Semi-supervised Experiment Problem Collection

ZHAO YANG MIN, Fudan University, Shanghai, China

April, 2017

Notation

- $\mathbf{X}_l := \{\mathbf{x}_1, \ldots, \mathbf{x}_l\}$ labeled data;
- $\mathbf{Y}_l := \{y_1, \ldots, y_l\}$ labels;
- $\mathbf{X}_u := \{\mathbf{x}_{l+1}, \ldots, \mathbf{x}_{l+u}\}$ unlabeled data;
- $f(\cdot)$ training model or learning machine;
- $g(\cdot)$ embedding mapping;
- θ, Θ model parameters and their set;
- $\mathcal{L}(\cdot)$ loss or object function;
- $\mathbf{W}_{ij}$ weighted adjacency matrix element;
- $\mathcal{N}(\cdot)$ neighborhood;
- $\mathcal{C}(\cdot)$ cluster;
- $||\cdot||$ norm;
- $dist(\cdot, \cdot)$ distance;
- $P(\cdot)$ probability or probability distribution;

Deep Learning

Motivation

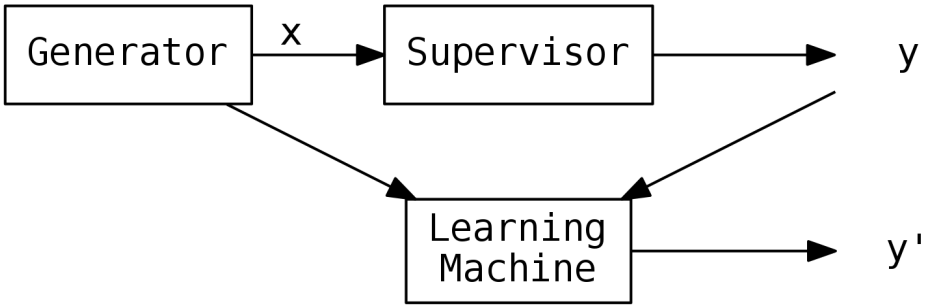
Why do semi-supervised deep learning for hyperspectral remote sensing data?

Because there is a profusion of unlabeled data points with a few labeled points.

Deep learning has strong ability of classifying complicated data, which is suitable for spectral space of sensing data. And the structure of neural network is easy to extend to give the model high ability of representing.

Semi-supervised Deep Learning

A general learning model has three components[1]:



1. Generator A generator would give vectors $\mathbf{x} \in \mathbf{X}$ according to the probability distribution $P(\mathbf{x})$ of the dataset $\mathbf{X}$.

2. Supervisor A supervisor would return an output value y to input vector $\mathbf{x}$ according to conditional distribution $P(y|\mathbf{x})$.

3. Learning Machine A learning machine would implement a set of functions $f(\mathbf{x}|\theta)$

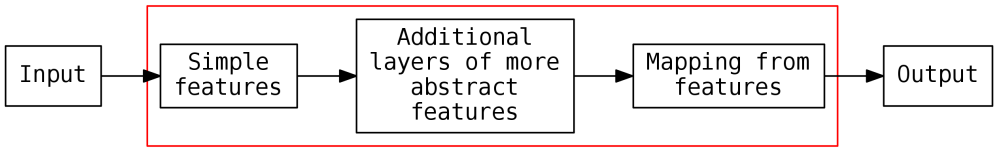
The generator gives a $\mathbf{x} \in \mathbf{X}$. The supervisor gives its label y. The acutual probability distribution of $\mathbf{X}$ is $P(\cdot)$, thus label would come from the conditional probability distribution $P(y|\mathbf{x})$.

For learning machine, it would generate an output $y' = f(\mathbf{x}|\theta), \theta \in \Theta$ for the input $\mathbf{x}$.

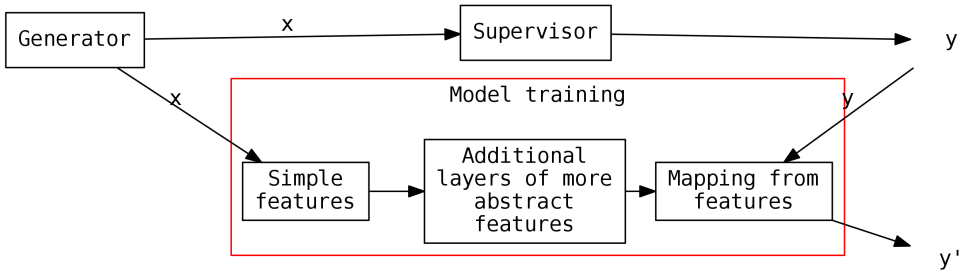
If the gap between the distribution $f(\cdot)$ and $P(\cdot)$ is $\mathcal{L}(y, y')$, then the task of model learning is to make $f(\cdot)$ approximates $P(\cdot)$ as much as possible:

$$\theta^\star = argmin_\theta \mathcal{L}(y, y')$$

For deep learning model, its working flow can be described as below[2]:



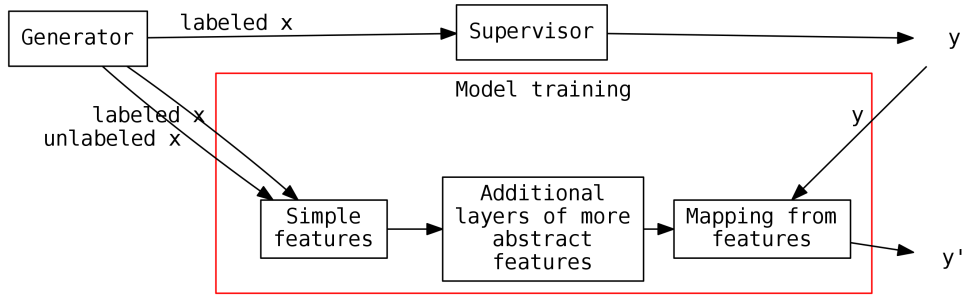
Thus the working frame of deep learning is:



For semi-supervised task, the difference is that:

$$\mathbf{X} = \mathbf{X}_l \cup \mathbf{X}_u$$

And the generator would give labeled data $\mathbf{x}_l \in \mathbf{X}_l$ and unlabeled data $\mathbf{x}_u \in \mathbf{X}_u$, while the supervisor would only supervise $P(y|\mathbf{x}_l)$.



Generally, people combine loss for labeled data: ParseError: KaTeX parse error: Expected 'EOF', got '$' at position 1: $\mathcal{L}_{\t…$

with the loss for unlabeled data:

$$\mathcal{L}_{\text{unlabeled}} = \mathcal{L}_u(f_u(\mathbf{X}_u|\theta_u))$$

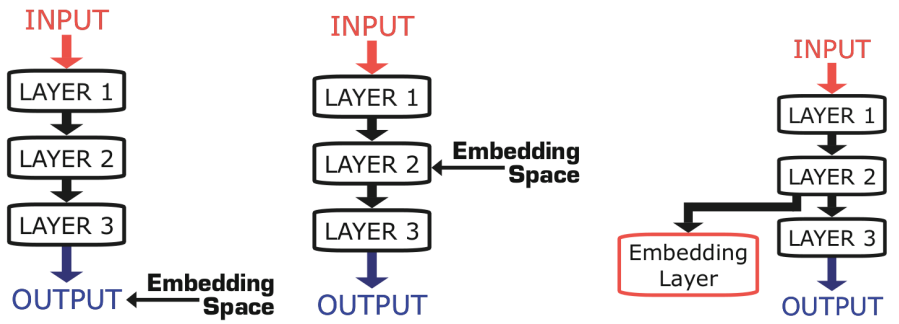
Thus the loss for semi-supervised is:

$$\mathcal{L} = F(\mathcal{L}_{\text{labeled}}, \mathcal{L}_{\text{unlabeled}})$$

Sometimes it takes the form as:

$$\mathcal{L} = \mathcal{L}_{\text{labeled}} + \alpha\mathcal{L}_{\text{unlabeled}}$$

Three different semi-supervised deep learning models[3]:



(a) Output (b) Internal (c) Auxiliary

Blind To Classifications

Try normalization if classification is not correct:

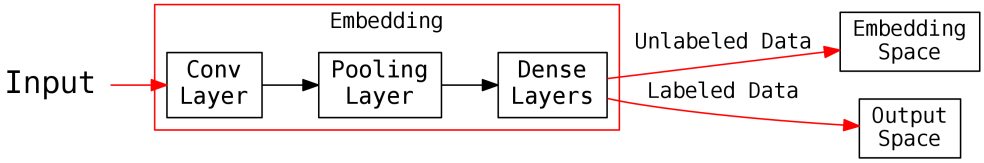
```
from sklearn import preprocessing
fn = '/home/yangminz/Semi-supervised_Embedding/dataset/PaviaU.mat'
dset = sio.loadmat(fn)
x = dset['paviaU'].reshape(shape_[0]*shape_[1], shape_[2])
x = preprocessing.scale(x)
x = x.reshape(shape_)
```

Design `JSON` configuration file for dataset

It would be much more easier to switch dataset when write the configurations in `JSON` at first.

Output Embedding

The feature mapping(neural network) is the same for labeled and unlabeled data. But because of the absence of label, loss function would be different.



The loss function in paper[4]:

$$\sum_{i=1}^{M} \mathcal{L}_l(f(\mathbf{x}_i), y_i) + \lambda \sum_{i,j=1}^{M+U} \mathcal{L}_u(g(\mathbf{x}_i), g(\mathbf{x}_j), \mathbf{W}_{ij})$$

$g(\cdot)$ is the embedding mapping, which has $g(\cdot) = f(\cdot)$ here for output model.

`Keras` Weight Reusing

Weights must be reused for $\mathbf{X}_l$ and $\mathbf{X}_u$, this is especially clear in output model. Originally, the code was designed like:

```
def forward(x):
    y = Dense(128, activation='tanh')(x)
    y = Dense(1, activation='sigmoid')(y)
    return y

y1 = forward(x1)
y2 = forward(x2)
```

However, weights would not be reused in this way because everytime the script calls `forward` , a new layer would be allocated when building up the graph. As a result, we get 2 seperate networks.

Layer initialization should be put outside the function:

```
layers = [
    Dense(128, activation='tanh'),
    Dense(1, activation='sigmoid')]

def forward(x):
    y = x
    for layer in layers:
```

```
        y = layer(y)

y1 = forward(x1)
y2 = forward(x2)
```

It's much more convinent since the structure of the network can easily be adjusted and extended in `list` like:

```
layer_3 = layer_1[:5] + layer_2[4:]
```

Such implemention is possible by `Keras` .

Random Sampling

According to the algorithm described in the paper, the actual loss is defined as:

$$\sum_{\mathbf{x}_i \in \mathbf{X}_l}^{\text{Epoch Num}} \left(\mathcal{L}_l(f(\mathbf{x}_i), y_i) + \lambda \sum_{\mathbf{W}_{ij}=0,1} \mathcal{L}_u(f(\mathbf{x}_i), f(\mathbf{x}_j), \mathbf{W}_{ij}) \right)$$

Here $\mathcal{L}_l(\cdot, \cdot)$ is for labeled data, $\mathcal{L}_u(\cdot, \cdot, \cdot)$ is the embedding algorithm.

This is actually ***sampling***. And this sampling makes it easier to impletement code in `Tensorflow` because calculating neighbor matrix of a batch is difficult, especially when the selecting mechanism varies.

Motivation for Random Sampling

Actually this is the work of generator in the general learning model.

The reason for doing random sampling is that the information of neighborhood $\mathcal{N}(\mathbf{x}_i)$ would be weaken if we take a random batch here.

It is easy to understand that if we pick up a random batch of labeled data and unlabeled data:

$$\lambda \sum_{i,j=1}^{M+U} \mathcal{L}_u(f(\mathbf{x}_i), f(\mathbf{x}_j), \mathbf{W}_{ij})$$

Most $\mathbf{W}_{ij}$ would be zero since $\mathcal{N}(\mathbf{x}_i)$ is small compared with $\mathbf{X}$. However, the experiment shows that the information of $\mathcal{N}(\mathbf{x}_i)$ is very important.

Thus directly do sampling here. The generator gives a random labeled data $\mathbf{x}_i$, accompanied with a random neighbor $\mathbf{x}_j \in \mathcal{N}(\mathbf{x}_i), \mathbf{W}_{ij} = 1$ and a random unlabeled data $ \mathbf{x}_j \notin \mathcal{N}(\mathbf{x}*i), \mathbf{W}*{ij}=0$.

Besides, the sampling makes it easier to add cluster-based loss: $\mathcal{L}_{\text{cluster}}$, which would be introduced in the following pages.

`Tensorflow` Engineering

It is difficult to impletement this embedding loss within a batch. Because a batch of data in `tf.placeholder` cannot get $\mathbf{W_{ij}}$ with each other. If we have a batch in size 3 like `[x[0], x[1], x[2]]` , then we must do something like:

```
for i in [0, 1, 2]:
    for j in [0, 1, 2]:
        W[i][j] = get_weight(x, i, j)
```

The only method is to calculate the matrix outside `tf.placeholder` and then pass it into `tensorflow` like:

```
x = tf.placeholder(tf.float32, input_dim)
W = tf.placeholder(tf.float32, [batch_size, batch_size])
```

Remember to make the loss to scalar by `tf.reduce_sum` or `tf.reduce_mean` .

`nan` Problem

If the model converges to `nan` , there may be some problem with loss and gradient. The problem may be the loss algorithm. The loss to be ∞ and gradient to be ∞ can both cause `nan` problem. The way to solve it is to clip gradient:

```
opt = tf.train.AdagradOptimizer(lr)
gvs = opt.compute_gradients(loss_)
clipped_gvs = [(tf.clip_by_value(grad, -1.0, 1.0), var) for grad, var in gvs]
train_step = opt.apply_gradients(clipped_gvs)
```

$\mathcal{L}_l$

For the loss of labeled data $\mathcal{L}_l(\cdot,\cdot)$, we have different losses:

1. Quadratic loss

$$\mathcal{L}_l(f(\mathbf{x}_i),y_i)=||y_i-f(\mathbf{x}_i)||^2$$

2. Hinge loss

$$\mathcal{L}_l(f(\mathbf{x}_i),y_i)=\max(0,1-y_i\cdot f(\mathbf{x}_i))$$

3. Quadratic hinge loss

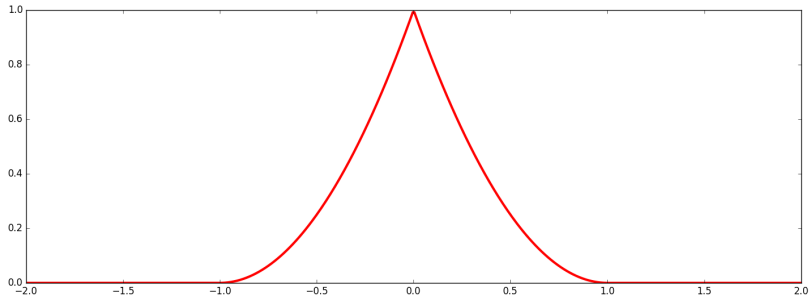
$$\mathcal{L}_l(f(\mathbf{x}_i),y_i)=\max(0,1-y_i\cdot f(\mathbf{x}_i))^2$$

4. Other losses

Watch Point

There should be no absolute $|\cdot|$ in hinge-like $\mathcal{L}_l$, which means no $ \max(0, 1 - | y_i \cdot f(\mathbf{x}_i) |)^2 $ nor $ \max(0, 1 - | y_i \cdot f(\mathbf{x}_i) |)$. The mechanism behind is how to design loss for *SVM* or *TSVM*:

with $|\cdot|$, the loss has a **non-convex hat** shape:



This is for unlabeled data to get close to the product $y_i \cdot f(\mathbf{x}_i)$ for labeled data.

$\mathcal{L}_l$ - one hot

In fact, y here should be one hot encoding. If this is not carefully dealt with, some strange but trivial errors may occur.

The classifier can distinguish classes from each other. However, the corresponding labels are misclassified. For example, `[2, 2, 4, 3, 3]` would be classified as `[1, 1, 6, 2, 2]` . They are correctly distinguished but wrongly classified.

The test data is the one to blame. One should be extremly careful when prepare “one-ho” labeling otherwise such weird error would occur.

For example, the following function

```
def OneHot(data, width):
    tmp = np.zeros((data.size, width))
    tmp[np.arange(data.size), data] = 1
    return tmp
```

returns one at their indices, `[0, 1, 0, 0]` for `data=1` and `width=4` . Sometimes similar function would give one-hot at their counting position, `[1, 0, 0, 0]` . There `data=1` means *the first digit* rather than index `1` .

$\mathcal{L}_u$

4 manifold or graph-based losses $\mathcal{L}_u$ for embedding $g(\cdot)$. It can be quite misleading if still use $f(\cdot)$ for embedding mapping here, since the neural network $f(\cdot)$ and $g(\cdot)$ mapping $\mathbf{X}$ to different spaces.

1. Multidimensional scaling

$$\mathcal{L}_u(g(\mathbf{x}_i), g(\mathbf{x}_j), \mathbf{W}_{ij}) = (||g(\mathbf{x}_i) - g(\mathbf{x}_j)|| - \mathbf{W}_{ij})^2$$

2. ISOMAP

3. Laplacian Eigenmaps

$$\sum_{ij} \mathcal{L}_u(g(\mathbf{x}_i), g(\mathbf{x}_j), \mathbf{W}_{ij}) = \sum_{ij} \mathbf{W}_{ij} ||g(\mathbf{x}_i) - g(\mathbf{x}_j)||^2$$

4. Siamese Networks

$$\mathcal{L}_u(g(\mathbf{x}_i), g(\mathbf{x}_j), \mathbf{W}_{ij}) = \begin{cases} ||g(\mathbf{x}_i) - g(\mathbf{x}_j)||_2, & \text{if } \mathbf{W}_{ij} = 1 \\ \max(0, m - ||g(\mathbf{x}_i) - g(\mathbf{x}_j)||_2)^2, & \text{if } \mathbf{W}_{ij} = 0 \end{cases}$$

$\mathbf{W}_{ij} \in \{0, 1\}$ indicates the neighbor relationship.

$\mathcal{L}_u$ - Metric

When we measure the size of a vector, we sometimes use norm:

$$||\mathbf{x}||_p = \left(\sum_i |x_i|^p \right)^{\frac{1}{p}}$$

For $p = 2$ and the subduction between 2 vectors, we have Euclidean distance:

$$dist(\mathbf{x}, \mathbf{y}) = ||\mathbf{x} - \mathbf{y}||_2 = \left(\sum_i |x_i - y_i|^2 \right)^{\frac{1}{2}} = \sqrt{(\mathbf{x} - \mathbf{y})^T (\mathbf{x} - \mathbf{y})}$$

The metric would influence the accuracy heavily. So the programer should be extremly careful and do not mix up the metrics.

For Laplacian Eigenmaps,

$$\sum_{ij} L(g(\mathbf{x}_i), g(\mathbf{x}_j), \mathbf{W}_{ij}) = \sum_{ij} \mathbf{W}_{ij} ||g(\mathbf{x}_i) - g(\mathbf{x}_j)||^2$$

can work well. But if we change the distance to Euclidean, that is $||g(\mathbf{x}_i) - g(\mathbf{x}_j)||_2 $, the accuracy drops drastically.

For Siamese Networks,

$$||g(\mathbf{x}_i) - g(\mathbf{x}_j)||_2 = \sqrt{(g(\mathbf{x}_i) - g(\mathbf{x}_j))^T (g(\mathbf{x}_i) - g(\mathbf{x}_j))}$$

and $(g(\mathbf{x}_i) - g(\mathbf{x}_j))^T(g(\mathbf{x}_i) - g(\mathbf{x}_j)) $ can both work normally.

$\mathcal{L}_u$ - $\mathbf{W}_{ij}$

The neighborhood, $\mathcal{N}(\cdot)$, is crucial for graph or manifold-based loss. We boardlly have 2 different mechanisms to make $\mathcal{N}(\cdot)$:

1. In the spectral space

   Compute the distance between the labeled point $\mathbf{x}_i$:

   $$\mathcal{D} = \{||\mathbf{x}_i - \mathbf{x}_j|| \quad | \quad \forall x_j \in \mathbf{X}_l \cup \mathbf{X}_u - \{\mathbf{x}_i\}\}$$

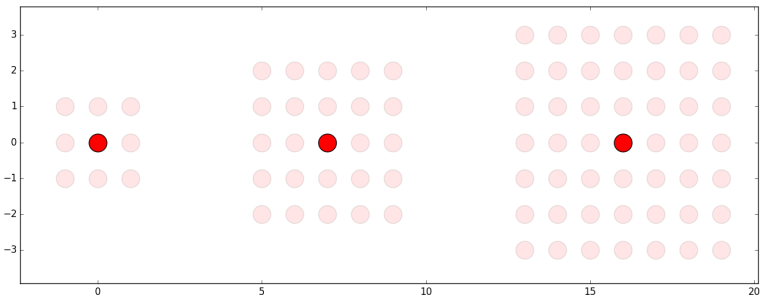
Then sort this distance set <eq><m

Euclidean distance is usually used in spectral space:

$$||\mathbf{x}_i - \mathbf{x}_j|| = \sqrt{(\mathbf{x}_i - \mathbf{x}_j)^T(\mathbf{x}_i - \mathbf{x}_j)}$$

2. In the spatial space

   Directly select the neighbors from the pixels around $\mathbf{x}_i$:



The 2 methods above provide us the neighborhood $\mathcal{N}(\mathbf{x}_i)$, and we can define the weight adjacency matrix as:

$$\mathbf{W}_{ij} = \begin{cases} 1, & \mathbf{x}_j \in \mathcal{N}(\mathbf{x}_i) \\ 0, & \mathbf{x}_j \notin \mathcal{N}(\mathbf{x}_i) \end{cases}$$

However, the time costs for these 2 methods varies greatly. It is apparent that the spatial way is much more faster. Besides, the accuracy in the spatial space is higher than the spectral space.

improve Output Model

There several ways to improve the accuracy of output model:

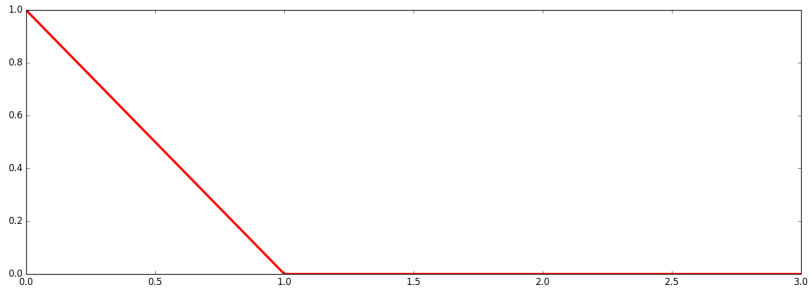
1. Change the supervised loss $\mathcal{L}_l$;
2. Change the unspervised loss $\mathcal{L}_u$;
3. Add cluster loss $\mathcal{L}_{\text{cluster}}$;

And one can try different radius of neighborhood.

These improvements reveal what is crucial to the accuracy of the model and data. The following results are based on `Pavia University` dataset[5].

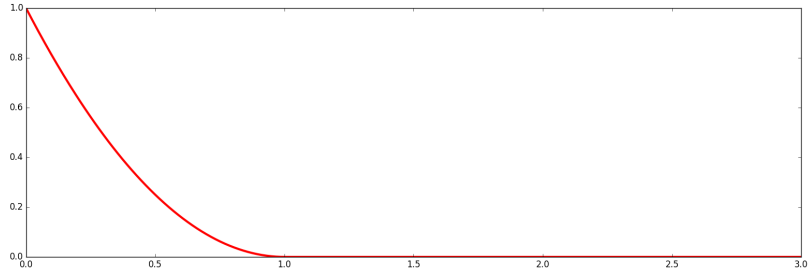
Change $\mathcal{L}_l$

We can replace the symmetric hinge loss: ParseError: KaTeX parse error: Expected 'EOF', got '$' at position 1: $\mathcal{L}_l (…$



with symmetric quadratic hinge loss:

ParseError: KaTeX parse error: Expected 'EOF', got '$' at position 1: $\mathcal{L}_l (…$



There is an improvement in accuracy:

sampling	supervised	manifold	α	cluster	β	Grids	Accuracy
100000	`hinge`	`LE`	1.0	`sym_hinge`	0.0	`3 * 3`	68.72
100000	`quadratic`	`LE`	1.0	`sym_hinge`	0.0	`3 * 3`	71.54

For comparision, I offer the result of $|\cdot|$ or symmetric(which is meaningless):

sampling	supervised	manifold	α	cluster	β	Grids	Accuracy
100000	`sym_hinge`	`LE`	1.0	`sym_hinge`	0.0	`3 * 3`	61.67
100000	`sym_quadratic`	`LE`	1.0	`sym_hinge`	0.0	`3 * 3`	68.71

Change $\mathcal{L}_{\text{manifold}}$

In the experiment, we found that Laplacian Eigenmaps(“LE”) is better than Siamese Networks(“SN”):

sampling	supervised	manifold	α	cluster	β	Grids	Accuracy
100000	`quadratic`	`SN`	1.0	`sym_hinge`	0.0	`3 * 3`	58.89
100000	`quadratic`	`LE`	1.0	`sym_hinge`	0.0	`3 * 3`	71.73

In code implementation, take advantage of $\mathbf{W}_{ij} \in \{0, 1\}$. Add $\mathbf{W}_{ij}$ and $1 - \mathbf{W}_{ij}$ in front of the terms to avoid condition discussion so that we can build up computation graph in `Tensorflow` :

$$\mathcal{L}_u(g(\mathbf{x}_i), g(\mathbf{x}_j), \mathbf{W}_{ij}) = \mathbf{W}_{ij}||g(\mathbf{x}_i) - g(\mathbf{x}_j)||_2 + (1 - \mathbf{W}_{ij}) \max(0, m - ||g(\mathbf{x}_i) - g(\mathbf{x}_j)||_2)^2$$

If we look into these 2 embedding algorithms:

1. Laplacian Eigenmaps

$$\mathcal{L}_u(g(\mathbf{x}_i), g(\mathbf{x}_j), \mathbf{W}_{ij}) = \mathbf{W}_{ij}||g(\mathbf{x}_i) - g(\mathbf{x}_j)||^2$$

2. Siamese Networks

$$\mathcal{L}_u(g(\mathbf{x}_i), g(\mathbf{x}_j), \mathbf{W}_{ij}) = \begin{cases} ||g(\mathbf{x}_i) - g(\mathbf{x}_j)||_2, & \text{if } \mathbf{W}_{ij} = 1 \\ \max(0, m - ||g(\mathbf{x}_i) - g(\mathbf{x}_j)||_2)^2, & \text{if } \mathbf{W}_{ij} = 0 \end{cases}$$

We can find that: when $\mathbf{W}_{ij} = 1$, the 2 algorithms give the same loss: $(g(\mathbf{x}_i) - g(\mathbf{x}_j))^T(g(\mathbf{x}_i) - g(\mathbf{x}_j)) $. However, when $\mathbf{W}_{ij} = 0$, the loss of LE disappears while SN still has contribution $\max(0, m - ||g(\mathbf{x}_i) - g(\mathbf{x}_j)||_2)^2$.

So we can conclude that in this task, the neighbor information is extremly important while the random unlabeled loss would distract the model training.

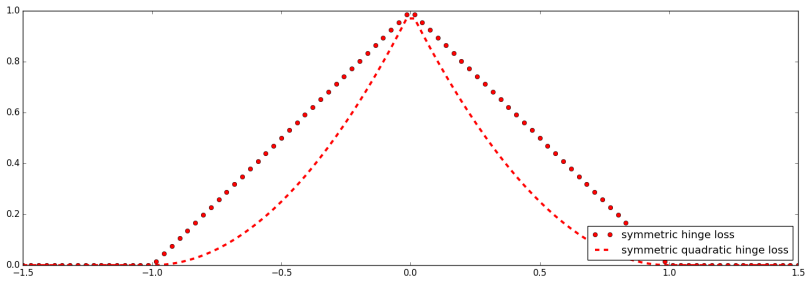
Add $\mathcal{L}_{\text{cluster}}$

The cluster loss is based on the cluster assumption[6]:

> **Cluster assumption: If points are in the same cluster, they are likely to be of the same class.**

The embedding algorithm above for unlabeled data is graph or manifold based. We can add cluster loss to the total loss and see how they influence the accuracy. Thus we can know the preference between manifold and cluster losses.

Symmetric Hinge Loss is a good candidate:



$$\mathcal{L}_{\text{cluster}}(\mathbf{x}) = \max(0, 1 - |\mathbf{x}|)$$

or its square:

$$\mathcal{L}_{\text{cluster}}(\mathbf{x}) = \max(0, 1 - |\mathbf{x}|)^2$$

Thus the unsupervised loss $\mathcal{L}_u$ can be decomposed as:

$$\mathcal{L}_u = \alpha\mathcal{L}_{\text{manifold}} + \beta\mathcal{L}_{\text{cluster}}$$

or:

$$\sum_{\mathbf{x}_i \in \mathbf{X}_{\text{labeled}}}^{\text{Epoch Num}} \left[\mathcal{L}_l(f(\mathbf{x}_i), y_i) + \sum_{\mathbf{W}_{ij}=0,1} \alpha \cdot \mathcal{L}_{\text{manifold}}(g(\mathbf{x}_i), g(\mathbf{x}_j), \mathbf{W}_{ij}) + \beta \cdot \mathcal{L}_{\text{cluster}} \right]$$

in detail(sampling). Here $\alpha + \beta = 1.0$.

Cluster in spatial space

In spatial space, we can consider the cluster space $\mathcal{C}(\mathbf{x}_i)$ as $\mathcal{N}(\mathbf{x}_i)$. This is reasonable since the information of the surface has some kind of "continuity". Here we have:

$$\mathcal{L}_{\text{cluster}} = \max(0, 1 - |g(\mathbf{x}_j)|)^2, \quad \mathbf{x}_j \in \mathcal{C}(\mathbf{x}_i) = \mathcal{N}(\mathbf{x}_i)$$

In fact we only sample a random neighbor $\mathbf{x}_j \in \mathcal{N}(\mathbf{x}_i)$ here. Though it seems that the term $\beta \mathcal{L}_{\text{cluster}}$ does not contain the information of labeled data, if we rewrite the loss as:

$$\sum_{\mathbf{W}_{ij}=0,1} \beta \mathbf{W}_{ij} \max(0, 1 - |g(\mathbf{x}_j)|)^2$$

we can learn that the condition:

$$\mathbf{W}_{ij} = 1 \Longleftrightarrow \mathbf{x}_j \in \mathcal{N}(\mathbf{x}_i)$$

guarantees that the $\mathbf{x}_i$ information is included.

The spatial way using $\mathcal{N}(\mathbf{x}_i)$ as cluster $\mathcal{C}(\mathbf{x}_i)$ works with the following code lines:

```
yl = forward(dconf['netName_label'], xl)
yle = forward(dconf['netName_embed'], xl)
yn = forward(dconf['netName_embed'], xn)
yu = forward(dconf['netName_embed'], xu)

loss_label = supervised_loss('quadratic')(yl, yt)
loss_manifold = graph_loss('LE')(yle, yn, 1.0) + graph_loss('LE')(yle, yu, 0.0)
# WATCH OUT HERE FOR CLUSTER
loss_cluster = cluster_loss('sym_quadratic')(yn)
```

Under the setting of

sampling	supervised	manifold	cluster	Grids
100000	`hinge`	`LE`	`sym_quadratic`	`3 * 3`

we can explore the preference of accuracy between $\mathcal{L}_{\text{manifold}}$ and $\mathcal{L}_{\text{cluster}}$:

α	β	**accuracy**	α	β	**accuracy**
0.0	0.0	68.94	0.5	0.5	70.09
0.0	1.0	69.56	0.6	0.4	70.78
0.1	0.9	72.91	0.7	0.3	69.00
0.2	0.8	71.54	0.8	0.2	62.79
0.3	0.7	70.43	0.9	0.1	69.78
0.4	0.6	72.16	1.0	0.0	67.39

with another setting:

sampling	**supervised**	**manifold**	**cluster**	**Grids**
100000	`quadratic`	`LE`	`sym_quadratic`	`3 * 3`

the preference:

α	β	**accuracy**	α	β	**accuracy**
0.0	0.0	70.69	0.5	0.5	73.02
0.0	1.0	85.05	0.6	0.4	70.78
0.1	0.9	85.39	0.7	0.3	72.01
0.2	0.8	84.45	0.8	0.2	72.94
0.3	0.7	80.25	0.9	0.1	71.89
0.4	0.6	76.73	1.0	0.0	71.44

A great leap here.

Cluster in spectral space

Another way to generate a cluster is in the spectral space. To achieve this, we can do clustering first. The result of K-means is not good at all: even if we use all the data to train, the accuracy is only around `12%`.

In this way, cluster $\mathcal{C}(\cdot)$ is generated by K-means algorithm:

$$\mathcal{L}_{\text{cluster}} = \max(0, 1 - |g(\mathbf{x}_j)|)^2, \quad \mathbf{x}_j \in \mathcal{C}(\mathbf{x}_i)$$

And we can see, even if the clustering algorithm is as rough as K-means, it can improve the accuracy. The mechanism behind may be that the spectral space is more structured after clustering than before. Thus though K-means itself cannot help directly, pre-clustering can improve the result.

sampling	supervised	manifold	cluster	Grids
100000	`quadratic`	`LE`	`sym_quadratic`	`3 * 3`

α	β	accuracy	α	β	accuracy
0.0	0.0	70.69	0.5	0.5	74.47
0.0	1.0	89.59	0.6	0.4	72.45
0.1	0.9	89.93	0.7	0.3	72.66
0.2	0.8	90.13	0.8	0.2	71.76
0.3	0.7	84.01	0.9	0.1	71.24
0.4	0.6	81.04	1.0	0.0	71.44

Under the same experiment setting, the accuracy of *Houston* dataset remains at `85%` , *indian* improves from `80%` to `85%` .

In implementation, we can use K-means to train at first(K-means can be easily implemented with `scikit-learn`), and then save the result to an independent file:

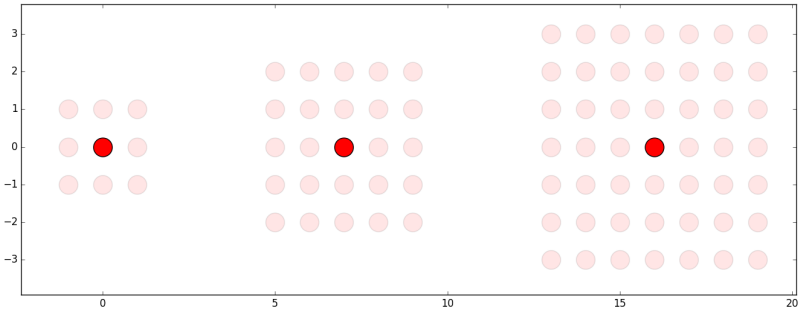
```
model = cluster.KMeans(n_clusters=dconf['nb_class'])
model.fit(x)
for i in range(dconf['shape'][0]):
    for j in range(dconf['shape'][1]):
        ypred = model.predict(dset.x[i][j].reshape(1, -1))
        label_cluster[i][j] = ypred[0]
np.save('../dataset/%s_cluster.npy'%dconf['name'], label_cluster)
```

And use this file in deep learning training. Thus the code should be modified to:

```
yl = forward(dconf['netName_label'], xl)
yle = forward(dconf['netName_embed'], xl)
yn = forward(dconf['netName_embed'], xn)
yu = forward(dconf['netName_embed'], xu)
yc = forward(dconf['netName_embed'], xc)

loss_label = supervised_loss('quadratic')(yl, yt)
loss_manifold = graph_loss('LE')(yle, yn, 1.0) + graph_loss('LE')(yle, yu, 0.0)
# WATCH OUT HERE FOR CLUSTER
loss_cluster = cluster_loss('sym_quadratic')(yc)
```

Radius of $\mathcal{N}(\mathbf{x}_i)$

Can we improve the robustness and overcome overfitting by increasing the radius r of neighborhood $\mathcal{N}(\mathbf{x}_i)$? The price is accuracy:



Thus the neighborhood plays an important role in the model.

sampling	supervised	manifold	α	cluster	β	Grids	Accuracy
100000	`quadratic`	`LE`	1.0	`sym_hinge`	0.0	`3 * 3`	71.73
100000	`quadratic`	`LE`	1.0	`sym_hinge`	0.0	`5 * 5`	68.91
100000	`quadratic`	`LE`	1.0	`sym_hinge`	0.0	`7 * 7`	65.82

Conclusion

We can learn the importance of spatial neighborhood from the methods above:

1. Change $\mathcal{L}_{\text{manifold}}$

   The comparison of LE and SN shows that a random unlabeled data $\mathbf{x}_j \notin \mathcal{N}(\mathbf{x}_i)$ would disturb the manifold structure. This is because only $\mathcal{N}(\mathbf{x}_i)$ contains the information of $\mathbf{x}_i$ to reinforce the result.

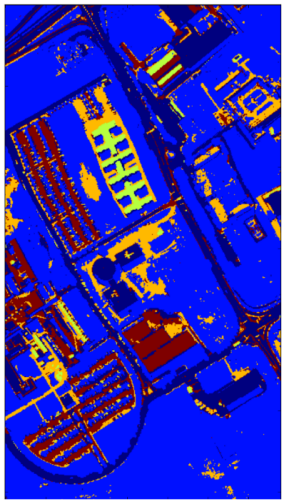
2. Add $\mathcal{L}_{\text{cluster}}$

   According to the experiment, the task prefers $\mathcal{L}_{\text{cluster}}$ to $\mathcal{L}_{\text{manifold}}$. In fact, for LE, $\mathcal{L}_{\text{manifold}}$ would devolve into $\mathcal{L}_{\text{cluster}}$. They emphasize on the cluster information of $\mathcal{N}(\mathbf{x}_i)$. Thus we can regard that a combination of different $\mathcal{L}_{\text{cluster}}$ may help to improve accuracy.

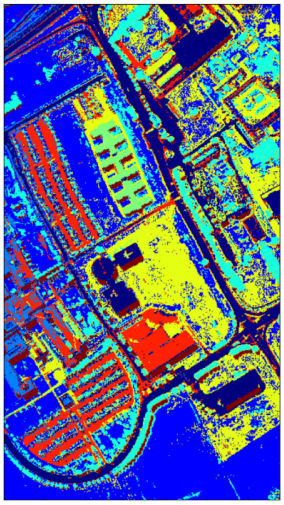
3. Change radius of $\mathcal{N}(\mathbf{x}_i)$

   Narrower $\mathcal{N}(\mathbf{x}_i)$ means $\mathbf{x}_j$ is more relative to $\mathbf{x}_i$, thus the accuracy would improve.

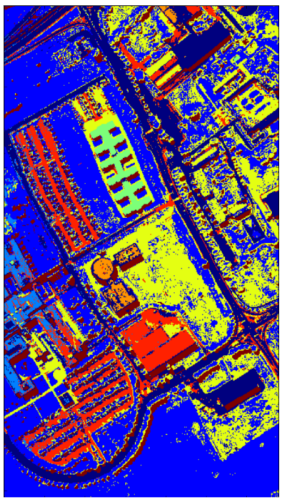
Danger of Overfitting



(a) Supervised only CNN: `71.28%`



(b) Semi-supervised CNN, spectral cluster: `90.13%`



(c) Semi-supervised CNN, spatial cluster: `86.39%`

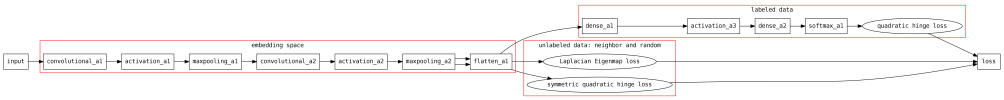
The blue area shows that there exists the danger of overfitting since they are not in the same class while they are classified into the same one.

Internal Model

The internal model add a semi-supervised loss to the k^{th} hidden layer directly:

$$\sum_{i=1}^{M} \mathcal{L}_l(f(\mathbf{x}_i), y_i) + \lambda \sum_{i,j=1}^{M+U} \mathcal{L}_u(f^k(\mathbf{x}_i), f^k(\mathbf{x}_j), \mathbf{W}_{ij})$$

We still do sampling here.



Preference on internal model under the setting:

sampling	supervised	manifold	cluster	Grids
100000	`hinge`	`LE`	`sym_hinge`	`3 * 3`

α	β	**accuracy**	α	β	**accuracy**
0.0	0.0	65.94	0.5	0.5	43.16
0.0	1.0	65.49	0.6	0.4	44.13
0.1	0.9	44.38	0.7	0.3	43.74
0.2	0.8	43.63	0.8	0.2	43.38
0.3	0.7	43.45	0.9	0.1	43.65
0.4	0.6	43.88	1.0	0.0	43.57

Here $\alpha = 0.0$ and $\beta = 0.0$ indicates supervised learning without unsupervised term.

Watch Point

It's vague in *output model*, but clear here: $\mathbf{x}_j$ must go through the embedding space. Actually, the embedding space $g(\cdot)$ is different from the neural network for labeled data $f(\cdot)$. Thus technically, the model should be written as:

1. output model

$$\sum_{i=1}^{M} \mathcal{L}_l(f(\mathbf{x}_i), y_i) + \lambda \sum_{i,j=1}^{M+U} \mathcal{L}_u(g(\mathbf{x}_i), g(\mathbf{x}_j), \mathbf{W}_{ij})$$

in which $g(\cdot) = f(\cdot)$.

2. output model with $\mathcal{L}_{\text{cluster}}$, sampling

$$\sum_{\mathbf{x}_i \in \mathbf{X}_{\text{labeled}}}^{\text{Epoch Num}} \left[\mathcal{L}_l(f(\mathbf{x}_i), y_i) + \sum_{\mathbf{W}_{ij}=0,1} \alpha \cdot \mathcal{L}_{\text{manifold}}(g(\mathbf{x}_i), g(\mathbf{x}_j), \mathbf{W}_{ij}) + \beta \cdot \mathcal{L}_{\text{cluster}} \right]$$

```
in which <eq><span class="katex"><span class="katex-mathml"><math><semantics><mro
```

3. internal model

$$\sum_{i=1}^{M} \mathcal{L}_l(f(\mathbf{x}_i), y_i) + \lambda \sum_{i,j=1}^{M+U} \mathcal{L}_u(g(\mathbf{x}_i), g(\mathbf{x}_j), \mathbf{W}_{ij})$$

in which $g(\cdot) = f^k(\cdot)$.

Be careful not to make it:

$$\sum_{i=1}^{M} \mathcal{L}_l(f(\mathbf{x}_i), y_i) + \lambda \sum_{i,j=1}^{M+U} \mathcal{L}_u(f(\mathbf{x}_i), g(\mathbf{x}_j), \mathbf{W}_{ij}), \quad g(\cdot) \neq f(\cdot)$$

I made this mistake in *auxiliary model*. The wrong code is:

```
yl = forward(dconf['netName_label'], xl)
yn = forward(dconf['netName_embed'], xn)
yu = forward(dconf['netName_embed'], xu)
```

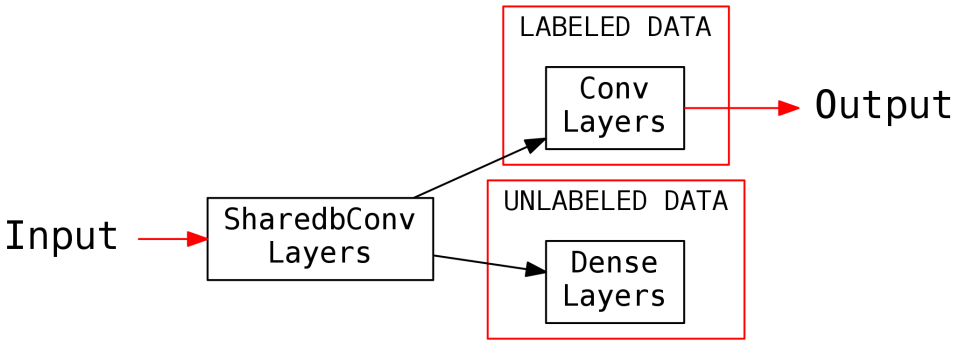
```
yt = tf.placeholder(tf.float32, output_dim)

loss_label = supervised_loss('abs_quadratic')(yl, yt)
loss_manifold = graph_loss('LE')(yl, yn, 1.0) + graph_loss('LE')(yl, yu, 0.0)
```

And the correct one is:

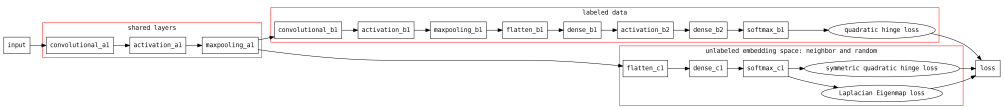
```
yl = forward(dconf['netName_label'], xl)
yle = forward(dconf['netName_embed'], xl)
yn = forward(dconf['netName_embed'], xn)
yu = forward(dconf['netName_embed'], xu)
yt = tf.placeholder(tf.float32, output_dim)

loss_label = supervised_loss('abs_quadratic')(yl, yt)
loss_manifold = graph_loss('LE')(yle, yn, 1.0) + graph_loss('LE')(yle, yu, 0.0)
```

Auxiliary Model

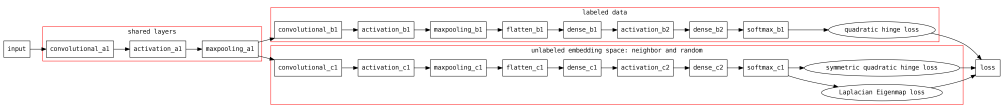
General description of auxiliary model is:



NN Auxiliary:



CNN Auxiliary:



We obey the conclusion from output model, give the experiment setting as:

sampling	supervised	manifold	α	cluster	β	Grids
100000	quadratic	LE	0.1	sym_quadratic	0.9	3 * 3

Then compare NN auxiliary and CNN auxiliary:

auxiliary model	accuracy
NN	69.12
CNN	66.59

The accuracy of CNN embedding is a little lower than NN embedding. Maybe we need not to stack complicate network layers for unlabeled data $\mathbf{x}_j$, since it has been separated from $\mathbf{x}_i$.

Preference on NN auxiliary model with $\mathcal{C}(\mathbf{x}_i) = \mathcal{N}(\mathbf{x}_i)$:

sampling	supervised	manifold	cluster	Grids
100000	`quadratic`	`LE`	`sym_quadratic`	`3 * 3`

α	β	accuracy	α	β	accuracy
0.0	0.0	68.70	0.5	0.5	66.51
0.0	1.0	67.32	0.6	0.4	71.89
0.1	0.9	65.74	0.7	0.3	72.39
0.2	0.8	68.56	0.8	0.2	68.58
0.3	0.7	66.16	0.9	0.1	64.92
0.4	0.6	64.72	1.0	0.0	68.03

Here $\alpha = 0.0$ and $\beta = 0.0$ indicates supervised learning without unsupervised term.

Transductive Support Vector Machine

SVM would take a lot of memory, e.g. $O(n^3)$ for kernel. Thus to run *S3VM*, the size of dataset should not be large. *SVM* is especially useful when the number of data is not so big, otherwise other brute algorithm would also work well, which makes *SVM* algorithm less competitive. So try less data to fit *S3VM* unless you have very large memory.

It's better to be noticed that currently it is not easy at all to find a repository for *multi-classification semi-supervised SVM*. The most common code is *multi-classification SVM* or *binary S3VM*.

Discriminative Restricted Boltzmann Machines

Semi-supervised DRBM algorithm is proposed in paper[7]:

$$\mathcal{L}_{\text{semi-sup}}(\mathcal{D}_{\text{train}}, \mathcal{D}_{\text{unlab}}) = \mathcal{L}_{\text{TYPE}}(\mathcal{D}_{\text{train}}) + \beta \mathcal{L}_{\text{unsup}}(\mathcal{D}_{\text{unlab}})$$

$$= -\sum_{i=1}^{|\mathcal{D}_{\text{train}}|} \log p(y_i|\mathbf{x_i}) - \beta \sum_{i=1}^{|\mathcal{D}_{\text{unlab}}|} \log p(\mathbf{x_i})$$

or

$$= -\sum_{i=1}^{|\mathcal{D}_{\text{train}}|} \log p(y_i|\mathbf{x_i}) - \alpha \sum_{i=1}^{|\mathcal{D}_{\text{train}}|} \log p(y_i, \mathbf{x_i}) - \beta \sum_{i=1}^{|\mathcal{D}_{\text{unlab}}|} \log p(\mathbf{x_i})$$

The implementation of semi-supervised DRBM consulted other people's code. The most common repository is *RBM*. I only found two repositories implemented by `Tensorflow` or `Theano`, both of them are supervised only.

My extension to semi-supervised:

```
def _calc_unsupervised_grads(self, x):
    x0, h0, x1, h1 = self._gibbs_un_sampling_step(x)

    h0 = tf.reshape(h0, [-1, self.num_hidden, 1])
    x0 = tf.reshape(x0, [-1, self.num_visible, 1])
    h1 = tf.reshape(h1, [-1, self.num_hidden, 1])
    x1 = tf.reshape(x1, [-1, self.num_visible, 1])

    # no gradient for U, but use zero to take place here
    d_U_unsup = tf.constant(0.0)
    d_W_unsup = tf.reduce_mean(tf.matmul(h0, x0, transpose_b=True) - tf.matmul(h1

    d_b_unsup = tf.reduce_sum(x0 - x1, 0)
    d_c_unsup = tf.reduce_sum(h0 - h1, 0)
    d_d_unsup = tf.constant(0.0)
    return d_U_unsup, d_W_unsup, d_b_unsup, d_c_unsup, d_d_unsup
```

We know that the difference between supervised and semi-supervised is one-hot label `y`. Thus make the gradients related to `y` be `tf.constant(0.0)` would make them contribute nothing to the loss. Meanwhile `Tensorflow` can work even we have no method to figure out the shape of the `tf.placeholder` batch.

The warning point that this kind of implementation is not checked by Contrastive Divergence.

Label Propagation

`Scikit-learn` has an API for *label propagation*. This graph-based algorithm is described in the paper[8]. The implementation is quite easy with `sklearn`, and the result is quite good:

PaviaU	Houston	India
77.81	35.86	63.17

Ladder Network

There is a `Tensorflow` implemented *Ladder Network* on github. The major problem is in *houston* dataset. It's weird because the *LN* with the same architecture works well on *paviaU* and *indian*. But for *houston*, the prediction will be `[0, 0, 0, ..., 0]`. A little debug tip here:

1. Run the model in very short time(10 epoches for example) first and print all predicted labels. In this way, one can save a lot of time to verify if his model works before entering the time-costing training.
2. Make sure that training dataset is reasonable. The split of labeled and unlabeled training data should reasonable: each class should at least have a certain and balanced number of data.
3. Transform dataset. Shuffle, crop or PCA the true dataset and then train it to see if model can make multi-label prediction. If the model still make prediction like `[0, ..., 0]`, there must be something wrong with the dataset instead of the model.
4. Remove the predicted class. If the model always predict `2` for dataset with `[0, 1, 2, 3, 4]`, try remove the `2` data and train with `[0, 1, 3, 4]` to see if the model still only predict `2`.
5. Check the weights and layer output. Check weights after training to see if it is the weights that cause the problem. Especially the weight of the last layer.

Then we can finally locate the problem is weight and loss become `nan` in training. This problem is most probably caused by infinite gradient or loss. Thus clip it would make the situation much better.

However, one may find that Tensorflow's `clip_by_value` would give errors. This is mainly because the gradient itself is `nan` and loss is `inf`. Here in *houston* dataset, when the labeled loss is around `2.5`, the unlabeled loss is `3000`, `5000`, `inf`.

This problem can be traced back to the selection of unlabeled data instead of the architecture of the network, if one break at the batch input code line to check. There are so many duplicated unlabeled data. This will not affect EmbedCNN since it picks only one unlabeled sample at a time, but will greatly influence *LN*.

Now can check if it is the unlabeled data unshuffled. Because if not shuffled, the data in batch would be very similar or even the same arccording to the physical properties of dataset. The problem can be solved by shuffling.

The original loss in the paper[9] is:

$$C = C_{\text{supervised cost}} + C_{\text{unsupervised denosing cost}}$$

Give the unsupervised part a coefficient `alpha < 1.0` so that the unstructured unlabeled part would not strongly affect the labeled one:

$$C = C_{\text{supervised cost}} + \alpha C_{\text{unsupervised denosing cost}}$$

1. Vapnik, The nature of statistical learning theory ↩

2. Bengio, Deep learning book ↩

3. Jason, Deep Learning via Semi-supervised Embedding ↩

4. Jason, Deep Learning via Semi-supervised Embedding ↩

5. Hyperspectral Remote Sensing Scenes - GIC ↩

6. Semi-Supervised Learning, MIT press, edited by Olivier Chapelle, Introduction to Semi-Supervised Learning. ↩

7. Classification using Discriminative Restricted Boltzmann Machines ↩

8. Yoshua Bengio, Olivier Delalleau, Nicolas Le Roux. Semi-Supervised Learning (2006), pp. 193-216 ↩

9. Semi-Supervised Learning with Ladder Network ↩