Tarea 1 Redes Neuronales Artificiales

Andrea Figueroa Retamal
Alejandro Sazo Gómez

September 12, 2016

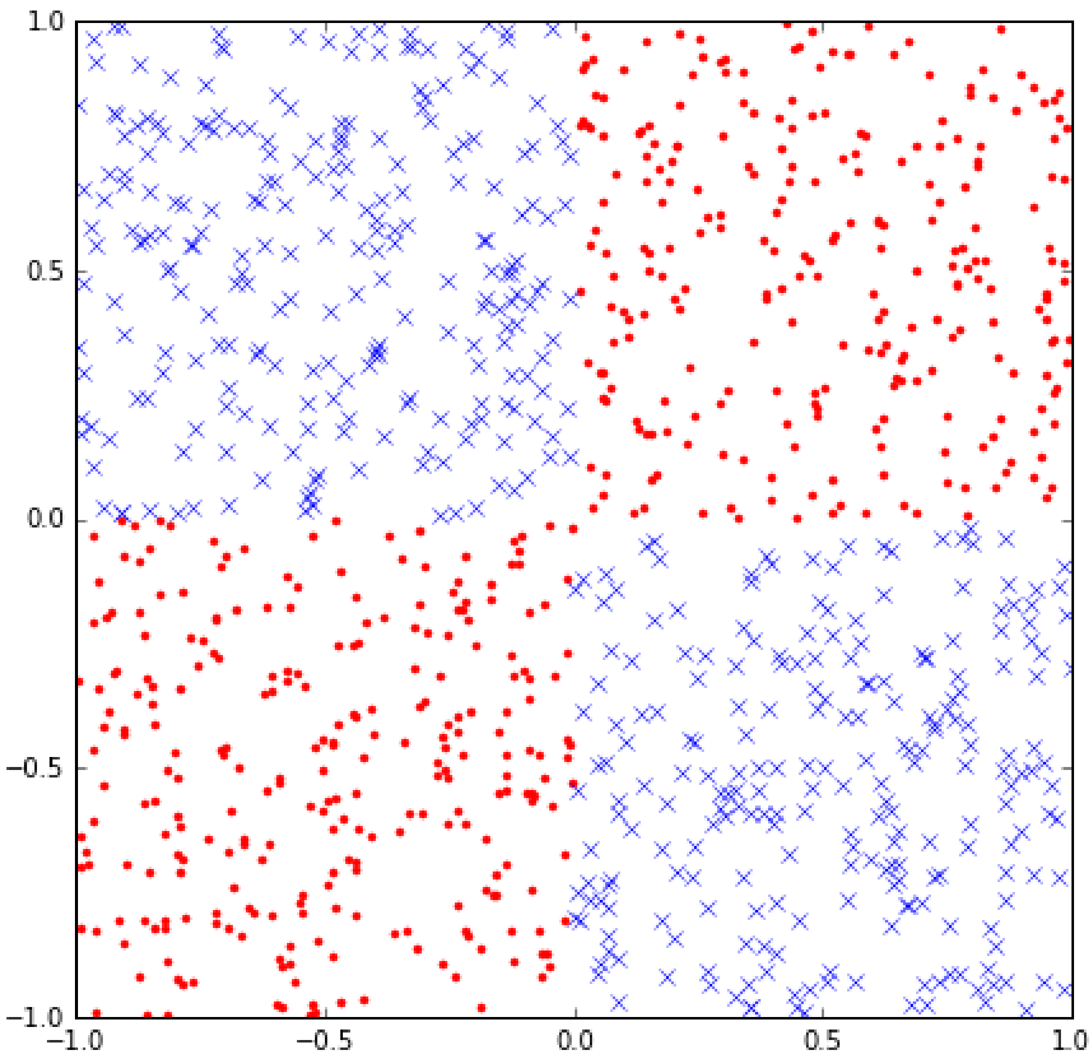
1 Ejercicio 1

(a) Generación de data aleatoria que represente la función lógica xor. Se generan 1000 datos de prueba y 1000 de entrenamiento.

```
smallIn [2]: # Importar Librerías
        import numpy as np
        import matplotlib.pyplot as plt
        from keras.models import Sequential
        from keras.layers.core import Dense, Dropout, Activation
        from keras.optimizers import SGD
```

```
In [2]: # Guardar semilla para numeros aleatorios
        seed = 21
        np.random.seed(seed)

        def generate_data(n):

            # Lista para guardar datos etiquetados
            output = list()

            # Generación de n tuplas aleatorias
            input = 2 * np.random.random_sample((n,2)) - 1
            # Asignación datos dependiendo del cuadrante
            for i in input:
                # Cuadrante 1
                if i[0] > 0 and i[1] > 0:
                    output.append(0)
                # Cuadrante 2
                elif i[0] < 0 and i[1] > 0:
                    output.append(1)
                # Cuadrante 3
                elif i[0] < 0 and i[1] < 0:
                    output.append(0)
                # Cuadrante 4
                elif i[0] > 0 and i[1] < 0:
                    output.append(1)
            return input, output

        (x_training, y_training) = generate_data(1000)
        (x_test, y_test) = generate_data(1000)
```

```python
# Plot de datos de entrenamiento
%matplotlib inline
plt.figure(figsize=(7,7))
set1 = x_training[np.logical_and(x_training[:,0] < 0, x_training[:,1] < 0)]
set2 = x_training[np.logical_and(x_training[:,0] < 0, x_training[:,1] > 0)]
set3 = x_training[np.logical_and(x_training[:,0] > 0, x_training[:,1] > 0)]
set4 = x_training[np.logical_and(x_training[:,0] > 0, x_training[:,1] < 0)]
set1 = np.concatenate((set1, set3), axis=0)
set2 = np.concatenate((set2, set4), axis=0)
plt.plot(set1[:,0], set1[:,1], 'r.')
plt.plot(set2[:,0], set2[:,1], 'bx')
plt.show()
```



El problema se considera XOR, o or exclusivo debido a que [(-),(-)] y [(+),(+)] son etiquetados con con circulos y [(-),(+)] y [(+),(-)] son etiquetados con cruces, podemos compararlo con el or exclusivo que obtiene 0 para [1,1] y [0,0] y obtiene 1 para [1,0] y [0,1]

(b) Generación de una neurona. Ha sido entrenada con 1000 epochs.

```
In [3]: # Creación de una neurona
        model = Sequential()

        # Dimensión input = 1, Dimensión output = 2, función de activación es Relu
        model.add(Dense(output_dim=1, input_dim=2, init="normal"))
        model.add(Activation("sigmoid"))
        model.compile(loss='mean_squared_error', optimizer='sgd', metrics=['accuracy'])

        print "Neurona inicializada"

        # Entrenar a la neurona
        model.fit(x_training, y_training, nb_epoch=1000,verbose=0)

        print "Neurona entrenada"

        # Evaluar la neurona
        loss_and_metrics = model.evaluate(x_test, y_test, batch_size=1000)
        print "Loss: "
        print loss_and_metrics[0]
        print "Accuracy: "
        print loss_and_metrics[1]*100
```

```
Neurona inicializada
Neurona entrenada
1000/1000 [==============================] - 0s
Loss:
0.249790877104
Accuracy:
51.4999985695
```

```
In [4]: print round(model.predict(np.array([-1,-1]).reshape(1,2))[0][0],4)
        print round(model.predict(np.array([1,1]).reshape(1,2))[0][0],4)
        print round(model.predict(np.array([-1,1]).reshape(1,2))[0][0],4)
        print round(model.predict(np.array([1,-1]).reshape(1,2))[0][0],4)
```

```
0.467
0.4841
0.4714
0.4796
```

Como se puede apreciar, la neurona a pesar de la cantidad de datos de entrenamiento utilizados, no es capaz de aprender la función XOR arrojando resultados inconsistentes, siendo incapaz de clasificar de forma determinante los ejemplos de testing en alguna clase. Esto es debido a que las clases del set no son linealmente separable.

(c) Una arquitectura de perceptron multicapa con 8 neuronas en su capa oculta permite aprender XOR de forma efectiva.

```
In [5]: from keras.models import Sequential
        from keras.layers.core import Dense, Activation

        xor = Sequential()
        # Dimensión input = 2, Dimensión output = 1, función de activación es Relu
```

```
xor.add(Dense(8, input_dim = 2, activation = "relu"))
xor.add(Dense(1, activation = "sigmoid"))
xor.compile(loss='mean_squared_error', optimizer='sgd', metrics=['accuracy'])

print "Red inicializada"

# Entrenar a la red
xor.fit(x_training, y_training, nb_epoch=1000, verbose=0)

print "Red entrenada"

# Evaluar la red
evaluacion = xor.evaluate(x_test, y_test, batch_size=1000)
print "Loss: "
print evaluacion[0]
print "Accuracy: "
print evaluacion[1]*100
```

```
Red inicializada
Red entrenada
1000/1000 [==============================] - 0s
Loss:
0.0562019199133
Accuracy:
93.4000015259
```

```
In [6]: print xor.predict_classes(np.array([-1,-1]).reshape(1,2))[0][0]
        print xor.predict_classes(np.array([1,1]).reshape(1,2))[0][0]
        print xor.predict_classes(np.array([-1,1]).reshape(1,2))[0][0]
        print xor.predict_classes(np.array([1,-1]).reshape(1,2))[0][0]
```

```
1/1 [==============================] - 0s
0
1/1 [==============================] - 0s
0
1/1 [==============================] - 0s
1
1/1 [==============================] - 0s
1
```

Se puede observar que con un perceptrón multicapa se obtiene un accuracy del 93% pudiendo clasificar correctamente los datos.

2 Ejercicio 2

(a) Construcción del dataframe para el set Boston Housing.

```
In [38]: import pandas as pd
         url = 'http://mldata.org/repository/data/download/csv/regression-datasets-housing/
         df = pd.read_csv(url, sep=',',header=None, names=['CRIM', 'ZN', 'INDUS', 'CHAS',
         'RM', 'AGE','DIS','RAD','TAX','PTRATIO','B','LSTAT','MEDV'])
         from sklearn.cross_validation import train_test_split
         df_train, df_test = train_test_split(df,test_size=0.25, random_state=0)
```

En las lineas 5 a 7 se puede observar que este set se divide para generar el training set y el testing set. Del total de datos, el 25% se deja aparte para pruebas y el restante 75% permanece para entrenamiento.

(b) Descripción del dataset.

Gracias a la herramienta pandas podemos obtener la información descriptiva del set de datos, ya sea la dimensión tipo de datos, columnas, entre otros.

```
In [39]: df.shape
         df.info()
         df.describe()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 506 entries, 0 to 505
Data columns (total 14 columns):
CRIM       506 non-null float64
ZN         506 non-null int64
INDUS      506 non-null float64
CHAS       506 non-null int64
NOX        506 non-null float64
RM         506 non-null float64
AGE        506 non-null float64
DIS        506 non-null float64
RAD        506 non-null int64
TAX        506 non-null int64
PTRATIO    506 non-null int64
B          506 non-null float64
LSTAT      506 non-null float64
MEDV       506 non-null float64
dtypes: float64(9), int64(5)
memory usage: 59.3 KB
```

```
Out[39]:             CRIM          ZN       INDUS        CHAS         NOX          RM  \
         count 506.000000  506.000000  506.000000  506.000000  506.000000  506.000000
         mean    3.613524   11.347826   11.136779    0.069170    0.554695    6.284634
         std     8.601545   23.310593    6.860353    0.253994    0.115878    0.702617
         min     0.006320    0.000000    0.460000    0.000000    0.385000    3.561000
         25%     0.082045    0.000000    5.190000    0.000000    0.449000    5.885500
         50%     0.256510    0.000000    9.690000    0.000000    0.538000    6.208500
         75%     3.677082   12.000000   18.100000    0.000000    0.624000    6.623500
         max    88.976200  100.000000   27.740000    1.000000    0.871000    8.780000

                     AGE         DIS         RAD         TAX     PTRATIO           B  \
         count 506.000000  506.000000  506.000000  506.000000  506.000000  506.000000
         mean   68.574901    3.795043    9.549407  408.237154   18.083004  356.674032
         std    28.148861    2.105710    8.707259  168.537116    2.280574   91.294864
         min     2.900000    1.129600    1.000000  187.000000   12.000000    0.320000
         25%    45.025000    2.100175    4.000000  279.000000   17.000000  375.377500
         50%    77.500000    3.207450    5.000000  330.000000   19.000000  391.440000
         75%    94.075000    5.188425   24.000000  666.000000   20.000000  396.225000
         max   100.000000   12.126500   24.000000  711.000000   22.000000  396.900000

                   LSTAT        MEDV
         count 506.000000  506.000000
```

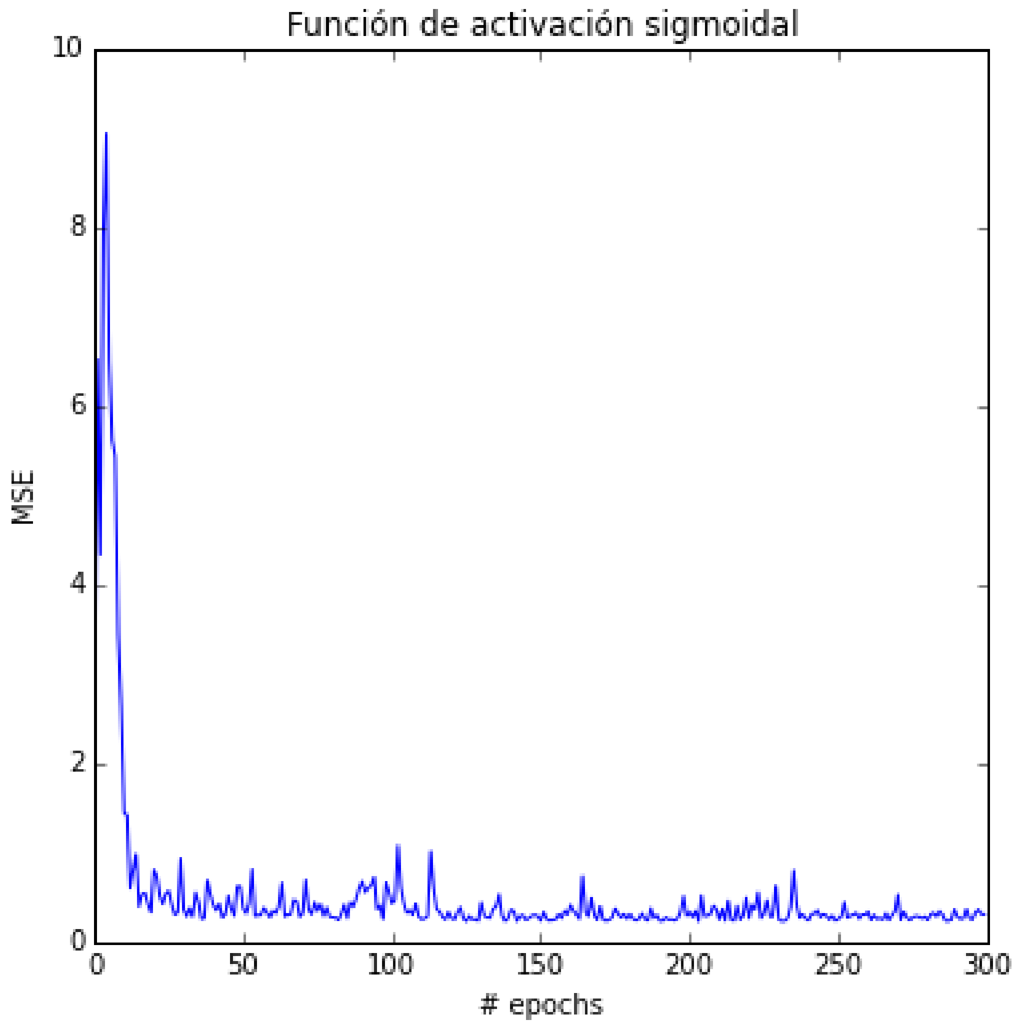
```
mean    12.653063   22.532806
std      7.141062    9.197104
min      1.730000    5.000000
25%      6.950000   17.025000
50%     11.360000   21.200000
75%     16.955000   25.000000
max     37.970000   50.000000
```

c) Normalización de datos. Este procedimiento es necesario para evitar cualquier clase de problemas con la convergencia de nuestra función de optimización, pues es posible que debido a los diversos rangos de datos la convergencia favorezca a ciertos valores, deteniéndose el algoritmo en un punto donde se ha aprendido mal sobre el training set y no se tenga capacidad de generalización.

```
In [40]: from sklearn.preprocessing import StandardScaler
         scaler = StandardScaler().fit(df_train)
         X_train_scaled = pd.DataFrame(scaler.transform(df_train),columns=df_train.columns)
         X_test_scaled = pd.DataFrame(scaler.transform(df_test),columns=df_test.columns)
         y_train_scaled = X_train_scaled.pop('MEDV')
         y_test_scaled = X_test_scaled.pop('MEDV')
```

d) Gráfico de MSE versus número de epochs utilizados para entrenar para red FF de 3 capas, 200 unidades ocultas y activación sigmoidal entrenada con SGD con parámetros η=0.02 y 300 epochs de entrenamiento.

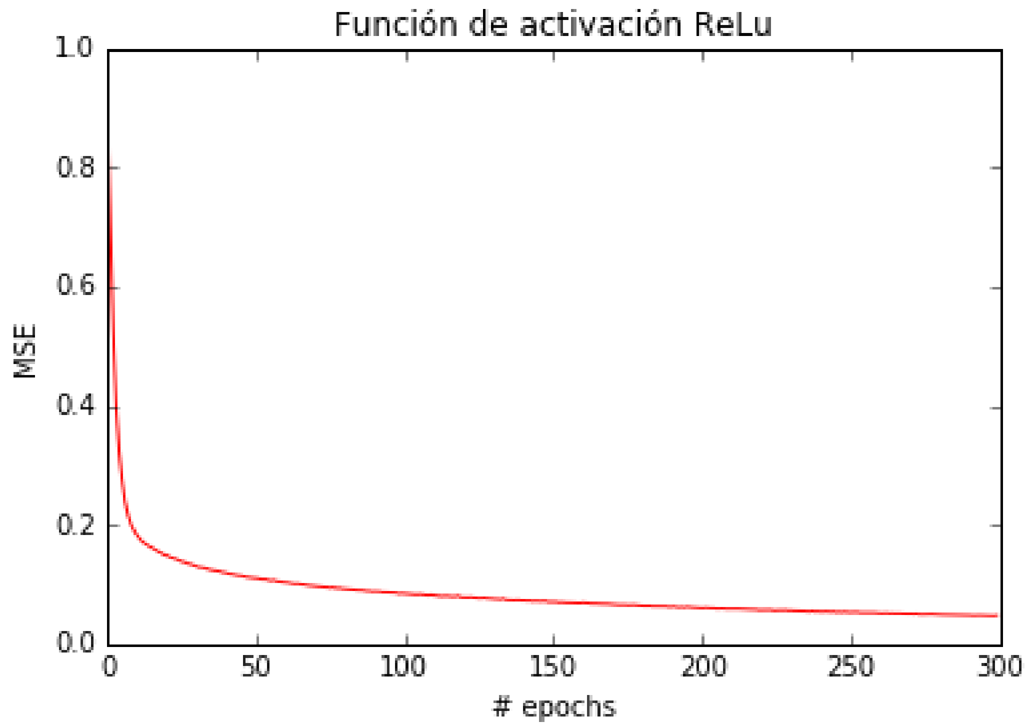
```
In [41]: def generate_model(optimizer, activation):
             model = Sequential()
             model.add(Dense(200, input_dim=X_train_scaled.shape[1], init='uniform'))
             model.add(Activation(activation))
             model.add(Dense(1, init='uniform'))
             model.add(Activation('linear'))

             model.compile(optimizer=optimizer,loss='mean_squared_error')

             return model

         sgd = SGD(lr=0.02)
         model_d = generate_model(sgd,"sigmoid")

         hist = model_d.fit(X_train_scaled.as_matrix(), y_train_scaled.as_matrix(),
              nb_epoch=300, verbose=0,
              validation_data=(X_test_scaled.as_matrix(), y_test_scaled.as_matrix()))


         %matplotlib inline
         epochs = np.arange(300)
         plt.figure(figsize=(6,6))
         plt.plot(epochs, hist.history['loss'], 'b-')
         plt.title(u"Función de activación sigmoidal")
         plt.xlabel("# epochs")
         plt.ylabel("MSE")
         plt.show()
```



El error obtenido en un comienzo es muy alto pero al aumentar el número de epochs éste disminuye considerablemente convergiendo a un número muy bajo pero a su vez la convergencia posee comportamiento oscilatorio.

e) Variar función de activación cambiandola por ReLu.

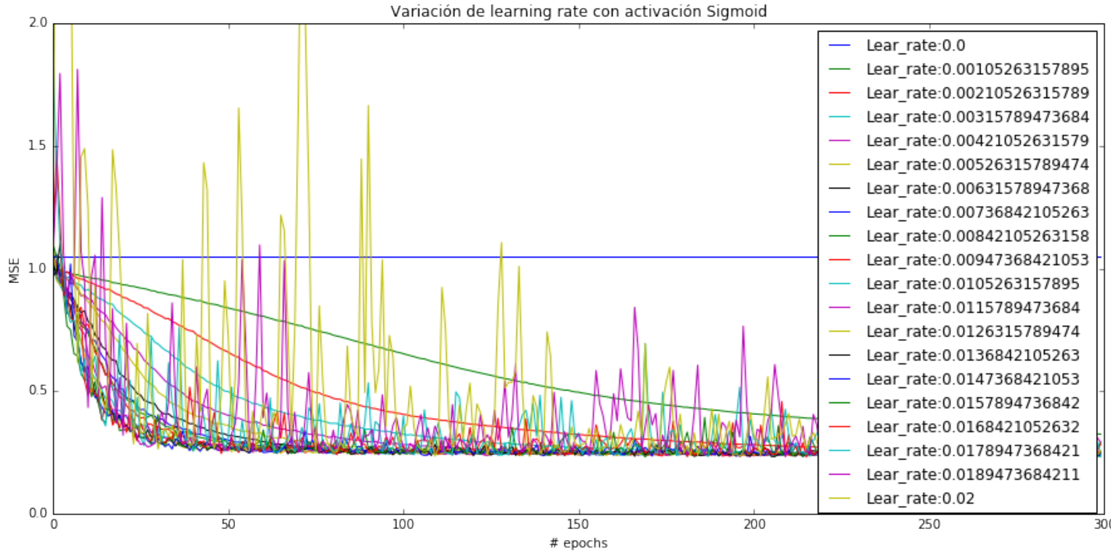
```
In [6]: sgd = SGD(lr=0.02)
        model2 = generate_model(sgd, "relu")

        hist2 = model2.fit(X_train_scaled.as_matrix(), y_train_scaled.as_matrix(),
                           nb_epoch=300, verbose=0,
                           validation_data=(X_test_scaled.as_matrix(), y_test_scaled.as_mat

        %matplotlib inline
        epochs = np.arange(300)
        plt.plot(epochs, hist2.history['loss'], 'r-')
        plt.title(u"Función de activación ReLu")
        plt.xlabel("# epochs")
        plt.ylabel("MSE")
        plt.show()
```



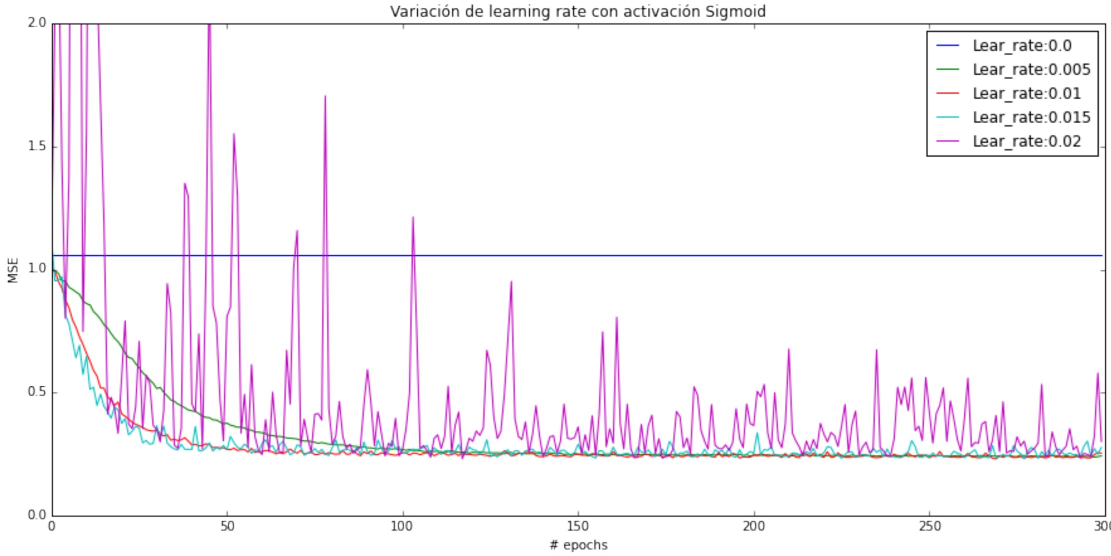
Podemos observar que con la función de activación ReLu se llega a un menor error cuadrático que con Sigmoid, además la convergencia es mucho más estable.

f) Variar learning rate

```
In [13]: n_lr = 20
         lear_rate = np.linspace(0,0.02,n_lr)

         %matplotlib inline
         epochs = np.arange(300)
         plt.figure(figsize=(15,7))
         for n, i in enumerate(lear_rate):
             sgd = SGD(lr=i)
             model = generate_model(sgd, "sigmoid")

             hist = model.fit(X_train_scaled.as_matrix(), y_train_scaled.as_matrix(),
                              nb_epoch=300, verbose=0,
                              validation_data=(X_test_scaled.as_matrix(), y_test_scaled.as_ma
             plt.plot(epochs, hist.history['loss'], label="Lear_rate:"+str(i))

         plt.title(u"Variación de learning rate con activación Sigmoid")
         plt.xlabel("# epochs")
         plt.ylabel("MSE")
         plt.ylim([0, 2])
         plt.legend()
         plt.show()
```



In [14]:
```
#Disminución de la cantidad de learning rates a probar con objetivo de una mejor 

n_lr = 5
lear_rate = np.linspace(0,0.02,n_lr)

%matplotlib inline
epochs = np.arange(300)
plt.figure(figsize=(15,7))
for n, i in enumerate(lear_rate):
    sgd = SGD(lr=i)
    model = generate_model(sgd, "sigmoid")

    hist = model.fit(X_train_scaled.as_matrix(), y_train_scaled.as_matrix(),
                     nb_epoch=300, verbose=0,
                     validation_data=(X_test_scaled.as_matrix(), y_test_scaled.as_ma
    plt.plot(epochs, hist.history['loss'], label="Lear_rate:"+str(i))

plt.title(u"Variación de learning rate con activación Sigmoid")
plt.xlabel("# epochs")
plt.ylabel("MSE")
plt.ylim([0, 2])
plt.legend()
plt.show()
```



Se puede observar que a medida se aumenta el learning rate la convergencia va cambiando, para lear_rate = 0 no aprende, pero mientras va aumentando se acerca de a poco a un error más pequeño debido a que aprende de forma lenta pero estable, a medida aumenta la el learning rate pasa a ser más oscilatoria, como se observa en lear_rate = 0.02 donde el resultado es muy oscilatorio.

g) Validación cruzada variando el número de folds

```
In [14]: from sklearn import cross_validation

         Xm = X_train_scaled.as_matrix()
         ym = y_train_scaled.as_matrix()
         mse_cvs1 = []
         mse_cvs2 = []
         for nfold in [5, 10]:
             kfold = cross_validation.KFold(len(Xm), nfold)
             cvscores1 = []
             cvscores2 = []
             for i, (train, val) in enumerate(kfold):
                 # create models
                 sgd = SGD(lr=0.02)
                 model1 = generate_model(sgd, 'sigmoid')
                 model2 = generate_model(sgd, 'relu')
                 # Fit the models
                 model1.fit(Xm[train], ym[train], nb_epoch=300, verbose=0)
                 model2.fit(Xm[train], ym[train], nb_epoch=300, verbose=0)
                 # evaluate the models
                 # sigmoid score
                 scores1 = model1.evaluate(Xm[val], ym[val])
                 # relu score
                 scores2 = model2.evaluate(Xm[val], ym[val])
                 # Store values
                 cvscores1.append(scores1)
                 cvscores2.append(scores2)
             mse_cvs1.append(np.mean(cvscores1))
```

```
            mse_cvs2.append(np.mean(cvscores2))

        print "Results for sigmoid and CV"
        print mse_cvs1
        print "Results for relu and CV"
        print mse_cvs2
```

```
32/37 [========================>...] - ETA: 0sResults for sigmoid and CV
[0.30503241192562536, 0.30517473007654333]
Results for relu and CV
[0.13131426443969996, 0.12684021550992433]
```

```
In [19]: model1 = generate_model(SGD(lr=0.02), "sigmoid")
         model2 = generate_model(SGD(lr=0.02), "relu")
         model1.fit(X_train_scaled.as_matrix(), y_train_scaled.as_matrix(), nb_epoch=300, v
         model2.fit(X_train_scaled.as_matrix(), y_train_scaled.as_matrix(), nb_epoch=300, v
         score_sigmoid = model1.evaluate(X_test_scaled.as_matrix(), y_test_scaled.as_matri
         score_relu = model2.evaluate(X_test_scaled.as_matrix(), y_test_scaled.as_matrix()
         print "Result for sigmoid in testing set"
         print score_sigmoid
         print "Result for relu in testing set"
         print score_relu
```

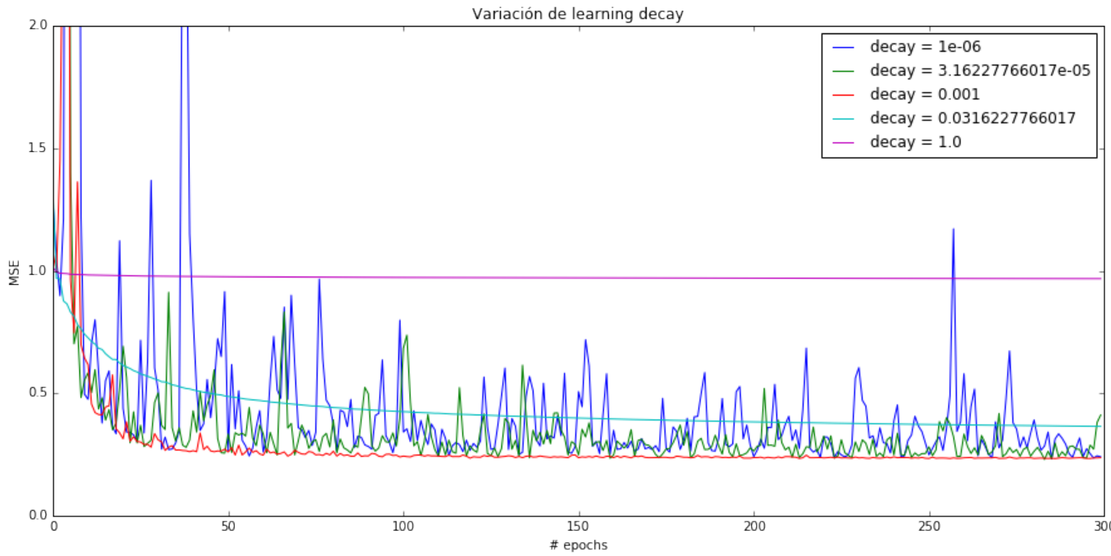
```
 32/127 [======>...] - ETA: 0sResult for sigmoid in testing set
0.412919790374
Result for relu in testing set
0.183559518571
```

Los resultados para el MSE utilizando función de activación sigmoid muestran errores en cross validation de 0.36 para 5 folds y 0.28 para 10 folds, mientras que los errores utilizando función relu bajan a aproximadamente 0.13 en ambos casos. La estimación con función de activación relu es más confiable y predijo de buena forma el comportamiento del testing set.

h) Entrenar el modelo en d) usando progressive decay

```
In [15]: n_decay = 5
         lear_decay = np.logspace(-6,0,n_decay)

         %matplotlib inline
         epochs = np.arange(300)
         plt.figure(figsize=(15,7))

         for i in lear_decay:
             sgd = SGD(lr=0.02, decay=i)
             model = generate_model(sgd, "sigmoid")

             hist = model.fit(X_train_scaled.as_matrix(), y_train_scaled.as_matrix(),
                              nb_epoch=300, verbose=0,
                              validation_data=(X_test_scaled.as_matrix(), y_test_scaled.as_ma
             plt.plot(epochs, hist.history['loss'], label="decay = "+str(i))

         plt.title(u"Variación de learning decay")
         plt.xlabel("# epochs")
```

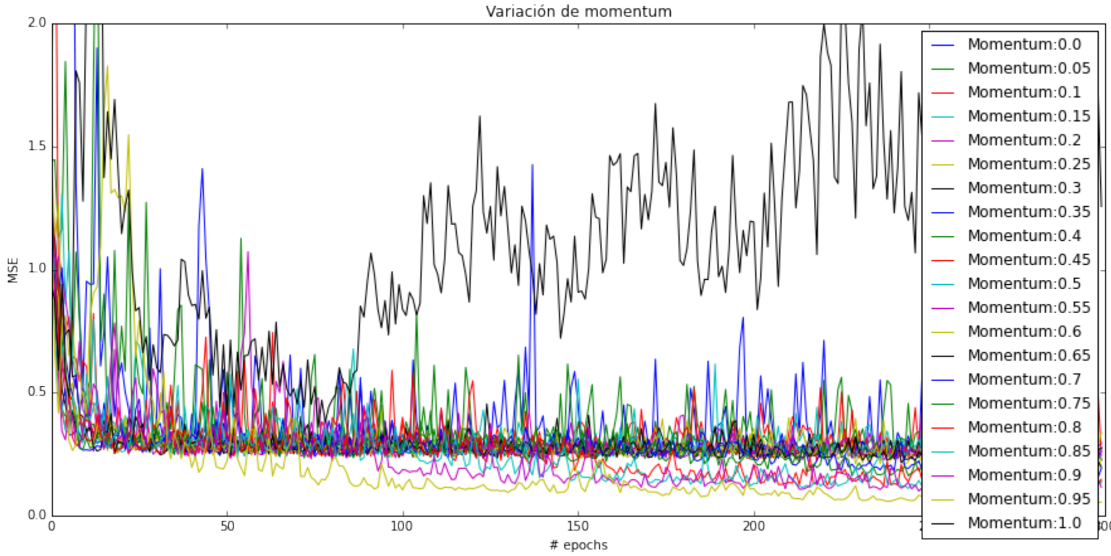
```
plt.ylabel("MSE")
plt.legend()
plt.ylim([0,2])
plt.show()
```



Se observa que un decay igual a 1.0 no tiene un efecto en la disminución del MSE. A medida que el factor de decaimiento disminuye el MSE tiende a descender de forma más rápida y converger a valores más bajos. Valores extremadamente bajos de decay poseen un comportamiento demasiado inestable, el MSE puede llegar a oscilar entre el mínimo encontrado (cualitativamente en MSE=0.25) y MSE = 1.0

i) Entrenar el modelo en d) usando momentum

```
In [46]: n_decay = 21
         momentum = np.linspace(0,1,n_decay)

         %matplotlib inline
         epochs = np.arange(300)
         plt.figure(figsize=(15,7))
         for i in momentum:
             sgd = SGD(lr=0.02,momentum=i)
             model = generate_model(sgd, "sigmoid")

             hist = model.fit(X_train_scaled.as_matrix(), y_train_scaled.as_matrix(),
                              nb_epoch=300, verbose=0,
                              validation_data=(X_test_scaled.as_matrix(), y_test_scaled.as_ma
             plt.plot(epochs, hist.history['loss'],label="Momentum:"+str(i))

         plt.title(u"Variación de momentum")
         plt.xlabel("# epochs")
         plt.ylabel("MSE")
         plt.legend()
         plt.ylim([0,2])
         plt.show()
```



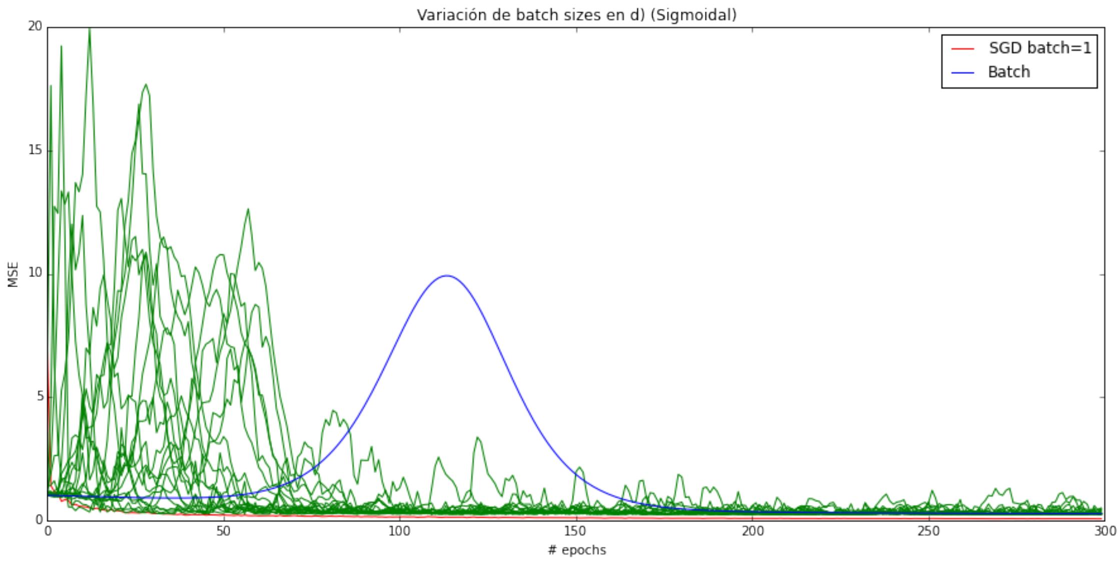
Es posible apreciar que para valores de momentum muy alto (cercanos a 1.0) el MSE tiende a diverger luego de muchas iteraciones. En el otro extremo, el momentum 0 arroja una minimización del MSE bastante inestable luego de 300 iteraciones. Existe una región entorno a momentums 0.25 y 0.75 que poseen una convergencia mucho mas estable que los casos extremos hacia MSE decreciente. De hecho en este caso el mejor MSE se logró con momentum 0.25

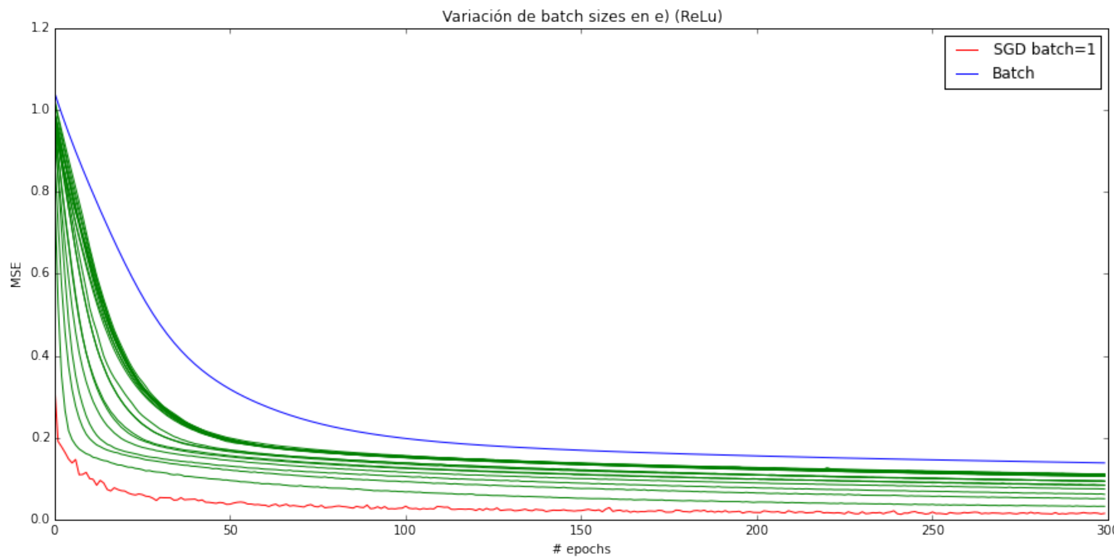
j) Entrenar el modelo en d) y e) cambiando el tamaño del batch. Comparar sgd, batch y mini-batch

```
In [47]: n_batches = 21
         batch_sizes = np.round(np.linspace(1,X_train_scaled.shape[0],n_batches))

         %matplotlib inline
         epochs = np.arange(300)
         plt.figure(figsize=(15,7))
         for i in batch_sizes:
             sgd = SGD(lr=0.02)
             model = generate_model(sgd, "sigmoid")

             hist = model.fit(X_train_scaled.as_matrix(), y_train_scaled.as_matrix(),
                              batch_size=i, nb_epoch=300, verbose=0,
                              validation_data=(X_test_scaled.as_matrix(), y_test_scaled.as_ma
             if i == 1:
                 plt.plot(epochs, hist.history['loss'], 'r-', label="SGD batch=1")
             elif i == X_train_scaled.shape[0]:
                 plt.plot(epochs, hist.history['loss'], 'b-', label="Batch")
             else:
                 plt.plot(epochs, hist.history['loss'], 'g-')

         plt.title(u"Variación de batch sizes en d) (Sigmoidal)")
         plt.xlabel("# epochs")
         plt.ylabel("MSE")
         plt.legend()
         plt.show()
```

```
%matplotlib inline
epochs = np.arange(300)
plt.figure(figsize=(15,7))
for i in batch_sizes:
    sgd = SGD(lr=0.02)
    model = generate_model(sgd, "relu")

    hist = model.fit(X_train_scaled.as_matrix(), y_train_scaled.as_matrix(),
                     batch_size=i, nb_epoch=300, verbose=0,
                     validation_data=(X_test_scaled.as_matrix(), y_test_scaled.as_ma
    # SGD
    if i == 1:
        plt.plot(epochs, hist.history['loss'], 'r-', label="SGD batch=1")
    # BATCH
    elif i == X_train_scaled.shape[0]:
        plt.plot(epochs, hist.history['loss'], 'b-', label="Batch")
    # MINIBATCH
    else:
        plt.plot(epochs, hist.history['loss'], 'g-')

plt.title(u"Variación de batch sizes en e) (ReLu)")
plt.xlabel("# epochs")
plt.ylabel("MSE")
plt.legend()
plt.show()
```

```
/usr/lib/python2.7/site-packages/keras/engine/training.py:807: VisibleDeprecationWarning: u
  batch_ids = index_array[batch_start:batch_end]
/usr/lib/python2.7/site-packages/keras/engine/training.py:914: VisibleDeprecationWarning: u
  batch_ids = index_array[batch_start:batch_end]
```





Podemos observar en los gráficos anteriores que al variar el tamaño del batch desde 1 hasta el tamaño total, tenemos 3 resultados distinguibles en cada gráfico: cuando el batch es igual a 1, lo que es mostrado en la línea roja y asemeja SGD. Mini-batch que va variando de 2 al tamaño de datos-1 mostrado en las línea verdes y cuando el batch es del tamaño total mostrado en la línea azul.

Además en el modelo con función de activación sigmoidal podemos observar que cuando el batch=1 la convergencia es casi inmediata, mientras va subiendo la convergencia oscila con un gran error al comienzo y llegando al tamaño de batch total se observa que comienza con un error bajo el cual sube y vuelve bajar establemente.

En el modelo con función de activación ReLu las convergencias en general son más estables, aunque se observa un poco de oscilación en batch=1, pero es cuando se alcanza el menor error, mientra que con tamaño de batch total se alcanza el mayor error pero con convergencia estable.

3 Ejercicio 3

a) La función load_CIFAR10 permite generar el training set, testing set y validation set a partir de los datos de CIFAR-

```
In [10]: import cPickle as pickle
         import os
         from scipy.misc import imread
         # Inicializar semilla aleatoria
         np.random.seed(20)

         # Carga de un archivo de CIFAR
         def load_CIFAR_one(filename):
             with open(filename, 'rb') as f:
                 datadict = pickle.load(f)
                 X = datadict['data']
                 Y = datadict['labels']
                 Y = np.array(Y)
                 return X, Y

         # Carga todos los archivos CIFAR y generar Training set, Testing set y Validation
```

```
def load_CIFAR10(PATH, n_files=6):
    xs = []
    ys = []
    # Juntar toda la data de entrenamiento
    for b in range(1, n_files):
        f = os.path.join(PATH, 'data_batch_%d' % (b, ))
        X, Y = load_CIFAR_one(f)
        xs.append(X)
        ys.append(Y)
    Xtr = np.concatenate(xs)
    Ytr = np.concatenate(ys)
    del X, Y
    # Obtener subconjunto para validacion a partir de data de entrenamiento
    v_size = np.random.randint(1000, 5000)
    indices = np.random.choice(np.arange(Xtr.shape[0]), v_size)
    mask_tr = np.ones(Xtr.shape[0], dtype=bool)
    mask_tr[indices] = False
    mask_v = np.invert(mask_tr)
    # Obtener conjunto de validacion
    Xv = Xtr[mask_v]
    Yv = Ytr[mask_v]
    # Obtener conjunto de prueba
    Xtr = Xtr[mask_tr]
    Ytr = Ytr[mask_tr]
    # Obtener data de prueba
    Xte, Yte = load_CIFAR_one(os.path.join(PATH, 'test_batch'))
    return Xtr, Ytr, Xte, Yte, Xv, Yv

# Cargar desde carpeta local data
Xtr, Ytr, Xte, Yte, Xv, Yv = load_CIFAR10("data")
label_names = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'ho
```

b) Escalamiento y centrado de datos. Aparte de las ventajas mencionadas anteriormente, los resultados experimentales con datos no centrados y no escalados ofrecen resultados peores que si la data se normaliza, por lo tanto a partir de este punto se trabajó sólo con datos normalizados.

In [11]:
```
from sklearn.preprocessing import StandardScaler

# Centrar dataset y escalar segun preferencia
def preprocess(X, with_mean=True, with_std=True):
    scaler = StandardScaler(with_mean, with_std).fit(X)
    return scaler.transform(X)

# Data solo centrada
#Xtr_c = preprocess(Xtr, with_mean=True, with_std=False)
# Data solo escalada
#Xtr_s = preprocess(Xtr, with_mean=False, with_std=True)
# Data centrada y escalada

# Centrar y escalar datos de entrenamiento y validación
Xtr_cs = preprocess(Xtr)
Xv_cs = preprocess(Xv)
Xte_cs = preprocess(Xte)
```

```
/usr/lib64/python2.7/site-packages/sklearn/utils/validation.py:420: DataConversionWarning:
  warnings.warn(msg, DataConversionWarning)
```

```
/usr/lib64/python2.7/site-packages/sklearn/utils/validation.py:420: DataConversionWarning:
  warnings.warn(msg, DataConversionWarning)
/usr/lib64/python2.7/site-packages/sklearn/utils/validation.py:420: DataConversionWarning:
  warnings.warn(msg, DataConversionWarning)
/usr/lib64/python2.7/site-packages/sklearn/utils/validation.py:420: DataConversionWarning:
  warnings.warn(msg, DataConversionWarning)
/usr/lib64/python2.7/site-packages/sklearn/utils/validation.py:420: DataConversionWarning:
  warnings.warn(msg, DataConversionWarning)
/usr/lib64/python2.7/site-packages/sklearn/utils/validation.py:420: DataConversionWarning:
  warnings.warn(msg, DataConversionWarning)
```

c) Creación de red neuronal para problema CIFAR. En primer lugar se adaptan las etiquetas a una representación manejable por la red.

```
In [12]: # Dimension de ejemplos, vectores de 3072 features (32x32x3 pixeles)
         input_dim = Xtr.shape[1]
         from keras.utils.np_utils import to_categorical
         # Convertir etiquetas a una representación amigable
         Ytr_conv = to_categorical(Ytr,10)
         Yv_conv = to_categorical(Yv,10)
         Yte_conv = to_categorical(Yte,10)
```

Luego se propone una primera arquitectura simple de una capa escondida de 50 neuronas, activación relu y salida softmax para clasificación. Un learning rate de 0.1 y decaimiento de 1e-6. Para un primera aproximación se utiliza como función de pérdida el error cuadrático medio.

```
In [13]: # Modelo 1: Red simple de 1 capa
         MLPmodel = Sequential()
         MLPmodel.add(Dense(50, input_dim=input_dim, init='uniform'))
         MLPmodel.add(Activation('relu'))
         MLPmodel.add(Dense(10))
         MLPmodel.add(Activation('softmax'))
         sgd = SGD(lr=0.1, decay=1e-6, momentum=0.9, nesterov=False)
         MLPmodel.compile(loss='mse',
                       optimizer=sgd,
                       metrics=['accuracy'])

In [14]: MLPmodel.fit(Xtr_cs, Ytr_conv, nb_epoch=20, batch_size=16, verbose=0)

Out[14]: <keras.callbacks.History at 0x7f9b07b8bd10>

In [16]: score = MLPmodel.evaluate(Xv_cs, Yv_conv,verbose=0, batch_size=16)
         print "Loss:",score[0], "Accuracy:",score[1]

Loss: 0.13602719058 Accuracy: 0.216306156431
```

Los resultados muestran que la precisión de esta red es pobre, sólo de 20% sobre el set de validación.

A continuación un segundo modelo surge luego de experimentar con funciones tangente hiperbolica. Se agregó otra capa oculta, y se activó la opción Nesterov para estimación del momentum.

```
In [25]: # Modelo 2
         # Combinacion de capas con activaciones tangente hiperbolica, momentum utilizando
         model2 = Sequential()
         model2.add(Dense(50, input_dim=input_dim, init='uniform'))
```

```
        model2.add(Activation('tanh'))
        model2.add(Dense(50, init='uniform'))
        model2.add(Activation('tanh'))
        model2.add(Dense(10, init='uniform'))
        model2.add(Activation('softmax'))
        sgd = SGD(lr=0.1, decay=1e-6, momentum=0.8, nesterov=True)
        model2.compile(loss='mse',
                      optimizer=sgd,
                      metrics=['accuracy'])
```

```
In [40]: model2.fit(Xtr_cs, Ytr_conv, nb_epoch=20, batch_size=32, verbose=0)

Out[40]: <keras.callbacks.History at 0x7f003015b910>

In [41]: score = model2.evaluate(Xv_cs, Yv_conv, batch_size=32, verbose=0)
         print "Loss:",score[0], "Accuracy:",score[1]

Loss: 0.0710594502123 Accuracy: 0.445923460969
```

Existe una mejora sustancial, con una precisión de 45% sobre el conjunto de validación, lo que da esperanzas acerca de la dirección en la cual el modelamiento debe seguir.

El tercer modelo surge luego de varios experimentos. La función de activación sigmoidal se presenta mejor, y la inicialización de los pesos con distribución normal también ayuda. Además y por resultados exhibidos anteriormente la métrica de pérdida es reemplazada por Categorical Crossentropy, que nos dice más información a la hora de clasificar.

```
In [65]: # Modelo 3
         # Combinación anterior cambiando funciones tanh por sigmoidales y perdida entropi
         model3 = Sequential()
         model3.add(Dense(256, input_dim=input_dim, init='uniform'))
         model3.add(Activation('sigmoid'))
         model3.add(Dense(256, init='normal'))
         model3.add(Activation('sigmoid'))
         model3.add(Dense(10, init='normal'))
         model3.add(Activation('softmax'))
         sgd = SGD(lr=0.1, decay=1e-6, momentum=0.8, nesterov=True)
         model3.compile(loss='categorical_crossentropy',
                       optimizer=sgd,
                       metrics=['accuracy'])

In [66]: model3.fit(Xtr_cs, Ytr_conv, nb_epoch=20, batch_size=32, verbose=0)

Out[66]: <keras.callbacks.History at 0x7f002ee8d910>

In [64]: score = model3.evaluate(Xv_cs, Yv_conv, batch_size=32, verbose=0)
         print "Loss:",score[0], "Accuracy:",score[1]

Loss: 1.65634136067 Accuracy: 0.497741858878
```

Los resultados son mejores aún, una mejora de 5% respecto al modelo propuesto anteriormente. Como no se mejoró más luego de experimentar este modelo será utilizado para la siguiente etapa.

(d) Extracción de features y experimentación. Las features son extraídas con la función provista, se observan cambios efectivos en la dimensionalidad del problema, pasando de manejar miles de variables a sólo cientas. Recordemos que nuestro rendimiento con las features iniciales y la mejor red experimentada aún no es de 50%

```
In [20]: from top_level_features import hog_features
         from top_level_features import color_histogram_hsv
         from top_level_features import extract_features
         Xtr, Ytr, Xte, Yte, Xv, Yv = load_CIFAR10("data")

         # Extraer features hog (cambios en gradiente) y histograma en espacio de color hsv
         featuresXtr = extract_features(Xtr,[hog_features, color_histogram_hsv])
         featuresV = extract_features(Xv,[hog_features, color_histogram_hsv])
         featuresTe = extract_features(Xte,[hog_features, color_histogram_hsv])
         Ytr_conv = to_categorical(Ytr, 10)
         Yte_conv = to_categorical(Yte, 10)
         Yv_conv = to_categorical(Yv, 10)
         # Obtuvimos en vez de 3072 features o variables, 154 variables
         print "Features information"
         print featuresXtr.shape
         print featuresV.shape
         print featuresTe.shape
```

```
(47145, 32, 32, 3)
(2855, 32, 32, 3)
(10000, 32, 32, 3)
Features information
(47145, 154)
(2855, 154)
(10000, 154)
```

Repetimos la definición del mejor modelo del item anterior y realizamos el fiting con las nuevas features.

```
In [23]: # El mejor modelo fue usado para probar las nuevas features
         # Probando con la extracción de ambos tipos de features
         model3 = Sequential()
         model3.add(Dense(256, input_dim=featuresXtr.shape[1], init='uniform'))
         model3.add(Activation('sigmoid'))
         model3.add(Dense(256, init='normal'))
         model3.add(Activation('sigmoid'))
         model3.add(Dense(10, init='normal'))
         model3.add(Activation('softmax'))
         sgd = SGD(lr=0.1, decay=1e-6, momentum=0.8, nesterov=True)
         model3.compile(loss='categorical_crossentropy',
                        optimizer=sgd,
                        metrics=['accuracy'])

         model3.fit(featuresXtr, Ytr_conv, nb_epoch=20, batch_size=32, verbose=0)

Out[23]: <keras.callbacks.History at 0x7f9ae6a14110>
```

```
In [28]: print "Loss and accuracy for validation set"
         print model3.evaluate(featuresV, Yv_conv, batch_size=32, verbose=0)
         print "Loss and accuracy for testing set"
         print model3.evaluate(featuresTe, Yte_conv, batch_size=32, verbose=0)
```

```
Loss and accuracy for validation set
[1.4806456514498816, 0.55271453593324238]
Loss and accuracy for testing set
[1.4744582109451294, 0.55310000000000004]
```

Es posible apreciar que el rendimiento tanto en validation set como en testing set es sobre 50%, lo cual es bastante bueno y permite valorar la utilidad de la extracción de features al aportar un 10% de mejora.

A continuación se trata la data sólo utilizando hog features.

```
In [30]: #Probando con la extracción solo de hog features
         featuresXtr = extract_features(Xtr,[hog_features])
         featuresV = extract_features(Xv,[hog_features])
         featuresTe = extract_features(Xte,[hog_features])
         model3 = Sequential()
         model3.add(Dense(256, input_dim=featuresXtr.shape[1], init='uniform'))
         model3.add(Activation('sigmoid'))
         model3.add(Dense(256, init='normal'))
         model3.add(Activation('sigmoid'))
         model3.add(Dense(10, init='normal'))
         model3.add(Activation('softmax'))
         sgd = SGD(lr=0.1, decay=1e-6, momentum=0.8, nesterov=True)
         model3.compile(loss='categorical_crossentropy',
                        optimizer=sgd,
                        metrics=['accuracy'])
         model3.fit(featuresXtr, Ytr_conv, nb_epoch=20, batch_size=32, verbose=0)
```

```
(47145, 32, 32, 3)
(2855, 32, 32, 3)
(10000, 32, 32, 3)
```

```
Out[30]: <keras.callbacks.History at 0x7f9ae5b9d850>
```

```
In [32]: print "Loss and accuracy for validation set"
         print model3.evaluate(featuresV, Yv_conv, batch_size=32, verbose=0)
         print "Loss and accuracy for testing set"
         print model3.evaluate(featuresTe, Yte_conv, batch_size=32, verbose=0)
```

```
Loss and accuracy for validation set
[1.5606305140329952, 0.53099824869695356]
Loss and accuracy for testing set
[1.5511400064468384, 0.53469999999999995]
```

```
In [33]: print featuresXtr.shape
```

```
(47145, 144)
```

Para el caso de utilizar sólo hog features, la mejora es aproximadamente de un 8% a 9%, La dimensionalidad efectivamente decrece nuevamente esta vez a sólo 144 features.

A continuación se prueba sólo con hsv features.

```
In [34]: #Probando con la extracción solo de hsv features

         featuresXtr = extract_features(Xtr,[color_histogram_hsv])
         featuresV = extract_features(Xv,[color_histogram_hsv])
         featuresTe = extract_features(Xte,[color_histogram_hsv])
         model3 = Sequential()
         model3.add(Dense(256, input_dim=featuresXtr.shape[1], init='uniform'))
         model3.add(Activation('sigmoid'))
```

```
        model3.add(Dense(256, init='normal'))
        model3.add(Activation('sigmoid'))
        model3.add(Dense(10, init='normal'))
        model3.add(Activation('softmax'))
        sgd = SGD(lr=0.1, decay=1e-6, momentum=0.8, nesterov=True)
        model3.compile(loss='categorical_crossentropy',
                      optimizer=sgd,
                      metrics=['accuracy'])
        model3.fit(featuresXtr, Ytr_conv, nb_epoch=20, batch_size=32, verbose=0)
```

```
(47145, 32, 32, 3)
(2855, 32, 32, 3)
(10000, 32, 32, 3)
```

```
Out[34]: <keras.callbacks.History at 0x7f9ae5ad3c50>
```

```
In [35]: print "Loss and accuracy for validation set"
         print model3.evaluate(featuresV, Yv_conv, batch_size=32, verbose=0)
         print "Loss and accuracy for testing set"
         print model3.evaluate(featuresTe, Yte_conv, batch_size=32, verbose=0)
```

```
Loss and accuracy for validation set
[2.0366123935178364, 0.22977232927825111]
Loss and accuracy for testing set
[2.0466881441116334, 0.23069999999999999]
```

```
In [36]: print featuresXtr.shape
```

```
(47145, 10)
```

La pérdida de precisión es bastante grande, alrededor de un 80%. Esta feature por si sola por lo tanto no es una buena indicadora de las imágenes en el dataset y es sólo un 10% mejor que si se elige arbitrariamente una clase para la imagen. La cantidad de features es sólo 10, lo que si bien permite pensar que la dimensionalidad se ha reducido y por lo tanto tendremos resultados más rápidos, en realidad estas features por si mismas no describen al espacio de hipótesis que se desea tratar. No obstante como complemento de HOG, como se vio en el primer experimento de este estilo, permitió reforzar el aprendizaje efectivamente.

```
In [ ]:
```