Universidad Nacional Autónoma de México

Facultad de Ciencias

Redes Neuronales Convolucionales

Modelado y Programación

Luis Erick Montes Garcia - 419004547

Diana Laura Nicolás Pavía - 314183093

Alex Gerardo Fernández Aguilar - 314338097

Johann Ramón Gordillo Guzmán - 418046090

José Jhovan Gallardo Valdez - 310192815

Proyecto presentado como parte del curso de **Modelado y Programación** impartido por el profesor **José de Jesús Galaviz Casas**.

14 de octubre del 2019

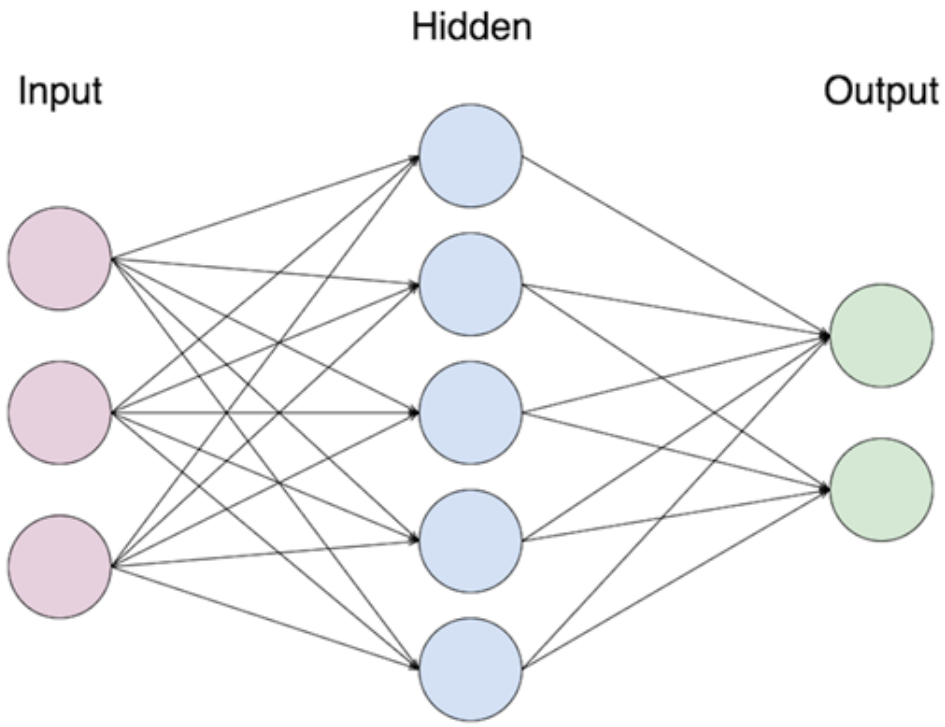
Link al código fuente: https://github.com/JohannGordillo/PumacateAnimales

1. Definición del problema

Se nos ha solicitado elaborar un programa que, por medio de redes neuronales convolucionales, nos permita decir si una imagen dada contiene un hipopótamo, un pingüino, un cuyo, una avestruz, una girafa o ninguno de los animales anteriores.

Se solicitan, además, pruebas unitarias para cada clase individual que clasifique animal por animal, y pruebas unitarias para el programa completo.

También se requiere documentación generada por **Pydoc** en una carpeta.



2. Análisis del problema

1. Requisitos funcionales

   Dada como entrada una imagen cualquiera, se debe entregar como salida el tipo de animal (siempre y cuando pertenezca al conjunto de animales definido en la página anterior) que contiene la imagen.

   Para lograr el objetivo anterior, se debe hacer uso de una red neuronal convolucional con una arquitectura eficiente y robusta, que permita hacer predicciones objetivas y con un rango de error bajo.

2. Requisitos No Funcionales

   Para que el programa sea **amigable**, se implementará una interfaz gráfica sencilla de usar que permita seleccionar una imagen del directorio de archivos del usuario.

   El programa será **escalable**, pues se han implementado redes neuronales para reconocer a un número muy pequeño de animales, y es probable que en un futuro se requiera reconocer a un número más grande de especimenes, ya sea a través de una sola red neuronal para todos los animales, o una red neuronal por animal como en el presente proyecto.

   Se cumplirá con la **eficiencia** necesaria por los usuarios de aplicaciones para clasificación de imagenes de animales, asegurándose el programa de que la información de entrada sea obtenida y procesada de manera rápida y de que la salida sea presentada al usuario de manera clara y objetiva.

   El programa deberá poseer también una **tolerancia a fallos**, pues es probable que se ingresen datos incorrectos en la entrada. Respecto a la salida, se procurará que la cantidad de predicciones incorrectas sea lo menor posible. Sin embargo, no es posible garantizar que la cantidad de dichas predicciones equívocas sea nula.

Para lograr que nuestro modelo logre reconocer imagenes de animales de manera eficiente, es necesario preprocesar las imagenes con las que lo entrenaremos. Esto es, necesitamos que dichas imagenes tengan las mismas dimensiones, y para facilitarnos el trabajo, también haremos que las imagenes de entrenamiento estén en blanco y negro.

Necesitamos construir una red neuronal robusta, por lo que la cantidad de imagenes que necesitamos para entrenarla será muy grande, y es muy difícil para un humano descargarlas sin ayuda de software especializado, así que se recurrirá a este tipo de software para descargcar miles de imagenes de cada clase de animal.

De una manera más particular, podemos dividir el análisis del problema como sigue:

2.1. Entrada

Una o varias imagenes de las que se desee saber si contiene alguno de los animales antes mencionados.

1. **Formato**
   Imagenes en cualquier formato, exceptuando .gif y similares.

2. **Tamaño**
   Imagenes de cualquier tamaño.

3. **Cantidad**
   Una imagen.

4. **Fuente**
   Directorio de archivos del usuario.

2.2. Salida

La predicción para la clase de la imagen de entrada, en la salida estándar.

1. **Formato**
   Texto plano.

2. **Cantidad**
   Una predicción por imagen.

3. **Fuente**
   Salida estándar.

3. Selección de la mejor alternativa

Se decidió usar el lenguaje de programación Python ya que tiene un paquete particular disponible que es útil para la resolución de nuestra tarea; Keras, la cual es posible ejecutar sobre Tensorflow, además es relativamente sencillo de aprender a usar dicho paquete.

Además, Python cuenta con otras librerias importantes que nos facilitarán la tarea de construir una red neuronal convolucional para clasificación de imagenes de animales, como son:

- TensorFlow
  Lo seleccionamos por encima de Thanos debido a que es más sencillo de usar y está mejor documentado, además de que es un equipo especializado en Ciencia de Datos de Google el que está detrás de su desarrollo y mantenimiento.

- Keras
  Usamos esta libreria por la facilidad con la que es posible crear modelos de redes neuronales artificiales convolucionales. Para agregar layers, funciones de activación, compilar y entrenar el modelo bastan unas cuantas líneas de código gracias a Keras.

- OpenCV
  Esta libreria es una de las razones por las que usamos Python para esta tarea, y no otros lenguajes como R y MatLab. OpenCV nos proporciona funciones muy útiles para el preprocesamiento de imagenes. Nos ayudará a leer las imagenes, cambiar sus dimensiones y también a ponerlas en blanco y negro.

- Numpy
  La libreria numpy es muy usada en la Ciencia de Datos, y nos permite trabajar con mucha facilidad con matrices y objetos matemáticos similares de gran importancia al momento de trabajar con redes neuronales.

- Unittest
  Dado que el resultado que ofrece cada una de las partes de nuestro programa es una respuesta binaria (Sí/No) y es sencillo implementarlo con dicha librería, se decidió que era la mejor alternativa.

Las arquitecturas seleccionadas para los modelos de cada animal fueron seleccionadas con base en artículos de Medium - Towards Data Science, libros y otras páginas de internet, todo citado en formato APA en la página de bibliografía.

4. División del trabajo

- Diana Laura Nicolás Pavia.

  Clase realizada: Red neuronal convolucional para reconocer si una imagen dada es un **cuyo** o no.

  Prueba realizada: Pruebas unitarias para el modelo que reconoce si una imagen dada es un **pingüino** o no.

- Johann Ramón Gordillo Guzmán.

  Clase realizada: Red neuronal convolucional para reconocer si una imagen dada es un **pingüino** o no.

  Prueba realizada: Pruebas unitarias para el modelo que reconoce si una imagen dada es una **jirafa** o no.

- José Jhovan Gallardo Valdéz.

  Clase realizada: Red neuronal convolucional para reconocer si una imagen dada es un **jirafa** o no.

  Prueba realizada: Pruebas unitarias para el modelo que reconoce si una imagen dada es un **cuyo** o no.

- Alex Fernández González.

  Clase realizada: Red neuronal convolucional para reconocer si una imagen dada es una **avestruz** o no.

  Prueba realizada: Pruebas unitarias para el modelo que reconoce si una imagen dada es un **hipopótamo** o no.

- Luis Erick Montes Garcia.

  Clase realizada: Red neuronal convolucional para reconocer si una imagen dada es un **hipopótamo** o no.

  Prueba realizada: Pruebas unitarias para el modelo que reconoce si una imagen dada es una **avestruz** o no.

La documentación del código fue realizada por cada uno de los integrantes del equipo en sus archivos correspondientes.

El presente documento, así como el main, fue elaborado por todos los integrantes del equipo en conjunto.

5. Modelo implementado para los clasificadores

La arquitectura de la red neuronal convolucional implementada fue la misma para cada una de las clases clasificadoras de animales particulares, lo único que cambió fue el conjunto de imagenes de entrenamiento. Siendo cada uno de ellos correspondiente al animal de la clase.

Cada uno de los modelos fue entrenado con miles de imagenes, con el fin de que fuera robusto e incrementar la tolerancia a errores del programa.

En esta arquitectura se usaron seis **layers convolucionales** Conv2D de Keras, todos con una función de activación **ReLU** (Rectified Linear Unit).

Los layers convolucionales lo que hacen es, como su nombre lo indica, aplicar una convolución, que es una operación matemática entre dos funciones que da como resultado otra función expresando como cambia la dimensión de una gracias a la otra. Visto de una manera menos formal, las convoluciones son filtros que aplicamos a las imagenes de entrada con el fin de obtener ciertos rasgos característicos de cada una.

Nosotros aplicaremos filtros de 3x3 a nuestra imagene representada como una matriz, pues es lo más usado en la comunidad del Aprendizaje Máquina, y lo que recomiendan nuestras fuentes consultadas. La función de activación elegida, ReLU, lo que hace es aumentar la no linealidad entre los datos, y elegimos esa función en particular también por ser la que mejores resultados da, citando a nuestras fuentes consultadas.

Otro tipo de layer que usamos fueron los **Pooling Layers**. Los pooling layers lo que hacen es reducir la dimensión espacial, pero no la profundidad, de la entrada. Reduce además el número de parámetros y previene el overfitting.

Respecto a los pooling layers, los usamos de 2x2, por ser lo que recomiendan nuestras fuentes y ser lo más usado por la comunidad científica especializada en Machine Learning.

Otra técnica que usamos para mejorar el entrenamiento del modelo y prevenir el overfitting, es el de usar **Dropout Layers** a un .25 y un .30, pues así nuestro modelo será más robusto y no correrá el riesgo de perder tanta información en el proceso de entrenamiento.

También usamos **Flattening**, que transforma una matriz bidimensional con nuestra entrada en un vector que puede alimentar un fully connected layer.

El último tipo de layer que usamos fue el anteriormente mencionado **Fully Connected Layer**, que en Keras se implementa con un Dense, especificando el número de neuronas escondidas como parámetro. Este número de unidades escondidas va disminuyendo conforme avanza la sesión de entrenamiento de la red neuronal. En el último de los layers, usamos la función de activación **softmax** que convierte valores reales en probabilidades, siendo ésto justamente lo que buscamos con nuestra red neuronal.

La arquitectura del modelo se verá más o menos como en el siguiente código:

```
    model = kr.Sequential([
        kr.layers.Conv2D(16, (5, 5 ),activation = 'relu',
        input_shape = train_images.shape[1:])

        kr.layers.Conv2D(32, (5, 5),activation = 'relu')
        kr.layers.MaxPooling2D(2,2)

        kr.layers.Conv2D(64, (3, 3),activation = 'relu')
        kr.layers.Conv2D(64, (3, 3),activation = 'relu')
        kr.layers.MaxPooling2D(2,2)

        kr.layers.Conv2D(128, (3, 3),activation = 'relu')
        kr.layers.MaxPooling2D(2,2)
        kr.layers.Dropout(0.25)

        kr.layers.Flatten()
        kr.layers.Dense(128, activation='relu')
        kr.layers.Dropout(0.3)

        kr.layers.Dense(128, activation= 'relu')
        kr.layers.Dropout(0.3)

        kr.layers.Dense(128, activation='relu')
        kr.layers.Dense(2, activation='softmax')])

```

Antes de construir esta arquitectura de red neuronal, desarrollamos otra arquitectura, a la cual le dio el visto bueno el Dr. Iván Meza del IIMAS, y que mostramos a continuación:

```
        model = Sequential()
        model.add(Conv2D(32, (3, 3), input_shape=features.shape[1:]))
        model.add(Activation("relu"))
        model.add(MaxPooling2D(pool_size=(2,2)))

        # 2nd convolutional layer.
        model.add(Conv2D(64, (3, 3)))
        model.add(Activation("relu"))
        model.add(MaxPooling2D(pool_size=(2,2)))

        # 3rd convolutional layer.
        model.add(Conv2D(64, (3, 3))) # 64 filters of 3x3, each.
        model.add(Activation("relu"))
        model.add(MaxPooling2D(pool_size=(2,2))) # 2x2 pooling.
        model.add(Dropout(0.25)) # Dropout 25% of data.

        # 1st hidden layer.
        model.add(Flatten())
        model.add(Dense(128))
        model.add(Activation("relu"))

        # 2nd hidden layer.
        model.add(Dense(128))
        model.add(Activation("relu"))

        # Output layer with a neuron for each of the classes.
        model.add(Dense(2))
        model.add(Activation("softmax"))

```

Después de un análisis detallado de ambas redes, se vió que la primera tenía un mejor desempeño que la segunda, además el conjunto de datos necesarios para entrenarla era sustancialmente más pequeño, llegar a esta conclusión fue un tanto complicado ya que al obtener las estadísticas, la primera lograba un acurracy de 0.8 y la segunda uno de 0.98, quizá esto no es favorable pues se prestaba para *overfitting*. En conclusión; se trabajó con la primera red porque nuestra experiencia verificó que era mejor.

6. Pruebas implementadas para los clasificadores

Analicemos el objetivo: decidir si una imagen contiene a un animal en particular o no.

Después de decidir la tentativa implementación del programa: unión de redes neuronales donde cada una decide si la imagen contiene la información con la que ha sido entrenada, se decidió que la forma genérica en que deberían funcionar nuestras pruebas unitarias sería:

Algorithm 1 CLASS : *ClassifierTest

```
1: procedure PRUEBA_VERDADERO                                  ▷ Se espera True
2:
3:     return AssertTrue(*_model.predict(*_FILE))
4: procedure PRUEBA_FALSO                                      ▷ Se espera False
5:
6:     return AssertFalse(*_model.predict(NOT_* _FILE))
```

*Observemos que el símbolo * puede ser sustituido por el prefijo de cualquier animal que estamos intentando clasificar y el algoritmo funciona para cada uno de ellos.*

Dicho lo anterior, las pruebas unitaras que tendremos serán de la misma forma para cada animal: hipopótamo, avestruz, cuyo, pingüino y jirafa.

Analizando las pruebas unitarias:

Es difícil hacer pruebas unitarias para este tipo de programas, la complejidad reside en que no está bien definido a partir de qué porcentaje de error una red es considerada de buena o mala calidad. Si lo anterior estuviera bien definido se tendría un camino para implementar pruebas que nos ayuden a verificar la calidad de dicha red. Creemos que la mejor forma de probar a una red no es con pruebas unitaras sino con un análisis estadístico de las veces que acierta y falla en sus predicciones.

Hoy en día ni aún las más grandes empresas con las mejores y más optimas redes implementadas y un poder de cómputo impresionante son capaces de generar redes con una precisión total. Sabiendo lo anterior intuimos que nuestra red tendrá problemas con la predicción debido a que el conjunto de datos con el que se cuenta es bastante pequeño, 8k. Por tanto para este trabajo se decidió que las pruebas antes descritas servirían para ilustrarnos cómo se estaba comportando nuestra red con el entrenamiento, más no para afirmar que el modelo utilizado es correcto o incorrecto.

Luego, para poder ganar un poco de objetividad sobre el comportamiento del programa, se eligieron imagenes al azar y que no formaban parte del conjunto de datos de entrenamiento del modelo para ver la calidad ganada con el conjunto de datos que se usó.

Consideramos que las pruebas propuestas son aceptables para este proyecto ya que en este campo todavía es difuso cómo deberen ser las pruebas unitaras para que puedan ser consideradas *robustas*.

Colocamos la siguiente gráfica con el porcentaje de éxito/fracaso del clasificador:

7. Pseudocódigo de la implementación

1. Descarga de imagenes
   ImageDownloader.py

Result: Carpeta con imagenes de las categorías solicitadas
input: limite, categorías, nombre_carpeta, formatoImg
 imgDownloader = googleImagesDownload(limite, categorías, limite, formatoImg, directorio)

---

2. Preprocesamiento de imágenes
   Preprocess.py

Result: Función: Imagenes preprocesadas para entrenamiento del modelo
for imagen, clase **in** nombre_carpeta:
 a = procesa(imagen)
 trainningData.append(a, clase)

Result: Función: Dar formato a la información de entrada del modelo
ordenAleatorio(trainningData)
 for imagen, clase **in** data:
 etiquetas.append(clase)
 imagenes.append(imagen)

Result: Función: Guardar información de entrenamiento para el modelo
f = creaArchivo(features.pickle)
 f.escribe(estadoDeObjeto(imagenes))
 f.escribe(estadoDeObjeto(etiquetas))
 f.cierra()

---

3. Clasificación de una imagen
   AnimalClassifier.py
   Clase para decidir si una imagen pertenece a una clase o no.

Result: Decidir si una imagen pertenece a una clase en particular
function: predict
 input: model, file_path
 modelo = load(model)
 prediction = modelo.predict(file_path)
 if *prediction == 1* **then**
 | **return** true
else
 | **return** false
end

***Se hace uso del algoritmo anterior para ayudarnos en la clasificación de cada uno de los animales. El símbolo (*) en la siguiente descripción significa que existe un script con un prefijo por cada animal, dicho script tiene terminación Classifier.*

*Classifier.py

Result: Decidir si una imagen contiene a un animal en particular
input: file_path
modelo = AnimalClassifier(modeloEntrenado)
return modelo.predict(file_path)

---

4. Clase principal del programa
   Main.py

Result: Veredicto de la clasificación de una imagen
input: path
if HippoClassifier.predict(path)
then return: *Hipopótamo*
elif: PinguinoClassifier.predict(path)
then return: *Pingüino*
elif: AvestruzClassifier.predict(path)
then return: *Avestruz*
elif: CuyoClassifier.predict(path)
then return: *Cuyo*
elif: JirafaClassifier.predict(path)
then return: *Jirafa*
else: return: *None*

---

8. Unión de los modelos y funcionalidad de la clase principal

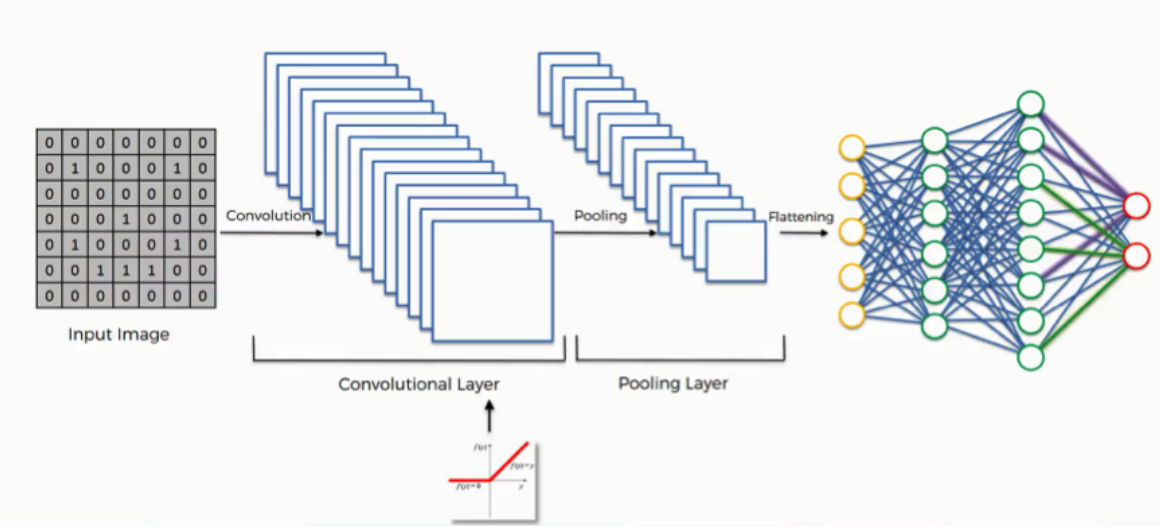
El programa principal funcionará de tal manera que, dada la imagen de entrada, se irá verificando por medio de condicionales **if** anidados si la imagen corresponde o no a uno de los modelos de cada clase de animales.

El primer modelo que utilizamos para predecir si la imagen de entrada corresponde o no a dicho animal, es el modelo para reconocer hipopótamos. En caso afirmativo, terminamos mostrando el tipo de animal. Si no, procedemos a usar el modelo que reconoce pingüinos para ver si la imagen que ingresamos corresponde a un pingüino. Y así sucesivamente, pasando después por el modelo para reconocer jirafas, luego cuyos y finalmente con avestruces.

Si la imagen no se corresponde con ninguno de los cinco animales mencionados en la definición del problema, únicamente informamos al usuario que su imagen no pertenece a ese conjunto de animales.

Al ser los modelos independientes entre sí, no tuvimos problemas en juntarlos dentro de los if's anidados.

La implementación en pseudocódigo de nuestra clase principal con la estructura anteriormente mencionada se encuentra en la sección anterior.



9. Pruebas generales del programa

Como se explicó anteriormente en la sección de *Pruebas implementadas para los clasificadores* creemos que las pruebas para este programa son suficientes si probamos el comportamiento de la red con algunas imagenes elegidas al azar. La estructura general del script **MainTest.py**

Algorithm 2 CLASS : MainTest

```
1:  procedure PRUEBA_VERDADERO_HIPOPOTAMO                    ▷ Se espera True
2:
3:      return AssertTrue(model.predict(HIPPO_FILE))
4:  procedure PRUEBA_VERDADERO_JIRAFA                        ▷ Se espera True
5:
6:      return AssertTrue(model.predict(JIRAFA_FILE))
7:  procedure PRUEBA_VERDADERO_CUYO                          ▷ Se espera True
8:
9:      return AssertTrue(model.predict(CUYO_FILE))
10:
11: procedure PRUEBA_VERDADERO_PINGUINO                      ▷ Se espera True
12:
13:     return AssertTrue(model.predict(PINGUINO_FILE))
14:
15: procedure PRUEBA_VERDADERO_AVESTRUZ                      ▷ Se espera True
16:
17:     return AssertTrue(model.predict(AVESTRUZ_FILE))
```

10. Retrospectiva final del proyecto

El programa funciona como se tenía esperado, pues a pesar de que a veces comete errores en clasificar algunas imagenes, es mayor el número de ocasiones en las que acierta.

Sin embargo, fue un proyecto un poco complicado, pues hizo falta un dataset mas especializado, y aunque es más sencillo modularizar las redes neuronales, es mas difícil entrenar una red para que aprenda a diferenciar entre qué es algo y qué no lo es.

En cambio, una red que pueda predecir entre cinco objetos, aunque es necesario un dataser mucho más grande, es más certero, dado que restringes su margen a solo cinco objetos.

Nos hemos dado cuenta también, de que un buen dataset consta de más de 25,000 imagenes para entrenamiento, y aunque al principio pareció trivial descargar imagenes con el script que realizamos para el presente proyecto, al final fue lo más importante dado que minar la información no fue tan fácil, pues además fue necesario filtrar y depurar las imagenes para que la red fuera más precisa y eficiente.

Consideramos que las dos arquitecturas de redes neuronales que implementamos fueron buenas, pues una de ellas incluso fue aprobada por un reconocido investigador del IIMAS, el Dr. Ivan Vladimir Meza Ruíz. Sin embargo, como él mismo menciona, se necesita tener mucha intuición matemática para saber qué está pasando con el modelo y poder ajustarlo para que devuelva valores que nos interesen. Además de que los datos de los que disponemos para entranar a nuestro modelo son limitados y poco limpios.

Nos gustaría añadir como una reflexión que es un tanto injusto que las empresas grandes conserven muchos data sets útiles como privados, todos contribuimos a construirlos, pero ellos los reservan para sus intereses, por un lado eso nos afecta como estudiantes al no dejarnos trabajar con información real en proyectos interesantes de ciencia de datos y aprendizaje máquina.

También nos dimos cuenta de que el Machine Learning hoy en día es un hype, y hay personas que implementan redes neuronales sin conocer bien del tema, generando desinformación y malas prácticas en la comunidad que apenas se inicia en el aprendizaje máquina y la Ciencia de Datos en general.

Para la realización de este proyecto, utilizamos GitHub y Slack. Plataformas muy importantes dentro del sector de la ingeniería de Software.

11. Bibliografía

- Grus, J. (2015). *Data Science from Scratch*. Editorial O'Reilly. Estados Unidos de América.

- Gonfalonieri, A. (2018). *Dealing with the Lack of Data in Machine Learning*. Medium Blogs - Towards Data Science. Recuperado el 14 de Octubre del 2019 de: https://medium.com/predict/dealing-with-the-lack-of-data-in-machine-learning-725f2abd2b92

- Varshney, K. (2018). *How to plot the model trainning in Keras - using custom callback function and using TensorBoard*. Medium Blogs. Recuperado 14 de Octubre del 2019 de: https://medium.com/@kapilvarshney/how-to-plot-the-model-training-in-keras-using-custom-callback-function-and-using-tensorboard-41e4ce3cb401

- Canuma, P. (2018). *Image Pre-processing*. Medium Blogs - Towards Data Science. Recuperado el 14 de Octubre del 2019 de: https://towardsdatascience.com/image-pre-processing-c1aec0be3edf

- Gupta, D. (2017). *Fundamentals of Deep Learning - Activation Functions and when to use them*. Analytics Vidhya. Recuperado el 14 de Octubre del 2019 de: https://www.analyticsvidhya.com/blog/2017/10/fundamentals-deep-learning-activation-functions-when-to-use-them/

- Mishra, M. (2019). *Convolutional Neural Networks, explained*. Oracle Data Science Blog. Recuperado el 14 de Octubre del 2019 de: https://blogs.oracle.com/datascience/convolutional-neural-networks%2c-explained

- Consejos y opiniones del Dr. Iván Vladimir Meza Ruíz del Instituto de Investigaciones en Matemáticas Aplicadas y Sistemas (IIMAS) de la Universidad Nacional Autónoma de México.