Escuela Técnica Superior de Ingenieros de Caminos, Canales y Puertos.
UNIVERSIDAD DE CANTABRIA

APLICACIÓN DE REDES NEURONALES CONVOLUCIONALES AL RECONOCIMIENTO DE SEÑALES DE TRÁFICO.

TRABAJO FINAL DE MASTER

Trabajo realizado por:

Isidre Cañellas Colom

Dirigido:

Miguel Cuartas Hernández

Amaya Lobo García de Cortázar

Titulación:

Máster Universitario en Ingeniería de Caminos, Canales y Puertos

Santander, septiembre del 2018

Índice

Resumen

PROYECTO:	**Aplicación de redes neuronales convolucionales al reconocimiento de imágenes de tráfico.**
AUTOR:	**Isidre Cañellas Colom**
DIRECTORES:	Miguel Cuartas Hernández
	Amaya Lobo García de Cortázar
PALABRAS CLAVE:	*Machine Learning, Python, Neural Nets, Convolutional Nets, Traffic Sign Recognition, Big Data*

1. Introducción

Las Redes Neuronales (NN: Neural Networks) son modelos matemáticos inspirados en el comportamiento biológico de las neuronas y en la estructura del cerebro fueron propuestas por primera vez en 1943 por McCulloch y Pitts [1]. Son utilizadas para resolver un amplio rango de problemas de clasificación. Estas redes son muy tolerantes a errores de medida o ruido en el conjunto de datos, también son robustas al fallo de pocas neuronas.

Las Redes Neuronales Convolucionales (RNC) son una arquitectura especial de Red Neuronal propuesta por Yan LeCun en 1988 [2]. Una de las aplicaciones más populares para este tipo de arquitectura es la clasificación de imágenes.

La aplicación de Redes Neuronales al reconocimiento de señales de tráfico ha supuesto un punto de inflexión en los resultados obtenidos y en la cantidad bibliografía publicada [3]. La mayoría de los estudios publicados, se centran en diseños de Redes Neuronales Convolucionales [4], [5], [6], [7], [8].

El objetivo de este trabajo es describir e implementar modelos de Redes Neuronales sobre un conjunto de datos reales. Se han desarrollado cuatro estructuras de Redes Neuronales con dificultad y complejidad creciente y se ha aplicado al reconocimiento de señales de tráfico, para ello, se han utilizado dos conjuntos de datos de señales de tráfico europeas: German Traffic Sign Recognition Benchmark (GTSRB) [9] y *Belgium Traffic Sign Dataset (BTSD)* [10].

2. Base teórica

El elemento básico de computación de las Redes Neuronales se llama habitualmente nodo o unidad y está asociado a un peso *"w"* que se va modificando durante el proceso de aprendizaje. Cada nodo recibe un input y devuelve el producto de este por su peso asociado.

La estrategia seguida para el análisis de los datos, consta de cuatro modelos de predicción con estructuras diferentes, de dificultad y complejidad creciente. La programación de los modelos se realizó en base al curso de *Deep Learning* de Andrew NG [11] y al artículo de Waleed Abdulla [12]

Para el primer modelo, se implementó una Red Neuronal muy simple con una capa oculta con número de nodos igual al número de categorías. El segundo modelo implementado, tiene la misma estructura que el primer modelo e incluyen mejoras en la función de activación (*ReLU*) y en el algoritmo de optimización (*Minibatches)*. Para el tercer modelo, se construyó una Red Neuronal más compleja con tres capas ocultas de tamaño decreciente.

El cuarto modelo implementado constituye una Red Neuronal Convolucional de cuatro capas. Las tres primeras capas son convolucionales y contienen 8, 12 y 13 filtros de 5x5 respectivamente junto con una capa tipo *Maxpool* de 2x2; generando 7800 parámetros a optimizar. La última capa es completamente conectada con 43 nodos y una función de clasificación tipo *Softmax.*

3. German Traffic Sign Recognition Benchmark (GTSRB)

El *GTSRB* contiene 43 clases y 50.000 imágenes repartidas en dos subconjuntos de datos para entrenamiento y validación, el subconjunto de entrenamiento contiene 39.209 imágenes y el subconjunto de validación contiene 12.630 imágenes

El análisis previo del conjunto de datos observó una dispersión en la distribución del número de imágenes por categoría muy elevada y gran variedad de apariencias visuales en las imágenes: condiciones lumínicas, rotación, climatología y escalamiento.

Para el análisis de los resultados, se estudió la precisión de los modelos sobre los subconjuntos de entrenamiento y validación. Con el objetivo de estudiar la variación de los resultados, se realizaron 5 pruebas independientes sobre el mismo modelo. En un análisis más profundo, se estudió la matriz de pesos optimizada “W” y la influencia del número de imágenes por categoría sobre los resultados.

Se observó un **aumento de la precisión** en el subconjunto de entrenamiento junto con la complejidad de los modelos. La varianza y la media de las diferencias de precisión entre el subconjunto de entrenamiento y validación también disminuyeron de forma general.

Finalmente, el **Modelo 4** (capas Convolucionales) obtuvo un 90,8% de precisión sobre el subconjunto de validación, una varianza entre pruebas de 0,63 y una media de las diferencias entre la precisión de los subconjuntos de 10,37%. De estos resultados, se obtiene un margen de error evitable del 8-9 %, lo que se puede considerar muy satisfactorio. La media de las **diferencias de precisión entre subconjuntos de 10,37%** junto con una **precisión del 99%** sobre el subconjunto de entrenamiento indicaron un **ligero sobreajuste** del Modelo 4 al subconjunto de prueba.

4. Belgium Traffic Sign Dataset

El BTSD contiene 63 clases y dos subconjuntos de entrenamiento y validación con 4.575 y 2.520 imágenes respectivamente.

El análisis de la matriz de pesos “W” y su relación con el número de imágenes por categoría y la precisión por categoría demostró una **relación directa** entre el **número de imágenes por categoría** y la **precisión del modelo por categoría**. Las imágenes con menor representación sobre el subconjunto de entrenamiento obtuvieron una menor precisión.

Las **tendencias** observadas sobre el conjunto de datos del GTSRB se **confirmaron** para el BTSD. Se observaron **mejores resultados para el BTSD** aunque el número de imágenes de este

conjunto de datos es mucho menor que para el GTSRB y su distribución de imágenes por categorías es más dispersa.

5. Conclusiones

Se observó en los dos conjuntos de datos utilizados (BTSD y GTSRB) una mejor precisión de las Redes Neuronales con capas Convolucionales por encima de las Redes Neuronales con capas completamente conectadas. La precisión de las Redes Neuronales Convolucionales para la clasificación de imágenes es mayor que la de los humanos.

Los resultados del análisis de la relación entre los histogramas de pesos y el número de imágenes por categoría demostraron, para los dos subconjuntos, una relación directa en modelos simples.

Se demostró que una mayor complejidad en los modelos de Redes Neuronales es capaz de corregir la relación entre el número de imágenes por categoría y su precisión. Una mayor complejidad en los modelos también es capaz de corregir un sobreajuste del modelo al subconjunto de prueba.

6. Bibliografía

[1] W. S. Mcculloch y P. Walter, «A logical calculus of the ideas immanent in neurons activity,» *Bulletin of matemathical biophysics,* vol. 5, 1945.

[2] Y. LeCun, «A theoretical framework for Back-Propagation,» *Proceedings of the 1988 Connectionist Models Summer School,* 1988.

[3] A. Wong, M. Jvad Shafiee y M. S. , «µNet: A Highly Compact Deep Convolutional Neural Network Architecture for Real-time Embedded Traffic Sign Classification,» 2018.

[4] Cireşan, Dan, Meier, Ueli, Masci, Jonathan y Schmid, «A committee of neural networks for traffic sign classification,» *IEEE International joint conference on neural networks,* 2011.

[5] Dan Ciresan, Ueli Meier, Jonathan Masci y Jurgen, «Multi-column deep neural network for traffic sign classification,» *Neural Networks,* 2012.

[6] Junqi Jin , Kun Fu y Changshui Zhang, «Traffic sign recognition with hinge loss trained convolutional neural networks,» *IEEE Transactions on Intelligent Transportation Systems,,* 2014.

[7] Hamed Habibi Aghdam, Elnaz Jahani Heravi y Domenec , «A practical approach for detection and classification of traffic signs using convolutional neural networks.».

[8] Arcos-García, Álvarez-García y Soria-Morillo, «Deep neural network for traffic sign recognition systems: An analysis of spatial transformers and stochastic optimisation methods.,» *Neural Networks,* vol. 99, pp. 158-165, 2018.

[9] Stallkamp, Johannes, Schlipsing, Marc, Salmen, Jan y Igel, Christian, «The German Traffic Sign Recognition Benchmark: A multi-class classification competition,» de *Proceedings of the International Joint Conference on Neural Networks*, 2011.

[10] R. Timofte y L. Van Gool, «Sparse Representation Based Projections. In British Machine Vision Conference,» *BMVC,* 2011.

[11] Andrew Ng, *Deep Learning Specialization,* 2017.

[12] W. Abdulla, «Medium,» Diciembre 2016. [En línea]. Available: https://medium.com/@waleedka/traffic-sign-recognition-with-tensorflow-629dffc391a6.

[13] J. Stallkamp, M. Schlipsing, J. Salmen y C. Ig, «Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition,» *Neural networks,* vol. 32, p. 323–332, 2012.

[14] D. E. Rumelhart, G. E. Hinto y Ronald J., «Learning representations by back-propagating errors,» *Nature,* 1986.

[15] M. Nielsen, «How the backpropagation algorithm works,» 2017. [En línea]. Available: http://neuralnetworksanddeeplearning.com/chap2.html.

[16] E. Mislej, «Minimum Viagle Models in Science Data,» Noviembre 2016. [En línea]. Available: https://www.kdnuggets.com/2016/11/practical-data-science-building-minimum-viable-models.html.

[17] K. Jalan, «How to Improve ML Algorithms,» [En línea]. Available: https://towardsdatascience.com/how-to-improve-my-ml-algorithm-lessons-from-andrew-ngs-experience-ii-f66926926f88.

[18] A. Ananthram, «Xavier Initialization For Neural Networks,» [En línea]. Available: https://towardsdatascience.com/random-initialization-for-neural-networks-a-thing-of-the-past-bfcdd806bf9e.

[19] «Rectifier (neural networks),» [En línea]. Available: https://en.wikipedia.org/wiki/Rectifier_(neural_networks).

APLICACIÓN DE REDES NEURONALES CONVOLUCIONALES AL RECONOCIMIENTO DE SEÑALES DE TRÁFICO

Isidre Cañellas Colom

Abstract

PROYECTO: **Convolutional Neural Nets applied to traffic sign recognition**

AUTOR: **Isidre Cañellas Colom**

DIRECTORES: Miguel Cuartas Hernández

Amaya Lobo García de Cortázar

PALABRAS CLAVE: *Machine Learning, Python, Neural Nets, Convolutional Nets, Traffic Sign Recognition, Big Data*

1. Introduction

Neural Networks are mathematical models inspired by the biological behaviour of neurons and the structure of the brain, this model was first proposed in 1943 by McCulloch and Pitts [1]. They are used to solve a wide range of classification problems. These networks are very tolerant to measurement and noise errors in the data set, they are also robust to the failure of few neurons.

Convolutional Neural Networks (CNN) are a special type of Neural Network proposed by Yan LeCun in 1988 [2]. One of the most popular applications for this type of architecture is the classification of images.

Application of Neural Networks to the recognition of traffic signals has been a breakthrough in the results performance and the published bibliography [3]. Most published studies focus on Convolutional Neural Nets [4], [5], [6], [7], [8].

The objective of this paper is to describe and implement models of Neural Networks on a real dataset. Four structures of Neural Networks have been developed with increasing difficulty and complexity and has been applied to the recognition of traffic signals. Two datasets of European traffic signs have been used: German Traffic Sign Recognition Benchmark (GTSRB) [9] y *Belgium Traffic Sign Dataset (BTSD)* [10].

2. Theoretical background

The basic computing element of Neural Networks is usually called a node or unit and is associated to a weight "w" that is modified during the learning process. Each node receives an input and returns the product of this by its associated weight.

The strategy followed for the analysis of the data, consists of four prediction models with different structures, of increasing difficulty and complexity. The programming of the models was based on the course of *Deep Learning* de Andrew NG [11] and the article by Waleed Abdulla [12].

For the first model, a very simple Neural Network with a hidden layer with number of nodes equal to the number of categories was implemented. The second model implemented had the same structure as the first model and included improvements in the activation function (ReLU)

and in the optimization algorithm (Minibatches). For the third model, a more complex Neural Network was constructed with three hidden layers of decreasing size.

The fourth model implemented constitutes a four-layer Convolutional Neuronal Network. The first three layers are convolutional and contain 8, 12 and 13 filters (5x5) respectively along with a 2x2 Maxpool layer; Generating 7.800 parameters to optimize. The last layer is completely connected with 43 nodes and a classification function called Softmax.

3. German Traffic Sign Recognition Benchmark (GTSRB)

The GTSRB contains 43 classes and 50,000 images divided into two subsets of data for training and validation, the training subset contains 39,209 images and the validation subset contains 12,630 images

The previous analysis of the data set observed a dispersion in the distribution of the number of images by category and great variety of visual appearances in the images: light conditions, rotation, climatology and scaling.

During the results analysis, the accuracy of the models on the training and validation subsets were studied. In order to study the variation of the results, 5 independent tests were performed on the same model. In a deeper analysis, we studied the optimized weights matrix "W" and the influence of the number of images per category on the results.

There was an increase in accuracy in the training subset along with the complexity of the models. The variance and the mean of the precision differences between the training and validation subset also decreased overall.

Finally, Model 4 (Convolutional layers) obtained 90.8% accuracy on the validation subset, a test variance of 0.63 and a mean of the differences between the subsets accuracy of 10.37%. From these results, an avoidable margin of error of 8-9% is obtained, which can be considered very satisfactory. The mean of the precision differences between subsets of 10.37% together with a precision of 99% on the training subset indicated a slight overfitting of Model 4 to the test subset.

4. Belgium Traffic Sign Dataset

The BTSD contains 63 classes and two subsets of training and validation with 4,575 and 2,520 images respectively.

The analysis of the weight matrix "W" and its relation to the number of images per category and the precision by category demonstrated a direct relationship between the number of images per category and the precision of the model by category. The images with less representation on the training subset obtained a lower precision.

The observed trends on the GTSRB data set were confirmed for the BTSD. Better results were observed for the BTSD although the number of images in this data set is much lower than for the GTSRB and its distribution of images by categories is more dispersed.

5. Conclusiones

It was observed in the two datasets used (BTSD and GTSRB) a better accuracy of the Neural Networks with Convolutional layers above the Neural Networks with completely connected layers. The accuracy of Convolutional Neural Networks for the classification of images is greater than that of humans.

The results of the analysis of the relationship between weight histograms and the number of images per category demonstrated, for the two subsets, a direct relationship in simple models.

It was demonstrated that a greater complexity in the models of Neural Networks is able to correct the relation between the number of images by category and its precision. Greater complexity in the models is also able to correct an overfitting of the model to the test subset.

6. Bibliography

[1] W. S. Mcculloch and P. Walter, "A logical calculus of the ideas immanent in neurons activity," *Bulletin of matemathical biophysics,* vol. 5, 1945.

[2] Y. LeCun, "A theoretical framework for Back-Propagation," *Proceedings of the 1988 Connectionist Models Summer School,* 1988.

[3] A. Wong, M. Jvad Shafiee and M. S. , "µNet: A Highly Compact Deep Convolutional Neural Network Architecture for Real-time Embedded Traffic Sign Classification," 2018.

[4] Cireşan, Dan, Meier, Ueli, Masci, Jonathan and Schmid, "A committee of neural networks for traffic sign classification," *IEEE International joint conference on neural networks,* 2011.

[5] Dan Ciresan, Ueli Meier, Jonathan Masci and Jurgen, "Multi-column deep neural network for traffic sign classification," *Neural Networks,* 2012.

[6] Junqi Jin , Kun Fu and Changshui Zhang, "Traffic sign recognition with hinge loss trained convolutional neural networks," *IEEE Transactions on Intelligent Transportation Systems,,* 2014.

[7] Hamed Habibi Aghdam, Elnaz Jahani Heravi and Domenec , "A practical approach for detection and classification of traffic signs using convolutional neural networks.".

[8] Arcos-García, Álvarez-García and Soria-Morillo, "Deep neural network for traffic sign recognition systems: An analysis of spatial transformers and stochastic optimisation methods.," *Neural Networks,* vol. 99, pp. 158-165, 2018.

[9] Stallkamp, Johannes, Schlipsing, Marc, Salmen, Jan and Igel, Christian, "The German Traffic Sign Recognition Benchmark: A multi-class classification competition," in *Proceedings of the International Joint Conference on Neural Networks*, 2011.

[10] R. Timofte and L. Van Gool, "Sparse Representation Based Projections. In British Machine Vision Conference," *BMVC,* 2011.

[11] Andrew Ng, *Deep Learning Specialization,* 2017.

[12] W. Abdulla, "Medium," Diciembre 2016. [Online]. Available: https://medium.com/@waleedka/traffic-sign-recognition-with-tensorflow-629dffc391a6.

[13] J. Stallkamp, M. Schlipsing, J. Salmen and C. Ig, "Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition," *Neural networks,* vol. 32, p. 323–332, 2012.

[14] D. E. Rumelhart, G. E. Hinto and Ronald J., "Learning representations by back-propagating errors," *Nature,* 1986.

[15] M. Nielsen, "How the backpropagation algorithm works," 2017. [Online]. Available: http://neuralnetworksanddeeplearning.com/chap2.html.

[16] E. Mislej, "Minimum Viagle Models in Science Data," Noviembre 2016. [Online]. Available: https://www.kdnuggets.com/2016/11/practical-data-science-building-minimum-viable-models.html.

[17] K. Jalan, "How to Improve ML Algorithms," [Online]. Available: https://towardsdatascience.com/how-to-improve-my-ml-algorithm-lessons-from-andrew-ngs-experience-ii-f66926926f88.

[18] A. Ananthram, "Xavier Initialization For Neural Networks," [Online]. Available: https://towardsdatascience.com/random-initialization-for-neural-networks-a-thing-of-the-past-bfcdd806bf9e.

[19] "Rectifier (neural networks)," [Online]. Available: https://en.wikipedia.org/wiki/Rectifier_(neural_networks).

APLICACIÓN DE REDES NEURONALES CONVOLUCIONALES AL RECONOCIMIENTO DE SEÑALES DE TRÁFICO

Isidre Cañellas Colom

Introducción

Las Redes Neuronales (NN: Neural Networks) son modelos matemáticos inspirados en el comportamiento biológico de las neuronas y en la estructura del cerebro, y es utilizada para resolver un amplio rango de problemas de *Machine Learning*.

El primer modelo de red neuronal fue propuesto en 1943 por McCulloch y Pitts [1] en términos de un modelo computacional de actividad nerviosa. Este modelo era un modelo binario, donde cada neurona tenía un escalón o umbral prefijado, y sirvió de base para los modelos posteriores.

Las Redes Neuronales artificiales han tenido un gran auge a partir de los años ochenta, debido al aumento en la potencia de computación, quedando demostrada su capacidad para resolver tareas de predicción sobre una serie temporal de datos. Estas redes son tolerantes a errores de medida o ruido en el conjunto de datos, también son muy robustas al fallo de pocas neuronas.

Algunos de los campos donde más se están aplicando este tipo de modelos de predicción es en el reconocimiento de voz, reconocimiento de imágenes, fraudes informáticos, *trading* de capital, modelos atmosféricos, etc.

Las Redes Neuronales Convolucionales (RNC) son una arquitectura especial de Red Neuronal propuesta por Yan LeCun en 1988 [2]. Una de las aplicaciones más populares para este tipo de arquitectura es la clasificación de imágenes. Por ejemplo, Facebook usa RNC para etiquetar de forma automática, Amazon para generar recomendaciones y Google para buscar entre las fotos de usuarios.

El reconocimiento de señales de tráfico es un problema de clasificación de imágenes sobre varias categorías. Las **señales de tráfico** muestran una amplia gama de **variaciones** entre sus clases en términos de **color, forma y presencia texto**. Sin embargo, existen un subconjunto de clases (por ejemplo, signos de límite de velocidad) que son muy similares entre sí.

El modelo de predicción tiene que hacer frente a grandes **variaciones** en las apariencias visuales debido a cambios en la **iluminación, oclusiones parciales, rotaciones, condiciones climáticas, escalamiento, etc**. Las señales de tráfico están diseñadas para ser reconocidas fácilmente por humanos, en consecuencia, los seres humanos son capaces de reconocer la gran variedad de señales de tráfico existente con una precisión del 98% [13].

La aplicación de Redes Neuronales al reconocimiento de señales de tráfico ha supuesto un punto de inflexión en los resultados obtenidos y en la cantidad bibliografía publicada [3]. La mayoría de los estudios publicados, se centran en diseños de Redes Neuronales Convolucionales [4], [5], [6], [7], [8].

El objetivo de este trabajo es describir e implementar un modelo de Red Neuronal a un conjunto de imágenes reales. Para ello, hemos desarrollado cuatro modelos de Redes Neuronales con dificultad y complejidad creciente que se han aplicado a dos bases de datos de señales de tráfico europeas: German Traffic Sign Recognition Benchmark (GTSRB) [9] y *Belgium Traffic Sign Dataset (BTSD)* [10].

El trabajo se organiza en tres bloques, en el primer bloque se describe la base teórica de las Redes Neuronales; el segundo y tercer bloque son una aplicación práctica de las técnicas descritas sobre las bases de datos GTSRB y BTSD.

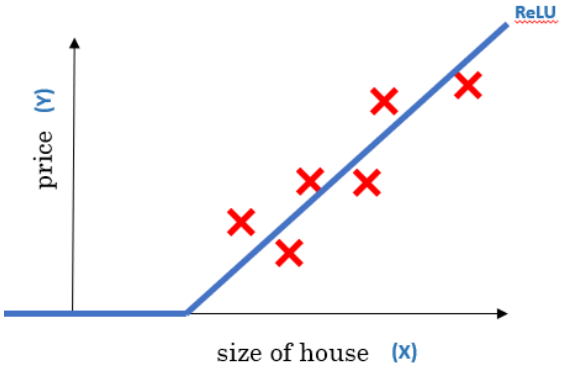
Base teórica de las Redes Neuronales

1. Introducción

EL elemento básico de computación de las Redes Neuronales se llama habitualmente nodo o unidad y está asociado a un peso *"w"* que se va modificando durante el proceso de aprendizaje. Cada nodo recibe un input y devuelve el producto de este por su peso asociado.

El ejemplo más simpe de una red neuronal, es una regresión lineal con una sola neurona, un solo *input* y un *output*. Se dispone de un conjunto de datos "X" (tamaño del inmueble) relacionados con "Y" (precio de un inmueble) y se quiere predecir para una nueva instancia x_i su correspondiente y_i.

Se implementa una regresión lineal con una función de optimización tipo ReLU y se optimizan los pesos *"W"* del modelo.

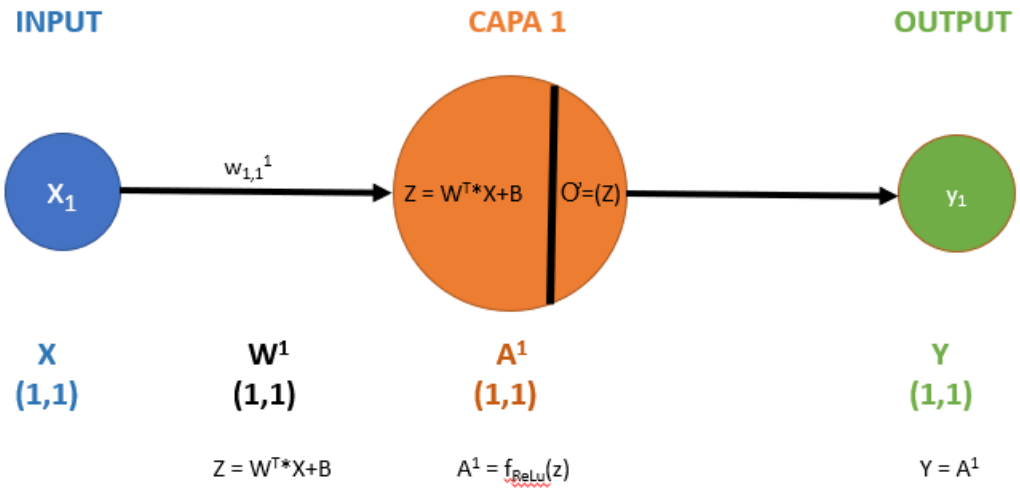


$$z_i = w * x_i + b$$

$$y_i = f_{ReLU}(z_i)$$

Ilustración 1. Regresión lineal

El objetivo del modelo es encontrar el valor de *"w"* y "b" para que los datos disponibles se ajusten de una manera precisa. Para ello, las Redes Neuronales introducen de forma iterativa el conjunto de datos "X" e "Y" para optimizar el valor de "W".



1. Redes Neuronales completamente conectadas

Redes Neuronales múltiples

Cuando el modelo a implementar debe representar subconjuntos de datos con más de un *input,* se habla de Redes Neuronales múltiples. Este tipo de red neuronal puede dotarse de uno o más nodos y varias capas.

Siguiendo el ejemplo definido anteriormente, el *output* de un modelo "Y" (precio de un inmueble) se puede ver afectado por más de un *input* "X" (tamaño, zona, número dormitorios, etc.). La representación de una red neuronal múltiple se muestra en la siguiente figura.

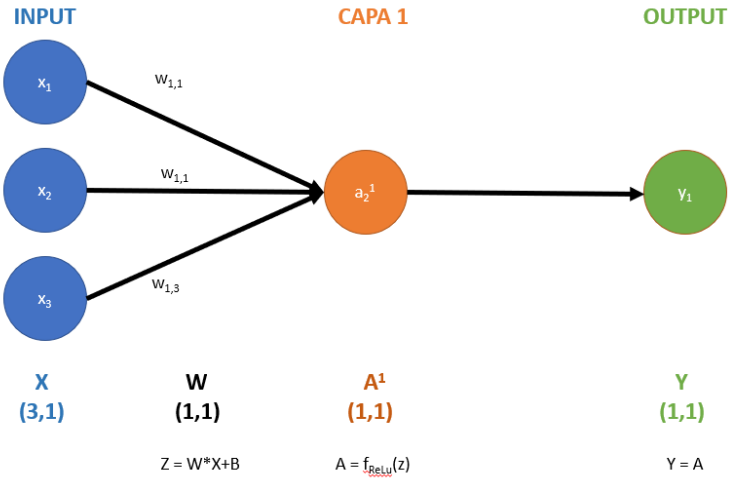


Ilustración 2. Red neuronal múltiple

La formulación de una red neuronal múltiple sigue siendo la misma que para una red neuronal simple (Y = W*X + b). Sin embargo, dado que los *inputs* del modelo "X" son un vector, la ecuación de regresión lineal es ahora matricial.

$$Z = W^T * X + B$$

DIMENSIONES → $(1,1) = (1,3) * (3,1) + (1,1)$

$$Y = f_{ReLU}(Z)$$

DIMENSIONES → $(1,1) = f_{ReLU}(1,1)$

Si el modelo debe representar un conjunto de datos con más de un output "Y", el conjunto de pesos se modificaría a un tamaño de 3x3. A continuación se muestra la formulación y el esquema de la red neuronal.

$$Z = W^T * X + B$$

DIMENSIONES → $(3,1) = (3,3) * (3,1) + (3,1)$

$$Y = f(Z)$$

DIMENSIONES → $(3,1) = f_{softmax}(3,1)$

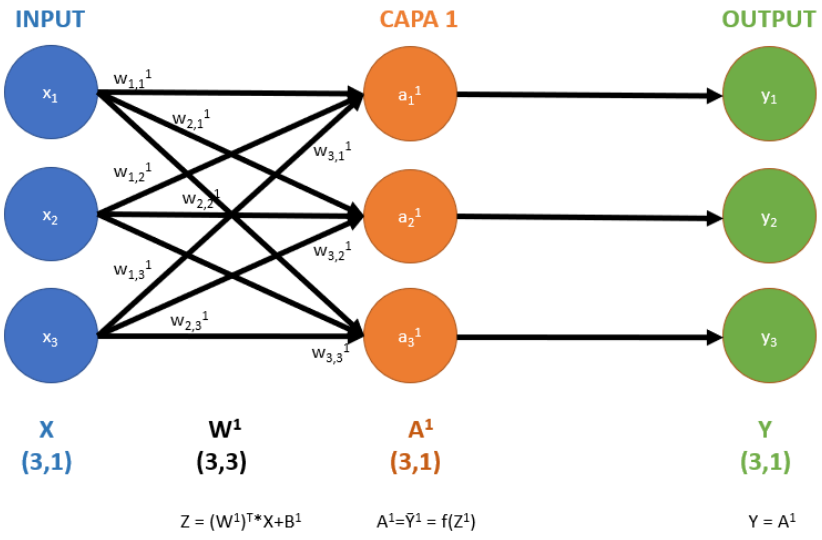


Ilustración 3. Red Neuronal múltiple (3,3)

Siguiendo los criterios de notación generales, los elementos a una capa se indica en superíndice.

Las estructuras de las Redes Neuronales de una sola capa, quedan completamente definidas por las dimensiones de los *inputs* y *outputs*. Sin embargo, para Redes Neuronales de más de una capa las dimensiones son arbitrarias y deben ser tratadas como un hiperparametro.

A continuación, se muestra la formulación y esquema de una red neuronal múltiple de dos capas ocultas.

$$Z^1 = (W^1)^T * X + B^1 \rightarrow A^1 = f_{ReLU}(Z^1) \rightarrow Z^2 = (W^2)^T * A^1 + B^2 \rightarrow Y = f_{sigmoid}(Z^2)$$

$$(3,1) = (3,3) * (3,1) + (3,1) \rightarrow (3,1) = f_{ReLU}(3,1) \rightarrow (2,1) = (3,2) * (2,1) + (2,1) \rightarrow (2,1) = f(2,1)$$

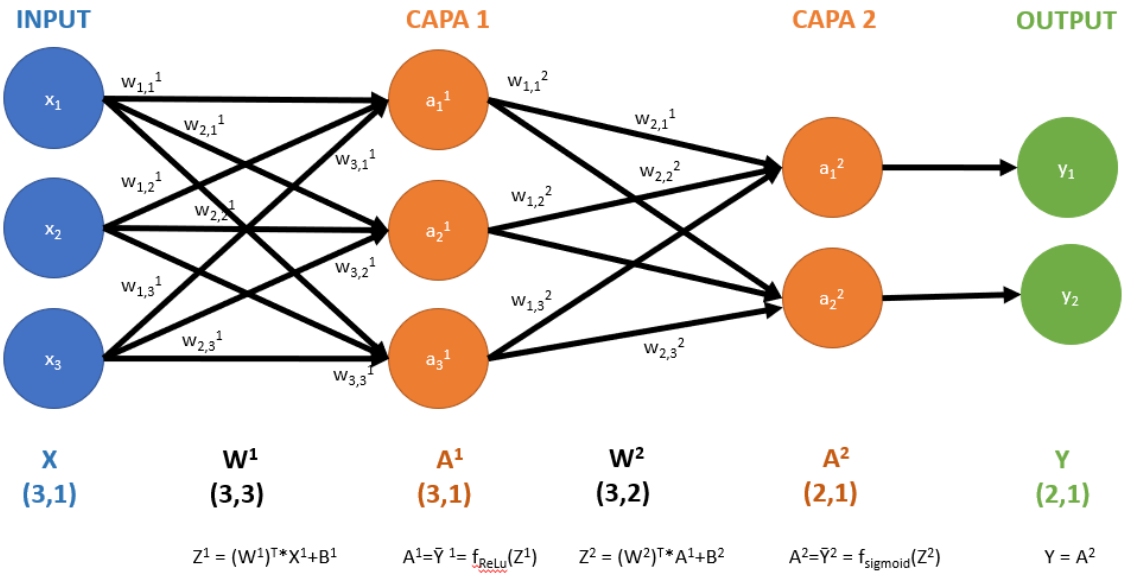


Ilustración 4. Red Neuronal múltiple con 2 capas

Función de activación

Las funciones de activación, se aplican al resultado de la regresión lineal (Z=W*X+B) con el objetivo de acotar el resultado. El valor de Z puede variar entre -∞ y +∞, por lo tanto, es necesario aplicar una función que acote estos resultados.

Las Redes Neuronales devuelven probabilidades sobre si un evento se va a producir o no, la función de activación, permite acotar entre 0 y 1 el resultado de las regresiones lineales en cada nodo y por lo tanto representan una probabilidad.

Existen multitud de funciones de activación, las más utilizadas para *Machine Learning* son las siguientes: *Sigmoid, Tanh, ReLU y Leaky ReLU.*

La función *Sigmoid()* se utiliza principalmente para problemas de clasificación entre dos clases. El *output* de la función se acota entre 0 y 1.

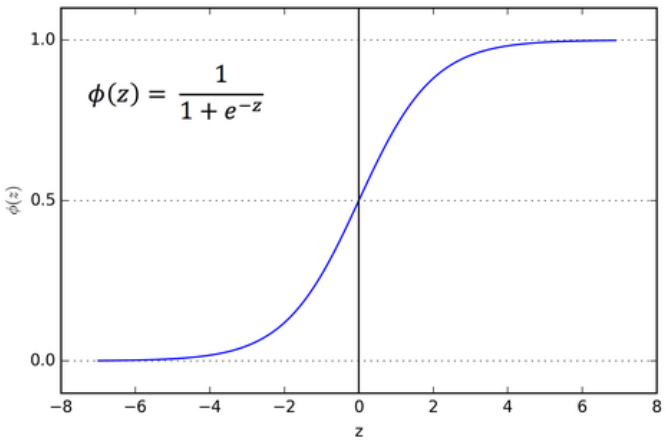


Ilustración 5. Función SIgmoid()

La función más utilizada en Redes Neuronales es ReLU. Esta función permite desactivar los valores negativos, igualándolos a 0. Para Redes Neuronales muy grandes, la función ReLU funciona muy bien porque desactiva neuronas (igualando a 0) que no aportan valor al modelo.

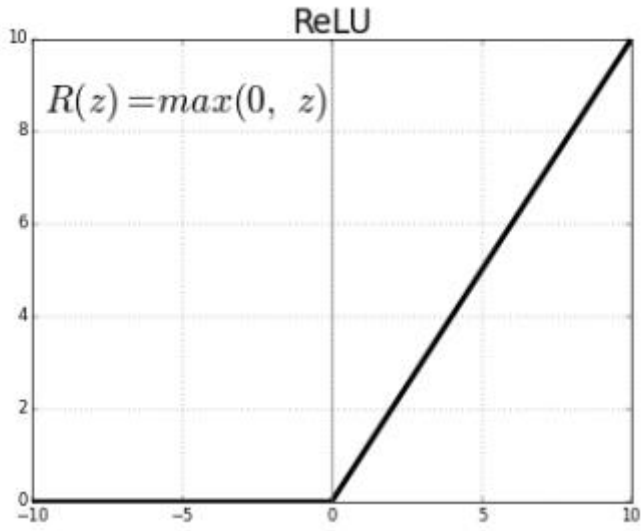


Ilustración 6. Función ReLU

Dado que la pendiente para valores negativos es horizontal y su derivada es igual a 0, en el proceso de optimización, aquellos pesos igualados a 0 no se pueden optimizar causando la muerta del nodo para ese modelo. Este problema (*dying ReLU problema)*, profundamente estudiado, se resuelve en la función *Leaky ReLU.*

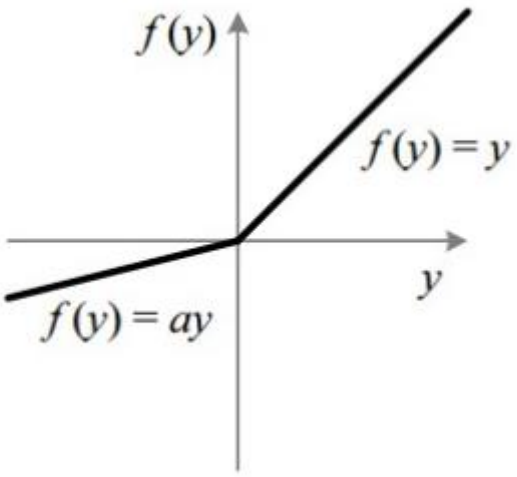


Ilustración 7. Función Leaky ReLU

La función Leaky ReLU se modificar con una pequeña pendiente en su parte negativa para evitar que el gradiente sea 0 y posibilitar la optimización de aquellos pesos igualados a 0.

La elección de la función de activación es clave para el rendimiento del modelo. Recientemente todos los estudios realizados recomiendan la utilización de *Leaky ReLu* como función de activación.

Función de clasificación

La función de clasificación se sitúa después de la última capa de las Redes Neuronales. Tiene como *input* los valores de la última regresión lineal (Z = W*X+B) y devuelve una probabilidad para cada categoría que se tienen que clasificar.

Si el modelo tiene que clasificar entre dos categorías se utiliza una función tipo *sigmoid()* que devuelve un valor entre 0 y 1. De esta manera, si el modelo de la Ilustración 4, obtiene los siguientes valores después de la función *sigmoid()* [0.2 , 0.8] significa que el modelo está otorgando a los datos de entrada "X" un 20% de probabilidad de pertenecer a la categoría 1 y un 80% de pertenecer a la categoría 2.

Para modelos con más de una categoría, es necesario aplicar una función distinta que devuelva un porcentaje a cada una de las categorías la función *Softmax* permite devolver porcentajes para clasificaciones de más de una categoría, su ecuación se muestra a continuación.

$$\varphi_{softmax}(Z^i) = \frac{e^{z_j^i}}{\sum_{j=0}^{n} z_j^i}$$

El esquema de una Red Neuronal con tres categorías a clasificar y un clasificador tipo Softmax se muestra a continuación.

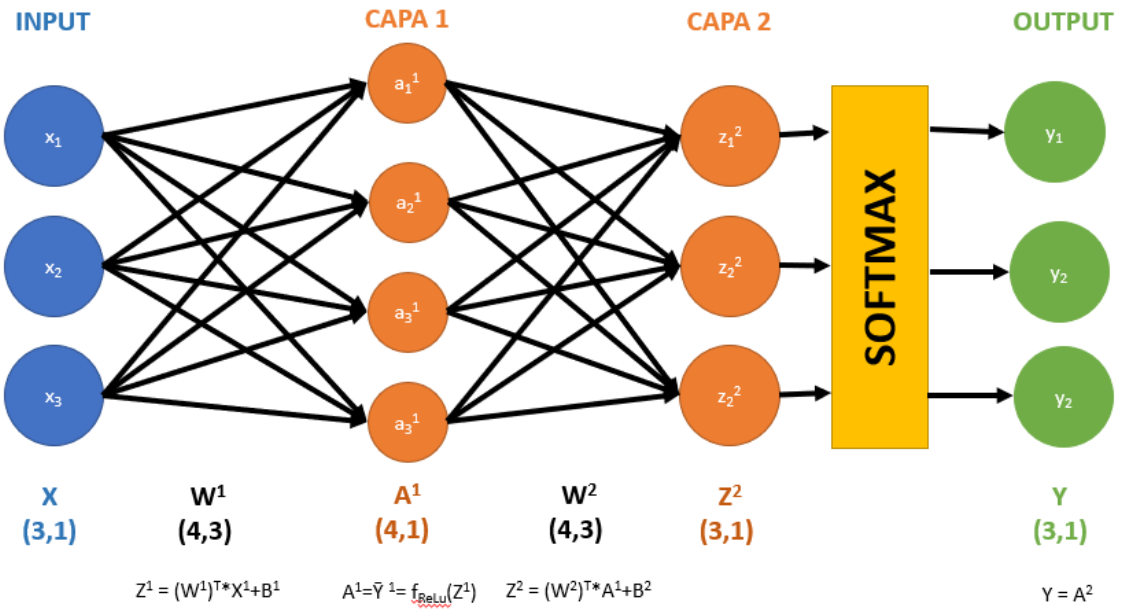


Ilustración 8. Esquema Red Neuronal tipo SOFTMAX

Optimización del modelo (*Back Propagation)*

La optimización de las Redes Neuronales se realiza introduciendo de forma iterativa el conjunto de datos "X" (*forward propagation*) y optimizando en cada iteración los pesos "W" (*back propagation)* [14].

La función pérdida (L(Y',Y)), mide el error de los datos obtenidos con el modelo (Y') respecto a los datos reales del conjunto de datos (Y). Las funciones más habituales para el cálculo del error del modelo son el error cuadrático medio y sobre todo el error logarítmico medio.

$$L(y_i', y_i) = -(y_i * \log(y_i') + (1 - y_i) * \log(1 - y_i')$$

La función coste, calcula la media de las funciones pérdida de todo el conjunto de datos. Esta función depende de los resultados obtenidos por el modelo (Y') y por lo tanto depende también de los pesos (W) y del vector bias (b). Tenemos por lo tanto que el coste del modelo depende de los parámetros W y b del modelo y también del vector Y del conjunto de datos, que es constante y no cambia con las iteraciones.

$$J(w, b) = \left(\frac{1}{m}\right) * \sum_{1}^{m} L(y_i', y_i)$$

Para toda red neuronal, se puede desarrollar la fórmula anterior en función de los pesos(w_i^l) y de los *bias* (b^l), teniendo en cuenta que los pesos y los *bias* son los parámetros a optimizar.

Por lo tanto, es posible derivar la función $J(w, b)$ respecto a cualquier peso (w_i^l) o *bias* (b^l). La formulación de las derivadas de las Redes Neuronales se ha desarrollado de forma extensa en muchos libros y artículos [11], [14], [15].

La función de optimización básica para las Redes Neuronales es *Gradient Descente*, esta función realiza las derivadas de todos los parámetros del modelo después de cada iteración y los actualiza según la siguiente ecuación.

$$W' = W - \alpha * d\left(\frac{J(w,b)}{dw}\right)$$

El parámetro "α" o tasa de aprendizaje marca el tamaño de la actualización del peso. A continuación, se muestra la diferencia entre una tasa de aprendizaje mayor y menor.

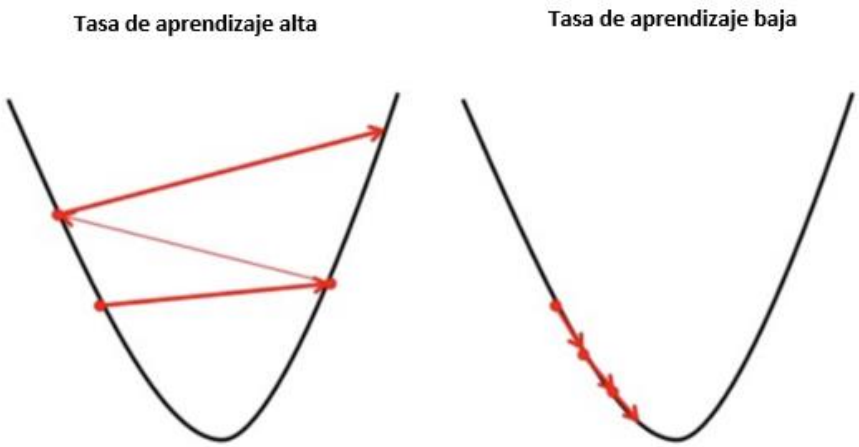


Ilustración 9. Tasa de aprendizaje

Existen gran variedad de algoritmos de optimización como los descritos anteriormente: *Exponentially weighted averages, Gradient descent with momentum, RMSprop, Adam optimization algorithm, Learning rate decay.* A continuación, se describen los más importantes.

Mini-batch Gradient descent

Para subconjuntos de datos muy grandes y Redes Neuronales profundas, introducir todo el subconjunto de datos entero en cada iteración supone ralentizar de forma innecesaria el modelo.

El algoritmo, divide el conjunto de datos total en mini conjuntos de datos o *mini-batches* y los introduce en el modelo uno a uno, realizando una optimización en cada mini subconjunto.

Si disponemos de un subconjunto de datos con 50.000 casos, el algoritmo Mini-batch Gradient descent puede dividir los 50.000 casos en 10.000 *batches* de 50 casos cada uno. Si nuestro modelo necesita 100 iteraciones para estabilizar el coste, el modelo se optimizará 5 millones de veces (100*10.000*50).

Gracient descent with momentum

Este algoritmo, actualiza los parámetros con una ponderación (ydW) de la derivada del *batch* actual (dW) y el anterior (vdW), con el objetivo de suavizar la curva. Para este algoritmo se disponen de dos tasas de aprendizaje, una para la ponderación de las derivadas (β) y la tasa de aprendizaje habitual (α).

A continuación, se muestra el código del algoritmo.

```
vdW = 0, vdb = 0
on iteration t:

        vdW = beta * vdW + (1 - beta) * dW
        vdb = beta * vdb + (1 - beta) * db
        W = W - learning_rate * vdW
        b = b - learning_rate * vdb
```

RMSprop

Este algoritmo utiliza la misma estructura que *Gracient descent with momentum* elevando al cuadrado la derivada del *batch* actual (dW) para el cálculo de la derivada ponderada (sdW). En la ecuación de actualización de los parámetros se utiliza el cociente de dW y sdW.

```
sdW = 0, sdb = 0
on iteration t:
        sdW = (beta * sdW) + (1 - beta) * dW^2  # squaring is element-wise
        sdb = (beta * sdb) + (1 - beta) * db^2  # squaring is element-wise
        W = W - learning_rate * dW / sqrt(sdW)
        b = B - learning_rate * db / sqrt(sdb)
```

Adam optimization

El algoritmo de optimización de Adam combina RMSprop y Gracient descent with momentum. De esta forma, este algoritmo dispone de cuatro tasas de aprendizaje (β_1, β_2, α y ¥).

```
vdW = 0, vdW = 0
sdW = 0, sdb = 0
on iteration t:
        # can be mini-batch or batch gradient descent
        compute dw, db on current mini-batch

        vdW = (beta1 * vdW) + (1 - beta1) * dW     # momentum
        vdb = (beta1 * vdb) + (1 - beta1) * db     # momentum

        sdW = (beta2 * sdW) + (1 - beta2) * dW^2   # RMSprop
        sdb = (beta2 * sdb) + (1 - beta2) * db^2   # RMSprop

        vdW = vdW / (1 - beta1^t)      # fixing bias
        vdb = vdb / (1 - beta1^t)      # fixing bias

        sdW = sdW / (1 - beta2^t)      # fixing bias
        sdb = sdb / (1 - beta2^t)      # fixing bias

        W = W - learning_rate * vdW / (sqrt(sdW) + epsilon)
        b = B - learning_rate * vdb / (sqrt(sdb) + epsilon)
```

Estructura de optimización

Es habitual dividir el conjunto de datos en dos subconjuntos con el objetivo de validar los resultados que obtiene el modelo. El modelo se optimiza con el subconjunto de prueba y una vez optimizados los parámetros y finalizadas todas las iteraciones se obtienen los resultados para el subconjunto de prueba o validación. De esta forma, el subconjunto de prueba no es utilizado para la optimización del modelo.

Es habitual que los modelos de Redes Neuronales tiendan a sobreajustar la optimización al subconjunto de prueba memorizando las imágenes. Con el subconjunto de prueba o validación se pueden comprobar y corregirá este tipo de tendencias.

2. Redes Neuronales Convolucionales y clasificación de imágenes

Las Redes Neuronales Convolucionales (RNC) son una arquitectura especial de Red Neuronal propuesta por Yan LeCun en 1988 [2]. Una de las aplicaciones más populares para este tipo de arquitectura es la clasificación de imágenes. Por ejemplo, Facebook usa RNC para etiquetar de forma automática, Amazon para generar recomendaciones y Google para buscar entre las fotos de usuarios.

El primer paso del reconocimiento de imágenes es introducir las imágenes en la Red Neuronal. Es importante destacar que las imágenes son, en realidad, un conjunto de números. La imagen se representa como una matriz con tantas columnas como número de píxeles del ancho de la imagen y filas igual al número de píxeles de alto de la imagen. Cada píxel tiene a su vez tres valores que representan el color en RGB.

De esta forma la imagen se representa como una matriz con dimensiones iguales a la resolución de la imagen. En cada posición de la matriz se encuentra un vector de tres valores representando el color del píxel. Dado que el *input* "X" de los modelos de *Machine Learning* debe ser un vector columna, la matriz se debe reconvertir. A continuación, se muestra la reestructuración de una imagen

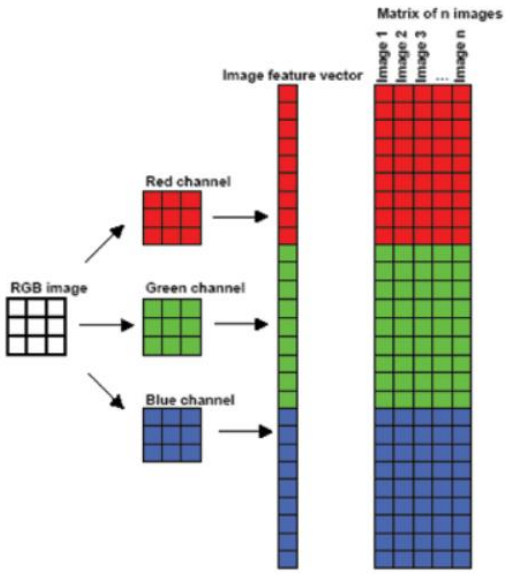


Ilustración 10. Imagen – matriz

De forma general los modelos de Redes Neuronales utilizan la misma técnica que los humanos para reconocer las imágenes, se buscan patrones identificativos generales tipo curvaturas, geometrías, colores, texto, etc. para poder clasificar las imágenes. En Redes Neuronales profundas, las primeras capas identifican las características más generales y las capas más profundas se especializan en detalles.

De una forma más específica, la imagen se pasa a través de una serie de Redes Convolucionales no lineales, capas de *pooling* y capas completamente conectadas para luego generar los outputs "Y".

Redes Convolucionales en 2D

Las Redes Convolucionales aplican una máscara o filtro a la matriz de entrada con el objetivo de encontrar patrones, estos filtros o máscaras son los pesos del modelo. El filtro, se va moviendo por toda la matriz de entrada y se multiplican cada instancia del filtro con la instancia de la matriz de entrada correspondiente.

Un ejemplo muy sencillo para un filtro es un detector de bordes como el que se muestra a continuación. En este caso tenemos una matriz de 6*6 (n,n) y un filtro 3*3 (f,f), se trata de colocar el filtro sobre la matriz y calcular el sumatorio de los productos de las celdas que coinciden. El tamaño de la matriz resultante siempre es n-f+1.

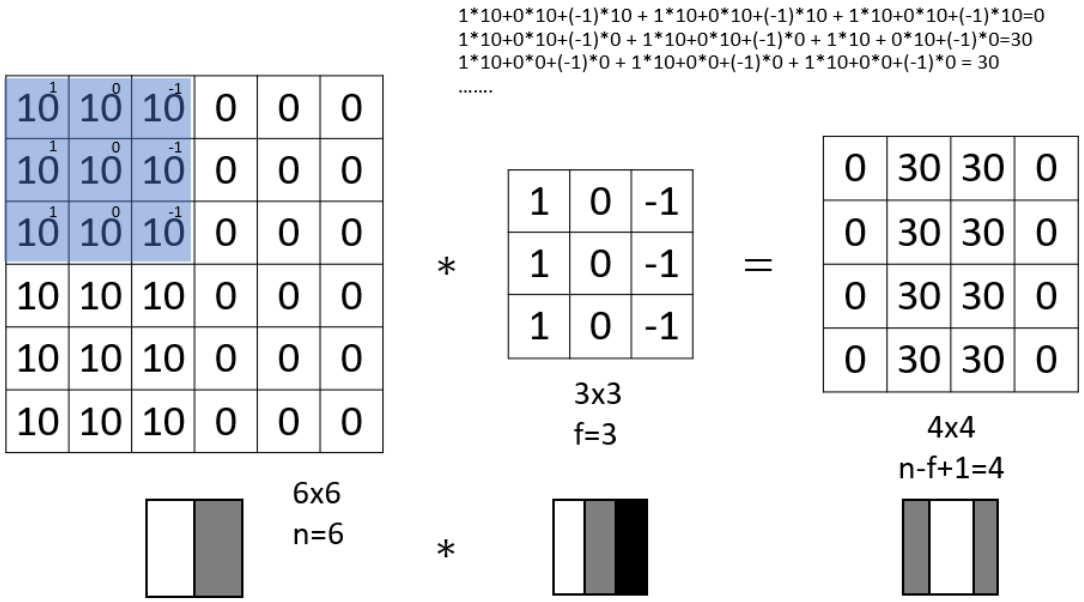


Ilustración 11. Filtro detector de bordes

Con el objetivo de no perder datos en los bordes de los imágenes, es posible empezar a aplicar el filtro fuera de la matriz con el objetivo de que todos los pesos del filtro se multipliquen por la matriz origen (padding). También se puede aplicar el filtro sobre la imagen saltando cada dos pixeles o cada tres (strading).

Redes Convolucionales en 3D

Los filtros sobre volúmenes son los que se utilizan para imágenes. Se utiliza la misma estructura de cálculo que en los filtros 2D, pero en este caso tenemos tantos filtros como profundas sean las matrices de entrada. Para las imágenes, dado que la profundidad es constante e igual a 3 (RGB) los filtros siempre tendrán una profundidad de 3.

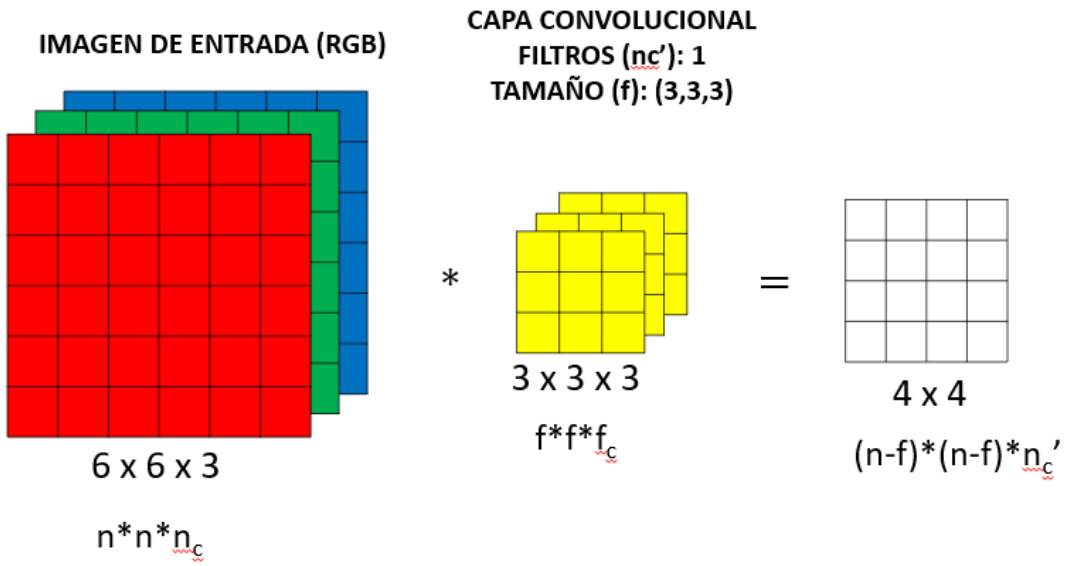


Ilustración 12. Capa convolucional

Los filtros representarían los nodos para las Redes Neuronales completamente conectadas y por lo tanto puede haber más de un filtro en una misma capa.

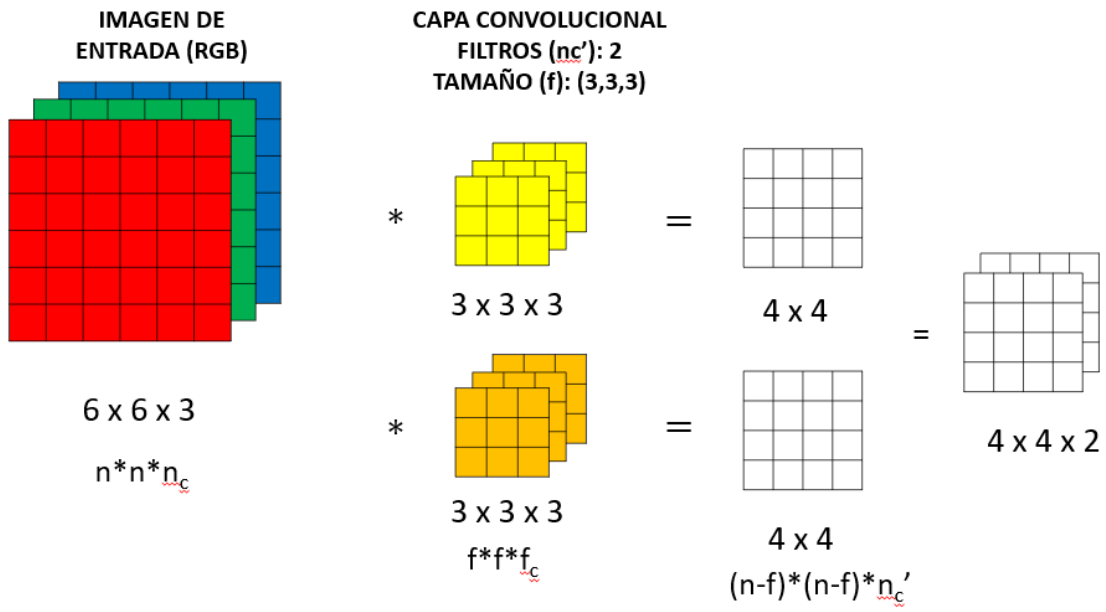
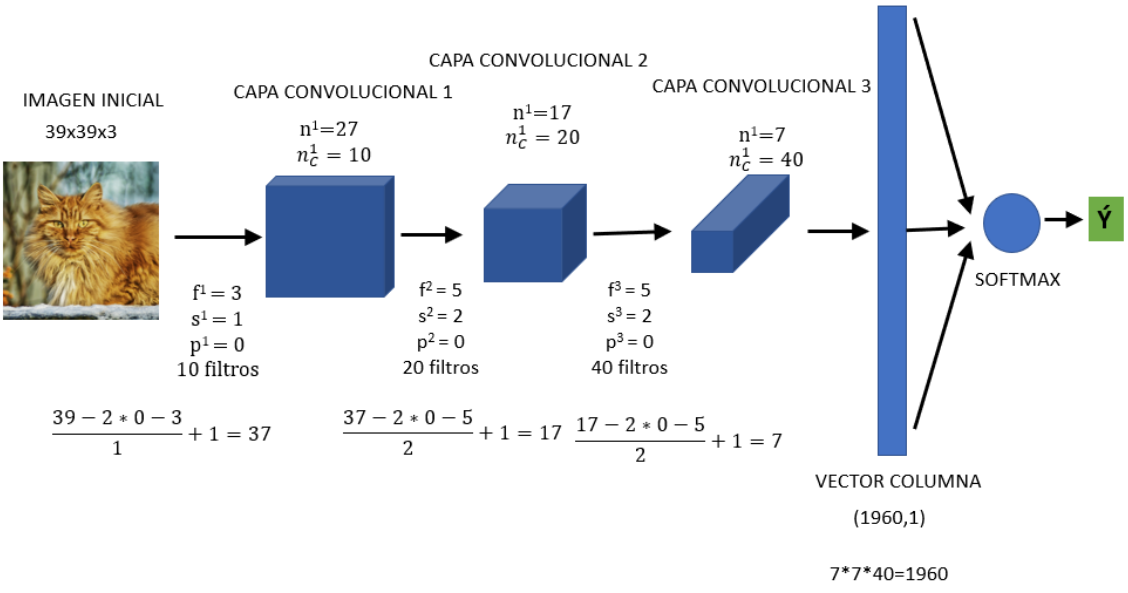


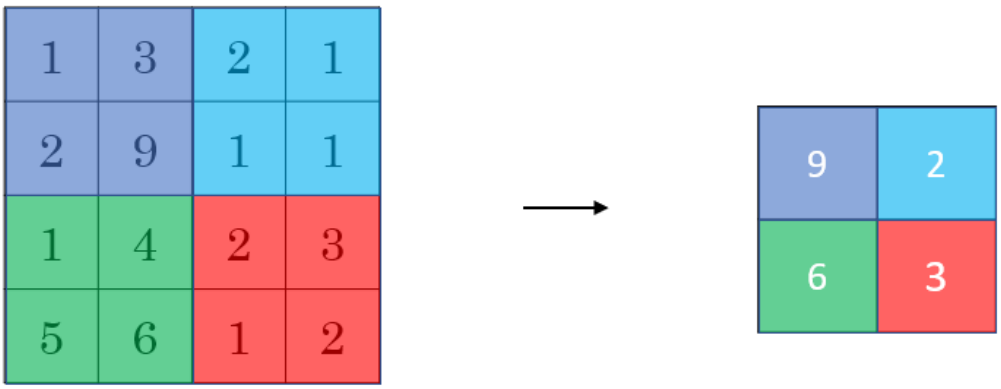
Ilustración 13. Red Convolucional con múltiples filtros

A continuación se muestra un ejemplo de una Red Neuronal Convolucional.



Capas pooling

Las capas pooling tienen el objetivo de reducir el tamaño de las capas Convolucionales. Su estructura es similar a las capas Convolucionales, sin embargo, devuelven el máximo o la media del trozo de la matriz origen sobre el que están aplicados.


1 3 2 1
2 9 1 1
1 4 2 3
5 6 1 2
9 2
6 3

German Traffic Sign Recognition Benchmark (GTSRB)

1. Introducción

El German Traffic Sign Dataset (GTSD) es un conjunto de datos muy utilizado y con gran presencia bibliográfica, en parte, debido a un concurso que se realizó en 2011 en la Conferencia Internacional de Redes Neuronales o IJCNN por sus siglas en inglés. Encontramos toda la información respecto al conjunto de datos y un resumen de las estrategias ganadoras del concurso, en el artículo *The German Traffic Sign Recognition Benchmark: A multi-class classification competition* [9].

Este conjunto de datos contiene más de 50.000 señales de tráfico clasificadas en 43 clases o categorías. Las imágenes abarcan distintas condiciones climatológicas y una gran variedad de exposiciones fotográficas. Este conjunto de datos fue extraído de 10 horas de vídeo filmadas en Alemania en los meses de marzo, octubre y noviembre del 2010.

Por cada señal de tráfico, filmada se extrae del vídeo un conjunto de fotografías o *frames* de esa señal. Disponemos entonces por cada señal de tráfico real o *instance en inglés*, de una o más imágenes para nuestro modelo, este conjunto de imágenes obtenidas de una misma instancia se denomina pista o *track*.

Para este conjunto de datos se obtuvieron un total 133.000 imágenes etiquetadas provenientes de 2.416 instancias reales que se clasificaron en 70 categorías. Se descartaron las pistas con menos de 30 imágenes y las categorías con menos de 9 pistas. En el caso de que la pista tuviera más de 30 imágenes, se tomaron 30 imágenes repartidas de forma equidistante.

Finalmente se obtuvieron más de 50.000 imágenes provenientes de 1.700 señales de trafico reales que se reparten en 43 clases. El tamaño de las imágenes varía entre 15x15 y 222X193 píxeles y siempre se respeta un margen del 10% para facilitar los detectores de bordes.

El conjunto de datos total se dividió en tres subconjuntos con el objetivo de utilizarlos como subconjuntos de entrenamiento, evaluación y validación. El primer subconjunto contiene el 50% de las imágenes y el segundo y tercero contienen 25% cada uno.

A fecha del 2018 en la página web del German Traffic Sign Dataset tenemos disponible el conjunto de datos utilizados en la primera fase el concurso o se puede optar por el conjunto de datos de la segunda fase. Para nuestro caso y con el objetivo de contar con el conjunto de datos más extenso optaremos por el segundo.

2. Análisis del conjunto de datos

El conjunto de datos se encuentra disponible en la web oficial del *German Dataset* y se encuentra dividido en archivos de entrenamiento y prueba.

El archivo de entrenamiento contiene 43 carpetas numeradas desde el 00000 hasta 00042. Cada carpeta contienen una serie de imágenes en formato *".ppm"* y un archivo *".csv"* que contiene información sobre la instancia original y su categoría. Las imágenes están numeradas indicando su pista y el número de la imagen respecto a la pista.

El conjunto de datos se importa a una *Python List* para luego convertirse en un *NDarray* dado que los modelos de *Machine Learning* utilizan *arrays* .El código para la importación de los datos se muestra a continuación. Este código está inspirado en los cursos de Andrew NG [11] y el artículo de Walled Abdulla [12].

```
def load_data(data_dir):
```

```
    directories = [d for d in os.listdir(data_dir)
                   if os.path.isdir(os.path.join(data_dir, d))]

    labels = []
    images = []
    #print(directories)
    for d in directories:
        label_dir = os.path.join(data_dir, d)
        file_names = [os.path.join(label_dir, f)
                      for f in os.listdir(label_dir) if f.endswith(".ppm")]
        for f in file_names:
            images.append(skimage.data.imread(f))
            labels.append(int(d))

    #print(label_dir)
    return images, labels
```

Para el archivo de prueba disponemos de una carpeta que contiene todas las imágenes en formato "*.ppm*" numeradas desde el '00000' hasta '12629' y un archivo .csv que contiene la información de la señal original y su categoría.

Dado que la estructura de las carpetas del subconjunto de entrenamiento y de evaluación son distintas, necesitaremos una nueva función para cargar los archivos a una lista Python. La solución utilizada consiste en modificar el archivo *.csv* para que solo contenga el nombre de las imágenes y su categoría, una vez modificado se puede utilizar un código similar al anterior.

```
def load_test_data (data_dir):
     labels   = []
     labels2  =[]
     images = []

     with open('GT-final_test.csv', 'r') as f:
        reader = csv.reader(f)
        labels = list(reader)
     #print(labels)
     for d in range(1,len(labels)):
          labels2.append(int(labels[d][0]))
     #print(len(labels))

     file_names = [os.path.join(data_dir, f)
                   for f in os.listdir(data_dir) if f.endswith(".ppm")]
     for f in file_names:
          images.append(skimage.data.imread(f))

     return images,labels2
```

Se obtiene un **conjunto de 39.209 imágenes para el entrenamiento** y otro **conjunto de 12.630 imágenes para la verificación** o prueba.

El conjunto de entrenamiento consta de dos listas Python de 39.209 instancias cada una. La lista de imágenes o vector "X" contiene 39.209 *arrays* 3D y la lista de categorías o vector "Y" contienen 39.209 instancias con el número de categoría de cada imagen.

A continuación, se muestra el contenido y las dimensiones de la lista de categorías.

```
print("ARRAY DE LABELS TRAIN")
print("Tipo de objeto Python: " , type(Y_train_orig))
print("Número de imágenes totales: ", len(Y_train_orig))
print("Número de clases totales: ", len(set(Y_train_orig)))
print(Y_train_orig)

Out[]:
ARRAY DE LABELS TRAIN
Tipo de objeto Python:  <class 'list'>
Número de imágenes totales:  39209
Número de clases totales:  43
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 1, 1, 1,…, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42,
42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42,
42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42,
42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42, 42]
```

Se trata de una liste de Python que contiene 39.209 imágenes y 43 categorías. Cuando se imprime la lista, se observa que contiene el número de categoría.

En un primer análisis visual del contenido del array de categorías, se observa que están orden descendiente y debemos tenerlo en cuenta dado que esto puede influir en la precisión del modelo, sobre todo a la hora de elegir el método de optimización.

La lista de las imágenes contiene 39.209 arrays de 3 dimensiones, las dos primeras dimensiones de cada vector son el número de pixeles de ancho y largo; la tercera dimensión, siempre igual a 3, se refiere al color en RGB. A continuación, se muestra las características de la lista.

```
Out[6]:
ARRAY DE IMÁGENES TRAIN
Tipo de objeto Python:  <class 'list'>
Número de imágenes totales:  39209
Dimensiones de la primera imagen:  (30, 29, 3)
Parámetros RGB del primer pixel:  [75 78 80]
Out[7]:
[array([[[ 75,  78,  80],
         [ 74,  76,  78],
         [ 86,  87,  84],
         ...,
         [ 68,  75,  75],
         [ 65,  69,  68],
         [ 66,  67,  66]],

        [[ 83,  84,  86],
         [ 80,  80,  82],
         [ 90,  89,  84],
         ...,
         [ 73,  77,  78],
         [ 76,  78,  75],
         [ 80,  80,  78]],

        [[ 78,  78,  80],
         [ 86,  85,  86],
         [ 93,  90,  91],
         ...,
         [ 72,  74,  72],
         [ 73,  74,  69],
         [ 78,  78,  74]],

        ...,

        [[133, 134, 139],
         [122, 127, 132],
         [112, 121, 124],
         ...,
         [ 94,  94,  89],
         [ 97,  98,  91],
         [ 99, 103,  99]],

        [[ 91,  95,  99],
         [ 91,  98, 103],
         [ 74,  85,  89],
         ...,
         [103, 101,  95],
         [110, 113, 104],
         [ 96, 104,  98]],

        [[ 85,  84,  87],
         [ 95, 101, 107],
         [ 61,  74,  79],
         ...,
         [102, 102,  95],
         [ 99, 102,  90],
```

```
         [ 90,  97,  89]]], dtype=uint8),

         ...,

        [[109, 107,  92],
         [109, 107,  92],
         [111, 108,  92],
         ...,
         [109,  85,  71],
         [114,  92,  75],
         [115,  96,  75]],

        [[ 98, 101,  88],
         [106, 107,  93],
         [110, 110,  94],
         ...,
         [119,  92,  77],
         [120,  91,  74],
         [115,  91,  72]],

        [[105, 105,  94],
         [108, 108,  94],
         [113, 113,  96],
         ...,
         [113,  88,  71],
         [113,  92,  75],
         [111,  91,  73]]], dtype=uint8), ...]
```

Se comprueba que se trata de una lista de Python con 39.209 vectores y que cada vector tiene 3 dimensiones. También se puede comprobar los valores RGB de cada píxel.

El subconjunto de prueba o verificación tiene una estructura muy similar y simplemente varia en el orden del vector “Y”, a diferencia del subconjunto de prueba en este caso, no está ordenado.

Visualización del conjunto de datos

Para aportar más claridad a los datos con los que se está trabajando y con el objetivo de poder comprobar que no se ha cometido ningún error al importarlos a Python, se va a visualizar las imágenes de nuestro conjunto de datos.

En la Ilustración 14, se muestra una imagen de referencia por cada categoría del GTSD. Entre paréntesis, al lado del número de cada categoría, tenemos el número de imágenes de nuestro conjunto de datos para esa categoría en concreto.

Ilustración 14, Imágenes referencia del GTSD

Se comprueba que nuestro conjunto de datos tiene 43 categorías. Se han separado las distintas señales de límite de velocidad, es habitual juntar todas las señales de límite de velocidad en una sola.

En la Ilustración 15, se muestra por cada categoría, una imagen real del GTSD junto con el número de categoría y el número de imágenes por cada categoría. El código utilizado para ello se basa en el artículo de W.A. y es el siguiente.

```
def display images and labels(images,labels):
     unique_labels = set(labels)
     plt.figure(figsize=(15, 15))
     i = 1
     for label in unique_labels:
          image = images[labels.index(label)]
          plt.subplot(8, 8,i)
          plt.axis('off')
          plt.title("Cat. {0} ({1})".format(label, labels.count(label)))
          i += 1
          _ = plt.imshow(image)

     plt.show()
```

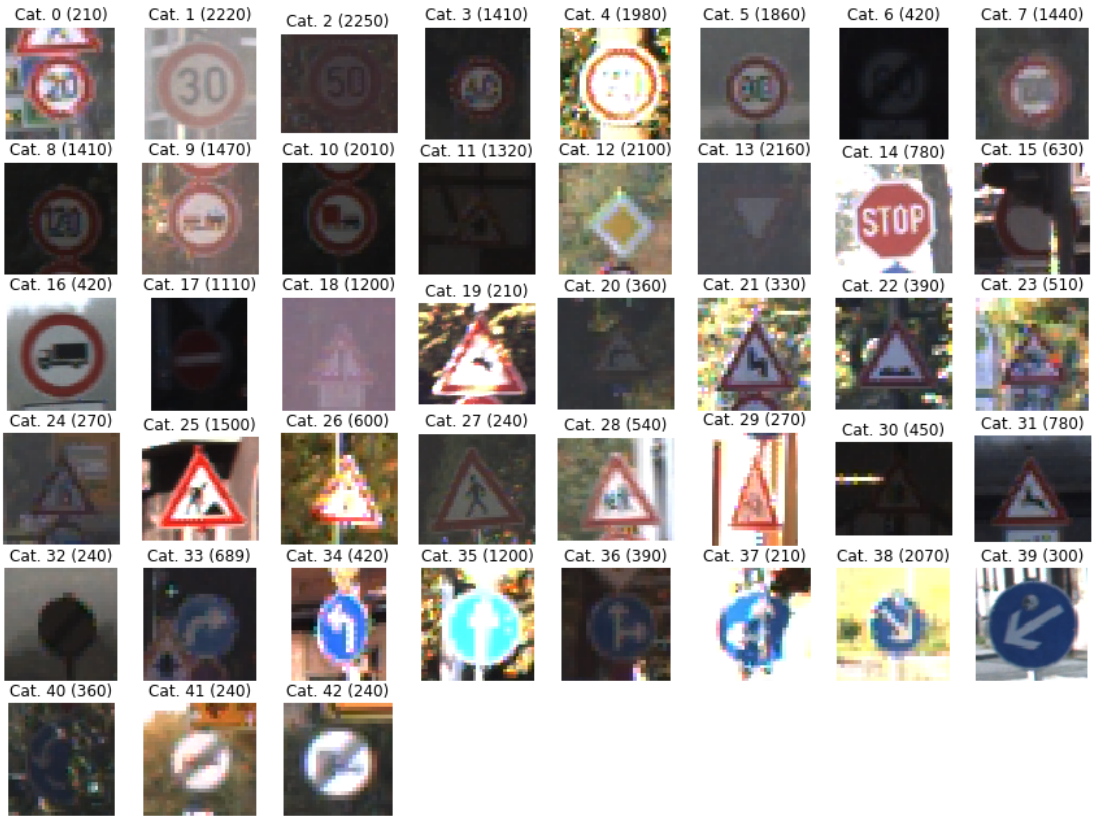


Ilustración 15, imágenes originales del GTSD

Con un análisis sobre la visualización de las imágenes del conjunto de datos, se consigue obtener una visión general de los datos con los que se está trabajando. Este análisis también permite sacar algunas conclusiones respecto a la calidad de las imágenes, el tipo de categorías y la relación entre cantidad de ejemplos y categorías.

Del análisis de las dos figuras anteriores se puede extraer las siguientes conclusiones:

- El número de imágenes por categoría es muy dispar.
- Todas las señales se encuentran centradas.
- El recorte de la señal no es preciso, se aprecian pequeños trozos de otras señales.
- Algunas categorías son muy parecidas.
- No todas las imágenes tienen el mismo tamaño.
- La calidad de las imágenes es baja.
- La exposición lumínica y el ángulo de la señal varía mucho.

De todas estas conclusiones, hay dos a las que habrá que prestar especial atención. Si el número de ejemplos por categoría es muy dispar se va a obtener un modelo que prediga algunas categorías muy bien y otras peor. Para los modelos de *Machine Learning*, necesitamos que todas las imágenes tengan las mismas dimensiones, por lo tanto deberemos modificarlas.

Distribución de ejemplos por categoría

Una distribución dispar de la cantidad ejemplos entre categorías suele generar problemas en los modelos de predicción de *Machine* Learing . La diferencia de representación causa un mayor acierto en las categorías mejor representadas. Una categoría peor representada tiene una menor importancia a la hora optimizar los pesos de la matriz “W”.

En la Ilustración 15, hemos detectado que la distribución del número de imágenes por categoría es muy dispar, por lo tanto, se va a realizar un análisis más detallado.

En la siguiente figura, se representa un histograma con el número de categorías por ejemplo.

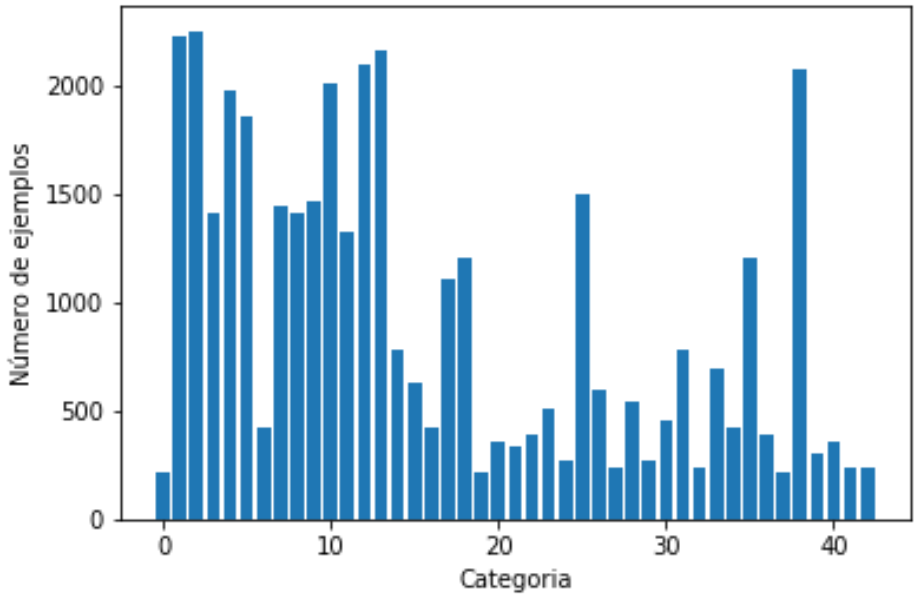


Ilustración 16, histograma número de ejemplos por categoría

Se observa que hay gran diferencia en el número de imágenes. De forma general, se puede dividir el histograma en tres zonas: categorías con más de 1500 imágenes, categorías entre 500 y 1000 imágenes y categorías con menos de 500 imágenes. Como ya se ha comentado, las categorías con menos representación son las que pueden generar problemas.

Se realiza un análisis más profundo sobre las categorías subrepresentadas. La siguiente función, devuelve un conjunto de imágenes a modo de ejemplo y el histograma para las categorías cuyo número de ejemplos es menor del indicado.

```
def display_categorias_menores_de_X(cat):
    unique_labels = set(Y_train_orig)
    #creamos una lista
    labels_count_y=[]
    labels_count_x=[]
    images_count=[]

    #con un for recorremos todas las categorias y metemos el número de imágenes
    #en cada categoria en la lista labels_count
    x=0
    for i in unique_labels:
        if Y_train_orig.count(i) < cat:
            labels_count_y.append(Y_train_orig.count(i))
            labels_count_x.append(i)

    #x=list(range(61))
    plt.bar(labels_count_x,labels_count_y)
    plt.xlabel('Categoria')
    plt.ylabel('Número de ejemplos')
    plt.title('CATEGORIAS CON MENOS DE '+str(cat)+' IMAGENES')
    plt.show()

    print('\n CATEGORIAS CON MENOS DE ',str(cat),' IMAGENES: ',labels_count_x)
    print('\n   HAY   UN   TOTAL   DE   ',len(labels_count_x),'CATEGORIAS   CON   MENOS   DE
',str(cat),'IMAGENES')

    plt.figure(figsize=(15, 15))
    i = 1
```

```
    for label in labels_count_x:
        image = X_train_orig[Y_train_orig.index(label)]
        plt.subplot(8, 8, i)
        plt.axis('off')
        plt.title("Label {0} ({1})".format(label, Y_train_orig.count(label)))
        i += 1
        _ = plt.imshow(image)

    plt.show()
```

En la Ilustración 17, se muestra el histograma de las categorías con menos de 500 ejemplos. Se observa un mínimo de 210 imágenes. Parece importante recordar que el mínimo para que una categoría entre a formar parte del conjunto de datos del GTSD debe tener 30 imágenes por pista y 9 pistas, en total 270 imágenes. Las 60 imágenes restantes de las categorías con 210 ejemplos se encuentran en el subconjunto de prueba.

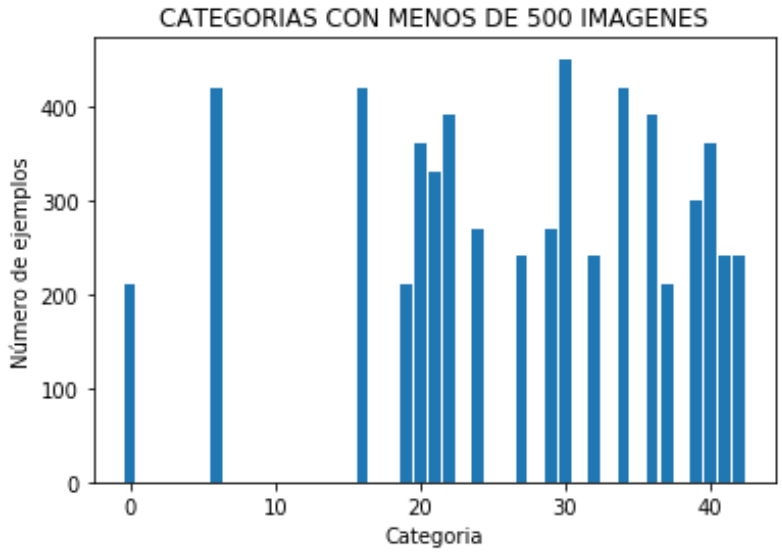


Ilustración 17, histograma categorías < 500

```
OUt[]:
CATEGORIAS CON MENOS DE  500  IMAGENES:  [0, 6, 16, 19, 20, 21, 22, 24, 27, 29, 30, 32,
34, 36, 37, 39, 40, 41, 42]

 HAY UN TOTAL DE  19 CATEGORIAS CON MENOS DE  500 IMAGENES
```

Ilustración 18, categorias < 500

La selección anterior muestra aquellas categorías más susceptibles de tener una baja representación durante la optimización de los pesos de la matriz “W”, lo que generará errores

en su clasificación. Son susceptibles aquellas categorías que tengan alguna similitud con otras categorías, especialmente si una de estas categorías está muy representada.

Deberemos prestar especial atención a la categoría 0, que puede confundirse con cualquier otra señal de restricción de velocidad que tenga mayor número de ejemplos. Las categorías 32, 34, 36, 37 y 39 muestran alguna similitud. Las categorías 41 y 42 también son susceptibles de generar error.

Análisis por categoría

Es necesario realizar un análisis del tipo de imágenes en cada categoría, con el objetivo de tener un mayor conocimiento sobre los datos. En este tipo de análisis se espera encontrar categorías que puedan incluir varios tipos de señales o alguna peculiaridad que pueda genera información beneficiosa para el modelo.

```
En la categoria 30, tenemos  450 ejemplos.
```

Ilustración 19, categoria 30

Vemos que en la categoría 30 las imágenes son muy oscuras incluso llegando a no ser identificables.

Las categorías 0 – 8 (límites de velocidad), ya se ha observado anteriormente que guardan gran parecido. Es importante anotar que la categoría 6, no es una señal de límite de velocidad y que además tiene una muy baja representación y por lo tanto puede ser confundida con otras categorías.

```
En la categoria 6, tenemos  420 ejemplos.
```

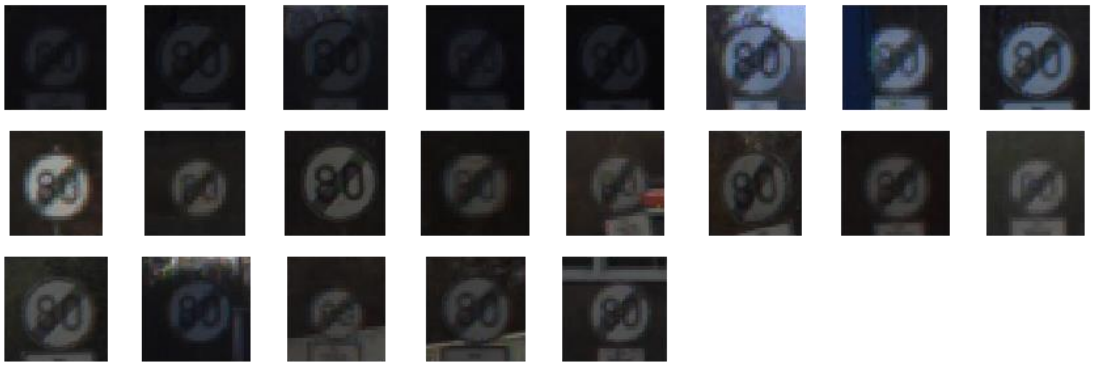


Ilustración 20, categoria 6

3. Preparación de los datos

Previamente al uso de los datos en un modelo, es necesario acondicionar la estructura de los mismos. Los modelos de *Machine Learning*, necesitan que los datos tengan las mismas dimensiones.

Los modelos optimizan la matriz de pesos "W" de una regresión lineal (Y=W*X), donde la imagen es el vector "X". El objetivo del modelo es obtener un vector columna "Y" que contenga un uno en la posición de la categoría a la que pertenece la imagen "X", y 0 en las otras filas. Por lo tanto, el vector columna "Y" tendrá 62 filas (número de categorías). El vector "X" contiene la imagen que hemos importado en un *array* 3D, redistribuido como un vector columna. Así, las dimensiones de la matriz "W" vienen dadas por las de los vectores "X" e "Y". Dado que el objetivo es encontrar una ecuación general que se cumpla para todas las imágenes, las dimensiones de la matriz "W" y de los vectores "X" e "Y" deben ser las mismas para todas las imágenes.

Como ya se ha observado anteriormente, las dimensiones de las imágenes varían mucho y por lo tanto es necesario redimensionar todas las imágenes para que tengan el mismo tamaño.

El tamaño de las imágenes influye directamente en la precisión del modelo, ya que el número de columnas de la matriz "W" es igual al tamaño del vector "X". Dado que este parámetro influye en la precisión del modelo deberemos tener cuidado al elegir el tamaño al que se va a redimensionar todas las imágenes.

Las funciones para acondicionar los datos "prepare_X" y "prepare_Y" se han basada en el articulo de Walled Abdulla [12] y Andrew NG [11].

El tamaño de imágenes que se va a elegir influye sobre el rendimiento del modelo. Siguiendo recomendaciones bibliográficas se va a definir un tamaña de imagen de 32x32.

```
images_shape=[]

for image in X_train_orig:
    images_shape.append(image.shape)

#print(images_shape)

print('El tamaño máximo de nuestras imágenes es: ', max(images_shape))
print('El tamaño mínimo de nuestras imágenes es: ', min(images_shape))

Out[5]:
El tamaño máximo de nuestras imágenes es:  (225, 243, 3)
El tamaño mínimo de nuestras imágenes es:  (25, 25, 3)
```

Se redimensionan las imágenes utilizando la función resize( ). Se observa que la función, además de redimensionar, también normaliza los valores RGB de las imágenes. La normalización de los valores de forma general es muy recomendable para los modelos de Machine Learning.

4. Modelo 1

El primer modelo que se va a implementar es una red neuronal completamente conectada, de una sola capa igual al número de categorías de nuestro conjunto de datos. Este modelo, extremadamente simple, nos sirve como Modelo Mínimo Viable [16], lo que nos permitirá identificar errores en el tratamiento de datos y empezar a implementar una metodología para entrenar el modelo.

Código y resultados

Para construir el modelos utilizaremos TensorFlow lo que nos permite no tener que entrar en la formulación de las matrices y simplemente definir la estructura. Para este caso, se va a ejecutar

el modelo dentro de una clase, con el objetivo de reducir las líneas de código, facilitar su revisión y optimización de hiperparametros. Como base para este modelo hemos utilizado el código de Walled Abdulla [12] adaptado al conjunto de datos con el que se está trabajando.

De forma general, el esquema de los modelos programados en TensorFlow sigue la siguiente estructura:

- *Crear placeholders*
- *Inicializar los parámetros*
- *Forward propagation*
- *Calcular el coste*
- *Optimizar → Adam optimizar*

Para adaptar el código de Walled Abdulla a nuestro conjunto de datos, hemos tenido que modificar el tamaño de las imágenes a 32x32x3 y cambiar la dimensión de la capa completamente conectada.

```
class Model1():
    #***BASIC MODEL WITH 1 FC LAYER***
    Def   init  (self):
        self.graph = tf.Graph()
        with self.graph.as_default():
            #Placeholders
            self.images=tf.placeholder(tf.float32,(None,32, 32,3), name='images')
            self.labels=tf.placeholder(tf.int32, [None], name='labels')
            #Flatten input
            self.images_flat=tf.contrib.layers.flatten(self.images)
            #Fully connected layer
            self.logits=tf.contrib.layers.fully_connected(self.images_flat,43,
tf.nn.relu)
            print(self.logits)
            #Convert one hot vector to label indexes (int)
            self.predicted_labels=tf.argmax(self.logits,1)
            #Loss
            self.loss = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(
                    logits=self.logits, labels=self.labels))
            #Training
            self.train=tf.train.AdamOptimizer(learning_rate=0.001).minimize(self.loss)
            #Initialization
            self.init=tf.global_variables_initializer()
            #Create session
            self.session=tf.Session()
            #Run Initialization
            self.session.run(self.init)
```

Las características de este modelo son las siguientes:

- Método de inicialización matriz de pesos: *Xavier Initilization.*
- Método de inicialización matriz de *bias*: Matriz nula.
- Estructura *Forward Propagation*: 1 capa completamente conectada de 43 nodos.
- Función de activación: *ReLu.*
- Clasificador: *Softmax*.
- Cálculo del coste: *Cross Entropy.*

Para el proceso de entrenamiento, se introduce en cada iteración todo el conjunto de datos en el modelo. Dado que el conjunto de datos es grande, este proceso ralentiza mucho el aprendizaje del modelo.

Para entrenar el modelo, se utiliza la función “train()”. Esta función imprime el coste y la precisión del modelo durante su entrenamiento, también imprime la gráfica final que permite observar la evolución del coste.

Se ha mejorado el código de W.A. añadiendo el *input* imprimir a las funciones de *train()* y *evaluate().* Esta mejora permite decidir si se quiere imprimir las gráficas y el coste. También se ha implementado dos *arrays* que permiten almacenar el coste y la precisión de entrenamiento y comprobación del modelo durante su entrenamiento. De esta forma, se puede identificar si se produce un sobreajuste del modelo al subconjunto de entrenamiento.

:

```
def train(model, images, labels, test_images, test_labels, train_count, imprimir=True):
    #Training loop
    costs_test=[]
    acc_test=[]
    costs_train=[]
    acc_train=[]
    for i in range(train_count):
        if i % 100 == 0:
            loss_test, accuracy_test = evaluate(model, test_images, test_labels, i,
imprimir)
            costs_test.append(loss_test)
            acc_test.append(accuracy_test)
            loss_train, accuracy_train = evaluate(model, images, labels, i, imprimir)
            costs_train.append(loss_train)
            acc_train.append(accuracy_train)
            if imprimir == True:
                print("{:4}, Loss: {:.3f}  Train accuracy: {:.3f}".format(i,loss_train,
accuracy_train))
                print("{:4}, Loss: {:.3f}  Test accuracy: {:.3f}".format(i,loss_test,
accuracy_test))

        model.session.run(model.train,{
            model.images:images,
            model.labels:labels})

    #Final evaluation
    loss_test, accuracy_test=evaluate(model, test_images, test_labels,i, imprimir=True)
    costs_test.append(loss_test)
    acc_test.append(accuracy_test)
    loss_train, accuracy_train = evaluate(model, images, labels, i, imprimir=True)
    costs_train.append(loss_train)
    acc_train.append(accuracy_train)
    print("{:4}, Loss: {:.3f}  Train accuracy: {:.3f}".format(i,loss_train,
accuracy_train))
    print("{:4}, Loss: {:.3f}  Test accuracy: {:.3f}".format(i,loss_test,
accuracy_test))

    if imprimir == True:
        # plot the cost
        plt.plot(np.squeeze(acc_train), label='Train')
        plt.plot(np.squeeze(acc_test), label='Test')
        plt.ylabel('Precisión')
        plt.xlabel('Iteraciones por centenares')
        plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
        plt.show()
        plt.plot(np.squeeze(costs_test), label='Test')
        plt.ylabel('Coste')
        plt.xlabel('iteraciones por centenares')
        plt.show()


def evaluate(model, images, labels, step, imprimir=True):
    #Run predictions against the full test set
    #print(images.shape)
    #print(labels.shape)
    predicted, loss = model.session.run([model.predicted_labels, model.loss],
                                        {model.images: images, model.labels:labels})
    #Calculate accuracy and print
    accuracy=np.sum(labels == predicted)/labels.shape[0]
    #if imprimir == True:
        #print("{:4}, Loss: {:.3f}  Test accuracy: {:.3f}".format(step,loss, accuracy))
    return loss, accuracy


m1 = Model1()
train(m1, X_train_orig, Y_train_orig, X_test_orig, Y_test_orig, 2000 , imprimir=True)
```

```
Out [11]:
Tensor("fully_connected/Relu:0", shape=(?, 43), dtype=float32)
   0, Loss: 3.870  Test accuracy: 0.016
  20, Loss: 3.270  Test accuracy: 0.295
  40, Loss: 3.040  Test accuracy: 0.386
  60, Loss: 2.896  Test accuracy: 0.417
  80, Loss: 2.791  Test accuracy: 0.438
 100, Loss: 2.713  Test accuracy: 0.455
 120, Loss: 2.651  Test accuracy: 0.470
 140, Loss: 2.599  Test accuracy: 0.478
…
1980, Loss: 1.327  Train accuracy: 0.696
1980, Loss: 1.621  Test accuracy: 0.635
1999, Loss: 1.325  Train accuracy: 0.696
1999, Loss: 1.621  Test accuracy: 0.635
```

La ilustración 8 muestra el coste del modelo 1. Se observa que decrece de forma continua y tiende a una asíntota horizontal.

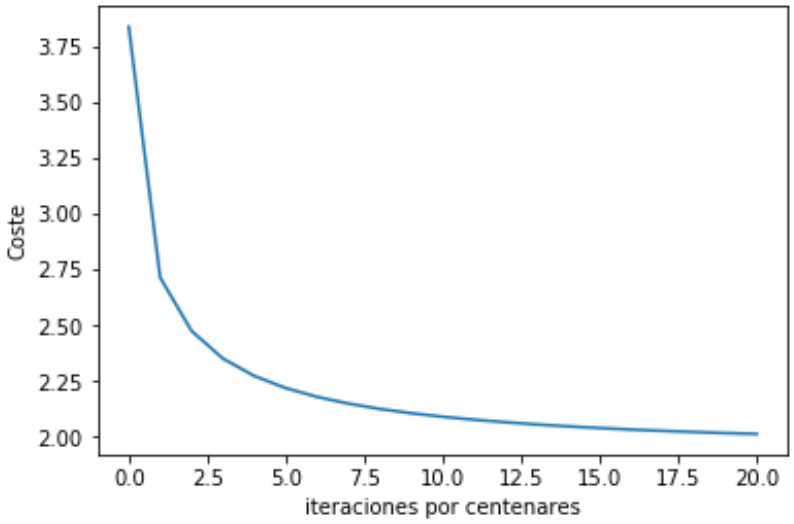


Ilustración 21, coste M1

A continuación, en Ilustración 22 se muestra la curva de precisión del modelo. Se muestra la evolución de la precisión en el subconjunto de entrenamiento y validación.

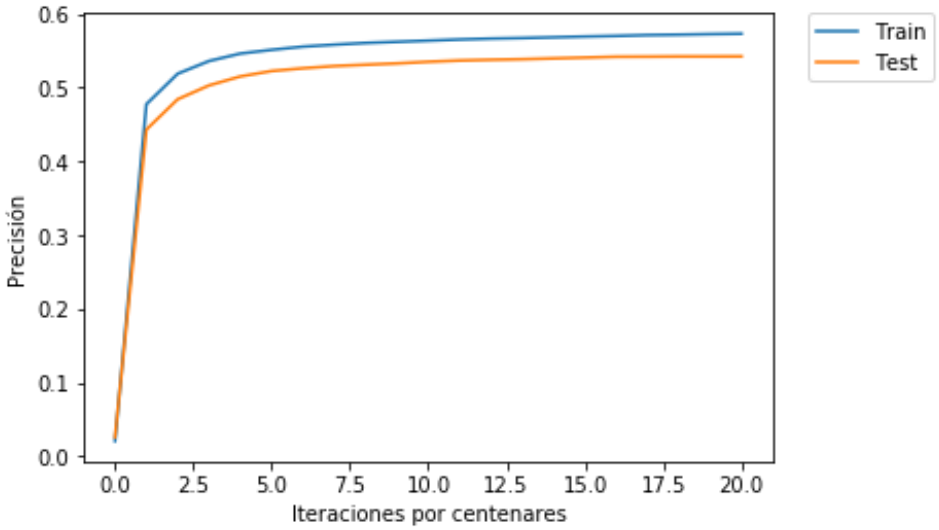


Ilustración 22, precisión M1

La precisión aumenta de forma drástica en las primeras iteraciones y se estabiliza a partir de la iteración 1000.

Dado que es un modelo muy sencillo, la precisión final depende mucho de la inicialización. Se observa que si se ejecuta el modelo varias veces la precisión varía considerablemente, así la precisión final que se obtiene varía entre un 50% y un 60% sobre el subconjunto de datos de prueba. Aunque la precisión no es muy alta, debemos tener en cuenta que este modelo es extremadamente sencillo: solo tiene una capa y las herramientas para el proceso de *Back Propagation* son las más básicas.

En las siguientes líneas de código se muestra la precisión del modelo sobre 5 pruebas totalmente independientes.

```
for i in range(5):
    m1 = Model1()
    train(m1, X_train_orig, Y_train_orig, X_test_orig, Y_test_orig, 200 ,
imprimir=False)

Out:
Tensor("fully_connected/Relu:0", shape=(?, 43), dtype=float32)
 499, Loss: 2.004  Test accuracy: 0.573
Tensor("fully_connected/Relu:0", shape=(?, 43), dtype=float32)
 499, Loss: 1.758  Test accuracy: 0.645
Tensor("fully_connected/Relu:0", shape=(?, 43), dtype=float32)
 499, Loss: 2.141  Test accuracy: 0.530
Tensor("fully_connected/Relu:0", shape=(?, 43), dtype=float32)
 499, Loss: 2.207  Test accuracy: 0.539
Tensor("fully_connected/Relu:0", shape=(?, 43), dtype=float32)
 499, Loss: 1.890  Test accuracy: 0.600
```

Análisis de los resultados

El análisis de los resultados se realiza sobre dos puntos básicos: los pesos obtenidos y la precisión por categoría.

La Tabla 1, muestra los valores de precisión para las 6 pruebas realizadas sobre el modelo 1.

	Precisión entrenamiento en porcentaje	Precisión validación en porcentaje	Diferencia en porcentaje
Prueba 1	69,6	63,5	6,1
Prueba 2	65,7	61	4,7
Prueba 3	62,4	60,4	2
Prueba 4	57,5	53,8	3,7
Prueba 5	70,2	64,3	5,9
Prueba 6	61,6	55,6	6
MEDIA	64,50	59,77	4,73
VARIANZA	24,35	17,87	2,67

Tabla 1. Precisión M1

Para el **modelo 1**, se obtiene una media de **64,5% de acierto** en el subconjunto de entrenamiento y 59,77% de acierto para el subconjunto de validación. A continuación, se muestran valores mínimos y máximos del modelo.

	Máximo	Mínimo	Diferencia

Subconjunto de entrenamiento	70,2	57,5	12,7
Subconjunto de validación	64,3	53,8	10,5

Tabla 2. Valores max y min M1

Se observa **gran dispersión en la precisión** entre las pruebas realizadas para los dos subconjuntos, se obtienen varianzas de 24,35 y 17,87. La diferencia de precisión entre los dos subconjuntos resulta más estable, 2,67 puntos de varianza. La precisión del Modelo 1 varía de forma importante entre pruebas, aunque se mantiene estable la diferencia de precisión entre los dos subconjuntos dentro de cada prueba.

La diferencia de las precisiones medias es de 4,73%. Este valor es relativamente bajo para la simplicidad del modelo y por lo tanto se puede deducir que **no existe un sobre ajuste del modelo al subconjunto de entrenamiento**.

Se llama margen de error evitable a la diferencia de precisiones entre el modelo y la precisión teórica de un humano [17]. Para nuestro conjunto de datos algunos artículos han establecido la precisión humana entre 95 y 99 % aunque se indica que puede ser debido a otros factores [13]. Si comparamos estos valores con la precisión de nuestro modelo obtenemos un margen de error evitable de entre 41,5% y 24,8%. Deducimos entonces que existe mucho margen de mejora con respecto a este modelo.

Los únicos valores no constantes en las 6 pruebas realizadas son los valores iniciales dados a la matriz de pesos.

Los valores de iniciación del modelo se generan según el método de *Xavier Initilization* [18]. Para TensorFlow no es necesario llamar a esta función de forma explícita dado que `tf.contrib.layers.fully_connected()` inicia de forma automática los pesos según el método *Xavier Initilization* para la matriz de pesos y con ceros para la matriz de *bias*.

Deducimos que estos parámetros son los causantes de la volatilidad observada. Sería discutible si la elección de *minibatches* provocaría también esta volatilidad, aunque para este modelo no se realiza la optimización con *minibatches*.

Visualización de los pesos

Una vez entrenado el modelo y dado que solo tiene una capa, resulta interesante visualizar los pesos para ver que imágenes se representan mejor y cuales peor.

Como ya hemos comentado, la matriz "W" de los modelos de *Machine Learning* (Y=W*X) contiene los parámetros a optimizar. Al multiplicar esta matriz por un vector columna que contiene la imagen debemos obtener un vector fila con un 1 en la categoría de la imagen que se está analizando y ceros en las demás filas. Por lo tantom las dimensione de la matriz "W" corresponden con: [Ancho x Alto x 3] para las columnas; y número de categorías para las filas.

Si cada fila contiene los pesos correspondientes a una categoría, se puede extraer la matriz "W" del modelo, implementar una escala de valores y representar en forma de imagen cada una de las filas para ver que se representa en ellas. A continuación, se muestra una fila de la matriz "W".

```
Out[11]:
weights shape: (3072, 43)    min: -1.1024    max: 1.1690
biases shape: (43,)          min: -0.1785    max: 1.4742
[[-0.02577288  0.099213   -0.07724005 ...  0.00369318 -0.03687607, 0.03027744]]
```

Como ya habíamos comentado, la matriz "W" contiene 43 filas con 3072 valores (ya normalizados). En las siguientes líneas de código, seleccionamos la fila de cada categoría y aplicamos una escala de colores en función de sus valores.

```
def normalize(image):
     return (image-image.min())/(image.max()-image.min())

def display_images(images, titles=None, cols=5, interpolation=None):

     titles = titles or [""] * len(images)
     rows = math.ceil(len(images)/(cols))
     height_ratio =1.2*(rows/cols)*(0.5 if type(images[0]) is not np.ndarray else 1)
     plt.figure(figsize=(11,11 * height_ratio))
     i=1
     for image, title in zip(images, titles):
          plt.subplot(rows, cols, i)
          plt.axis("off")
          if type(image) is not np.ndarray:
               image = [normalize(g) for g in image]
               image = np.concatenate(image, axis=1)
          else:
               image = normalize(image)
          plt.title(title, fontsize = 9)
          plt.imshow(image, cmap="Greys_r", interpolation=interpolation)
          i += 1
```

En la siguiente figura se observa los pesos de cada fila de la matriz "W" junto con un ejemplo de esa categoría. Para facilitar el análisis se va a mostrar, entre paréntesis, al lado del número de categoría, el número de imágenes de esa categoría.

```
def display_weights(weights, limit):
    titles= ["Label {}  ({})".format(l,c) for l, c in enumerate(label_counts)]
    images = list(zip(label_exemplars, weights.T.reshape(-1, 32, 32,3)))
    display_images(images[:limit], titles)
```

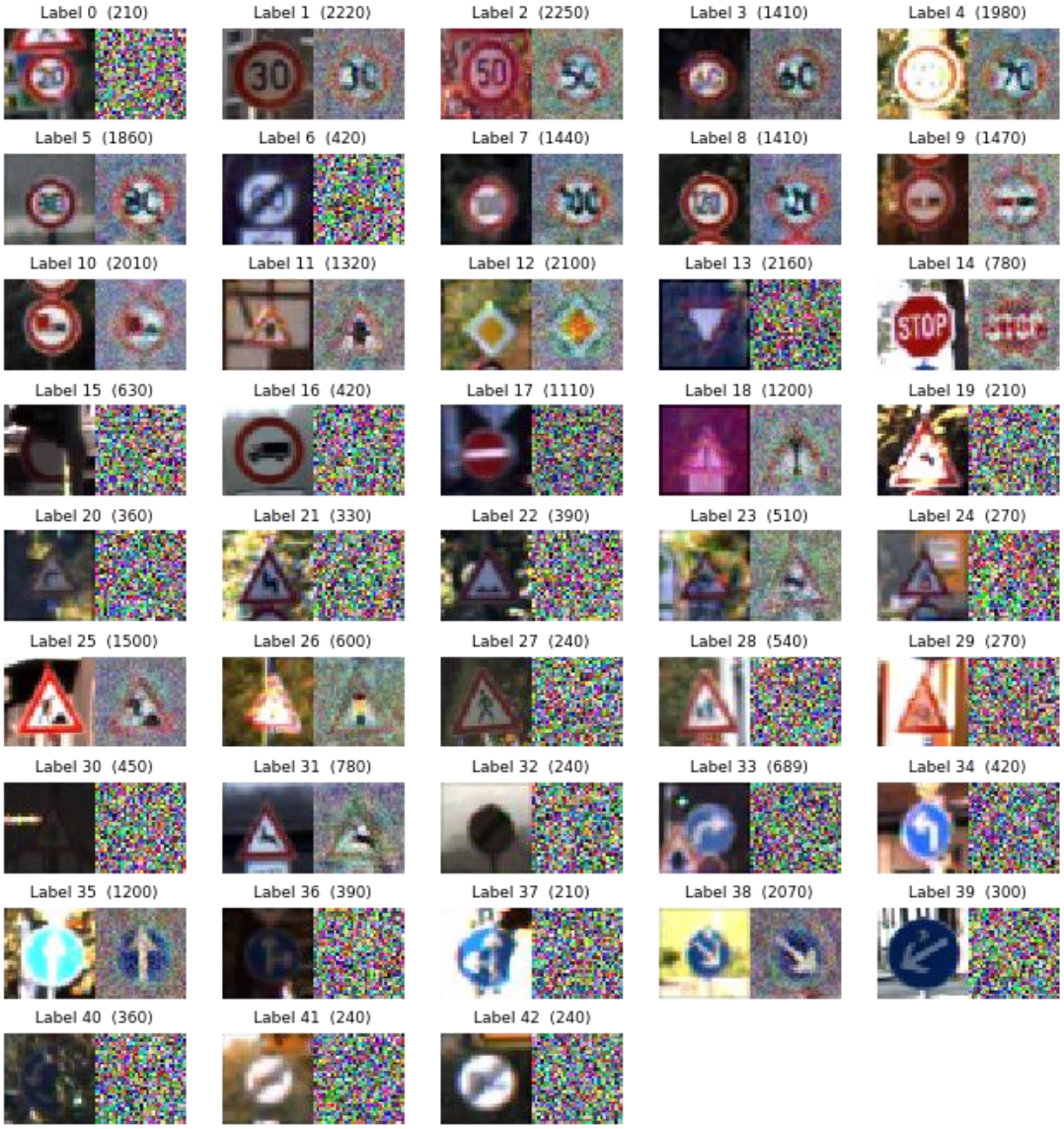


Ilustración 23. Representación pesos M1

Se observa que hay algunas imágenes que guardan cierto parecido con la representación de los pesos.

La representación de los pesos, también se puede realizar con histogramas. Los histogramas son representaciones que nos permiten cuantificar y valorar los pesos de una forma más precisa y por lo tanto más fácil de medir.

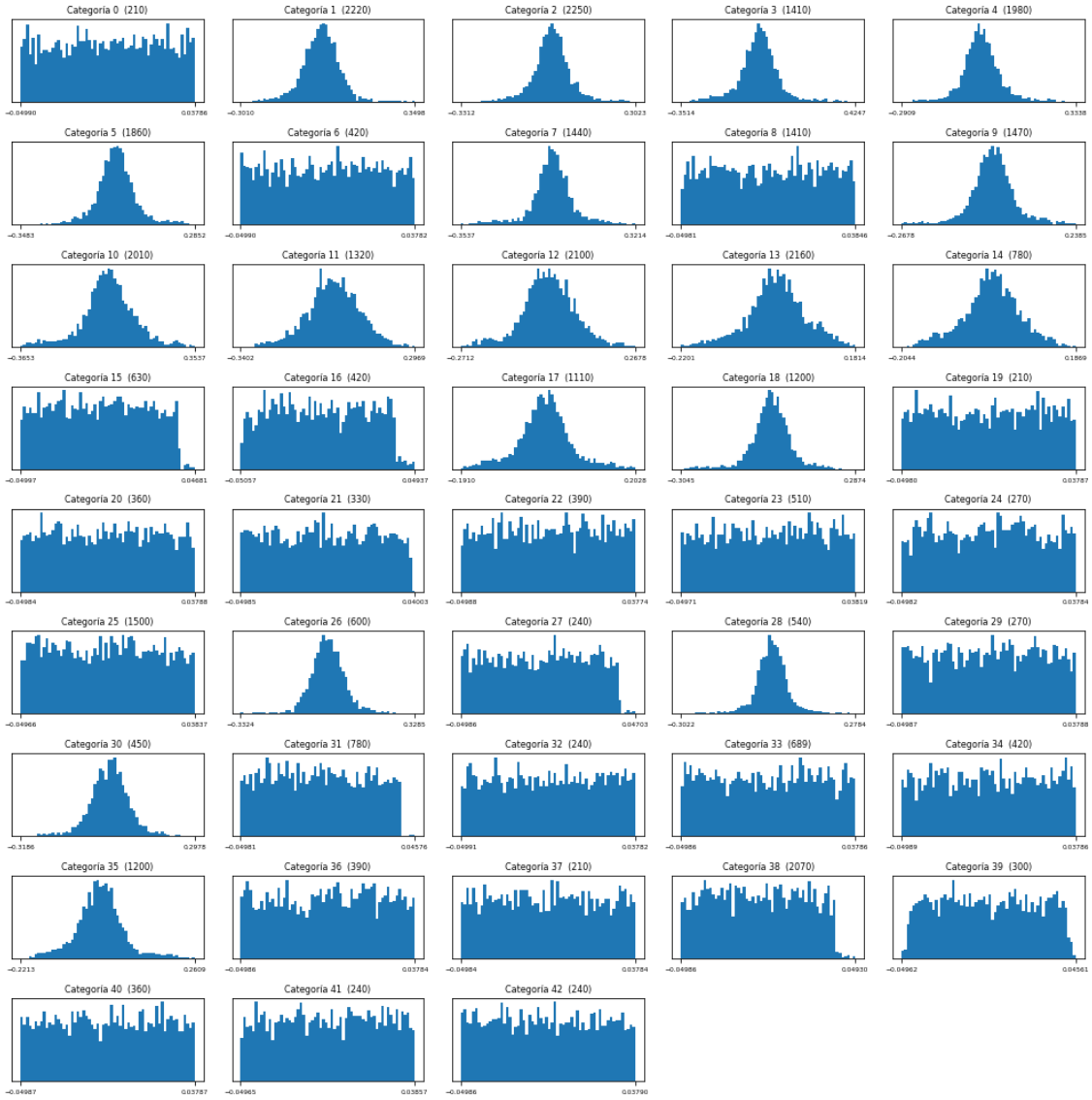


Ilustración 24. Representación histogramas M1

Tanto en la representación en forma de imagen, como en la representación en forma de histograma, se observa el mismo patrón. Los histogramas con una distribución Gaussiana se relacionan con los histogramas con una representación más similar a la imagen. De forma visual, seleccionamos las siguientes categorías como las que mejor representadas están: 1, 2, 3, 4, 5, 7, 8, 9, 10, 11, 12, 14, 18, 23, 25, 26, 31, 35 y 38.

De la representación de histogramas deducimos que hay categorías cuyos pesos responden a una distribución uniforme u aleatoria, en cambio aquellas categorías con representaciones más parecidas a la imagen original tienen un histograma con distribución Guassiana.

Suponemos que las categorías cuya representación de los pesos se asemeja a las imágenes de ejemplo, son las que el modelo representa mejor. Para comprobarlo, se va a realizar un análisis más profundo.

Se observa que, a excepción de las categorías 14 (780), 23 (510), 26 (600) y 31 (780) las demás categorías contienen todas más de 1000 ejemplos. Suponemos que el número de imágenes puede influir en la mejor representación de cada categoría. A continuación, representamos un histograma con el número de ejemplos por categoría, señalando en rojo las categorías

anteriormente mencionadas, donde la representación de los pesos coincide con la imagen de ejemplo de la categoría.

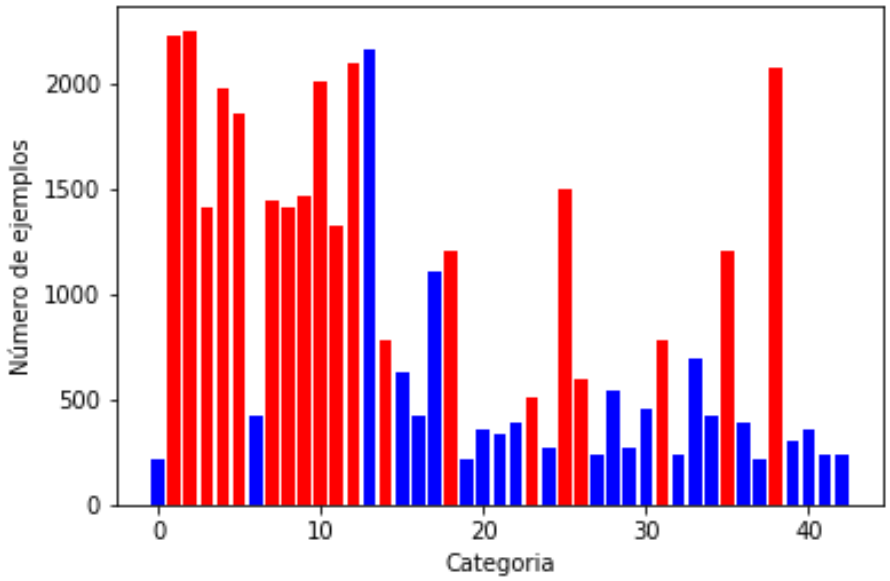


Ilustración 25. Categorías mal representadas M1

Como suponíamos, las categorías seleccionadas en rojo, con una mejor representación de los pesos, tienen de forma general un mayor número de ejemplos en su categoría.

Confussion matrix

Para un análisis más profundo, se va a calcular una matriz de confusión. La función *confusion_matrix()*, representa mediante una matriz las categorías en las que se han clasificado las imágenes. Dado que no todas las imágenes se clasifican correctamente, es necesario realizar un análisis sobre cuáles son las categorías en las que se han clasificado aquellas imágenes mal clasificadas.

En la Ilustración 26 se muestra una representación gráfica de la matriz de confusiones. En abscisas se representan las categorías obtenidas del modelo y en ordenadas las categorías originales. En cada posición, encontramos el número de imágenes pertenecientes a la categoría “y” (en ordenadas) y clasificadas como “x” (en abscisas). Cuando un modelo categoriza todas las imágenes de forma correcta, el número total de imágenes por categoría se encuentra en la diagonal, de tal manera que una matriz de confusión diagonal representa un modelo que ha clasificado todas las imágenes correctamente.

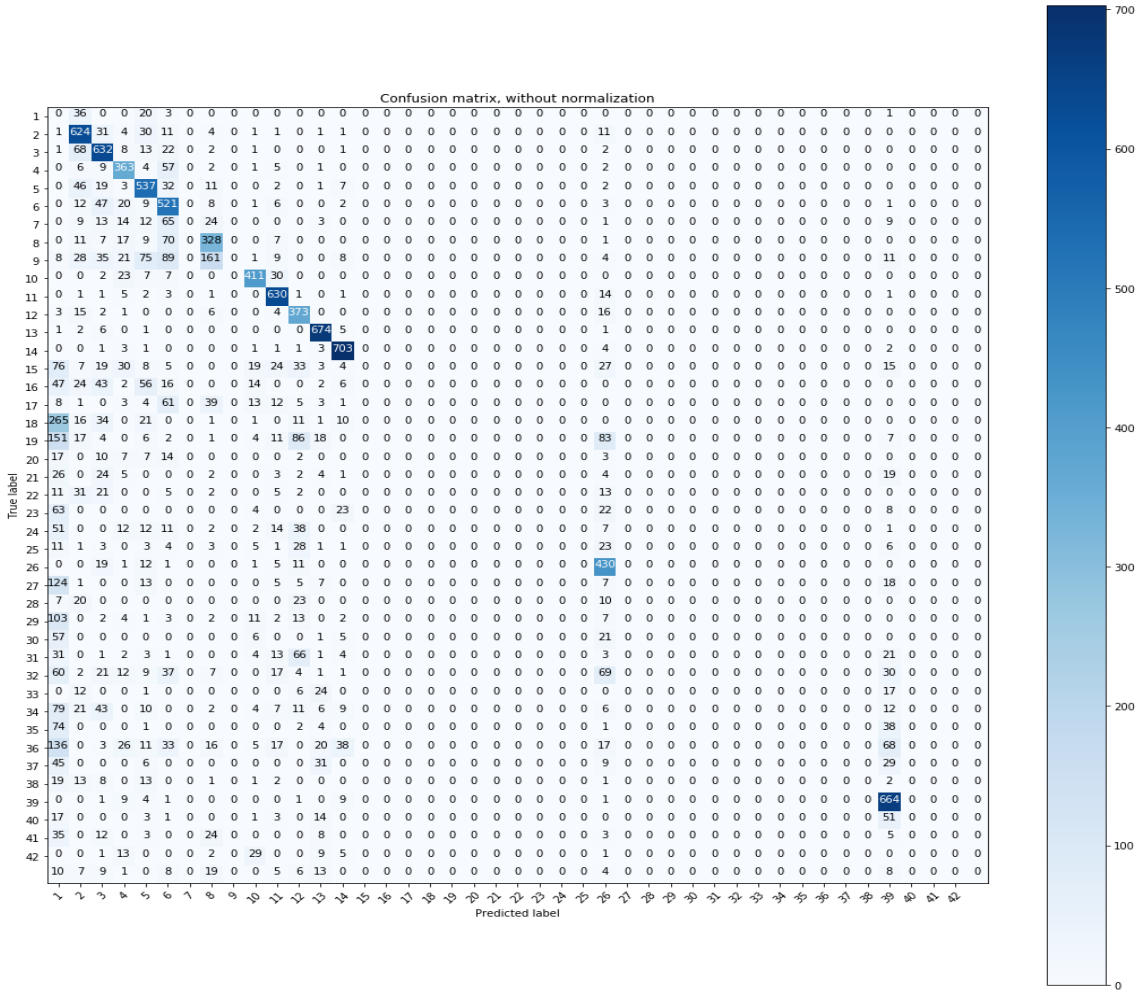


Ilustración 26. Matriz de confusión M1

Todos los valores que no están en la diagonal han sido mal clasificados. Las siguientes categorías no tienen ninguna imagen correctamente clasificada: 1, 6, 9, 15-25; 27-38; 40-43.

La categoría 1 contienen imágenes clasificadas de otras categorías. Se observa como en la columna de la categoría 1 se encuentran imágenes clasificadas que pertenecen a otras categorías: 14, 16, 29, 30 y 36.

Las categorías (en abscisas) 6, 9, 15-25; 27-38; 40-43 no tienen ninguna imagen clasificada, si se observa estas mismas categorías en ordenadas se comprueba que todas las imágenes han sido erróneamente clasificadas en otras categorías. El modelo no ha sido capaz de clasificar ninguna imagen de estas categorías correctamente y tampoco ha clasificado otras imágenes de forma errónea dentro de estas categorías.

Deducimos que el modelo no es suficientemente complejo como para tener en cuenta y representar todas las categorías.

Porcentajes de aciertos

Para poder entender mejor los resultados, es recomendable no solo fijarse en la precisión global del modelo sino también en la precisión del modelo en cada categoría. Existen muchas metodologías para poder visualizar el rendimiento del modelo por categoría, una de ellas es el porcentaje de acierto en cada categoría.

Para visualizar el porcentaje de aciertos, hemos extraído del "Graph" de nuesto modelo en TensoFlow (M1) el vector "Y" de resultados predichos y lo comparamos con el de vector "Y" real.

```
aciertos = [cnf_matrix[i,i]/label_count_test[i]*100 for i in range(43)]
display_aciertos(label_exemplars_test, aciertos)
```

A continuación, se muestran los resultados sobre una imagen a modo de ejemplo de cada señal.

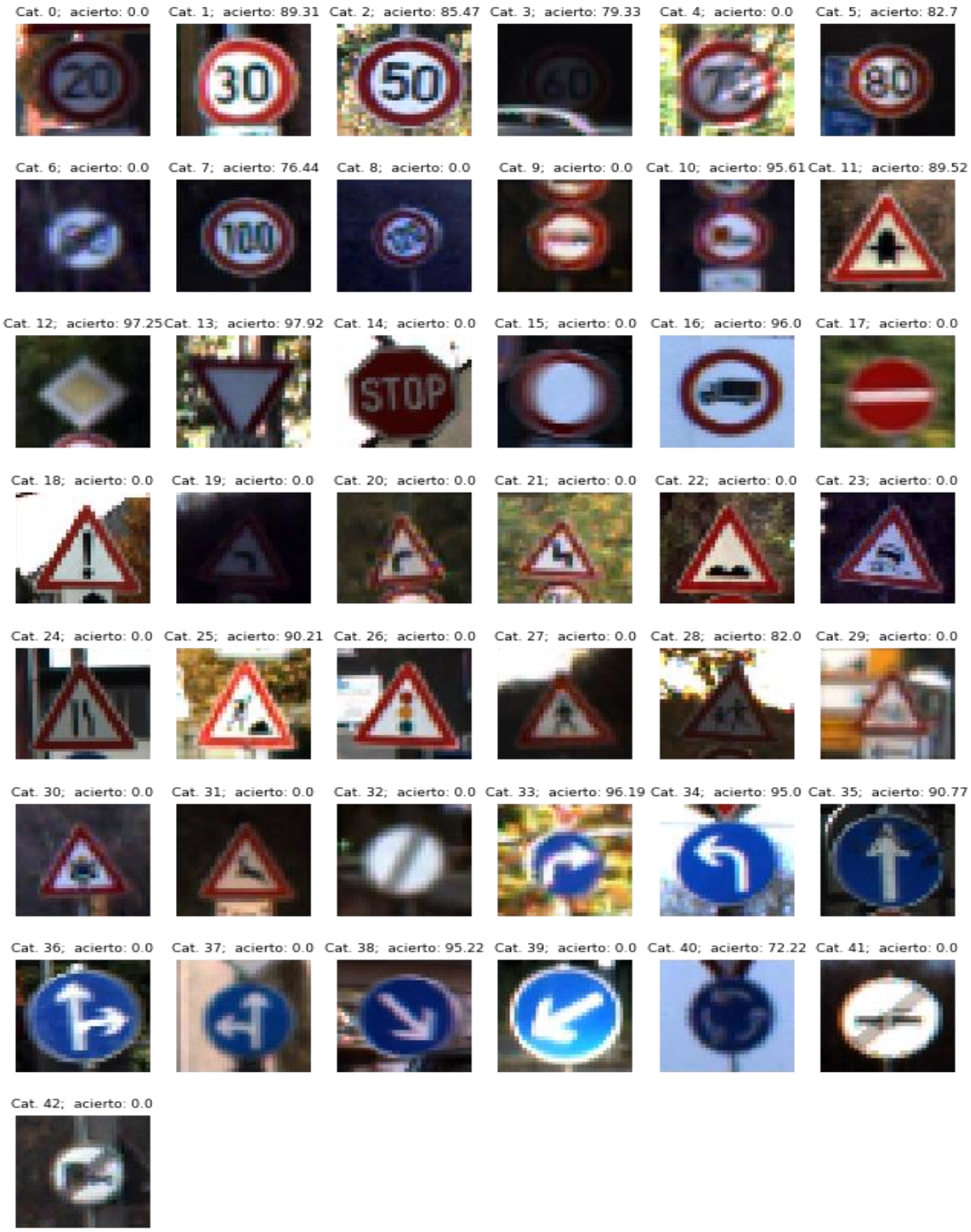


Ilustración 27. Porcentaje de acierto

Se observa, como ya se había comentado en el apartado de la matriz de confusión, que hay algunas categorías que obtienen un 0,0% de acierto. Gracias a la matriz de confusión sabemos que para estas categorías no hay ninguna imagen mal clasificada.

A continuación, se muestra el porcentaje de acierto sobre los histogramas anteriormente representados de los pesos.

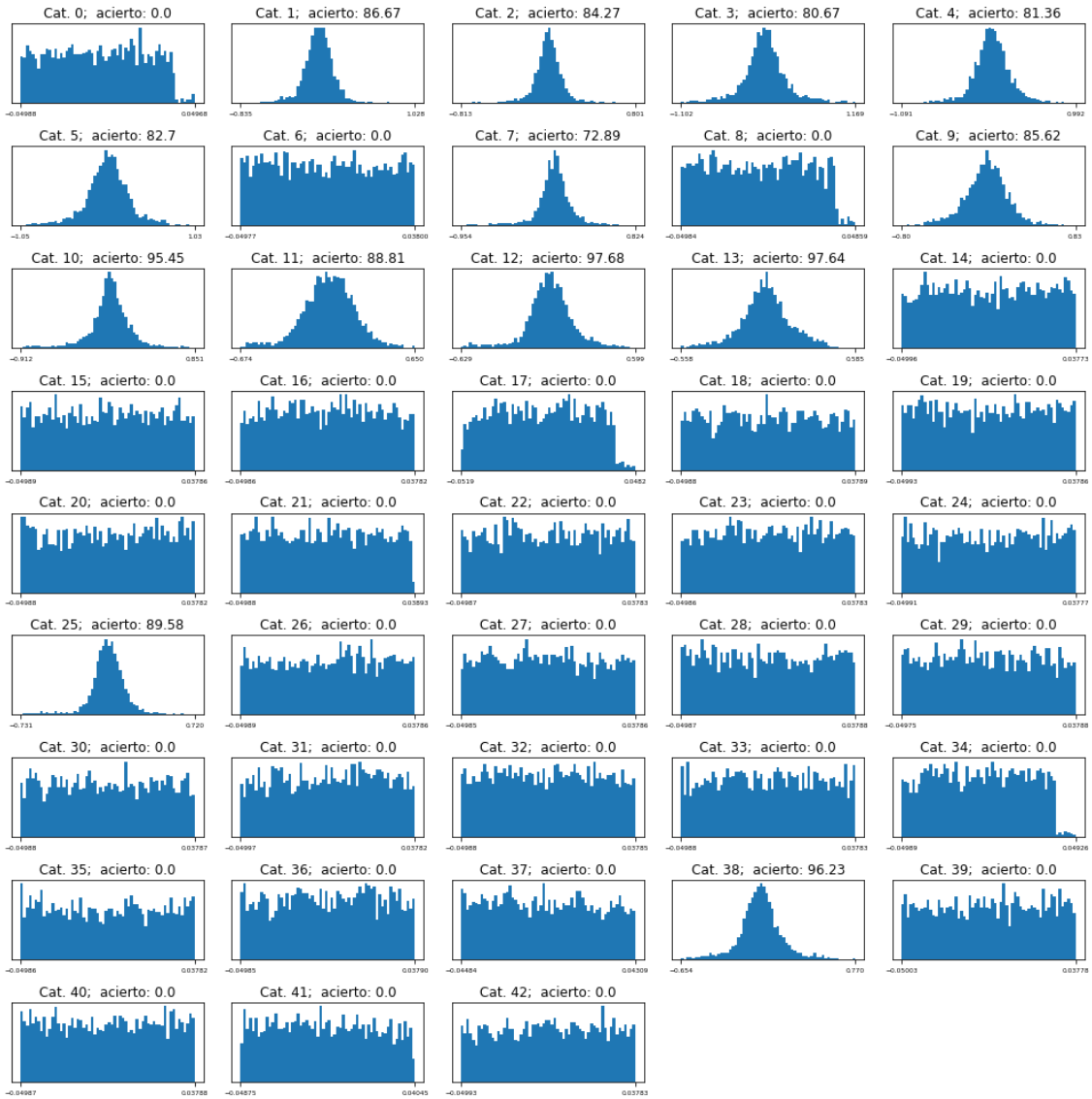


Ilustración 28. Histogramas M1

Volvemos a comprobar que aquellas categorías con un histograma uniforme obtienen un 0,0% de precisión.

A continuación, se muestra el porcentaje de acierto sobre la representación de los pesos.

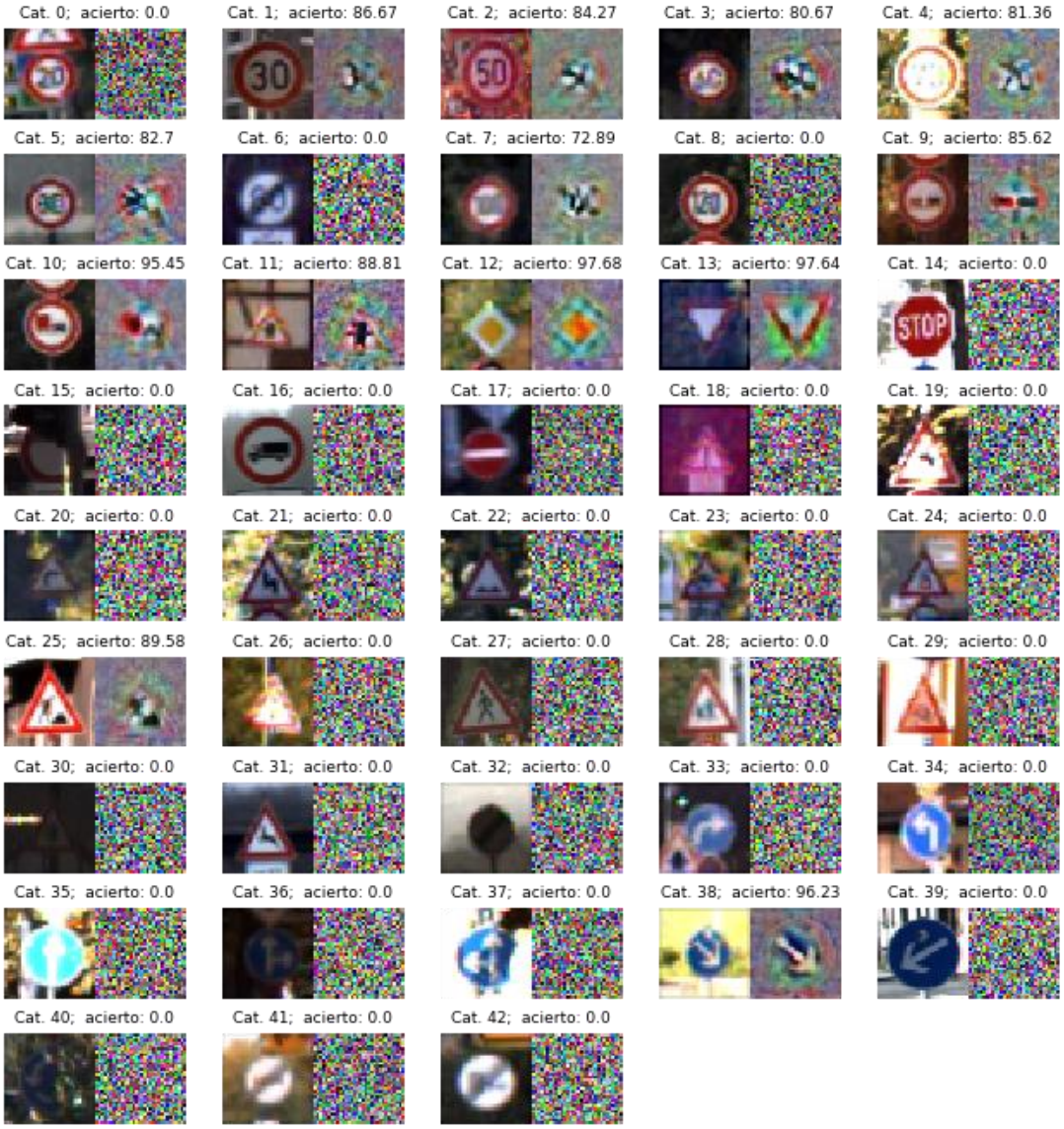


Ilustración 29. Visualización pesos M1

Hemos obtenido 30 categorías con un 0% de acierto.

Conclusiones

Como ya hemos comentado durante el análisis, este modelo sirve a modo de depuración de errores y visualización del conjunto de datos. Sin embargo, hemos obtenido un porcentaje de acierto máximo del 70% sobre 43 categorías.

Con un análisis más profundo, hemos detectado que **no todas las categorías han sido representadas**. De las 43 categorías originales, se observa 30 categorías con un porcentaje de acierto del 0%, de las cuales 29 no han sido tomadas en cuenta por el modelo.

Se ha observado una **relación directa entre el porcentaje de acierto y la cantidad de imágenes**.

Finalmente concluimos que, aunque la precisión global del modelo es bastante mayor de lo esperado para un modelo tan simple, la gran diferencia en cuanto a la cantidad de imágenes entre categorías ha influido muy negativamente en el modelo.

5. Modelo 2

En el segundo modelo, se realiza una mejora del modelo 1 para corregir los errores detectados de infrarrepresentación de algunas categorías.

Se va a implementar dos cambios, en el algoritmo de optimización y en la función de activación. Se va a implementar un algoritmo de optimización tipo *Mini-Batch gradient descent* y una función de activación tipo *Leaky-ReLU*.

Código y resultados

En la siguientes líneas se muestra el código del modelo 2 en detalle.

```
class Model2():

    def   init  (self):
        self.graph = tf.Graph()
        with self.graph.as_default():
            #Global setp counter
            self.global_step = tf.Variable(0, trainable=False,name='global_step')
            #Placeholders
            self.images=tf.placeholder(tf.float32,(None,32, 32,3), name='images')
            self.labels=tf.placeholder(tf.int32, [None], name='labels')
            #Flatten input
            self.images_flat=tf.contrib.layers.flatten(self.images)
            #Fully connected layer
            self.logits=tf.contrib.layers.fully_connected(self.images_flat,62, lrelu)
            #Convert one hot vector to label indexes (int)
            self.predicted_labels=tf.argmax(self.logits,1)
            #Loss
            self.loss = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(
                    logits=self.logits, labels=self.labels))
            #Training
            # Notice that we're passing the gloal_step variable as a parameter.
            # The minimize() function increments it with every training step.
            self.train=tf.train.AdamOptimizer(learning_rate=0.001)\
                        .minimize(self.loss, global_step=self.global_step)
            #Initialization
            self.init=tf.global_variables_initializer()
            #Create session
            self.session=tf.Session()
            #Run Initialization
            self.session.run(self.init)
```

Para implementar la función de activación de Leaky-ReLU hemos tenido que definirla manualmente dado que desconocemos la existencia de alguna función similar en TensorFlow. Para la función de optimización tipo *mini batch*, que se va a implementar, utilizaremos el mismo código que para el modelo 1. Se implementa una selección aleatoria de 50 imágenes previa, para luego introducir esa selección en el modelo. El código utilizado se muestra a continuación.

```
        indexes = np.random.choice(np.arange(images.shape[0]), 50, replace=False)
        model.session.run(model.train,
                          {model.images: images[indexes],
                           model.labels: labels[indexes]})
```

Este tipo de algoritmo afecta a la curva de coste. Dado que el entrenamiento del modelo no se ejecuta con todas las imágenes de una vez, sino que se va a introduciendo 50 imágenes por cada iteración, la curva de coste varía mucho de una iteración a otra. Aunque se pueda observar ruido en la curva, desciende de forma continua y tiende a una asíntota horizontal.

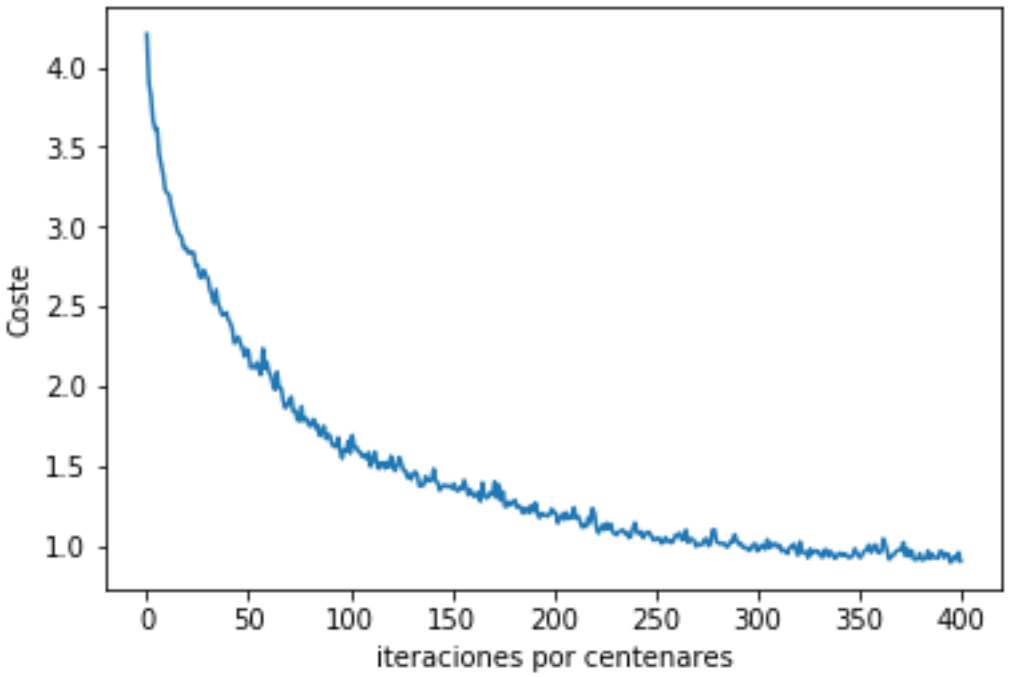


Ilustración 30. Coste M2

Después de 4000, iteraciones se observa en la Ilustración 30, que el coste ya está estabilizado. En la Ilustración 31, se observa la evolución del coste.

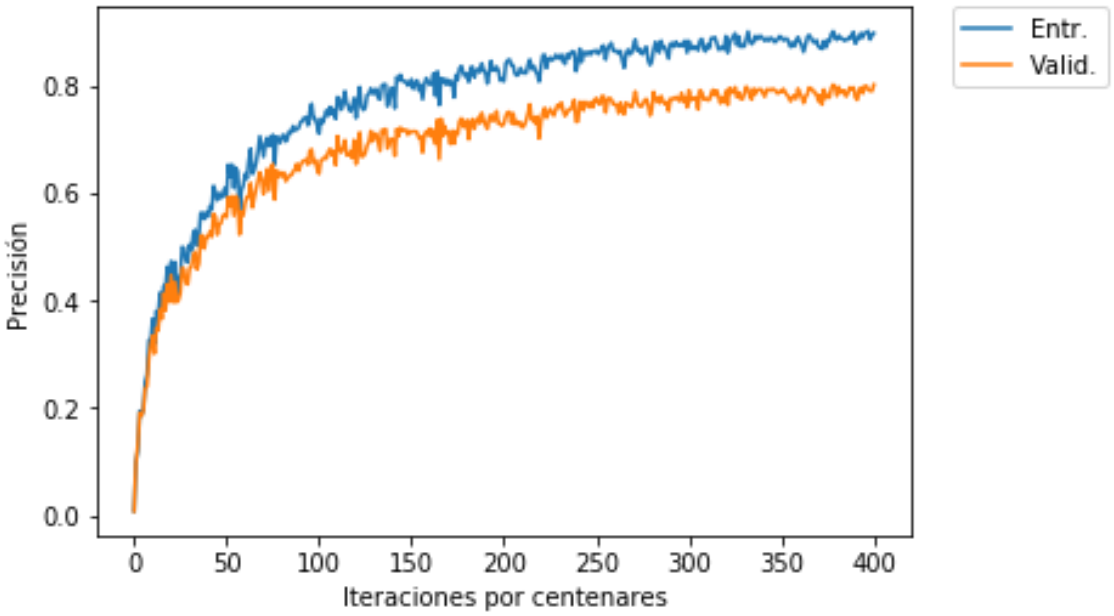


Ilustración 31. Precisión M2

La precisión final del modelo es de 0.895 para el subconjunto de entrenamiento y 0.795 para el subconjunto de prueba.

Se ejecutan 6 pruebas para obtener los valores de precisión y analizar las variaciones de entre ellas.

Se observa que la variación es bastante menor que para el Modelo1: 0.87-0.91 (train set) y 0.78-0.8 (test set).

Análisis de resultados

La Tabla 1, muestra los valores de precisión para las 6 pruebas realizadas sobre el modelo 1.

	MODELO 2		
Columna1	Precisión Ent.	Precisión Val.	Diferencia
ITER. 1	91,90	82,00	9,90
ITER. 2	92,40	82,70	9,70
ITER. 3	91,00	80,50	10,50
ITER. 4	91,10	80,30	10,80
ITER. 5	92,00	81,30	10,70
ITER. 6	90,80	80,20	10,60
MEDIA	91,53	81,17	10,37
VARIANZA	0,42	1,04	0,21

Tabla 3. Resultados M2

Para el modelo 2, se obtiene una media de 91,97% de acierto en el subconjunto de entrenamiento y 81,13% de acierto para el subconjunto de validación. A continuación, se muestran valores mínimos y máximos del modelo.

	Máximo	Mínimo	Diferencia
Subconjunto de entrenamiento	92,4	90,8	1,4
Subconjunto de validación	82,7	80,2	2,6

Tabla 4. Valores max y min M2

En la siguiente tabla se comparan los valores característicos del Modelo 1 y 2.

SUBCONJUNTO	MODELO 1	MODELO 2
Precisión Ent.	64,50	91,53
Precisión Val.	59,77	81,17
Varianza Ent.	24,35	0,42
Varianza Val.	17,87	1,04
Media de la dif.	4,73	10,37
MAX	70,20	92,40
MIN	53,80	80,20

Tabla 5. Comparación modelos

Se observa un aumento significativo de la precisión y una reducción de la dispersión. Sin embargo, la media de las diferencias ha aumentado.

El Modelo 1 obtiene una media en las diferencias entre subconjuntos de 4,73%, con este resultado deducimos que no existe un sobreajuste del Modelo 1 al subconjunto de entrenamiento. Para el Modelo 2 obtenemos una media de las diferencias entre subconjuntos de 10,37%, lo que significa que la diferencia entre la precisión del subconjunto de entrenamiento y validación es mayor. Aunque no es el caso, esta tendencia puede derivar en un sobreajuste del modelo.

Visualización de los pesos

En la siguiente figura se puede ver la representación de los pesos ya descrita anteriormente.

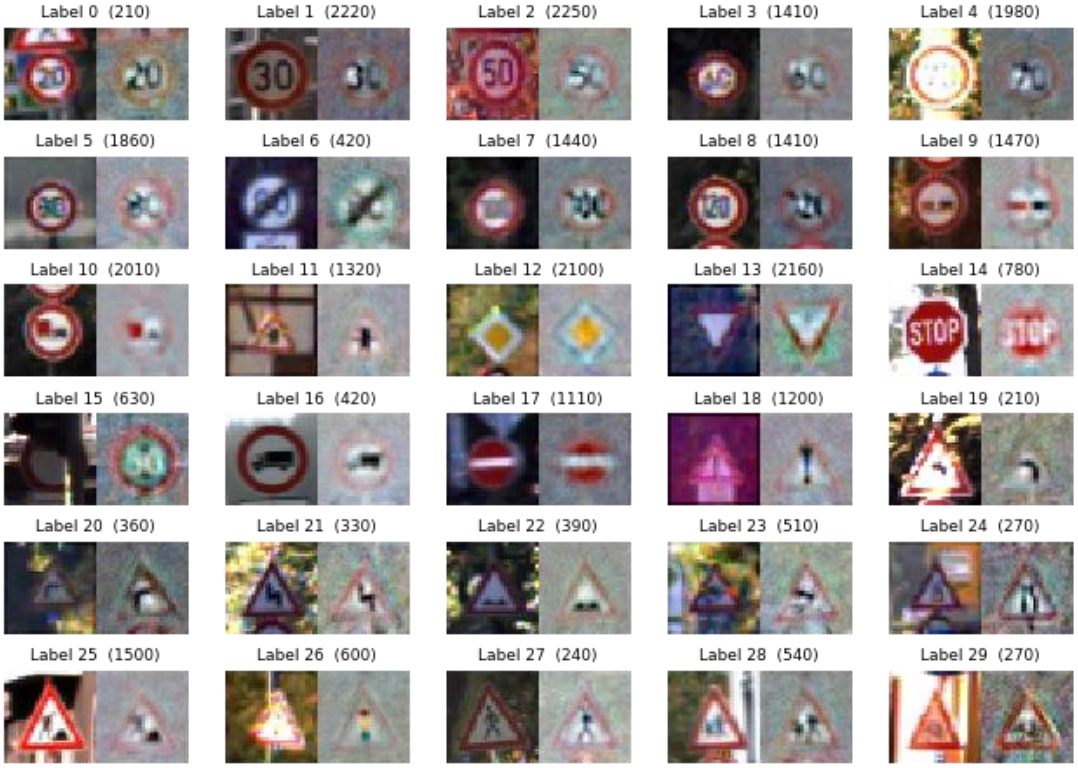


Ilustración 32. Representación pesos M2

En la siguiente figura representamos los histogramas de los pesos.

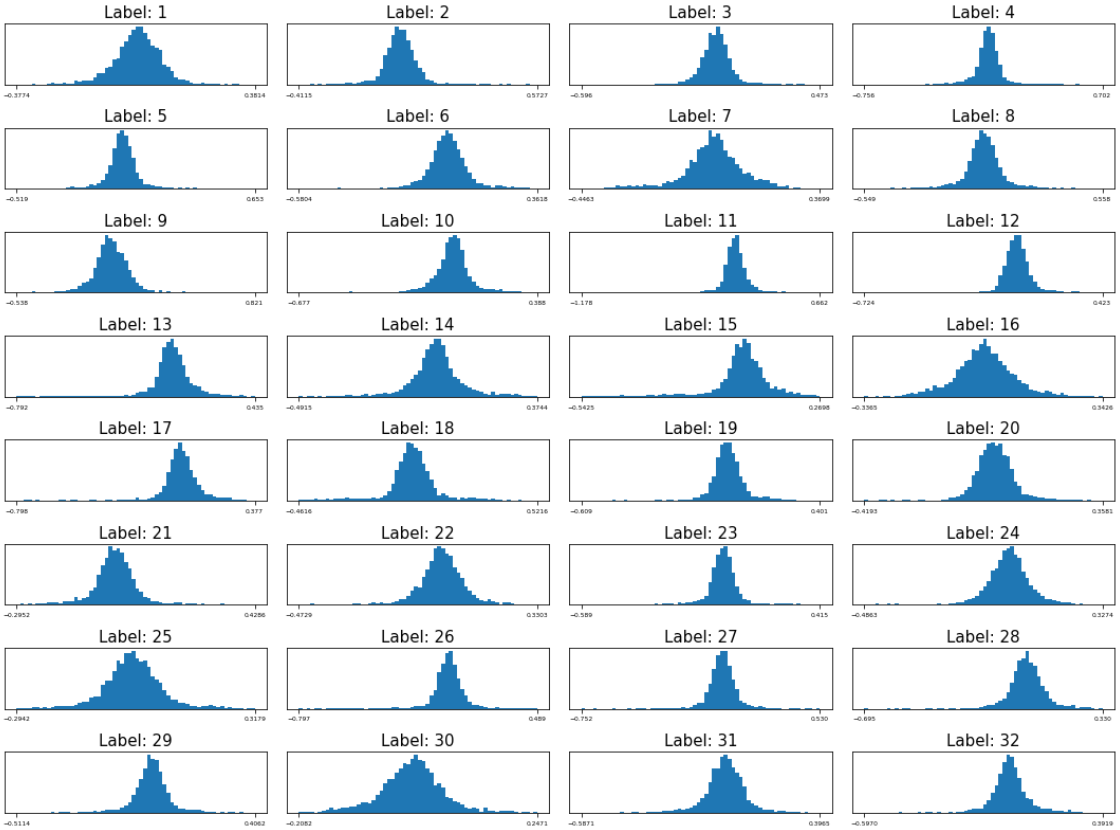
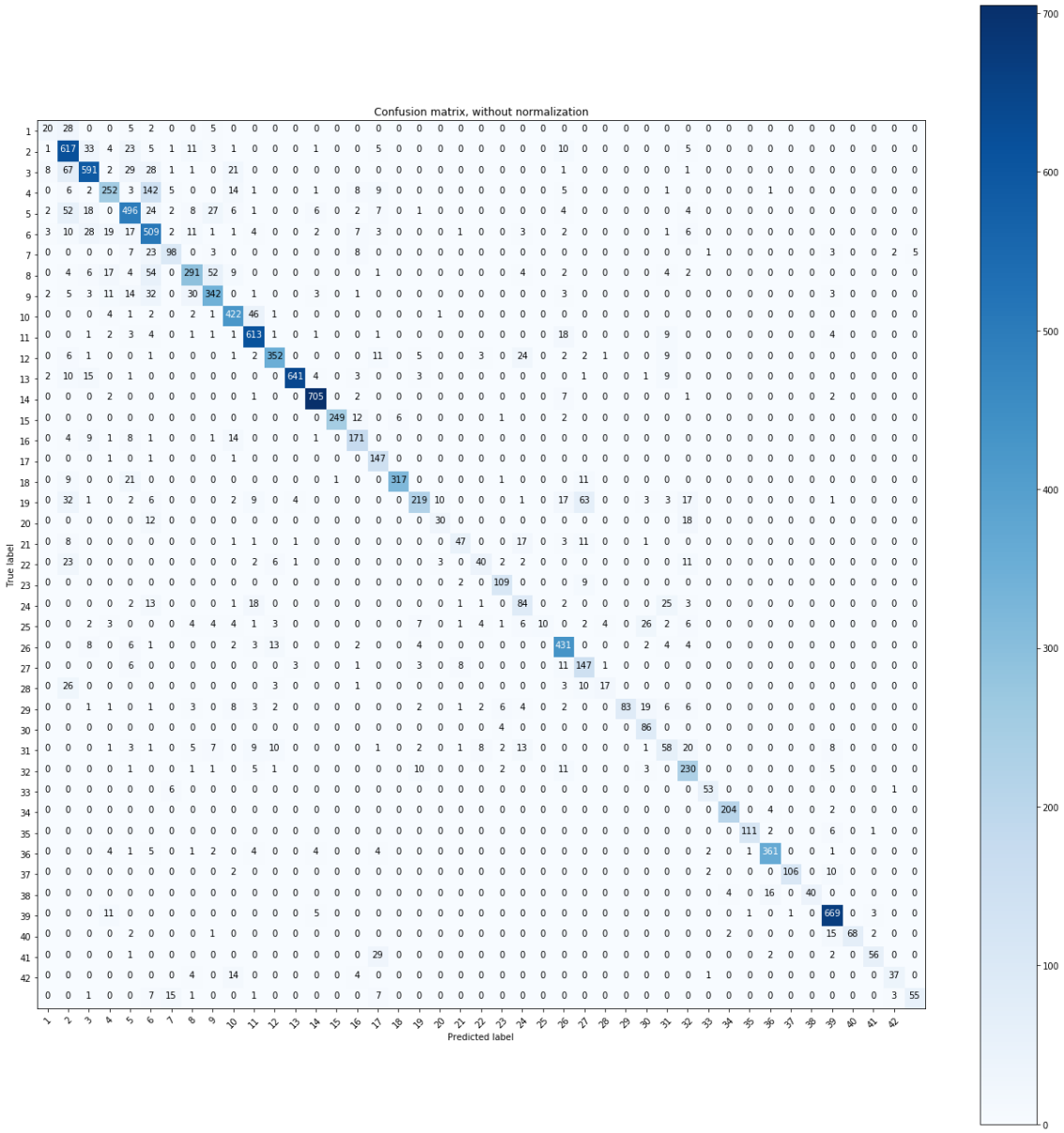


Ilustración 33. Histogramas M2

Como ya habíamos observado en la visualización de los pesos, este modelo se acerca más a la realidad de cada imagen. A diferencia del modelo 1 donde algunas categorías estaban representadas con un histograma uniforme, vemos que en este caso los histogramas son todos Gaussianos lo que quiere decir que son de buena calidad.

Confussion matrix

A continuación se muestra la matriz de confusiones.



Se observa una mayor diagonalidad respecto al Modelo 1. Para este modelo no encontramos ninguna categoría 0 aciertos.

Porcentaje de aciertos

En la siguiente figura se observa el porcentaje de aciertos por categoría.

Ilustración 34. Precisión sobre categoría M2 GTSRB

Conclusiones

De forma general, se observa que el Modelo 2 obtiene una mayor precisión. A diferencia del Modelo 1, no obtenemos ninguna categoría con 0%.

La precisión general del Modelo 1 se basaba en una gran precisión sobre las categorías con mayor número de imágenes, para el Modelo 2 se observa una **homogeneización de las precisiones y una aportación de todas las categorías a la precision global**.

Este modelo nos ha permitido clasificar todas las categorías del subconjunto de prueba. También hemos conseguido mejorar la precisión para las señales de límite de velocidad y obligación de dirección donde antes no conseguíamos buenos resultados.

Aparte del acierto y la precisión también es importante notar que hemos conseguido reducir de forma notable el tiempo de convergencia de los resultados.

6. Modelo 3

Con el segundo modelo, se mejora al máximo la precisión de un modelo de Redes Neuronales con una sola capa. En el Modelo 3 se propone aumentar la precisión aumentando el número de capas de la red neuronal.

Para este caso utilizaremos una estructura con tres capas ocultas y el siguiente esquema: [FC-LRL]x2 + FC – SOFTMAX. Las dos primeras capas serán completamente conectadas (FC) y el tipo de activación será tipo *Leaky-ReLU* (LRL), la última capa será también completamente conectada y con una activación tipo *Softmax*. El tipo de optimización será el mismo que para el modelo 2: *batch gradient descent*.

Según la bibliografía consultada, el tamaño de las capas es un factor decisivo para mejorar el rendimiento del modelo. La optimización del tamaño de las capas se realiza de forma empírica. Se recomienda que las capas decrezcan de forma escalonada y que el tamaño de la última capa coincida con el número de categorías.

Para un primer intento se va a con las siguientes dimensiones para las tres capas: 200, 100 y 43.

```
class Model3():

    def __init__(self):
        self.graph = tf.Graph()
        with self.graph.as_default():
            #Global setp counter
            self.global_step = tf.Variable(0, trainable=False,name='global_step')
            #Placeholders
            self.images=tf.placeholder(tf.float32,(None,32, 32,3), name='images')
            self.labels=tf.placeholder(tf.int32, [None], name='labels')
            #Flatten input
            self.images_flat=tf.contrib.layers.flatten(self.images)
            #Fully connected layer
            self.h1=tf.contrib.layers.fully_connected(self.images_flat,200, lrelu)
            self.h2=tf.contrib.layers.fully_connected(self.h1,100, lrelu)
            self.logits=tf.contrib.layers.fully_connected(self.h2,43, lrelu)
            #Convert one hot vector to label indexes (int)
            self.predicted_labels=tf.argmax(self.logits,1)
            #Loss
            self.loss = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(
                    logits=self.logits, labels=self.labels))
            #Training
            # Notice that we're passing the gloal_step variable as a parameter.
            # The minimize() function increments it with every training step.
            self.train=tf.train.AdamOptimizer(learning_rate=0.001)\
                        .minimize(self.loss, global_step=self.global_step)
            #Initialization
            self.init=tf.global_variables_initializer()
            #Create session
```

```
            self.session=tf.Session()
            #Run Initialization
            self.session.run(self.init)
```

A continuación, se muestra la gráfica del coste.

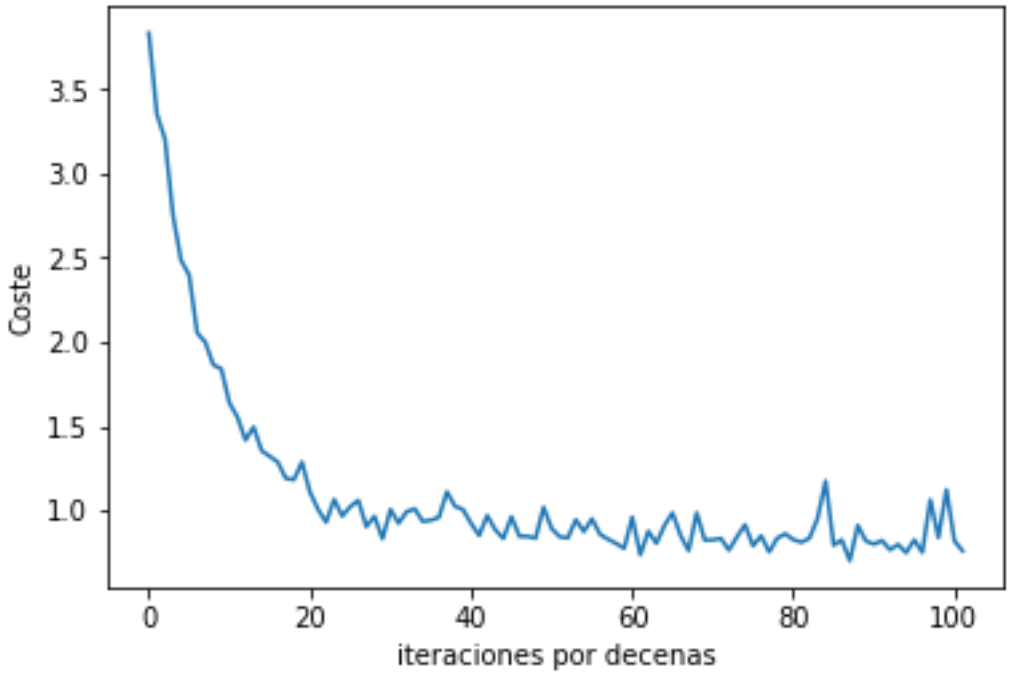


Ilustración 35. Coste M3

A continuación, se muestra la gráfica de precisión del modelo.

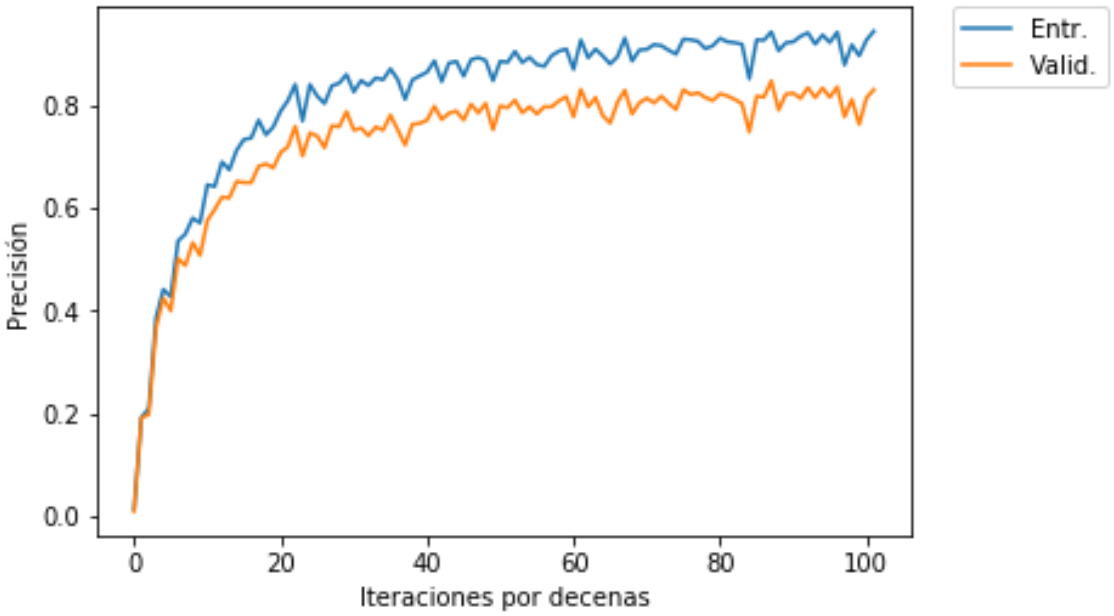


Ilustración 36. Precisión M3

```
Out[21]:
4950, Loss: 0.257  Train accuracy: 0.928
4950, Loss: 0.818  Test accuracy: 0.815
```

Después de 4950 iteraciones, obtenemos una precisión de entrenamiento de 92,8 % y una precisión sobre el subconjunto de validación de 81,5%.

No se observa mejora respecto a la precisión del Modelo 2. Para evaluar el potencial del modelo, se va a realizar un análisis de sensibilidad del tamaño de las capas.

Análisis de sensibilidad

Para realizar un análisis de sensibilidad en este modelo, se va a implementar un pequeño código que nos ejecute el modelo con distintos valores y guarde los resultados en un vector para luego poder representarlo.

Se define de nuevo la clase Model3, pero en este caso con tres inputs: n1, n2 y n3 que corresponden con el tamaño de las capas.

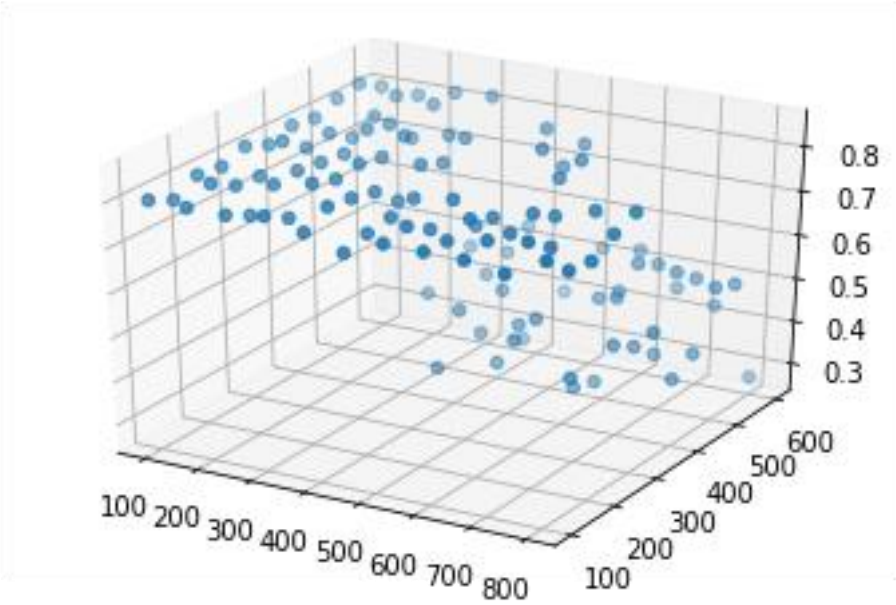
```
class Model3(n1, n2, n3):

    def __init__(self):
        self.graph = tf.Graph()
        with self.graph.as_default():
            #Global setp counter
            self.global_step = tf.Variable(0, trainable=False,name='global_step')
            #Placeholders
            self.images=tf.placeholder(tf.float32,(None,32, 32,3), name='images')
            self.labels=tf.placeholder(tf.int32, [None], name='labels')
            #Flatten input
            self.images_flat=tf.contrib.layers.flatten(self.images)
            #Fully connected layer
            self.h1=tf.contrib.layers.fully_connected(self.images_flat,n1, lrelu)
            self.h2=tf.contrib.layers.fully_connected(self.h1,n2, lrelu)
            self.logits=tf.contrib.layers.fully_connected(self.h2,n3, lrelu)
            #Convert one hot vector to label indexes (int)
            self.predicted_labels=tf.argmax(self.logits,1)
            #Loss
            self.loss = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(
                    logits=self.logits, labels=self.labels))
            #Training
            # Notice that we're passing the gloal_step variable as a parameter.
            # The minimize() function increments it with every training step.
            self.train=tf.train.AdamOptimizer(learning rate=0.001)\
                        .minimize(self.loss, global_step=self.global_step)
            #Initialization
            self.init=tf.global_variables_initializer()
            #Create session
```

Una vez modificada la clase, simplemente tenemos que ejecutar dos bucles que ejecuten el modelo cada vez y rectifiquen los valores de n1 y n2, dado que n3 y el número de categorías son valores fijos.

```
RESULTADOS=[]
for i in range(5):
    n1=500-i*50
    for j in range(5):
        n2=300-j*50
        n3=43
        print("***INTENTO ",i,",",j,"***")
        print("n1: ",n1)
        print("n2: ",n2)
        print("n3: ",n3)
        m3 = Model3(n1,n2,n3)
        loss = train_minibatch(m3, X_train_orig, Y_train_orig, X_test_orig, Y_test_orig,
3000, True)
        y_pred,_ = m3.session.run([m3.predicted_labels, m3.loss],{m3.images:
X_test_orig, m3.labels:Y_test_orig})
        acc=np.sum(Y_test_orig == y_pred)/Y_test_orig.shape[0]
        RESULTADOS.append([n1,n2,n3,acc,y_pred,loss])
```

Si reproducimos el código y representamos los resultados guardados en el array RESULTADOS obtenemos la siguiente figura:



En el eje X (entre 100 y 800) tenemos el tamaño de la primera capa y en el eje Y (entre 100 y 600) tenemos el tamaño de la segunda capa.

Se observa gran sensibilidad del modelo al tamaño de la primera capa y establecemos un rango óptimo entre 100 y 300 nodos. La precisión del modelo no es afectada por el tamaño de la segunda capa.

Siguiendo las recomendaciones bibliográficas, se puede establecer la siguiente hipótesis: el tamaño de las capas deberá ser decreciente. Por lo tanto, si establecemos el rango óptimo entre 300 y 100 nodos para la primera capa, la segunda capa tendrá entre 50 y 200 nodos.

Se observa que la variación de la precisión dentro del rango óptimo seleccionado, es muy pequeño, menor del 5%. Resulta que las variaciones de la precisión debidas a la optimización por *mini batches* es mayor. Por lo tanto, se implementa una función para obtener la mayor precisión del modelo.

Análisis de resultados

A continuación, se muestra la tabla de resultados para las 6 pruebas realizadas en el Modelo 3.

	MODELO 3		
	Precisión Ent.	**Precisión Val.**	**Diferencia**
ITER. 1	92,80	81,50	11,30
ITER. 2	93,00	81,50	11,50
ITER. 3	91,10	79,70	11,40
ITER. 4	91,90	79,80	12,10
ITER. 5	94,00	83,00	11,00
ITER. 6	91,40	80,00	11,40

MEDIA	92,37	80,92	11,45
VARIANZA	1,20	1,72	0,13

Tabla 6. Tabla de precisiones M3

A continuación se muestra una comparativa de los 3 modelos.

SUBCONJUNTO	MODELO 1	MODELO 2	MODELO 3
Precisión Ent.	64,50	91,53	92,37
Precisión Val.	59,77	81,17	80,92
Varianza Ent.	24,35	0,42	1,20
Varianza Val.	17,87	1,04	1,72
Media de la dif.	4,73	10,37	11,45
MAX	70,20	92,40	94,00
MIN	53,80	80,20	79,70

Tabla 7. Comparativa M3

Se observa un ligero aumento en la precisión del subconjunto de entrenamiento del Modelo 3 respecto al Modelo 2. Para el subconjunto de validación la precisión global baja. Obtenemos un aumento de la precisión para el subconjunto de entrenamiento y una disminución para el subconjunto de validación, este incremento en la diferencia de precisiones entre subconjuntos expresa un aumento del sobre ajuste al subconjunto de entrenamiento.

Las varianzas del Modelo 3 aumentan de forma considerable respecto al Modelo 2.

El Modelo 3 es un modelo con una estructura más compleja que el Modelo 2. Con el Modelo 3 hemos obtenido una precisión máxima para el subconjunto de entrenamiento de 94%, superior al 92,4% del Modelo 2, sin embargo, se obtiene unos valores de precisión más dispersos y una varianza mayor.

La mayor complejidad de una red neuronal profunda respecto a una red neuronal superficial, no ha producido el aumento en la precisión y rendimiento del modelo esperado.

Visualización de los pesos

Para las Redes Neuronales de más de una capa, la visualización de los pesos es más complicada dado que las primeras capas se centran, por lo general, en los bordes y esquinas y a medida que la red neuronal es más profunda las capas captan otros detalles más precisos. En la siguiente figura se representan los pesos de la primera capa del Modelo 3.

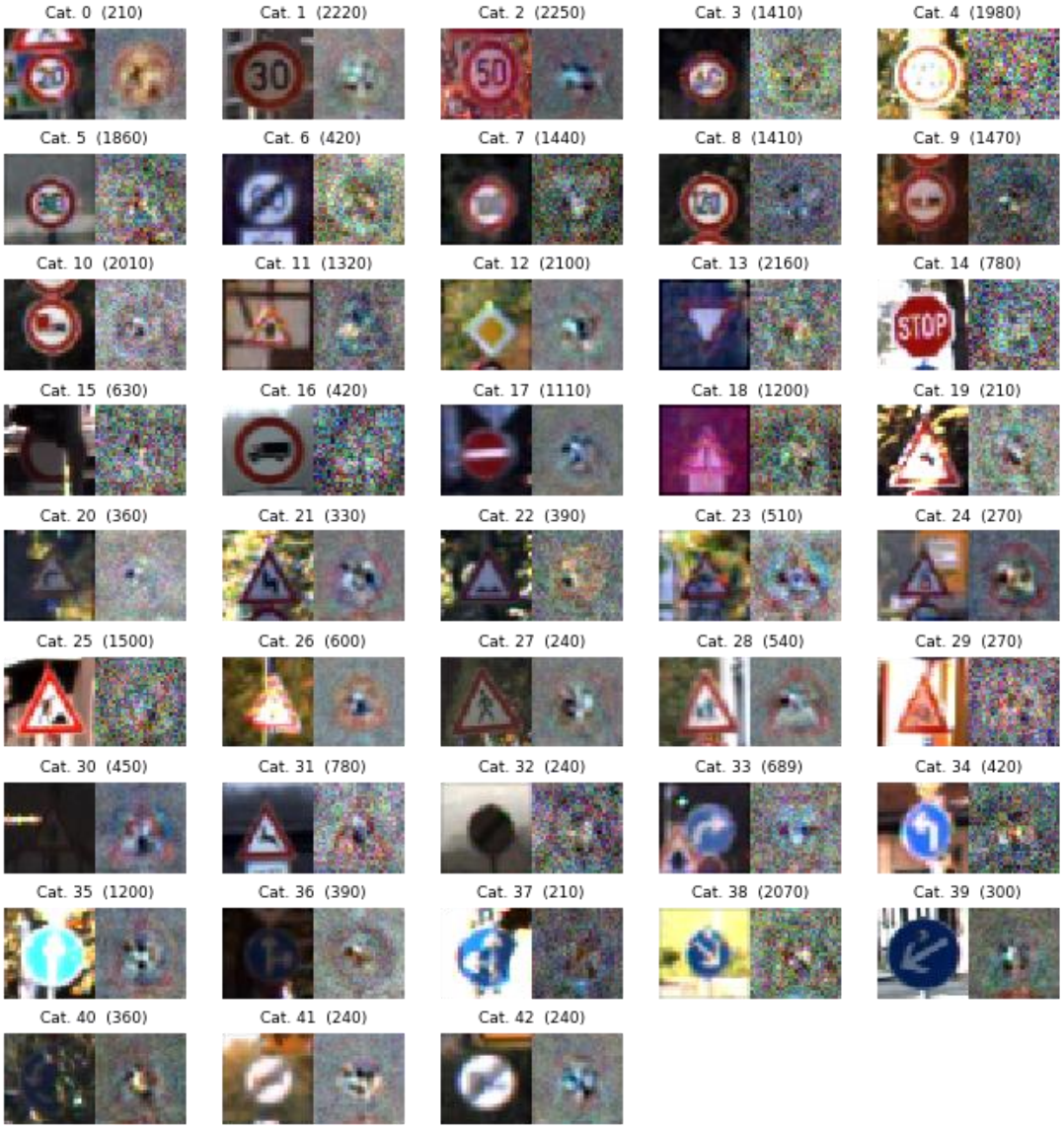


Ilustración 37. Visualización imagenes M3

En la visualización de los pesos, se observa una menor calidad que en Modelo 2. Como ya se ha explicado, esto se debe a que la información se reparte entre las tres capas del modelo y no solo en la primera.

Los histogramas nos confirman las observaciones ya mencionadas, se puede comprobar como estos son más anchos afectando a un mayor número de pixeles.

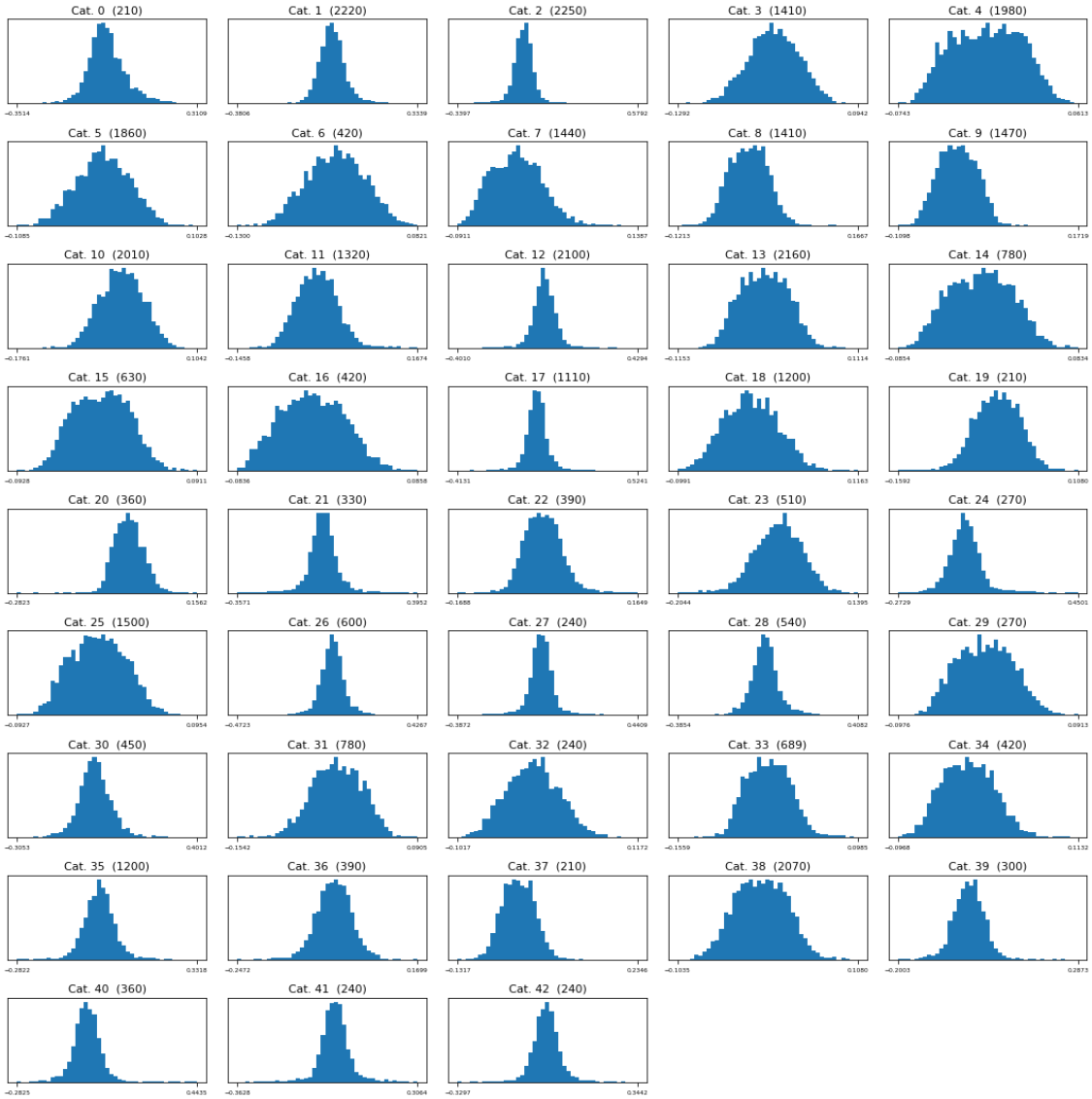


Ilustración 38. Histogramas pesos M3.

Matriz de confusiones

La matriz de confusiones es la siguiente:

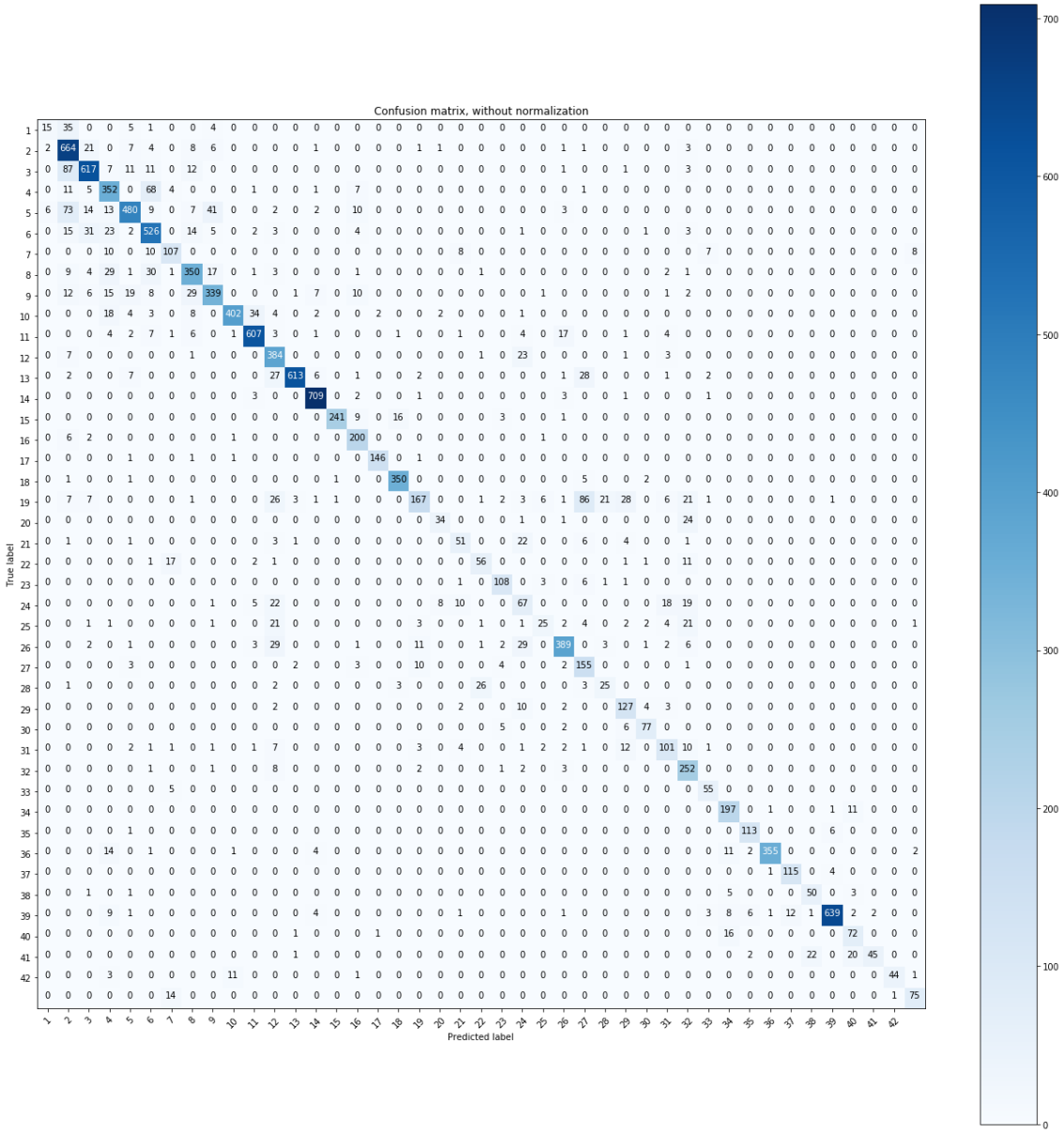


Ilustración 39. Matriz de confusión M3

Se observa de forma general, que la matriz es más diagonal que en el modelo 1 y 2.

Porcentaje de aciertos

A continuación, se muestran los porcentajes de aciertos:

Ilustración 40. Porcentaje de acierto M3

En el porcentaje de acierto de cada categoría se puede observar como la categoría con el menor acierto, ha mejorado respecto al modelo 2. En el modelo 2 teníamos un mínimo de acierto del 10%, con este nuevo modelo hemos conseguido un 28% de acierto en la peor categoría.

Se observa como la mejoría en el porcentaje de aciertos por categoría es importante. De forma general, la precisión global no ha mejorado sustancialmente, pero si lo ha hecho la precisión local en las categorías menos representadas.

Conclusiones

En este nuevo modelo la precisión global obtenida es de 92,37%, se consigue mejorar muy poco la precisión respecto al modelo 2 que era del 91%. Sin embargo, se consigue una ligera **mejoría en la precisión por categorías.**

Las Redes Neuronales son sistemas que requieren de muchos datos, a mayor complejidad de estas, mayor cantidad de datos es necesario. Por lo general la bibliografía consultada obtiene mejores resultados con Redes Neuronales más profundas, y generalmente la problemática reside en evitar un sobre ajuste del modelo o incluso en aumentar el conjunto de datos.

Se ha realizado un **análisis de sensibilidad** sobre el tamaño de las capas y hemos obtenido **muy poca sensibilidad en la primera capa y ninguna en la segunda capa**. Se deduce que un modelo de estas características resulta muy básico para la complejidad del problema, sobre todo si es necesario obtener mejores resultados.

Con la intención de seguir mejorando la precisión de nuestro modelo, aumentaremos la complejidad del modelo e implantaremos una Red Neuronal Convolucional de cuatro capas.

7. Modelo 4

Hasta el momento, hemos realizado 3 modelos de Redes Neuronales, dos con una sola capa y una con tres capas. El siguiente paso para el reconocimiento de imágenes son las Redes Neuronales Convolucionales. Este tipo de red neuronal se distingue respecto a las Redes Neuronales totalmente conectadas en que no todos los nodos de una capa están conectados con los nodos de la siguiente.

La primera capa constará de 8 filtros tipo convolucionales de un tamaño de 5x5 con un paso de un pixel, un padding tipo ‘same’. Al final de esta capa añadiremos una capa de Maxpool de 2x2 y un paso de 2 pixels. Para esta primera capa el número de pesos será 5*5*3*8 = 600.

La segundo capa constará de 12 filtros tipo convolucionales de 5x5, con paso de 1 pixel y padding tipo ‘same’. Aplicaremos una capa tipo max pool de 2x2 con un paso de 2. El número de pesos en este caso serán 2400.

La tercera capa tendrá 13 filtros tipo convolucionales de 5x5, con paso de 1 pixel y padding tipo ‘same’. Aplicaremos una capa pool max de 2x2 son paso de 2 pixeles. El número de pesos en este caso serán 4800.

La última capa estará completamente conectada y tendrá 43 nodos. Está última capa tendrá 1256 pesos y finalmente utilizaremos un algoritmo de optimización tipo Softmax.

Dado que esta red neuronal es bastante más compleja que todas las demás, cambiaremos ligeramente el código para poder contabilizar el tiempo de entrenamiento y poder visualizar de forma continua la evolución del coste.

```
class Model4():
    def conv(self, input, num_outputs, name=None):
```

```
        return layers.convolution2d(
            input, num_outputs=num_outputs, kernel_size=(5,5), stride=(1,1),
            padding='SAME', activation_fn=lrelu, normalizer_fn=layers.batch_norm
            )
    def pool(self, input):
        return layers.max_pool2d(input, kernel_size=(2,2),
                                 stride=(2,2), padding='SAME')
    def __init__(self):
        #History of training state as tuples (step, loss, accuracy, training loss, time)
        self.train_log =[]
        self.train_time=[]

        self.graph = tf.Graph()
        with self.graph.as_default():
            #Global step counter
            self.global_step = tf.Variable(0, trainable=False, name='global_step')
            #Placeholders
            self.images=tf.placeholder(tf.float32, (None, 32,32,3), name='images')
            self.labels=tf.placeholder(tf.int32, (None), name='labels')
            #Block input shape [32,32,3], output shape [16,16,16]
            self.conv1 = self.conv(self.images,8)
            self.pool1=self.pool(self.conv1)
            #Block input shape: [16,16,16], output shape [8,8,32]
            self.conv2=self.conv(self.pool1,12)
            self.pool2=self.pool(self.conv2)
            #Block input shape: [8,8,32], ouutput shpae [4,4,64]
            self.conv3=self.conv(self.pool2,16)
            self.pool3=self.pool(self.conv3)
            #Fully connected layer
            self.flat=layers.flatten(self.pool3)
#            self.hl=layers_fully:connected(sefl.flat,200,lrelu)
            #self.shapeflat=self.flat.shape
            self.logits = layers.fully_connected(self.flat,62,lrelu)
            #Convert one hot vector to label index
            self.predicted labels=tf.argmax(self.logits,1)
            #Loss
            self.loss = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(
                    logits=self.logits, labels=self.labels))
            #Training OP
            self.train=tf.train.AdamOptimizer(learning_rate=0.001)\
                                .minimize(self.loss, global_step=self.global_step)
            #Initialization Op
            self.init=tf.global_variables_initializer()
            #Create session
            self.session=tf.Session()
            #Run initialization Op
            self.session.run(self.init)
```

Las funciones de evaluación, entrenamiento y gráfico se muestran a continuación.

```
def train_graph(model, train_images, train_labels, val_images, Val_labels, train_count,
imprimir=True):
    t_start=time.time()
    #Training loop
    for i in range(1, train_count+1):
        ##??? improve picking batches
        #COGEMOS UN PEQUEÑO BATCH DE 100 IMAGENES RANDOM
        indexes = np.random.choice(np.arange(train_images.shape[0]),100,replace=False)
        batch_images=train_images[indexes]
        batch_labels=train_labels[indexes]
        #ENTRENAMOS ESTE BATCH CON NUESTRO MODELO
        _, loss = model.session.run([model.train, model.loss], {
            model.images:batch_images,
            model.labels:batch_labels,
            })
        # Evaluate
        if i % 50 == 0:
            evaluate_graph(model, batch_images, batch_labels, val_images, Val_labels,
t_start)
    #Final evalution
    evaluate_graph(model, batch_images, batch_labels, val_images, Val_labels, t_start,
imprimir=True)
    #shape = model.session.run([model.shapeflat], {
    #        model.images:batch_images,
    #        model.labels:batch_labels,
    #        })
```

```
    #print(shape)
def evaluate_graph(model, train_images, train_labels, val_images, val_labels,
training_time, imprimir=False):
    #Run predicionts against a batch of the training set
    train_predicted, train_loss, step = model.session.run(
        [model.predicted_labels, model.loss, model.global_step],
        {model.images:train_images, model.labels:train_labels})

    #Run predictions against the full test set
    val_predicted, val_loss = model.session.run(
            [model.predicted_labels, model.loss],
            {model.images: val_images, model.labels:val_labels})

    #Calculate accuracy
    train_accuracy=np.sum(train_labels == train_predicted) / train_labels.shape[0]
    val_accuracy = np.sum(val_labels == val_predicted) / val_labels.shape[0]
    #Append to train log
    model.train_log.append((step, train_loss, train_accuracy, val_loss, val_accuracy,
training_time))
    #Plot
    draw_graph(model.train_log)

    if imprimir == True:
        print("{:4}, Loss: {:.3f}  Train set accuracy: {:.3f}".format(step,train loss,
train_accuracy))
        print("{:4}, Loss: {:.3f}  Validation set accuracy: {:.3f}".format(step
,val_loss, val_accuracy))

def draw_graph(logs):
    #Expand log tuples to lists
    steps, train_losses, train_accuracies, val_losses, val_accuracies, times =
zip(*logs)
    #Clear output
    display.clear_output(wait=True)
    fig, (ax1,ax2) = plt.subplots(2,sharex=True,figsize=(8,6))
    #Graph 1: Accuracies
    ax1.set_title("Step: {}     Training Time:  {:.0f}  seconds\n"\
                  "Training Accuracy: {:.3f}    Validation Accuracy: {:.3f}"\
                  .format(steps[-1], times[-1], train_accuracies[-1], val_accuracies[-
1]),
                  fontsize=5)
    ax1.plot(steps, train_accuracies, label="Training Accuray")
    ax1.plot(steps, val_accuracies, label="Validation Accuracy")
    ax1.set_ylabel("Accuracy")
    ax1.legend(fontsize=8, loc="lower right")
#    ax1.set_y1(0,1.1)
    #Graph 2 : Losses
    ax2.set_title("Training loss: {:.3f} Validation Loss: {:.3}"\
                 .format(train_losses[-1],val_losses[-1]),fontsize=5)
    ax2.set_yscale('log')
    ax2.plot(steps, train_losses, label="Training loss")
    ax2.plot(steps,val_losses, label="Validation Loss")
    ax2.set_ylabel("Loss")
    ax2.legend(fontsize=8, loc='lower left')
    ax2.set_xlabel("Steps")
    _ = plt.show()
```

Después de 2000 iteraciones obtenemos una precisión sobre el subconjunto de entrenamiento de 0.98 y de 0.91 sobre el subconjunto de validación. Las gráficas de precisión y coste se muestran a continuación.

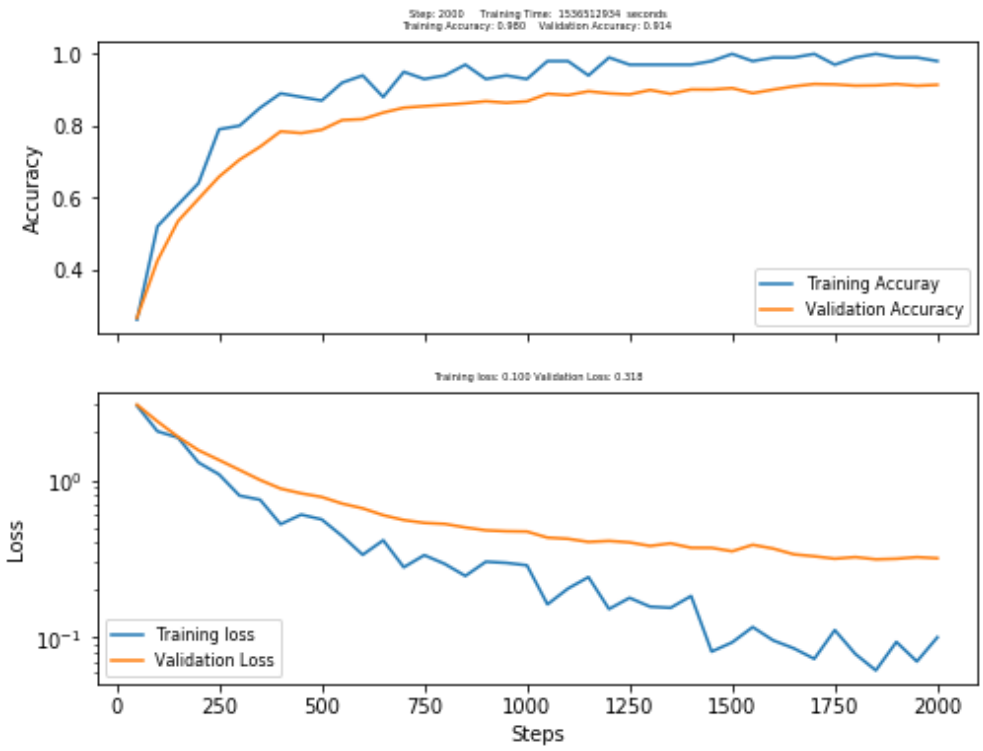


Ilustración 41. Precisión M4

Análisis de resultados

A continuación, se muestra la tabla de resultados obtenidos sobre 6 pruebas distintas.

	MODELO 4		
	Precisión Ent.	**Precisión Val.**	**Diferencia**
ITER. 1	98,00	90,20	9,90
ITER. 2	99,00	90,80	9,70
ITER. 3	99,00	90,50	10,50
ITER. 4	100,00	91,10	10,80
ITER. 5	100,00	92,20	10,70
ITER. 6	98,00	90,00	10,60
MEDIA	99,00	90,80	10,37
VARIANZA	0,80	0,63	0,21

Tabla 8. Precisión M4

Obtenemos una media de precisión para el subconjunto de entrenamiento de 99% y de 90,8 para el subconjunto de validación.

A continuación, se muestra una tabla comparativa de los 4 modelos.

SUBCONJUNTO	MODELO 1	MODELO 2	MODELO 3	MODELO 4
Precisión Ent.	64,50	91,53	92,37	99,00
Precisión Val.	59,77	81,17	80,92	90,80
Varianza Ent.	24,35	0,42	1,20	0,80
Varianza Val.	17,87	1,04	1,72	0,63
Media de la dif.	4,73	10,37	11,45	10,37
MAX	70,20	92,40	94,00	100,00
MIN	53,80	80,20	79,70	90,00

Tabla 9. Comparativa M4

Se observa una mejoría notable en la precisión del modelo 4, el subconjunto de entrenamiento obtiene un 99,0% y el subconjunto de validación u 90,8 de acierto. La varianza se reduce en un 50% respecto al Modelo 3. La media de las diferencias de precisión entre subconjuntos se sitúa con el mismo valor que el Modelo 2. Finalmente obtenemos una precisión máxima para el subconjunto de entrenamiento del 100% de aciertos.

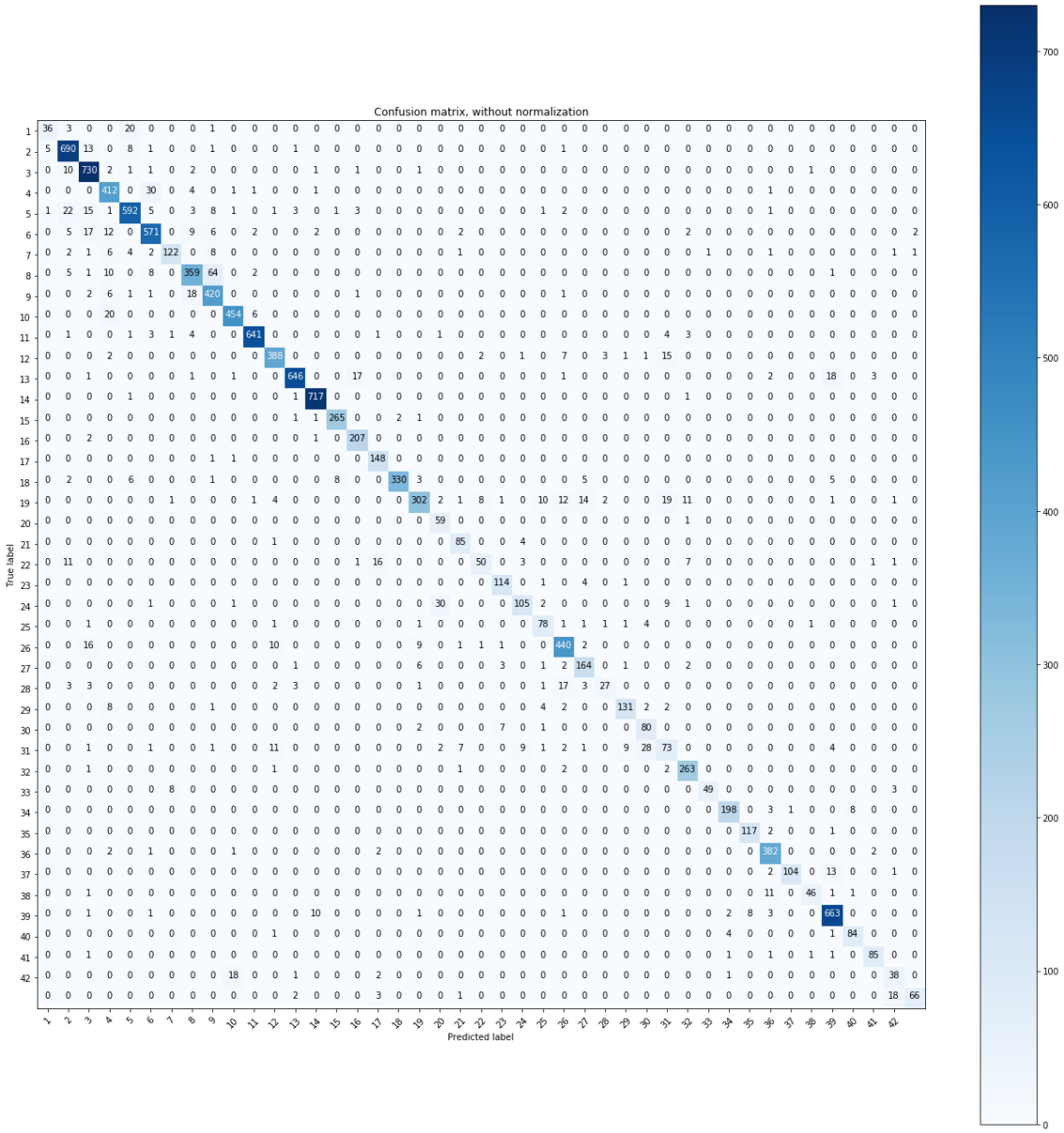
El Modelo 4 ha reducido un margen de error evitable para el subconjunto de entrenamiento a aproximadamente 0%, sin embargo, para el subconjunto de validación, el margen de error evitable es del 9%.

La mejoría respecto al Modelo 3 es notable, sin embargo, el modelo no se consigue bajar la dispersión ni tampoco la diferencia de precisión entre modelos.

Dado que hemos obtenido una precisión del 100% en el subconjunto de entrenamiento, deducimos que la estructura de la red neuronal es suficientemente compleja como para representar correctamente el problema. No obstante, el Modelo 4 sobre ajusta al subconjunto de entrenamiento. Para la prevención del sobre ajuste es necesario aplicar técnicas de *overfitting*.

Confussion matrix

Se observa que la matriz es más diagonal que en los modelos anteriores. Sin embargo, vemos que el error se sigue focalizando en algunas categorías puntuales. Para los límites de velocidad vemos que hay muchas imágenes mal clasificadas.



Porcentaje de aciertos

Se observa como la mejoría en el porcentaje de aciertos por categoría es importante. La categoría con menos precisión de todo el modelo tiene un 45% mientras que para los anteriores modelos nos encontrábamos en 20-25%. De forma general, la precisión global no ha mejorado sustancialmente, pero si lo ha hecho la precisión local en las categorías menos representadas.

Ilustración 42. Porcentaje de aciertos M4

La categoría con menor porcentaje de acierto obtiene un 45%. Solo 10 categorías obtienen una precisión por debajo del 80%.

Conclusiones

El Modelo 4 (capas Convolucionales) obtiene un 90,8% de precisión sobre el subconjunto de validación, una varianza entre pruebas de 0,63 y una media de las diferencias entre la precisión de los subconjuntos de 10,37%. De estos resultados, se obtienen un margen de error evitable del 8-9 %, lo que se puede considerar muy satisfactorio.

Con el Modelo 4, se ha obtenido una precisión mayor a la de un ser humano, no obstante, el modelo obtiene, de media, un 10% menos de precisión para el subconjunto de verificación. Se deduce que el modelo está sobre ajustando al subconjunto de entrenamiento.

8. Conclusiones sobre los resultados del GTSRB

El análisis previo del conjunto de datos observó una **dispersión en la distribución del número de imágenes por categoría** muy elevada y gran variedad de apariencias visuales en las imágenes: condiciones lumínicas, rotación, climatología y escalamiento.

Para el Modelo 1, se confirmó la relación entre la dispersión del número de imágenes por categoría, los histogramas de pesos con distribución uniforme y la baja precisión por categoría. Se observó que aquellos histogramas con una distribución de pesos uniforme se relacionaban de forma directa con las categorías con menor precisión que a su vez eran las categorías con menor número de imágenes.

La relación entre las categorías con menor número de imágenes y el porcentaje de acierto se diluyó a medida que la complejidad de los modelos aumentó. Para el Modelo 1 se detectaron 30 categorías sin representación en el modelo y con un 0% de acierto, ya en el Modelo 2 esta tendencia se rompió y la categoría con menor precisión mostró un 30% de acierto.

Se observó de forma general un **aumento de la precisión** en el subconjunto de entrenamiento junto con la complejidad de los modelos. La varianza y la media de las diferencias de precisión entre el subconjunto de entrenamiento y validación también disminuyeron de forma general.

El Modelo 4 (capas Convolucionales) obtuvo un 90,8% de precisión sobre el subconjunto de validación, una varianza entre pruebas de 0,63 y una media de las diferencias entre la precisión de los subconjuntos de 10,37%. De estos resultados, se obtienen un margen de error evitable del 8-9 %, lo que se puede considerar muy satisfactorio.

La media de las **diferencias de precisión entre subconjuntos de 10,37%** junto con una **precisión del 99%** sobre el subconjunto de entrenamiento indicaron un ligero **sobreajuste del Modelo 4 al subconjunto de prueba.**

Consideramos que la estructura del Modelo 4 es suficientemente compleja como para representar el problema presentado y que se puede mejorar el modelo con técnicas que aumenten la precisión del subconjunto de verificación y reduzcan el sobreajuste.

Belgium Traffic Sign Dataset (BTSD)

1. Introducción

El BelgiumTS (BTS) es un conjunto que se publicó en 2010 de más de 10.000 imágenes etiquetadas de miles de señales de tráfico físicas. Se gravaron 4 secuencias de video a través de 8 cámaras de alta resolución, en total se obtuvieron 3 horas de video con etiquetas sobre las imágenes. Las imágenes fueron grabadas en entornos urbanos de Bélgica, concretamente en la región de Flandes.

En fecha del 2018 disponemos en la página web del BTS de dos archivos de entrenamiento y prueba con imágenes etiquetadas en 62 clases de señales de tráfico. También se ha puesto a disposición del público los archivos de vídeo etiquetados para tareas de detección de señales.

2. Análisis del conjunto de datos

Los archivos publicados tienen una estructura muy similar al GTSD. Para el subconjunto de entrenamiento tenemos 62 carpetas numeradas desde el 00000 hasta el 00061, en cada carpeta se encuentran las imágenes en formato ".ppm" y un archivo ".csv" con información sobre las dimensiones de cada imagen, su posición sobre el *frame* de la imagen original y la categoría. El subconjunto de entrenamiento se estructura de forma idéntica.

Para la importación de los datos se va a utilizar el mismo procedimiento que para el GTSD. De esta forma, obtenemos dos listas de Python de longitud igual al número de imágenes. La lista Python contendrá las imágenes en *arrays* de 3 dimensiones de la misma forma que para el GTSD.

Con el siguiente código y sus resultados se comprueba que los datos importados se han ordenado según se ha descrito y tienen el tamaño que se espera.

```
print("ARRAY DE LABELS TRAIN")
print("Tipo de objeto Python: " , type(Y_train_orig))
print("Número de imágenes totales: ", len(Y_train_orig))
print("Número de clases totales: ", len(set(Y_train_orig)))
print(Y_train_orig)
```

```
ouT[7]:
ARRAY DE LABELS TRAIN
Tipo de objeto Python:  <class 'list'>
Número de imágenes totales:  4575
Número de clases totales:  62
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ,…, 61, 61, 61, 61, 61, 61, 61, 61]
```

Para el subconjunto de entrenamiento tenemos 4575 imágenes con 62 clases ordenados en una lista de Python.

```
Out[8]:
ARRAY DE LABELS TEST
Tipo de objeto Python:  <class 'list'>
Número de imágenes totales:  2520
Número de clases totales:  53
HAY 9 CATEGORÍAS SIN REPRESENTAR
Categorías NO representadas: [9, 11, 15, 26, 33, 36, 48, 50, 52]
```

Para el subconjunto de verificación obtenemos 2520 imágenes de 52 categorías. Hay 9 categorías que no se representan en el subconjunto de verificación y que si lo hacen en el subconjunto de prueba.

Para aportar más claridad sobre los datos con los que se está trabajando, se va a mostrar en la siguiente figura, el tipo de señal que representa cada categoría junto al número de imágenes del subconjunto de prueba y subconjunto de validación.

Ilustración 43. Imágenes ejemplo BTS

Para poder realizar un análisis previo complementario al de la figura anterior, se va a imprimir un ejemplo por cada categoría junto con el número de ejemplos de esa categoría.

Ilustración 44. Imágenes ejemplo BTS

Del análisis de las dos figuras anteriores se puede extraer las siguientes conclusiones:

- El número de imágenes por categoría es muy dispar.
- En el subconjunto de validación no se representan todas las categorías.
- Todas las señales se encuentran encuadradas.
- Algunas categorías son muy parecidas.
- No todas las imágenes tienen el mismo tamaño.
- La calidad de las imágenes es baja.
- La exposición lumínica y el ángulo de la señal varía mucho.
- Solo hay una categoría de límite de velocidad.

Distribución de ejemplos por categoría

Como ya hemos comentado para el GTSD, la distribución dispar del número de imágenes por categorías es una fuente de error habitual en los modelos de *Machine Learning*. Por ello se va a realizar un análisis más detallado de la distribución de imágenes por categoría.

En la siguiente imagen se muestra un histograma con la distribución de imágenes por categoría del subconjunto de prueba y validación.

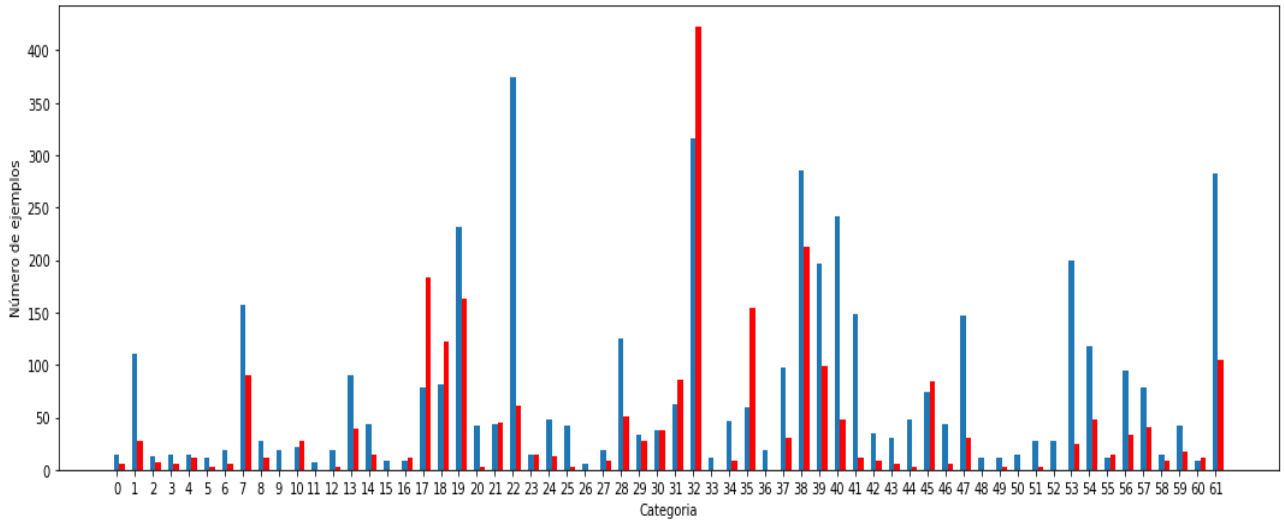


Ilustración 45. Comparativa número imagenes subconjuntos BTS

En la Ilustración 43, se observa una comparativa del número de imágenes del subconjunto de prueba (en azul) y el subconjunto de validación (en rojo). Se observa que la distribución de los subconjuntos no es la misma. Las categorías 17, 18, 32 y 35 del subconjunto de validación contienen mayor número de imágenes que el subconjunto de prueba. Las categorías 9, 11, 15, 26, 33, 36, 48, 50, 52 no se representan en el subconjunto de validación.

En la siguiente imagen se muestra la distribución de imágenes por categoría del subconjunto de entrenamiento-

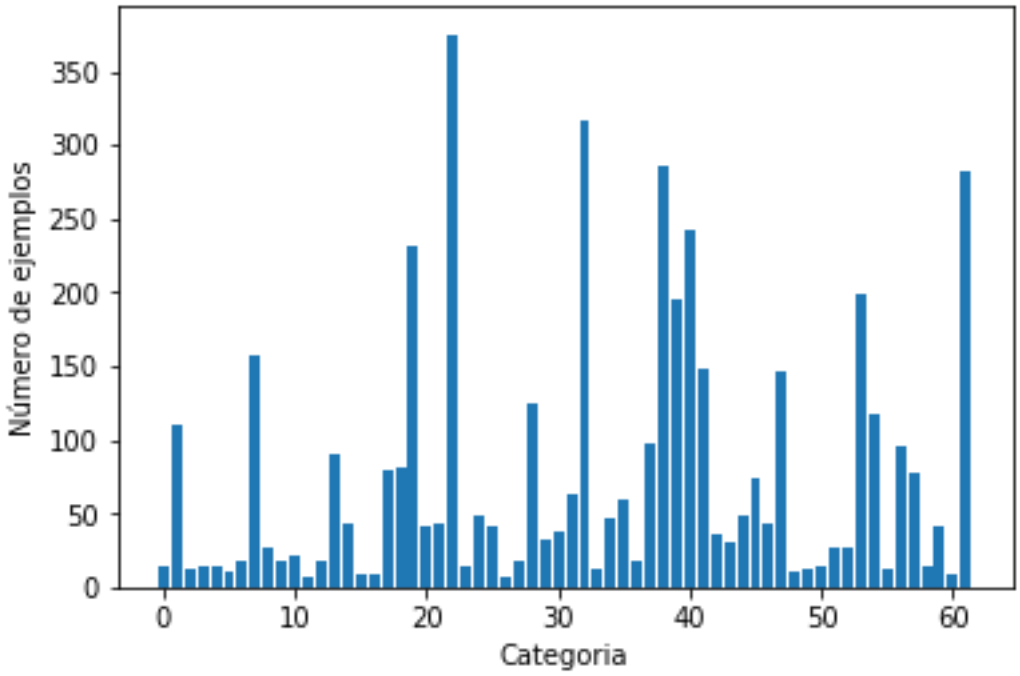


Ilustración 46. Número imágenes por categoria BTS.

Se observa que efectivamente la distribución por categoría es muy dispar. Para este análisis nos interesa sobre todo focalizar en las categorías con menor representación y para ello representaremos un nuevo histograma con las categorías con menos de 20 imágenes.

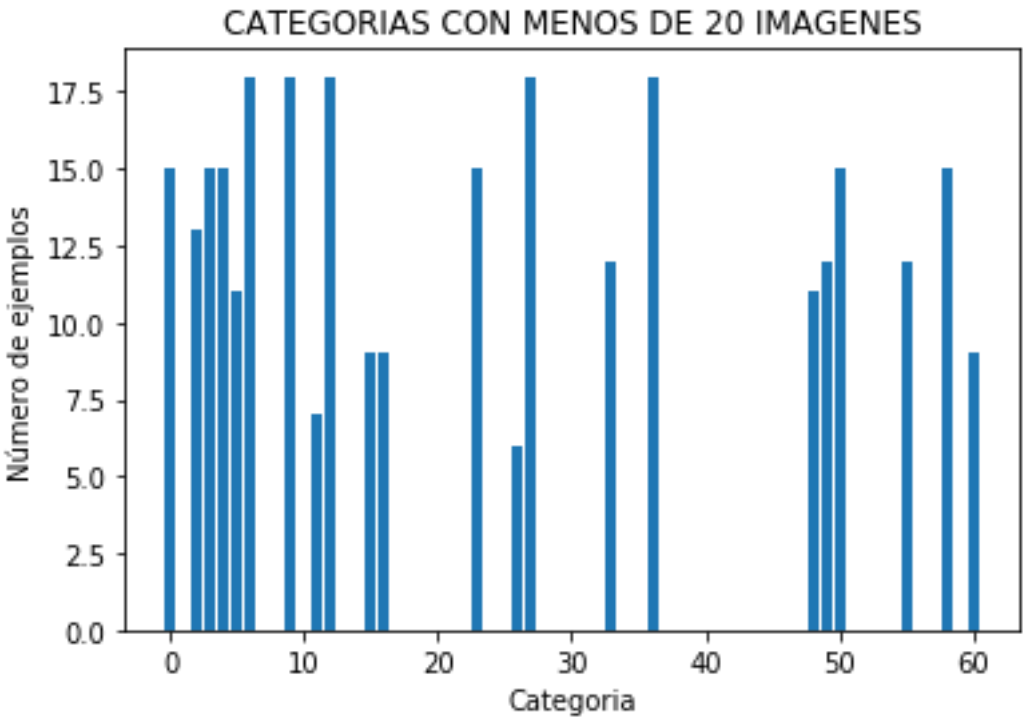


Ilustración 47. Categorias < 20 imágenes BTS

A continuación, se muestra una imagen a modo de ejemplo de las categorías identificadas anteriormente.

De este análisis se puede estraer que se pueden generar tres tipos de problemas por baja representación de categorías:

- Categorías con baja representación similares a otras categorías de mayor representación. (60; 48,49 y 50)
- Categorías con baja representación similares a otras categorías también de baja representación. (4 y 5)
- Categorías de baja representación no a semejables a otras categorías. (23)

En el primero de estos tres casos se puede producir que el modelo no sea capaz de asimilar estas categorías de baja representación y los identifique como aquellas de mayor representación. Para el segundo problema se puede producir una confusión entre estas dos categorías. Para el tercero de los problemas se puede producir una no representación de la categoría.

Análisis por categoría

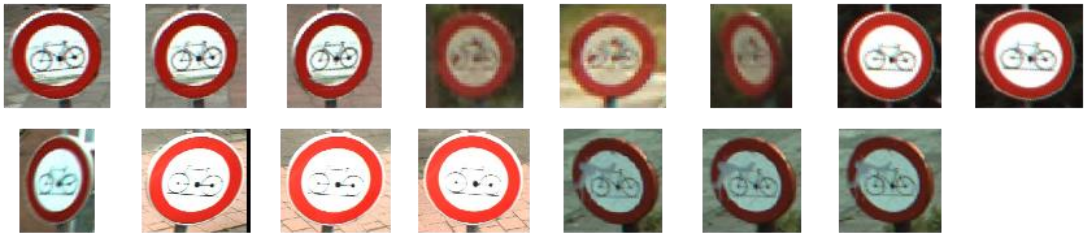
Anteriormente se ha observado que la categoría 32 es la única de límite de velocidad. Se va a realizar un examen más exhaustivo de esta categoría para saber qué tipo de imágenes nos se puede encontrar en esta categoría.

```
Out:
En la categoria 32, tenemos  316 ejemplos.
```

Se observa como todas las señales de límite de velocidad se han incluido en la misma categoría. También se puede apreciar, como se ha observado anteriormente, que las condiciones lumínicas, el ángulo y la calidad de la imagen varía.

Es interesante realizar este mismo análisis en otras categorías con el objetivo de detectar singularidades de la categoría que puedan ayudarnos a mejorar y analizar mejor los datos.

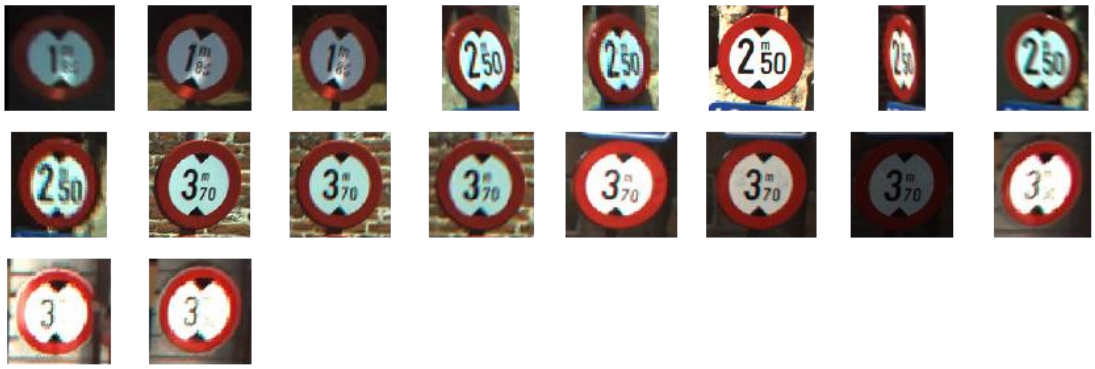
```
Out:
En la categoria 23, tenemos  15 ejemplos.
```



Se observa como la categoría 23, ya identificada como de baja representación, también destaca por su baja calidad y la distorsión de la bicicleta.

```
Out:
En la categoria 24, tenemos  48 ejemplos.
```

```
Out:
En la categoria 27, tenemos  18 ejemplos.
```



Las categorías 24 y 27 son un ejemplo de categorías parecidas. La categoría 24 con 48 ejemplos va a tener más peso dentro de nuestro modelo que la categoría 27 con solo 18 ejemplos, esto pude provocar que el modelo clasificar las imágenes de la categoría 27 en la 24.

3. Preparación de los datos

Como hemos comentado en el análisis del conjunto de datos, las imágenes no tienen el mismo tamaño. Para que un modelo de *Machine Learning* funcione es necesario que las dimensiones de las imágenes que recibe sean iguales.

Para nuestro modelo se va a redimensionar las imágenes a un tamaño de 32x32, dado que es un tamaño muy usual para la clasificación de este tipo de imágenes [12].

En el siguiente código importamos las imágenes como listas Python, las redimensionamos a 32x32 y finalmente las transformamos las listas de Python a *arrays* 3D.

```
def load_data(data_dir, width, height):
    directories = [d for d in os.listdir(data_dir)
                   if os.path.isdir(os.path.join(data_dir, d))]

    labels = []
    images = []
    for d in directories:
        label_dir = os.path.join(data_dir, d)
        file_names = [os.path.join(label_dir, f)
                      for f in os.listdir(label_dir) if f.endswith(".ppm")]
        for f in file_names:
            labels.append(int(d))
            images.append(skimage.transform.resize(skimage.data.imread(f), (width,
height)))

    labels=np.array(labels)
    images=np.array(images)

    return images, labels
```

También es necesario normalizar los valores RGB de las imágenes. Algunos autores, normalizan las imágenes después de redimensionarlas, aunque también hay autores que señalan que la función skimage.transform.resize() de Python ya normaliza las imágenes y no es necesario volver a normalizar [12]. Como se demuestra en los siguientes resultados la función skimage.transform.resize() para Python 3 normaliza las imágenes.

```
print('Las 5 primeras imágenes originales tienen los siguientes valores max y min de
RGB: ')
i=1
for image in X_train_orig[:5]:
    print('Valor max de la imagen',i, image.max())
    print('Valor min de la imagen',i, image.min())
    i+=i
```

```
Out:
Las 5 primeras imágenes modificadas tienen los siguientes valores max y min de RGB:
Valor max de la imagen 1 1.0
Valor min de la imagen 1 0.0073912377450982
Valor max de la imagen 2 1.0
Valor min de la imagen 2 0.003576899509804572
Valor max de la imagen 4 1.0
Valor min de la imagen 4 0.0015567555147058805
Valor max de la imagen 8 0.969267003676469
Valor min de la imagen 8 0.056774662990195915
Valor max de la imagen 16 0.9895220588235292
Valor min de la imagen 16 0.026654411764708223
```

4. Modelo 1

El primer modelo que se va a implementar se trata de una red neuronal completamente conectada de una sola capa igual al número de categorías de nuestro conjunto de datos.

Código y resultados

El código se ejecuta en TensorFLow. Como base para este modelo hemos utilizado el código de Walled Abdulla [12] adaptado al conjunto de datos con el que se está trabajando.

```
class Model1():
    #***BASIC MODEL WITH 1 FC LAYER***
    Def __init__(self):
        self.graph = tf.Graph()
        with self.graph.as_default():
            #Placeholders
            self.images=tf.placeholder(tf.float32,(None,32, 32,3), name='images')
            self.labels=tf.placeholder(tf.int32, [None], name='labels')
            #Flatten input
            self.images_flat=tf.contrib.layers.flatten(self.images)
            #Fully connected layer
            self.logits=tf.contrib.layers.fully_connected(self.images_flat,43,
tf.nn.relu)
            #print(self.logits)
            #Convert one hot vector to label indexes (int)
            self.predicted_labels=tf.argmax(self.logits,1)
            #Loss
            self.loss = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(
                    logits=self.logits, labels=self.labels))
            #Training
            self.train=tf.train.AdamOptimizer(learning_rate=0.001).minimize(self.loss)
            #Initialization
            self.init=tf.global_variables_initializer()
            #Create session
            self.session=tf.Session()
            #Run Initialization
            self.session.run(self.init)
```

Para este modelo, como método de inicialización, hemos utilizado *Xavier Initilization* para la matriz de pesos y ceros para la matriz de *bias*. Como ya hemos comentado, el *Forward Propagation* es una sola capa completamente conectada, la función de activación es *ReLu*, se va a utilizar un clasificador Softmax y para el cálculo del coste utilizaremos *Cross Entropy.*

La función *Train()* imprime el coste y la precisión del modelo durante su entrenamiento y también nos imprime la gráfica final que nos permite ver el decrecimiento del coste.

A continuación, se muestra la gráfica final que muestra a la vez, la evolución de la precisión del modelo sobre el subconjunto de entrenamiento y también el de validación por cada iteración. También se representa la gráfica de evolución del coste.

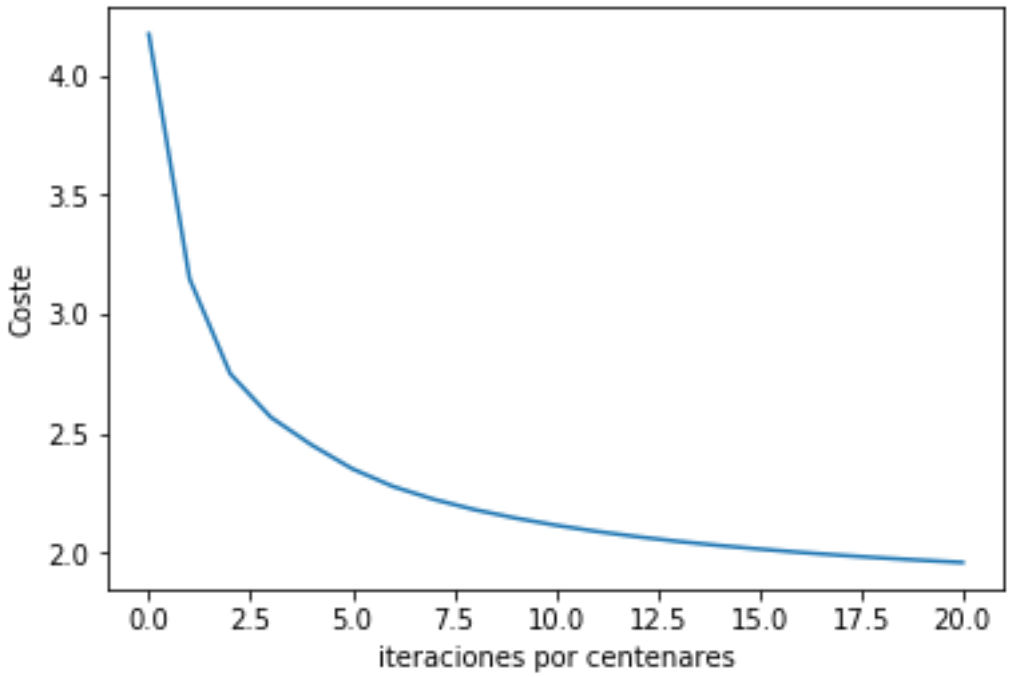


Ilustración 48. Coste M1 BTS

La curva del coste tiene la forma esperada, decrece de forma constante y tiende a una asíntota horizontal.

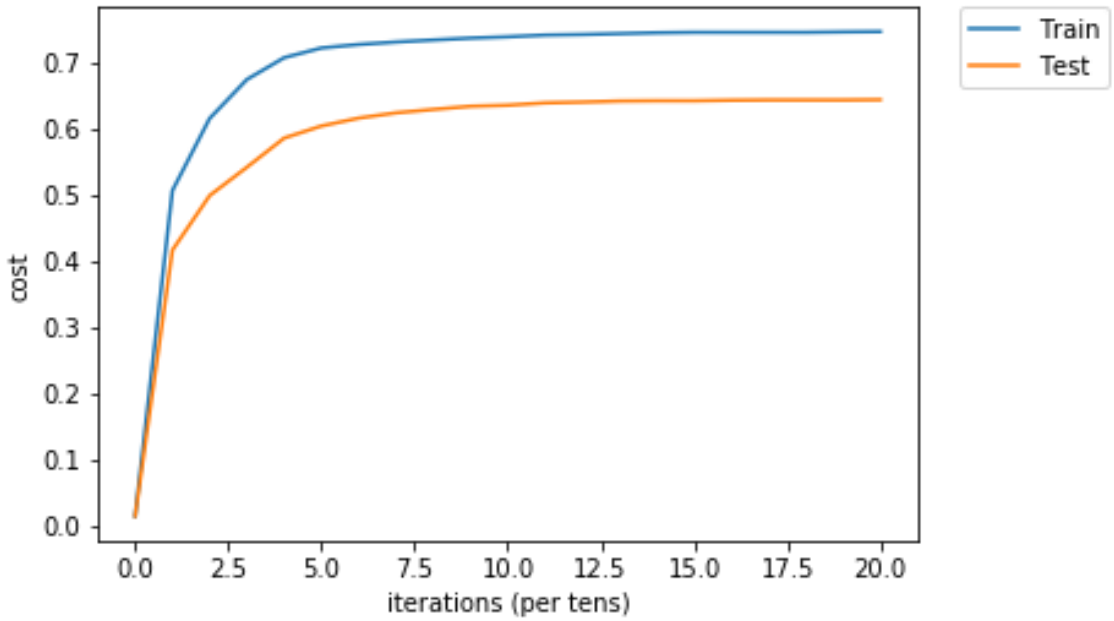


Ilustración 49. Precisión M1 BTS

```
Out:
199, Loss: 1.296  Train accuracy: 0.746
199, Loss: 1.771  Test accuracy: 0.643
```

La precisión final del modelo, para el subconjunto de validación, es de 64.3% y para el subconjunto de entrenamiento es de 74,6%.

Dada la simplicidad del modelo, es posible que la precisión final varíe mucho en función del punto en el que se inicie la optimización. Se va a realizar un conjunto de pruebas para observar como varia la precisión final del modelo.

```
Tensor("fully_connected/Relu:0", shape=(?, 62), dtype=float32)
 199, Loss: 1.400  Train accuracy: 0.724
 199, Loss: 1.643  Test accuracy: 0.687
Tensor("fully_connected/Relu:0", shape=(?, 62), dtype=float32)
 199, Loss: 1.760  Train accuracy: 0.632
 199, Loss: 1.782  Test accuracy: 0.650
Tensor("fully_connected/Relu:0", shape=(?, 62), dtype=float32)
 199, Loss: 2.038  Train accuracy: 0.566
 199, Loss: 2.059  Test accuracy: 0.580
Tensor("fully_connected/Relu:0", shape=(?, 62), dtype=float32)
 199, Loss: 1.177  Train accuracy: 0.780
 199, Loss: 1.214  Test accuracy: 0.781
Tensor("fully_connected/Relu:0", shape=(?, 62), dtype=float32)
 199, Loss: 1.969  Train accuracy: 0.576
 199, Loss: 2.537  Test accuracy: 0.444
```

Se observa que efectivamente la precisión final del modelo varía entre un 55 y 75%. Detectamos también que la diferencia de precisión entre el subconjunto de entrenamiento y validación no es constante y varía con cada experimento.

Análisis de los resultados

Los resultados de las 6 iteraciones realizadas se resumen en la siguiente tabla.

	MODELO 1		
Columna1	**Precisión Ent.**	**Precisión Val.**	**Diferencia**
ITER. 1	74,60	64,30	10,30
ITER. 2	72,40	68,70	3,70
ITER. 3	63,20	65,00	-1,80
ITER. 4	56,60	58,00	-1,40
ITER. 5	78,00	78,10	-0,10
ITER. 6	57,60	44,40	13,20
MEDIA	67,07	63,08	3,98
VARIANZA	83,77	127,42	40,81

Tabla 10. Precisión M1 BTS

Obtenemos una precisión medio del 67,07 % para el subconjunto de datos y 63,08% para el subconjunto de validación. La varianza de las precesiones para los subconjuntos de datos es de 83,77 y 127,42.

La diferencia de precisión entre subconjuntos varía entre 13,2 y -1,8 %. Es muy significativo es que la precisión del subconjunto de validación supera a la del subconjunto de entrenamiento en 3 de las 6 iteraciones.

A continuación, se muestra una tabla con los valores mínimos y máximos de cada subconjunto.

	Máximo	Mínimo	Diferencia
Subconjunto de entrenamiento	78%	56.6%	22
Subconjunto de validación	78.1%	44.4%	33.7

Tabla 11. Valores referencia M1 BTS

Es importante puntualizar que los valores mínimos y máximos de todas las iteraciones se corresponden con el mismo experimento. Los valores mínimos se obtienen en el 6º experimento y los valores máximos en el 5º.

Es difícil extraer conclusiones sobre si este modelo se está sobre ajustando al subconjunto de entrenamiento dado que existe mucha variación en los resultados. Para las iteraciones 1º y 6º parece claro que el modelo sobre ajusta al subconjunto de entrenamiento, sin embargo, para las iteraciones 3, 4 y 5 se produce el efecto contrario.

Se llama margen de error evitable a la diferencia de precisiones entre el modelo y la precisión teórica de un humano. Nuestro modelo obtiene un margen de error evitable de entre 20% y 55,6%. Deducimos entonces que tenemos mucho margen de mejora con respecto a este modelo.

De forma general, se puede observar que el modelo varía mucho en todos los aspectos. Los únicos valores que cambian en cada nueva iteración o experimento son los valores iniciales dados a la matriz de pesos. Se deduce que son los causantes de la volatilidad observada. Sería discutible si la elección de *minibatches* provocaría también esta volatilidad, aunque para este modelo no se realiza la optimización con *minibatches*.

Visualización de pesos

De la misma forma que para el GTSD se va a realizar un análisis de los pesos obtenidos con el modelo.

La siguiente figura muestra un ejemplo de cada una de las categorías de nuestro conjunto de datos y al lado la representación de los pesos para esa categoría. El título de cada imagen es el número de categoría y entre paréntesis el número de imágenes por categoría.

Ilustración 50. Representcion pesos M1 BTS

En un primer análisis visual se observa que las siguientes categorías obtienen unos pesos muy parecidos a las imágenes de la categoría: 7,9, 17, 18, 19, 22, 24, 28, 34, 37, 38, 40, 41, 42, 44, 47, 53, 54, 56 y 57.

A continuación, se representan los histogramas de los pesos que acabamos de representar de en forma de imagen.

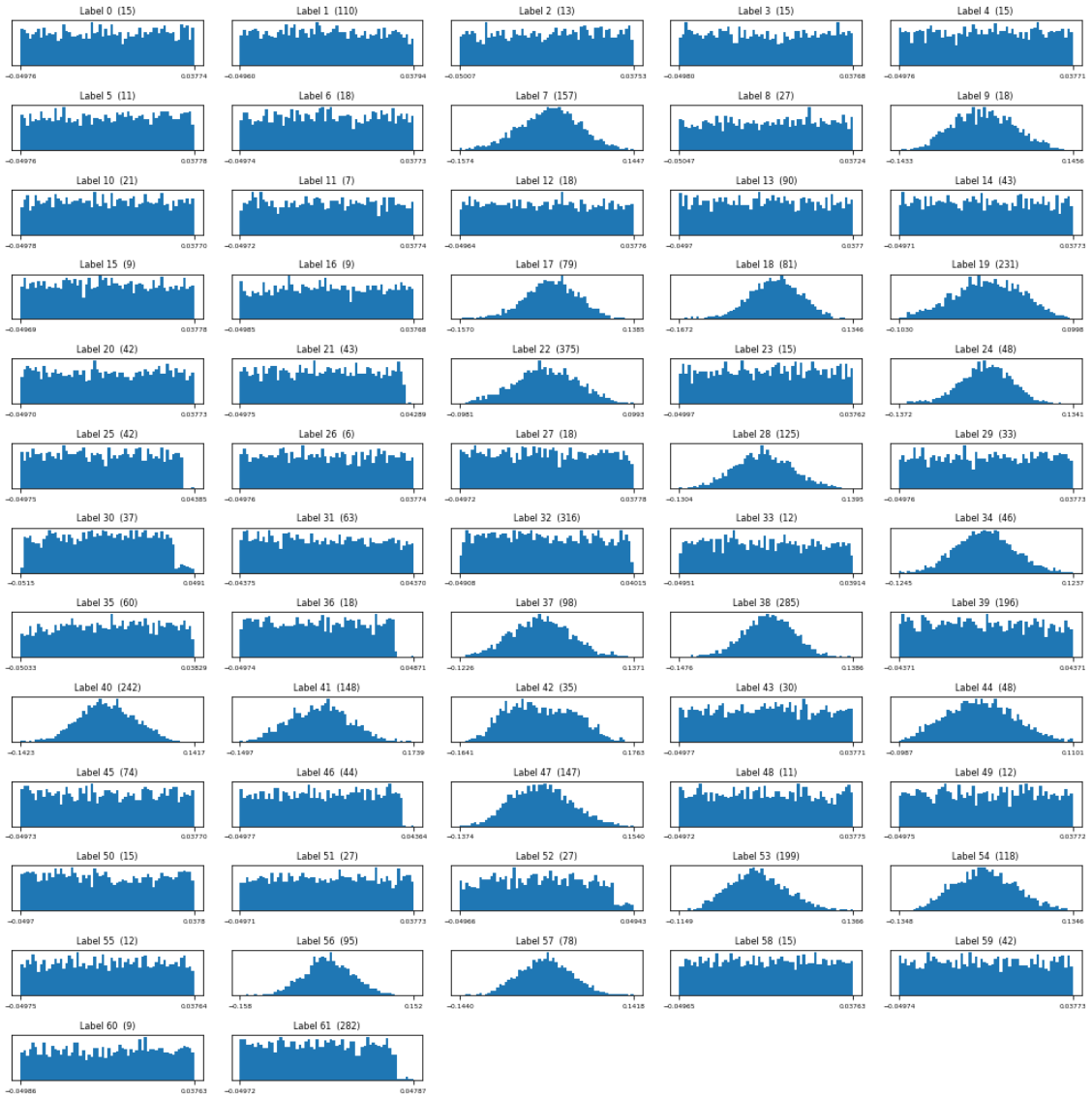


Ilustración 51. Histogramas pesos M1 BTS

Se observa que aquellas representaciones que habíamos detectado anteriormente y que mantenían cierto parecido con la imagen del ejemplo, en este caso, se manifiestan como histogramas Gaussianos. Aquellas categorías que no ha sido representadas con precisión obtienen un histograma aleatorio y uniforme, en cambio las categorías con una representación de los pesos más fiel a las imágenes obtienen un histograma Gaussiano.

A continuación, se representa un histograma de barras representando en rojo aquellas categorías cuyos histogramas son Gaussianos.

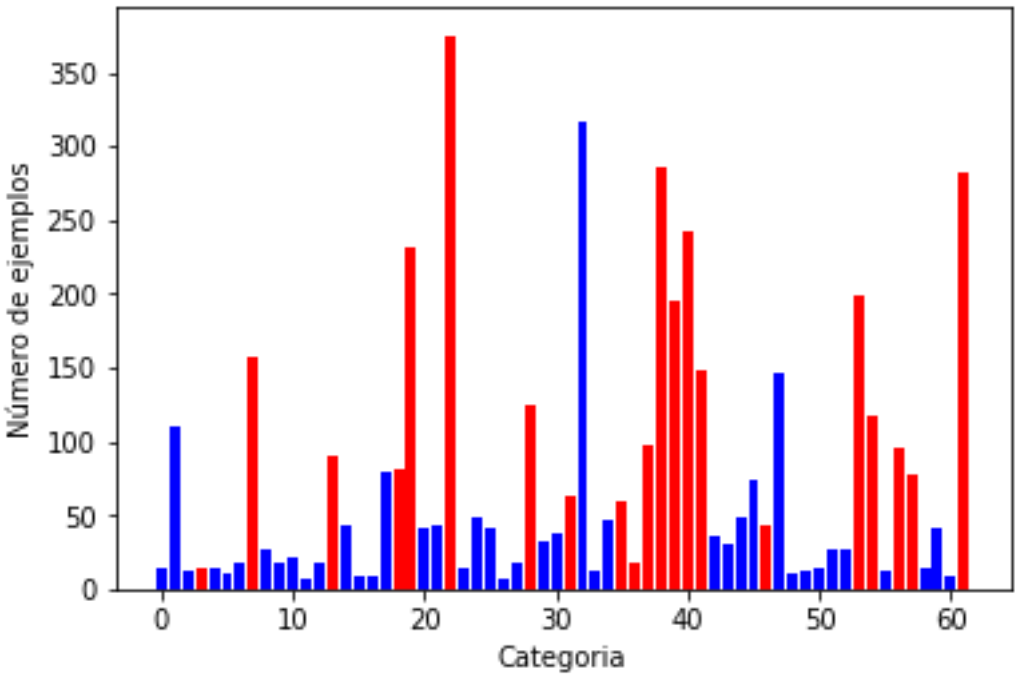


Ilustración 52. Histograma de pesos y número de imágenes BTS

Se observa una tendencia general de las categorías con mayor número de imágenes a tener histogramas Gaussianos. Se observa que la variación en la precisión general del modelo afecta también a la representación de los pesos. A continuación, se representa un histograma de pesos idéntico a la Ilustración 50 con los pesos de la segunda prueba con el Modelo 1.

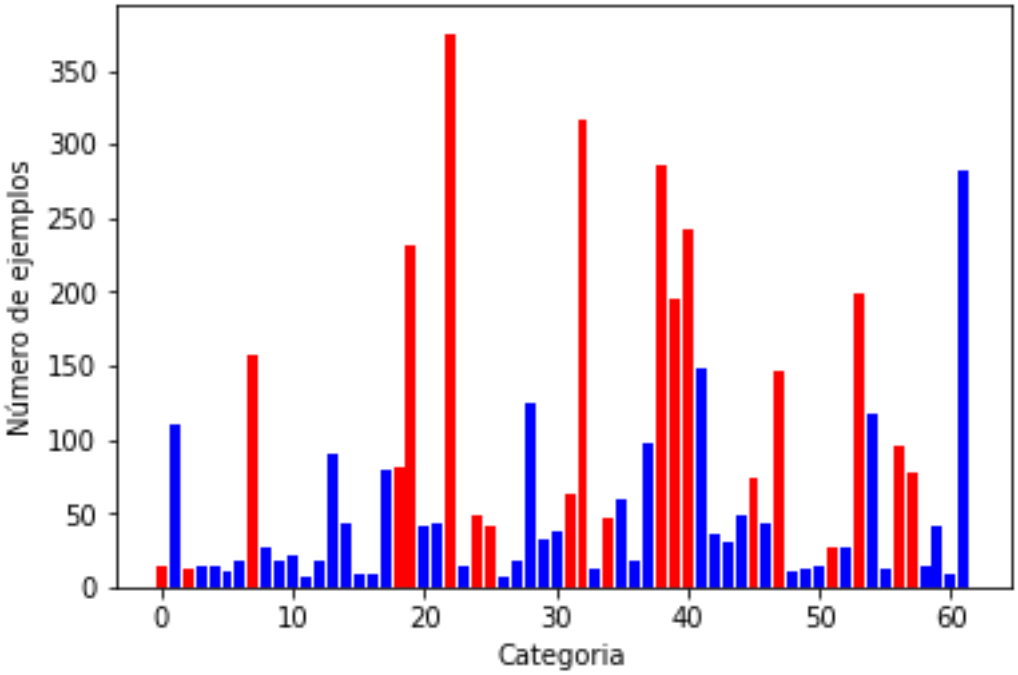
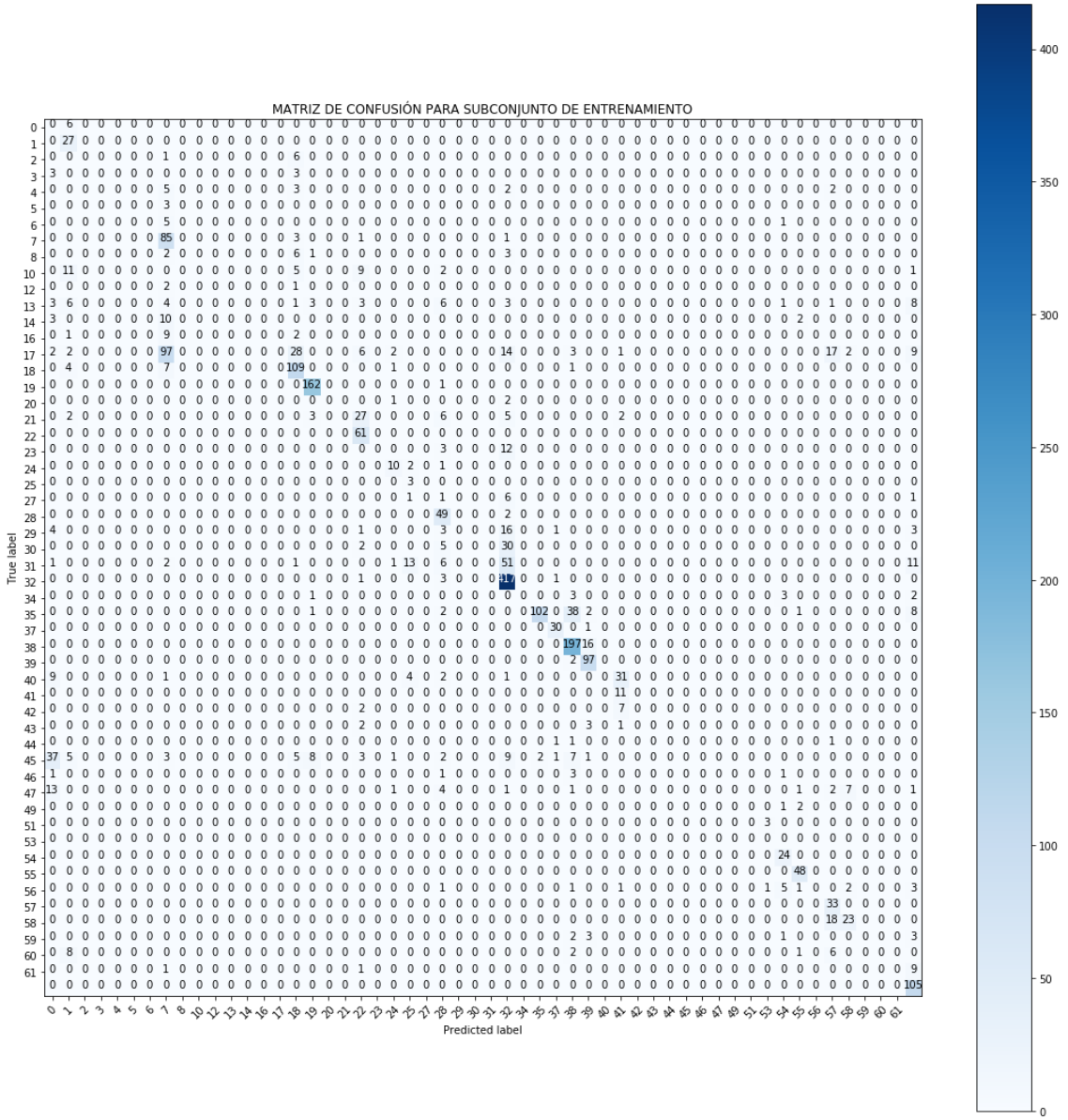


Ilustración 53. .Histograma de pesos y número de imágenes BTS (2)

De la comparación entre los dos histogramas se observan diferencias. La categoría 33 se representa en azul para el primer histograma y en rojo para el segundo. De forma general se reafirma la gran variación que sufre el modelo para distintas pruebas.

Matriz de confusión

Como ya se ha explicado, esta matriz representa los valores predichos por el modelo contra los valores reales. Para este caso y dado que el subconjunto de entrenamiento no contiene todas las categorías del subconjunto de entrenamiento deberemos analizar de forma minuciosa la información representada.



A continuación, se muestran las categorías representadas por el modelo y las categorías que debería haber representado.

```
print('CATEGORÍAS SUBCONJUNTO VAL.: ', y_test)
print('TOTAL: ',len(set(y_test)))
Out[100]:
CATEGORÍAS SUBCONJUNTO VAL: {0, 1, 2, 3, 4, 5, 6, 7, 8, 10, 12, 13, 14, 16, 17, 18, 19,
20, 21, 22, 23, 24, 25, 27, 28, 29, 30, 31, 32, 34, 35, 37, 38, 39, 40, 41, 42, 43, 44,
45, 46, 47, 49, 51, 53, 54, 55, 56, 57, 58, 59, 60, 61}
TOTAL: 53
```

```
Print('CATEGORÍAS PREDICHAS POR EL MODELO: ',y_pred)
Print('TOTAL: ', len(set(y_pred)))
Out[101]:
CATEGORÍAS PREDICHAS POR EL MODELO: {0, 1, 7, 12, 17, 18, 19, 21, 22, 24, 25, 28, 32,
35, 37, 38, 39, 40, 41, 42, 44, 45, 47, 52, 53, 54, 57, 61}
TOTAL: 28
```

La predicción del modelo se limita a 28 categorías, sin embargo, el subconjunto de validación tiene 53 categorías. Se observa cómo se han predicho menos categorías de las representadas en el subconjunto de validación.

A continuación, se analiza el número de columnas nulas de la matriz de confusión. Las categorías con columna nula representan categorías que no se han tenido en cuenta en la predicción, es decir no hay predicciones sobre estas categorías.

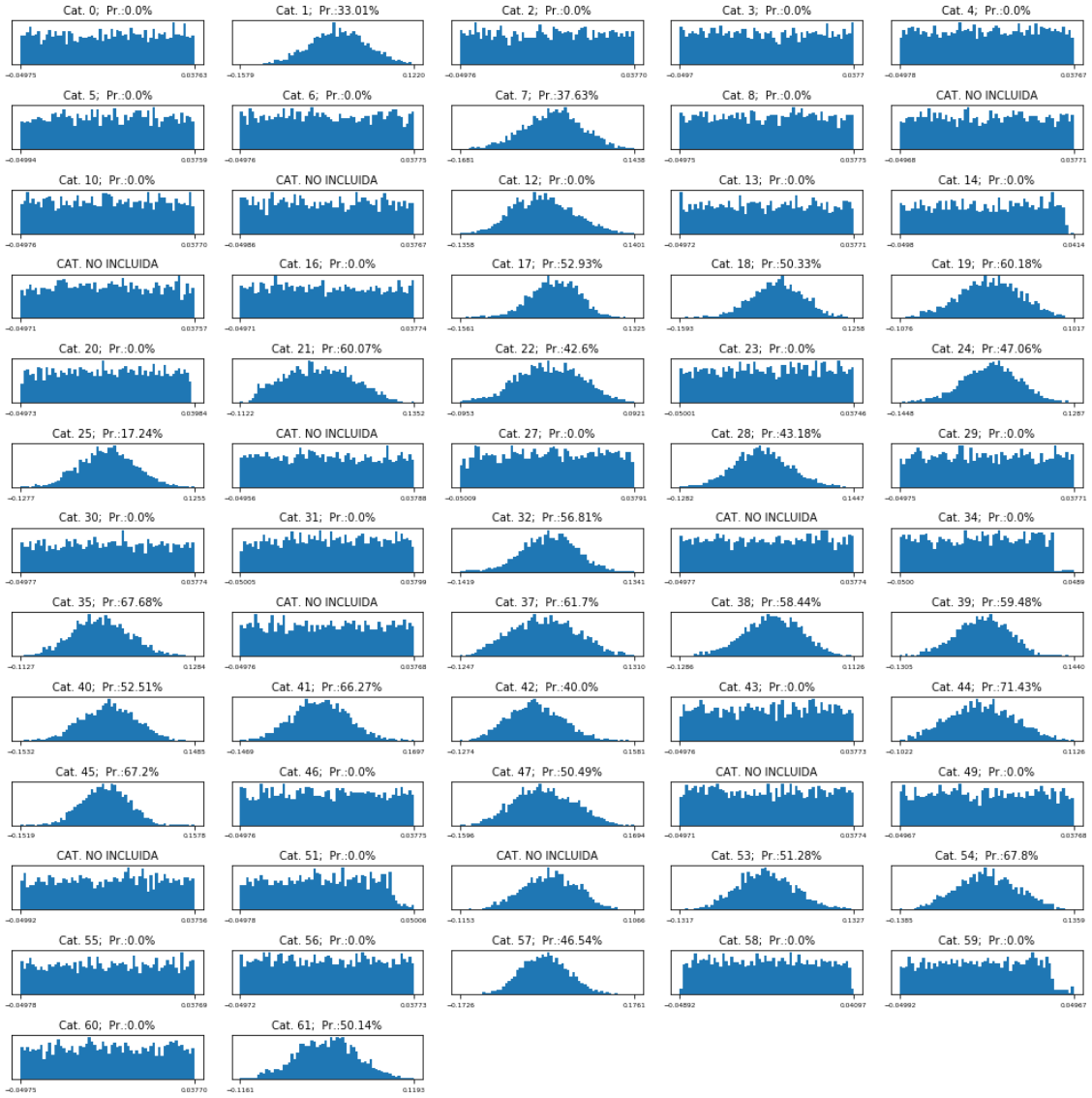
```
Out[55]:
Número DE CATEGORÍAS SIN CLASIFICAR:  26
CATEGORÍS NO CLASIFICADAS:  [2, 3, 4, 5, 6, 8, 9, 11, 12, 13, 17, 20, 23, 25, 26, 27,
29, 37, 40, 42, 43, 47, 48, 50, 51, 52]
```

Existen 26 categorías en la matriz de confusión con han sido olvidadas por el modelo. Además, la categoría 52, no estaba representada en el subconjunto de validación y ha sido predicha por el modelo.

Precisión por categoría

A continuación, se muestra la precisión en cada categoría junto con una imagen de la categoría. En esta representación, las categorías no incluidas en el subconjunto de prueba se han marcado con el siguiente título: “CAT. NO INCLUIDA”.

En la siguiente figura se muestra la precisión según los histogramas.



De nuevo, se observa como aquellas categorías con un histograma Gaussiano son las que aportan una mejor precisión.

Conclusiones

Como ya hemos comentado durante el análisis, este modelo sirve a modo de depuración de errores y visualización del conjunto de datos. Sin embargo, hemos obtenido un porcentaje de acierto máximo del 78% sobre 62 categorías.

Con un análisis más profundo, hemos detectado que **no todas las categorías han sido representadas**. De las 62 categorías originales, se observan 43 categorías con un porcentaje de acierto del 0%. Se comprueba que aquellas que no han sido representadas en el subconjunto de validación obtienen un 0% de acierto.

Se ha observado una **relación directa entre el porcentaje de acierto y la cantidad de imágenes en el conjunto de datos.**

Finalmente concluimos que, aunque la precisión global del modelo es bastante mayor de lo esperado para un modelo tan simple, la gran diferencia en cuanto a la cantidad de imágenes entre categorías ha influido muy negativamente en el modelo.

5. Modelo 2

En el segundo Modelo 2, se aplica la función de activación *LeakyReLu* y una función de optimización tipo *Minibatches*.

Código y resultados

A continuación, se muestran las gráficas del coste y precisión después de 4000 iteraciones.

```
3990, Loss: 0.028  Train accuracy: 0.997
3990, Loss: 0.369  Test accuracy: 0.909
```

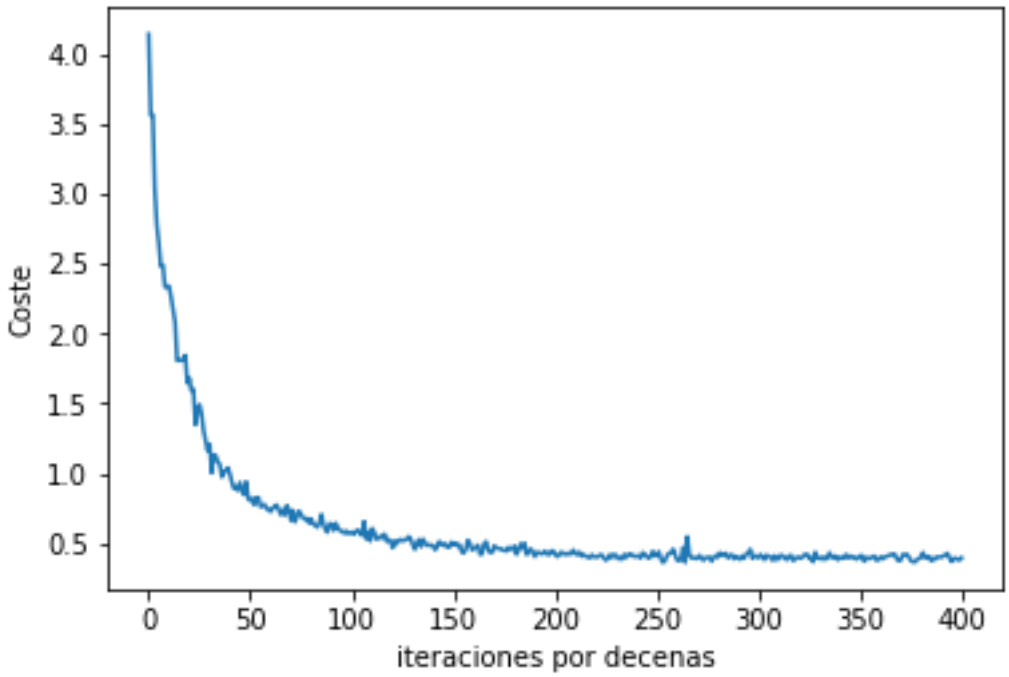


Ilustración 54. Coste M2

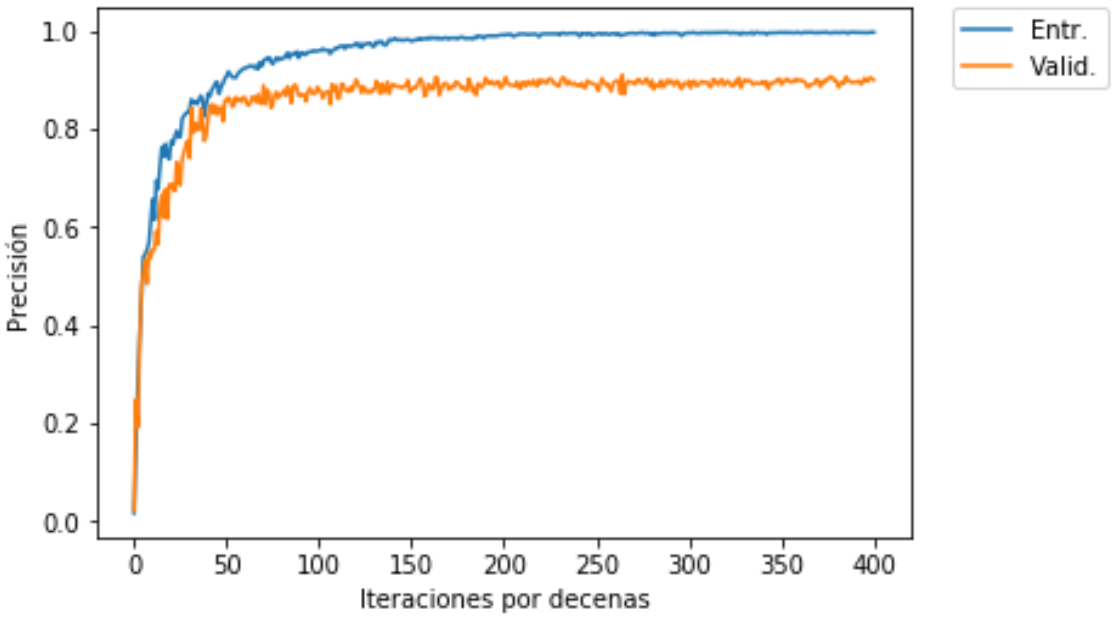


Ilustración 55. Precision M2

El modelo converge de forma muy rápida.

A continuación, se muestran los resultados de las 6 pruebas realizadas sobre este modelo.

```
4000, Loss: 0.029  Train accuracy: 0.996
4000, Loss: 0.400  Test accuracy: 0.893
4000, Loss: 0.024  Train accuracy: 0.997
4000, Loss: 0.390  Test accuracy: 0.901
4000, Loss: 0.029  Train accuracy: 0.996
4000, Loss: 0.369  Test accuracy: 0.911
4000, Loss: 0.027  Train accuracy: 0.997
4000, Loss: 0.402  Test accuracy: 0.900
4000, Loss: 0.027  Train accuracy: 0.997
4000, Loss: 0.402  Test accuracy: 0.901
4000, Loss: 0.025  Train accuracy: 0.998
4000, Loss: 0.438  Test accuracy: 0.888
```

Análisis de los resultados

En la siguiente tabla, se muestran los valores de precisión del modelo.

	MODELO 2		
Columna1	Precisión Ent.	Precisión Val.	Diferencia
ITER. 1	99,60	89,30	10,30
ITER. 2	99,70	90,10	9,60
ITER. 3	99,60	91,10	8,50
ITER. 4	99,70	90,00	9,70
ITER. 5	99,70	90,10	9,60
ITER. 6	99,80	88,80	11,00
MEDIA	99,68	89,90	9,78

VARIANZA	0,01	0,62	0,69

Ilustración 56. Precision M2

Obtenemos una precisión media global de 99,8% para el subconjunto de entrenamiento y 89,9% para el subconjunto de validación. La media de las diferencias de precisión entre subconjuntos es de 9,78% y las varianzas para el subconjunto de entrenamiento y validación son de 0,01 y 0,62. La precisión para el subconjunto del entrenamiento es muy alta y la dispersión de los resultados es muy baja.

SUBCONJUNTO	MODELO 1	MODELO 2
Precisión Ent.	67,07	99,68
Precisión Val.	63,08	89,90
Varianza Ent.	83,77	0,01
Varianza Val.	127,42	0,62
Media de la dif.	3,98	9,78
MAX	78,10	99,80
MIN	44,40	88,80

Ilustración 57. Comparación M2

Si comparamos el Modelo 1 con el segundo modelo se puede observar que la mejoría es muy importante. La dispersión de los resultados se ha reducido drásticamente y la precisión máxima llega al nivel humano. No obstante, se observa como la diferencia de precisión entre los dos subconjuntos se ha duplicado.

De forma general el rendimiento del Modelo 2 es positivo, aunque es importante destacar el aumento de la media de las diferencias de precisión que indica un sobreajuste del modelo.

Histogramas

En la figura Ilustración 56 se muestran los histogramas de los pesos junto con la precisión obtenida por el Modelo 3. Para aquellas categorías no representadas en el subconjunto de validación se ha titulado la imagen como “CAT NO INCLUIDA”.

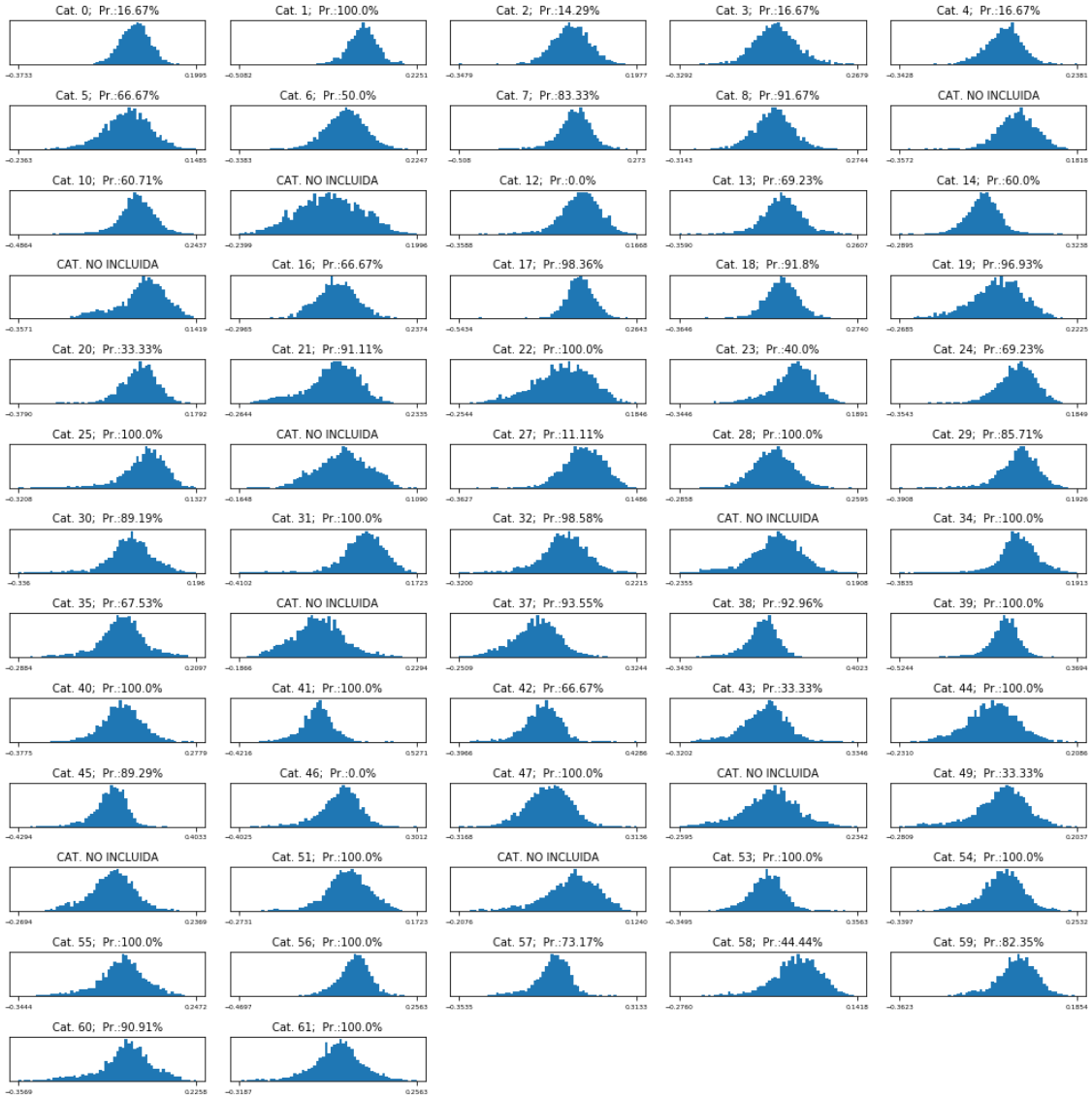


Ilustración 58. Histogramas M2 BTS

Se observa, a diferencia del Modelo 1 que todas las categorías obtienen un histograma Guassiano. La estructura de este modelo consigue representar todas las categorías del subconjunto de datos. Sin embargo, se observa que la precisión para el subconjunto de validación es más variable.

Aunque el Modelo 2 consigue representar el subconjunto de prueba de manera muy ajustada falla cuando se prueba con el subconjunto de validación. Concluimos que el modelo sobre ajusta al subconjunto de prueba.

Conclusiones

De forma general, se observa que el Modelo 2 obtiene una mayor precisión. A diferencia del Modelo 1, no obtenemos ninguna categoría con 0% a excepción de aquellas categorías que no ha sido representadas en el subconjunto de validación.

La precisión general del Modelo 1 se basaba en una gran precisión sobre las categorías con mayor número de imágenes, para el Modelo 2 se observa una **homogeneización de las precisiones y una aportación de todas las categorías a la precisión global**.

Aparte del acierto y la precisión también es importante notar que hemos conseguido reducir de forma notable el tiempo de convergencia de los resultados.

6. Modelo 3

Durante el análisis del Modelo 2 para el GTSD se concluyó que era necesario aumentar la complejidad de los modelos con capas completamente conectadas para conseguir mejores resultados y debido a esa conclusión, se implementó el Modelo 3 con 3 capas completamente conectadas. Finalmente, los resultados del Modelo 3 aplicados al GTSD no supusieron una mejora con respecto a los resultados del Modelo 2.

Para el caso del BTS se ha concluido que el Modelo 2 sobre ajusta al subconjunto de prueba. Por lo tanto, no es necesario aumentar la complejidad de este tipo de Redes Neuronales. De esta forma, la implementación del Modelo 3 al conjunto de datos BTS no resulta recomendable para aumentar la precisión.

Sin embargo, se va a implementar el modelo y analizaremos sus resultados de forma somera. Para el tamaño de las capas, utilizaremos la recomendación que realiza Waleed Abdulla en su artículo [12].

Resultados y análisis

Después de 7000 iteraciones, se muestran las siguientes curvas de coste y precisión.

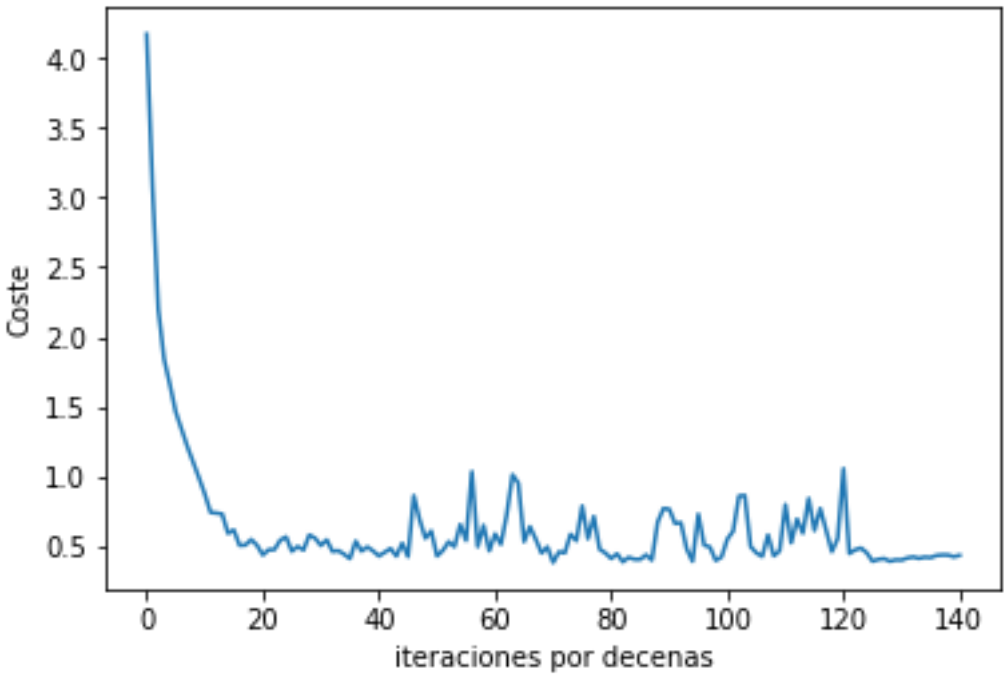


Ilustración 59. M3 coste (escala de abcisas ½)

Las precisiones sobre el subconjunto de validación y prueba son los siguientes:

```
6950, Loss: 0.011  Train accuracy: 0.997
6950, Loss: 0.434  Test accuracy: 0.926
```

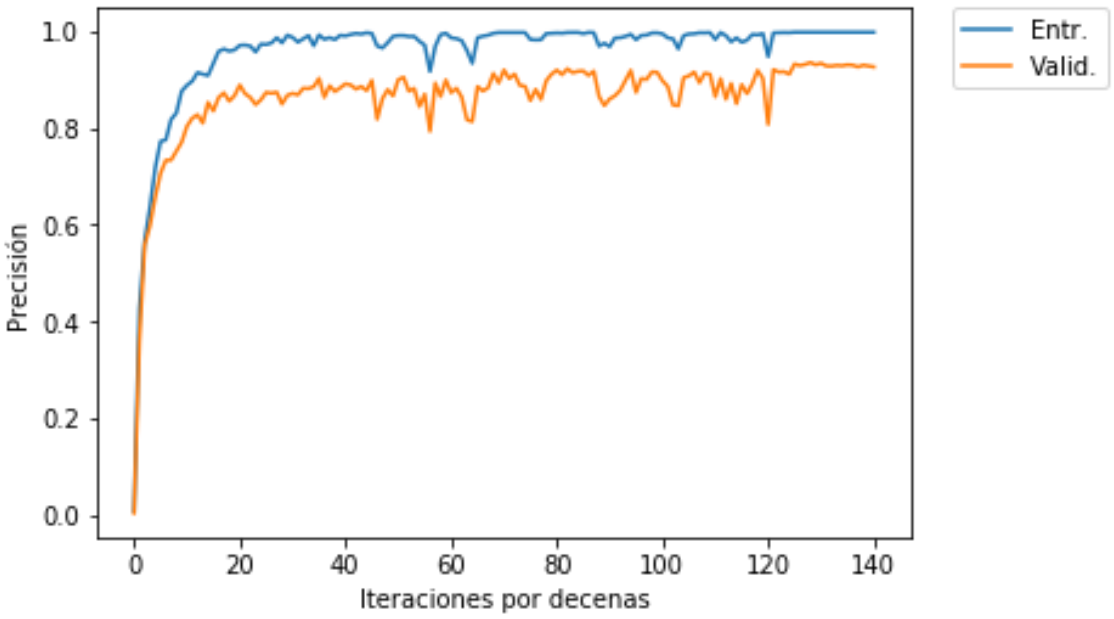


Ilustración 60. M3 precision (escala de abcisas ½)

Se observa como tanto la precisión, como el coste se estabilizan en 2000 iteraciones. Sin embargo, hemos continuado el entrenamiento hasta 7000 iteraciones y se observa como las oscilaciones debidas al tipo de optimización en *minibatches* se estabiliza.

En las tablas Tabla 12 y Tabla 13 se plasman los resultados obtenidos del Modelo 3.

	MODELO 3		
	Precisión Ent.	**Precisión Val.**	**Diferencia**
ITER. 1	99,80	91,20	8,60
ITER. 2	99,80	93,50	6,30
ITER. 3	99,70	92,60	7,10
ITER. 4	94,80	84,70	10,10
ITER. 5	100,00	92,50	7,50
ITER. 6	99,70	92,60	7,10
MEDIA	98,97	91,18	7,78
VARIANZA	4,18	10,63	1,85

Tabla 12. M3 precision

La precisión del Modelo 3 es muy parecida a las obtenidas en el Modelo 2.

SUBCONJUNTO	MODELO 1	MODELO 2	MODELO 3
Precisión Ent.	67,07	99,68	98,97
Precisión Val.	63,08	89,90	91,18

Varianza Ent.	83,77	0,01	4,18
Varianza Val.	127,42	0,62	10,63
Media de la dif.	3,98	9,78	7,78
MAX	78,10	99,80	100,00
MIN	44,40	88,80	84,70

Tabla 13. M3 comparativa

Comparando el Modelo 2 y 3 deducimos que la precisión se mantiene aproximadamente estable, la dispersión de los valores aumenta y la media de las diferencias disminuye.

Conclusiones

El Modelo 3 obtiene una precisión más baja que el Modelo 2 en el subconjunto de entrenamiento y una precisión superior para el subconjunto de validación, no obstante, estos resultados resultan más dispersos entre pruebas. También se observa una disminución en la media de las diferencias de precisión entre los subconjuntos de prueba y validación.

Se concluye entonces que **el Modelo 3 no sobreajusta** tanto como el Modelo 2 a expensas de una **disminución de la precisión** en el subconjunto de prueba. Por lo tanto, se deduce que una **red neuronal de capas completamente conectadas con una estructura más compleja es capaz de reducir el sobre ajuste de un modelo más simple.**

Con la intención de seguir mejorando la precisión de nuestro modelo, aumentaremos la complejidad del modelo e implantaremos una Red Neuronal Convolucional de cuatro capas.

7. Modelo 4

Hasta el momento, hemos realizado 3 modelos de Redes Neuronales, dos con una sola capa y una con tres capas. El siguiente paso para el reconocimiento de imágenes son las Redes Neuronales Convolucionales. Este tipo de red neuronal se distingue respecto a las Redes Neuronales totalmente conectadas en que no todos los nodos de una capa están conectados con los nodos de la siguiente.

Las características de este Modelo 4 son idénticas a las del Modelo 4 aplicado al GTSD.

Resultados y análisis

Las gráficas de coste y precisión del Modelo 4 se muestran a continuación.

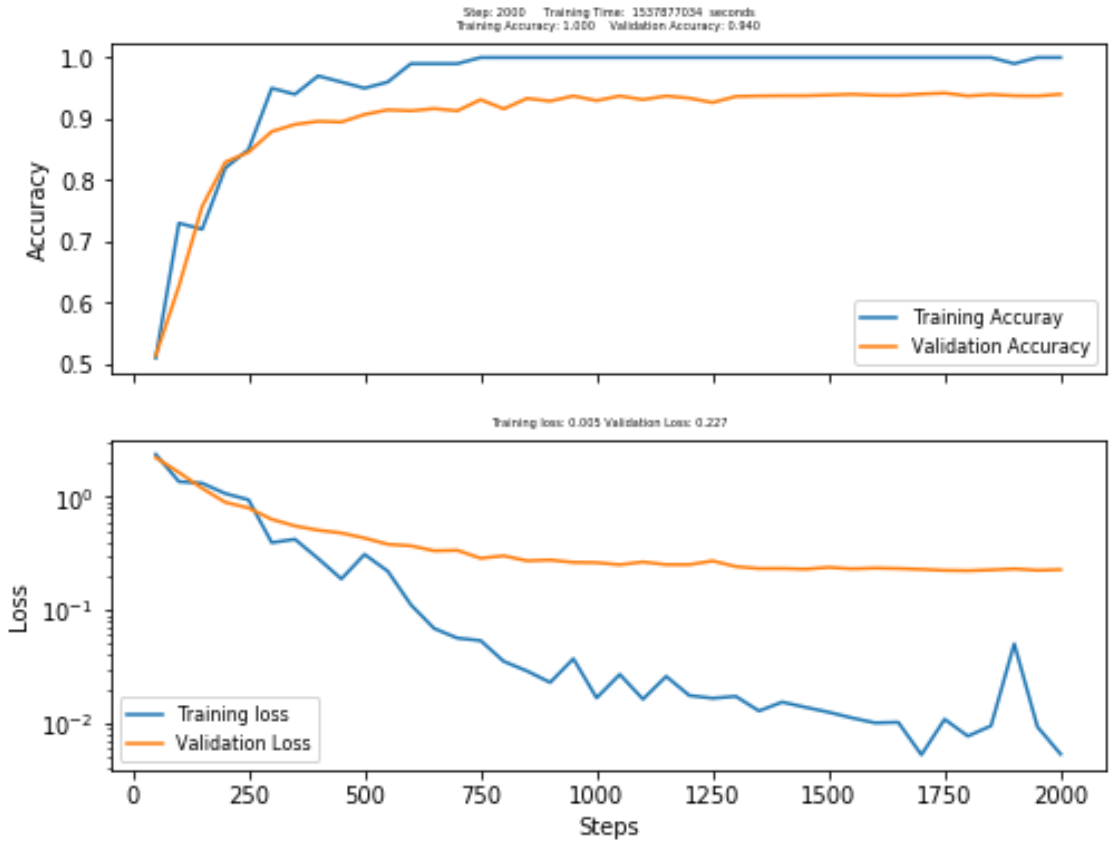


Ilustración 61. M4 precision

Los resultados obtenidos después de 2000 iteraciones para la precisión del subconjunto de prueba son del 100% de precisión. El subconjunto de validación obtiene un 94% de precisión.

```
2000, Loss: 0.005  Train set accuracy: 1.000
2000, Loss: 0.227  Validation set accuracy: 0.940
```

A continuación, se muestran los resultados de las 6 pruebas realizadas con este modelo.

	MODELO 3		
	Precisión Ent.	**Precisión Val.**	**Diferencia**
ITER. 1	100,00	94,00	6,00
ITER. 2	100,00	94,10	5,90
ITER. 3	100,00	94,00	6,00
ITER. 4	100,00	94,80	5,20
ITER. 5	100,00	94,80	5,20
ITER. 6	100,00	95,40	4,60
MEDIA	100,00	94,52	5,48
VARIANZA	0,00	0,33	0,33

Tabla 14. Precisión M4

Obtenemos un 100% de precisión en el subconjunto de entrenamiento y un 94,52 % de precisión en el subconjunto de validación.

La comparativa de los valores obtenidos para los cuatro modelos se muestra en la Tabla 15.

SUBCONJUNTO	MODELO 1	MODELO 2	MODELO 3	MODELO 4
Precisión Ent.	67,07	99,68	98,97	100,00
Precisión Val.	63,08	89,90	91,18	94,52
Varianza Ent.	83,77	0,01	4,18	0,00
Varianza Val.	127,42	0,62	10,63	0,33
Media de la dif.	3,98	9,78	7,78	5,48
MAX	78,10	99,80	100,00	100,00
MIN	44,40	88,80	84,70	94,00

Tabla 15. Comparativa M4

Conclusiones

Se mejora la precisión sobre el subconjunto de entrenamiento y sobre el subconjunto de validación. Obtenemos un máximo de precisión para el subconjunto de validación de 95,4%, lo que supone un 4,5% de diferencia de precisión entre subconjuntos. La dispersión de los resultados se ha reducido notablemente hasta un 0,33 para el subconjunto de validación.

Con el Modelo 4, se ha obtenido una **precisión mayor a la de un ser humano**, no obstante, el modelo obtiene, de media, un 5,5% menos de precisión para el subconjunto de verificación. Se deduce que el modelo está **sobre ajustando al subconjunto de entrenamiento**.

8. Conclusiones sobre los resultados del BTSD

De forma general, se confirman las tendencias observadas sobre el conjunto de datos del GTSRB. Aunque el número de imágenes del BTSD es mucho menor que para el GTSRB y la distribución de imágenes por categorías es más dispersa.el sub

Del análisis del conjunto de datos del BTSD se concluye que el subconjunto de entrenamiento no tiene representadas todas las categorías del subconjunto de prueba.

El análisis de la matriz de pesos "W" y su relación con el número de imágenes por categoría y la precisión por categoría demostró una **relación directa** entre el **número de imágenes por categoría** y la **precisión del modelo por categoría**. Las imágenes con menor representación sobre el subconjunto de entrenamiento obtuvieron una menor precisión.

Se observó de forma general un **aumento de la precisión** en el subconjunto de entrenamiento junto con la complejidad de los modelos. La varianza y la media de las diferencias de precisión entre el subconjunto de entrenamiento y validación también disminuyeron de forma general.

Para el Modelo 3 se observó una disminución de la precisión en el subconjunto de prueba y un aumento de precisión en el subconjunto de validación. Aunque la varianza de los resultados entre pruebas aumentó, se concluye que un aumento en la complejidad del modelo corrigió al sobreajuste del Modelo 1.

El Modelo 4 (capas convolucionales), obtuvo una **precisión mayor a la de un ser humano**, no obstante, el modelo obtiene, de media, un 5,5% menos de precisión para el subconjunto de verificación. Se deduce que el modelo está **sobreajustando al subconjunto de entrenamiento**.

Consideramos que la estructura del Modelo 4 es suficientemente compleja como para representar el problema presentado y que es necesario mejorar el modelo con técnicas que aumenten la precisión del subconjunto de verificación.

Conclusiones finales

Se ha realizado una descripción de las técnicas más recientes en modelos de predicción con Redes Neuronales para implementar cuatro modelos de predicción. Los modelos de predicción, se han aplicado al reconocimiento de señales de tráfico, para ello se han utilizado dos conjuntos de datos de señales de tráfico europeas: German Traffic Sign Recognition Benchmark (GTSRB) [9] y *Belgium Traffic Sign Dataset (BTSD)* [10].

El *GTSRB* contiene 43 clases y 50.000 imágenes repartidas en dos subconjuntos de datos para entrenamiento y validación, el subconjunto de entrenamiento contiene 39.209 imágenes y el subconjunto de validación contiene 12.630 imágenes. El BTSD contiene 63 clases y dos subconjuntos de entrenamiento y validación con 4.575 y 2.520 imágenes respectivamente.

El análisis previo de los dos conjuntos de datos observó una dispersión en la distribución del número de imágenes por categoría muy elevada y gran variedad en la apariencia visual de las imágenes: condiciones lumínicas, rotación, climatología y escalamiento. Los resultados demostraron una relación directa entre la distribución de las imágenes y la precisión del modelo por categoría.

La estrategia seguida para el análisis de los datos consta de cuatro modelos de predicción con estructuras diferentes de dificultad y complejidad creciente. La programación de los modelos se realizó en base al curso de *Deep Learning* de Andrew NG [11] y al artículo de Waleed Abdulla [12]

Para el primer modelo, se implementó una estructura de red neuronal muy simple con una capa oculta y número de nodos igual a la suma de categorías. El segundo modelo implementado tiene la misma estructura que el primer modelo y se incluyen mejoras en la función de activación (*ReLU*) y en el algoritmo de optimización (*Minibatches)*. Para el tercer modelo, se construyó una estructura de red neuronal más compleja con tres capas ocultas de tamaño decreciente. El cuarto modelo implementado constituye una Red Neuronal Convolucional de cuatro capas.

Para el análisis de los resultados, se estudió la precisión de los modelos sobre los subconjuntos de entrenamiento y validación. Con el objetivo de estudiar la variación de los resultados, se analizaron 5 pruebas independientes sobre el mismo modelo. En un análisis más profundo, se estudió la matriz de pesos optimizada “W” y la influencia del número de imágenes por categoría sobre los resultados.

Los resultados obtenidos sobre el GTSRB se muestran en la siguiente tabla.

GERMANT TRAFFIC SIGN RECOGNITION BENCHMARK				
SUBCONJUNTO	**MODELO 1**	**MODELO 2**	**MODELO 3**	**MODELO 4**
Precisión Ent.	64,50	91,53	92,37	99,00
Precisión Val.	59,77	81,17	80,92	90,80
Varianza Ent.	24,35	0,42	1,20	0,80
Varianza Val.	17,87	1,04	1,72	0,63
Media de la dif.	4,73	10,37	11,45	10,37

Tabla 16. Comparativa M4

Se observó un aumento de la precisión en el subconjunto de entrenamiento junto con la complejidad de los modelos. La varianza y la media de las diferencias de precisión entre el subconjunto de entrenamiento y validación también disminuyeron de forma general.

Se confirmó la relación entre la dispersión del número de imágenes por categoría, los histogramas de pesos con distribución uniforme y la baja precisión por categoría.

El Modelo 4 (redes convolucionales) obtuvo un 90,8% de precisión sobre el subconjunto de validación, una varianza entre pruebas de 0,63 y una media de las diferencias entre la precisión de los subconjuntos de 10,37%. De estos resultados, se obtiene un margen de error evitable del 8-9 %, lo que se puede considerar muy satisfactorio. La media de las diferencias de precisión entre subconjuntos de 10,37%, junto con una precisión del 99% sobre el subconjunto de entrenamiento, indican un ligero sobreajuste del Modelo 4 al subconjunto de prueba.

Los resultados obtenidos para el BTSD se muestran en la siguiente tabla.

SUBCONJUNTO	MODELO 1	MODELO 2	MODELO 3	MODELO 4
Precisión Ent.	67,07	99,68	98,97	100,00
Precisión Val.	63,08	89,90	91,18	94,52
Varianza Ent.	83,77	0,01	4,18	0,00
Varianza Val.	127,42	0,62	10,63	0,33
Media de la dif.	3,98	9,78	7,78	5,48

Para el Modelo 3 se observó una disminución de la precisión en el subconjunto de prueba y un aumento de precisión en el subconjunto de validación. Aunque la varianza de los resultados entre pruebas aumentó, se concluye que un aumento en la complejidad del modelo corrigió al sobreajuste del Modelo 1.

El Modelo 4 (capas convolucionales), obtuvo una **precisión mayor a la de un ser humano**, no obstante, el modelo obtiene, de media, un 5,5% menos de precisión para el subconjunto de verificación. Se deduce que el modelo el modelo **sobreajusta al subconjunto de entrenamiento**.

Las **conclusiones** del proyecto son:

- El rendimiento de las Redes Neuronales con capas convolucionales es mayor que con capas completamente conectadas.
- Las Redes Neuronales Convolucionales son capaces de resolver tareas de clasificación de imágenes con una precisión superior a los humanos.
- Los resultados obtenidos con la estructura de cuatro modelos de dificultad y complejidad creciente son consistentes en los dos conjuntos de datos.
- Existe una relación directa, en modelos simples, entre el número de imágenes por categoría y su precisión. Una mayor complejidad del modelo es capaz de corregir esta tendencia.
- Una mayor complejidad de las Redes Neuronales es capaz de corregir tendencias de sobreajuste al subconjunto de entrenamiento.

Bibliografía

[1] W. S. Mcculloch y P. Walter, «A logical calculus of the ideas immanent in neurons activity,» *Bulletin of matemathical biophysics,* vol. 5, 1945.

[2] Y. LeCun, «A theoretical framework for Back-Propagation,» *Proceedings of the 1988 Connectionist Models Summer School,* 1988.

[3] A. Wong, M. Jvad Shafiee y M. S. , «µNet: A Highly Compact Deep Convolutional Neural Network Architecture for Real-time Embedded Traffic Sign Classification,» 2018.

[4] Cireşan, Dan, Meier, Ueli, Masci, Jonathan y Schmid, «A committee of neural networks for traffic sign classification,» *IEEE International joint conference on neural networks,* 2011.

[5] Dan Ciresan, Ueli Meier, Jonathan Masci y Jurgen, «Multi-column deep neural network for traffic sign classification,» *Neural Networks,* 2012.

[6] Junqi Jin , Kun Fu y Changshui Zhang, «Traffic sign recognition with hinge loss trained convolutional neural networks,» *IEEE Transactions on Intelligent Transportation Systems,,* 2014.

[7] Hamed Habibi Aghdam, Elnaz Jahani Heravi y Domenec , «A practical approach for detection and classification of traffic signs using convolutional neural networks.».

[8] Arcos-García, Álvarez-García y Soria-Morillo, «Deep neural network for traffic sign recognition systems: An analysis of spatial transformers and stochastic optimisation methods.,» *Neural Networks,* vol. 99, pp. 158-165, 2018.

[9] Stallkamp, Johannes, Schlipsing, Marc, Salmen, Jan y Igel, Christian, «The German Traffic Sign Recognition Benchmark: A multi-class classification competition,» de *Proceedings of the International Joint Conference on Neural Networks*, 2011.

[10] R. Timofte y L. Van Gool, «Sparse Representation Based Projections. In British Machine Vision Conference,» *BMVC,* 2011.

[11] Andrew Ng, *Deep Learning Specialization,* 2017.

[12] W. Abdulla, «Medium,» Diciembre 2016. [En línea]. Available: https://medium.com/@waleedka/traffic-sign-recognition-with-tensorflow-629dffc391a6.

[13] J. Stallkamp, M. Schlipsing, J. Salmen y C. Ig, «Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition,» *Neural networks,* vol. 32, p. 323–332, 2012.

[14] D. E. Rumelhart, G. E. Hinto y Ronald J., «Learning representations by back-propagating errors,» *Nature,* 1986.

[15] M. Nielsen, «How the backpropagation algorithm works,» 2017. [En línea]. Available: http://neuralnetworksanddeeplearning.com/chap2.html.

[16] E. Mislej, «Minimum Viagle Models in Science Data,» Noviembre 2016. [En línea]. Available: https://www.kdnuggets.com/2016/11/practical-data-science-building-minimum-viable-models.html.

[17] K. Jalan, «How to Improve ML Algorithms,» [En línea]. Available: https://towardsdatascience.com/how-to-improve-my-ml-algorithm-lessons-from-andrew-ngs-experience-ii-f66926926f88.

[18] A. Ananthram, «Xavier Initialization For Neural Networks,» [En línea]. Available: https://towardsdatascience.com/random-initialization-for-neural-networks-a-thing-of-the-past-bfcdd806bf9e.

[19] «Rectifier (neural networks),» [En línea]. Available: https://en.wikipedia.org/wiki/Rectifier_(neural_networks).

Tabla de resultados