Sumario

[Dataset 1](#__RefHeading___Toc1178_3774321295)

[LSTM + restos Conv 16](#__RefHeading___Toc1264_1465717332)

[Sin uso de embeddings 18](#__RefHeading___Toc1181_3774321295)

[Efecto del número de capas 18](#__RefHeading___Toc1186_3774321295)

[Efecto función de activación final 18](#__RefHeading___Toc1183_3774321295)

[Efecto del batch\_size 19](#__RefHeading___Toc605_4015247920)

[Efecto de la longitud de la secuencia 20](#__RefHeading___Toc847_3133881978)

[Efecto de la dimensión de embeddings 21](#__RefHeading___Toc607_4015247920)

[Base 1 capa 23](#__RefHeading___Toc611_4015247920)

[Efecto en el número de celdas, batch y reg en una capa 24](#__RefHeading___Toc613_4015247920)

[Efecto del gradient clipping 25](#__RefHeading___Toc615_4015247920)

[Efecto de la regularización en dos capas 26](#__RefHeading___Toc619_4015247920)

[Normalización de capa vs Normalización batch después de LSTM y Dropout 30](#__RefHeading___Toc851_3133881978)

[Características adicionales 31](#__RefHeading___Toc1191_3774321295)

[Redes Conv1D 32](#__RefHeading___Toc621_4015247920)

[Papers 33](#__RefHeading___Toc754_3590608229)

[Temporal Convolutional Networks (TCN) 33](#__RefHeading___Toc1034_1923091505)

[LSTM-FCM 35](#__RefHeading___Toc840_3133881978)

[InceptionTime 36](#__RefHeading___Toc1036_1923091505)

[LSTM and 1-D Convolutional Neural Networks for Predictive Monitoring of the Anaerobic Digestion Process 36](#__RefHeading___Toc842_3133881978)

[LSTM and Convolution Networks exploration for Parkinson’s Diagnosis 36](#__RefHeading___Toc623_4015247920)

[Automated diagnosis of arrhythmia using combination of CNN and LSTM techniques with variable length heart beats 36](#__RefHeading___Toc1104_3990033325)

[Deep learning based multi-temporal crop classification 37](#__RefHeading___Toc844_3133881978)

[Convolutional Recurrent Neural Networks 37](#__RefHeading___Toc756_3590608229)

[Machine-learned prediction of annual crop planting in the U.S. Corn Belt based on historical crop planting maps 37](#__RefHeading___Toc1188_3774321295)

# Dataset

train/test split: 30%

Optimización de la secuencia

\* Reduce number of parameters of the embedding; embedding\_size = vocab\_size/2

Loss on test data: [2.915931224822998, 0.4563530683517456]

La red está sobre ajustando, modificar parámetros

\* La proporción sigue muy imbalanced, increase the rate for the underrepresented categories

\* Probar con diferenes tamaños de secuencia

\* Chage split to structured and increase test split to 0.3

\* tune the batch size

\* Change activation to relu

Hay un problma con el relu, si se utiliza una función de activación unbounded, los pesos dentro de las células LSTM se escapan a infinito, la función de pérdida se dispara y el modelo pierde precisión. Opciones:

- Normalización batch antes de la célula LSTM si antes viene una salida outbounded

- Normalización batch antes de la activación, desactivar de la céluna y poner esquema:

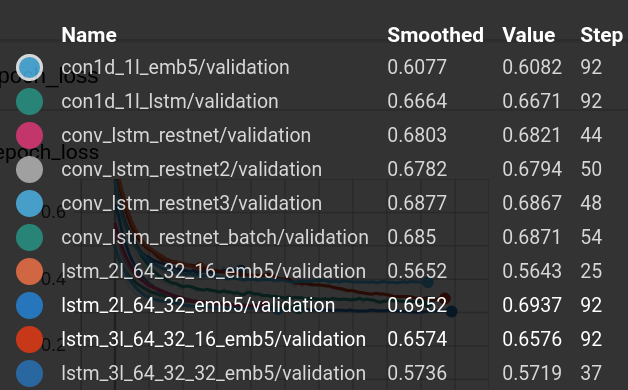
LSTM > batch > activación (relu)

===Train/test size: 14432, 6186

Número de características: 6

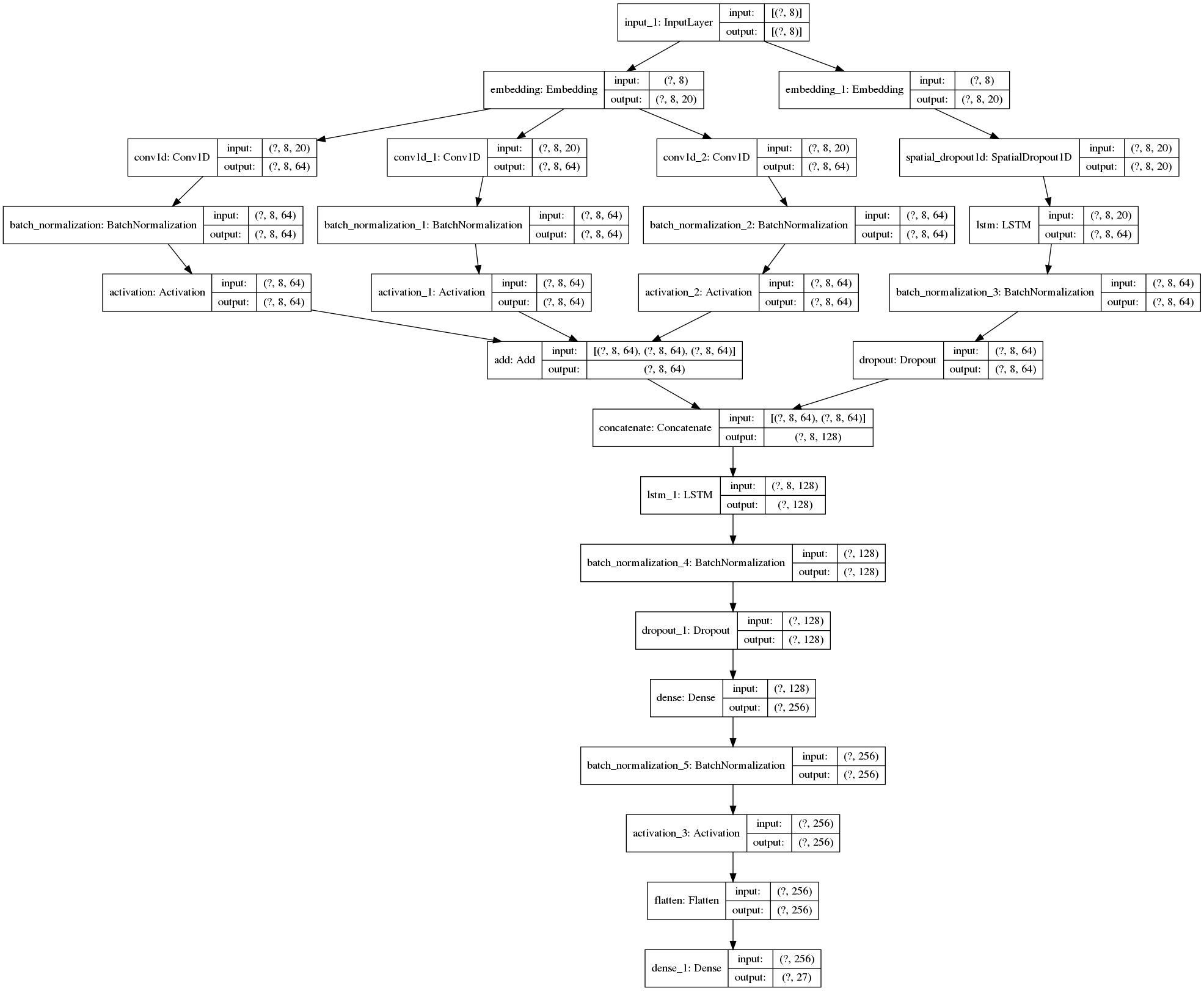
Clasificador base:

|  |
| --- |
| **Modelo base**  Lanzamos TPOT para obtener un clasificador automático  Best pipeline: ExtraTreesClassifier(input\_matrix, bootstrap=False, criterion=entropy, max\_features=0.7500000000000001, min\_samples\_leaf=10, min\_samples\_split=13, n\_estimators=100)  0.5094219775566378 |
| **Reducir número de celdas de LSTM**, reducir el número de celdas para evitar que el modelo sobreajuste a los datos, bajar de 50 a 30  model.add(Embedding(vocab\_size, embedding\_size,input\_length=sequence\_length)) model.add(LSTM(**30**)) model.add(Dense(vocab\_size, activation='sigmoid')  Epoch 220/300  loss: 1.0106  accuracy: 0.6650  val\_loss: 1.7342  val\_accuracy: 0.5099  Conclusiones  La red se siguen sobreajustando, aumenta el tiempo de entrenamiento. |
| **Añadir dropout para reducir overfitting**  Modelo:  model.add(Embedding(vocab\_size, embedding\_size, input\_length=sequence\_length)) model.add(LSTM(30, **dropout=0.3**)) model.add(Dense(vocab\_size, activation='sigmoid'))  Epoch 251/300 loss: 1.2098  accuracy: 0.5962 -  val\_loss: 1.5169  val\_accuracy: 0.5233  Conclusiones  La red se siguen sobreajustando, aumenta el tiempo de entrenamiento. |
| **Reducir tamaño de vocabulario a la mitad**  Reducir las dimensiones de representación de las secuencias de códigos para que tengan menos dimensiones. Bajar el embedding\_size a lamitad del vocabulario.  Epoch 200/200  loss: 1.0741  accuracy: 0.6452  val\_loss: 1.6498  val\_accuracy: 0.5118  Conclusiones  No tiene un efecto significativo claro, pero parece que provoca más sobreajuste, el % de acierto sobre el conjunto de entrenamiento sube y baja sobre el de test. |
| **Añadir capa de normalización antes de la capa densa**  model.add(LSTM(30)) model.add(BatchNormalization()) model.add(Dense(vocab\_size, activation='sigmoid')) print(model.summary())  Epoch 150/150  loss: 1.1742 -  accuracy: 0.5996 -  val\_loss: 1.6011 -  val\_accuracy: 0.5338  Conclusiones  Mejora muy escasa en la función de pérdida |
| **Cambiar función de activación a RELU en capa densa**  Se produce el efecto contrario, las métricas de train/test van paralelas y la función de pérdida decrece muy lentamente. El modelo mejora significativamente.  model.add(Embedding(vocab\_size, embedding\_size, input\_length=sequence\_length)) model.add(LSTM(30, dropout=0.3)) model.add(BatchNormalization()) model.add(Dense(vocab\_size, activation='relu'))  Results file: training\_history\_6\_1594539140.00881.csv  Last 10 records avg:  epoch 263.500000  accuracy 0.483107  loss 1.782723  val\_accuracy 0.480925  val\_loss 1.790821  Hay que incrementar la capacidad del modelo y bajamos el dropout a 0.2 y aumentamos a 200 epocas |
| **Aumentar capacidad del modelo y número de épocas**  Aumentamos el número de unidades de la capa LSTM y el dropout  model.add(Embedding(vocab\_size, embedding\_size, input\_length=sequence\_length)) model.add(LSTM(**50, dropout=0.3**)) model.add(BatchNormalization()) model.add(Dense(vocab\_size, activation='relu'))  Last 10 records avg:  epoch 294.500000  accuracy 0.499515  loss 1.726744  val\_accuracy 0.484853  val\_loss 1.807326  No hay mejora significativa, volvemos a 30 unidades    Aumetnar o reducir el drop out tiene efectos negativos, reduciendo a 0.2 o aumentando a 0.4, la respuesta de las métricas es errática.  Con un dropout de 40%: |
| **Cambiar tamaño del batch size para ver cómo influye en el entrenamiento**  Al cambiar el tamaño del batch\_size está mejorando el entrenamiento, esto se tiene que deber a que las clases están muy desbalanceadas, probar con un dataset nuevo que equilibre al menos el número de clases de y.  Fijamos 128 y creamos nuevo dataset  model.add(LSTM(30, dropout=0.3)) model.add(BatchNormalization()) model.add(Dense(vocab\_size, activation='relu')) |
| **Cambiar función de activación en LSTM a RELU**  model.add(Embedding(vocab\_size, embedding\_size, input\_length=sequence\_length)) model.add(LSTM(30, dropout=0.3, activation=None)) model.add(BatchNormalization()) model.add(Activation("relu")) model.add(Dense(vocab\_size, activation='relu'))  Mejora pero se sigue quedando estancado en torno al 55% hay que mejorar la capacidad del modelos.    Last 10 records avg:  epoch 224.500000  accuracy 0.529968  loss 1.761819  val\_accuracy 0.525994  val\_loss 1.799849 |
| **Crear conjunto de datos balanceado**  Utilizando undersampler para clases mayoritarias para crear un conjunto balanceado respecto a la categoría.  epoch 300  accuracy 0.408273  loss 2.170180  val\_accuracy 0.414272  val\_loss 2.154978  epoch 300  accuracy 0.443459  loss 1.974530  val\_accuracy 0.453401  val\_loss 1.925942   |  |  | | --- | --- | | Distrución de clases en dataset original | Distribución de clases en dataset balanceado |   Tiene efecto negativo, se ha reducido el dataset a la mitad.    Probamos de nuevo aumentando el número de mínimo de muestras a 1000   |  |  | | --- | --- | |  |  | |
| **Aumentar el número de celdas internas**  Aumentar número de unidades a 40  **epoch 300**  accuracy 0.470519  loss 1.855217  val\_accuracy 0.478633  val\_loss 1.877754  dtype: float64    Con 50 unidades empieza a aparecer el sobreajuste  epoch 300  accuracy 0.493722  loss 1.787970  val\_accuracy 0.498817  val\_loss 1.951795  dtype: float64 |
| **Cambiar función de ajuste de categorical cross-entropy to f1-score**  El aprendizaje es más estable, pero se llega a una tasa en la que la red empieza a sobrentrenar. Antes de empezar con las optimizaciones, hay que intentar mejorar la capacidad del modelo.  epoch 100  f1 0.726099  loss 0.273565  val\_f1 0.686879  val\_loss 0.313024 |
| **Parametrización LSTM**   * Recurrent drop-out: 0.2 y mover dropout a capa inpendendiente antes del dense. → No influye   model = Sequential() model.add(Embedding(vocab\_size, embedding\_size,input\_length=sequence\_length)) model.add(LSTM(40, **recurrent\_dropout=0.2**)) model.add(Dropout(0.2)) model.add(BatchNormalization()) model.add(Dense(vocab\_size, activation='relu'))  epoch 100  f1 0.726099  loss 0.273565  val\_f1 0.686879  val\_loss 0.313024    \* Cambiar dropout antes de la normalización batch: sin efecto  epoch 26.500000  f1 0.694709  loss 0.304934  val\_f1 0.671864  val\_loss 0.327877  dtype: float64 |
| **Utilizar modelo con stacked LSTM**  model.add(Embedding(vocab\_size, embedding\_size, input\_length=sequence\_length)) model.add(LSTM(32, return\_sequences=True)) model.add(BatchNormalization()) model.add(LSTM(64)) # , recurrent\_dropout=0.2 model.add(BatchNormalization()) model.add(Dense(vocab\_size, activation='relu'))  Aumenta el score sobre el conjunto de entrenamiento pero no sobre el conjunto de test, el modelo se sobreajusta a los datos.   |  |  | | --- | --- | | LSTM one layer | LSTM two layers | |
| **Utilizar capa TimeDistributed**  No tiene sentido en este caso, cada elemento temporal de la secuencia es una de las dimensiones del embedding, no tiene sentido aplicar una red interca sobre cada columna por separado.  A TimeDistributed wrapper layer is used around the output layer so that one value per timestep can be predicted given the full sequence provided as input. This requires that the LSTM hidden layer returns a sequence of values (one per timestep) rather than a single value for the whole input sequence. |
| **Modificar embeddings**  Cambiar la dimensión de la capa de embeddings no tiene efecto, parace que son pocas combinaciones y pocas columnas para este tipo de técnica.  Haciendo un plot de los pesos de la capa de embeddings y proyectándolo en dos dimensiones con TSNE parecen equidistantes, por lo que lo se esan encontrando similitudes en**tre los cultivos, el lstm está recibiendo básicamente la misma entrada.**  Conclusiones: utilizamos dimensión 5. |
| **Reducción programada del learning rate**  \* Como inversa de exponencial: sin efecto  \* Drop en base a épocas mal resultado |
| **Uso de redes de convolución 1D**  Uso de filtros 1D para extraer patrones en las secuencias temporales.  2 redes de convolución con dropout y global average  20200718\_094307\_con1d\_1l\_emb5   |  |  | | --- | --- | |  | epoch 87.500000  f1 0.606053  loss 0.393834  val\_f1 0.607326  val\_loss 0.391150 | |
| **Red de convolución 1D con LSTM por encima**  20200718\_161056\_lstm\_2l\_64\_32\_16\_emb5   |  |  | | --- | --- | |  | epoch 87.500000  f1 0.672397  loss 0.327495  val\_f1 0.665538  val\_loss 0.332781  dtype: float64 | |
| **Red LSTM de tres capas**  689/689 - 49s - loss: 0.3248 - f1: 0.6751 - val\_loss: 0.4267 - val\_f1: 0.5719  Epoch 00038: early stopping  Loss on test data: [0.4267488121986389, 0.5719085335731506]  **20200718\_161056\_lstm\_2l\_64\_32\_16\_emb5**   |  |  | | --- | --- | |  | **epoch 92.500000**  f1 0.664051  loss 0.335830  val\_f1 0.657944  val\_loss 0.340288  dtype: float64 | |
| **Red 1D + LSTM utilizando redes de restos**   |  |  | | --- | --- | |  | epoch 39.500000  f1 0.705121  loss 0.294734  val\_f1 0.678709  val\_loss 0.319534 |  |  |  | | --- | --- | |  | epoch 49.500000  f1 0.715732  loss 0.284151  val\_f1 0.684259  val\_loss 0.313946  Aquí hay que añadir dropOut o normBatch, la red se está sobresajustando |   **Conclusión**  Las redes con 1D consiguen llegar a valores máximos mucho antes que solo LSTM. Los resultados son similares, no consiguen mejorar una red LSTM de dos niveles   |  |  | | --- | --- | |  | 0.7649 / 0.73432 | |  | 0.7866 - val\_loss: 0.2624 - val\_f1: 0.7372 | |  | 0.76, 0.730 | |  | 0.76, 0.735 | |  | La conv1D está traspuesta, está aprendiendo patrones sobre el R20 de los embeddings en lugar de la secuencia.  Con 64 unids en LSTM  loss: 0.1902 - f1: 0.8094 - val\_loss: 0.2732 - val\_f1: 0.7264  Con 12 unid.  loss: 0.1883 - f1: 0.8113 -  val\_loss: 0.2679 - val\_f1: 0.7318  Con 128 unid.  loss: 0.1806 - f1: 0.8213 - val\_loss: 0.2695 - val\_f1: 0.7329  **\* Quitando rama lstm**  **loss: 0.1823 - f1: 0.8174 - val\_loss: 0.2644 - val\_f1: 0.7356**  \* Añadiendo conv1d como feature extraction para la lstm  loss: 0.2180 - f1: 0.7842 - val\_loss: 0.2748 - val\_f1: 0.7278  \* Flatten de embeddings  loss: 0.2333 - f1: 0.7663 - val\_loss: 0.2660 - val\_f1: 0.7339 | | **Uso de flatten antes de LSTM**  La idea era aplanar la información de las secuencias de salida de LSTM para ver si tener las secuencias más largas mejoraba algo.  Tiene efecto negativo, el modelo tarda mucho más en entrenar y la función de pérdida salta de unos valores a otros de forma errática. |  | | **Bidirectional**  No mejora respecto a LSTM de una direccion  model.add(Embedding(vocab\_size, embedding\_size, input\_length=sequence\_length))  model.add(BatchNormalization())  model.add(Bidirectional(LSTM(units = layer\_size, recurrent\_dropout=0.4)))  model.add(BatchNormalization())  model.add(Dropout(0.3))  model.add(Dense(vocab\_size, "relu")) | loss: 0.1960 - f1: 0.8033 - val\_loss: 0.2704 - val\_f1: 0.7276 | | A Simple but Hard-to-Beat Baseline for Session-based Recommendations  <https://www.researchgate.net/profile/Alexandros_Karatzoglou/publication/327050395_A_Simple_but_Hard-to-Beat_Baseline_for_Session-based_Recommendations/links/5bf473a04585150b2bc4afd6/A-Simple-but-Hard-to-Beat-Baseline-for-Session-based-Recommendations.pdf> |  | | **Repetir secuencia con spatialDropout**  model.add(Embedding(vocab\_size, embedding\_size, input\_length=sequence\_length))  model.add(SpatialDropout1D(0.2)) ##This  model.add(LSTM(units = layer\_size, recurrent\_dropout=0.4))  model.add(RepeatVector(sequence\_length))  model.add(LSTM(units = layer\_size))  model.add(Dense(27, activation = "relu")) | loss: 0.2600 - f1: 0.7395 - val\_loss: 0.2841 - val\_f1: 0.7141 | |  |  | |  |  | |  |  | |  |  | |
|  |
| **Parametrización del optimizador**  **Gradient Clipping**  (lr=0.001, clipvalue=0.3),  No hay mejora  **Learning rate:** |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |



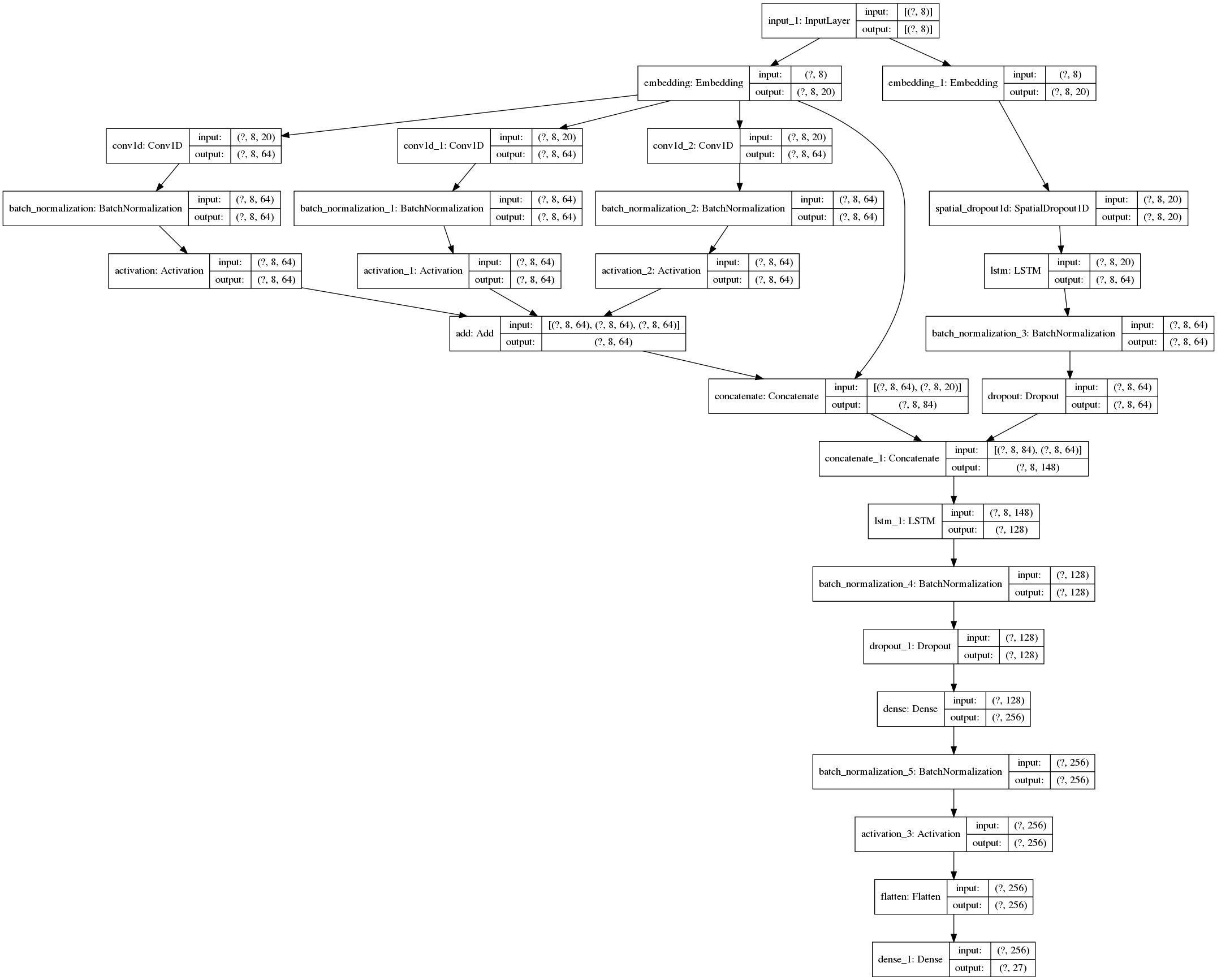
# LSTM + restos Conv

1. 687/687 - 34s - loss: 0.2020 - f1: 0.7977 - val\_loss: 0.2592 - val\_f1: **0.7387**
2. restos de kernels de diferentes tamaños add.



Añadiendo skip conection de los embeddings se reduce el rendimento:

loss: 0.1990 - f1: 0.8003 - val\_loss: 0.2629 - v**al\_f1: 0.7349**



Añadiendo skip conections:

Utilizando una estructura de inception empeora, dos cambios principales:

loss: 0.1914 - f1: 0.8079 - val\_loss: 0.2663 - val\_f1: 0.7317

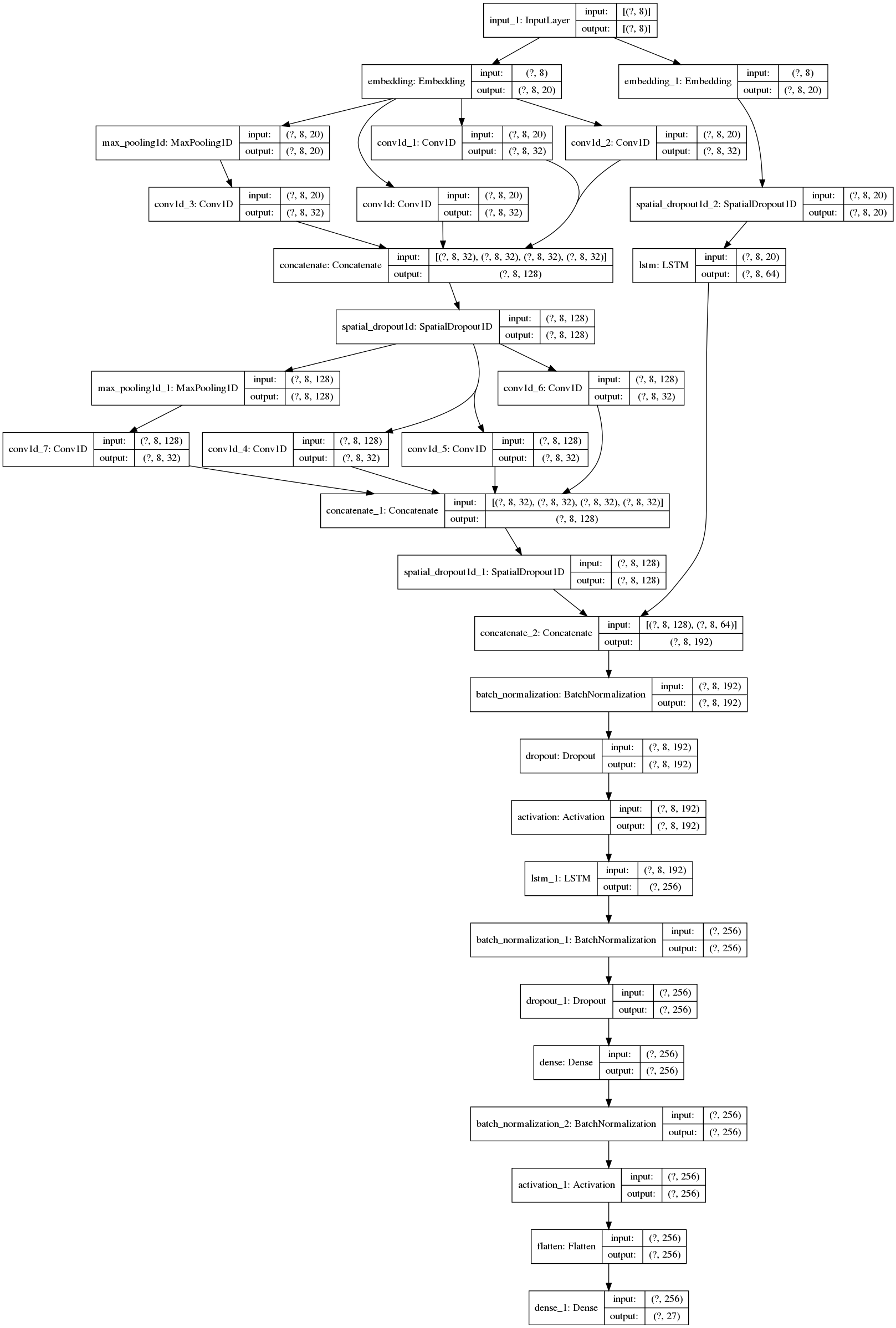
- Rama de Pooling

- Las capas se agregan con depth-Concatenate en lugar de hacer Add, esto hace que a la capa LSTM le lleguen 128 features en lugar de 32.

- He añadido dos módulos de inception en lugar de solo uno.

Añadiendo spatialDropout:

35s - loss: 0.2153 - f1: 0.7840 - val\_loss: 0.2592 - val\_f1: 0.7387



Reduciendo los embeddings a 12 y eliminando la rama de Pooling

Reduciendo los embeddings a 12 y eliminando la rama de Pooling y Sustituyendo concat por Add en los módulos de Inception

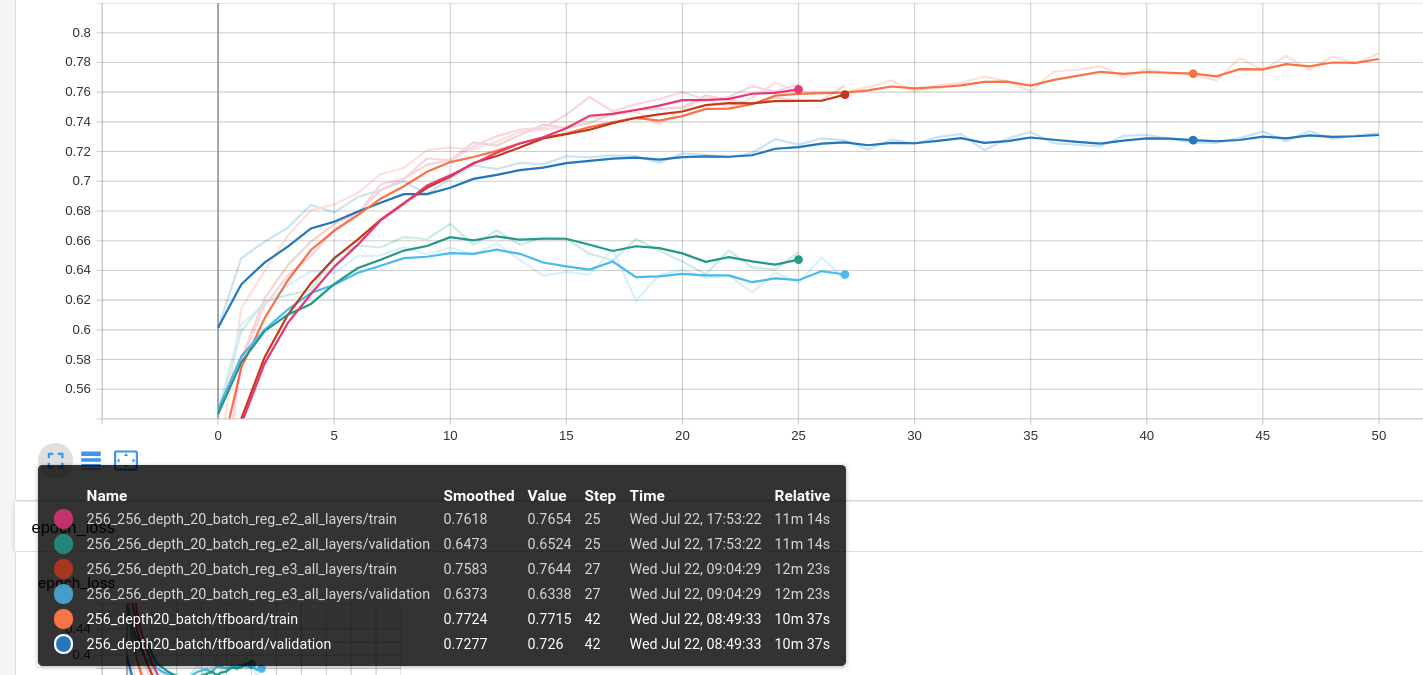
0.2300 - f1: 0.7696 - val\_loss: 0.2585 - val\_f1: 0.7395

Cambiando los embeddings a 20:

# Sin uso de embeddings

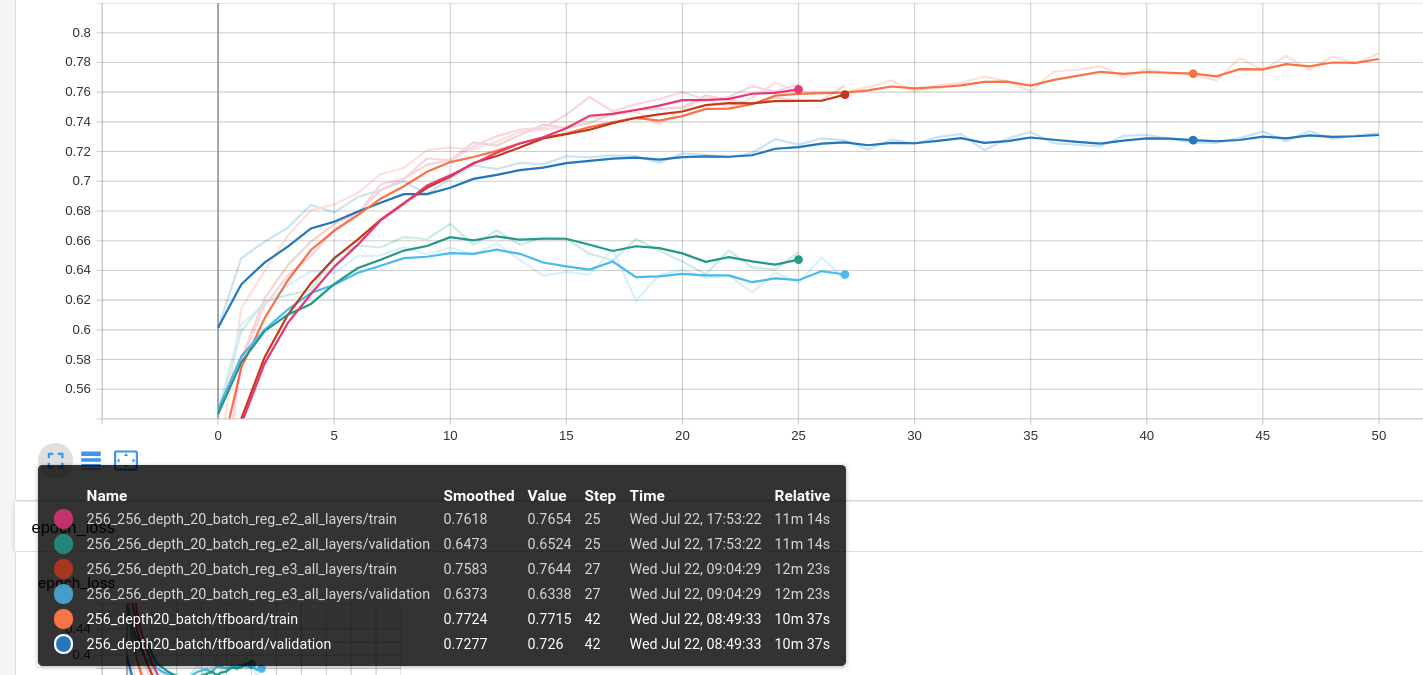
1. Utilizando una Conv1D directamente con los códigos de producto normalizados a 0-1. No lleba a 0.30 en f1\_val.
2. model.add(Reshape((8, 1)))
3. model.add(Conv1D(filters=64, kernel\_size=2, strides=1, activation='relu'))
4. model.add(SpatialDropout1D(0.3))
5. model.add(Conv1D(filters=64, kernel\_size=5, strides=1, activation='relu'))
6. # model.add(Conv1D(32, kernel\_size=3, activation='relu'))
7. # model.add(MaxPooling1D(4))
8. # # model.add(Dropout(0.5))
9. # model.add(GlobalMaxPooling1D())
10. # model.add(MaxPooling1D(pool\_size=3))
11. model.add(Flatten())
12. model.add(Dense(256, activation='relu'))
13. # model.add(BatchNormalization())
14. model.add(Dense(vocab\_size, activation='relu'))
15. return model

# Efecto del número de capas

1. Añadiendo una segunda capa no se mejora, ni se mejora el entrenamiento ni en validación. Es mejor dejar una única capa de 256 filtros que dos.
2. Hay que buscar explicación a esto, puede que las características que devuelve la capa no sean significativas.

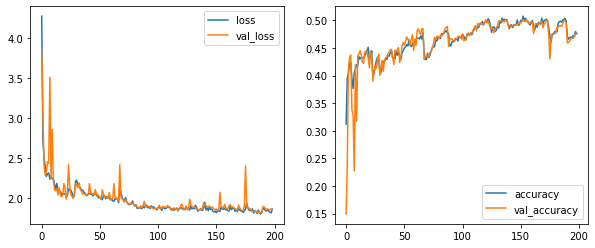
# Efecto función de activación final

|  |  |
| --- | --- |
| model = Sequential()  model.add(Embedding(vocab\_size, embedding\_size, input\_length=sequence\_length))  model.add(BatchNormalization())  model.add(LSTM(units = layer\_size, recurrent\_dropout=0.4))  model.add(BatchNormalization())  model.add(Dropout(0.3))  model.add(Dense(vocab\_size, "relu"))  f1: 0.8022 - val\_f1: 0.7213 | Con softmax  f1: 0.3679 - val\_f1: 0.3297  Con sigmoid  f1: 0.3605 - val\_f1: 0.3508  con Arctan  con leaky relyu |
|  |  |

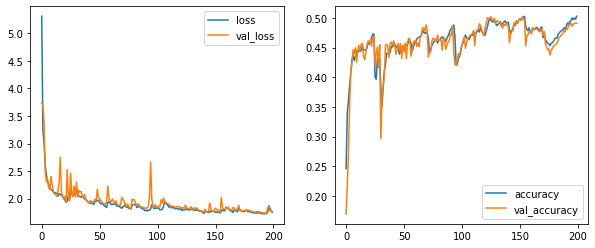


# Efecto del batch\_size

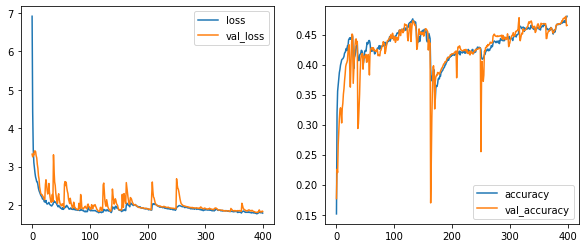
64



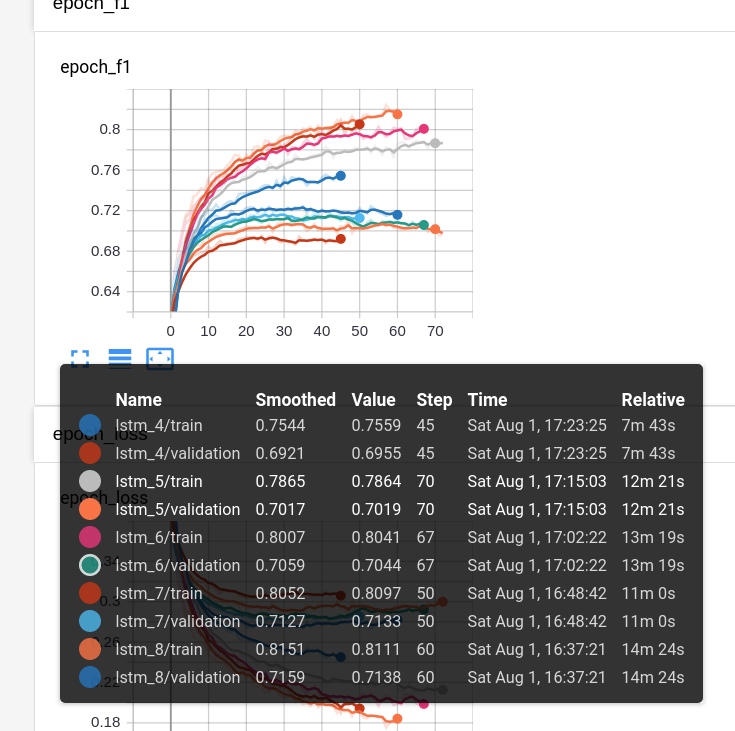
128



512



# Efecto de la longitud de la secuencia

1. El rendimiento empeora, tanto en entrenamiento como en validación. Hay que tener la secuencia de mayor longitud posible.
2. 

# Efecto de la dimensión de embeddings

1) Prueba sin embeddings, utilizando como entrada directa los códigos de producto.

La profundidad del embedding mejora el entrenamiento.

Tomando una red de 128 unidades, entrenamos para profundidades de 5,12,20

depth 5 (0.7/0.71) > depth 12 (0.73/0.72) > **depth 20 (0.74/0.73)** > depth 25 (0.75, 0.72)

1. 

\* Probar con profundidad 25

|  |  |
| --- | --- |
| Embeddings size 5 – 40 epocas | Embeddings size 10 -40 epocas |
|  |  |
| 100 epocas | 100 epocas |
|  |  |
|  |  |

# Base 1 capa

256 embeddings de 20 con batch y sin regularizació**n. 0.78,0.73.**

dropout=0.3, recurrent\_dropout=0.4 clipping: 0.3, lr:0.001

Sin callback para lr dinámica

**trail f1: 0.8004 - val\_loss: 0.2635 - val\_f1: 0.7363**

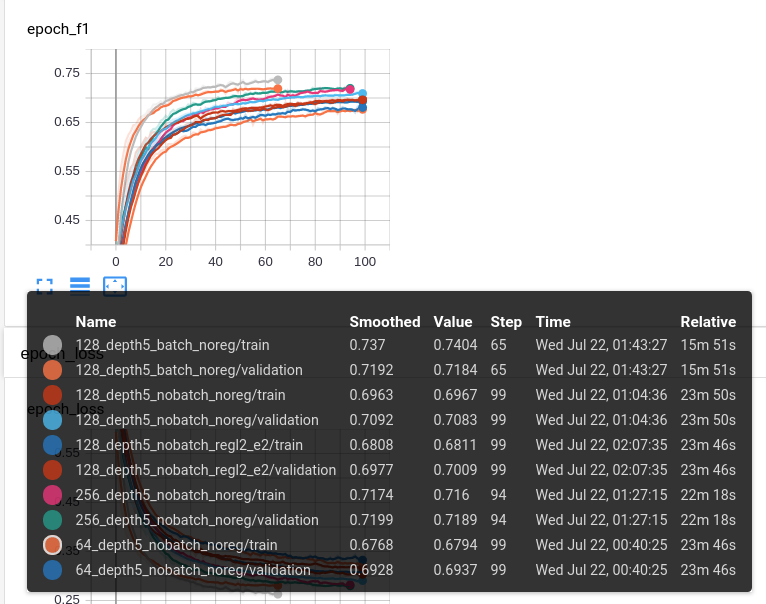
Optimización bayesiana:

{'target': 0.7342096567153931, 'params': {'dropout': 0.37460037107263344, 'opt\_clipvalue': 0.44211898516408055, 'opt\_lr': 0.0009835405503902273, 'recurrent\_dropout': 0.40005772597798706}}

# Efecto en el número de celdas, batch y reg en una capa

1. Con una única capa, añadimos celdas para ver cuál funciona mejor.
2. Una capa de 128 unidades que tomamos como base y sin regularización.
3. Conclusiones:

* El aumento en el número de neuronas mejora 256 (0.72/0.72) > 128 (0.7/0.71) > 64 (0.68/0.68)
* La capa batch reduce el tiempo de entrenamiento 128 batch (0.74/0.72) > 128 no batch (0.7/0.71)
* La regularización recurrente tiene un efecto negativo: 128 no reg (0.7/0.71) > 128 reg (0.68, 0.69)



Más experimentos:

\* Capa con 512 y 256 con batch: se mejora respecto a 128 batch pero no demasiado: (0.74/0.72) < 256batch (0.74/0.73) < 256batch (0.76/0.73).

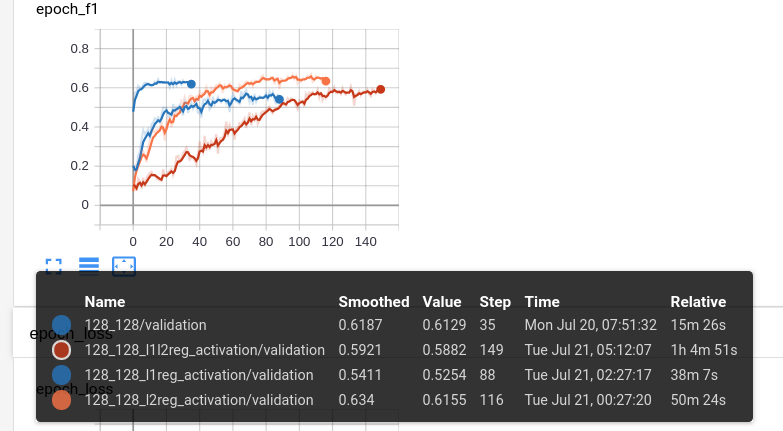
# Efecto del gradient clipping

1. <https://danijar.com/tips-for-training-recurrent-neural-networks/>

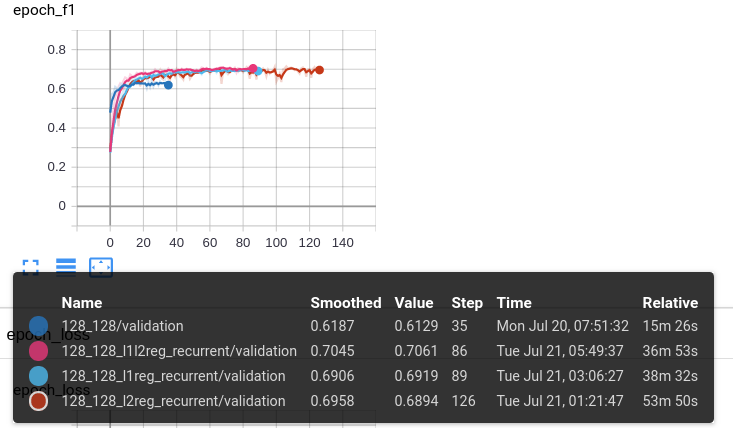
# 

# Efecto de la regularización en dos capas

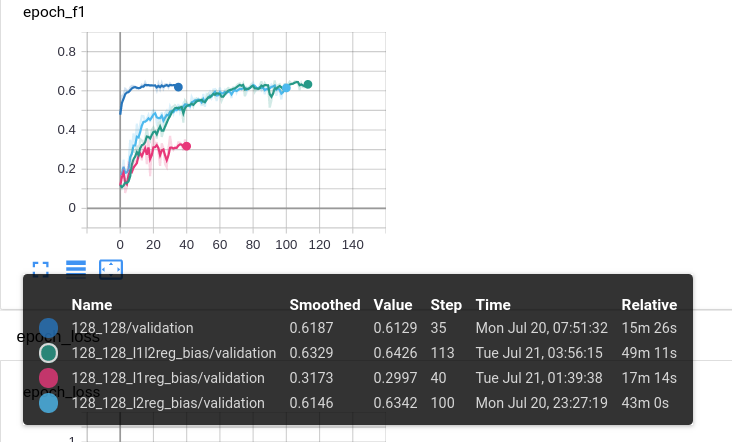
1. Las capas LSTM permiten cuatro tipos de regularizaciones, kernel, bias, activación y recurrencia, y para cada tipo se pueden aplicar tres normalizaciones, l1, l2 y elastica (combinada).
2. En todos los casos se ha aplicado regularización 0.001
3. La regularización **de kernel**, en los tres casos lleva a una pérdida de rendimiento drástica:
4. 
5. Regularizaciones de **activación**

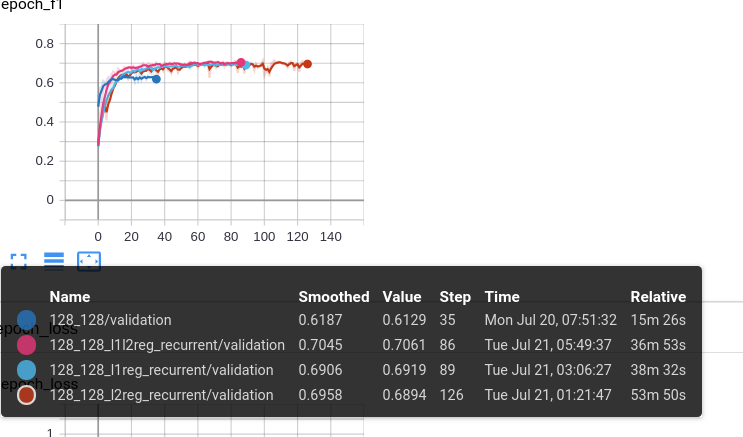


Regularizaciones **recurrentes**

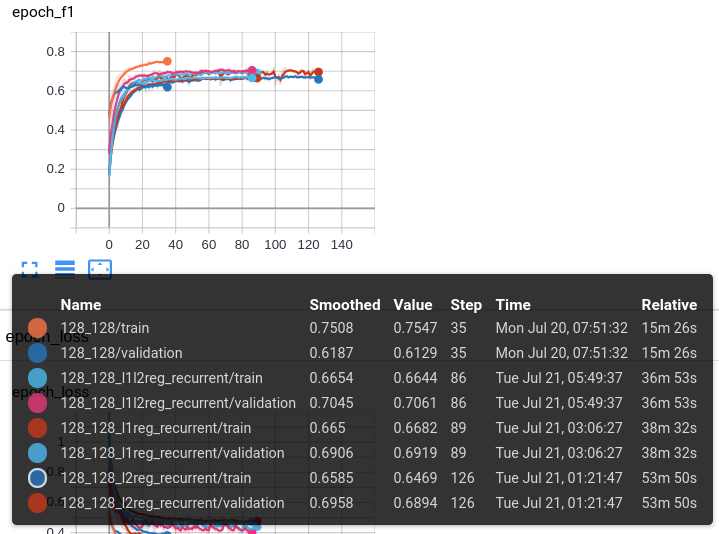


**Regularización de bias**

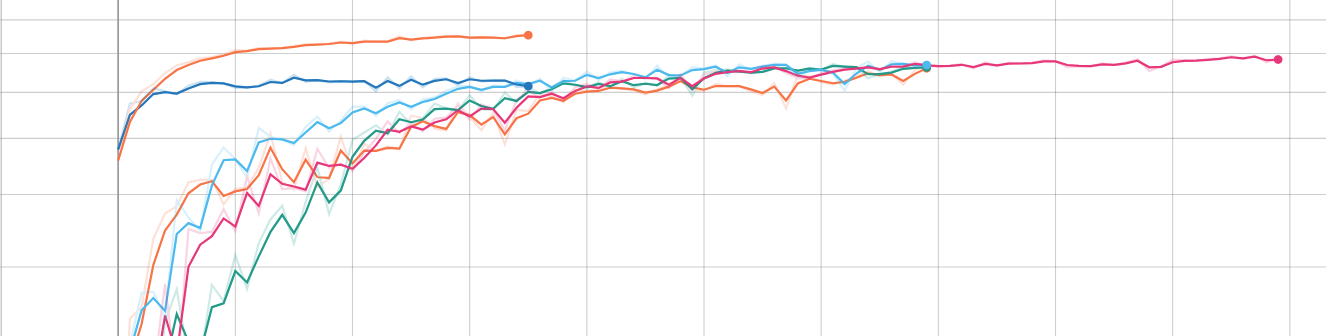




La regularización que mejor funciona es la recurrente con norma l2 o elastica (l1/l2), pero hay que ajustar el valor porque está provocando el efecto contrario, underfitting, la métrica de validación está por encima de la de entrenamiento. Hay que recudir el valor de regularización.



Utilizar r**egularización de recurrencia con valor 0.01 como máximo**, pero los tiempos de entrenamiento se alargan, pero la validación mejora.

Parámetro de regularización

Conclusiones:

* Aplicar regularización de kernel con valor máximo de 0.01, a partir de ese valor empieza a ocurrir el caso contrario, underfitting
* La regularización hay que aplicarla a todas las capas. Solo capa final (0.62,0.66) < Todas las capas (0.66,0.68)

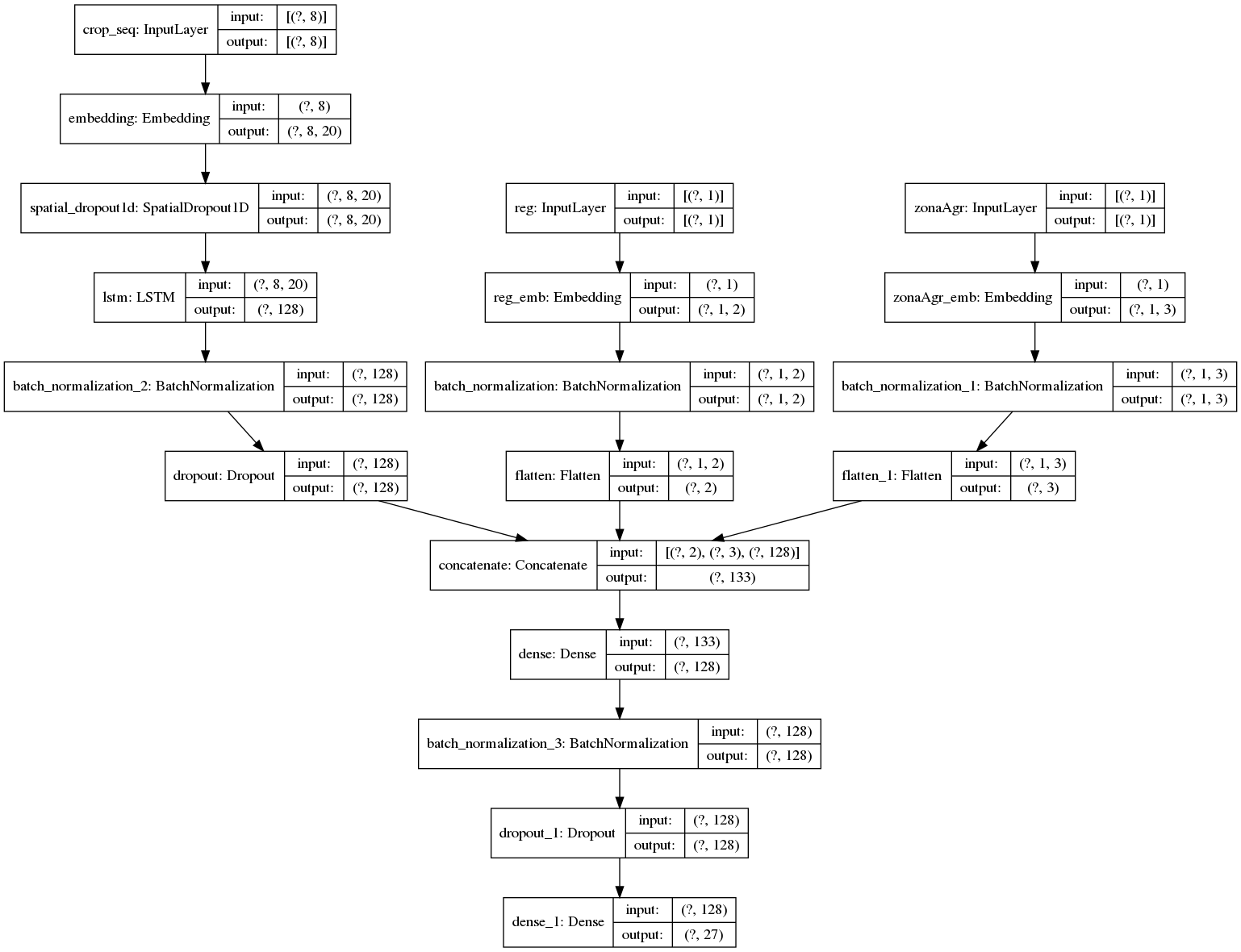
Probar regularización 0.1 → Influye negativamente.

# Normalización de capa vs Normalización batch después de LSTM y Dropout

layer\_size=256

|  |  |
| --- | --- |
| **Sin normalización**  model = Sequential()  model.add(Embedding(vocab\_size, embedding\_size, input\_length=sequence\_length))  model.add(LSTM(units = layer\_size, recurrent\_dropout=0.4))  model.add(Dropout(0.3))  model.add(Dense(vocab\_size, "relu"))  f1: 0.7661 - val\_f1: 0.7036 | Layer norm después de embedding  model = Sequential()  model.add(Embedding(vocab\_size, embedding\_size, input\_length=sequence\_length))  model.add(Normalization())  model.add(LSTM(units = layer\_size, recurrent\_dropout=0.4))  model.add(Dropout(0.3))  model.add(Dense(vocab\_size, "relu"))  f1: 0.7509 - val\_f1: 0.7053 |
| model = Sequential()  model.add(Embedding(vocab\_size, embedding\_size, input\_length=sequence\_length))  model.add(LSTM(units = layer\_size, recurrent\_dropout=0.4))  model.add(Dropout(0.3))  model.add(BatchNormalization())  model.add(Dense(vocab\_size, "relu"))  f1: 0.7801 - val\_f1: 0.7172 | model = Sequential()  model.add(Embedding(vocab\_size, embedding\_size, input\_length=sequence\_length))  model.add(BatchNormalization())  model.add(LSTM(units = layer\_size, recurrent\_dropout=0.4))  model.add(Normalization())  model.add(Dropout(0.3))  model.add(Dense(vocab\_size, "relu"))  f1: 0.7831 - val\_f1: 0.7098 |
| **model = Sequential()**  **model.add(Embedding(vocab\_size, embedding\_size, input\_length=sequence\_length))**  **model.add(BatchNormalization())**  **model.add(LSTM(units = layer\_size, recurrent\_dropout=0.4))**  **model.add(BatchNormalization())**  **model.add(Dropout(0.3))**  **model.add(Dense(vocab\_size, "relu"))**  **f1: 0.8022 - val\_f1: 0.7213**  El dropout debe ir después de la normalización (y la activación) | model = Sequential()  model.add(Embedding(vocab\_size, embedding\_size, input\_length=sequence\_length))  model.add(BatchNormalization())  model.add(LSTM(units = layer\_size, recurrent\_dropout=0.4))  model.add(Dropout(0.3))  model.add(BatchNormalization())  model.add(Dense(vocab\_size, "relu"))  f1: 0.7849 - val\_f1: 0.7196 |
| model = Sequential()  model.add(Embedding(vocab\_size, embedding\_size, input\_length=sequence\_length))  model.add(Dropout(0.2))  model.add(LSTM(units = layer\_size, recurrent\_dropout=0.4))  model.add(BatchNormalization())  model.add(Dropout(0.3))  model.add(Dense(vocab\_size, "relu"))  f1: 0.7806 - val\_loss: 0.2690 - val\_f1: **0.7289** | model.add(Embedding(vocab\_size, embedding\_size, input\_length=sequence\_length))  model.add(SpatialDropout1D(0.4))  model.add(LSTM(units = layer\_size, recurrent\_dropout=0.4))  model.add(BatchNormalization())  model.add(Dropout(0.3))  model.add(Dense(vocab\_size, "relu"))  - f1: 0.7738 - val\_loss: 0.2692 - val\_f1**: 0.7289** |

# Características adicionales

1. **Como embeddings**
2. Añadimos la zona agraria y el el tipo de explotación (R/S) como embeddings
3. Con una básica
4. loss: 0.2504 - f1: 0.7488 - val\_loss: 0.2988 - val\_f1: 0.6987
5. Añadiendo spatialDropout
6. loss: 0.2739 - f1: 0.7255 - val\_loss: 0.3043 - val\_f1: 0.6938
7. 
8. **Como DenseFeatures**

# Redes Conv1D

Mejor dos capas que una.

La capa dense intermedia no parece tener efecto, lo mismo que el flatten. Probar a mover a 128.

No se pasa del 0.7 ni en validación ni en entrenamiento.

**Conv1d32**

**Conv1d64**

Epoch 00082: val\_f1 did not improve from 0.70829

679/679 - 1s - loss: 0.2457 - f1: 0.7538 - val\_loss: 0.2984 - val\_f1: 0.7014

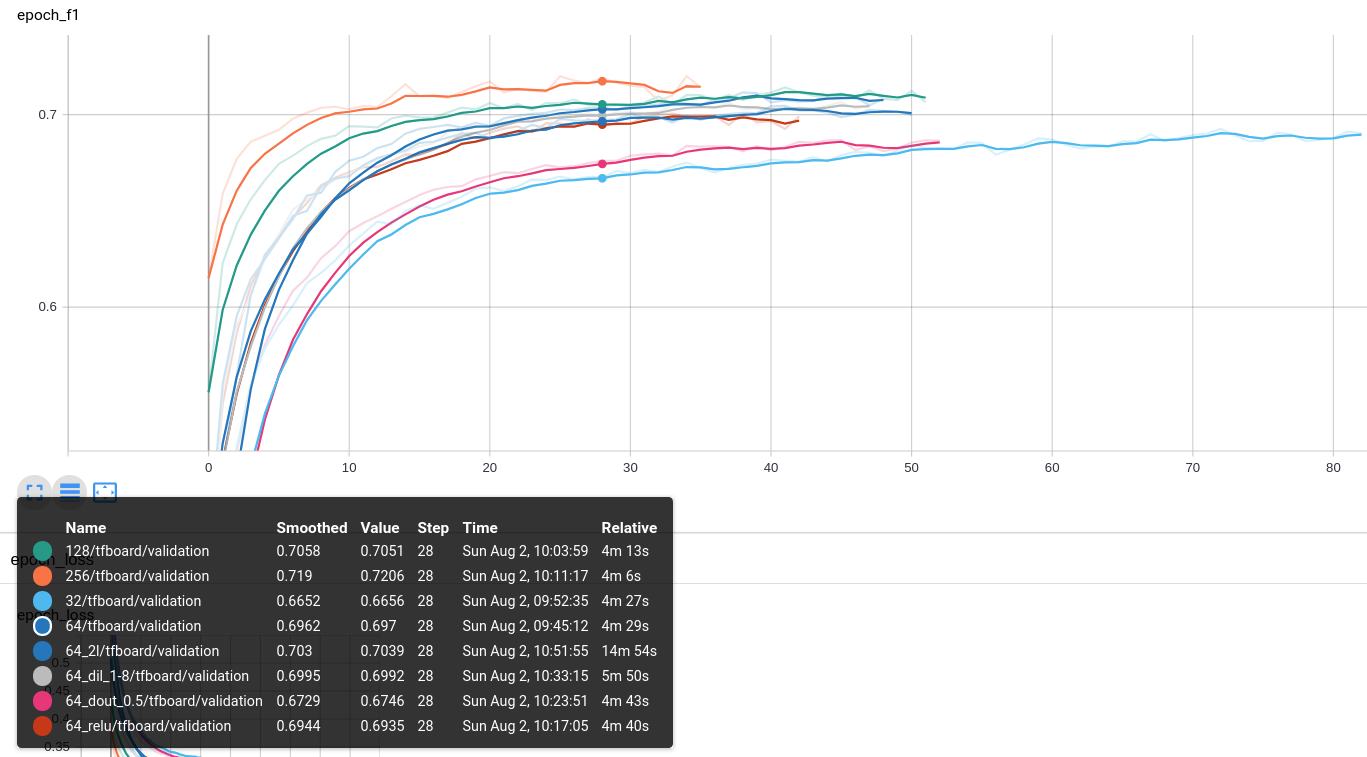
loss: 0.2737 - f1: 0.7257 - val\_loss: 0.2948 - val\_f1: 0.7050

**Conv1d32Conv64**

No tenemos suficiente longitud de secuencia para utilizar un maxpooling entre capas.

# Papers

## Temporal Convolutional Networks (TCN)

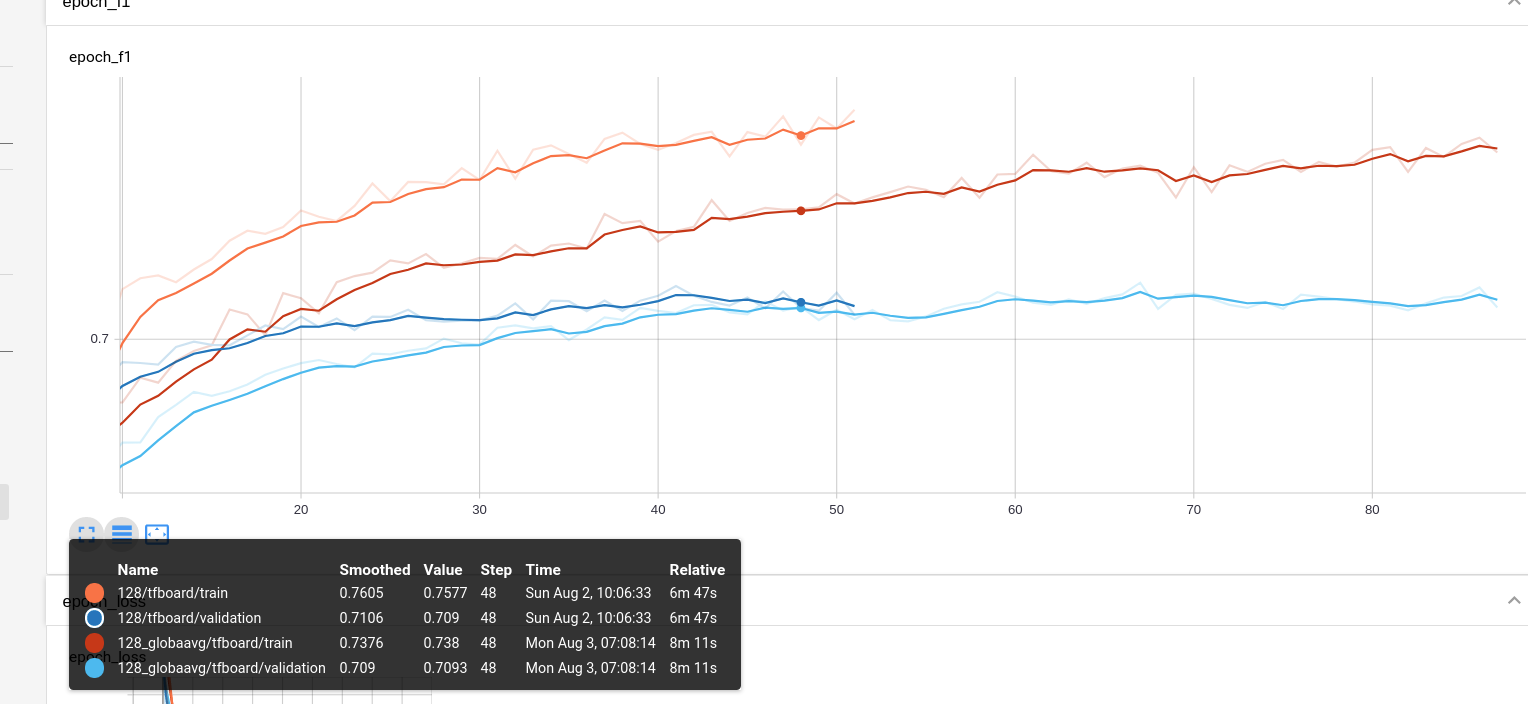
* 1. 
  + Aumentar el número de celdas mejora: 256 > 128 > 64 > 32, y el overfitting no es significativamente mayor. Con 512 empieza a empeorar
  + En todos los casos hay overfitting, pero aumentar el dropout no mejora
  + Utilizar activación relu da peores resultados.
  + Aumentar la dilatación a 8 mejora mínimamente, aunque no debería ser así. Mejora con 64 filtros, al aumentar a 512 no hay diferencia.
  + Aumentar un segundo nivel de restos mejora train y test (stack3 > stack2 > stack 1)
  + Añadir una segunda capa mejora train y test.
  + Añadir batchNorm entre dos capas empeora en train y test.
  + Dropout a 0.5 (está en 0.3) empeora, el rendimiento baja por debajo de 0.7.
  + Efecto del embeddings: 5,13,20 con 64:
    - 5 y 13 prácticamente igual, con 20 un poco mejor tanto en 256 como en 64.
    - En el conjunto de entrenamiento hay mayor diferencia que en el de validación.



* + Skip connections: no mejora, en pruebas de 256 y 256 con skip connections tiene mejor resultado sin añadir la opción. Hay que probar en capas más profundas que es donde puede tener más ventajas.
* Con 128 filtros,
* Stacks vs layers: ni stacks ni layers mejoran una única capa de 256 filtros sobre el conjunto de validación. En entrenamiento el mejo caso lo da 256 filtros con 2 stacks pero sobreajusta.
  + Con 256 filtros, aumentar la profundidad no mejora, ni en capas ni en número de restos apilados mejora el rendimiento. Buscar equilibrio entre número de capas y número de restos.
    - Pruebas hechas: Dos layers de 3 stacks de 256, tres capas de dos pilas de 256, dos capas de dos pilas de 256
  + Con 128 filtros: el mejor caso es una capa con dos stacks. Añadir stacks (probado 1-5) o añadir capas (probado hasta 3 capas) empeora tanto entranamiento como en datos de validación.

Mejor caso: 1 capa de 1 pila de 256 filtros: **0.722**

* GlobalAvgPooling1D mejora mínimamente de 0.7154 a 0.7163 en 256. En 128 los valores son similares, un poco pero el globa avg. El overfitting es menos marcado en ambos casos.



* Eliminando la capa batch y haciendo de nuevo pruebas con 128/256, la que mejor se comporta es 128-stacks3 0.729 y un poco peor 256-stacks2 0.728 y 256-stacks1. Probarmos añadiendo capa batch después de embeddings y regularización con 128-stacks3 y 256-stacks2

Segunda iteración:

* Aumentar número de filtros a 512
* Probar normalización entre capas.

Tercera iteración, make it deeper

* Dos layers de 3 stacks de 256
* tres capas de dos pilas de 256
* Dos capas de dos pilas de 256

Cuarta iteración

* Buscar equilibrio deep vs wide: utilizar 128 kernels por capa y probar stack vs layer
* Añadir normalización batch después del embedding
* Eliminar normalización batch después de TCN, ya está incluida en la capa.
* Añadir globalAvgPooling al final antes de dense (la última tcn debe devolver la secuencia)

Quita iteración: regularización

* Dropout
* BatchNorm después de embedding
* LayerNorm en TCN
* Weight constrains:
  + Activity regularization
  + Kernel regularizer
  + Bias regularizer
* Quitar clipvalue al optimizador
* Conv + BN + GlobalAvgPooling1D (returnSeq = True antes) + Activation + Dense
* Activation = elu
* Activatio
* <https://keras.io/api/layers/regularizers/>
  1. <https://www.kaggle.com/christofhenkel/temporal-cnn>
  2. <https://arxiv.org/abs/1608.08242>

## LSTM-FCM

1. <https://towardsdatascience.com/lstm-fcn-for-cardiology-22af6bbfc27b>

## InceptionTime

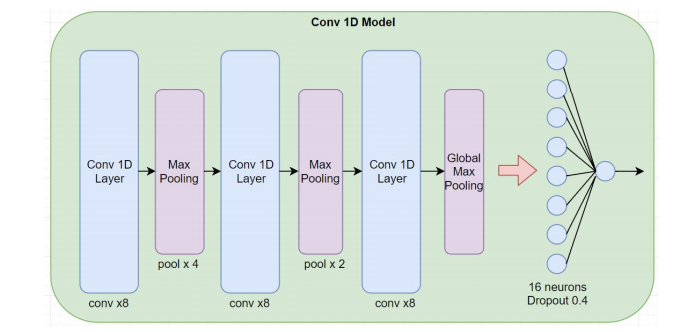
<https://towardsdatascience.com/deep-learning-for-time-series-classification-inceptiontime-245703f422db>

## LSTM and 1-D Convolutional Neural Networks for Predictive Monitoring of the Anaerobic Digestion Process

|  |  |
| --- | --- |
|  |  |

1. [**https://sci-hub.tw/10.1007/978-3-030-30493-5**](https://sci-hub.tw/10.1007/978-3-030-30493-5)

## LSTM and Convolution Networks exploration for Parkinson’s Diagnosis

1. Lstm, conv y conv → stack. El que mejor funciona CONV1D
2. 
3. <https://ieeexplore.ieee.org/abstract/document/8809160>

## Automated diagnosis of arrhythmia using combination of CNN and LSTM techniques with variable length heart beats

<https://www.sciencedirect.com/science/article/abs/pii/S0010482518301446>

## Deep learning based multi-temporal crop classification

1. Conv1d y lstm para series temporales de índices de vegetación.
2. <https://sci-hub.tw/https://doi.org/10.1016/j.rse.2018.11.032>
   1. <https://towardsdatascience.com/lstm-fcn-for-cardiology-22af6bbfc27b>

## Convolutional Recurrent Neural Networks

## Machine-learned prediction of annual crop planting in the U.S. Corn Belt based on historical crop planting maps

**Land Use Land Cover  (LULC)**

<https://sci-hub.tw/https://doi.org/10.1016/j.compag.2019.104989>

The **Cellular Automata-Markov,** which is an effective model to handle the continuous process, has been widely adopted for the LULC change prediction. The basic idea of these works is predicting the future LULC change from the historical Landsat TM time series.

* Corner, R.J., Dewan, A.M., Chakma, S., 2014. Monitoring and Prediction of Land-Use and Land-Cover (LULC) Change. In: Dewan, A., Corner, R. (Eds.), Dhaka Megacity: Geospatial Perspectives on Urbanisation, Environment and Health, Springer Geography. Springer N
* Halmy, M.W.A., Gessler, P.E., Hicke, J.A., Salem, B.B., 2015. Land use/land cover change detection and prediction in the north-western coastal desert of Egypt using Markov-CA. Appl. Geogr. 63, 101–112. <https://doi.org/10.1016/j.apgeog.2015.06.015>
* Singh, S.K., Mustak, Sk, Srivastava, P.K., Szabó, S., Islam, T., 2015. Predicting spatial and decadal LULC changes through cellular automata Markov chain models using earth observation datasets and geo-information. Environ. Process. 2, 61–78. <https://doi.org/10.1007/s40710-015-0062-x>.
* Mondal, Md.S., Sharma, N., Garg, P.K., Kappas, M., 2016. Statistical independence test and validation of CA Markov land use land cover (LULC) prediction results. Egypt. J. Remote Sens. Space Sci. 19, 259–272. <https://doi.org/10.1016/j.ejrs.2016.08.001>.
* Xu, X., Du, Z., Zhang, H., 2016. Integrating the system dynamic and cellular automata models to predict land use and land cover change. Int. J. Appl. Earth Obs. Geoinf. 52, 568–579. <https://doi.org/10.1016/j.jag.2016.07.022>.

Markov logic networks-based model for early crop mapping was proposed, which can predict the crop type from the historical crop rotation at the accuracy of 60% before the growing season

* Osman, J., Inglada, J., Dejoux, J.-F., 2015. Assessment of a Markov logic model of crop rotations for early crop mapping. Comput. Electron. Agric. 113, 234–243. https://doi.org/10.1016/j.compag.2015.02.015

Data and methods are two keys to address the proposed question.

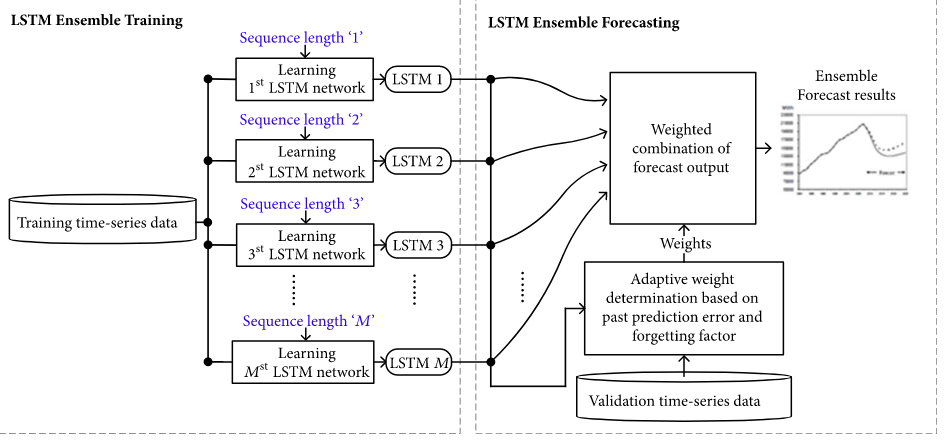
The Cropland Data Layer (CDL).

Agricultural Statistics Districts (ASDs)

El paper no muestra la arquitectura exacta seguida, ni número de capas, se limita a decir que utilizaciones

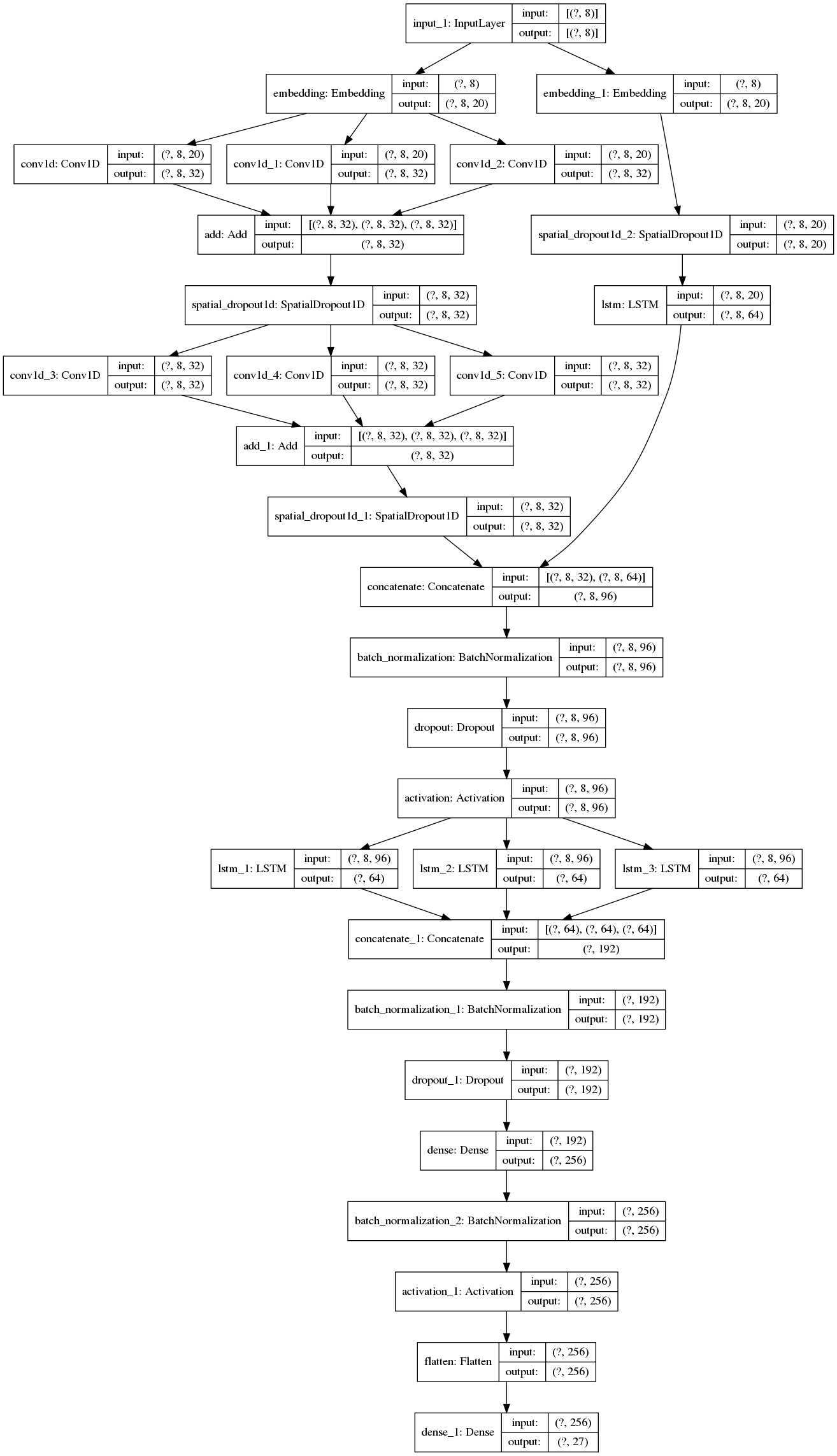
|  |  |
| --- | --- |
| Utilizan one-hot-enconding para las características    Tres capas 512,256, 128, flatten, añadiendo dropout 0.7169  Utilizando one-hot encoding para los 8 años como proponen en el paper:  0.72899  Añadiendo características de tipo explota y zona agro:  0.74489  Aumentando capacidad a 1024/512/256  0.74920  Aumentando capacidad a 1024/512/256/256  No mejora y se produce sobre ajuste  0.8258 - val\_loss: 0.2575 - val\_f1: 0.7405  Comprobar si realmente detecta patrones, entrenamos con secuencia t-n,t e intentamos predecir en t+1. |  |

# LSTM emsembles



<https://www.hindawi.com/journals/mpe/2018/2470171/>

loss: 0.2262 - f1: 0.7734 - val\_loss: 0.2646 - val\_f1: 0.7332



Take aways:

* Dimension shuffling permite poder entrenar crear una rama de convolución.
* El stacking de LSTM y Conv no suele funcionar, ni por arriba ni por abajo es mejor crear dos ramas separadas, con embeddings distintos
* Aunque la secuencia es corta, al utilizar un número de dimensiones muy alto (20) el rendimiento de la red mejora al aumentar el número de filtros, tanto en LSTM como en arquitecturas derivadas de filtros de convolución.
* A