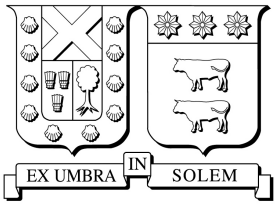


INF-477. Redes Neuronales Artificiales.
Tarea 2 - CNNs y AEs.

Temas

- Diseño y entrenamiento de Redes Convolucionales (CNNs).
- Diseño y entrenamiento de autoencoders (AEs).
- Apilamiento de autoencoders (Stacked AEs).
- Denoising y Reducción de Dimensionalidad usando AEs.
- Pre-entrenamiento.

Formalidades

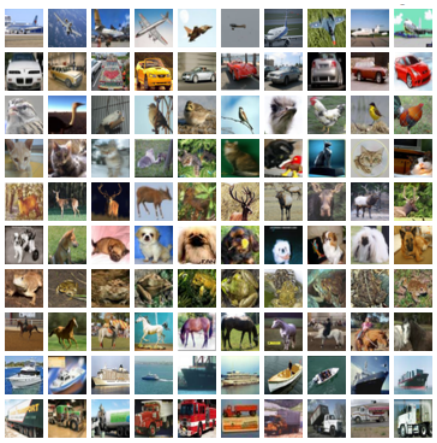
- Equipos de trabajo de: 2 personas.*.
- Se debe preparar un (breve) Jupyter/IPython notebook que explique la actividad realizada y las conclusiones del trabajo.
- Los itemes identificados con los símbolos $\star$ (menos sencillo) y $\star\star$ (más difícil) tendrán un peso mayor y mucho mayor (respectivamente) en la evaluación. Los estudiantes de postgrado tendrán preferencia para exponer estos puntos durante las presentaciones.
- Se debe preparar una presentación de 20 minutos. Presentador será elegido aleatoriamente.
- Se debe mantener un respaldo de cualquier tipo de código utilizado, informe y presentación en Github.
- Formato de entrega: envío de link Github al correo electrónico del ayudante (joaquin.velasquez@alumnos.inf.utfsm.cl), con copia a y asunto: [Tarea2-INF477-I-2017].

Notación: Con el objetivo de referirnos con mayor facilidad a una determinada arquitectura convolucional, utilizaremos la siguiente notación (simplificada con respecto a lo que se lee en algunos papers). La expresión "arquitectura $H^1 \times H^2 \times \ldots H^L$" denotará una red de $L+1$ capas donde se omite la capa de entrada (que correspondería a H^0) y H^ℓ puede ser de tipo C (capa convolucional), P (capa de pooling) ó F (capa MLP clásica, totalmente conectada). Así por ejemplo la expresión "arquitectura $C \times C \times P \times F \times F$" denota una CNN con dos capas convolucionales, seguidas de 1 de pooling y coronada con 2 capas MLP convencionales (1 oculta y 1 de salida).

*Cada miembro del equipo debe estar en condiciones de realizar una presentación y discutir sobre cada punto del trabajo realizado.

1 CNNs en CIFAR10

En esta sección, experimentaremos con redes convolucionales, conocidas como *CNNs* ó *ConvNets*. Para ello y con el fin de comparar con los resultados obtenidos por redes FF, reutilizaremos el dataset *CIFAR10* utilizado en la tarea anterior. Si no dispone ya del dataset en su máquina, por favor siga a las instrucciones de descarga entregadas en la tarea anterior. **Nota:** Para esta actividad es bastante aconsejable entrenar las redes usando una GPU, ya que de otro modo los tiempos de entrenamiento son largos.



(a) Prepare subconjuntos de entrenamiento, validación y pruebas idénticos a los que utilizó en el último ítem de la tarea 1. Normalice las imágenes de entrenamiento y pruebas, dividiendo las intensidades originales de pixel en cada canal por 255. Es importante recordar que en la tarea1 representamos las imágenes como un vector unidimensional. Por lo tanto, antes de trabajar con CNNs será necesario recuperar la forma original de las imágenes. Además, si desea trabajar con el orden de las dimensiones denominado 'tf' (por defecto para TensorFlow[†]) deberá hacer realizar la transposición correspondiente. Finalmente, genere una representación adecuada de las salidas deseadas de la red.

```
import keras
x_train = x_train.reshape((x_train.shape[0],32,32,3))
x_train = x_train.transpose([0, 2, 3, 1]) #only if 'tf' dim-ordering is to be used
x_test= x_test.reshape((x_test.shape[0],32,32,3))
x_test= x_test.transpose([0, 2, 3, 1])#remove if 'th' dim-ordering is to be used
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)
```

(b) Defina una CNN con arquitectura $C \times P \times C \times P \times F \times F$. Para ambas capas convolucionales utilice 64 filtros de 3×3 y funciones de activación *ReLu*. Para las capas de pooling utilice filtros de 2×2 con stride 2. Para la capa MLP escondida use 512 neuronas. Genere un esquema lo más compacto posible que muestre los cambios de forma (dimensionalidad) que experimenta un patrón de entrada a medida que se ejecuta un forward-pass y el número de parámetros de cada capa.

```
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers import Convolution2D, MaxPooling2D

model = Sequential()
model.add(Convolution2D(64, 3, 3, border_mode='same', input_shape=x_train.shape[1:]))
model.add(Activation('relu'))
```

[†]Si usa theano como backend, es posible usar el orden 'tf' especificándolo en el archivo .keras/keras.jason disponible en su instalación de keras.

```
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Convolution2D(64, 3, 3, border_mode='same'))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(512))
model.add(Activation('relu'))
model.add(Dense(num_classes))
model.add(Activation('softmax'))
model.summary()
```

(c) Entrene la CNN definida en (b) utilizando SGD. En este dataset, una tasa de aprendizaje "segura" es $\eta = 10^{-4}$ o inferior, pero durante las primeras "epochs" el entrenamiento resulta demasiado lento. Para resolver el problema aprenderemos a controlar la tasa de aprendizaje utilizada en el entrenamiento. Implemente la siguiente idea: deseamos partir con una tasa de aprendizaje $\eta = 10^{-3}$ y dividir por 2 ese valor cada 10 epochs. Suponga además que no queremos usar una tasa de aprendizaje menor a $\eta = 10^{-5}$. Defina esta regla de ajuste para η y entrene la CNN definida en (a) durante 25 epochs. Construya un gráfico que muestre los errores de entrenamiento, validación y pruebas como función del número de "epochs".

```
from keras.optimizers import SGD, rmsprop
from keras.callbacks import LearningRateScheduler
import math
def step_decay(epoch):
  initial_lrate = 0.001
  lrate = initial_lrate * math.pow(0.5, math.floor((1+epoch)/5))
  lrate = max(lrate,0.00001)
  return lrate
opt = SGD(lr=0.0, momentum=0.9, decay=0.0)
lrate = LearningRateScheduler(step_decay)
model.compile( ... )
model.fit(x_train, y_train,batch_size=batch_size,nb_epoch=epochs,
              validation_data=(x_test, y_test),shuffle=True,callbacks=[lrate])
```

(d) Entrene la CNN definida en (b) utilizando RMSProp durante 25 epochs. Elija la función de pérdida más apropiada para este problema. Construya finalmente un gráfico que muestre los errores de entrenamiento, validación y pruebas como función del número de "epochs".

```
from keras.optimizers import SGD, rmsprop
opt = rmsprop(lr=0.001, decay=1e-6)
model.compile( ... )
model.fit(x_train, y_train,batch_size=batch_size,nb_epoch=epochs,
              validation_data=(x_test, y_test),shuffle=True)
```

(e) ⋆ Se ha sugerido que la práctica bastante habitual de continuar una capa convolucional con una capa de pooling puede generar una reducción prematura de las dimensiones del patrón de entrada. Experimente con una arquitectura del tipo $C \times C \times P \times C \times C \times P \times F \times F$. Use 64 filtros para las primeras 2 capas convolucionales y 128 para las últimas dos. Reflexione sobre qué le parece más sensato: ¿mantener el tamaño de los filtros usados anteriormente? o ¿usar filtros más grandes en la segunda capa convolucional y más pequeños en la primera? o ¿usar filtros más pequeños en la segunda capa convolucional y más grandes en la primera? Hint: con esta nueva arquitectura debiese superar el 70% de accuracy (de validación/test) antes de 5 epochs, pero la arquitectura es más sensible a overfitting por lo que podría ser conveniente agregar un regularizador. Como resultado final de esta actividad gráficque los errores de entrenamiento, validación y pruebas como función del número de "epochs" (fijando el máximo en un número razonable como $T = 25$).

```
model = Sequential()
model.add(Convolution2D(64, 3, 3, border_mode='same', input_shape=x_train.shape[1:]))
model.add(Activation('relu'))
model.add(Convolution2D(64, 3, 3, border_mode='same'))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
...
```

(f) ⋆ Algunos investigadores, han propuesto que las capas de pooling se pueden reemplazar por capas convoluciones con stride 2. ¿Se reduce dimensionalidad de este modo? Compruébelo verificando los cambios de forma (dimensionalidad) que experimenta un patrón de entrada a medida que se ejecuta un forward-pass. Entrene la red resultante con el método que prefiera, gráficando los errores de entrenamiento, validación y pruebas como función del número de "epochs" (fijando el máximo en un número razonable como $T = 25$).

```
...
model.add(Convolution2D(128, 3, 3, border_mode='same'))
model.add(Activation('relu'))
model.add(Convolution2D(128, 3, 3, border_mode='same'))
model.add(Activation('relu'))
model.add(Convolution2D(64, 3, 3, subsample=(2, 2), border_mode='valid'))
...
```

(g) ⋆⋆ Una forma interesante de regularizar modelos entrenados para visión artificial consiste en "aumentar" el número de ejemplos de entrenamiento usando transformaciones sencillas como: rotaciones, corrimientos y reflexiones, tanto horizontales como verticales. Explique porqué este procedimiento podría ayudar a mejorar el modelo. Evalúe experimentalmente la conveniencia de incorporarlo.

```
datagen = ImageDataGenerator(
        featurewise_center=False,  # set input mean to 0 over the dataset
        samplewise_center=False,  # set each sample mean to 0
        featurewise_std_normalization=False,  # divide inputs by std of the dataset
        samplewise_std_normalization=False,  # divide each input by its std
        zca_whitening=False,  # apply ZCA whitening
        rotation_range=0, # randomly rotate images (degrees, 0 to 180)
        width_shift_range=0.1, # randomly shift images horizontally (fraction of width)
        height_shift_range=0.1, # randomly shift images vertically (fraction of height)
        horizontal_flip=True, # randomly flip images
        vertical_flip=False) # randomly flip images
datagen.fit(x_train)
model.fit_generator(datagen.flow(x_train, y_train,batch_size=batch_size),
                        steps_per_epoch=x_train.shape[0]// batch_size,
                        epochs=epochs,validation_data=(x_test, y_test))
```

(h) Elija una de las redes entrenadas en esta sección y determine los pares de objetos (por ejemplo "perros" con "gatos") que la red tiende a confundir. Conjeture el motivo de tal confusión.

2 CNNs en Google Street View

En esta sección, trabajaremos con el dataset *SVHN* (Street View House Numbers), correspondiente a imágenes naturales de dígitos de direcciones obtenidos desde Google Street View. El dataset contiene más de 600.000 imágenes de entrenamiento y 26.032 imágenes de test. Para facilitar la realización de experimentos, el dataset de entrenamiento se divide usualmente en un conjunto pequeño de 73.257 imágenes y un conjunto "extra" de 531.131 imágenes. En esta tarea trabajaremos sólo con la versión pequeña. Los valientes pueden verificar que entrenando sobre el conjunto grande los resultados mejoran significativamente.

Los datos pueden ser obtenidos (en formato Matlab) ejecutando los siguientes comandos

```
wget http://ufldl.stanford.edu/housenumbers/train_32x32.mat
wget http://ufldl.stanford.edu/housenumbers/extra_32x32.mat
wget http://ufldl.stanford.edu/housenumbers/test_32x32.mat
```

(a) Cargue los datos de entrenamiento y pruebas ("train_32x32.mat" y "test_32x32.mat"). Determine el tamaño de las imágenes, el número de clases diferentes y de ejemplos en cada categoría. Finalmente, visualice 5 imágenes de entrenamiento y 5 de test (elegidas aleatoriamente). Comente.

```
import scipy.io as sio
import numpy as np
train_data = sio.loadmat('train_32x32.mat')
test_data = sio.loadmat('test_32x32.mat')
X_train = train_data['X'].T
y_train = train_data['y'] - 1
X_test = test_data['X'].T
y_test = test_data['y'] - 1
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
n_classes = len(np.unique(y_train))
print np.unique(y_train)
```

(b) Normalice las imágenes, dividiendo las intensidades originales de pixel por 255. Represente adecuadamente la salida deseada de la red de modo de tener un vector de tamaño igual al número de clases[‡].

```
from keras.utils import np_utils
X_train /= 255
X_test /= 255
Y_train = np_utils.to_categorical(y_train, n_classes)
Y_test = np_utils.to_categorical(y_test, n_classes)
```

(c) Defina una CNN con arquitectura $C \times P \times C \times P \times F \times F$. Para la primera capa convolucional utilice 16 filtros de 5×5 y para la segunda 512 filtros de 7×7. Para la capa MLP escondida use 20 neuronas. Esta arquitectura, con algunas diferencias, fue una de las primera CNNs entrenadas sobre SVHN y consiguió una accuracy de 94.28% [11]. Genere un esquema lo más compacto posible que muestre los cambios de forma que experimenta un patrón de entrada a medida que se ejecuta un forward-pass. Entrene la red anterior un máximo de 10 epochs. ¿Logra mejorar o al menos igualar el resultado reportado en la literatura?

[‡]Antes de seguir, se le recomienda también revisar su archivo de configuración *keras.json*, para verificar que el índice correspondiente a los canales de una imagen de entrada sea consistente con las operaciones aquí realizadas. Puede desear también configurar su archivo .keras/keras.json para definir el orden por defecto.

```
from keras.models import Sequential
from keras.layers.core import Dense, Dropout, Activation, Flatten
from keras.layers.convolutional import Convolution2D, MaxPooling2D, AveragePooling2D
from keras.optimizers import SGD, Adadelta, Adagrad
model = Sequential()
model.add(Convolution2D(16, 5, 5, border_mode='same', activation='relu',
                        input_shape=(n_channels, n_rows, n_cols)))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Convolution2D(512, 7, 7, border_mode='same', activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(20, activation='relu'))
model.add(Dense(n_classes, activation='softmax'))
model.summary()
model.compile(loss='binary_crossentropy', optimizer=adagrad,  metrics=['accuracy'])
adagrad = Adagrad(lr=0.01, epsilon=1e-08, decay=0.0)
model.fit(X_train, Y_train, batch_size=1280, nb_epoch=12, verbose=1, \
          validation_data=(X_test, Y_test))
```

(d) ⋆ Evalúe el efecto de modificar el tamaño de los filtros (de convolución y pooling) reportando la sensibilidad del error de pruebas a estos cambios. Presente un gráfico o tabla resumen. Por simplicidad entre durante sólo 10 epochs.

(e) ⋆ Evalúe el efecto de modificar el número de filtros para las capas convolucionales tanto en los tiempos de entrenamiento como en el desempeño de la red. Presente un gráfico o tabla resumen. Por simplicidad entre durante sólo 10 epochs.

(f) Proponga una mejora sobre la red definida en (c) que mejore el error de pruebas. Recuerde que debe definir un subconjunto de validación si necesita elegir entre arquitecturas.

(g) ⋆ Elija una de las redes entrenadas (preferentemente una con buen desempeño) y visualice los pesos correspondientes a los filtros de la primera capa convolucional. Visualice además el efecto del filtro sobre algunas imágenes de entrenamiento. Comente.

(h) Elija una de las redes entrenadas en esta sección y determine los pares de dígitos (por ejemplo "1" con "7") que la red tiende a confundir. Conjeture el motivo de tal confusión.

(i) (Opcional, Bonus +10 en certamen) Evalúe la conveniencia de utilizar todo el dataset ("extra_32x32.mat") en el entrenamiento de la red.

3 Autoencoders (AEs) en MNIST

Como hemos discutido en clases, las RBM's y posteriormente los AE's (redes no supervisadas) fueron un componente crucial en el desarrollo de los modelos que entre 2006 y 2010 vigorizaron el área de las redes neuronales artificiales con logros notables de desempeño en diferentes tareas de aprendizaje automático. En esta sección aprenderemos a utilizar el más sencillo de estos modelos: un autoencoder o AE. Consideraremos tres aplicaciones clásicas: reducción de dimensionalidad, denoising y pre-entrenamiento. Con este objetivo en mente, utilizaremos un dataset denominado *MNIST* [7]. Se trata de una colección de 70.000 imágenes de 28×28 pixeles correspondientes a dígitos manuscritos (números entre 0 y 9). En su versión tradicional, la colección se encuentra separada en dos subconjuntos: uno de entrenamiento de 60.000 imágenes y otro de test de 10.000 imágenes. La tarea consiste en construir un programa para que aprenda a identificar correctamente el dígito representado en la imagen.

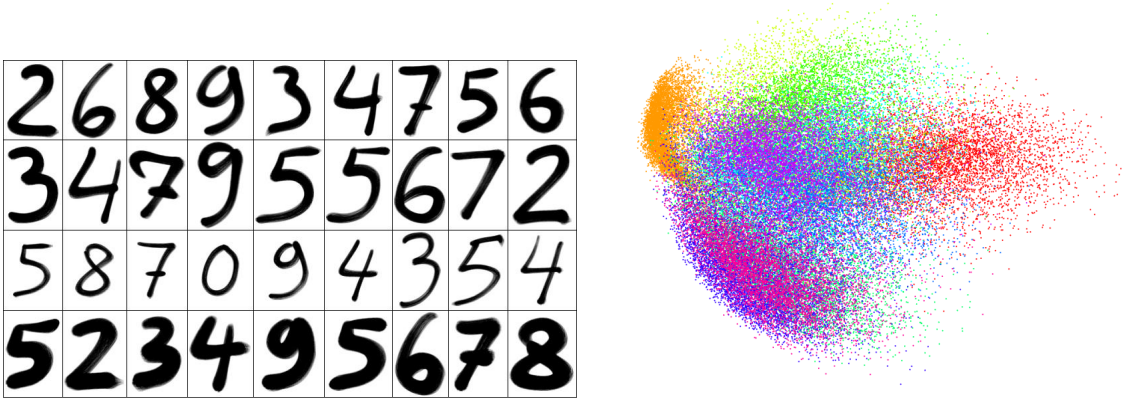


Fig. 1: Dataset MNIST y visualización obtenida usando las primeras dos componentes principales.

(a) Escriba una función que cargue los datos desde el repositorio de *keras*, normalice las imágenes de modo que los pixeles queden en $[0, 1]$, transforme las imágenes en vectores ($\in \mathbb{R}^{784}$) y devuelva tres subconjuntos disjuntos: uno de entrenamiento, uno de validación y uno de pruebas. Construya el conjunto de validación utilizando los últimos NVAL $= 5000$ casos del conjunto del entrenamiento. El conjunto de entrenamiento consistirá en las primeras $60000 -$ NVAL imágenes.

```
from keras.datasets import mnist
import numpy as np
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train = x_train.astype('float32') / 255.
x_train = x_train.reshape((len(x_train), np.prod(x_train.shape[1:])))
x_test = x_test.reshape((len(x_test), np.prod(x_test.shape[1:])))
#Define here your validation set
Y_train = np_utils.to_categorical(y_train, 10)
Y_test = np_utils.to_categorical(y_test, 10)
```

3.1 Reducción de Dimensionalidad

Una de las aplicaciones típicas de un AE es reducción de dimensionalidad, es decir, implementar una transformación $\phi : \mathbb{R}^d \to \mathbb{R}^{d'}$ de objetos representados originalmente por d atributos en una nueva representación de $d' \ll d$ atributos, de modo tal que se preserve lo mejor posible la "información" original. Obtener tal representación es útil desde un punto de vista computacional (compresión) y estadístico (permite construir modelos con un menor número de parámetros libres). Un AE es una técnica de reducción de dimensionalidad *no supervisada* porque no hace uso de información acerca de las clases a las que pertenecen los datos de entrenamiento.

(a) Entrene un AE básico (1 capa escondida) para generar una representación de MNIST en $d' = 2, 8, 32, 64$ dimensiones. Justifique la elección de la función de pérdida a utilizar y del criterio de entrenamiento en general. Determine el porcentaje de compresión obtenido y el error de reconstrucción en cada caso. ¿Mejora el resultado si elegimos una función de activación ReLU para el Encoder? ¿Podría utilizarse esta activación en el Decoder?

```
from keras.layers import Input, Dense
from keras.models import Model
from keras.optimizers import SGD
input_img = Input(shape=(784,))
encoded = Dense(32, activation='sigmoid')(input_img)
decoded = Dense(784, activation='sigmoid')(encoded)
```

```
autoencoder = Model(input=input_img, output=decoded)
encoder = Model(input=input_img, output=encoded)
encoded_input = Input(shape=(32,))
decoder_layer = autoencoder.layers[-1]
decoder = Model(input=encoded_input, output=decoder_layer(encoded_input))
autoencoder.compile(optimizer=SGD(lr=1.0), loss='binary_crossentropy')
autoencoder.fit(x_train,x_train,nb_epoch=50,batch_size=25,shuffle=True,
                        validation_data=(x_val, x_val))
autoencoder.save('basic_autoencoder_768x32.h5')
#save other stuff ...
```

(b) Compare visualmente la reconstrucción que logra hacer el autoencoder desde la representación en $\mathbb{R}^{d'}$ para algunas imágenes del conjunto de pruebas. Determine si la percepción visual se corresponde con el error de reconstrucción observada. Comente.

```
from keras.models import load_model
autoencoder = load_model('basic_autoencoder_768x32.h5')
#load other stuff ...
encoded_test = encoder.predict(x_test)
decoded_test = decoder.predict(encoded_test)
import matplotlib
n = 10
plt.figure(figsize=(20, 4))
for i in range(n):
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(x_test[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(decoded_test[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```

(c) Para verificar la calidad de la representación obtenida, implemente el siguiente clasificador, denominado *kNN* (k-nearest neighbor): dada una imagen $\mathbf{x}$, el clasificador busca las $k = 10$ imágenes de entrenamiento más similares $N_{\mathbf{x}} = \{\mathbf{x}^{(k_i)}\}_{i=1}^{10}$ (de acuerdo a una distancia, e.g. euclidiana) y predice como clase, la etiqueta más popular entre las imágenes $N_{\mathbf{x}}$. Mida el error de pruebas obtenido construyendo este clasificador sobre la data original y luego sobre la data reducida. Compare además los tiempos medios de predicción en ambos escenarios.

```
encoded_train = encoder.predict(x_train)
encoded_test = encoder.predict(x_test)
from sklearn.neighbors import KNeighborsClassifier
clf = KNeighborsClassifier(10)
clf.fit(encoded_train, y_train)
clf.fit(encoded_train, y_train)
print 'Classification Accuracy %.2f' % clf.score(encoded_test,y_test)
```

(e) ⋆ Compare la calidad de la representación reducida obtenida por el autoencoder básico con aquella obtenida vía PCA (una técnica clásica de reducción de dimensionalidad) utilizando el mismo número de dimensiones $d' = 2, 4, 8, 16, 32$. Considere tanto el error de reconstrucción como el desempeño en clasificación (vía kNN) de cada representación. Comente.

```
from sklearn.decomposition import PCA
from sklearn.neighbors import KNeighborsClassifier
pca = PCA(n_components=32)
pca.fit(x_train)
pca_train = pca.transform(x_train)
pca_test = pca.transform(x_test)
clf = KNeighborsClassifier(10)
clf.fit(pca_train, y_train)
print 'PCA SCORE %.2f' % clf.score(pca_test,y_test)
```

(g) ⋆⋆ Modifique el autoencoder básico construido en (a) para implementar un *deep autoencoder* (deep AE), es decir, un autoencoder con al menos dos capas ocultas. Demuestre experimentalmente que este autoencoder puede mejorar la compresión obtenida por PCA utilizando el mismo número de dimensiones d'. Experimente con $d' = 2, 4, 8, 16, 32$ y distintas profundidades ($L = 2, 3, 4$). Considere en esta comparación tanto el error de reconstrucción como el desempeño en clasificación (vía kNN) de cada representación. Comente.

```
target_dim = 2 #try other and do a nice plot
input_img = Input(shape=(784,))
encoded1 = Dense(1000, activation='relu')(input_img)
encoded2 = Dense(500, activation='relu')(encoded1)
encoded3 = Dense(250, activation='relu')(encoded2)
encoded4 = Dense(target_dim, activation='relu')(encoded3)
decoded4 = Dense(250, activation='relu')(encoded4)
decoded3 = Dense(500, activation='relu')(encoded3)
decoded2 = Dense(1000, activation='relu')(decoded3)
decoded1 = Dense(784, activation='sigmoid')(decoded2)
autoencoder = Model(input=input_img, output=decoded1)
encoder = Model(input=input_img, output=encoded3)
autoencoder.compile(optimizer=SGD(lr=1.0), loss='binary_crossentropy')
autoencoder.fit(x_train,x_train,nb_epoch=50,batch_size=25,shuffle=True,
                        validation_data=(x_val, x_val))
autoencoder.save('my_autoencoder_768x1000x500x250x2.h5')
from sklearn.decomposition import PCA
from sklearn.neighbors import KNeighborsClassifier
pca = PCA(n_components=target_dim)
pca.fit(x_train)
```

(h) Elija algunas de las representaciones aprendidas anteriormente y visualícelas usando la herramienta TSNE disponible en la librería *sklearn*. Compare cualitativamente el resultado con aquel obtenido usando PCA con el mismo número de componentes.

```
nplot=5000 #warning: mind your memory!
encoded_train = encoder.predict(x_train[:nplot])
from sklearn.manifold import TSNE
model = TSNE(n_components=2, random_state=0)
encoded_train = model.fit_transform(encoded_train)
colors={0:'b',1:'g',2:'r',3:'c',4:'m',5:'y',6:'k',7:'orange',8:'darkgreen',9:'maroon'}
markers={0:'o',1:'+',2: 'v',3:'<',4:'>',5:'^',6:'s',7:'p',8:'*',9:'x'}
plt.figure(figsize=(10, 10))
for idx in xrange(0,nplot):
    label = y_train[idx]
    line = plt.plot(encoded_train[idx][0], encoded_train[idx][1],
                color=colors[label], marker=markers[label], markersize=6)
pca_train = pca.transform(x_train)
```

```
encoded_train = pca_train[:nplot]
... #plot PCA
```

(i) ⋆ Modifique el autoencoder construido en (a) para trabajar directamente sobre las imágenes de MNIST, sin tratarlas como vectores de 784 atributos, sino como matrices de tamaño $1 \times 28 \times 28$. Es posible lograr este objetivo utilizando capas convolucionales para definir el Encoder y el Decoder. Compare la calidad de la representación reducida obtenida por el nuevo autoencoder con aquella obtenida anteriormente utilizando el mismo número de dimensiones. Comente.

```
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = np.reshape(x_train, (len(x_train), 28, 28, 1))  #modify for th dim ordering
x_test = np.reshape(x_test, (len(x_test), 28, 28, 1))

input_img = Input(shape=(28, 28, 1))
x = Conv2D(16, 3, 3, activation='relu', border_mode='same')(input_img)
x = MaxPooling2D((2, 2), border_mode='same')(x)
x = Conv2D(8, 3, 3, activation='relu', border_mode='same')(x)
encoded = MaxPooling2D((2, 2))(x)
x = Conv2D(8, 3, 3, activation='relu', border_mode='same')(encoded)
x = UpSampling2D((2, 2))(x)
x = Conv2D(16, 3, 3, activation='relu', border_mode='same')(x)
x = UpSampling2D((2, 2))(x)
decoded = Conv2D(1, 3, 3, activation='sigmoid', border_mode='same')(x)
autoencoder = Model(input_img, decoded)
autoencoder.compile(optimizer='adadelta', loss='binary_crossentropy')
autoencoder.summary()
```

3.2 Denoising

Como hemos discutido en clases, un *denoising autoencoder* (dAE) es esencialmente un *autoencoder* entrenado para reconstruir ejemplos parcialmente corruptos. Varios autores han demostrado que mediante esta modificación simple es posible obtener representaciones más robustas y significativas que aquellas obtenidas por un AE básico. En esta sección exploraremos la aplicación más "natural" o "directa" del método.

(a) Genere artificialmente una versión corrupta de las imágenes en MNIST utilizando el siguiente modelo de ruido (masking noise): si $\mathbf{x} \in \mathbb{R}^d$ es una de las imágenes originales, la versión ruidosa $\tilde{\mathbf{x}}$ se obtiene como $\tilde{\mathbf{x}} = \mathbf{x} \odot \boldsymbol{\xi}$ donde $\odot$ denota el producto de Hadamard (componente a componente) y $\boldsymbol{\xi} \in \mathbb{R}^d$ es un vector aleatorio binario con componentes $\text{Ber}(p)$ independientes.

```
from numpy.random import binomial
noise_level = 0.1
noise_mask = binomial(n=1,p=noise_level,size=x_train.shape)
noisy_x_train = x_train*noise_mask
noise_mask = binomial(n=1,p=noise_level,size=x_val.shape)
noisy_x_val = x_val*noise_mask
noise_mask = binomial(n=1,p=noise_level,size=x_test.shape)
noisy_x_test = x_test*noise_mask
```

(b) Entrene un autoencoder para reconstruir las imágenes corruptas generadas en el ítem anterior. Mida el error de reconstrucción y evalúe cualitativamente (visualización de la imagen corrupta y reconstruida) el resultado para un subconjunto representativo de imágenes. Experimente diferentes valores de p en el rango $(0, 1)$.

```
# DEFINE YOUR AUTOENCODER AS BEFORE
autoencoder.fit(noisy_x_train, x_train, nb_epoch=50, batch_size=25,
                        shuffle=True, validation_data=(noisy_x_val, x_val))
```

(c) Genere artificialmente una versión corrupta de las imágenes en MNIST utilizando el siguiente modelo de ruido (Gaussian noise): si $\mathbf{x} \in \mathbb{R}^d$ es una de las imágenes originales, la versión ruidosa $\tilde{\mathbf{x}}$ se obtiene como $\tilde{\mathbf{x}} = \mathbf{x} + \boldsymbol{\xi}$ donde $\boldsymbol{\xi} \in \mathbb{R}^d$ es un vector aleatorio binario con componentes $\mathcal{N}(0, \sigma^2)$ independientes.

```
from numpy.random import standard_normal
devst = 0.5
noise_mask = devst*standard_normal(size=x_train.shape)
noisy_x_train = x_train*noise_mask
noise_mask = devst*standard_normal(size=x_val.shape)
noisy_x_val = x_val*noise_mask
noise_mask = devst*standard_normal(size=x_test.shape)
noisy_x_test = x_test*noise_mask
```

(d) Entrene un autoencoder para reconstruir las imágenes corruptas generadas en el ítem anterior. Mida el error de reconstrucción y evalúe cualitativamente (visualización de la imagen corrupta y reconstruida) el resultado para un subconjunto representativo de imágenes. Experimente diferentes valores de σ.

(e) ⋆ Utilice imágenes intencionalmente corruptas como en (a) para entrenar un AE con fines de reducción de dimensionalidad. Durante el entrenamiento, proceda exactamente como en (b), pero su objetivo no será hacer *denoising* sino obtener una representación comprimida de alta calidad de las imágenes originales. Al final del entrenamiento, mida el error de reconstrucción como el desempeño en clasificación (vía kNN como en la sección anterior) de la representación obtenida. Esta idea es esencialmente la que presenta en [**?**]. Comente.

3.3 Pre-entrenamiento

En esta sección utilizaremos un AE para pre-entrenar redes profundas. Como hemos discutido en clases, el efecto esperado es regularizar el modelo, posicionando el modelo de partida en una buena zona del espacio de parámetros.

(a) Construya y entrene una red FF para clasificar las imágenes de MNIST. Utilice SGD básico con tasa de aprendizaje fija $\eta = 1$ y no más de 50 "epochs". Para empezar, utilice una arquitectura $768 \times 1000 \times 1000 \times 10$ y funciones de activación sigmoidales. Determine error de clasificación alcanzado por el modelo en el conjunto de test.

```
from keras.models import Sequential
model = Sequential()
model.add(Dense(1000, activation='sigmoid', input_shape=(784,)))
model.add(Dense(1000, activation='sigmoid'))
model.add(Dense(10, activation='softmax'))
model.summary()
optimizer_ = SGD(lr=1.0)
model.compile(optimizer=optimizer_,loss='binary_crossentropy', metrics=['accuracy'])
model.fit(x_train, Y_train,nb_epoch=50, batch_size=25,
              shuffle=True, validation_data=(x_val, Y_val))
model.save('ReluNet-768x1000x1000x10-NFT-50epochs.h5')
#TRAINING CAN THEN BE RESUMED FROM THIS POINT :-)
```

(b) Construya y entrene una red neuronal profunda para clasificar las imágenes de MNIST utilizando la arquitectura propuesta en (a) y pre-entrenando los pesos de cada capa mediante un autoencoder básico. Proceda en modo clásico, es decir, entrenando en modo no-supervisado una capa a la vez y tomando como input de cada nivel la representación (entrenada) obtenida en el nivel anterior. Después del entrenamiento efectúe un entrenamiento supervisado convencional (*finetunning*). Compare los resultados de clasificación sobre el conjunto de pruebas con aquellos obtenidos en (a), sin pre-entrenamiento. Evalúe también los resultados antes del *finetunning*. Comente.

```
## Load and preprocess MNIST as usual
from keras.datasets import mnist

###AUTOENCODER 1
input_img1 = Input(shape=(784,))
encoded1 = Dense(n_hidden_layer1,activation=activation_layer1)(input_img1)
decoded1 = Dense(784, activation=decoder_activation_1)(encoded1)
autoencoder1 = Model(input=input_img1, output=decoded1)
encoder1 = Model(input=input_img1, output=encoded1)
autoencoder1.compile(optimizer=optimizer_, loss=loss_)
autoencoder1.fit(x_train, x_train, nb_epoch=epochs_, batch_size=batch_size_,
                          shuffle=True, validation_data=(x_val, x_val))
encoded_input1 = Input(shape=(n_hidden_layer1,))
autoencoder1.save('autoencoder_layer1.h5')
encoder1.save('encoder_layer1.h5')

###AUTOENCODER 2
x_train_encoded1 = encoder1.predict(x_train) #FORWARD PASS DATA THROUGH FIRST ENCODER
x_val_encoded1 = encoder1.predict(x_val)
x_test_encoded1 = encoder1.predict(x_test)

input_img2 = Input(shape=(n_hidden_layer1,))
encoded2 = Dense(n_hidden_layer2, activation=activation_layer2)(input_img2)
decoded2 = Dense(n_hidden_layer2, activation=decoder_activation_2)(encoded2)
autoencoder2 = Model(input=input_img2, output=decoded2)
encoder2 = Model(input=input_img2, output=encoded2)
autoencoder2.compile(optimizer=optimizer_, loss=loss_)
autoencoder2.fit(x_train_encoded1,x_train_encoded1,nb_epoch=epochs_,batch_size=batch_size_,
                           shuffle=True, validation_data=(x_val_encoded1, x_val_encoded1))
encoded_input2 = Input(shape=(n_hidden_layer2,))
autoencoder2.save('autoencoder_layer2.h5')
encoder2.save('encoder_layer2.h5')

#FINE TUNNING
from keras.models import Sequential
model = Sequential()
model.add(Dense(n_hidden_layer1, activation=activation_layer1, input_shape=(784,)))
model.layers[-1].set_weights(autoencoder1.layers[1].get_weights())
model.add(Dense(n_hidden_layer2, activation=activation_layer2))
model.layers[-1].set_weights(autoencoder2.layers[1].get_weights())
model.add(Dense(10, activation='softmax'))
model.summary()
model.compile(optimizer=optimizer_,loss='binary_crossentropy', metrics=['accuracy'])
model.fit(x_train, Y_train,nb_epoch=20, batch_size=25,
              shuffle=True, validation_data=(x_val, Y_val))
model.save('Net-768x1000x1000x10-finetunned.h5')
```

(c) Construya y entrene una red neuronal profunda para clasificar las imágenes de MNIST utilizando la arquitectura propuesta en (a) y pre-entrenando los pesos de cada capa mediante un *denoising autoencoder*. Compare los resultados con aquellos obtenidos en (a), (b) y (c). Comente.

(d) ⋆ Repita (b) usando funciones de activación *Tanh* y *ReLu*. Comente.

References

[1] Vincent, P., Larochelle, H., Lajoie, I., Bengio, Y., Manzagol, P. A. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. Journal of Machine Learning Research 11. pp 3371–3408, 2010.

[2] Hastie, T., Tibshirani, R., Friedman, J. (2009), The Elements of Statistical Learning, Second Edition. Springer New York Inc.

[3] Bishop, Christopher M. (1995). Neural Networks for Pattern Recognition, Clarendon Press.

[4] Krizhevsky, A., Hinton, G. (2009). Learning multiple layers of features from tiny images.

[5] Harrison, D. and Rubinfeld, D. (1978). Hedonic prices and the demand for clean air, Journal of Environmental Economics and Management, 5, 81-102

[6] Dalal, N., Triggs, B. (2005, June). Histograms of oriented gradients for human detection. In 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) (Vol. 1, pp. 886-893). IEEE.

[7] Forsyth, D. A., Ponce, J. (2002). Computer vision: a modern approach. Prentice Hall Professional Technical Reference.

[8] Yann LeCun, Leon Bottou, Yoshua Bengio, Patrick Haffner. (1998). *Gradient-based Learning Applied to Document Recognition.* Proceedings of the IEEE, 86(11), 2278-2324.

[9] Kunihiko Fukushima, Sei Miyake, Takayuki Ito. *Neocognitron: A neural network model for a mechanism of visual pattern recognition.* IEEE Transactions on Systems, Man, and Cybernetics 5 (1983): 826-834.

[10] Yann LeCun, Fu Jie Huang, and Leon Bottou. *Learning methods for generic object recognition with invariance to pose and lighting.* Proceedings of the 2004 Computer Vision and Pattern Recognition Conference. CVPR 2004. IEEE Computer Society, 2004.

[11] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. *Deep Sparse Rectifier Neural Networks.* International Conference on Artificial Intelligence and Statistics. 2011.

[12] Pierre Sermanet, Soumith Chintala, and Yann LeCun. *Convolutional neural networks applied to house numbers digit classification.* International Conference on Pattern Recognition (ICPR), 2012. IEEE, 2012.

[13] *Scikit-learn: Machine Learning in Python.* http://scikit-learn.org/stable/

LaTeX