Pontificia Universidad Javeriana
Departamento de Ingeniería de sistemas y computación

# Identificación automatica de sitios de clivaje en familia de virus Potyviridae

Federico Molina Chavez

Sometido como requerimiento para el grado de
Ingeniero de sistemas y computación de la Pontificia Universidad Javeriana
y el diploma de pregrado, Diciembre 2018

# Abstract

Text of the Abstract.

# Acknowledgements

I would like to express (whatever feelings I have) to:

- My supervisor
- My second supervisor
- Other researchers
- My family and friends

# Dedication

Dedication here.

‘Quote text here.’

*Guy Quoted*

# Índice general

# Índice de cuadros

# Índice de figuras

# Capítulo 1

# Introduction

## 1.1. Motivation and Objectives

Motivation and Objectives here.

## 1.2. Contributions

Contributions here.

## 1.3. Statement of Originality

Statement here.

## 1.4. Publications

Publications here.

# Capítulo 2

# Background Theory

## 2.1. Introduction

Text of the Background.

# Capítulo 3

# LSTM

## 3.1. Planteamiento del experimento

En el diagrama 1 se describe la forma del experimento con LSTM. En esta prueba durante el entrenamiento se utilizará ventaneo dentro de las muestras. El tamaño de las ventanas varía de acuerdo al punto de clivaje que se esté buscando. La secuencia es la representación numérica de los aminoácidos que componen el virus. Dado que la entrada de la red es un arreglo, se debe realizar un ventaneo de las muestras para obtenerlos. El experimento utiliza las ventanas una a una para que sean procesadas para calcular los pesos correspondientes del grafo de ejecución.

En el diagrama 2 se muestra el funcionamiento de la ventana. Dentro de la muestra completa,

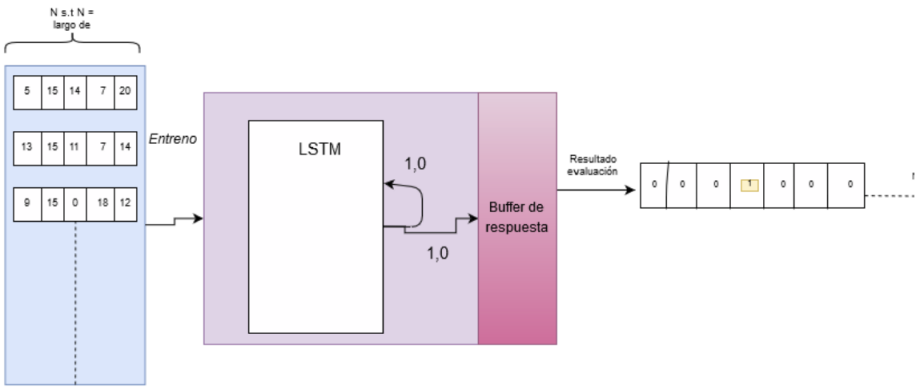



Figura 3.1: diagrama 1

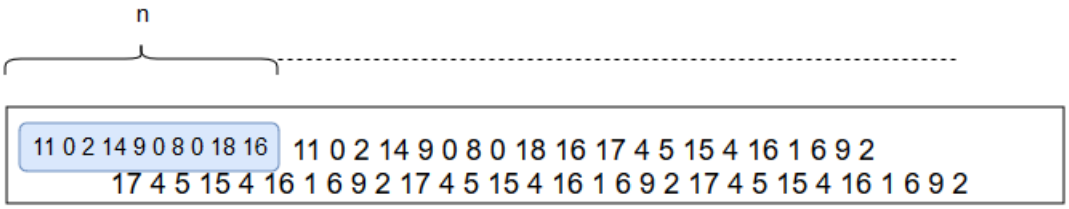



Figura 3.2: diagrama 2

se obtienen vectores de tamaño n separados por un elemento. La definición de sí en una ventana se cuenta con un sitio de clivaje, se da a partir de la localización del mismo, solo se considera una muestra positiva cuando el punto está localizado adecuadamente en la ventana.

Para la evaluación de una secuencia, la entrada es la secuencia completa, que luego es dividida de acuerdo a la ventana escogida. El resultado obtenido por la red será un vector de [1 0] y [0 1] donde [1 0] representa una ventana sin sitio y [0 1] representa una ventana contenedora de sitio de clivaje. La respuesta obtenida por la red es almacenada en un buffer de salida.

En el diagrama 3 se muestra la forma en la que se distribuirá la base de datos para realizar la experimentación. Para poder realizar un correcto ajuste del sistema se utilizará una distribución de 80 % a 20 % dentro del conjunto de entrenamiento donde el primero equivale a los datos de entreno y el segundo a la validación. En cuanto a los porcentajes globales de entrenamiento y prueba se utilizará la división 80 % a 20 %. Dentro la base de datos la distribución de los mismos no es uniforme, esto implica que la cantidad de muestras negativas es mucho mayor a la cantidad de muestras positivas ; dado que se tienen tanta muestras negativas se considera que esto introduce dentro del sistema un error de sesgo.

Para poder revisar la veracidad del efecto que tiene el desbalance dentro de los resultados finales. En la experimentación se seleccionará dentro de la base global conjuntos distintos de porcentaje de distribución entre muestras positivas y negativas. La primera opción es utilizar porcentajes varios de distribución 80 % a 20 % y luego probar con 70 % a 30 %, para poder demostrar lo pensado. En el diagrama 4 y 5 se describe la estructura interna de la LSTM que se va a desarrollar. Los valores principales que buscamos determinar dentro de la estructura es el número de células por capa y el número de capas. En cuanto al número de células por capa,

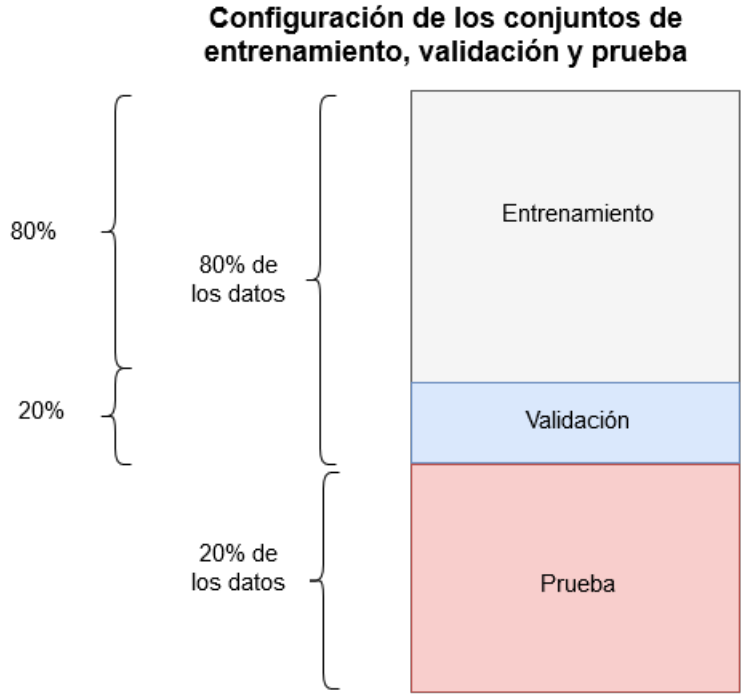



Figura 3.3: diagrama 3

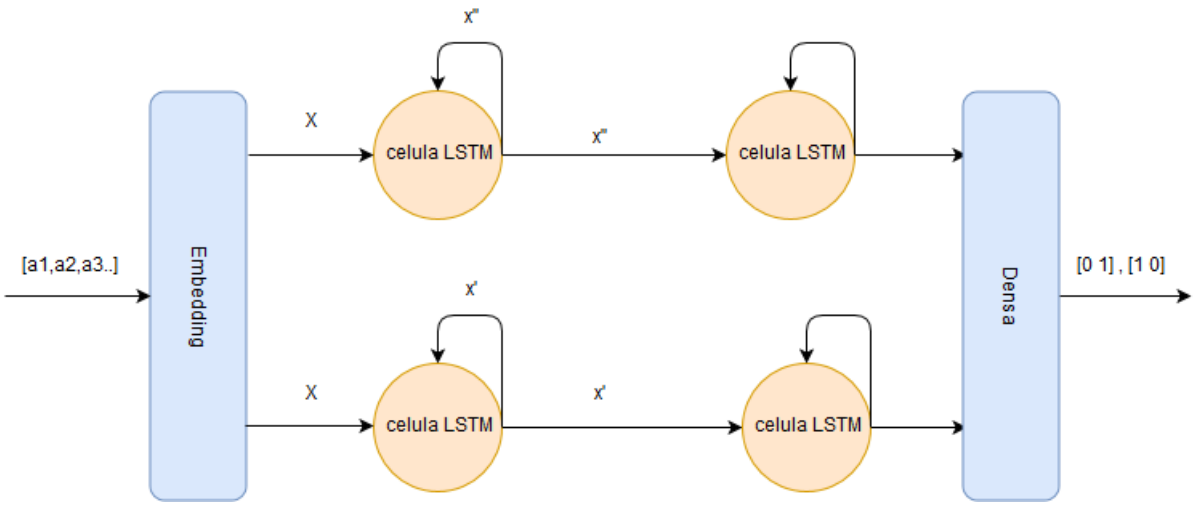



Figura 3.4: diagrama 4

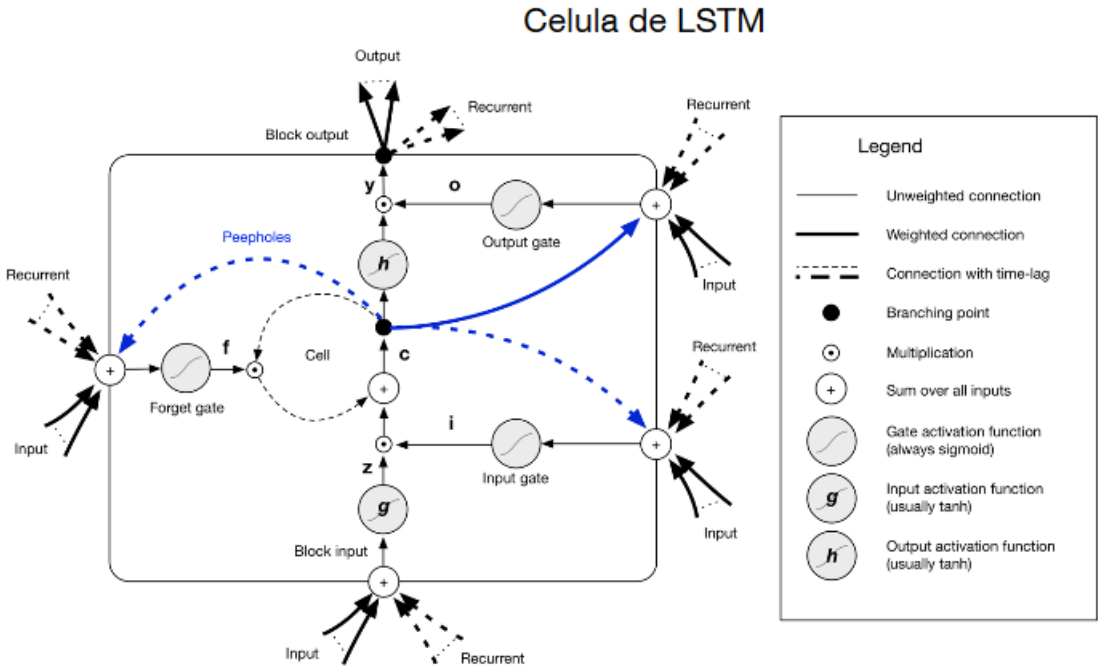



Figura 3.5: diagrama 5 [1]

el número mínimo definido es la cantidad de valores del parámetro de entrada; en el caso de el ventaneo es el ancho de la ventana. Se busca probar que configuración del número de células por capa es el ideal, para ello se probará dentro de un rango de 500 realizando pruebas con distintos valores entre el número mínimo y máximo del rango para encontrar el mejor valor.

En cuanto a la cantidad de capas dentro de la red, se define el mínimo en una dos capas de células. Se cree que es posible obtener una buena eficiencia con una red profunda de 8 capas, pero para encontrar el mejor conjunto se probará en el rango de 2n, desde 21como fue definida y hasta 25.

Además de la cantidad de células por capa y la cantidad de capas se buscará encontrar cuál función de activación dentro del modelo brinda mejores resultados. Las funciones que se explorarán son softmax, tanh, Relu y sigmoid. Para la optimización dentro del sistema se piensa analizar varios algoritmos, SGD (stochastic gradient descent), Adam, Adagrad, RMSprop y algunas variaciones de Adam. Otro punto importante es que dentro de la optimización se probará distintos ratios de aprendizaje y se probará utilizar decaimiento de ratio de aprendizaje.

Otros valores que se van a optimizar, es el tamaño de los abandonos entre capa de LSTM y du-

rante la recurrencia, el tamaño del lote antes de retropropagación y las épocas de entrenamiento con el conjunto datos.

Para poder encontrar la mejor configuración de parámetros en la red para el problema en cuestión, se realizará experimentación utilizando el conjunto de validación antes definido y variando uno a uno los parámetros mientras se deja el resto como constantes para probar de manera eficiente cuál opción obtiene el mejor resultado. Se espera que después de realizar pruebas con todos los parámetros del sistema se encuentre la mejor solución para luego poder experimentar con el conjunto de prueba.

En el diagrama 7 se puede apreciar como esta constituida cada célula individual de LSTM. Una célula de LSTM cuenta con cuatro entradas, el input normal que tiene una FFN, una entrada de bloqueo, una entrada de olvido y la compuerta de salida. Todas estas entradas reciben la información del tiempo actual y la justo anterior. Es importante denotar que en el diseño planteado en el diagrama 6 no se muestra las 4 entradas, pero se asume que las dos conexiones del tiempo anterior y siguiente conectan con estas.

Para poder comprobar si los valores que se están obteniendo a partir de la experimentación son buenos en términos estadísticos se utilizara 6 métricas diferentes:

1. Certeza = $\frac{Vp+Vn}{Vp+Vn+Fp+Fn}$
2. Presición= $\frac{Vp}{Vp+Fp}$
3. Sensibilidad= $\frac{Vp}{Vp+Fn}$
4. Especificidad= $\frac{Vn}{Vn+Fn}$
5. MCC (coeficiente de correlación de Mathew) = $\frac{Vp*Vn-Fp*Fn}{\sqrt{Vn+Fn*Vn+Fn}}$

## 3.2. Implementación

Para realizar la implementación de LSTM se utilizó un framework llamado Keras, pero para poder entender cómo funciona este es necesario primero explicar Tensorflow que es el backend

de Keras en la implementación que se desarrolló. Tensorflow es una biblioteca de código abierto desarrollada por Google y liberada el 9 de Noviembre del 2015. Tiene varias api (Interfaz de programación de aplicaciones) para desarrollo, como es el caso de python, la cual fue escogida para este proyecto porque es de fácil manejo. Tensorflow funciona con el uso de tensores como su nombre lo menciona, los cuales le permiten realizar grafos de flujo de datos con estado para representar las redes neuronales y ejecutar todos los algoritmos que se pueden plantear sobre estas. Tensorflow es reconocido como el framework líder en el mercado actual, siendo utilizada para fines industriales e investigativos a la par.

Keras por su parte es una interfaz de alto nivel que permite realizar declaraciones de redes neuronales recurrentes y convolucionales entre otras, que a la hora de ejecutarse realiza la conexión con alguno de los grandes frameworks de inteligencia artificial como es el caso de Tensorflow o Theano. Keras permite utilizar dos tipos de Modelos para la creación de arquitecturas de redes neuronales. La primera es el modelo secuencial que va uniendo las capas una a una para ejecutarse como se especifican. El segundo modelo es el funcional, este permite organizar distintas maneras de ejecución, como por ejemplo sería el ejecutar en paralelo dos capas de convolución para luego unirlas en un pooling.

Lo que hace a Keras la mejor opción para este proyecto es que permite agilidad y velocidad para la implementación y que asegura que las conexiones entre las capas son realizadas de la manera correcta sin la necesidad de ponerle mucho esfuerzo a acomodar conexiones. La ejecución de los algoritmos de forward y backpropagation son ejecutados de la manera correcta. También es importante notar que se podría haber realizado todo directamente en Tensorflow pero validar que ciertas partes de la red están correctas requiere de un mayor tiempo y no es tan fácil hacer la pila de capas de LSTM o ajustar la cantidad de células por capa en Tensorflow como se hace en Keras.

Para la implementación en Keras se utilizó tres distintas capas para poder expresar la red neuronal completa y obtener unos resultados ideales. Se describe cada una de ellas y los parámetros que contienen en esta versión. La capa de LSTM implementada en Keras tiene la siguiente descripción en código.

```
keras.layers.LSTM(units, activation='tanh',
recurrent_activation='hard_sigmoid', use_bias=True,
kernel_initializer='glorot_uniform', recurrent_initializer='orthogonal',
bias_initializer='zeros', unit_forget_bias=True, kernel_regularizer=None,
recurrent_regularizer=None, bias_regularizer=None,
activity_regularizer=None, kernel_constraint=None,
recurrent_constraint=None, bias_constraint=None,
dropout=0.0, recurrent_dropout=0.0, implementation=1,
return_sequences=False, return_state=False, go_backwards=False,
stateful=False, unroll=False)
```

Esta implementación de LSTM está realizada bajo lo descrito en el artículo de Hochreiter 1997 que es la primera descripción de una LSTM de donde Keras parte para crear la implementación de la capa con los respectivos parámetros. Los valores más importantes son las unidades, las activaciones, las inicializaciones y los abandonos. Estos valores anteriores afectan enormemente el desempeño dependiendo del problema de clasificación que se esté realizando y la profundidad que se desee obtener. Como está planteado la capa, el retorno de la secuencia lo realiza hacia el frente si se le activa el $return_sequences$. Durante la recurrencia las conexiones se realizan con la capa misma.

En cuanto a las inicializaciones, la del Kernel sigue una activación glorot uniforme que realiza una distribución normal donde el límite está calculado a partir de $\sqrt{\frac{6}{fan_in+fan_out}}$ dónde $fan_in$ es el número de unidades de entrada y $fan_out$ es el número de unidades de salida del tensor de peso. En cuanto a la inicialización recurrente se utiliza una matriz ortogonal de valores aleatorios a partir de una distribución normal. En la implementación original la inicialización del bias se usa cero para cada valor.

Hay dos funciones de activación, una para el paso recurrente y una para el paso normal. Por defecto se utilizan tangente hiperbólica para la activación durante el paso frontal normal y se utiliza sigmoidal fuerte para el paso recurrente. En cuanto a los abandonos utilizados se definen dos, uno para el paso frontal y el otro para la recurrencia. Inicialmente los abandonos están definidos en 0.

La capa densa implementada dentro de Keras tiene la siguiente descripción en código.

```
keras.layers.Dense(units, activation=None, use_bias=True, kernel_initializer='
    glorot_uniform', bias_initializer='zeros', kernel_regularizer=None,
    bias_regularizer=None, activity_regularizer=None, kernel_constraint=None,
    bias_constraint=None)
```

Una capa densa hace lo mismo que una capa de red neuronal feedforward completamente conexa, que recibe unas entradas, realiza la multiplicación con los pesos, le suma el bias y pasa los valores por la función de activación. Esta es usada para clasificar finalmente los resultados obtenidos por la LSTM y usarlos para calcular la pérdida y realizar el retropropagación. Los atributos utilizados en esta capa son: el número de unidades que tendrá la capa, el tipo de activación, si se utilizará un bias, la inicialización para el kernel y el bias, regularizador de kernel, bias, restricción de kernel y de bias. Los atributos más importantes de los anteriores son las unidades, la función de activación utilizada y las inicializaciones.

Por defecto la función de activación para la capa es None que para Keras traduce a activación lineal $f(x) = x$. En cuanto al número de unidades de cada capa, este define la cantidad de clases que van a ser clasificadas por la red. Para el caso del problema de identificación de sitios de clivaje esta capa debe contener 2 unidades para poder identificar el [1 0] o [0 1] del sitio.

La última capa por describir es la capa de embedding, esta capa tiene la siguiente descripción en código.

```
keras.layers.Embedding(input_dim, output_dim, embeddings_initializer='uniform',
    embeddings_regularizer=None, activity_regularizer=None, embeddings_constraint
    =None, mask_zero=False, input_length=None)
```

La capa de embedding logra convertir un valor entero positivo en un vector denso, esto permite optimizar los valores de entrada durante el proceso de optimización del sistema. Los atributos que recibe la capa son: dimensión de entrada, dimension de salida, inicialización de la matriz de embedding, un regularizador de actividad y embedding, una restricción, una máscara cero y el largo de la entrada. La dimensión de entrada se refiere al número total de valores posibles de entrada que tiene la capa. La dimensión de salida es el tamaño del vector denso resultante

después de la operación. El tamaño de la entrada describe la cantidad de valores por vector de la entrada. La inicialización es uniforme y no se describe inicialmente ningún valor para los demás atributos.

Para poder entrenar y probar los datos se requiere realizar un preprocesamiento de los mismos para que mantengan el formato necesario definido en la entrada de la capa de embedding. Para esto se codificó una función que lee la base de datos y la organiza en varios arreglos de numpy (librería de manejo de vectores y matrices) y los retorna. Retorna seis arreglos distintos que van en pares según su punto de uso. Los tres pares representan entrenamiento, validación y prueba. Cada par de vectores contienen los valores de referencia y los valores a clasificar. La función tiene la siguiente signatura.

```
trainx,trainy,testx,testy,Valx,Valy = createFeatureVector(positive, negative,
    size=0.2)
```

El atributo size se refiere al valor de porcentaje que tendrá el conjunto de prueba con respecto al de entrenamiento. Como la base de datos está dividida entre valores positivos y valores negativos es necesario pasar la locación de los archivos correspondientes a cada uno.

Para dejar más claro cómo se implementó la solución utilizando LSTM se mostrará el código de la solución con dos capas de LSTM para poder demostrar la manera en la que se unen las capas usando Keras.

```
model = Sequential()
model.add(Embedding(30, 128, input_length = trainx.shape[1]))
model.add(LSTM(200, dropout = 0.2, recurrent_dropout=0.3, return_sequences =
    True, unroll = True,recurrent_activation='hard_sigmoid',bias_initializer='
    RandomNormal',implementation=1))
model.add(LSTM(200, dropout = 0.2, recurrent_dropout=0.3, unroll = True,
    recurrent_activation='hard_sigmoid',bias_initializer='RandomNormal',
    implementation=1))
model.add(Dense(2,activation='softmax', bias_initializer='RandomNormal'))
```

Desde el principio se debe declarar cuál es el empaque (wrapper) que se va a utilizar ya sea el modelo secuencial o el modelo funcional para unir los bloques. Para este caso se utiliza

el secuencial dado que la ejecución de la red no requiere procesos en paralelo. Se adicionan los bloques a la pila de capas utilizando la función add que sirve de conexión entre ellas. Es importante denotar que la primera capa ya sea una capa de embedding o una capa de entrada normal deben contener el tamaño del vector a clasificar. Para poder escalar el modelo, solo debe añadirse nuevas capas utilizando la función add y Keras se encarga de conectar adecuadamente las salidas de una capa con la siguiente. Para mejorar la experimentación se utiliza una inicialización aleatoria en esta capa, Dado que originalmente la capa de bias es inicializada en cero.

Posterior a declarar la estructura del grafo de ejecución, debe declararse la optimización a utilizar, en este caso se usa adam el cual se declara de la siguiente manera.

```
keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=None, decay
    =0.0, amsgrad=False)
```

Adam es un algoritmo de optimización que utiliza gradiente estocástico, tiene una ventaja que es utilizar gradientes del primer orden y no necesitar tanta memoria para realizar sus cálculos. El método logra computar ratios de aprendizaje individuales para los distintos parámetros del sistema. En cuanto a la implementación en Keras se tiene 6 parámetros que se pueden ajustar para encontrar la mejor combinación para el problema, pero no es recomendado cambiar el beta 1 o el beta 2, pero la proporción de aprendizaje y el amsgrad (variante del algoritmo Adam descrita en el articulo .ºn the Convergence of Adam and Beyond") pueden cambiar bastante el resultado.

```
model.compile(loss = 'categorical_crossentropy', optimizer = optimizerx,metrics
    =['accuracy'])
```

Posterior a declarar que optimizador se utiliza se agrega el compilador, que hace las mismas de ajustar el grafo con la función para calcular la pérdida y el optimizador. Luego de declarar el compilador se agrega la función fit que realiza todo el entrenamiento de la red y retorna el valor de certeza y deja el grafo ajustado para poder realizar posteriores pruebas con el conjunto de validación y de prueba.

```
model.fit(trainx,trainy, batch_size=16, epochs=10,verbose=5)
```

Por último se realiza la validación y el test haciendo uso de la función $predict_classes$ en el caso del modelo secuencial.

```
model.predict_classes(testx, verbose=1, batch_size=4)
```

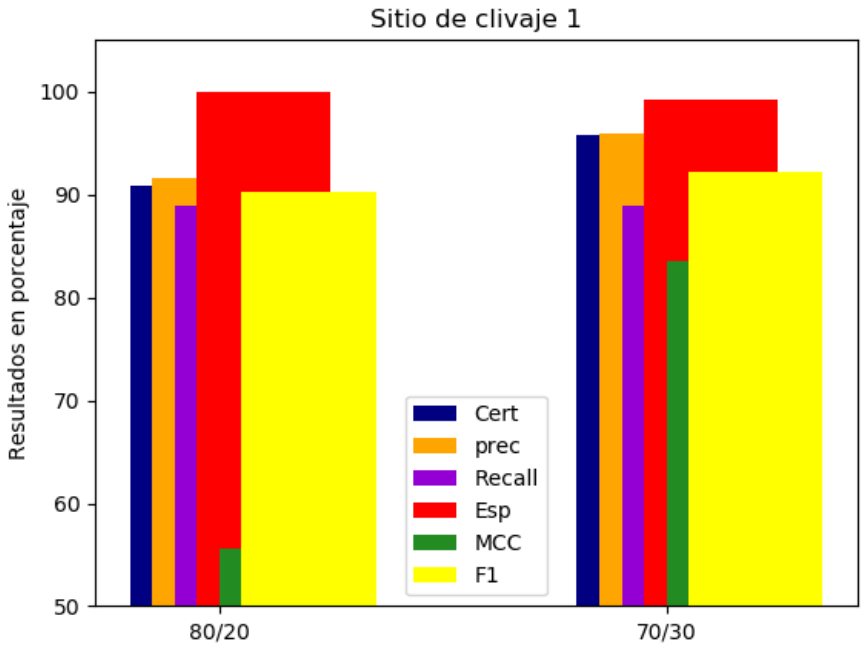
Con estas funciones se logra crear una red LSTM funcional que sirve para clasificar sitios de clivaje y puede aplicarse también a otros problemas de clasificación.

Para encontrar los valores estadísticos de la respuesta dada por la red y los valores de muestra, se creó una clase que recibe como entrada los dos vectores, el obtenido y el teórico. Con estos dos vectores crea la matriz de confusión, de la cual se obtienen todas las métricas que son necesarias para validar cada iteración de prueba.

## 3.3. Resultados

### 3.3.1. Sitio de clivaje 1

Para la obtención de los resultados para este sitio de clivaje se realizó múltiples experimentaciones con los diferentes valores declarados en el diseño de la implementación y se utilizó los diferentes estadísticos para medir el desempeño de cada implementación. A continuación se presenta el resumen estadístico del sitio de clivaje número uno.




En la gráfica siguiente se puede observar los 6 valores estadísticos para las dos distribuciones de valores positivos y negativos que se plantearon para la experimentación con LSTM. La Experimentación con distribución 80/20 tiene una especificidad mayor que la 70/30 pero un coeficiente de correlación de matthews mucho menor que la segunda, con una diferencia de 28 por ciento. Para el presente problema se considera mejor una implementación cuando tiene una mayor sensibilidad, mayor coeficiente de correlación y mayor F1. Lo anterior es tomado dado que la base de datos tiene una minoría de datos verdaderos positivos y estos valores estadísticos son determinantes para conocer qué tan bien se está dando la clasificación de la clase verdadera positiva y cómo se relaciona la clasificación de las dos clases en conjunto.

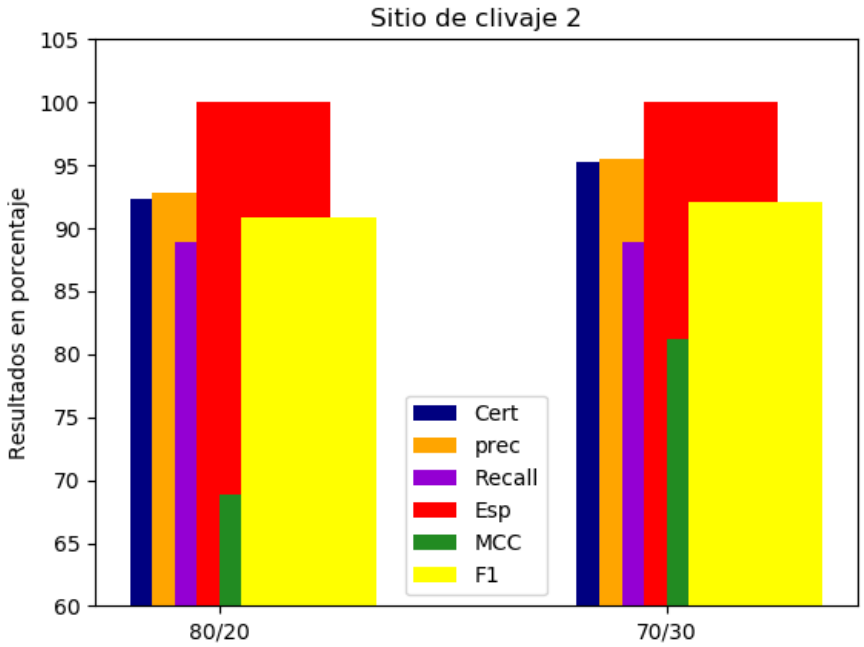
En cuanto a las arquitecturas que obtuvieron los mejores resultados, se detalla que los valores de los atributos entre las dos implementaciones en varios casos son iguales, como el número de células por capa o el número de capas. Para el sitio de clivaje 1 la mejor implementación es la que tiene distribución 70/30, dado que los valores estadísticos más relevantes presentan un mayor resultado con respecto a los de la experimentación con 80/20.

| Atributo | Valor | Modelo |
|---|---|---|
| capas_LSTM | 4 | 80/20 |
| num | 150 | |
| capas_densas | 1 | |
| optim | Nadam | |
| loss | categorical_crossentropy | |
| activ_densa | softmax | |
| batch | 32 | |
| window | 11 | |
| epochs | 1000 | |
| recurr_activ | hard_sigmoid | |
| learning_r | 0.002 | |
| amsgrad | None | |
| recurr_drop | 0.1 | |
| imp | 1 | |
| drop | 0.1 | |

| Atributo | Valor | Modelo |
|---|---|---|
| capas_LSTM | 4 | 70/30 |
| num | 150 | |
| capas_densas | 1 | |
| optim | Adagrad | |
| loss | mean_squared_error | |
| activ | softmax | |
| batch | 32 | |
| window | 11 | |
| epochs | 1000 | |
| recurr_activ | hard_sigmoid | |
| learning_r | 0.002 | |
| amsgrad | None | |
| recurr_drop | 0.1 | |
| imp | 1 | |
| drop | 0.3 | |

### 3.3.2. Sitio de clivaje 2

Para el segundo sitio de clivaje se realizó una experimentación similar a la del sitio 1 en donde se varió los distintos parámetros de las redes buscando mejorar los valores estadísticos resultantes. La gráfica resumiendo los resultados es la siguiente:




Los valores en cuanto a especificidad y ha F1 son bastantes similares pero es correcto observar que en cuanto a MCC la versión de 70/30 se desempeña mejor la versión de 80 20. Por lo antes expuesto el desempeño se mide bajo el comportamiento general de las estadísticas, entonces la arquitectura que se desempeña mejor en este sitio es la que fue entrenada con la distribución 70/30.

En las Siguientes tablas se muestran los valores de los parámetros finales después de la experimentación, para el sitio de clivaje 2.

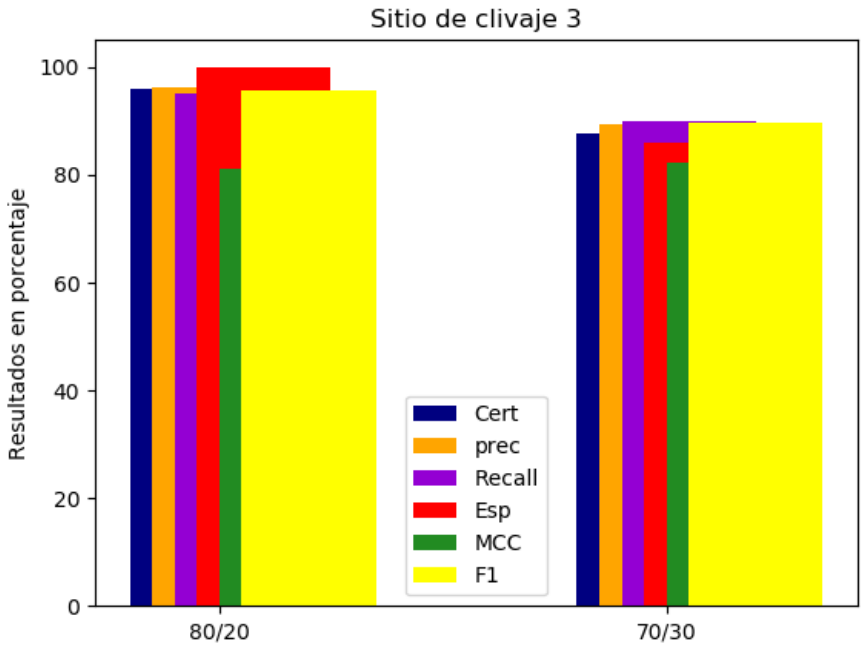
| Atributo | Valor |
|---|---|
| capas_LSTM | 4 |
| num | 500 |
| capas_densas | 1 |
| optim | adam |
| loss | mean_squared_error |
| activ | softmax |
| batch | 32 |
| window | 11 |
| epochs | 1000 |
| recurr_activ | hard_sigmoid |
| learning_r | 0.001 |
| amsgrad | None |
| recurr_drop | 0.1 |
| imp | 1 |
| drop | 0.1 |

Modelo 80/20

| Atributo | Valor |
|---|---|
| capas_LSTM | 2 |
| num | 200 |
| capas_densas | 1 |
| optim | Nadam |
| loss | poisson |
| activ | softmax |
| batch | 32 |
| window | 11 |
| epochs | 15 |
| recurr_activ | hard_sigmoid |
| learning_r | 0.002 |
| amsgrad | None |
| recurr_drop | 0.1 |
| imp | 1 |
| drop | 0.1 |

Modelo 70/30

### 3.3.3. Sitio de clivaje 3




Los valores de la implementación de 70/30, se reducen bastante para el sitio de clivaje 3 en comparación a los resultados de los sitios 1 y 2. Contando con un solo valor alcanzando el 90 %, el resto están por debajo de este límite. El único valor que posee la implementación de 70/30 que tiene un mejor desempeño con respecto a la otra es el MCC que tiene 1.20 por ciento más.

El desempeño de la versión con distribución 80/20 es considerablemente mejor que la la segunda, obteniendo mejores valores en 5 de 6 estadísticas. La implementación de 80/20 converge rápido al valor, solo necesita de 10 épocas de entrenamiento para alcanzar su valor, por el contrario la segunda necesita de 1000 épocas y no alcanza un mejor resultado a pesar de tener tantos ciclos de entrenamiento.

| Atributo | Valor |
|---|---|
| capas_LSTM | 2 |
| num | 300 |
| capas_densas | 1 |
| optim | adagrad |
| loss | categorical_crossentrpy |
| activ | softmax |
| batch | 32 |
| window | 10 |
| epochs | 10 |
| recurr_activ | hard_sigmoid |
| learning_r | 0.001 |
| amsgrad | None |
| recurr_drop | 0.1 |
| imp | 1 |
| drop | 0.1 |

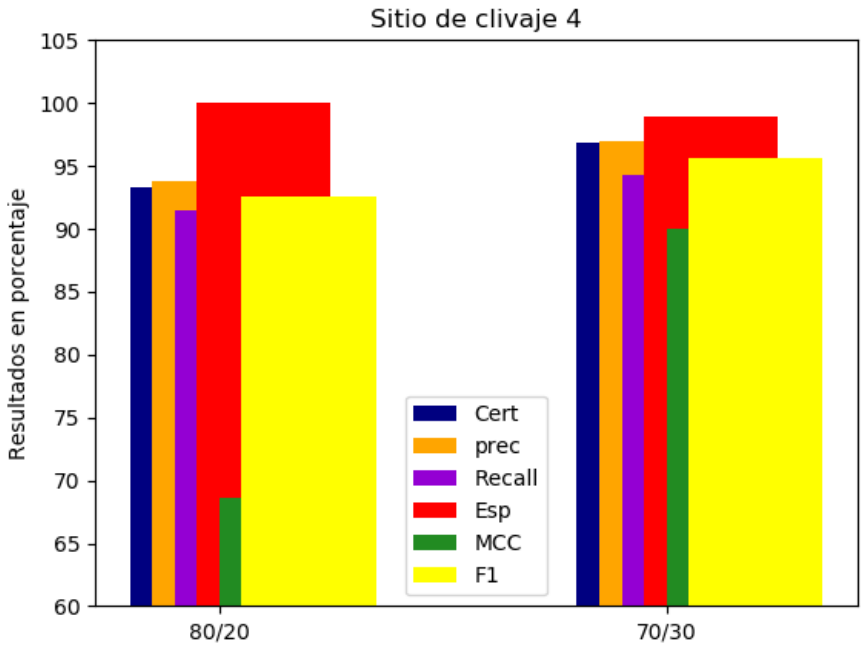
Modelo 80/20

| Atributo | Valor |
|---|---|
| capas_LSTM | 4 |
| num | 200 |
| capas_densas | 1 |
| optim | adam |
| loss | categorical_crossentropy |
| activ | softmax |
| batch | 32 |
| window | 10 |
| epochs | 1000 |
| recurr_activ | hard_sigmoid |
| learning_r | 0.001 |
| amsgrad | None |
| recurr_drop | 0.1 |
| imp | 1 |
| drop | 0.2 |

Modelo 70/30

### 3.3.4. Sitio de clivaje 4

Para el cuarto de clivaje se obtienen resultados similares para la mayor parte de los estadísticos. Como se puede observar en la gráfica, la implementación uno y dos se comportan similar en cuanto ha recall y especificidad, siendo la segunda ligeramente mejor. La diferencia es un poco mayor para la certeza y la precisión siendo la segunda implementación mejor que la primera. En donde se obtiene una diferencia considerable es en F1 y MCC donde la distribución 70/30 supera con creces a la primera. El MCC de 80/20 es 68.57 por ciento en cambio el de 70/30 es 90.01 por ciento.



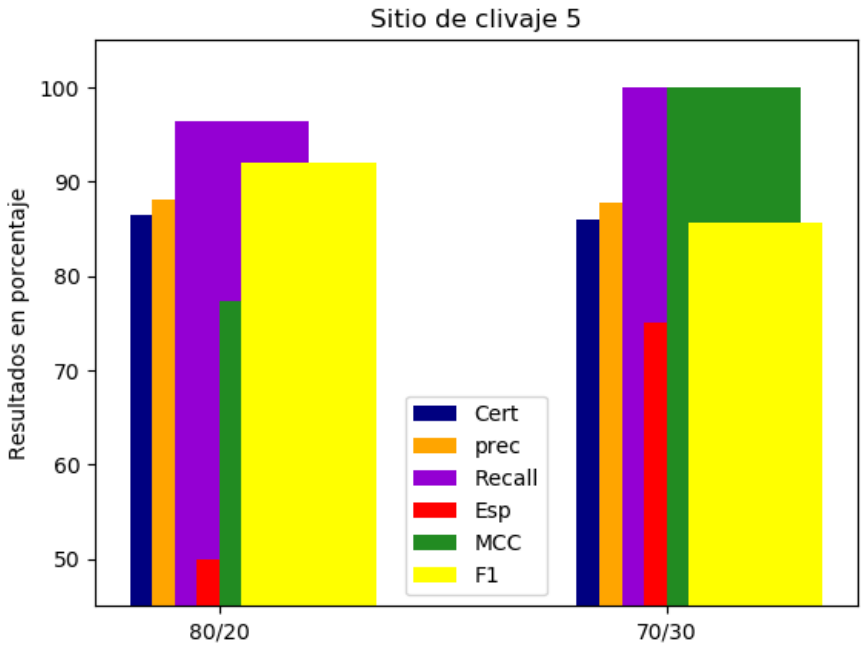

En las tablas siguientes se muestra los valores de los parámetros utilizados en las versiones finales para cada una de las implementaciones.

| Atributo | Valor | Modelo |
|---|---|---|
| capas_LSTM | 2 | 80/20 |
| num | 500 | |
| capas_densas | 1 | |
| optim | adam | |
| loss | categorical_crossentrpy | |
| activ | softmax | |
| batch | 32 | |
| window | 9 | |
| epochs | 50 | |
| recurr_activ | hard_sigmoid | |
| learning_r | 0.01 | |
| amsgrad | None | |
| recurr_drop | 0.1 | |
| imp | 1 | |
| drop | 0.1 | |

| Atributo | Valor | Modelo |
|---|---|---|
| capas_LSTM | 2 | 70/30 |
| num | 500 | |
| capas_densas | 1 | |
| optim | Nadam | |
| loss | categorical_crossentropy | |
| activ | softmax | |
| batch | 32 | |
| window | 9 | |
| epochs | 20 | |
| recurr_activ | tanh | |
| learning_r | 0.002 | |
| amsgrad | None | |
| recurr_drop | 0.2 | |
| imp | 1 | |
| drop | 0.1 | |

### 3.3.5. Sitio de clivaje 5

El sitio de clivaje 5 presenta una anomalía con respecto a los demás, es el único en el cual se aprende mejor la clase con menos muestras. El recall en la implementación con 70/30 es de 100 por ciento, en consecuencia el MCC también. La implementación que tiene un mejor desempeño para el sitio de clivaje 5 es la de 70/30 dado que la versión de 80/20 clasifica muy mal los casos negativos, mis clasificando 50 por ciento de los casos, lo que lo hace una solución aleatoria en cuanto a la clase negativa para el problema.



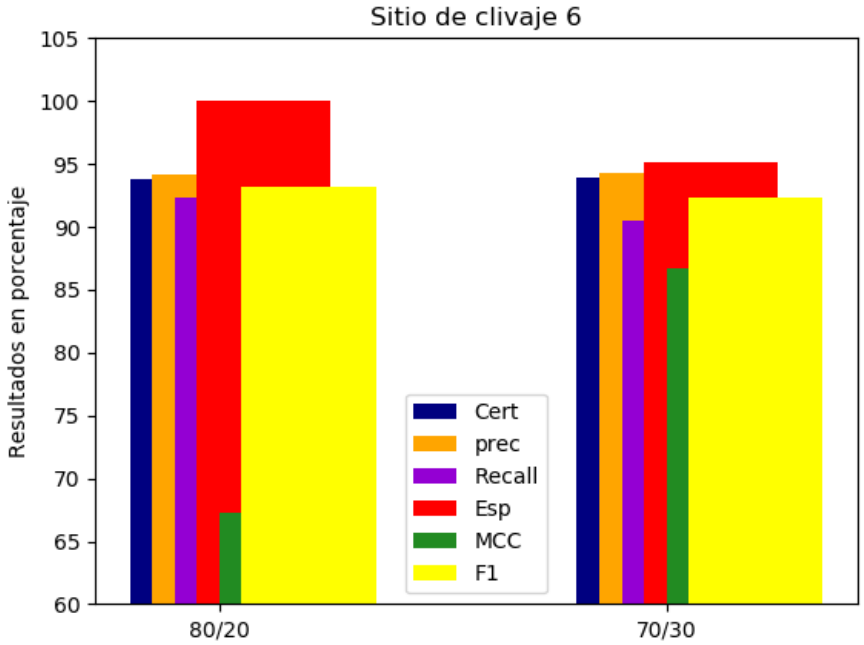

En las tablas siguientes se muestra los valores de los parámetros utilizados en las versiones finales para cada una de las implementaciones del sitio de clivaje 5.

| Atributo | Valor | Modelo |
|---|---|---|
| capas_LSTM | 8 | 80/20 |
| num | 200 | |
| capas_densas | 1 | |
| optim | adam | |
| loss | poisson | |
| activ | softmax | |
| batch | 32 | |
| window | 11 | |
| epochs | 15 | |
| recurr_activ | hard_sigmoid | |
| learning_r | 0.001 | |
| amsgrad | None | |
| recurr_drop | 0.1 | |
| imp | 1 | |
| drop | 0.1 | |

| Atributo | Valor | Modelo |
|---|---|---|
| capas_LSTM | 8 | 70/30 |
| num | 40 | |
| capas_densas | 1 | |
| optim | adam | |
| loss | categorical_crossentropy | |
| activ | softmax | |
| batch | 16 | |
| window | 11 | |
| epochs | 20 | |
| recurr_activ | hard_sigmoid | |
| learning_r | 0.001 | |
| amsgrad | None | |
| recurr_dropAms | 0.1 | |
| imp | 1 | |
| drop | 0.1 | |

### 3.3.6. Sitio de clivaje 6

Para el sitio de clivaje 6 se observa que en cuanto a los valores de F1, Especificidad y recall se observa un mejor resultado para la implementación 80/20. Pero el MCC es mucho menor en el caso de esta con respecto a la de 70/30, lo que hace a la segunda una implementación que aprenda a clasificar mejor, esto dado que el MCC es una estadística con mayor valor dado que toma el espectro completo de la matriz de confusión. También es bueno notar que la diferencia en las métricas que generales como lo son precisión y certeza los resultados son muy cercanos.



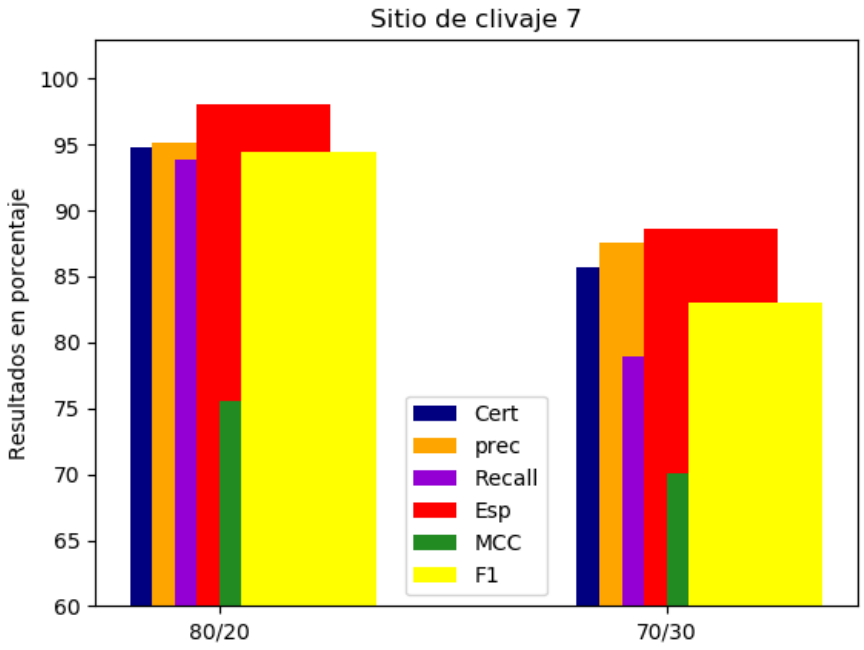

En las tablas siguientes se muestra los valores de los parámetros utilizados en las versiones finales para cada una de las implementaciones del sitio de clivaje 6.

| Atributo | Valor | Modelo |
|---|---|---|
| capas_LSTM | 2 | 80/20 |
| num | 200 | |
| capas_densas | 1 | |
| optim | adam | |
| loss | poisson | |
| activ | softmax | |
| batch | 32 | |
| window | 9 | |
| epochs | 1000 | |
| recurr_activ | hard_sigmoid | |
| learning_r | 0.01 | |
| amsgrad | None | |
| recurr_drop | 0.1 | |
| imp | 1 | |
| drop | 0.1 | |

| Atributo | Valor | Modelo |
|---|---|---|
| capas_LSTM | 2 | 70/30 |
| num | 300 | |
| capas_densas | 1 | |
| optim | adagrad | |
| loss | categorical_crossentropy | |
| activ | softmax | |
| batch | 32 | |
| window | 9 | |
| epochs | 50 | |
| recurr_activ | relu | |
| learning_r | 0.01 | |
| amsgrad | None | |
| recurr_dropAms | 0.1 | |
| imp | 1 | |
| drop | 0.1 | |

### 3.3.7. Sitio de clivaje 7

La implementación con distribución 80/20 se desempeña mucho mejor en todas las estadísticas con respecto a la de 70/30. La arquitectura resultante para las dos distribuciones es la siguiente.

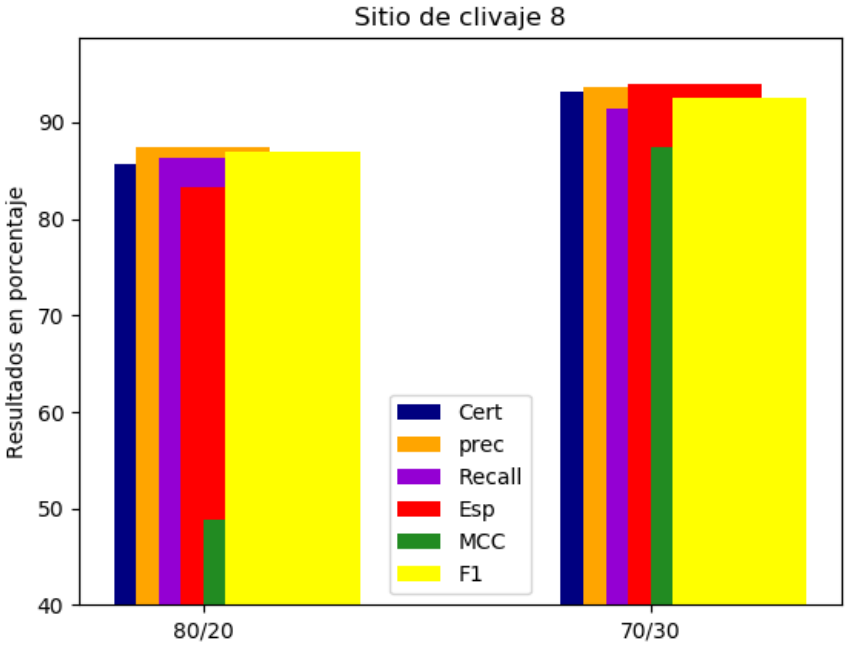


| Atributo | Valor | Modelo |
|---|---|---|
| capas_LSTM | 2 | 80/20 |
| num | 300 | |
| capas_densas | 1 | |
| optim | adam | |
| loss | categorical_crossentropy | |
| activ | softmax | |
| batch | 32 | |
| window | 10 | |
| epochs | 50 | |
| recurr_activ | hard_sigmoid | |
| learning_r | 0.1 | |
| amsgrad | Yes | |
| recurr_drop | 0.1 | |
| imp | 1 | |
| drop | 0.1 | |

| Atributo | Valor | Modelo |
|---|---|---|
| capas_LSTM | 2 | 70/30 |
| num | 100 | |
| capas_densas | 1 | |
| optim | adam | |
| loss | categorical_crossentropy | |
| activ | softmax | |
| batch | 32 | |
| window | 10 | |
| epochs | 1000 | |
| recurr_activ | hard_sigmoid | |
| learning_r | 0.1 | |
| amsgrad | None | |
| recurr_dropAms | 0.1 | |
| imp | 1 | |
| drop | 0.01 | |

### 3.3.8. Sitio de clivaje 8

Para el sitio de clivaje, la implementación con distribución 70/30 tienen un mejor desempeño que la 80/20, dado que en todas las estadísticas tienen mejores resultados. Se puede notar que en la distribución 80/20 a pesar de no tener resultados tan bajos en cuanto a la mayoría de las métricas, el MCC está por debajo del 50 por ciento y como se explicó arriba el MCC es la métrica más representativa para un problema de clasificación binaria como este.



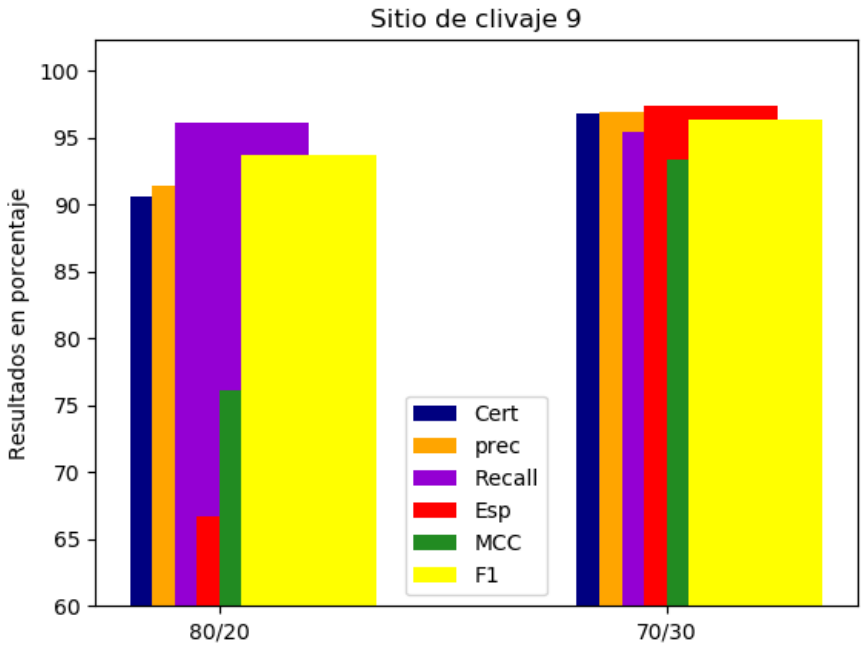

En las tablas siguientes se muestra los valores de los parámetros utilizados en las versiones finales para cada una de las implementaciones del sitio de clivaje 7.

| Atributo | Valor | Modelo |
|---|---|---|
| capas_LSTM | 2 | 80/20 |
| num | 10 | |
| capas_densas | 1 | |
| optim | Nadam | |
| loss | categorical_crossentropy | |
| activ | softmax | |
| batch | 32 | |
| window | 11 | |
| epochs | 15 | |
| recurr_activ | hard_sigmoid | |
| learning_r | 0.002 | |
| amsgrad | None | |
| recurr_drop | 0.1 | |
| imp | 1 | |
| drop | 0.1 | |

| Atributo | Valor | Modelo |
|---|---|---|
| capas_LSTM | 4 | 70/30 |
| num | 200 | |
| capas_densas | 1 | |
| optim | Nadam | |
| loss | categorical_crossentropy | |
| activ | softmax | |
| batch | 32 | |
| window | 11 | |
| epochs | 1000 | |
| recurr_activ | hard_sigmoid | |
| learning_r | 0.002 | |
| amsgrad | None | |
| recurr_dropAms | 0.1 | |
| imp | 1 | |
| drop | 0.5 | |

### 3.3.9. Sitio de clivaje 9

Los resultados para el sitio de clivaje 9 muestran un desbalance entre sensitividad y especificidad para la prueba con porcentaje 80/20, en cambio muestra un resultado bastante bueno para la prueba con porcentajes 70/30, con todas las métricas superando el 90 por ciento. Es claro que la segunda implementación es mejor que la primera. Los valores de los parámetros de las dos pruebas son los siguientes:



| Atributo | Valor |
|---|---|
| capas_LSTM | 4 |
| num | 40 |
| capas_densas | 1 |
| optim | adagrad |
| loss | categorical_crossentropy |
| activ | softmax |
| batch | 32 |
| window | 11 |
| epochs | 1000 |
| recurr_activ | hard_sigmoid |
| learning_r | 0.001 |
| amsgrad | None |
| recurr_drop | 0.1 |
| imp | 1 |
| drop | 0.1 |

Modelo 80/20

| Atributo | Valor |
|---|---|
| capas_LSTM | 4 |
| num | 200 |
| capas_densas | 1 |
| optim | adam |
| loss | categorical_crossentropy |
| activ | softmax |
| batch | 32 |
| window | 11 |
| epochs | 1000 |
| recurr_activ | hard_sigmoid |
| learning_r | 0.0001 |
| amsgrad | None |
| recurr_dropAms | 0.3 |
| imp | 1 |
| drop | 0.1 |

Modelo 70/30

# Capítulo 4

# CNN

## 4.1. Planteamiento del experimento

Para este segundo experimento, realizaremos una aproximación para solucionar el problema de encontrar sitios de clivaje de manera automática. Para poder realizar la aproximación se construye primero un diseño de la solución y se definen los parámetros que se deben tener en cuenta para construir una buena red neuronal convolucional. A continuación se describe el diseño planteado para la solución utilizando CNN.

En el diagrama 8 se muestra el funcionamiento de la entrada de datos a la red y la salida esperada después de entrenamiento al realizar la validación. Para el experimento por la naturaleza de las redes convolucionales la entrada necesariamente debe ser como mínimo un arreglo para poder realizar la operación de convolución en una dimensión. Entonces se utilizará un sistema de ventanas sobre la muestra original, dividiéndola en arreglos de tamaño uniforme N. Las muestras son presentadas durante el entrenamiento una a una para que la red las procese y como resultado al procesamiento la red convolucional retorna un valor [1,0] o [0,1] representando el que posea un sitio de clivaje o no para poder realizar la optimización de los pesos en las conexiones de la misma.

Posterior a la obtención de un resultado procesado, este se anexa a una lista de resultados,

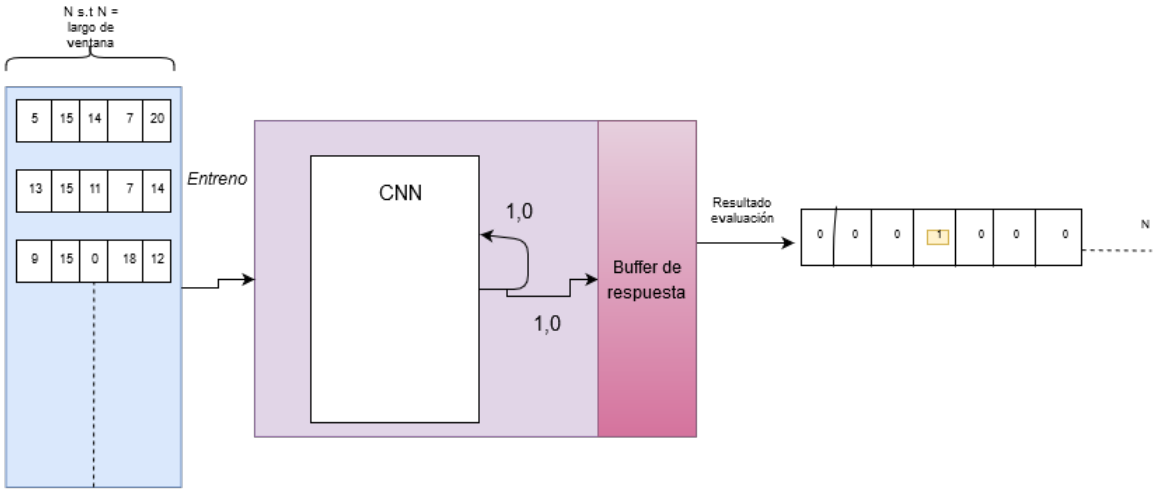



Figura 4.1: diagrama 8

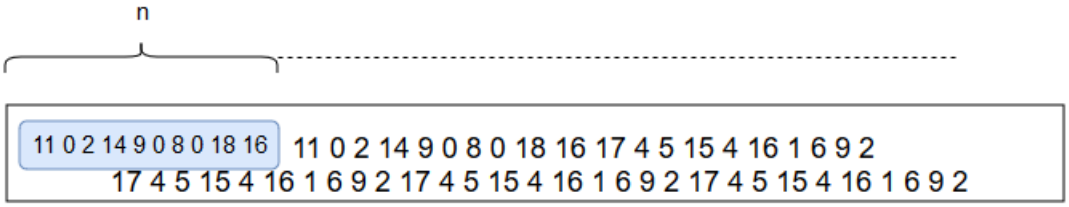



Figura 4.2: diagrama 9

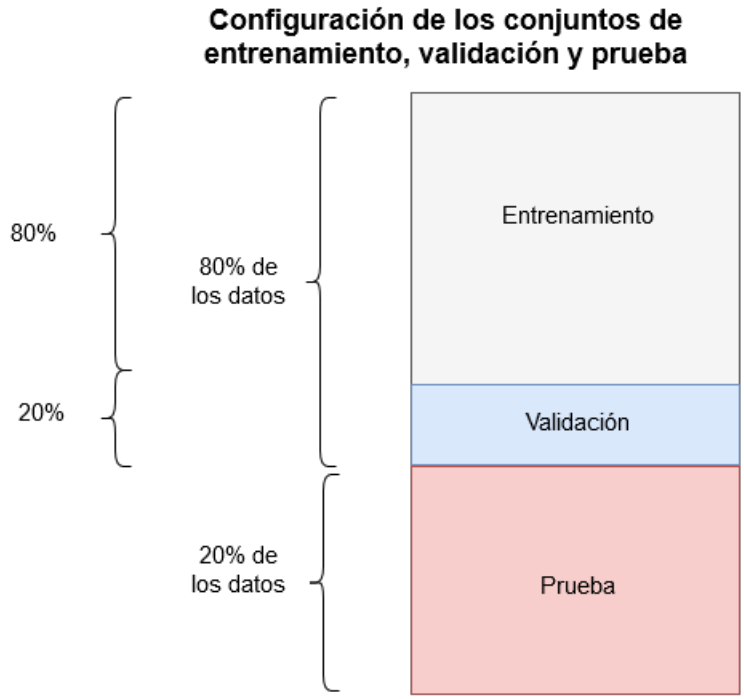



Figura 4.3: diagrama 10

que siguiente es utilizada para comparar con los valores teóricos y poder determinar que tan bien está clasificando la implementación. Para realizar el testeo de la red, se utilizara el mismo sistema, solo que en esta ocasión no se realizará optimizaciones posteriores a la red, sino que solo se presentará los datos a clasificar. Utilizando el buffer de salida para acumular las respuestas se pasa a comparar los resultados con los valores teóricos. La forma en la que se realiza el ventaneo es igual a la propuesta en el diseño de LSTM (Diagrama 4).

En cuanto a la distribución de la base de datos, se pretende utilizar una distribución 80 a 20 en cuanto al número de datos para realizar entrenamiento y prueba de la red. Dentro del conjunto de entrenamiento se subdivide en dos conjuntos el de entrenamiento como tal y el de validación para poder probar cada iteración con un conjunto de datos. Es importante notar que para esta implementación se mantiene el problema del desbalance en los datos, donde en la base de datos global, el número de casos positivos es de menos del 1 % con respecto al de casos negativos, entonces es necesario que se compruebe con una distribución de datos entre los casos negativos y positivos de 70 a 30 % que es una distribución que no presenta un desbalance grande entre

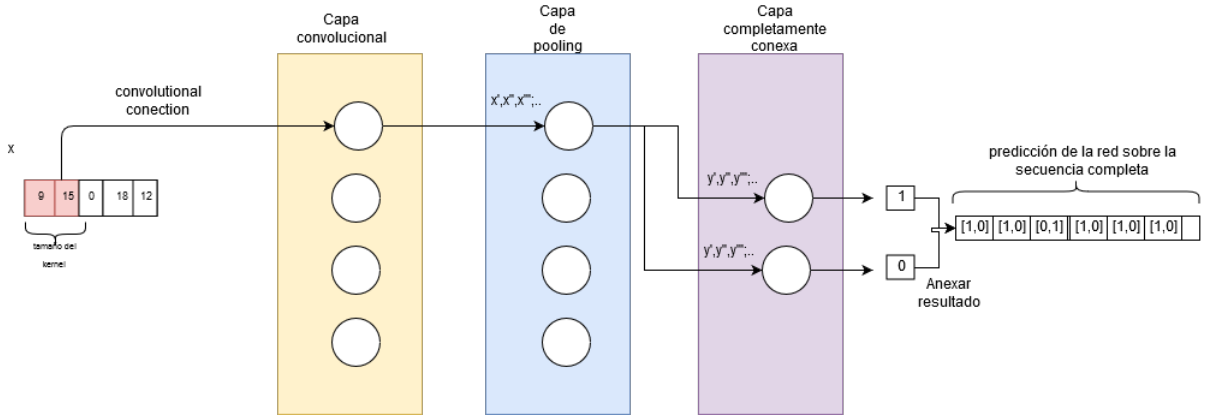



Figura 4.4: diagrama 11

las dos clases utilizadas.

En el diagrama 11 se detalla una red neuronal convolucional simple, con una sola capa de convolución, una capa de pooling y una capa completamente conexa. Este es el modelo base del cual se pretende partir para encontrar una buena aproximación al problema. Para este tipo de red neuronal se tienen los siguientes parámetros que se pueden variar: el tamaño del kernel, el número de filtros, la función de activación en cada capa, el número de capas de convolución, el número de capas de pooling y el número de capas completamente conexas. Además de esto la forma en la que se combinan las capas entre sí.

Para encontrar la mejor combinación de capas se implementará tres arquitecturas de red convolucional. Primero una red sencilla con una capa de Embedding, segundo una capa convolucional, tercero una max pooling, cuarto una completamente conexa y por último una capa completamente conexa de salida. La segunda será implementar una red inspirada en GoogleNet según como se describe en el artículo "Google deeper with convolutions"[2]. Esta red utiliza una capa de "inception" como lo describen en el artículo, donde se realiza en paralelo varias convoluciones y poolings antes de aplicarle una capa completamente conexa y por último una de salida. En la tercera implementación, se realizará una red inspirada en VGGnet (Visual Geometry Group net) que contará con dos bloques de profundización y su respectiva capa de salida. El bloque de profundización será una convolución con kernel 3 y un maxpooling [3].

## 4.2. Implementación

Para poder realizar la implementación de la red neuronal convolucional, se debe entender un poco del framework (marco de referencia) en el que se va a realizar y el cómo se estructura la red dentro de este. En el capítulo de implementación de LSTM se describe de manera general el funcionamiento de Keras y Tensorflow, para esta implementación se utilizará el mismo framework, entonces para referencias en cuanto al comportamiento básico del mismo regresar a este capítulo.

En este capítulo se describe la forma en la que Keras utiliza las capas de convolución, pooling y flatten que componen una red neuronal convolucional y siguiente se explica cómo se realizó la implementación de las arquitecturas descritas en el capítulo anterior. La capa de convolución en Keras tiene la siguiente signatura:

```
Conv1D(filters, kernel_size, strides=1, padding='valid', data_format='
    channels_last', dilation_rate=1, activation=None, use_bias=True,
    kernel_initializer='glorot_uniform', bias_initializer='zeros',
    kernel_regularizer=None, bias_regularizer=None, activity_regularizer=None,
    kernel_constraint=None, bias_constraint=None)
```

Los parámetros principales son: los filtros, el tamaño del kernel y los strides. Como es una capa de convolución de una dimensión, el kernel es de una dimensión también, esto representa la cantidad de datos que saca por filtro, en el caso de ser un kernel de 2 utilizaría dos valores por filtro. Filtros son la cantidad de sub espacios que toma por cada entrada, que luego son pasados a la siguiente capa. El stride representa el desplazamiento que realiza la convolución durante la misma.

La segunda capa que se requiere es la de pooling, esta está codificada de la la siguiente manera:

```
MaxPool1D(pool_size=2, strides=None, padding='valid', data_format='channels_last
    ')
```

Los parámetros de esta capa son: el tamaño del pool, el número de stride, si se utiliza padding y el formato de la información. Cabe aclarar que Keras tiene también una implementación

para pooling en 2D pero dado que el problema que se está tratando es en una dimensión no es necesario utilizarlo.

Por último tenemos la capa de flatten, esta capa permite convertir un resultado matricial en su respectivo valor en forma de vector, lo anterior se utiliza para poder pasar el resultado del pooling a la capa completamente conexa. El código para esta capa es muy sencillo:

```
Flatten()
```

Teniendo estos bloques para construir, se pueden unir de distintas maneras para crear las implementaciones diseñadas en el capítulo anterior. En cuanto a la implementación vainilla y la inspirada en la VGGnet se utiliza el modelo secuencial, en cambio para la inspirada en Googlenet es necesario utilizar el modelo funcional. Para una descripción detallada del modelo secuencial, leer el capítulo de implementación de LSTM.

El modelo funcional, permite declarar estructuras más complejas de ejecución, como es el caso de una capa de inception de la Googlenet, Una capa de inception contiene varias convoluciones en paralelo de una misma entrada para luego aplicarles maxpooling a las salidas de todas las capas. El modelo funcional permite realizar declaraciones de la siguiente manera:

```
tower_1 = Conv1D(64, 1, padding='same', activation='relu')(input_c)
tower_1 = Conv1D(64, 3, padding='same', activation='relu')(tower_1)
```

En este código se estan juntando dos convoluciones en una misma pila, recibiendo de entrada el vector original y la segunda convolución los valores de la anterior. Para unir las distintas convoluciones en una misma se utiliza una capa de concatenación, esta se describe de la siguiente forma :

```
keras.layers.concatenate([tower_1, tower_2, tower_3], axis = -1)
```

Se le asigna un vector con las distintas capas especificadas y esta capa permite unir todas las respuestas anteriores en una sola para poder pasarlas en paralelo a la capa siguiente. En el caso de la inception sería una capa densa que realiza la clasificación según los valores obtenidos.

En el modelo funcional, no existe la función de predict_classes como en el modelo secuencial, entonces para poder identificar las clases se debe utilizar una función llamada predict que

retorna la probabilidad de pertenecer a una clase dado el vector de entrada. Para encontrar la clase a la que pertenece se debe aplicar un argmax (índice del vector que posea el valor máximo) al vector de probabilidad obtenido, como se muestra en el código siguiente:

```
r  = model.predict(x, verbose=1, batch_size=4)
r = np.argmax(r, axis=1)
```

Para encontrar los valores estadísticos de la respuesta dada por la red y los valores de muestra, se creó una clase que recibe como entrada los dos vectores, el obtenido y el teórico. Con estos dos vectores crea la matriz de confusión, de la cual se obtienen todas las métricas que son necesarias para validar cada iteración de prueba.

Para la implementación de la red vainilla y la inspirada en VGG, las capas completamente conexas son representadas utilizando la capa densa que trae implementada Keras, cuya explicación está en el capítulo de implementación de LSTM, dado que es la misma.

En cuanto a la forma en la que se estructuraron las tres implementaciones, fue la siguiente. Para la red convolucional simple se agregó un bloque de convolución con kernel 2, luego se paso ese valor por un maxpooling, al resultado del pool se le aplica un flatten, que luego es alimentado a una capa completamente conexa, esta capa luego pasa a una capa densa de 2 nodos que clasifica [1 0] [0 1].

Para implementar la inspirada en VGG se utilizó dos bloques de convolución pooling utilizando kernel de tamaño 3 y pooling de tamaño 2. Los valores anteriores se les hace el flattening y luego son clasificados con la capa densa. En cuanto a la implementación inspirada en Google net se utilizó un bloque de inception como se describe anteriormente y una capa densa para clasificación. Todas las implementaciones cuentan con una capa de embedding antes de iniciar las respectivas convoluciones.

# Capítulo 5

# Conclusion

## 5.1. Summary of Thesis Achievements

Summary.

## 5.2. Applications

Applications.

## 5.3. Future Work

Future Work.

# Bibliografía

[1] Josh Patterson and Adam Gibson. *Deep Learning: A Practitioner's Approach*. .º'Reilly Media, Inc.", 2017.

[2] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[3] Limin Wang, Sheng Guo, Weilin Huang, and Yu Qiao. Places205-vggnet models for scene recognition. *arXiv preprint arXiv:1508.01667*, 2015.