# INTRODUCCIÓN

## Neural nets NN

### Introducción (Y = W\*X+B)

### Función de activación

### Clasificador

### Coste

### Función de optimización

#### Mini batch gradient descent

## Convolutional NN

### Intro

### Pooling

## TensorFlow

# FALTA

## ACABAR ANALISIS DE SENSIBILIDAD EN MODELO 3 Y 4

## BELGIUM DATASET

No va a ser necesario alargarnos tanto en las explicaciones como en el German dataset.

Tiempo aproximado 2 días.

## INTRODUCCIÓN

Creo que es fundamental escribir una buena introducción y explicar bien todos los conceptos que he utilizado dado que es un tema poco conocido por el profesorado. La intención es explicarlo desde 0 y para que cualquier persona lo entienda.

## MEJORAS AL ANÁLISIS DE RESULTADO

Me da la sensación que el análisis de resultados es bastante pobre. ¿Debería mejorarlo?

## MEJORAS AL MODELO

Creo que aún hay sitio para mejorar el modelo. Por ejemplo en el algoritmo de selección de los minibatches o por ejemplo en el tamaño de la imagen.

## OVERFITTING

La precisión entre para el subconjunto de entrenamiento y de prueba es muy distante, parece que tenemos overfitting. Tenemos muchas soluciones posibles:

* Aumentar el tamaño del dataset modificando las imágenes (rotación, espejo, cambio luminosidad, etc.)
* Regularización
* GERMAN + BELGIUM 🡪 mexcla algunas categorías y ver que pasa.

## LE-NET 5

Aplicaremos un modelo clásico

# GERMAN DATASET

## Introducción

El German Dataset es un conjunto de datos muy utilizado y extensamente bibliografiado, 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.

Este conjunto de datos contiene más de 50.000 señales de trafico clasificadas en 43 clases. Las imágenes corresponden a distintas condiciones climatológicas y una gran variedad de exposiciones fotográficas.

El conjunto de datos fue extraído de 10 horas de vídeo filmadas en Alemania en los meses de marzo, octubre y noviembre del 2010 [1].

De cada señal de tráfico real o instancia se extraen varias imágenes, el conjunto de imágenes obtenidos de una misma señal de tráfico se denomina pista. Se obtuvieron un total 133.000 imágenes etiquetadas de 2416 instancias reales que se clasificaron en 70 categorias. Se descartaron las pistas con menos de 30 imágenes, 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. [1]

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. [1]

El conjunto de datos total se dividió en tres subconjuntos con el objetivo de realizar utilizarlos como subconjunto de entrenamiento, verificación y prueba. 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 podemos 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.

## Análisis del conjunto de datos

EL conjunto de datos se puede descargar desde la web en dos 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 imagen respecto a la pista.

**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

Par el archivo de prueba tenemos 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. Esto nos plantea un problema dado que no mantiene la misma estructura que el archivo de entrenamiento y cargarlo al modelo resulta más complicado. Finalmente, la solución utilizada consiste en exportar del archivo .csv el nombre de las imágenes y su categoría y de esta forma subir las imágenes al modelo.

**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

Finalmente tenemos un conjunto de 39.209 imágenes para el entrenamiento y otro conjunto de 12.630 para la verificación o prueba.

Una vez importado el subconjunto de entrenamiento a Python tenemos dos Python List de 39209 instancias cada una. Tenemos por un lado las imágenes o vector “X” y por otro las categorías o vector “Y”.

Por otro lado la Python List de categorías o vector “Y”, tiene una longitud de 39209 (igual al número de imágenes totales) y en cada instancia tiene el número de la categoria.

En un primer análisis visual del contenido del array de categorías observamos que están ordenadas y debemos tenerlo en cuenta dado que esto puede influir en la actuación del modelo.

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)

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, 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]

La Python List de las imágenes es un array de 3 dimensiones, las dos primeras dimensiones de cada instancia son el número de pixeles en X e Y; la tercera dimensión siempre igual a 3 se refiere al color en RGB.

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), ...]

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

Con el objetivo de poder comprobar que no hemos cometido ningún error el importar los datos a Python vamos a visualizar las imágenes. Esto también nos dará un primera visión del tipo de datos y de las estrategias que podemos utilizar para mejorar nuestros modelos.

**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("Label **{0}** (**{1}**)".format(label, labels.count(label)))

i += 1

\_ = plt.imshow(image)

plt.show()

Con esta función no solo conseguimos obtener una visión general del conjunto de datos sino que también podemos ver un ejemplo de cada categoría y la cantidad de ejemplos por categoría.

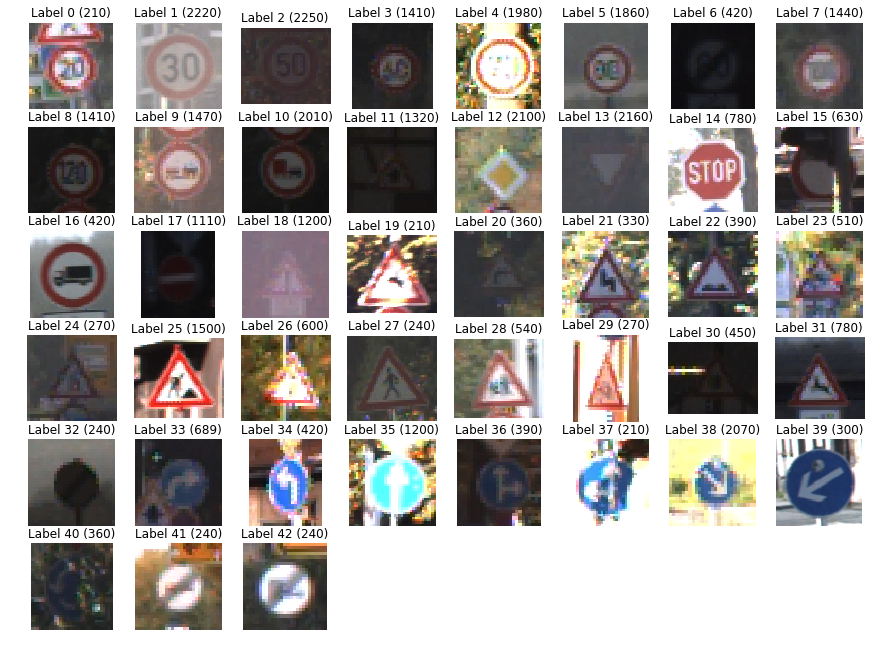
De la siguiente figura podemos obtener las siguientes conclusiones:

* El número de ejemplos por categoría es muy dispar
* Las imágenes están muy bien encuadradas
* La calidad de las imágenes no es buena
* No todas las imágenes tienen el mismo tamaño
* La calidad de alguna de las imágenes deja que desear
* En el primer ejemplo vemos que aparecen parte de otra señal y eso puede ser un problema

De todas estas conclusiones, hay dos a las que habrá que prestar especial atención.

Si el número de ejemplos por categoría es muy dispar vamos a obtener un modelo que prediga algunas categorías muy bien y otras no tan peor.

Para los modelos de Machine Learning necesitamos que todas las imágenes sean iguales por lo tanto deberemos modificarlas.

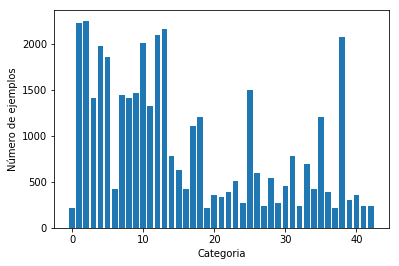


### Distribución de ejemplos por categoría

La distribución dispar de ejemplos entre categorías suele generar problemas en los modelos de predicción de Machine Learing [REFERENIA]. La diferencia de representación causa un mayor acierto en las categorías mejor representadas. Una categoría menor representada tiene un menor peso a la hora de actualizar los pesos de las matrizes durante el proceso de Back Propagation.

Dado que la distribución dispar entre categorías puede suponer un problema para el rendimiento de nuestro modelo vamos a realizar un análisis más profundo sobre como se distribuyen las imágenes por las categorías.

El primer paso es realizar un histograma de todas las categorías y el número de ejemplos.



Vamos a focalizar más el análisis para fijarnos en las categorías subrepresentadas y para ello hemos creado una función que nos devuelve los datos más interesantes, un conjunto de imágenes a modo de ejemplo y el histograma.

:

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')

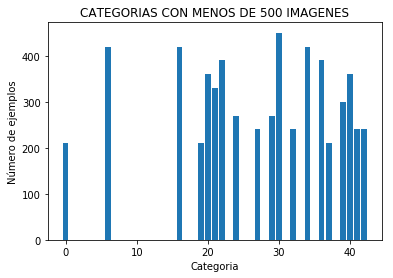
plt.title("Label {0} ({1})".format(label, Y\_train\_orig.count(label)))

i += 1

\_ = plt.imshow(image)

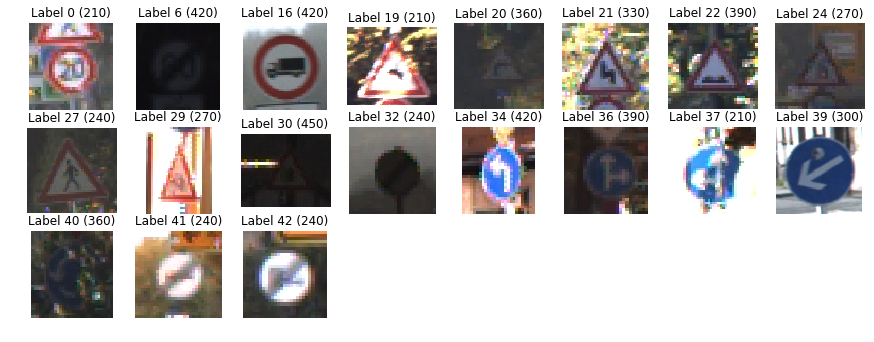
plt.show()

EL Histograma nos muestra que dentro de las categorías con menos de 500 ejemplos, encontramos un mínimo de 210 imágenes. Parece importante recordar que el mínimo para que una categoría entrara a formar parte del conjunto de datos debía tener mínimo 30 imágenes por pista y 9 pistas, en total 270 imágenes. Las 60 imágenes restantes de las categorías con 210 ejemplos estarán en el subconjunto de prueba.

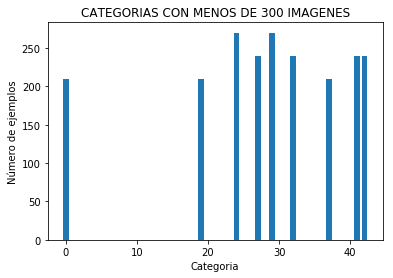


CATEGORIAS CON MENOS DE 500 IMAGENES: [0, 6, 16, 19, 20, 21, 22, 24, 27, 29, 30, 32, 34, 36, 37, 39, 40, 41, 42]

HAY UN TOTAL DE 19 CATEGORIAS CON MENOS DE 500 IMAGENES



En la siguiente figura analizamos las categorías con menos de 300 imágenes.



CATEGORIAS CON MENOS DE 300 IMAGENES: [0, 19, 24, 27, 29, 32, 37, 41, 42]

HAY UN TOTAL DE 9 CATEGORIAS CON MENOS DE 300 IMAGENES



Observamos que hay algunos grupos de categorías muy parecidos y que se encuentran subrepresentadas. Esto puede suponer un problema y una fuente de error en el modelo que debemos poder identificar. Tanto las categorías 34, 36, 37 y 39 como la 41 y 42 podrán inducir error de catalogación en el modelo.

### Análisis por categoría

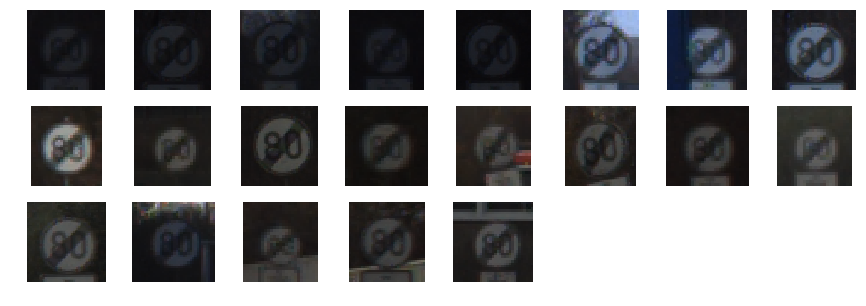
Antes de empezar con los modelos, es importante realizar un análisis del tipo de imágenes que nos encontraremos en cada categoría. En este tipo de análisis buscamos categorías que puedan incluir varias tipos de señales, peculiaridades de algunas categorías que puedan darnos pistas para construir nuestro modelo o incluso algún error que podemos detectar.

En la categoria 30, tenemos 450 ejemplos.



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

Las categorías 1-8 ya hemos observado que son muy parecidas, pero es importante notar que la categoría 6 no es una señal de límite de velocidad.



## Preparación de los datos

Antes de poder utilizar estos datos para entrenar modelos de Machine Learning es importante realizar algunos ajustes.

Los modelos de Machine Learning necesitan que los datos tengan el mismo tamaño, que se trate de un NumPy array y que el vector “X” tenga una única dimensión. Para ello preparamos la función “prepare\_X” y “prepare\_Y” [REFERENIA] .

El tamaño de imágenes que vamos a elegir puede influir sobre el rendimiento del modelo [REFERENIA] por el momento nosotros tomaremos como tamaño 32x32 que es un tamaño bastante estándar.

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:

El tamaño máximo de nuestras imágenes es: (225, 243, 3)

El tamaño mínimo de nuestras imágenes es: (25, 25, 3)

Observamos que el tamaño mínimo es 25x25, deberemos estudiar si esto afecta al rendimiento del modelo y si es necesario bajar el tamaño de todas las imágenes a 25x25.

Una vez redimensionadas las imágenes observamos que la función resize() también normaliza los valores RGB. Normalizar los valores muy recomendable para los modelos de Machine Learning [REFERENIA] .

## Modelo 1

El primero modelo que vamos a implementar se trata de una red neuronal completamente conectada de una sola capa igual al número de categorías de nuestro conjunto de datos. Este modelo extremadamente fácil nos sirve como Modelo Mínimo Viable, lo que nos permitirá identificar errores en el tratamiento de datos y empezar a implementar una metodología para entrenar al modelo.

### Código y resultados

Para construir nuestros modelos utilizaremos TensorFlow lo que nos permite no tener que entrar en la formulación de las matrices y nos evita muchos errores. Para este caso vamos a ejecutar el modelo dentro de una clase con el objetivo de reducir las líneas de código y facilitar su revisión.

**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)

El esquema en TensorFlow para construir los modelos siempre suele ser el mismo:

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

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 red neuronal completamente conectada, la función de activación es ReLu y vamos a utilizar un calsificador Softmax y para el cálculo del coste utilizaremos Cross Entropy.

Para el proceso de entrenamiento, introducimos 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 vamos a utilizar una función que hemos llamado “train”. Esta función 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.

**def** train(model, images, labels, test\_images, test\_labels, train\_count, imprimir=**True**):

*#Training loop*

costs=[]

**for** i **in** range(train\_count):

*#print(images.shape)*

*#print(labels.shape)*

**if** i % 20 == 0:

lossvalue = evaluate(model, test\_images, test\_labels, i, imprimir)

costs.append(lossvalue)

model.session.run(model.train,{

model.images:images,

model.labels:labels})

*#Final evaluation*

lossvalue=evaluate(model, test\_images, test\_labels,i)

costs.append(lossvalue)

**if** imprimir == **True**:

*# plot the cost*

plt.plot(np.squeeze(costs))

plt.ylabel('cost')

plt.xlabel('iterations (per tens)')

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** los

m1 = Model1()

train(m1, X\_train\_orig, Y\_train\_orig, X\_test\_orig, Y\_test\_orig, 1000 , imprimir=**True**)

Este Código nos devuelve las siguientes instancias:

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

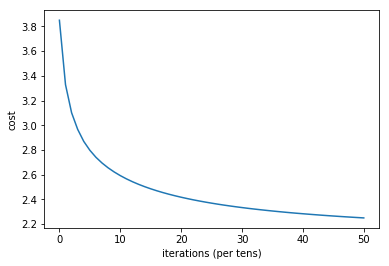
60, Loss: 2.896 Test accuracy: 0.417

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



Finalmente obtenemos la curva del cálculo del coste, que se asemeja mucho a la curva ideal: decrece constantemente y tiende a una asíntota horizontal.

Dado que es un modelo muy sencillo, la precisión final depende mucho de la inicialización. Observamos que si ejecutamos el modelo varias veces la precisión varía considerablemente, así la precisión final que hemos obtenido 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 como si ejecutamos el modelo 5 veces seguidas los resultados varían considerablemente.

**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

Para el análisis de los resultados obtenidos vamos a fijarnos en dos puntos básicos: los pesos obtenidos y la precisión por categoría.

#### Visualización de los pesos

Dado que nos encontramos ante un modelo de una sola capa es posible analizar el conjunto de pesos que representa cada categoría. Los pesos se encuentran dentro de la matriz de pesos o “W”, esta matriz debe tener el mismo número de categorías que de columnas y el mismo número de filas que el nº de datos de una imagen, en nuestro caso 32x32x3. Por lo tanto, la matriz de pesos debe ser de (3072,43).

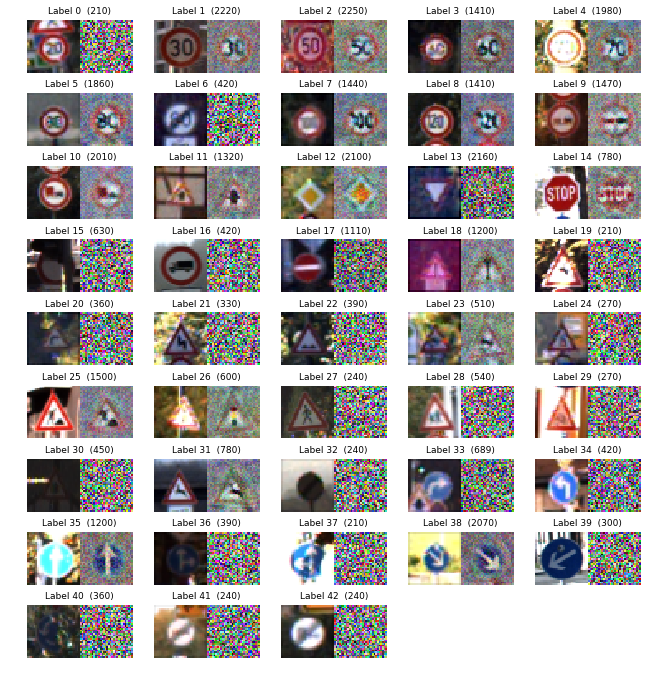
Vamos a imprimir cada una de las columnas que representa cada categoría para de esta forma poder ver que categorías son las mejor representadas y como afecta esto a los pesos del modelo.

**def** display\_weights(weights, limit):

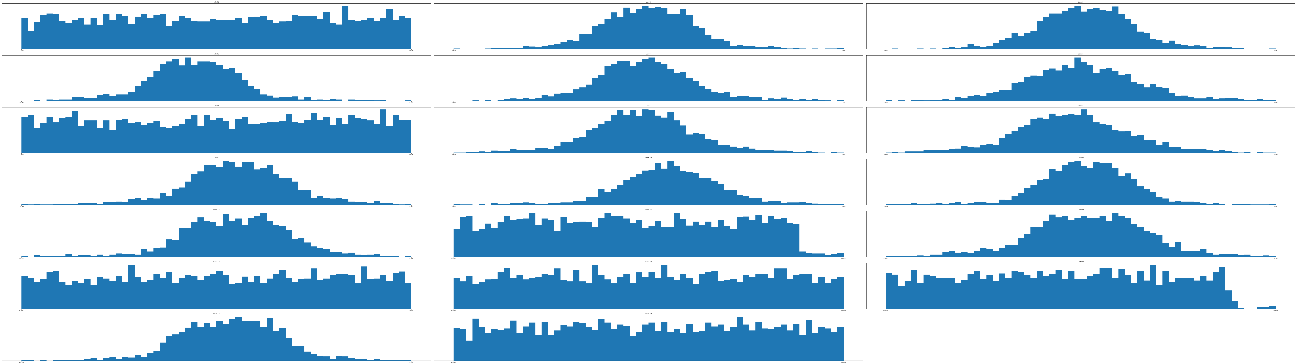
titles= ["Label **{}** (**{}**)".format(l,c) **for** l, c **in** enumerate(label\_counts)]

images = list(zip(label\_exemplars, weights.T.reshape(-1, 32, 32,3)))

display\_images(images[:limit], titles)

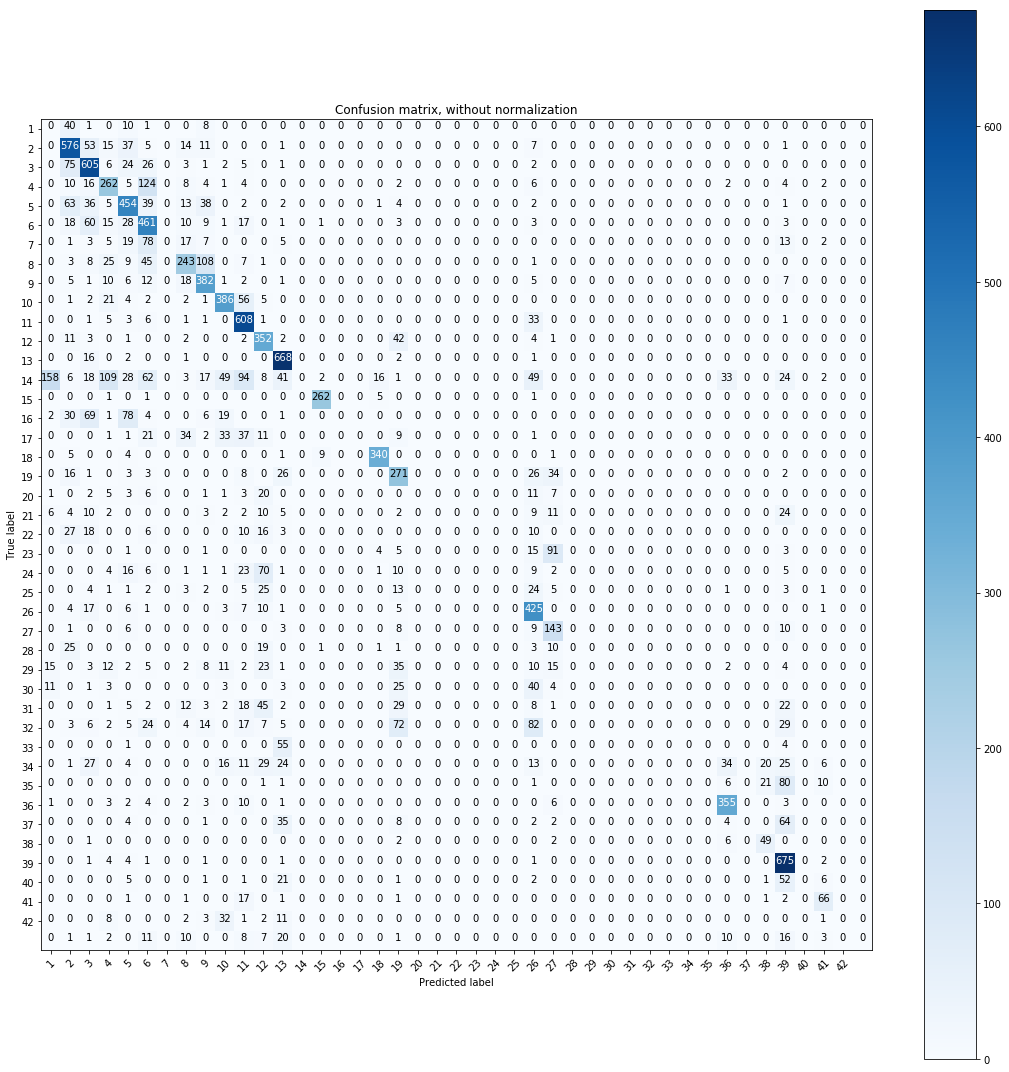


Observamos que aquellas categorías mejor representadas son las que tienen unos pesos más parecidos a las señales.



También podemos analizar los histogramas de los pesos, en la figura anterior vemos los histogramas de las 20 primeras categorías. Observamos que aquellos histogramas que tienen una distribución parecida a la Gaussiana son los de mejor rendimiento, en cambio aquellos con una distribución uniforme son aquellos que no han sufrido modificación respecto a su inicialización y por lo tanto tienen un rendimiento bajo.

#### Confussion matrix

Para un análisis más profundo vamos a visualizar una matriz de confusión. En esta matriz se representan los errores de clasificación entre categorías, de esta manera, podemos analizar si hay clases que se confunden mucho, en nuestro caso tenemos las imágenes de velocidad que pueden ser fácilmente confundidas.

En el caso ideal, tendríamos una matriz diagonal. Todos los valores que no están en la diagonal han sido mal clasificados. Para nuestro caso vemos que hay muchas categorías que no tienen ningún acierto, algunas de estas categorías han sido clasificadas erróneamente en otra categoría y en otras casos vemos que el modelo no ha conseguido clasificar ninguna imagen en esa categoría.

Observamos que las categorías 16-17; 20-25; 28-35; 37; 40; 42 y 43 no tienen ninguna imagen clasificada. Este echo puede ser indicación de error sin embargo si analizamos detalladamente los resultados obtenidos observamos que para estas categorías el model no clasifica ni correctamente ni erróneamente ninguna imagen.

#### Porcentajes de aciertos

Para poder entender mejor los resultados es recomendable no solo fijarse en la precisión del modelo en global 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 mostramos los resultados sobre una imagen a modo de ejemplo de cada señal.



## Modelo 2

En el segundo modelo, vamos a implementar algunos cambios para que utilizando una estructura parecida al primero, podamos corregir los problemas de las categorías menos representadas.

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

En la siguiente figura se muestra el código del modelo 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 definrla manualmente dado que desconocemos de una función similar en TensorFlow. Para la función de optimización tipo mini batch que vamos a implementar utilizaremos el mismo código que para el modelo 1, simplemente introduciremos una selección aleatoria de 50 imágenes previa para luego introducir esa selección en el modelo. El código utilizado se muestra a continuación.

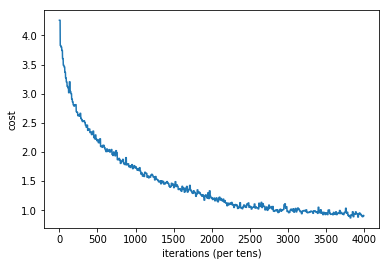
indexes = np.random.choice(np.arange(images.shape[0]), 50, replace=**False**)

model.session.run(model.train,

{model.images: images[indexes],

model.labels: labels[indexes]})

Este tipo de algoritmo afecta a la curva del coste. Dado que el entrenamiento del modelo no se hace con todas las imágenes de una vez sino que vamos introduciendo 50 imágenes por cada iteración la curva del coste varía mucho en cada iteración. Aunque se pueda observar algo de ruido en la curva, esta tiene que decrecer y tender a una asíntota horizontal.



Después de 4000 iteraciones observamos en la gráfica que el coste ya está estabilizado, la precisión final del modelo después de esas 4000 iteraciones es de 0.895 para el subconjunto de entrenamiento y 0.795 para el subconjunto de prueba.

Igual que el Modelo1 vamos a ejecutar este mismo modelo 6 veces para ver las variaciones de precisión entre ellas. Observamos 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

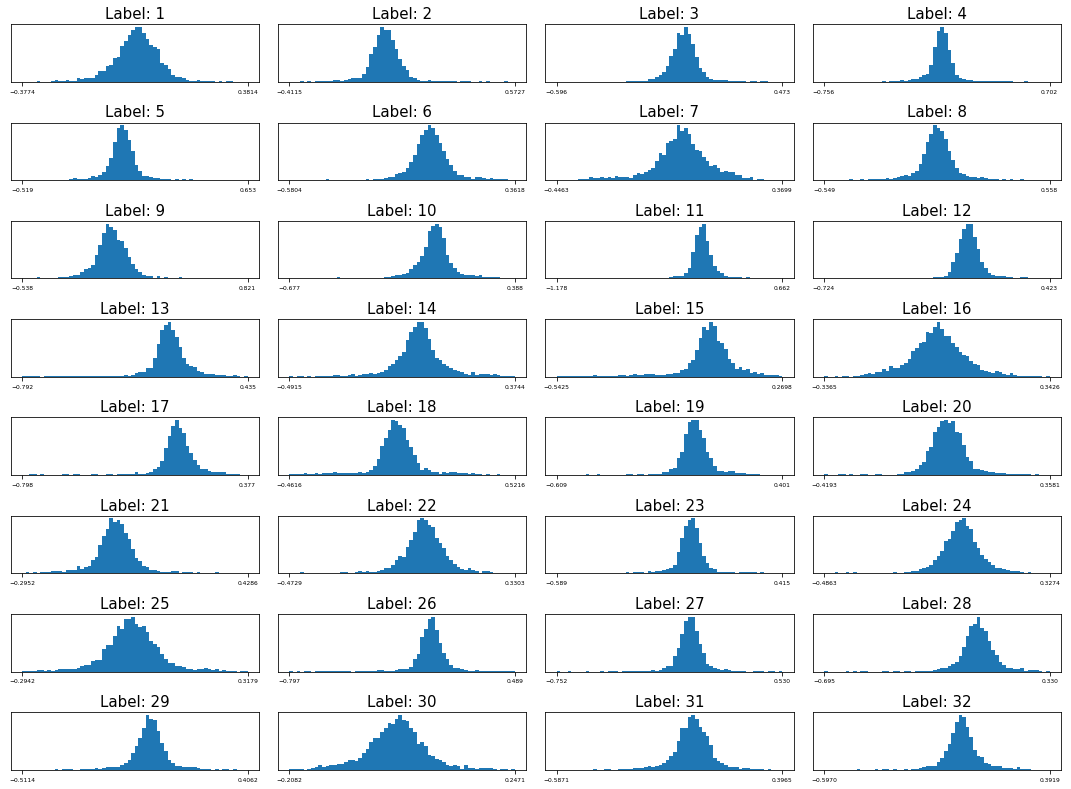
Para el análisis de resultados vamos a proceder de la misma forma que para el modelo 1, primero analizaremos los pesos del modelo para luego analizar los resultados y los errores.

#### Visualización de los pesos

En la siguiente figura podemos ver la representación de los pesos ya descrita en el Modelo 1 y lo primero que observamos respecto a los resultados del Modelo 1 es que todas las imágenes tienen unos pesos bastante parecidos a sus respectivas imágenes.



Los histogramas de los pesos para este modelo son los siguientes:

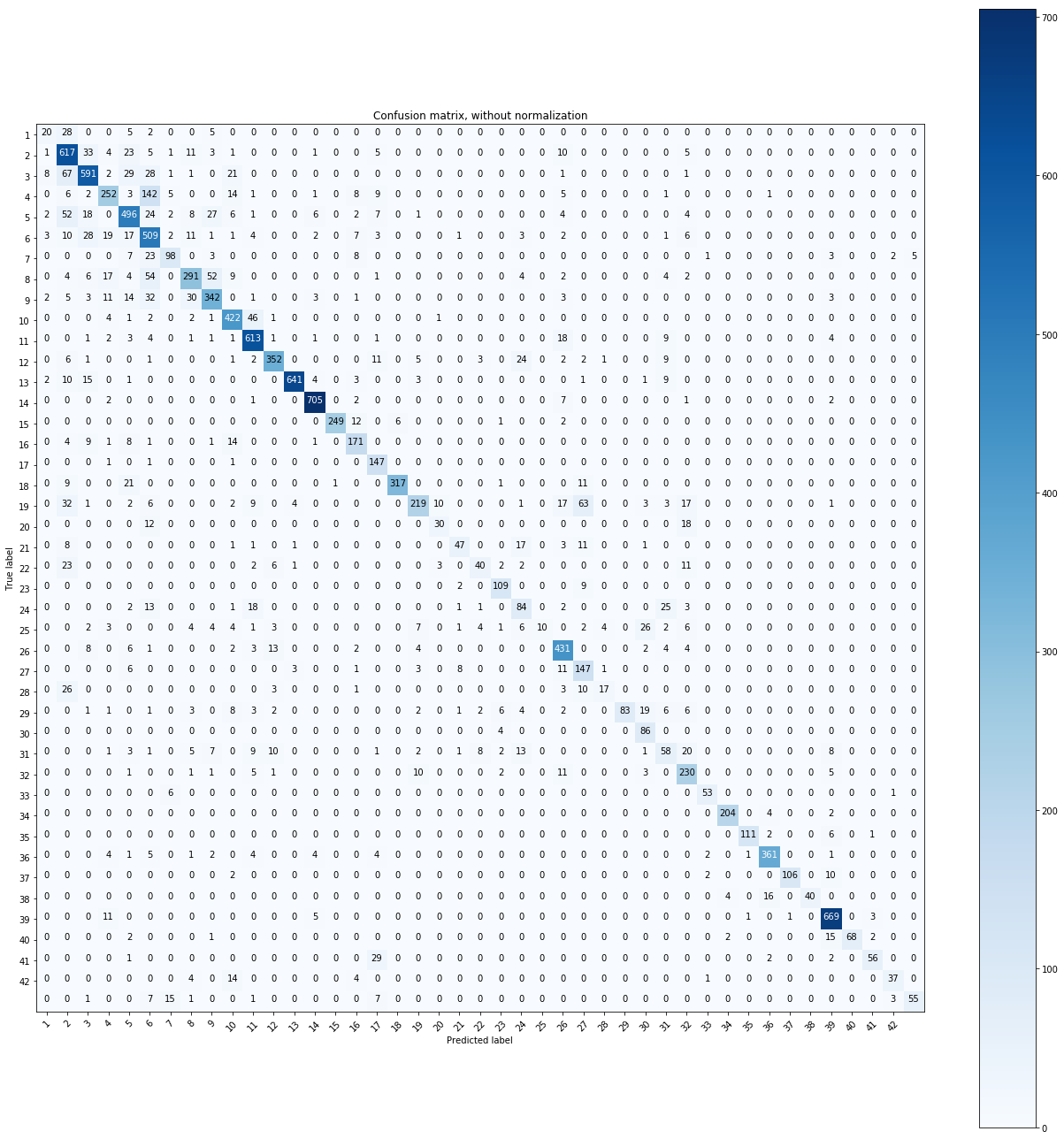


Como ya habíamos observado en la visualización de los pesos, este modelo se acerca más a la realidad de cada imagen. A diferencia del modelo 1 donde algunas categorías estaban representadas con un histograma uniforme, vemos que en este caso los histogramas son todos Gaussianos lo que quiere decir que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

#### Confussion matrix

Para este modelo, observamos que la confussion matrix es más diagonal que en el modelo 1 lo que nos indica una mejor calidad. Observamos también que todas las categorías tienen aciertos a diferencia del modelo 1.

Seguimos teniendo problemas en general para las categorías de límite de velocidad, donde existe mucho error de clasificación. De una forma más puntual hay parejas de categorías que también generan error: 19-17; 22-2; 12-24; 18-5.



#### Porcentaje de aciertos

En la siguiente figura observamos el porcentaje de aciertos por categoría.



De forma general hemos observado que este nuevo modelo no solo nos da mayor precisión global sobre el conjunto de datos, si no que también homogeniza esa precisión. Con este nuevo modelo hemos sido capaces de obtener una precisión sobre el subconjunto de prueba de un 70% y un mínimo de acierto respecto a las categorías de un 22%. Para estos mismos parámetros en el modelo 1 teníamos un 40% y un 0%.

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.

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

## Modelo 3

Con el segundo modelo, mejoramos todo lo posible la precisión de un modelo de redes neuronales con una sola capa. Para este tercer modelo, dado que no nos es posible mejorar la precisión de un modelo con una sola capa vamos a implementar una estructura con varias capas.

Para este caso utilizaremos una estructura con tres capas ocultas y el siguiente esquema: [FC-L - RL]x2 + FC – SOFTMAX. Las dos primeras capas serán completamente conectadas y el tipo de activación será tipo Leaky-ReLU, 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.

Una vez decididos los parámetros del modelo nos queda definir el tamaño de las capas intermedias. Los consejos generales para este tipo de redes neuronales son elegir capas que decrezcan de forma escalonada y que el tamaño de la última capa coincida con el número de categorías.

Para un primer intento vamos 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()

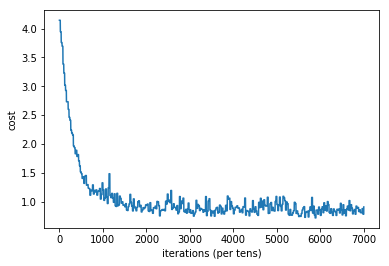
*#Create session*

self.session=tf.Session()

*#Run Initialization*

self.session.run(self.init)

La precisión global de este modelo varía entre 0.92 y 0.86 esta variación es debida al algoritmo de optimización de tipo mini batches. Al introducir un nuevo subconjunto de 50 imágenes al modelo con cada iteración los resultados varían dentro de unos límites como podemos ver en la figura de la función de coste que adjuntamos.



Observamos que la precisión con este modelo con tres capas no mejora respecto al modelo 1 con una sola capa, esto puede deberse a que los tamaños de capa seleccionados no han sido optimizados. Vamos a realizar un análisis de sensibilidad de estos parámetros para conseguir una mayor precisión global del sistema.

### Análisis de sensibilidad

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

Vamos a modificar el código del modelo para conseguir. Vamos a definir 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 loop tipo for que ejecute el modelo cada vez y rectifique los valores de n1 y n2, dado que n3 ya hemos comentado que debe ser fija e igual al número de categorías.

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)

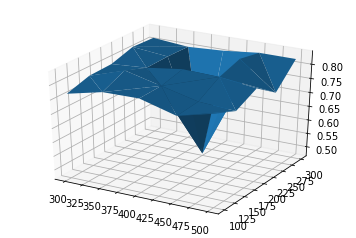
loss = train\_minibatch(m3, X\_train\_orig, Y\_train\_orig, X\_test\_orig, Y\_test\_orig, 3000, True)

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:



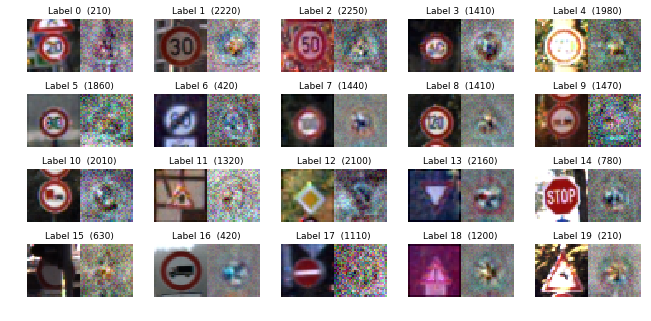
ACTUALIZAR CON NUEVO TUNNING Y MAS PRUEBAS.

### Análisis de resultados

Vamos a analizar los resultados de la misma forma que para los modelos anteriores.

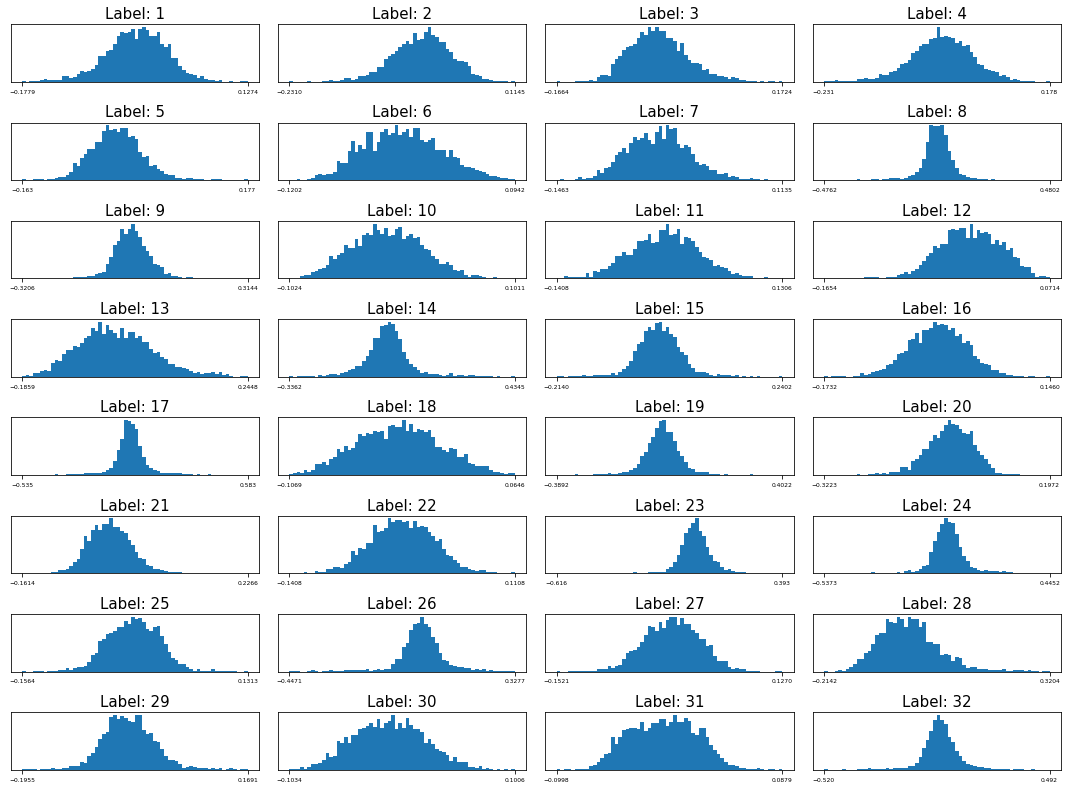
#### Visualización de los pesos

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.



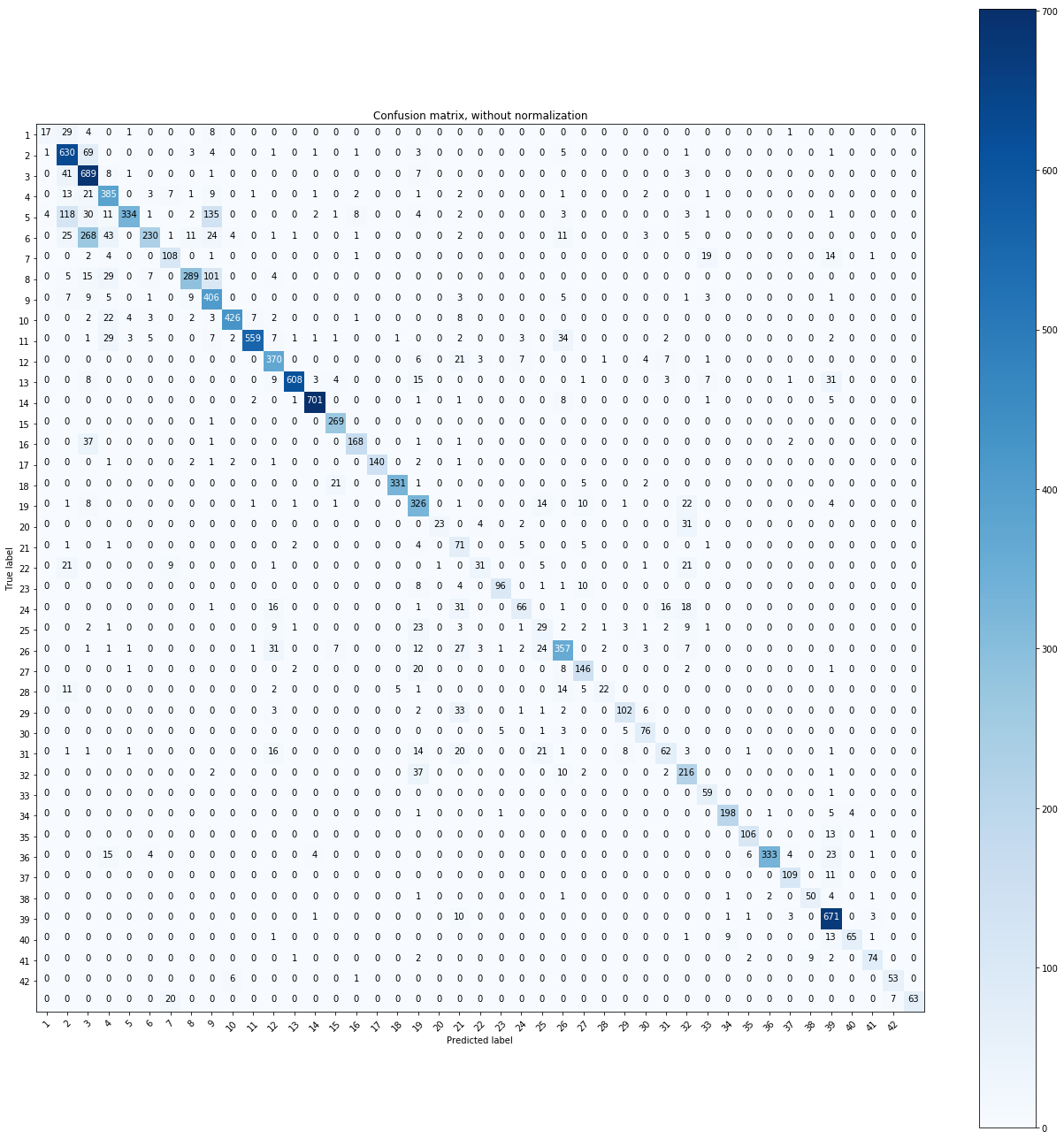
Observamos en la visualización de los pesos como, aunque el modelo de forma general consigue una mayor precisión, los pesos de la primera capa tienen una calidad inferior a los del modelo 2. Como ya hemos explicado, esto se debe a que la información se reparte entre las tres capas del modelo y no solo en la primera.

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



#### Confussion matrix

La matriz de confusiones es la siguiente:



Observamos de forma general, que la matriz es más diagonal que en el modelo 1, lo que significa un menor error.

#### Porcentaje de aciertos

En el porcentaje de acierto de cada categoría podemos observar como la categoría con el menor acierto ha mejorado respecto al modelo 2. En el modelo 2 teníamos un mínimo de acierto del 20%, con este nuevo modelo hemos conseguido un 30% de acierto en la peor categoría.



En este nuevo modelo hemos la precisión global obtenida es de 92%, conseguimos mejorar muy poco la precisión respecto al modelo 2 que era del 91%. Sin embargo, conseguimos mejorar en la precisión dentro de las categorías.

TUNING DE HIPERPARAMETROS

## 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 del 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)

Como siempre definimos el modelo 4 como una clase y dentro de esta clase definimos dos funciones para las capas convolucionales y para las capas tipo pool max. 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")

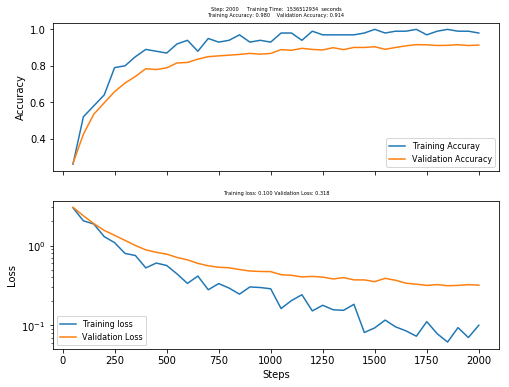
ax2.set\_ylabel("Loss")

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.



### Análisis de sensibilidad

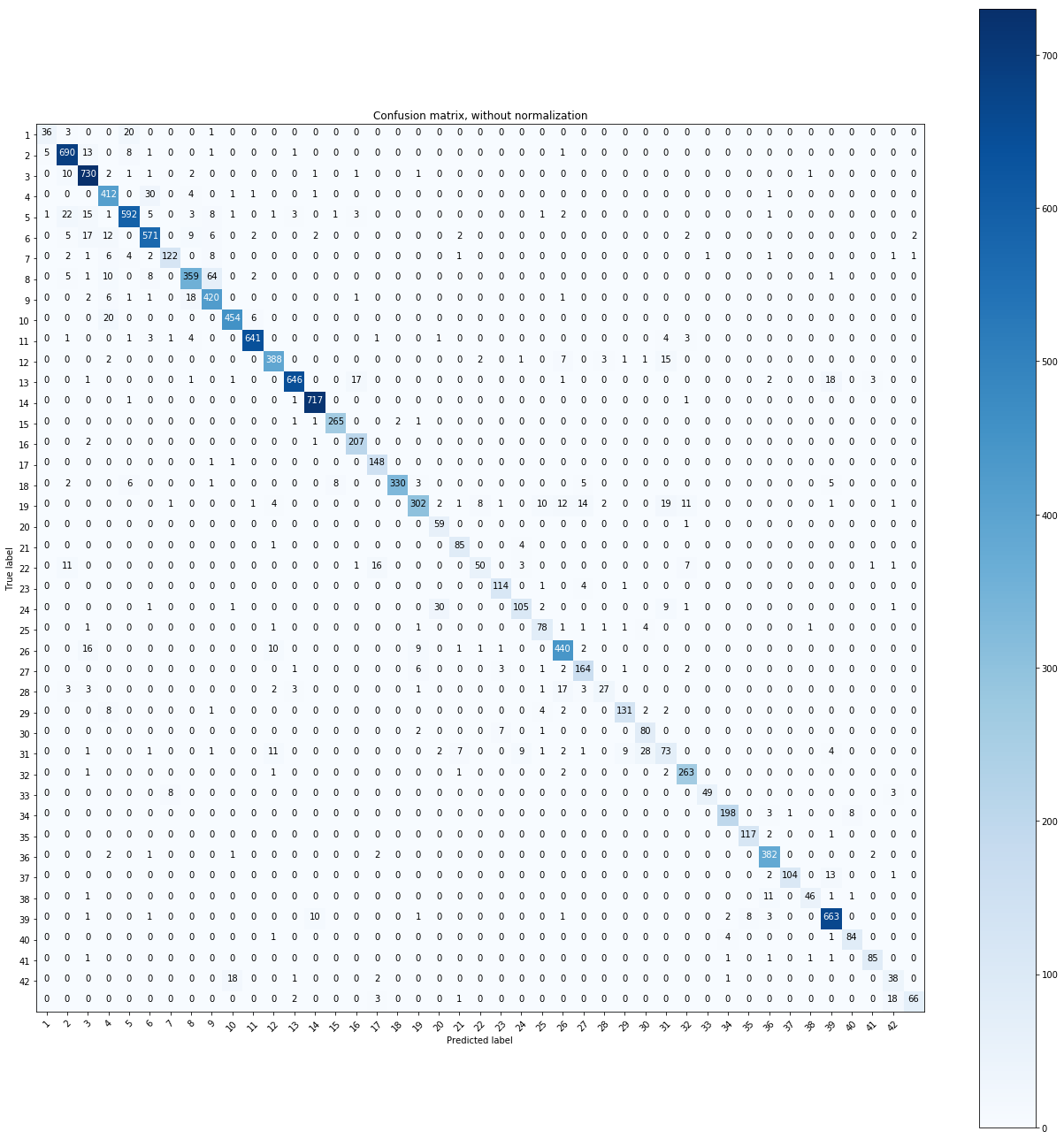
Al igual que para el modelo 3 será necesario realizar un análisis de sensibilidad sobre el tamaño de las tres primeras capas.

### Análisis de resultados

Para este modelo, no vamos a poder visualizar los pesos dado que se trata de una estructura mucho más complicada. Analizaremos los resultados con la confussion matrix y el porcentaje de aciertos por categoría.

#### Confussion matrix

Observamos 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

Observamos 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.



# Bibliografía

|  |  |
| --- | --- |
| [1] | 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. |