2-2-2021

PROBLEMA 1: MNIST

Trabajo adicional teoría

Inteligencia de Negocio

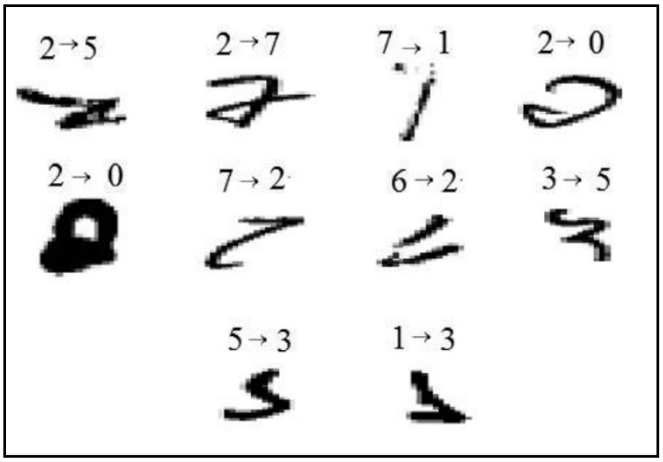
JUAN CARLOS GONZÁLEZ QUESADA. -
UGR

Contenido

Descripción y análisis del problema

La base de datos MNIST es una gran base de datos de dígitos escritos a mano que se utiliza comúnmente para la formación de procesamientos de imágenes en el sistema. Fue creado "remezclando" las muestras de los conjuntos de datos originales de NIST. La base de datos MNIST contiene 60,000 imágenes de entrenamiento y 10,000 imágenes de prueba.

Desde mi punto de vista, uno de los problemas que tiene es la dificultad de entendimiento de algunas imágenes, no sólo para los algoritmos, sino que también para las personas. Por ejemplo, uno de los fallos que tiene el clasificador que marca actualmente el récord mundial (Herrera, Siham Tabika Ricardo F.Alvear-Sandoval María M.Ruiza José-Luis Sancho-Gómezc Aníbal R.Figueiras-Vidal Francisco, 2020), es el siguiente:



Números a mano con los que falla el récord mundial

El primer número es lo que tenía que haber predicho y al que apunta la flecha es el resultado de la predicción. Como se puede observar, incluso a una persona le cuesta distinguir qué tipo de números son los que se han escrito.

Para la realización de la práctica, será necesario un tratamiento de los datos que nos dan, debido a que tienen muchos fallos. El primero, será renombrar el archivo, puesto que viene así: t10k-images.idx3-ubyte, y para poder ser leído el correcto formato es t10k-images-idx3-ubyte. Luego los cuatro archivos se añaden a una carpeta que llamaremos "samples". Finalmente, habrá que realizarle unas transformaciones para poder trabajar con los datos y ya podremos visualizarlos.

```
#Leemos los datos
mndata = MNIST('samples')
#Cargamos los datos en las variables
x_train, y_train = mndata.load_training()
x_test, y_test = mndata.load_testing()
#Transformamos las variables de "list" a "numpy array"
x_train = numpy.array(x_train)
y_train = numpy.array(y_train)
x_test = numpy.array(x_test)
y_test = numpy.array(y_test)
```

Tratamiento inicial a los datos

Sin embargo, a la hora de la visualización y el trabajo en pruebas con algoritmos, daban varios errores e investigando por internet, me he dado cuenta de que es necesario convertir el array a cuatro dimensiones:

```
#Será necesario dotar de cuatro dimensiones al array
x_train = x_train.reshape(x_train.shape[0], 28, 28, 1)
x_test = x_test.reshape(x_test.shape[0], 28, 28, 1)
input_shape = (28, 28, 1)
```

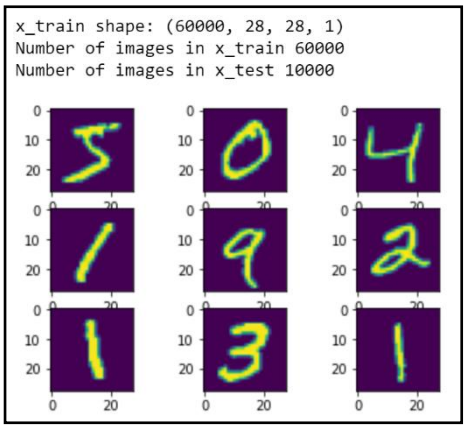
Un ejemplo de visualización de la base de datos sería el siguiente:

```
#Mostramos el formato y el número de elementos en cada variable
print('x_train shape:', x_train.shape)
print('Number of images in x_train', x_train.shape[0])
print('Number of images in x_test', x_test.shape[0])

#Ejemplo de visualización
import matplotlib.pyplot as plt
for i in range(9):
    image = np.asarray(x_train[i]).squeeze()
    pyplot.subplot(330 + 1 + i)
    pyplot.imshow(image)

pyplot.show()
```

```
x_train shape: (60000, 28, 28, 1)
Number of images in x_train 60000
Number of images in x_test 10000
```



Visualización de algunos de los elementos del array

Descripción de los algoritmos

Para el desarrollo de esta práctica trabajaré con algoritmos de Deep Learning, aunque mostraré un par de ejemplos con algoritmos clásicos, para dejar patente que no son una buena solución para tratar con este problema.

Algoritmos clásicos:

- Nayve-Bayes Gaussiano: con el que no podremos probar parámetros puesto que no los permite.
- LGBMClassifier: este algoritmo lo pruebo porque fue el que mejor resultados me dio en la práctica de Kaggle. Usaré los parámetros con los que obtuve los mejores resultados.

Algoritmos de Deep Learning:

- Convolutional Neural Networks (CNNs): Las CNN, también conocidas como ConvNets, constan de múltiples capas y se utilizan principalmente para el procesamiento de imágenes y la detección de objetos. Yann LeCun desarrolló la primera CNN en 1988, cuando se llamaba LeNet. Se utilizó para reconocer caracteres como códigos postales y dígitos. (Bagnato, 2018)

¿Cómo funcionan las CNN?

Las CNN tienen múltiples capas que procesan y extraen características de los datos:

- Capa de convolución: tiene varios filtros para realizar la operación de convolución.
- Unidad lineal rectificada (ReLU): realizar las operaciones sobre los elementos. La salida es un mapa de características rectificado.
- Capa de agrupación: El mapa de características rectificado pasa a una capa de agrupación. El *pooling* es una operación de muestreo descendente que reduce las dimensiones del mapa de características. La capa de agrupación convierte las matrices bidimensionales resultantes del mapa de características agrupado en un único vector lineal largo y continuo, aplanándolo. (Saha, 2018)
- Capa totalmente conectada: se forma cuando la matriz aplanada de la capa de agrupación se introduce como entrada, lo que clasifica e identifica las imágenes.

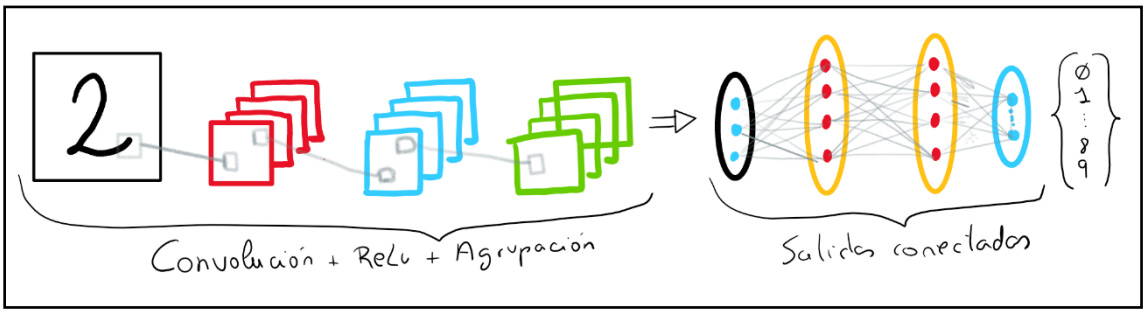
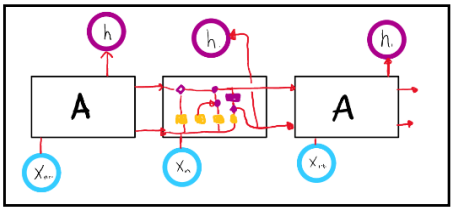


Ilustración gráfica del funcionamiento de la CNN

- Long short-term memory (LSTMs)

Las redes de memoria a corto plazo (LSTM) son un tipo de red neuronal recurrente capaz de aprender la dependencia del orden en problemas de predicción de secuencias. Se trata de un comportamiento necesario en dominios de problemas complejos como la traducción automática o el reconocimiento del habla, entre otros. (DeepAl, s.f.) Las LSTM son un área compleja del aprendizaje profundo. Puede ser difícil entender lo que son los LSTMs y cómo términos tales como bidireccional y secuencia-a-secuencia se relacionan con el campo. (Colah, 2015)



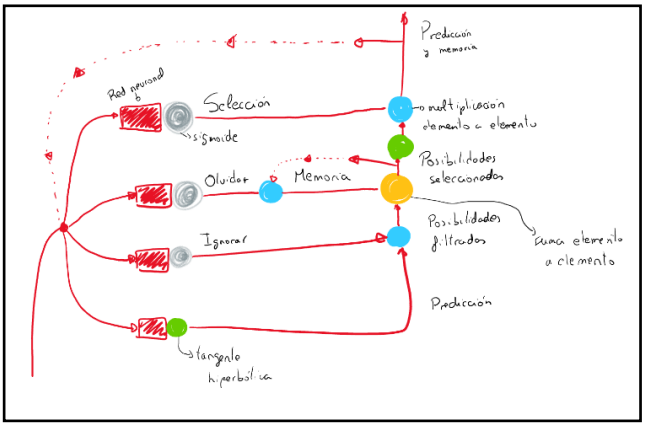
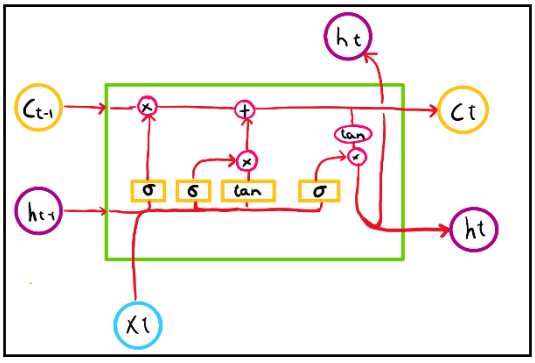


Ilustración gráfica del funcionamiento de la LSTM

Las redes neuronales recurrentes convencionales presentan problemas en su entrenamiento debido a que los gradientes retropropagados tienden a crecer enormemente o a desvanecerse con el tiempo debido a que el gradiente depende no sólo del error presente sino también de los pasados. La acumulación de errores provoca dificultades para memorizar dependencias a largo plazo. Estos problemas son solventados por las redes LSTM, que para ello incorporan una serie de pasos para decidir qué información va a ser almacenada y cual borrada. (Calvo, 2019)

¿Cómo funciona una red LSTM?



Ct es la memoria, Ht son los outputs, el que está a la derecha regresa a la red para seguir alimentando a la información y el de arriba sale hacia el mundo. Tanto Ct-1 como Ht-1 transportan la información que proviene de la anterior neurona. (En el caso de tener varias neuronas.) Xt es la información de ingreso (input) y es un vector con varias capas. Los círculos con X son válvulas que controlan el paso de la información de los distintos inputs. La X más a la izquierda representa el olvido y decide si se olvida la información que llega proveniente de Ct-1. La X del medio es la de memoria, e indica si es necesario grabar la información que llega desde Ht-1 y Xt y la última X, es la de salida e indica si se va a dar o no la información que se posee como salida. Los recuadros amarillos son funciones que deciden si la información tiene que pasar o no a la válvula. El recuadro de "tan", es la función sigmoide y toma la decisión sobre la información, y será la válvula según el valor que se le ha indicado la que decidirá si la información se suma, con el + a la información procedente de la neurona anterior.

- Recurrent Neural Networks (RNNs):

Una red neuronal recurrente es un tipo de red neuronal que contiene bucles, lo que permite almacenar información dentro de la red. En resumen, las Redes Neuronales Recurrentes utilizan su razonamiento a partir de experiencias anteriores para informar de los próximos eventos. Los modelos recurrentes son valiosos por su capacidad de secuenciar vectores, lo que abre la API para realizar tareas más complicadas. (Saxena, 2016)

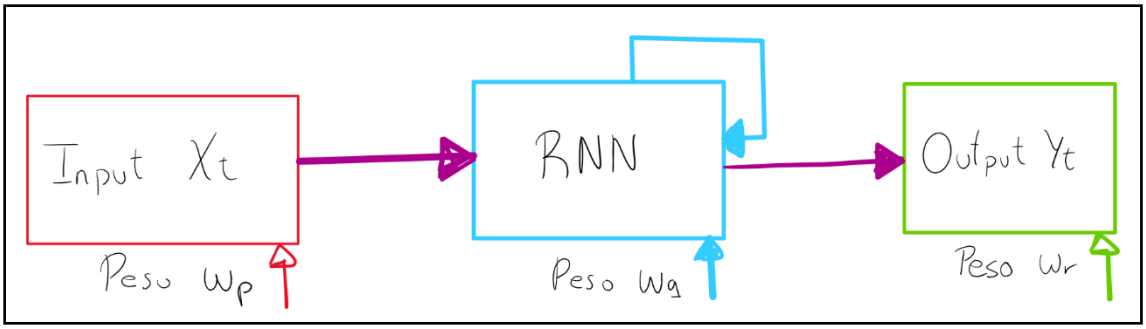


Ilustración gráfica del funcionamiento de la LSTM

¿Por qué surgen las RNN?

Al principio de las redes neuronales. Sólo se tenían las redes CNN y este tipo de redes son unidireccionales, lo que las hace muy importantes para el reconocimiento de patrones. Sin embargo, uno de los inconvenientes que poseen, es que este tipo de red no consideran la información anteriormente leída, por lo que sólo se centran en la información que actualmente tiene como entrada. Podríamos decir que sufren de "amnesia", siendo esta su principal limitación. Por este motivo, surgen las RNN que poseen una cierta memoria de los datos ya leídos anteriormente.

¿Cómo logra la RNN predecir el próximo valor?

Esto se debe a que cada neurona recibe dos parámetros de entrada A_{t-1}, la activación anterior y X_t, el dato actual y la salida es A_t la activación actual (conocida como el estado oculto, permiten preservar y compartir la información entre dos instantes de tiempos) e Y_t la predicción actual.

¿Cómo se calculan los valores de salida?

Para A_t, se calcula con una transformación y una función de activación:

$A_t = f(W_{aa}A_{t-1}+W_{ax}X_t+B_a)$ En donde los valores de las W y de B se calculan con el entrenamiento.

$Y_t=g(W_{ya}A_t+B_y)$ En donde los valores de W y B se calculan durante el proceso de entrenamiento.

Con estas dos funciones, podemos observar el término de recurrencia y de memoria, debido a que para la salida Y_t depende de la activación A_t, pero a su vez la activación depende no sólo de la entrada actual X_t sino del valor previo de la activación A_{t-1} que es precisamente la memoria de la red.

- MultiLayer Perceptron (MLP):

El perceptrón multicapa es una red neuronal artificial formada por múltiples capas, de tal manera que tiene capacidad para resolver problemas que no son linealmente separables, lo cual es la principal limitación del perceptrón. El perceptrón multicapa puede estar total o localmente conectado. (Larranaga, Inza, & Moujahid, s.f.)

El MLP está compuesto:

- Con una capa de entrada x: la neurona de entrada contiene I neuronas.
- Con una capa de salida y: la neurona de salida contiene J neuronas.
- Capas ocultas (intermedias): situadas entre la capa de entrada y la de salida, que no tienen contacto con el mundo exterior. Cada capa oculta está compuesta por un número de neuronas. (Daoui, El Kessab, Kafir, & Moro, 2013)

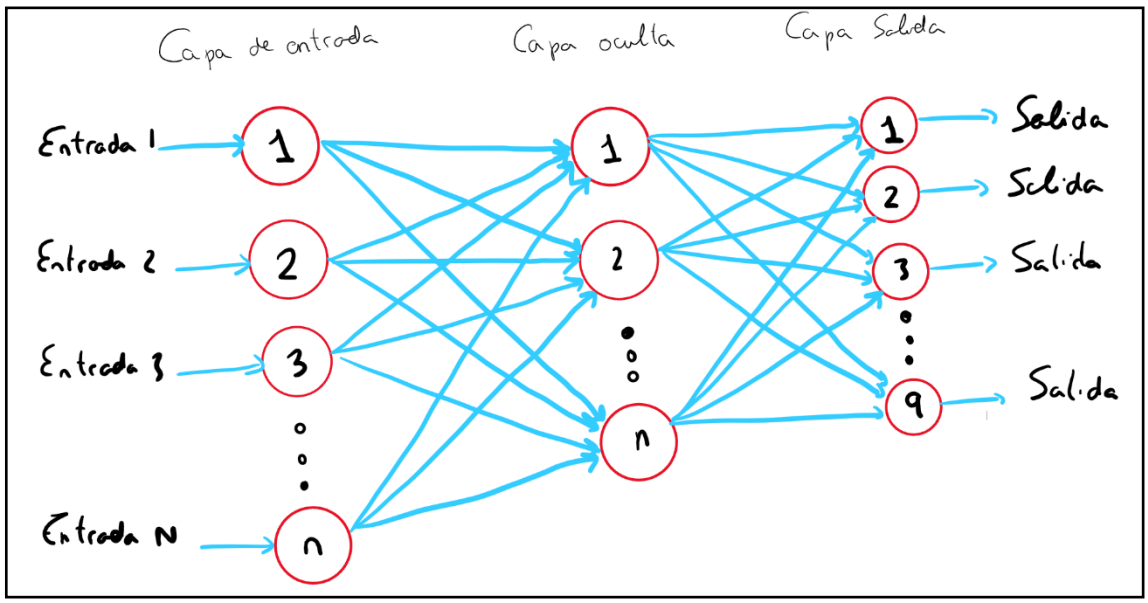


Ilustración gráfica del funcionamiento de la LSTM

Las conexiones están todas orientadas de la capa de entrada a la de salida, es decir cada neurona está conectada a todas las neuronas de la capa siguiente. El número de neuronas en la red es:

- Siete neuronas en la capa de entrada (el número siete corresponde a los valores encontrados en el vector de extracción).
- Diez neuronas en la capa de salida (el número diez corresponde a los números de dígitos utilizados en el reconocimiento).

El número de neuronas en la capa oculta se elige según sus tres condiciones:

- Igual al número de neuronas en la capa de entrada.
- Igualar el 75% del número de neuronas de la capa de entrada.
- Igualar la raíz cuadrada del producto de las dos capas de salida y entrada.

El método utilizado para el aprendizaje es el de retro-propagación del algoritmo gradiente. (Ghosh, Junayed, Hasib, & Emran, 2021)

Estudio experimental

Comenzaremos con los algoritmos clásicos:

Nayve-Bayes:

El tipo Gaussiano fue con el que mejor resultado obtuve en la práctica uno y será el primero que pruebe en este trabajo.

La peculiaridad de usar este algoritmo es que necesitará que el array sea de dos dimensiones, por lo que habrá que deshacer lo que hicimos para poder probarlo.

```
#Nayve-Bayes Gaussian
gnb = GaussianNB()
modeloNBgau = gnb.fit(x_train, y_train)
predNBgau = modeloNBgau.predict(x_test)

scoresCom = cross_val_score(modeloNBgau, x_test, y_test, cv=5, scoring='accuracy')
print("Score Validacion Cruzada con el MODELO", np.mean(scoresCom)*100)
scores = cross_val_score(gnb, x_train, y_train, cv=5, scoring='accuracy')
print("Score Validacion Cruzada sin el Modelo", np.mean(scores)*100)
```

```
Score Validacion Cruzada con el MODELO 57.17
Score Validacion Cruzada sin el Modelo 56.17666666666666
```

Ejecución de Nayve-Bayes

Podemos observar cómo efectivamente el algoritmo Gaussiano no es la solución a este problema.

LGBMClassifier

Usaré los parámetros que me dieron los mejores resultados en la práctica tres. Volverá a ser necesario el uso de dos dimensiones.

```
#Construimos el modelo
lgbm = LGBMClassifier(learning_rate=0.2, objective='binary', n_estimators=550, n_jobs=2, num_leaves=11, max_depth=-1)
modeloLgbm = lgbm.fit(x_train, y_train)
preLGBM = modeloLgbm.predict(x_test)

scores = cross_val_score(lgbm, x_train, y_train, cv=5, scoring='accuracy')
print("Score Validacion Cruzada CON MODELO", np.mean(scores)*100)

scoresCom = cross_val_score(modeloLgbm, x_test, y_test, cv=5, scoring='accuracy')
print("Score Validacion Cruzada SIN el MODELO", np.mean(scoresCom)*100)
```

```
Score Validacion Cruzada CON MODELO 98.045
Score Validacion Cruzada SIN el MODELO 95.48
```

Aunque parece tener un alto porcentaje de acierto, tiene una alta tasa de fallo. Por lo que este algoritmo, tampoco se acerca a ser la solución a este problema.

Empezamos con el estudio de los algoritmos de Deep Learning

Convolutional Neural Networks (CNNs)

Primero, realizamos un pre-procesado sencillo a los datos.

```
#Formato para el primer algoritmo
#Remodelamos el array a 4-dims para que pueda funcionar con la API de Keras
x_train = x_train.reshape(x_train.shape[0], 28, 28, 1)
x_test = x_test.reshape(x_test.shape[0], 28, 28, 1)
input_shape = (28, 28, 1)

#Asegurar de que los valores son float para que podamos obtener puntos decimales después de la división
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')

#Normalizar los códigos RGB dividiéndolos por el valor RGB máximo.
x_train /= 255
x_test /= 255
```

En este momento, realizamos la construcción del modelo:

Se importa el modelo secuencial de Keras y se añaden las capas Conv2D, MaxPooling, Flatten, Dropout y Dense, de las que se ha hablado anteriormente. Además, las capas Dropout luchan contra el sobreajuste al ignorar algunas de las neuronas mientras se entrena, mientras que las capas Flatten aplanan matrices 2D a matrices 1D antes de construir las capas totalmente conectadas.

```
#Creamos el modelo secuencial y añadimos los atributos
model = Sequential()
model.add(Conv2D(28, kernel_size=(3,3), input_shape=input_shape))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten()) #"Aplanar" las matrices 2D para capas totalmente conectadas
model.add(Dense(128, activation=tf.nn.relu))
model.add(Dropout(0.2))
model.add(Dense(10,activation=tf.nn.softmax))
```

Creación del modelo CNN

Se puede experimentar con cualquier número para la primera capa Densa; sin embargo, la capa Densa final debe tener 10 neuronas ya que tenemos 10 clases numéricas (0, 1, 2, ..., 9). Siempre se puede experimentar con el tamaño del kernel, el tamaño del pool, las funciones de activación, la tasa de abandono y el número de neuronas en la primera capa Densa para obtener un mejor resultado, pero podremos tener un elevado overfiting.

El siguiente paso es compilar y entrenar el modelo:

```
#Compilamos el modelo
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
#Entrenamos el modelo
model.fit(x=x_train,y=y_train, epochs=10)
```

Utilizaremos la métrica accuracy y también visualizaremos la pérdida que obtenemos.

```
Epoch 1/10
1875/1875 [==============================] - 44s 22ms/step - loss: 0.3516 - accuracy: 0.8950
Epoch 2/10
1875/1875 [==============================] - 43s 23ms/step - loss: 0.0842 - accuracy: 0.9743
Epoch 3/10
1875/1875 [==============================] - 39s 21ms/step - loss: 0.0541 - accuracy: 0.9836
Epoch 4/10
1875/1875 [==============================] - 41s 22ms/step - loss: 0.0402 - accuracy: 0.9867
Epoch 5/10
1875/1875 [==============================] - 44s 23ms/step - loss: 0.0329 - accuracy: 0.9896
Epoch 6/10
1875/1875 [==============================] - 41s 22ms/step - loss: 0.0272 - accuracy: 0.9914
Epoch 7/10
1875/1875 [==============================] - 41s 22ms/step - loss: 0.0211 - accuracy: 0.9928
Epoch 8/10
1875/1875 [==============================] - 40s 21ms/step - loss: 0.0189 - accuracy: 0.9934
Epoch 9/10
1875/1875 [==============================] - 42s 22ms/step - loss: 0.0173 - accuracy: 0.99370s - loss: 0.0172 - accura
Epoch 10/10
1875/1875 [==============================] - 41s 22ms/step - loss: 0.0138 - accuracy: 0.9955
```

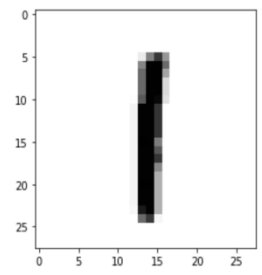
Resultado de la ejecución

Para este primer algoritmo, hemos logrado un 98,5% de precisión con un modelo básico. En muchos casos de clasificación de imágenes (por ejemplo, para coches autónomos) no podemos tolerar ni siquiera un 0,1% de error, ya que, como analogía, provocaría 1 accidente de cada 1000. Sin embargo, para nuestro primer modelo, diría que el resultado sigue siendo bastante bueno.

Para poder luego utilizar nuestro modelo, podremos darle valores y nos devolverá su predicción junto a la imagen del número que tenía que predecir:

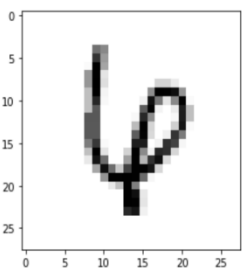
```
image_index = 341
plt.imshow(x_test[image_index].reshape(28, 28),cmap='Greys')
pred = model.predict(x_test[image_index].reshape(1, 28, 28, 1))
print(pred.argmax())
```

```
1
```




```
image_index = 2421
plt.imshow(x_test[image_index].reshape(28, 28),cmap='Greys')
pred = model.predict(x_test[image_index].reshape(1, 28, 28, 1))
print(pred.argmax())
```

```
6
```



Pruebas con nuestro primer modelo

El código pertenece a Jason Brownlee (Brownlee, 2019)

Long short-term memory (LSTMs)

Primero, creamos el modelo:

```
#Creación del modelo
model = Sequential()
model.add(LSTM(128, input_shape=(x_train.shape[1:]), activation='relu', return_sequences=True))
model.add(Dropout(0.2))
model.add(LSTM(128, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(32, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(10, activation='softmax'))

#Añadimos la optimización Adam
opt = tf.keras.optimizers.Adam(lr=1e-3, decay=1e-5)
```

Añadimos la capa de la red recurrente LSTM y las capas Dende y Dropuot de las que ya hemos hablado anteriormente.

Lo compilamos y entrenamos:

```
model.compile(loss='sparse_categorical_crossentropy', optimizer=opt, metrics=['accuracy'])
model.fit(x_train, y_train, epochs=10, validation_data=(x_test, y_test))
```

Los resultados serán:

```
Epoch 1/10
1875/1875 [==============================] - 121s 61ms/step - loss: 1.0965 - accuracy: 0.6182 - val_loss: 0.1205 - val_accur
acy: 0.9620
Epoch 2/10
1875/1875 [==============================] - 112s 60ms/step - loss: 0.1625 - accuracy: 0.9558 - val_loss: 0.1148 - val_accur
acy: 0.9665
Epoch 3/10
1875/1875 [==============================] - 96s 51ms/step - loss: 0.1076 - accuracy: 0.9718 - val_loss: 0.0626 - val_accura
cy: 0.9818
Epoch 4/10
1875/1875 [==============================] - 79s 42ms/step - loss: 0.0822 - accuracy: 0.9776 - val_loss: 0.0576 - val_accura
cy: 0.9840
Epoch 5/10
1875/1875 [==============================] - 95s 51ms/step - loss: 0.0662 - accuracy: 0.9826 - val_loss: 0.0732 - val_accura
cy: 0.9800
Epoch 6/10
1875/1875 [==============================] - 124s 66ms/step - loss: 0.0613 - accuracy: 0.9836 - val_loss: 0.0765 - val_accur
acy: 0.9747
Epoch 7/10
1875/1875 [==============================] - 110s 59ms/step - loss: 0.0495 - accuracy: 0.9864 - val_loss: 0.0462 - val_accur
acy: 0.9863
Epoch 8/10
1875/1875 [==============================] - 174s 93ms/step - loss: 0.0463 - accuracy: 0.9878 - val_loss: 0.0402 - val_accur
acy: 0.9896
Epoch 9/10
1875/1875 [==============================] - 161s 86ms/step - loss: 0.0352 - accuracy: 0.9905 - val_loss: 0.0399 - val_accur
acy: 0.9878
Epoch 10/10
1875/1875 [==============================] - 127s 68ms/step - loss: 0.0335 - accuracy: 0.9912 - val_loss: 0.0370 - val_accur
acy: 0.9893
```

Como vemos, obtenemos un mejor resultado que para el ejemplo anterior, aunque no con mucha diferencia,

El código pertenece a Toofan (Toofan, 2020)

Recurrent Neural Networks (RNNs)

Creamos el modelo con la función SimpleKNN.

```
#Creación del modelo
inputs = KL.Input(shape=(28, 28))
x = KL.SimpleRNN(64)(inputs)
outputs = KL.Dense(10, activation="softmax")(x)

#Creación del modelo
model = tf.keras.models.Model(inputs, outputs)
model.summary()
model.compile(optimizer="adam", loss="sparse_categorical_crossentropy", metrics=["acc"])
model.fit(x_train, y_train, epochs=10)

#Obtención de accuracy y loss
test_loss, test_acc = model.evaluate(x_test, y_test)
print("Loss. {0} - Acc: {0}".format(test_loss, test_acc))
```

Usamos la función SimpleRNN y le añadimos la capa Dense. Es una red RNN de las más básicas, pero que ha sido calibrada con parámetros y hace que tenga una baja tasa de fallo.

Compilamos y evaluamos el modelo:

```
Model: "model_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_10 (InputLayer)        [(None, 28, 28)]          0
_________________________________________________________________
simple_rnn_2 (SimpleRNN)     (None, 64)                5952
_________________________________________________________________
dense_2 (Dense)              (None, 10)                650
=================================================================
Total params: 6,602
Trainable params: 6,602
Non-trainable params: 0
_________________________________________________________________
Epoch 1/10
1875/1875 [==============================] - 13s 5ms/step - loss: 0.8069 - acc: 0.7442
Epoch 2/10
1875/1875 [==============================] - 8s 5ms/step - loss: 0.2607 - acc: 0.9245
Epoch 3/10
1875/1875 [==============================] - 11s 6ms/step - loss: 0.2128 - acc: 0.9382
Epoch 4/10
1875/1875 [==============================] - 10s 5ms/step - loss: 0.1955 - acc: 0.9441
Epoch 5/10
1875/1875 [==============================] - 9s 5ms/step - loss: 0.1783 - acc: 0.9489
Epoch 6/10
1875/1875 [==============================] - 10s 5ms/step - loss: 0.1689 - acc: 0.9506
Epoch 7/10
1875/1875 [==============================] - 8s 4ms/step - loss: 0.1547 - acc: 0.9571
Epoch 8/10
1875/1875 [==============================] - 7s 4ms/step - loss: 0.1542 - acc: 0.9560
Epoch 9/10
1875/1875 [==============================] - 7s 4ms/step - loss: 0.1461 - acc: 0.9584
Epoch 10/10
1875/1875 [==============================] - 7s 4ms/step - loss: 0.1429 - acc: 0.9593
313/313 [==============================] - 1s 2ms/step - loss: 0.1435 - acc: 0.9590
Loss. 0.14349783957004547 - Acc: 0.14349783957004547
```

El código pertenece a Idiot Developer (Developer, 2019)

MultiLayer Perceptron (MLP)

El conjunto de datos de formación está estructurado como una matriz tridimensional de instancia, ancho y alto de imagen. Como en un modelo perceptrón es necesario trabajar con píxeles, en ese caso, las 28 imágenes de tamaño 28 serán vectores de entrada de 784 píxeles. Los valores de los píxeles son en escala de grises entre 0 y 255. Debido a que la escala es bien conocida y se comporta bien, podemos rápidamente normalizar los valores de los píxeles en el rango 0 y 1 dividiendo cada valor por el máximo de 255.

```
#Creamos el modelo
model = Sequential()
model.add(Dense(num_pixels, input_dim=num_pixels, kernel_initializer='normal',activation='relu'))
model.add(Dense(num_classes, kernel_initializer='normal', activation='softmax'))
#Compilamos el modelo
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
#Ajustamos el modelo
model.fit(x_train, y_train, validation_data=(x_test, y_test), epochs=10, batch_size=200,verbose=2)
#Evaluamos del modelo
scores = model.evaluate(x_test, y_test, verbose=0)
print("Baseline Error: %.2f%%" % (100-scores[1]*100))
```

Una función de activación de rectifier se utiliza para las neuronas en la capa oculta. Una función de activación softmax se utiliza en la capa de salida para convertir las salidas en valores similares a los de la probabilidad y permitir que una clase de las 10 sea seleccionada como predicción de salida del modelo.

La pérdida logarítmica se utiliza como función de pérdida (llamada categorical_crossentropy en Keras) y el algoritmo de descenso de gradiente ADAM se utiliza para aprender a los pesos.

El resultado de la ejecución es el siguiente:

```
Epoch 1/10
300/300 - 6s - loss: 0.2736 - accuracy: 0.9243 - val_loss: 0.1399 - val_accuracy: 0.9575
Epoch 2/10
300/300 - 5s - loss: 0.1082 - accuracy: 0.9681 - val_loss: 0.0945 - val_accuracy: 0.9710
Epoch 3/10
300/300 - 4s - loss: 0.0696 - accuracy: 0.9799 - val_loss: 0.0720 - val_accuracy: 0.9774
Epoch 4/10
300/300 - 5s - loss: 0.0493 - accuracy: 0.9861 - val_loss: 0.0716 - val_accuracy: 0.9779
Epoch 5/10
300/300 - 5s - loss: 0.0349 - accuracy: 0.9903 - val_loss: 0.0632 - val_accuracy: 0.9806
Epoch 6/10
300/300 - 5s - loss: 0.0262 - accuracy: 0.9929 - val_loss: 0.0605 - val_accuracy: 0.9802
Epoch 7/10
300/300 - 5s - loss: 0.0188 - accuracy: 0.9953 - val_loss: 0.0592 - val_accuracy: 0.9811
Epoch 8/10
300/300 - 5s - loss: 0.0133 - accuracy: 0.9972 - val_loss: 0.0558 - val_accuracy: 0.9823
Epoch 9/10
300/300 - 5s - loss: 0.0100 - accuracy: 0.9983 - val_loss: 0.0579 - val_accuracy: 0.9818
Epoch 10/10
300/300 - 5s - loss: 0.0080 - accuracy: 0.9986 - val_loss: 0.0570 - val_accuracy: 0.9824
```

El código pertenece a Rana Singh (singh, 2019)

Comparativa de resultados

Algoritmo	Accuracy	Val_Loss	Baseline Error
CNN	0,9939	0,0171	1,46
RNN	0,9623	0,134	5,88
LSMT	0,9906	0,036	1,11
MLP	0,9986	0,008	1,76

Para el análisis de los resultados vamos a tener en cuenta que CNN fue de los primeros algoritmos de DeepLearning, que RNN surge para resolver los problemas que no puede CNN y que LSTM surge para mejorar los problemas de RNN.

Basándonos sólo en el Accuracy, vemos como el mejor resultado es el uso de MLP, algoritmo que curiosamente es creado para resolver problemas de asociación de patrones, sin embargo, esta medida debe evitarse, ya que mide el porcentaje de casos que el modelo ha acertado, pero tiene un problema con la exactitud que nos puede llevar a engaño, colocando como mejor algoritmo a uno que no lo es. No obstante, observamos como LSTM > RNN, pero ambos son peores que CNN.

Basándonos en el valor de Val_loss, que mide el error en los datos test, debemos quedarnos con aquel algoritmo que presenta una tasa pequeña, en este caso, vuelve a ser el algoritmo MLP con un 0.008%. Aunque, podemos ya observar en esta medida, lo que comentábamos en el primer párrafo, y es que con el val_loss LSMT<CNN<RNN.

Por último, vamos a estudiar el Baseline error, calculado de la siguiente forma:

(100-scores[1]*100), donde scores es model.evaluate(x_test, y_test, verbose=0)

Podríamos decir que con Baseline Error, obtenemos la cantidad de errores que comete nuestro algoritmo. De forma sorprendente, es el algoritmo LSTM quien mejor dato obtiene.

Sólo nos queda decidir entre MLP y LSMT qué algoritmo es más beneficioso para la resolución de este problema

Algoritmo	Val_Loss	Baseline Error	Val_Loss/Baseline_Error
LSMT	0,036	1,11	0,032432...
MLP	0,008	1,76	0,004545...
Algoritmo	Accuracy	Baseline Error	Accuracy/Baseline_Error
LSMT	0,9906	1,11	0,892432...
MLP	0,9986	1,76	0,5673 ...

Algoritmo	Val_Loss/Baseline_Error	Accuracy/Baseline_Error	Accuracy/Val_error
LSMT	0,032432...	0,892432...	27,51668...
MLP	0,004545...	0,5673 ...	124,8181...

Basándonos en estas relaciones, parece que finalmente, el algoritmo más adecuado de los probados es el MLP.

Planteamiento de futuro, ¿Qué harías si puedes trabajar durante 6 meses en el problema?

Para este apartado, voy a realizar dos líneas de investigación:

La primera será en torno a la mejora de la predicción y la tasa de error y la segunda será para el uso de este problema y su algoritmo en el desarrollo de una aplicación para su uso en la sociedad.

Mejora del algoritmo

Para el desarrollo de este apartado, tendré en cuenta los últimos estudios sobre el problema, al igual que los errores en los resultados.

Para empezar, y sin contar con el último informe con el que se consiguen sólo 10 imágenes mal clasificadas, los pasos para mi trabajo serían los siguientes:

Para ciertos algoritmos, un buen ajuste del parámetro del "learning_rate". Si el ajuste es demasiado grande, el algoritmo puede no converger y oscilar. De lo contrario, la velocidad de convergencia puede ser demasiado lenta. Como la tasa de aprendizaje se ajusta cada vez, los datos serán demasiado grandes y ocuparán demasiada memoria. (Panchal, 2020)

También se puede realizar un estudio acerca del sobreajuste, que podremos tratar con la capa de *Dropout* antes de la capa de salida y puede ser controlada por el parámetro *keep_prob*. En el aprendizaje profundo, el sobreajuste está mejorando el rendimiento en el conjunto de datos de entrenamiento, mientras que el rendimiento en el conjunto de datos de prueba disminuye. La causa esencial del sobreajuste se debe a la incompatibilidad del aprendizaje supervisado. (Wang, y otros, 2020)

Hay varias formas de resolver el sobreajuste:

- Aumentar el conjunto de datos de entrenamiento: Este método, no nos va a beneficiar para reducir esas diez imágenes que dan error, debido a que aumentar el número de imágenes y trabajar con el algoritmo y métodos del actual récord mundial seguiría dándonos ese mismo problema. Aunque cabe la posibilidad de que aprenda de forma diferente y consiga clasificarlas de otra forma.
- Regularización: uso y tratamiento de los parámetros a los algoritmos y redes utilizadas. Aunque se podría realizar un amplio y extenso trabajo sobre los distintos parámetros que puede afectar y reducir el sobre ajuste, no se trata ya de un problema de parametrización. (Ramiah, 2019)

Una de las soluciones que planteo y en las que se podría trabajar es en las etiquetas. Como dije al principio de este trabajo, los errores que el algoritmo no consigue acertar pueden ser ilegibles para una persona también. Para comprobar esto, hice una encuesta por mis redes sociales, en la que preguntaba qué número era para ellos el siguiente:

En dicha encuesta, pregunto qué número ven en las siguientes imágenes.

Queda reflejado que ni las personas viendo esos números y con uso de la vista y de la razón, consiguen verificar que la primera opción es un 6 y la segunda un 2 tal y como está etiquetado en la base de datos.

Por tanto, para intentar resolver este problema, habría varias opciones:

- Borrar esos valores: debido a que están mal etiquetados. Aunque sería una opción que le daría el récord absoluto al último trabajo de investigación ya que se consigue una tasa de error del 0.00%
- Re-etiquetar los valores: Para ello, se realizará un laborioso estudio con diversos algoritmos que no superen la tasa de error del 0.15%. De entre todos ellos, se obtendrán los valores que los algoritmos le asocian a esos casos particulares. Si hay una amplia mayoría de algoritmos que etiquetan alguno de los números como otro, será quizás porque no se realizó en su momento una buena etiquetación del mismo, y, por lo tanto, se podría plantear su re-etiquetado.
  Para aquellos valores, que los algoritmos no consigan una mayoría o al menos no se consigan sólo dos opciones. Serán porque hay que seguir trabajando y perfeccionando el algoritmo con el que se consiguió el 0.1%.

He realizado una encuesta a mis seguidores en mis redes sociales formados por padres, de los futbolistas a los que entreno, mi familia, amigos y conocidos para ver qué números del 0 al 9, observaban en la anterior imagen.

Tras finalizarla y haber recogido 50 votaciones, los resultados han sido los siguientes:

Para el primer número, que está etiquetado como un 6, obtenemos:

Como podemos visualizar en el gráfico, nadie ha reconocido la imagen como un 6. Sin embargo, algo que nos da rigor para realizar el reetiquetado es que el número más votado ha sido el 2, que es justo lo que el algoritmo de DL nos predice. Se podría sumar los NN, ya que en los comentarios han dicho que no es ningún número, pero que, de serlo, verían algo similar a un dos. Teniendo esto en cuenta, nos damos cuenta de que un 90% de los encuestados ven en la imagen lo mismo que el algoritmo del récord mundial.

Para la segunda imagen, obtenemos los siguientes resultados:

Para el segundo ejemplo ocurre los mismo que para la anterior imagen. Lo que ven los votantes es lo mismo que predice el algoritmo, no obstante, la diferencia es que si hay gente que visualiza un 2 tal y como está etiquetado.

Los resultados de las votaciones son los siguientes:

Número	Veces votado
1	2
2	37
4	1
5	1
7	1
NN	8

Número	Veces votado
2	6
5	1
6	2
7	41

Podemos concluir que de forma experimental que los etiquetados en las imágenes que acaban siendo mal clasificadas, podría deberse a un mal etiquetado. Para comprobarlo, como he dicho anteriormente haría un estudio con los diversos algoritmos y técnicas que consiguen no superar la tasa de error del 1,5% y decidir que realizar con los etiquetados.

Si se decide realizar un trabajo sobre los algoritmos para conseguir que no se tengan fallos porque no se obtengan buenos resultados en lo realizado anteriormente, yo optaría por trabajar teniendo como base el estudio del último algoritmo.

Utilizaría el ensembled method con la CNN y añadiría una red LSMT, para poder ir recuperando los datos de forma recurrente y poder conseguir que se cambie el resultado para las imágenes que falla. No obstante, puede que se haya llegado al límite de lo que un algoritmo puede conseguir de tasa de acierto, con estos etiquetados. Otro de los métodos que utilizaría es el hecho de añadir probabilidad a las redes neuronales, debido a que este tipo de problemas tienen mucha incertidumbre y la probabilidad los dota de una base teórica sólida. Por último, realizaría una mezcla, entre una CNN y una LSMT, que combinadas, puedan conseguir reducir la tasa de alguna forma. Sería necesario un estudio previo, y una configuración de sus capas.

Otras de las mejoras por la que podemos optar es una predicción por similitud. En aquellas imágenes que no se tenga una concordancia entre resultados o su predicción no se corresponda con su etiquetado, podremos intentar realizar un sistema, que compare las imágenes con el resto de estas que sí ha clasificado. Es decir, hay 40 imágenes clasificadas como un dos, pues entonces la imagen X que no ha sido posible clasificarla se compara con esas 40 imágenes, y así con todos los números del 0 al 9. Finalmente, se optará por clasificarla o predecirla como el número con el que mayor similitud tenga.

Sin embargo, los inconvenientes que posee es que se haría una predicción una vez se ha terminado el algoritmo, por lo que quizás implementar dos salidas de dos algoritmos, y ver en cuales no coincide, sería una de las posibilidades para comparar por similitud, sin embargo, si fallan en la misma imagen, si sería necesario conocer en la que se ha errado, y calcular la similitud.

Aplicación

Durante los meses que dure mi trabajo, realizaré una aplicación basada en leer los datos de entrada de una profesora de primaria que escribirá los números, ya sean operaciones, fracciones, horas, decimales... en una pizarra digital, los leerá y los transcribirá a un pdf. De esta forma, el trabajo que se realice de forma online, o con semi-presencialidad, podrá ser entregado a los alumnos, en un formato legible y que puede ser de gran utilidad, sobre todo en las edades tempranas en donde tomar apuntes, no es algo todavía habitual.

Bibliografía

Bagnato, J. I. (29 de Noviembre de 2018). *¿Cómo funcionan las Convolutional Neural Networks? Visión por Ordenador*. Obtenido de https://www.aprendemachinelearning.com/como-funcionan-las-convolutional-neural-networks-vision-por-ordenador/

Brownlee, J. (2019). *How to Develop a CNN for MNIST Handwritten Digit Classification*. Obtenido de https://machinelearningmastery.com/how-to-develop-a-convolutional-neural-network-from-scratch-for-mnist-handwritten-digit-classification/

Calvo, D. (2019). *Red Neuronal Recurrente - RNN*. Obtenido de https://www.diegocalvo.es/red-neuronal-recurrente/

Colah. (2015). *Understanding LSTM Networks*. Obtenido de https://colah.github.io/posts/2015-08-Understanding-LSTMs/

Daoui, C., El Kessab, B., Kafir, M., & Moro, K. (2013). *Extraction Method of Handwritten Digit Recognition Tested on the MNIST Database*. Obtenido de https://www.researchgate.net/profile/Badre_Eddine_El_Kessab2/publication/324845327_Extraction_Method_of_Handwritten_Digit_Recognition_Tested_on_the_MNIST_Database/links/5ae7914ea6fdcc03cd8db36d/Extraction-Method-of-Handwritten-Digit-Recognition-Tested-on-

DeepAl. (s.f.). *Recurrent Neural Network*. Obtenido de https://deepai.org/machine-learning-glossary-and-terms/recurrent-neural-network

Developer, I. (2019). *RNN for Digit Classification*. Obtenido de https://www.easy-tensorflow.com/tf-tutorials/recurrent-neural-networks/vanilla-rnn-for-classification

Ghosh, P., Junayed, M. S., Hasib, K. M., & Emran, A. N. (2021). *A Comparative Study of Different Deep Learning Model for Recognition of Handwriting Digits*. Obtenido de 207103102008007127084020001084087077015002001000090086121026070112086094029103095091030096049125038001052021080098017100125000046002046043009070006115070111114117111068050057068089109088112103104098100113120114122099066107093027089102104115119107007065

Herrera, Siham Tabika Ricardo F.Alvear-Sandoval María M.Ruiza José-Luis Sancho-Gómezc Aníbal R.Figueiras-Vidal Francisco. (2020). *MNIST-NET10: A heterogeneous deep networks fusion based on the degree of certainty to reach 0.1% error rate. Ensembles overview and proposal.*

Larranaga, P., Inza, I., & Moujahid, A. (s.f.). *Tema 8. Redes Neuronales*. Obtenido de http://www.sc.ehu.es/ccwbayes/docencia/mmcc/docs/t8neuronales.pdf

Panchal, A. (2020). *Improving Classification accuracy on MNIST using Data Augmentation*. Obtenido de https://towardsdatascience.com/improving-accuracy-on-mnist-using-data-augmentation-b5c38eb5a903

Ramiah, M. (2019). *How I increased the accuracy of MNIST prediction from 84% to 99.41%*. Obtenido de https://madhuramiah.medium.com/how-i-increased-the-accuracy-of-mnist-prediction-from-84-to-99-41-63ebd90cc8a0

Saha, S. (2018). *A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way*. Obtenido de https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

Saxena, A. J. (2016). *Structural-RNN: Deep Learning on Spatio-Temporal Graphs*. Obtenido de https://openaccess.thecvf.com/content_cvpr_2016/html/Jain_Structural-RNN_Deep_Learning_CVPR_2016_paper.html

singh, R. (2019). *Multi-layer Perceptron using Keras on MNIST dataset for Digit Classification*. Obtenido de https://medium.com/analytics-vidhya/multi-layer-perceptron-using-keras-on-mnist-dataset-for-digit-classification-problem-relu-a276cbf05e97

Toofan. (2020). *Mnist with RNN and LSTM*. Obtenido de https://www.kaggle.com/muhammedfathi

Wang, Y., Li, F., Sun, H., Li, W., Zhong, C., Wang, X. W., & Wang, P. (2020). *Improvement of MNIST Image Recognition Based on CNN*. Obtenido de https://iopscience.iop.org/article/10.1088/1755-1315/428/1/012097/pdf

Andy. (2020). Improve your neural networks – Part 1 [TIPS AND TRICKS]. Obtenido de https://adventuresinmachinelearning.com/improve-neural-networks-part-1/