágenes de dígitos manuscritos (de 0 a 9), y otro conjunto de pruebas con 10.000 muestras adicionales.

Las muestras incluidas en el conjunto de entrenamiento fueron el resultado de escanear dígitos manuscritos de 250 personas. El dataset de pruebas contiene dígitos escaneados de otras 250 personas diferentes, lo que permite asegurar que los modelos obtenidos son capaces de interpretar dígitos incluso de personas no involucradas en la generación de los datos de entrenamiento.

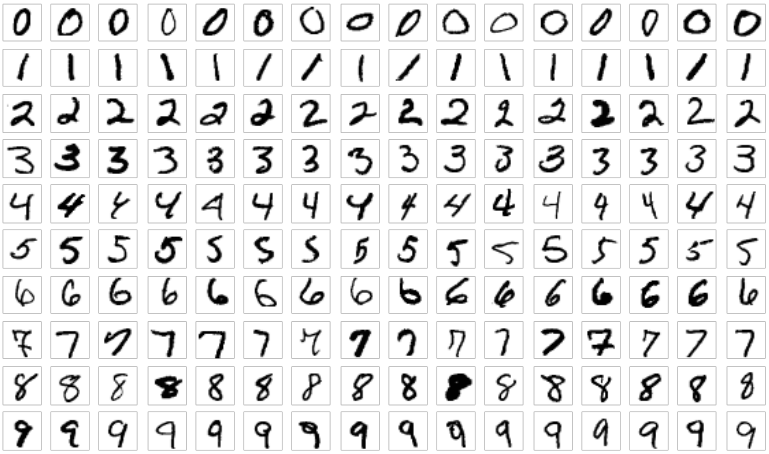


Figura 2: MNIST test dataset

3. IMPLEMENTACIÓN DEL ALGORITMO Y RESULTADOS

3.1. Implementación

3.1.1. Paso 1

Se descarga el Dataset proveniente de keras y se subdivide en dos datas una para el entrenamiento y otra para el testing.

```python
from keras.datasets import mnist, fashion_mnist
from keras.utils import np_utils
# cargar MNIST
(x_train, y_train), (x_test, y_test) = mnist.load_data()
# training data : 60000 datos
# reshape y normalizacion de los datos de entrada
x_train = x_train.reshape(x_train.shape[0], 1, 28*28)
x_train = x_train.astype('float32')
x_train /= 255
# codificar la salida que es un numero en el rango [0,9] en un vector de tamaño 10
# p.ej. el numero 3 se convertira  [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
y_train = np_utils.to_categorical(y_train)

# lo mismo para datos de prueba: 10000 muestras
x_test = x_test.reshape(x_test.shape[0], 1, 28*28)
x_test = x_test.astype('float32')
x_test /= 255
y_test = np_utils.to_categorical(y_test)

# separamos y guardamos el dataset de MNIST
np.savetxt('mnist_images_train.csv', x_train.reshape(len(x_train),784).tolist())
np.savetxt('mnist_images_test.csv', x_test.reshape(len(x_test),784).tolist())
np.savetxt('mnist_labels_train.csv', y_train.tolist())
np.savetxt('mnist_labels_test.csv', y_test.tolist())
```

Figura 3: Descarga de MNIST test dataset

3.1.2. Paso 2

Se separó del Dataset solo los números 0 y 1 para que no sea pesada la carga del entrenamiento y el testing sólo se filtro.

```python
# hacer un map en el rdd como imagen, label(lo restringimos para la prediccion de 0 y 1)
train_ds_rdd = x_train.join(y_train).map(lambda x: x[1]).map(lambda x: (x[0], np.array([x[1][0][:2]])))
test_ds_rdd = x_test.join(y_test).map(lambda x: x[1]).map(lambda x: (x[0], np.array([x[1][0][:2]])))
# filtrar de acuerdo a los labels
train_rdd = train_ds_rdd.filter(lambda x: np.array_equal(x[1][0], [1., 0.]) or np.array_equal(x[1][0], [0., 1.]))
test_rdd = test_ds_rdd.filter(lambda x: np.array_equal(x[1][0], [1., 0.]) or np.array_equal(x[1][0], [0., 1.]))

#guardar en memoria
train_rdd.cache()

print(train_rdd.take(1))
print("Tamaño del Trainset:", train_rdd.count())
print("Tamaño del Testset:", test_rdd.count())
```

Figura 4: Filtro de MNIST test dataset

3.1.3. Paso 3

Se define las funciones de activación, de propagación y la función de retropropagación.

```python
# Funcion general para aplicar cualquier función de activacion
def activation(x, f):
    return f(x)

# Funcion de activacion sigmoidea
def sigmoid(X):
    return 1 / (1 + np.exp(-X))

# Funcion principal sigmoide (utilizada para backward propagation)
def sigmoid_prime(x):
    sig = sigmoid(x)
    return sig * (1 - sig)
```

Figura 5: Función de Activación

```python
# Calcular los pesos antes de aplicar la funcion de activacion
def preforward(x, w, b):
    return np.dot(x, w) + b

# Calcular los pesos despues de aplicar la funcion de activacion
# Es equivalente a la prediccion una vez que se ha entrenado el modelo
def predict(x, W1, B1, W2, B2):
    return sigmoid(preforward(sigmoid(preforward(x , W1, B1)), W2, B2))
```

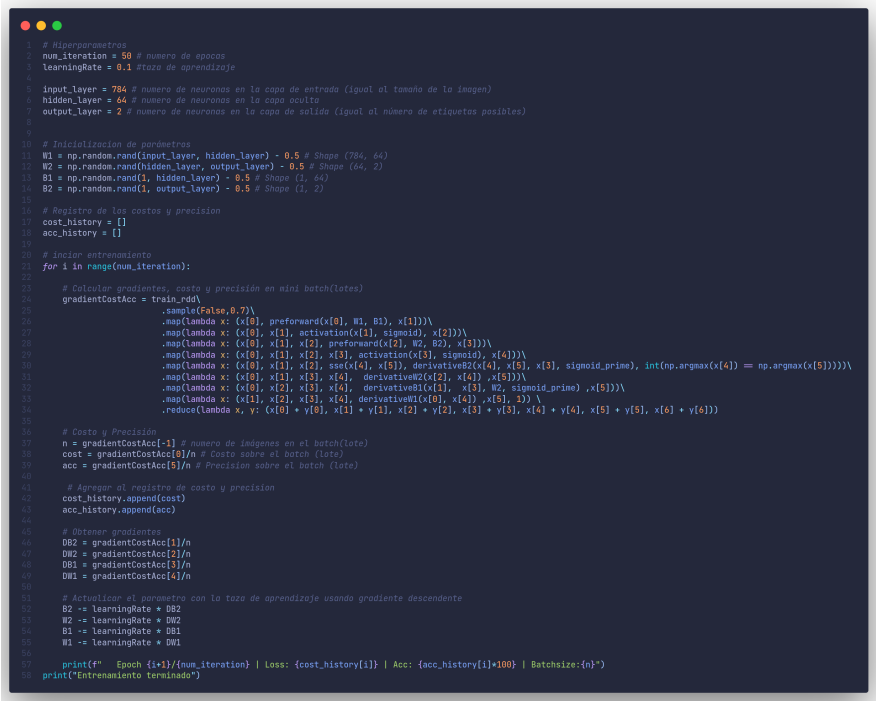
Figura 6: Función de propagación

```python
# Calcular la derivada del error con respecto a B1
def derivativeB1(h_h, dB2, W2, f_prime):
    return np.dot(dB2, W2.T) * f_prime(h_h)

# Calcular la derivada del error con respecto a W1
def derivativeW1(x, dB1):
    return np.dot(x.T, dB1)

# Calcular la derivada del error con respecto a B2
def derivativeB2(y_pred, y_true, y_h, f_prime):
    return (y_pred - y_true) * f_prime(y_h)

# Calcular la derivada del error con respecto a W2
def derivativeW2(h, dB2):
    return np.dot(h.T, dB2)
```

Figura 7: Función de retropropagación

3.1.4. Paso 4

Se procede a hacer las funciones para el entrenamiento usando Pyspark se dividió en 3 capas con los datos de entrada la capa oculta y la capa de salida se definió las variables para esos datos mencionados juntos con el número de iteraciones y la tasa de aprendizaje.



```python
# Hiperparametros
num_iteration = 50 # numero de epocas
learningRate = 0.1 #taza de aprendizaje

input_layer = 784 # numero de neuronas en la capa de entrada (igual al tamaño de la imagen)
hidden_layer = 64 # numero de neuronas en la capa oculta
output_layer = 2 # numero de neuronas en la capa de salida (igual al número de etiquetas posibles)


# Inicializacion de parámetros
W1 = np.random.rand(input_layer, hidden_layer) - 0.5 # Shape (784, 64)
W2 = np.random.rand(hidden_layer, output_layer) - 0.5 # Shape (64, 2)
B1 = np.random.rand(1, hidden_layer) - 0.5 # Shape (1, 64)
B2 = np.random.rand(1, output_layer) - 0.5 # Shape (1, 2)

# Registro de los costos y precision
cost_history = []
acc_history = []

# inciar entrenamiento
for i in range(num_iteration):

    # Calcular gradientes, costo y precisión en mini batch(lotes)
    gradientCostAcc = train_rdd\
                        .sample(False,0.7)\
                        .map(lambda x: (x[0], preforward(x[0], W1, B1), x[1]))\
                        .map(lambda x: (x[0], x[1], activation(x[1], sigmoid), x[2]))\
                        .map(lambda x: (x[0], x[1], x[2], preforward(x[2], W2, B2), x[3]))\
                        .map(lambda x: (x[0], x[1], x[2], x[3], activation(x[3], sigmoid), x[4]))\
                        .map(lambda x: (x[0], x[1], x[2], sse(x[4], x[5]), derivativeB2(x[4], x[5], x[3], sigmoid_prime), int(np.argmax(x[4]) == np.argmax(x[5]))))\
                        .map(lambda x: (x[0], x[1], x[3], x[4],  derivativeW2(x[2], x[4]) ,x[5]))\
                        .map(lambda x: (x[0], x[2], x[3], x[4],  derivativeB1(x[1],  x[3], W2, sigmoid_prime) ,x[5]))\
                        .map(lambda x: (x[1], x[2], x[3], x[4], derivativeW1(x[0], x[4]) ,x[5], 1)) \
                        .reduce(lambda x, y: (x[0] + y[0], x[1] + y[1], x[2] + y[2], x[3] + y[3], x[4] + y[4], x[5] + y[5], x[6] + y[6]))

    # Costo y Precisión
    n = gradientCostAcc[-1] # numero de imágenes en el batch(lote)
    cost = gradientCostAcc[0]/n # Costo sobre el batch (lote)
    acc = gradientCostAcc[5]/n # Precision sobre el batch (lote)

     # Agregar al registro de costo y precision
    cost_history.append(cost)
    acc_history.append(acc)

    # Obtener gradientes
    DB2 = gradientCostAcc[1]/n
    DW2 = gradientCostAcc[2]/n
    DB1 = gradientCostAcc[3]/n
    DW1 = gradientCostAcc[4]/n

    # Actualizar el parametro con la taza de aprendizaje usando gradiente descendente
    B2 -= learningRate * DB2
    W2 -= learningRate * DW2
    B1 -= learningRate * DB1
    W1 -= learningRate * DW1

    print(f"  Epoch {i+1}/{num_iteration} | Loss: {cost_history[i]} | Acc: {acc_history[i]*100} | Batchsize:{n}")
print("Entrenamiento terminado")
```

Figura 8: Función de retropropagación

```
Epoch 44/50 | Loss: 0.10473138955081028 | Acc: 97.76624548736463 | Batchsize:8864
Epoch 45/50 | Loss: 0.10299284892147133 | Acc: 97.97251633250733 | Batchsize:8878
Epoch 46/50 | Loss: 0.10050727453123252 | Acc: 98.12831209832 | Batchsize:8869
Epoch 47/50 | Loss: 0.09953736521449322 | Acc: 97.93267058291912 | Batchsize:8852
Epoch 48/50 | Loss: 0.09730965116933456 | Acc: 98.01812004530012 | Batchsize:8830
Epoch 49/50 | Loss: 0.09558744579282435 | Acc: 98.0561555075594 | Batchsize:8797
Epoch 50/50 | Loss: 0.09317895994298649 | Acc: 98.3308898617233 | Batchsize:8867
Entrenamiento terminado
```

Figura 9: Resultados del entrenamiento

Se muestran los datos del entrenamiento con una exactitud del 98 %.

3.2. Resultados

3.2.1. Paso 5

Se muestran los gráficos de pérdida y precisión.

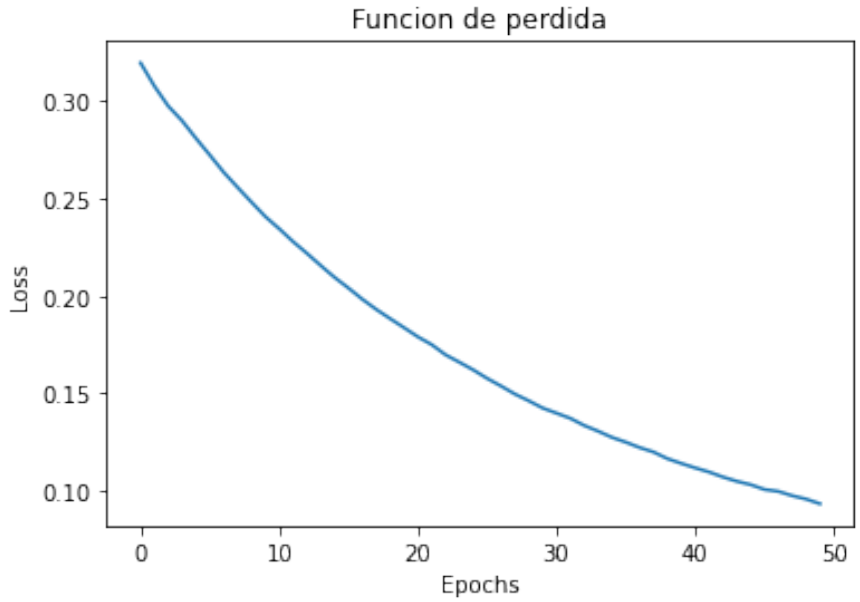


Figura 10: Función de pérdida

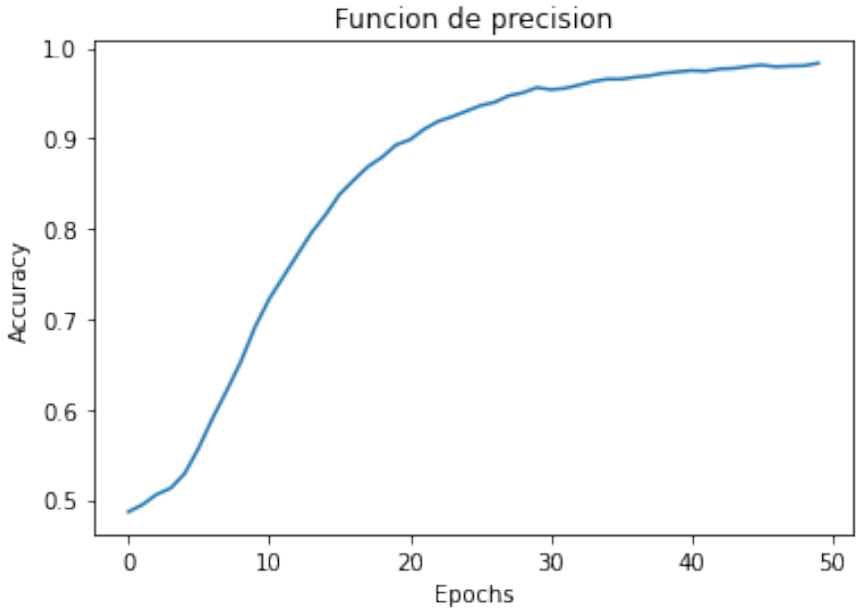


Figura 11: Función de precisión

3.2.2. Paso 6

Se procede con el proceso de evaluación del entrenamiento y obtener la matriz de confusión.

```python
# Usar el modelo entrenado sobre Testset y obtener la matriz de confusion por clase
metrics = test_rdd.map(lambda x: get_metrics(np.round(predict(x[0], W1, B1, W2, B2)), x[1])).reduce(lambda x, y: x + y)

# Para cada clase, proporcione TP (Verdadero Positivo), FP(Falso Positivo), FN(Falso Negativo), TN(Verdadero Negativo) y
# Accuracy, y recall, y score F1
for label, label_metrics in enumerate(metrics):

    print(f"\n---- Digito {label} ------")
    tn, fp, fn, tp = label_metrics.ravel()
    print("TP:", tp, "FP:", fp, "FN:", fn, "TN:", tn)

    precision = tp / (tp + fp + 0.000001)
    print(f"\Accuracy : {precision}")

    recall = tp / (tp + fn + 0.000001)
    print(f"Recall: {recall}")

    F1 = 2 * (precision * recall) / (precision + recall + 0.000001)
    print(f"F1 score: {F1}")
```

Figura 12: Evaluación

```
---- Digito 0 ------
TP: 898 FP: 29 FN: 82 TN: 1106
\Accuracy : 0.9687162880596373
Recall: 0.9163265296772178
F1 score: 0.9417928921622527

---- Digito 1 ------
TP: 1128 FP: 67 FN: 7 TN: 913
\Accuracy : 0.9439330536034032
Recall: 0.9938325982433192
F1 score: 0.9682398428483485
```

Figura 13: Resultados de Evaluación

3.3. Predicciones

3.3.1. Paso 7

Se procede a hacer la predicción con datos que no pertenecen al conjunto de entrenamiento.

```python
for image_test in test_rdd.map(lambda x: (x[0], predict(x[0], W1, B1, W2, B2), np.argmax(x[1]))).takeSample(False, 15):
    pred = np.argmax(image_test[1])
    print(f'prediccion: {pred}, probabilidad: {round(image_test[1][0][pred], 2)} ')
    image = np.reshape(image_test[0], (28, 28))
    plt.imshow(image, cmap='binary')
    plt.show()
```

Figura 14: Predicción

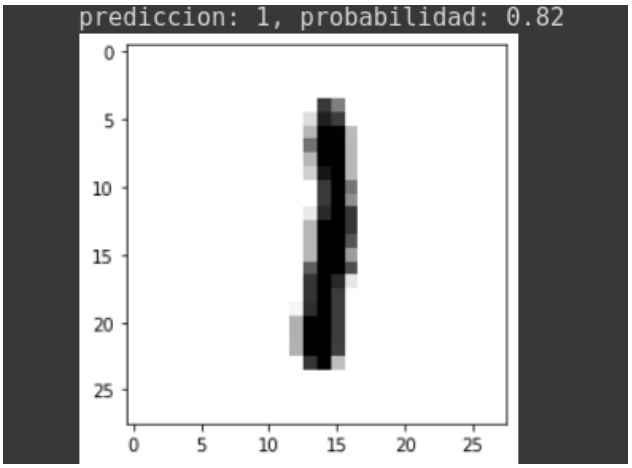


Figura 15: Resultado de Predicción

4. TRABAJOS FUTUROS

- Incrementar la cantidad de neuronas en la capas cultas y mejorar la arquitectura del modelo para clasificar números de 0 a 9.

- Agregar capas convolucionales es para permitir la clasificación de cualquier tipo de imágenes. por ejemplo especies de aves, mariposas, gatos y perros, etc.

- Agregar funciones de early stopping esto con el objetivo de evitar el overfitting (sobre entrenamiento).

- Optimizar el algoritmo:
  - Usar un enfoque más modular para agregar capas más fácilmente.
  - Utilizar técnicas más avanzadas (técnicas de regularización, CNN para clasificación de imágenes, mejores optimizadores, entre otros).
  - Usar particiones, con mapPartitions o mapPartitionWithIndex para ejecutar cálculos en mini lotes.

5. CONCLUSIONES

Las redes neuronales por sí solas pueden ser difíciles de entender, incluso usando Python simple. Poder extenderlo a sistemas más escalables como Spark es un gran proyecto y puede ayudar a comprender mejor los conceptos complejos detrás de él.

Spark permite trabajar con datos más o menos estructurados (RDDs, data frames, data sets) dependiendo de las necesidades y preferencias del usuario.

Spark se integra de manera muy cómoda con otras herramientas Big Data.

Aplicar ecuaciones matemáticas en redes neuronales hace que ejecutarlas mediante la lógica Spark (MapReduce) en Python sea más fácil.

Referencias

[1] Crear una red neuronal en python desde cero — aprende machine learning. https://www.aprendemachinelearning.com/crear-una-red-neuronal-en-python-desde-cero/. (Accessed on 02/13/2022).

[2] Deep learning con pyspark. en este artículo mostraré como crear... — by jonathan quiza — ciencia y datos — medium. https://medium.com/datos-y-ciencia/deep-learning-con-pyspark-7377022aa09b. (Accessed on 02/13/2022).

[3] Greenwade, G. D. The Comprehensive Tex Archive Network (CTAN). *TUGBoat 14*, 3 (1993), 342–351.

[4] Pérez Carrasco, J. A., Serrano Gotarredona, M. d. C., Acha Piñero, B., Serrano Gotarredona, M. T., and Linares Barranco, B. Red neuronal convolucional rápida sin fotogramas para reconocimientos de dígitos. In *XXVI Simposio de la URSI (2011), p 1-4* (2011), Unión Científica Internacional de Radio.

[5] Singh, P. *Machine Learning with PySpark: With Natural Language Processing and Recommender Systems*. Apress, 2018.

[6] Torres, J. *Python deep learning*. Marcombo, 2020.