“UNIVERSIDAD MAYOR

DE SAN ANDRES”

CARRERA DE INFORMATICA



**PROYECTO DE INTELIGENCIA ARTIFICIAL**

**(RESONANCIA MAGNETICA CON INTELIGENCIA ARTIFICIAL)**

**NOMBRE:** RENE ALDHAIR PEREZ CONDORI

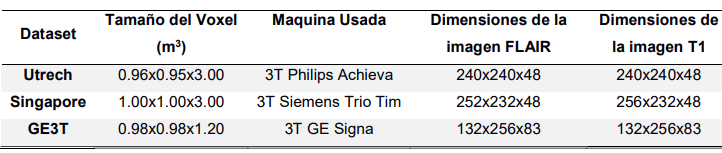
**MATERIA:** INTELIGENCIA ARTIFICIAL INF-354

**DOCENTE:** MOISES SILVA

**La Paz - Bolivia**

**Tópico:**  Resonancia magnética con IA

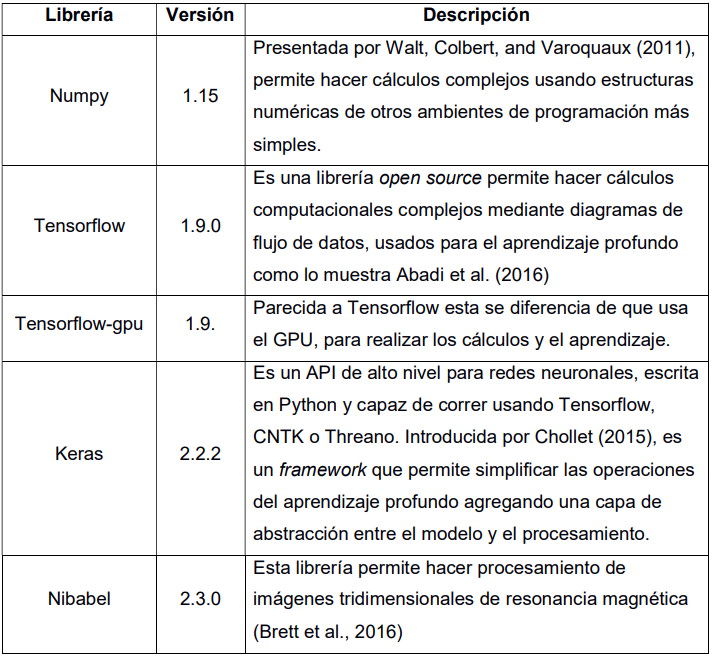
**Dataset:** Para el cual se usaron imágenes adquiridas en tres escáneres diferentes de tres centros de estudios diferentes ubicados en Holanda, Ámsterdam y Singapur. En este concurso el conjunto de datos constaba de un total de 60 sujetos con imágenes de resonancia magnética FLAIR y T1 de tres escáneres diferentes junto con sus máscaras binarias, las cuales consisten en mapas binarios de datos que contienen la información de las lesiones segmentadas por manualmente, lo mismo que se conoce como “ground truth”.

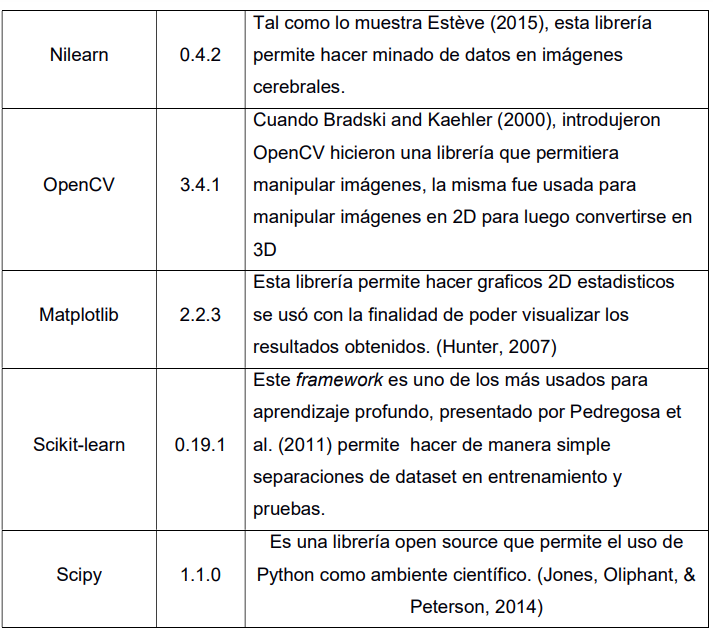


**Proceso:**

* **Objetivo**: La presente investigación está basada en el área de la medicina específicamente con el tema de resonancia magnética con lo cual queremos ver de qué manera ayuda la inteligencia artificial a los pacientes que hacen uso de la resonancia magnética en sus distintos laboratorios.
* Realizar un método de preprocesado de las imágenes para aumentar el conjunto de datos de entrenamiento.
* Elaborar un modelo de aprendizaje profundo basado en la red U-Net 3D
* Validar la efectividad del método por medio de diferentes coeficientes de comparación.
* **Descripción detallada de los campos del dataset**

Se usaron diferentes librerías para poder hacer el análisis y segmentación de las lesiones, todas estas fueron usadas en el ambiente de Python 2.7 y Anaconda como package manager. Las librerías usadas se muestran en la Tabla 2-2, junto con una breve descripción





Tensorflow TensorFlow es un framework basado en Python, el cual se basa en grafos para realizar los cálculos. Cada estructura en TensorFlow, se representa como un grafo computacional, logrando así que los cálculos se realicen en paralelo. Los datos no se almacenan como enteros, punto flotante, cadenas de caracteres u otros tipos de datos primitivos; sino que estos valores son encapsulados en un objeto llamado tensor. Esto permite tener un conjunto de valores primitivos agrupados en un arreglo de cualquier número de dimensiones. El rango de un tensor indica el número de dimensiones. El programa principal de TensorFlow se basa en la idea de un grafo computacional. Esto es un grafo dirigido que consta de dos acciones principales: • Construir el modelo computacional del grafo. • Ejecutar el modelo computacional del grafo: Cada grafo computacional se ejecuta dentro de una sesión; la cual es un entorno de tiempo de ejecución para el grafo computacional. Este asigna la CPU o GPU y mantiene el estado del tiempo de ejecución de TensorFlow. 2.1.2 Keras Keras es una API de redes neuronales profundas de alto nivel en Python que se ejecuta en la parte superior de TensorFlow, CNTK o Theano. Aquí hay algunos conceptos básicos que necesita saber para trabajar con Keras. TensorFlow es una biblioteca de aprendizaje profundo para computación numérica e inteligencia artificial. Es de código abierto y usa gráficos de flujo de datos para el cálculo numérico. Las operaciones matemáticas están representadas por nodos y matrices de datos multidimensionales; es decir, los tensores están representados por los bordes del grafo. Este marco es extremadamente técnico y, por lo tanto, probablemente sea difícil para los analistas de datos, científicos, entre otros. Por esto motivo aparece Keras, el cual hace que la codificación de redes neuronales sea simple. También funciona perfectamente en máquinas CPU y GPU. Los modelos son la estructura de datos básicos de Keras. El modelo secuencial, que consiste en una pila lineal de capas, siendo este el tipo de modelo más simple. Para facilitar el análisis este proporciona funciones comunes, como fit, evaluate y compile. Una capa de Keras, es como una capa de red neuronal. Esto implica que hay capas conectadas, “max pool layers” y capas de activación. Se puede agregar una capa al modelo utilizando la función add de un modelo

* **Proceso básico de análisis de datos:**
  + **Módulos del programa:**

Dentro de la metodología planteada se realizó un programa que sea robusto a los cambios de los parámetros de configuración que se pueden dar en máquinas de resonancia magnética, aunque es una meta utópica, se puede lograr probando con diferentes datasets. Teniendo en cuenta que se requieren más sujetos de prueba para poder hacer el algoritmo más robusto. Se dividirá el programa en tres módulos principales:

* + • Módulo de Preprocesamiento
  + • Módulo de Segmentación
  + • Módulo de Postprocesamiento

Hay que resaltar que el módulo de segmentación va a ser hecho en Python 2.7 usando la librería Tensorflow y Keras.

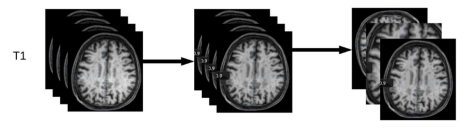
* + **2.3 Módulo de Preprocesado**

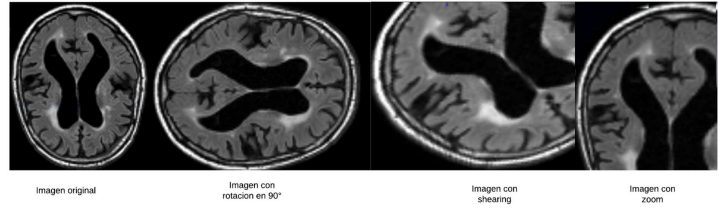
Las imágenes requieren de un tratamiento previo antes de poder ser usadas para entrenar a la red neuronal. Si estas fueran ingresadas tal como se obtienen de los escáneres de resonancia magnética se tendría una cantidad baja de detección de lesiones cerebrales. Hace menos de 5 años se usaban bastantes técnicas de preprocesado para poder lograr hacer la segmentación, esto se daba porque en técnicas como la que muestra Chancay et al. (2015), en donde se usa segmentación por umbral.

Las CNN por naturaleza son buenas para poder identificar y clasificar características de la imagen sin procesar, sin embargo, tal como lo muestra la Ilustración 2-1, se requiere que se potencie la imagen T1, la imagen FLAIR y que también se haga una corrección de bias para potenciar la sustancia blanca cerebral, esto fue realizado previamente y vino junto con el dataset de entrenamiento. Con esta finalidad se harán los métodos mostrados en este capítulo. Selección del clasificador (acorde a los datos supervisado, no supervisado). El clasificador puede, pero no necesariamente depender del preprocesamiento.

Justificar el clasificador (máximo 2 planas con fuente ISBN, DOI)

* + **Primera ejecución: Confiabilidad, matriz de confusión**

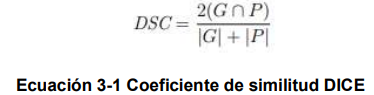


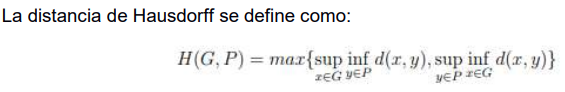


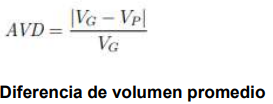
**Imagen después de hacer el aumento ala data**

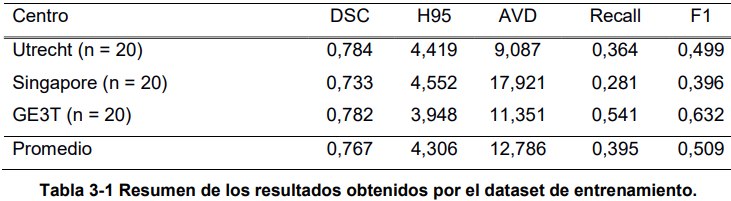
* + Splits: al menos 100 asignaciones, la mediana de la confiabilidad

Académico (primera ejecucion) 80(train)/20(test) – Investigación 50/50 (segunda ejecución)









* + Primer Código:

|  |
| --- |
| import cv2 |
|  | import h5py |
|  | import imageio |
|  | import keras |
|  | import matplotlib.pyplot as plt |
|  | import numpy as np |
|  | import tensorflow as tf |
|  | from IPython.display import Image |
|  | from keras import backend as K |
|  | from keras.engine import Input, Model |
|  | from keras.layers import ( |
|  | Activation, |
|  | Conv3D, |
|  | Deconvolution3D, |
|  | MaxPooling3D, |
|  | UpSampling3D, |
|  | ) |
|  | from keras.layers.merge import concatenate |
|  | from keras.optimizers import Adam |
|  | from keras.utils import to\_categorical |
|  | from tensorflow.compat.v1.logging import INFO, set\_verbosity |
|  |  |
|  | set\_verbosity(INFO) |
|  |  |
|  | K.set\_image\_data\_format("channels\_first") |
|  | def plot\_image\_grid(image): |
|  | data\_all = [] |
|  |  |
|  | data\_all.append(image) |
|  |  |
|  | fig, ax = plt.subplots(3, 6, figsize=[16, 9]) |
|  |  |
|  | # coronal plane |
|  | coronal = np.transpose(data\_all, [1, 3, 2, 4, 0]) |
|  | coronal = np.rot90(coronal, 1) |
|  |  |
|  | # transversal plane |
|  | transversal = np.transpose(data\_all, [2, 1, 3, 4, 0]) |
|  | transversal = np.rot90(transversal, 2) |
|  |  |
|  | # sagittal plane |
|  | sagittal = np.transpose(data\_all, [2, 3, 1, 4, 0]) |
|  | sagittal = np.rot90(sagittal, 1) |
|  |  |
|  | for i in range(6): |
|  | n = np.random.randint(coronal.shape[2]) |
|  | ax[0][i].imshow(np.squeeze(coronal[:, :, n, :])) |
|  | ax[0][i].set\_xticks([]) |
|  | ax[0][i].set\_yticks([]) |
|  | if i == 0: |
|  | ax[0][i].set\_ylabel('Coronal', fontsize=15) |
|  |  |
|  | for i in range(6): |
|  | n = np.random.randint(transversal.shape[2]) |
|  | ax[1][i].imshow(np.squeeze(transversal[:, :, n, :])) |
|  | ax[1][i].set\_xtick |
|  | ax[1][i].set\_yticks([]) |
|  | if i == 0: |
|  | ax[1][i].set\_ylabel('Transversal', fontsize=15) |
|  |  |
|  | for i in range(6): |
|  | n = np.random.randint(sagittal.shape[2]) |
|  | ax[2][i].imshow(np.squeeze(sagittal[:, :, n, :])) |
|  | ax[2][i].set\_xticks([]) |
|  | ax[2][i].set\_yticks([]) |
|  | if i == 0: |
|  | ax[2][i].set\_ylabel('Sagittal', fontsize=15) |
|  |  |
|  | fig.subplots\_adjust(wspace=0, hspace=0) |
|  | def visualize\_data\_gif(data\_): |
|  | images = [] |
|  | for i in range(data\_.shape[0]): |
|  | x = data\_[min(i, data\_.shape[0] - 1), :, :] |
|  | y = data\_[:, min(i, data\_.shape[1] - 1), :] |
|  | z = data\_[:, :, min(i, data\_.shape[2] - 1)] |
|  | img = np.concatenate((x, y, z), axis=1) |
|  | images.append(img) |
|  | imageio.mimsave("/tmp/gif.gif", images, duration=0.01) |
|  | return Image(filename="/tmp/gif.gif", format='png') |
|  | # References: https://github.com/ellisdg/3DUnetCNN/blob/master/unet3d/ |
|  | def create\_convolution\_block(input\_layer, n\_filters, batch\_normalization=False, |
|  | kernel=(3, 3, 3), activation=None, |
|  | padding='same', strides=(1, 1, 1), |
|  | instance\_normalization=False): |
|  | """ |
|  | :param strides: |
|  | :param input\_layer: |
|  | :param n\_filters: |
|  | :param batch\_normalization: |
|  | :param kernel: |
|  | :param activation: Keras activation layer to use. (default is 'relu') |
|  | :param padding: |
|  | :return: |
|  | """ |
|  | layer = Conv3D(n\_filters, kernel, padding=padding, strides=strides)( |
|  | input\_layer) |
|  | if activation is None: |
|  | return Activation('relu')(layer) |
|  | else: |
|  | return activation()(layer) |
|  | def get\_up\_convolution(n\_filters, pool\_size, kernel\_size=(2, 2, 2), |
|  | strides=(2, 2, 2), |
|  | deconvolution=False): |
|  | if deconvolution: |
|  | return Deconvolution3D(filters=n\_filters, kernel\_size=kernel\_size, |
|  | strides=strides) |
|  | else: |
|  | return UpSampling3D(size=pool\_size) |
|  | def unet\_model\_3d(loss\_function, input\_shape=(4, 160, 160, 16), |
|  | pool\_size=(2, 2, 2), n\_labels=3, |
|  | initial\_learning\_rate=0.00001, |
|  | deconvolution=False, depth=4, n\_base\_filters=32, |
|  | include\_label\_wise\_dice\_coefficients=False, metrics=[], |
|  | batch\_normalization=False, activation\_name="sigmoid"): |
|  | """ |
|  | Builds the 3D UNet Keras model.f |
|  | :param metrics: List metrics to be calculated during model training (default is dice coefficient). |
|  | :param include\_label\_wise\_dice\_coefficients: If True and n\_labels is greater than 1, model will report the dice |
|  | coefficient for each label as metric. |
|  | :param n\_base\_filters: The number of filters that the first layer in the convolution network will have. Following |
|  | layers will contain a multiple of this number. Lowering this number will likely reduce the amount of memory required |
|  | to train the model. |
|  | :param depth: indicates the depth of the U-shape for the model. The greater the depth, the more max pooling |
|  | layers will be added to the model. Lowering the depth may reduce the amount of memory required for training. |
|  | :param input\_shape: Shape of the input data (n\_chanels, x\_size, y\_size, z\_size). The x, y, and z sizes must be |
|  | divisible by the pool size to the power of the depth of the UNet, that is pool\_size^depth. |
|  | :param pool\_size: Pool size for the max pooling operations. |
|  | :param n\_labels: Number of binary labels that the model is learning. |
|  | :param initial\_learning\_rate: Initial learning rate for the model. This will be decayed during training. |
|  | :param deconvolution: If set to True, will use transpose convolution(deconvolution) instead of up-sampling. This |
|  | increases the amount memory required during training. |
|  | :return: Untrained 3D UNet Model |
|  | """ |
|  | inputs = Input(input\_shape) |
|  | current\_layer = inputs |
|  | levels = list() |
|  |  |
|  | # add levels with max pooling |
|  | for layer\_depth in range(depth): |
|  | layer1 = create\_convolution\_block(input\_layer=current\_layer, |
|  | n\_filters=n\_base\_filters \* ( |
|  | 2 \*\* layer\_depth), |
|  | batch\_normalization=batch\_normalization) |
|  | layer2 = create\_convolution\_block(input\_layer=layer1, |
|  | n\_filters=n\_base\_filters \* ( |
|  | 2 \*\* layer\_depth) \* 2, |
|  | batch\_normalization=batch\_normalization) |
|  | if layer\_depth < depth - 1: |
|  | current\_layer = MaxPooling3D(pool\_size=pool\_size)(layer2) |
|  | levels.append([layer1, layer2, current\_layer]) |
|  | else: |
|  | current\_layer = layer2 |
|  | levels.append([layer1, layer2]) |
|  |  |
|  | # add levels with up-convolution or up-sampling |
|  | for layer\_depth in range(depth - 2, -1, -1): |
|  | up\_convolution = get\_up\_convolution(pool\_size=pool\_size, |
|  | deconvolution=deconvolution, |
|  | n\_filters= |
|  | current\_layer.\_keras\_shape[1])( |
|  | current\_layer) |
|  | concat = concatenate([up\_convolution, levels[layer\_depth][1]], axis=1) |
|  | current\_layer = create\_convolution\_block( |
|  | n\_filters=levels[layer\_depth][1].\_keras\_shape[1], |
|  | input\_layer=concat, batch\_normalization=batch\_normalization) |
|  | current\_layer = create\_convolution\_block( |
|  | n\_filters=levels[layer\_depth][1].\_keras\_shape[1], |
|  | input\_layer=current\_layer, |
|  | batch\_normalization=batch\_normalization) |
|  |  |
|  | final\_convolution = Conv3D(n\_labels, (1, 1, 1))(current\_layer) |
|  | act = Activation(activation\_name)(final\_convolution) |
|  | model = Model(inputs=inputs, outputs=act) |
|  |  |
|  | if not isinstance(metrics, list): |
|  | metrics = [metrics] |
|  |  |
|  | model.compile(optimizer=Adam(lr=initial\_learning\_rate), loss=loss\_function, |
|  | metrics=metrics) |
|  | return model |
|  |  |
|  |  |
|  | def visualize\_patch(X, y): |
|  | fig, ax = plt.subplots(1, 2, figsize=[10, 5], squeeze=False) |
|  |  |
|  | ax[0][0].imshow(X[:, :, 0], cmap='Greys\_r') |
|  | ax[0][0].set\_yticks([]) |
|  | ax[0][0].set\_xticks([]) |
|  | ax[0][1].imshow(y[:, :, 0], cmap='Greys\_r') |
|  | ax[0][1].set\_xticks([]) |
|  | ax[0][1].set\_yticks([]) |
|  |  |
|  | fig.subplots\_adjust(wspace=0, hspace=0) |
|  | class VolumeDataGenerator(keras.utils.Sequence): |
|  | def \_\_init\_\_(self, |
|  | sample\_list, |
|  | base\_dir, |
|  | batch\_size=1, |
|  | shuffle=True, |
|  | dim=(160, 160, 16), |
|  | num\_channels=4, |
|  | num\_classes=3, |
|  | verbose=1): |
|  | self.batch\_size = batch\_size |
|  | self.shuffle = shuffle |
|  | self.base\_dir = base\_dir |
|  | self.dim = dim |
|  | self.num\_channels = num\_channels |
|  | self.num\_classes = num\_classes |
|  | self.verbose = verbose |
|  | self.sample\_list = sample\_list |
|  | self.on\_epoch\_end() |
|  |  |
|  | def on\_epoch\_end(self): |
|  | 'Updates indexes after each epoch' |
|  | self.indexes = np.arange(len(self.sample\_list)) |
|  | if self.shuffle == True: |
|  | np.random.shuffle(self.indexes) |
|  |  |
|  | def \_\_len\_\_(self): |
|  | 'Denotes the number of batches per epoch' |
|  | return int(np.floor(len(self.sample\_list) / self.batch\_size)) |
|  |  |
|  | def \_\_data\_generation(self, list\_IDs\_temp): |
|  | 'Generates data containing batch\_size samples' |
|  |  |
|  | # Initialization |
|  | X = np.zeros((self.batch\_size, self.num\_channels, \*self.dim), |
|  | dtype=np.float64) |
|  | y = np.zeros((self.batch\_size, self.num\_classes, \*self.dim), |
|  | dtype=np.float64) |
|  |  |
|  | # Generate data |
|  | for i, ID in enumerate(list\_IDs\_temp): |
|  | # Store sample |
|  | if self.verbose == 1: |
|  | print("Training on: %s" % self.base\_dir + ID) |
|  | with h5py.File(self.base\_dir + ID, 'r') as f: |
|  | X[i] = np.array(f.get("x")) |
|  | # remove the background class |
|  | y[i] = np.moveaxis(np.array(f.get("y")), 3, 0)[1:] |
|  | return X, y |
|  |  |
|  | def \_\_getitem\_\_(self, index): |
|  | 'Generate one batch of data' |
|  | # Generate indexes of the batch |
|  | indexes = self.indexes[ |
|  | index \* self.batch\_size: (index + 1) \* self.batch\_size] |
|  | # Find list of IDs |
|  | sample\_list\_temp = [self.sample\_list[k] for k in indexes] |
|  | # Generate data |
|  | X, y = self.\_\_data\_generation(sample\_list\_temp) |
|  |  |
|  | return X, y |
|  | def get\_labeled\_image(image, label, is\_categorical=False): |
|  | if not is\_categorical: |
|  | label = to\_categorical(label, num\_classes=4).astype(np.uint8) |
|  |  |
|  | image = cv2.normalize(image[:, :, :, 0], None, alpha=0, beta=255, |
|  | norm\_type=cv2.NORM\_MINMAX, dtype=cv2.CV\_32F).astype( |
|  | np.uint8) |
|  |  |
|  | labeled\_image = np.zeros\_like(label[:, :, :, 1:]) |
|  |  |
|  | # remove tumor part from image |
|  | labeled\_image[:, :, :, 0] = image \* (label[:, :, :, 0]) |
|  | labeled\_image[:, :, :, 1] = image \* (label[:, :, :, 0]) |
|  | labeled\_image[:, :, :, 2] = image \* (label[:, :, :, 0]) |
|  |  |
|  | # color labels |
|  | labeled\_image += label[:, :, :, 1:] \* 255 |
|  | return labeled\_image |
|  |  |
|  |  |
|  | def predict\_and\_viz(image, label, model, threshold, loc=(100, 100, 50)): |
|  | image\_labeled = get\_labeled\_image(image.copy(), label.copy()) |
|  |  |
|  | model\_label = np.zeros([3, 320, 320, 160]) |
|  |  |
|  | for x in range(0, image.shape[0], 160): |
|  | for y in range(0, image.shape[1], 160): |
|  | for z in range(0, image.shape[2], 16): |
|  | patch = np.zeros([4, 160, 160, 16]) |
|  | p = np.moveaxis(image[x: x + 160, y: y + 160, z:z + 16], 3, 0) |
|  | patch[:, 0:p.shape[1], 0:p.shape[2], 0:p.shape[3]] = p |
|  | pred = model.predict(np.expand\_dims(patch, 0)) |
|  | model\_label[:, x:x + p.shape[1], |
|  | y:y + p.shape[2], |
|  | z: z + p.shape[3]] += pred[0][:, :p.shape[1], :p.shape[2], |
|  | :p.shape[3]] |
|  |  |
|  | model\_label = np.moveaxis(model\_label[:, 0:240, 0:240, 0:155], 0, 3) |
|  | model\_label\_reformatted = np.zeros((240, 240, 155, 4)) |
|  |  |
|  | model\_label\_reformatted = to\_categorical(label, num\_classes=4).astype( |
|  | np.uint8) |
|  |  |
|  | model\_label\_reformatted[:, :, :, 1:4] = model\_label |
|  |  |
|  | model\_labeled\_image = get\_labeled\_image(image, model\_label\_reformatted, |
|  | is\_categorical=True) |
|  |  |
|  | fig, ax = plt.subplots(2, 3, figsize=[10, 7]) |
|  |  |
|  | # plane values |
|  | x, y, z = loc |
|  |  |
|  | ax[0][0].imshow(np.rot90(image\_labeled[x, :, :, :])) |
|  | ax[0][0].set\_ylabel('Ground Truth', fontsize=15) |
|  | ax[0][0].set\_xlabel('Sagital', fontsize=15) |
|  |  |
|  | ax[0][1].imshow(np.rot90(image\_labeled[:, y, :, :])) |
|  | ax[0][1].set\_xlabel('Coronal', fontsize=15) |
|  |  |
|  | ax[0][2].imshow(np.squeeze(image\_labeled[:, :, z, :])) |
|  | ax[0][2].set\_xlabel('Transversal', fontsize=15) |
|  |  |
|  | ax[1][0].imshow(np.rot90(model\_labeled\_image[x, :, :, :])) |
|  | ax[1][0].set\_ylabel('Prediction', fontsize=15) |
|  | ax[1][1].imshow(np.rot90(model\_labeled\_image[:, y, :, :])) |
|  | ax[1][2].imshow(model\_labeled\_image[:, :, z, :]) |
|  |  |
|  | fig.subplots\_adjust(wspace=0, hspace=.12) |
|  |  |
|  | for i in range(2): |
|  | for j in range(3): |
|  | ax[i][j].set\_xticks([]) |
|  | ax[i][j].set\_yticks([]) |
|  | return model\_label\_reformatted |

**Conclusiones**

• El método y la metodología aquí planteadas pueden ser usadas en una maquina con recursos limitados obteniendo una eficacia del 81%

• El sistema propuesto mostro un buen desempeño en diferentes escáneres y protocolos usados.

• No se puede decir que se hizo un análisis exhaustivo de diferencias entre las arquitecturas 3D y 2D, lo que implica que se necesitaría verificar con mayor detalle porque un sistema actuó mejor que el otro y porque los resultados lo corroboran.

• El método empleado muestra la alta eficacia de las redes neuronales convolucionales al momento de realizar la tarea de segmentación automática, de tal manera que muestra potencial para pruebas reales.

**BIBLIOGRAFÍA**

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., . . . Isard, M. (2016). Tensorflow: a system for large-scale machine learning. Paper presented at the OSDI.

Amazon, A. (2014). Amazon.

Bradski, G., & Kaehler, A. (2000). OpenCV. Dr. Dobb’s journal of software tools.

Brett, M., Hanke, M., Cipollini, B., Côté, M.-A., Markiewicz, C., Gerhard, S., . . . Kastman, E. (2016). nibabel: 2.1. 0. Zenodo.

Caligiuri, M. E., Perrotta, P., Augimeri, A., Rocca, F., Quattrone, A., & Cherubini, A. (2015). Automatic Detection of White Matter Hyperintensities in Healthy Aging and Pathology Using Magnetic Resonance Imaging: A Review. Neuroinformatics, 13(3), 261-276. doi: 10.1007/s12021-015-9260-y

Çiçek, Ö., Abdulkadir, A., Lienkamp, S. S., Brox, T., & Ronneberger, O. (2016). 3D UNet: learning dense volumetric segmentation from sparse annotation. Paper presented at the International Conference on Medical Image Computing and Computer-Assisted Intervention.

ENLACE DE HITGUB