Implementing Zero-Shot Learning :

An Embarassingly Simple Approach to Zero-Shot Learning (ICML 2015)

Sandipan Sarma

Quick Recap

ZSL objective:

Visual-Semantic mapping function learned for seen class objects + Semantic representations of unseen class object
= Unseen class object recognized without training on any of its examples!

Training : Capture knowledge of attributes
Inference: Use this knowledge to recognize new classes of objects

Towards ESZSL

A zero-shot learning approach that can be implemented in just one line of code !

Terminologies

- Signature of a class = Attribute vector of a class
- Semantic matrix (S : (a,z)) = Continous or binary
- Training examples (X : (d,m)) = d-dimensional CNN features
- Training labels (Y : (m,z)) = ground truths for each example in X
- Weight matrix (W : (d,z)) = Weights learnt
- Visual-to-semantic matrix (V : (d,a)) = Mapper

Working Principle (1/2)

$$\underset{W \in \mathbb{R}^{d \times z}}{\text{minimise}} \, L\left(X^\top W, Y\right) + \Omega\left(W\right)$$

$$\underset{V \in \mathbb{R}^{d \times a}}{\text{minimise}} \, L\left(X^\top V S, Y\right) + \Omega\left(V\right)$$

$$\underset{i}{\text{argmax}} \, x^\top V S_i'.$$

Working Principle (2/2)

$$\Omega(V; S, X) = \gamma \|VS\|_{\mathrm{Fro}}^2 + \lambda \left\|X^\top V\right\|_{\mathrm{Fro}}^2 + \beta \|V\|_{\mathrm{Fro}}^2$$

$$L(P, Y) = \|P - Y\|_{\mathrm{Fro}}^2.$$

$$\beta = \gamma\lambda$$

$$V = \left(XX^\top + \gamma I\right)^{-1} XYS^\top \left(SS^\top + \lambda I\right)^{-1}$$

Overall picture

An embarrassingly simple approach to zero-shot learning

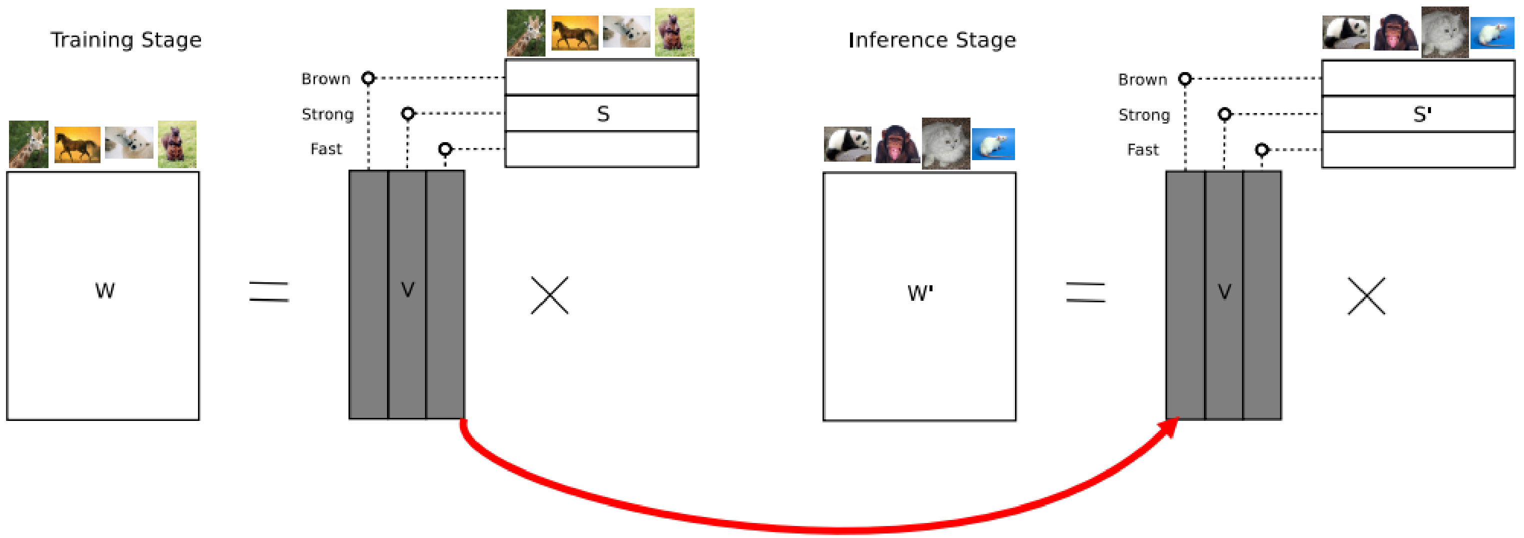

Figure 1. Summary of the framework described in Sec. 3. At training stage we use the matrix of signatures S together with the training instances to learn the matrix V (in grey) which maps from the feature space to the attribute space. At inference stage, we use that matrix V, together with the signatures of the test classes, S', to obtain the final linear model W'.

ESZSL - A Python-based implementation

Data acquisition and pre-processing

In [1]:
```
import numpy as np
import os
import scipy.io
from sklearn.metrics import classification_report,confusion_matrix
```

From the .mat files extract all the features from resnet and the attribute splits.

- The res101 contains features and the corresponding labels.
- att_splits contains the different splits for trainval, train, val and test set.

In [2]:
```
dataset = 'CUB'
res101 = scipy.io.loadmat('/home/sandipan/IITG/Academics/TA duty/July-Nov 2020/Z
SL lecture/zsl coding session/xlsa17/data/'+dataset+'/res101.mat')
att_splits = scipy.io.loadmat('/home/sandipan/IITG/Academics/TA duty/July-Nov 20
20/ZSL lecture/zsl coding session/xlsa17/data/'+dataset+'/att_splits.mat')
```

In [3]:
```
res101.keys()
```

Out[3]:
```
dict_keys(['__header__', '__version__', '__globals__', 'image_files', 'feature
s', 'labels'])
```

In [4]:
```
att_splits.keys()
```

Out[4]:
```
dict_keys(['__header__', '__version__', '__globals__', 'allclasses_names', 'at
t', 'original_att', 'test_seen_loc', 'test_unseen_loc', 'train_loc', 'trainval
_loc', 'val_loc'])
```

In [5]:
```
# Using the correct naming conventions to get the locations
trainval_loc = 'trainval_loc'
train_loc = 'train_loc'
val_loc = 'val_loc'
test_loc = 'test_unseen_loc'
```

We need the corresponding ground-truth labels/classes for each training example for all our train, val, trainval and test set according to the split locations provided. In this example we have used the CUB dataset which has 200 unique classes overall.

In [6]:

```
labels = res101['labels']
# np.squeeze() removes single-dimensional entries from the shape of an array.
labels_train = labels[np.squeeze(att_splits[train_loc]-1)]
labels_val = labels[np.squeeze(att_splits[val_loc]-1)]
labels_trainval = labels[np.squeeze(att_splits[trainval_loc]-1)]
labels_test = labels[np.squeeze(att_splits[test_loc]-1)]
```

In [7]:
```
labels_train[:10]
```

Out[7]:
```
array([[197],
       [198],
       [ 31],
       [ 25],
       [ 22],
       [ 86],
       [ 28],
       [136],
       [190],
       [177]], dtype=uint8)
```

In [8]:
```
unique_labels = np.unique(labels)
unique_labels
```

Out[8]:
```
array([  1,   2,   3,   4,   5,   6,   7,   8,   9,  10,  11,  12,  13,
        14,  15,  16,  17,  18,  19,  20,  21,  22,  23,  24,  25,  26,
        27,  28,  29,  30,  31,  32,  33,  34,  35,  36,  37,  38,  39,
        40,  41,  42,  43,  44,  45,  46,  47,  48,  49,  50,  51,  52,
        53,  54,  55,  56,  57,  58,  59,  60,  61,  62,  63,  64,  65,
        66,  67,  68,  69,  70,  71,  72,  73,  74,  75,  76,  77,  78,
        79,  80,  81,  82,  83,  84,  85,  86,  87,  88,  89,  90,  91,
        92,  93,  94,  95,  96,  97,  98,  99, 100, 101, 102, 103, 104,
       105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117,
       118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130,
       131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143,
       144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156,
       157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 168, 169,
       170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 181, 182,
       183, 184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195,
       196, 197, 198, 199, 200], dtype=uint8)
```

In a typical zero-shot learning scenario, there are no overlapping classes between training and testing phase, i.e the train classes are completely different from the test classes.

- During training phase we have z classes
- During the testing phase we have z' classes

In [9]:
```
train_labels_seen = np.unique(labels_train)
val_labels_unseen = np.unique(labels_val)
trainval_labels_seen = np.unique(labels_trainval)
test_labels_unseen = np.unique(labels_test)
print(len(train_labels_seen))
```

```
100
```

In [10]:
```
print("Number of overlapping classes between train and val:",len(set(train_label
s_seen).intersection(set(val_labels_unseen))))
print("Number of overlapping classes between trainval and test:",len(set(trainva
l_labels_seen).intersection(set(test_labels_unseen))))
```

```
Number of overlapping classes between train and val: 0
Number of overlapping classes between trainval and test: 0
```

Labels initially were given w.r.t entire set of classes. But after the split, classes for, say train, would be only, maybe 40 out of 50 overall classes. So accordingly changing the labels of the samples.

In [11]:

```
i = 0
for labels in train_labels_seen:
    labels_train[labels_train == labels] = i
    i = i+1
j = 0
for labels in val_labels_unseen:
    labels_val[labels_val == labels] = j
    j = j+1
k = 0
for labels in trainval_labels_seen:
    labels_trainval[labels_trainval == labels] = k
    k = k+1
l = 0
for labels in test_labels_unseen:
    labels_test[labels_test == labels] = l
    l = l+1

labels_train[:10]
```

Out[11]:

```
array([[96],
       [97],
       [23],
       [18],
       [15],
       [51],
       [21],
       [74],
       [93],
       [87]], dtype=uint8)
```

Let us denote the features $X \in [d \times m]$ available at training stage, where d is the dimensionality of the data, and m is the number of instances. We are using resnet features which are extracted from CUB dataset.

In [12]:
```
X_features = res101['features']
train_vec = X_features[:,np.squeeze(att_splits[train_loc]-1)]
val_vec = X_features[:,np.squeeze(att_splits[val_loc]-1)]
trainval_vec = X_features[:,np.squeeze(att_splits[trainval_loc]-1)]
test_vec = X_features[:,np.squeeze(att_splits[test_loc]-1)]

print("Features for train:", train_vec.shape)
print("Features for val:", val_vec.shape)
print("Features for trainval:", trainval_vec.shape)
print("Features for test:", test_vec.shape)
```

```
Features for train: (2048, 4702)
Features for val: (2048, 2355)
Features for trainval: (2048, 7057)
Features for test: (2048, 2967)
```

Each of the classes in the dataset have an attribute (a) description. This vector is known as the `Signature matrix` of dimension $S \in [0, 1]^{a \times z}$. For training stage there are z classes and z' classes for test $S \in [0, 1]^{a \times z'}$.

In [13]:

```
#Signature matrix
signature = att_splits['att']
train_sig = signature[:,(train_labels_seen)-1]
val_sig = signature[:,(val_labels_unseen)-1]
trainval_sig = signature[:,(trainval_labels_seen)-1]
test_sig = signature[:,(test_labels_unseen)-1]

print(signature)
```

```
[[0.0106384  0.         0.         ... 0.         0.         0.04378019]
 [0.0106384  0.01133243 0.         ... 0.00334966 0.11184146 0.02814441]
 [0.00709227 0.00944369 0.00742474 ... 0.         0.         0.        ]
 ...
 [0.00918617 0.00266542 0.         ... 0.00556558 0.08207164 0.06022509]
 [0.02526198 0.02132333 0.00885258 ... 0.         0.05836206 0.07695428]
 [0.02066889 0.05863916 0.01770516 ... 0.15027069 0.01823814 0.06189801]]
```

In [14]:
```
print("Signature for train:", train_sig.shape)
print("Signature for val:", val_sig.shape)
print("Signature for trainval:", trainval_sig.shape)
print("Signature for test:", test_sig.shape)
```

```
Signature for train: (312, 100)
Signature for val: (312, 50)
Signature for trainval: (312, 150)
Signature for test: (312, 50)
```

In [15]:

```
#params for train and val set
m_train = labels_train.shape[0]
n_val = labels_val.shape[0]
z_train = len(train_labels_seen)
z1_val = len(val_labels_unseen)

#params for trainval and test set
m_trainval = labels_trainval.shape[0]
n_test = labels_test.shape[0]
z_trainval = len(trainval_labels_seen)
z1_test = len(test_labels_unseen)
```

The ground truth is a one-hot encoded vector

In [16]:

```
#ground truth for train and val set
gt_train = 0*np.ones((m_train, z_train))
gt_train[np.arange(m_train), np.squeeze(labels_train)] = 1

#grountruth for trainval and test set
gt_trainval = 0*np.ones((m_trainval, z_trainval))
gt_trainval[np.arange(m_trainval), np.squeeze(labels_trainval)] = 1

print(gt_train[:1,:100])
```

```
[[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
  0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
  0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
  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. 0. 0. 0.]]
```

Training

The one-line code solution proposed.

```
V = inverse(XX' + γI) XYS' inverse(SS' + λI)
```

In [19]:

```
def find_hyperparameters(train_vec, train_sig, val_vec, val_sig, labels_val, val
_labels_unseen, gt_train):

    #train set
    d_train = train_vec.shape[0]
    a_train = train_sig.shape[0]

    accu = 0.10
    alph1 = 4
    gamm1 = 1

    #Weights
    V = np.zeros((d_train,a_train))
    for alpha in range(-3, 4):
        for gamma in range(-3,4):
            #One line solution
            part_1 = np.linalg.pinv(np.matmul(train_vec, train_vec.transpose())
+ (10**alpha)*np.eye(d_train))
            part_0 = np.matmul(np.matmul(train_vec,gt_train),train_sig.transpose
())
            part_2 = np.linalg.pinv(np.matmul(train_sig, train_sig.transpose())
+ (10**gamma)*np.eye(a_train))

            V = np.matmul(np.matmul(part_1,part_0),part_2)
            #print(V)

            #predictions
            outputs = np.matmul(np.matmul(val_vec.transpose(),V),val_sig)
            preds = np.array([np.argmax(output) for output in outputs])

            #print(accuracy_score(labels_val,preds))
            cm = confusion_matrix(labels_val, preds)
            cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
            avg = sum(cm.diagonal())/len(val_labels_unseen)

            if avg > accu:
```

```
                accu = avg
                alph1 = alpha
                gamm1 = gamma
                print('A: {}  G: {}  Avg. Acc: {}'.format(alph1, gamm1, avg))

    print("Final Alpha and gamma:",alph1, gamm1)
    return alpha, gamma
```

In [20]:
```
def train_ESZSL(alpha, gamma, trainval_vec, trainval_sig, gt_trainval):

    d_trainval = trainval_vec.shape[0]
    a_trainval = trainval_sig.shape[0]
    W = np.zeros((d_trainval,a_trainval))

    part_1_test = np.linalg.pinv(np.matmul(trainval_vec, trainval_vec.transpose
()) + (10**alpha)*np.eye(d_trainval))
    part_0_test = np.matmul(np.matmul(trainval_vec, gt_trainval),trainval_sig.tr
anspose())
    part_2_test = np.linalg.pinv(np.matmul(trainval_sig, trainval_sig.transpose
()) + (10**gamma)*np.eye(a_trainval))
    W = np.matmul(np.matmul(part_1_test,part_0_test),part_2_test)

    return W
```

In [21]:
```
alpha, gamma = find_hyperparameters(train_vec, train_sig, val_vec, val_sig, labe
ls_val, val_labels_unseen, gt_train)
```

```
A: -3  G: -3  Avg. Acc: 0.18585450209673038
A: -3  G: -2  Avg. Acc: 0.24968040478867645
A: -3  G: -1  Avg. Acc: 0.34653258722533276
A: -3  G: 0  Avg. Acc: 0.4093230883557413
A: 0  G: 0  Avg. Acc: 0.41178783716475353
A: 1  G: 0  Avg. Acc: 0.437161019212587
A: 2  G: -1  Avg. Acc: 0.44549392114624486
A: 2  G: 0  Avg. Acc: 0.4909643364371752
A: 3  G: -1  Avg. Acc: 0.5002568244155162
A: 3  G: 0  Avg. Acc: 0.5062013087372061
Final Alpha and gamma: 3 0
```

In [22]:
```
W = train_ESZSL(alpha, gamma, trainval_vec, trainval_sig, gt_trainval)
```

Testing / Inference

For inference stage,

```
argmax(x'VS)
```

Where S is the signature matrix of the test_set

In [23]:
```
#predictions
outputs_1 = np.matmul(np.matmul(test_vec.transpose(),W),test_sig)
preds_1 = np.array([np.argmax(output) for output in outputs_1])
```

In [24]:

```
cm = confusion_matrix(labels_test, preds_1)
cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
avg = sum(cm.diagonal())/len(test_labels_unseen)
print("The top 1% accuracy is:", avg*100)
```

```
The top 1% accuracy is: 40.96625130031721
```

Thank You !