Report

Deep Learning

Author:
Manh Tu Vu

Supervisor:
MarieBeurton-Aimar

September 21, 2017

Contents

Abstract

In this project, we use Deep Learning method to automatic classify images from https://heobs.org into 4 classes, include:

Heritage
A place of cultural, historical, or natural significance for a group or society.

Beings
Any form of life, such as a plant or a living creature, whether human or other animal.

Scenery
Any form of landscapes which show little or no human activity and are created in the pursuit of a pure, unsullied depiction of nature, also known as scenery.

Other
Any other type of image that doesn't represent a photograph, such as painting, illustration, any object.

1 Introduction

2 Preparing dataset

2.1 Fetch all images

The entire image dataset described on the text file "photos.txt" line by line. Each line includes the image id and image description.

```
5a36f382-dbdf-11e6-95fd-d746d863c3eb | Những người ăn xin  | vie
5a36f382-dbdf-11e6-95fd-d746d863c3eb | Mendiants  | fra
17be8122-dbe0-11e6-860c-5fea02802d0a | Chợ Cũ (3) | vie
17be8122-dbe0-11e6-860c-5fea02802d0a | Vieux marché (3) | fra
400286c8-dbe1-11e6-bb4d-ff975c68de04 | Ngân hàng Đông Dương  | vie
400286c8-dbe1-11e6-bb4d-ff975c68de04 | La Banque de l'Indochine  | fra
```

In order to get the image dataset, we have to fetch each image one by one by join the image id with heobs cdn url https://cdn.heobs.org/photo/. For example, with the first line in the record above, we have the following URL:

```
https://cdn.heobs.org/photo/5a36f382-dbdf-11e6-95fd-d746d863c3eb
```

We wrote a small python script to automatic read this text file & download images one by one. Totally, we have 144564 images in our dataset.

2.2 Remove duplicate images

In the "photos.txt", some image has two languages and then, it consumes two lines. As the record above, we have six lines, but actually, a half of them was duplicated.
Because of the duplicate images doesn't help deep learning anything otherwise consume

more time to train. So, to reject those images, before fetching each image, we check if this image id already exists or not.
After removed duplicate, we have 142459 images left.

2.3 Remove broken images

After downloaded & look around all images, we found that it has a lot of broken images, which can't be displayable. So, we write a small script to filter all of those broken images automatically.
Finally, we have 89850 images left in our dataset.

3 Unsuppervised Deep Learning

Because of when classifying images by hand, it has some special case when one image may refer to more than one class. Thus, we need machine help us to make decisions by comparing two images are the same class or not. We also want to separate images of one class into multi unknown sub-classes. So, by using Unsupervision Deep Learning, we want to let the machine to classify a set of images unlabeled into some unknown classes.
After some research, we found the Unsupervised Deep Embedding for Clustering Analysis[1] paper, which propose a new method that simultaneously learns feature representations and cluster assignments using deep neural networks to classify unlabeled images. They also provide the implement code at https://github.com/piiswrong/dec

3.1 Trying to run the implement code with MNIST dataset

The DEC has two phases:

Phase 1: parameter initialization with a deep autoencoder

Phase 2: parameter optimization (i.e., clustering)

Because they already provide pretrained autoencoder weights, so we just have to test the phase 2 of DEC.
We already tried to run this implement code with the latest Caffe version (Jul 2017), but it doesn't work because this Caffe model required some deprecated parameters. However, because of the code delivery with Caffe and Docker, so, we continue to try with Docker and their Caffe version. The docker file in the original source code has exception but fixed by the following patch:

```
- liblmdb-dev libboost1.54-all-dev libatlas-base-de
+ liblmdb-dev libboost1.54-all-dev libatlas-base-dev
```

We'll first try to test with mnist dataset - the simplest example provided with the source code.

3.1.1 Failure when training with CPU

The implement code using GPU to train. However, I don't have the machine with GPU at this time, so I tried to move the entire code from GPU to CPU. However, it seem to the caffemodel phase 1 trained with GPU, and then it notwork with CPU version

```
multi_t_loss_layer.cpp:83] Forward_cpu not implemented.
multi_t_loss_layer.cpp:89] Backward_cpu not implemented.
```

3.1.2 Run success with GPU

After get the machine with GPU (Nvidia Quadro 4000) and installed Nvidia Docker. We build and deploy the source code into the Docker container.
After run success, the model generated **init.caffemodel** and **net.prototxt** files.
Because of both the paper and the implement code doesn't tell us how to achieve the result, so, we suppose that this code will generate a caffe model and then we have to write a caffe network deploy to make this work.

3.1.3 Build caffe deploy network model

We modify the net.prototxt by the following changes:

Remove the Data Layer(s) that were used for training
These four layers are no longer valid, as we will not be providing labelled data:

```
layers {
    name: "data"
    type: DATA
    top: "data"
    data_param {
        seek: '00000000'
        source: "mnist_total"
        backend: LEVELDB
        batch_size: 256
    }
    transform_param {
        scale: 1.0
    }
    include: { phase: TRAIN }
}
layers {
    name: "data"
    type: DATA
    top: "data"
    data_param {
        seek: '00000000'
        source: "mnist_total"
```

```
        backend: LEVELDB
        batch_size: 100
    }
    transform_param {
        scale: 1.0
    }
    include: { phase: TEST }
}
layers {
    name: "label"
    type: DATA
    top: "label"
    data_param {
        seek: '00000000'
        source: "train_weight"
        backend: LEVELDB
        batch_size: 256
    }
    transform_param {
        scale: 1.0
    }
    include: { phase: TRAIN }
}
layers {
    name: "label"
    type: DATA
    top: "label"
    data_param {
        seek: '00000000'
        source: "train_weight"
        backend: LEVELDB
        batch_size: 100
    }
    transform_param {
        scale: 1.0
    }
    include: { phase: TEST }
}
```

Remove any layer that is dependent upon data labels
The MULTI_T_LOSS Layer and the SILENCE Layer are still expecting labels, but there are none to provide, thus these layers are no longer needed as well:

```
layers {
  name: "loss"
  type: MULTI_T_LOSS
  bottom: "output"
```

```
  bottom: "label"
  blobs_lr: 1.
  blobs_lr: 0.
  blobs_lr: 0.
  top: "loss"
  top: "std"
  top: "ind"
  top: "proba"
  multi_t_loss_param {
    num_center: 10
    alpha: 1
    lambda: 2
    beta: 1
    bandwidth: 0.1
    weight_filler {
      type: 'gaussian'
      std: 0.5
    }
  }
}
layers {
  name: "silence"
  type: SILENCE
  bottom: "label"
  bottom: "ind"
  bottom: "proba"
}
```

Set the Network up to accept data
The MNist data is of size 28x28 and in gray. For simplicity we will keep the batch size at 1. This new data entry point in our network looks like this:

```
input: "data"
input_dim: 1
input_dim: 1
input_dim: 28
input_dim: 28
```

3.1.4 Acquire and analytics the test result

After run this network model with the **init.caffemodel**, we acquire the result describe as the following table:

	Predict 0	1	2	3	4	5	6	7	8	9
Actual 0	0	357	1012	0	1900	24	95	705	3	35
1	399	0	471	10	10	428	227	0	127	3011
2	348	237	11	785	643	302	707	18	684	441
3	0	106	169	2834	88	616	20	33	151	333
4	497	4	55	94	8	35	634	451	15	2278
5	3	20	2002	3	41	587	417	423	2	21
6	2	1083	1037	0	205	105	1067	69	0	568
7	11	0	32	0	284	5	23	42	551	3452
8	151	231	1312	144	27	444	130	243	1301	79
9	30	26	547	24	24	24	59	591	112	2750

Table 1: Mnist init.caffemodel test result

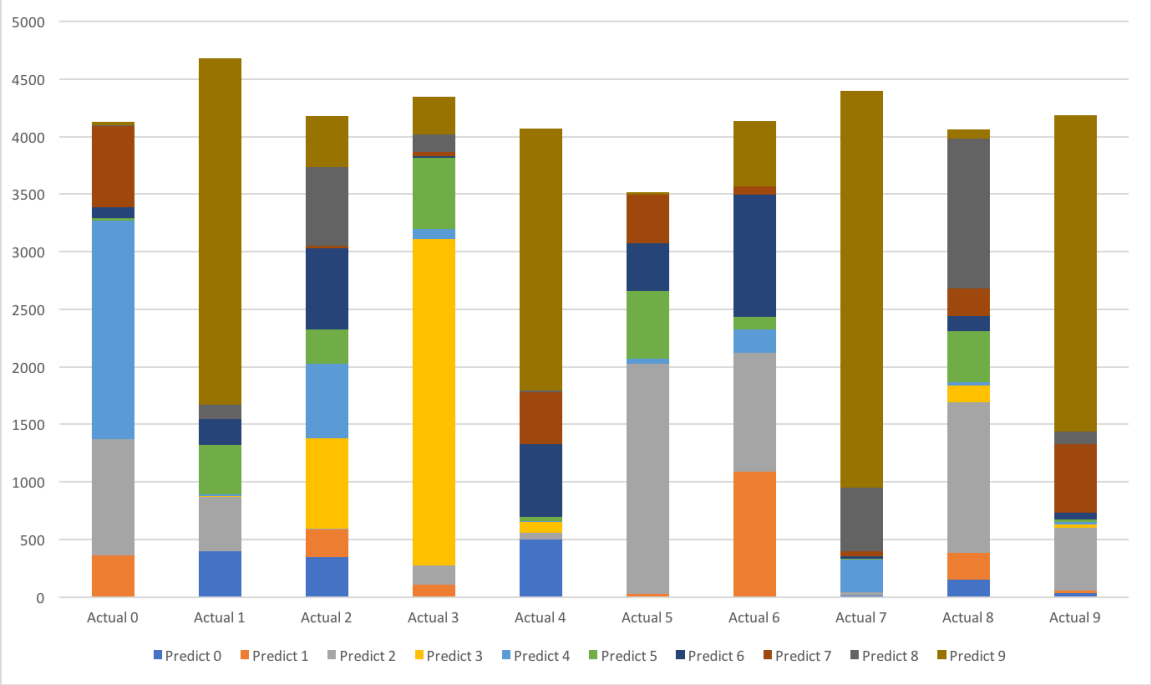

Figure 1: Mnist init.caffemodel test result

Because of Unsupervised Classification don't know what classes actually are. They just grouping the images, which they think there's the same into one class. So, we've to figure what these classes are.
As the result above, we can easy to found some thing incorrect here: this model predicts 9 with very high percent in many classes: 1, 4, 7 and 9.
The result scattered in different classes, so, we unable to know exactly what class actually are.

3.1.5 Found the correct way to achieve the result

After rereading the source code, carefully. I found that they were using KMeans Cluster to try to predict the closest cluster each sample in the test image belongs to.
The most important part is:

```
if (Y_pred != Y_pred_last).sum() < 0.001*N:
    print acc_list
    return acc, nmi
```

This is the terminal, the application achive the goal when the current predict (Y_pred) is not different with the last predict. The value 0.001*N is threshold.
So, at this time, we already have list of predicted images (in binary)

3.1.6 Convert binary images into displayable images

Because of the images used in the application is stored in binary & read directly into memory. So, to be able to get the classified result, we've to convert image stored in memory (RAM) to the displayable image file.

```
def dispSingleImg(X, n, fname=None):
    h = X.shape[1]
    w = X.shape[2]
    c = X.shape[3]
    buff = np.zeros((h, w, c), dtype=np.uint8)

    buff[0:h, 0:w, :] = X[n]

    if fname is None:
        cv2.imshow('a', buff)
        cv2.waitKey(0)
    else:
        cv2.imwrite(fname, buff)
```

X: array of image dataset

n: index of image

fname: destination to save

3.1.7 Classify image dataset into X classes

This feature put the images from dataset, which has the same class into one folder. This helps us to easy to verify the classified result

```
def classify_dataset(predicts, imgs, db):
    classes_dir = "classes_" + db
    if not os.path.isdir(classes_dir):
        os.makedirs(classes_dir)
    for idex, pred in enumerate(predicts):
```

```
        tmp_dir = os.path.join(classes_dir, str(pred))
        if not os.path.isdir(tmp_dir):
            os.makedirs(tmp_dir)
        dispSingleImg(imgs, idex, os.path.join(tmp_dir, str(idex) + ".jpg"))
```

We call this function when the application predict done

```
if (Y_pred != Y_pred_last).sum() < 0.001*N:
     classify_dataset(Y_pred, img, db)
     print acc_list
     return acc, nmi
```

3.1.8 Generate the test summary result

Because with mnist dataset, we already have the correct label of each image. So, to be able to summary the result, we add the following function to compare between predict & actual result:

```
def show_result(predicts, actuals):
    summary = {}
    for idx, value in enumerate(actuals):
        actual = str(value)
        predict = str(predicts[idx])
        if actual in summary:
            if predict in summary[actual]:
                summary[actual][predict] += 1
            else:
                summary[actual][predict] = 1
        else:
            summary[actual] = {}
    print summary
```

And call this function when the application predict done

```
if (Y_pred != Y_pred_last).sum() < 0.001*N:
     classify_dataset(Y_pred, img, db)
     show_result(Y_pred, Y)
     print acc_list
     return acc, nmi
```

3.1.9 Acquire and analytics the test result

After run this network model with the **init.caffemodel**, we acquire the result describe as the following table:

	Predict 0	1	2	3	4	5	6	7	8	9
Actual 0	17	35	53	15	1	141	6629	1	4	6
1	7658	14	16	8	153	3	1	9	2	3
2	278	15	20	229	6094	42	73	29	112	87
3	108	128	13	178	171	6	12	23	67	6434
4	65	8	3060	8	7	58	3	3612	4	0
5	60	4968	41	133	6	54	7	37	13	899
6	233	268	2	37	14	6124	187	9	0	1
7	251	16	336	17	103	1	22	317	6226	3
8	167	142	62	5922	32	36	73	72	23	295
9	80	12	3624	60	5	16	47	2910	97	106

Table 2: Mnist init.caffemodel test result

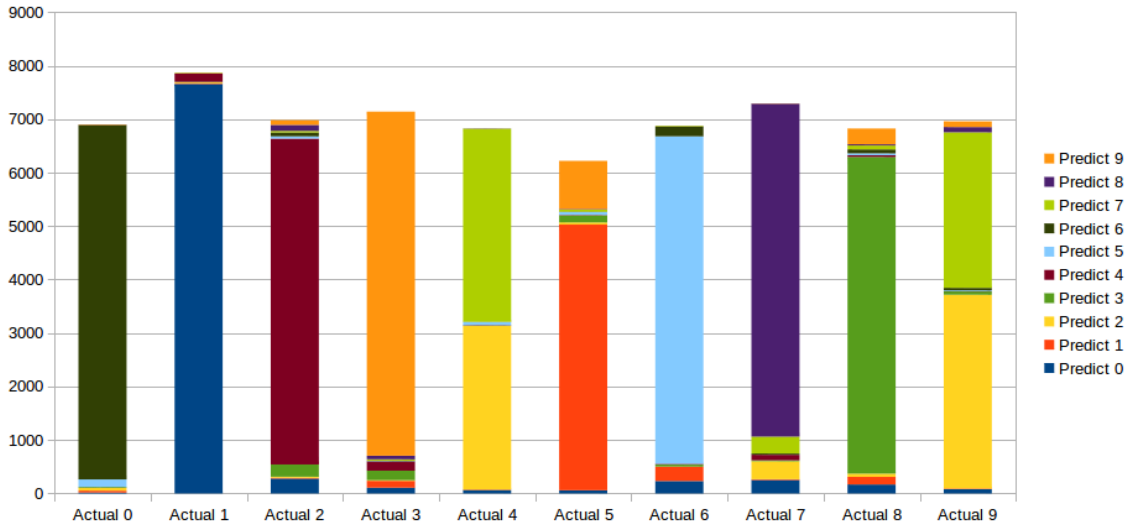

Figure 2: Mnist init.caffemodel test result v2

As the result above, we can easy to figure out that almost it can predict the images belong to which classes in high performance. Although it has a little confuse when predicting 2 and 7 but we can understand because of this two digit is similar.

Now we know how to receive the result of this implement source code. However, because of Mnist dataset is too simple. It just contains images, which has 28x28 pixel and only black & white colors while our image Dataset has both bigger and true color. So, we need to try with the more complex Dataset.

In the test Datasets they provide us has STL dataset, which closest with our Dataset (96 x 96 pixel, true color). We're going to try with this Dataset before try with we own dataset.

3.2 Trying to run the implement code with STL dataset

The application failure when execute **make_stl_data.py** python file to create STL LevelDB because of the "features.pyx" file is not delivery with the source code. I found that the author mention to the features.pyx at https://github.com/cvondrick/pyvision/blob/master/vision/features However, when I test this file, I got another error

```
Type Error: "int" object is not iterable
```

As the author say "It's been too long and I almost forgot about it, sorry. But you can try to refer to stackoverflow. I believe you can find your answer" (https://github.com/piiswrong/dec/issues/1). So, after some debug & research, I found that the problem caused by the parameter of **function hog** in **feature.pyx** is an image object. But we provided an numpy object. So, I fixed it by replaced the following line in **features.pyx**

```
line 48.  with, height = im.size
```

by the following line:

```
line 48.  with, height = im.shape[:2]
```

Finally, it works. The accuracy result is 35.7% (35.9% in the paper).

3.3 Preparing we own image dataset

Below is the code we using to generate & save data into LevelDB:

```
import sys
import os
import Image
import numpy as np
import cv2
import cv
from joblib import Parallel, delayed
import features
import dec

def load_data(images_dir):
    ims = [read(os.path.join(images_dir, filename)) for filename in
        os.listdir(images_dir)]
    X = np.array(ims, dtype='uint8')
    n_jobs = 10
    cmap_size = (6, 10)
    N = X.shape[0]

    H = np.asarray(Parallel(n_jobs=n_jobs)(delayed(features.hog)(X[i]) for i in
        xrange(N)))

    H = H.reshape((H.shape[0], H.size / N))

    X_small = np.asarray(Parallel(n_jobs=n_jobs)(delayed(cv2.resize)(X[i],
        cmap_size) for i in xrange(N)))
```

```
    crcb = np.asarray(Parallel(n_jobs=n_jobs)(delayed(cv2.cvtColor)(X_small[i],
        cv.CV_RGB2YCrCb) for i in xrange(N)))
    crcb = crcb[:, :, :, 1:]
    crcb = crcb.reshape((crcb.shape[0], crcb.size / N))

    feature = np.concatenate(((H - 0.2) * 10.0, (crcb - 128.0) / 10.0), axis=1)
    print feature.shape

    return feature, X[:, :, :, [2, 1, 0]]


def load_label(images_dir, classes, determine):
    return np.array([classes.index(filename.split(determine)[0]) for filename in
        os.listdir(images_dir)], dtype='uint8')


def load_named_label(images_dir):
    return np.array([filename for filename in os.listdir(images_dir)], dtype='str')

if __name__ == '__main__':
    classes = ["heritage", "being", "scenery", "other"]
    images_dir = sys.argv[1]
    read = lambda imname: np.asarray(Image.open(imname).convert("RGB"))
    if os.path.isdir(images_dir):
        X_train, img_train = load_data(images_dir)
        Y = load_label(images_dir, classes, "_")
        labeled = load_named_label(images_dir)
        p = np.random.permutation(X_train.shape[0])
        X_total = X_train[p]
        Y_total = Y[p]
        labeled_total = labeled[p]
        np.save("custom_named_label", labeled_total)
        img_total = img_train[p]
        dec.write_db(X_total, Y_total, 'custom_total')
        dec.write_db(img_total, Y_total, 'custom_img')
        N = X_total.shape[0] * 4 / 5
        dec.write_db(X_total[:N], Y_total[:N], 'custom_train')
        dec.write_db(X_total[N:], Y_total[N:], 'custom_test')
    else:
        raise Exception("Please specific image url")
```

3.4 Train with our dataset

After train with our dataset, we get accurency 68.74%

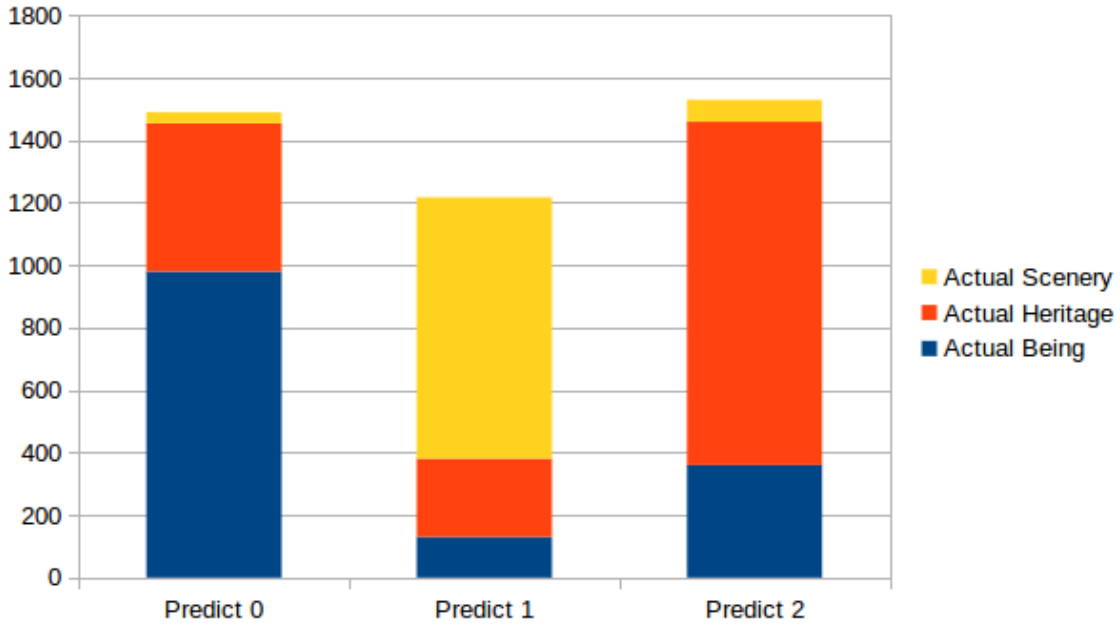

Figure 3: Unsupervised train result

4 Supervised Deep Learning

Supervised deep learning requires us to provide a training dataset, which includes a list of images labeled. However, our image dataset is not, and we can't classify the entire dataset by hand. So, we propose the following method, which includes three steps:

- Train a Convolution Neural Network (CNN) model with a small dataset based on the original dataset, which labeled by hand.
- Use the CNN model trained above to classify the entire original dataset
- Review & Train a CNN model with the original dataset.

4.1 Train a CNN model with a test dataset

4.1.1 Preparing test dataset based on original dataset

We created this dataset by random 1000 images from the original dataset and classify them by hand into four classes:

- Heritage
- Beings
- Scenery
- Other

Below is the summary of each class in our dataset:

	Being	Heritage	Scenery	Other
count	209	389	139	263

Table 3: Test dataset summary

Because in our dataset contain a lot of images, which have different resolution. So, we've to resize them all to the smallest resolution. The following script mainly to find & resize all images

```
def get_lowest_resolution(image_dir):
    min_width, min_height = 9999, 9999
    for root, dirs, files in os.walk(images_dir):
        for f in files:
            image = os.path.join(root, f)
            width, height = Image.open(image).size
            min_width = lower(min_width, width)
            min_height = lower(min_height, height)
    return min_width, min_height


def lower(first, last):
    if first < last:
        return first
    return last

if __name__ == '__main__':
    images_dir = sys.argv[1]
    if os.path.isdir(images_dir):
        images_dir = os.path.abspath(images_dir)
        w, h = get_lowest_resolution(images_dir)
        for root, dirs, files in os.walk(images_dir):
            for f in files:
                image = os.path.join(root, f)
                print image
                im = Image.open(image).resize((w, h), Image.BICUBIC)
                im.save(image)

    else:
        raise Exception("Please specific image url")
```

4.1.2 The caffe model for this CNN

Our model is reuse from bvlc_reference_caffenet model, which is a replication of AlexNet with a few modifications. The original bvlc reference caffenet was designed for a classification problem with 1000 classes. However, we just need to classify into 4 classes. So, we change num output of the last InnerProduct layer from 1000 to 4.

4.1.3 The slover definition

Our model using Stochastic Gradient Descent solver method. We run our model with 4000 iterators, drop leaning rate every 500 iterators and take a snapshot every 500 iterators.

```
net: "caffe_model/caffenet_train.prototxt"
test_iter: 500
test_interval: 500
base_lr: 0.001
lr_policy: "step"
gamma: 0.1
stepsize: 500
display: 50
max_iter: 4000
momentum: 0.9
weight_decay: 0.0005
snapshot: 1000
snapshot_prefix: "caffe_model/snapshot"
solver_mode: GPU
```

4.1.4 Acquire and analytics the test result

After train the model with the test dataset, we obtain the result with very high loss & low accuracy

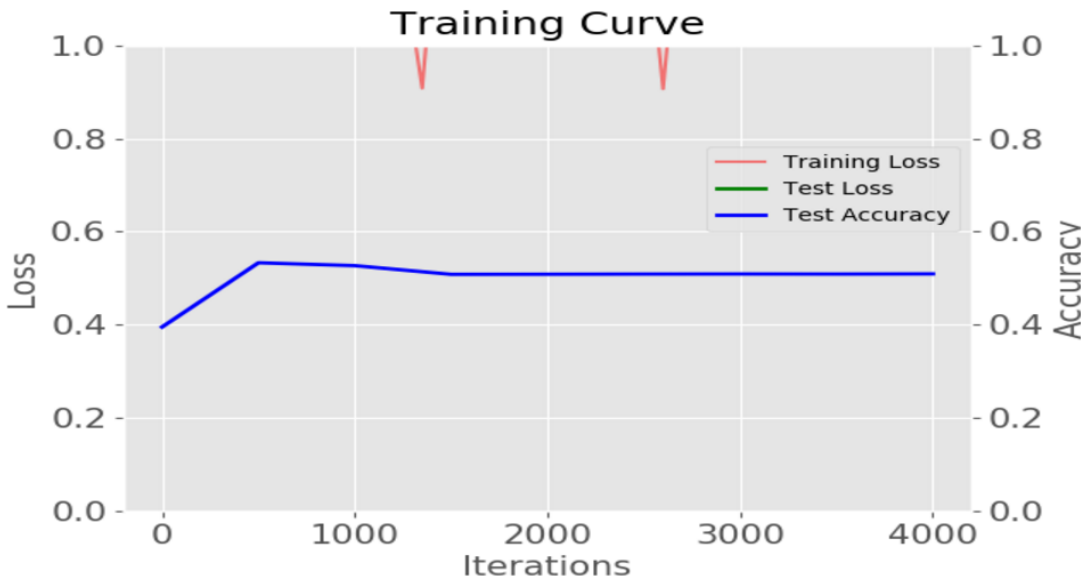

Figure 4: Train result

We tried to change the gamma parameter to 0.01, 0.001, 0.0001 but the result still the same. Trying to find the problem.

4.1.5 Resolution problem

Below is the summary of images dataset resolution. We assume that the image has resolution different no more than 10 pixels is the same solution.

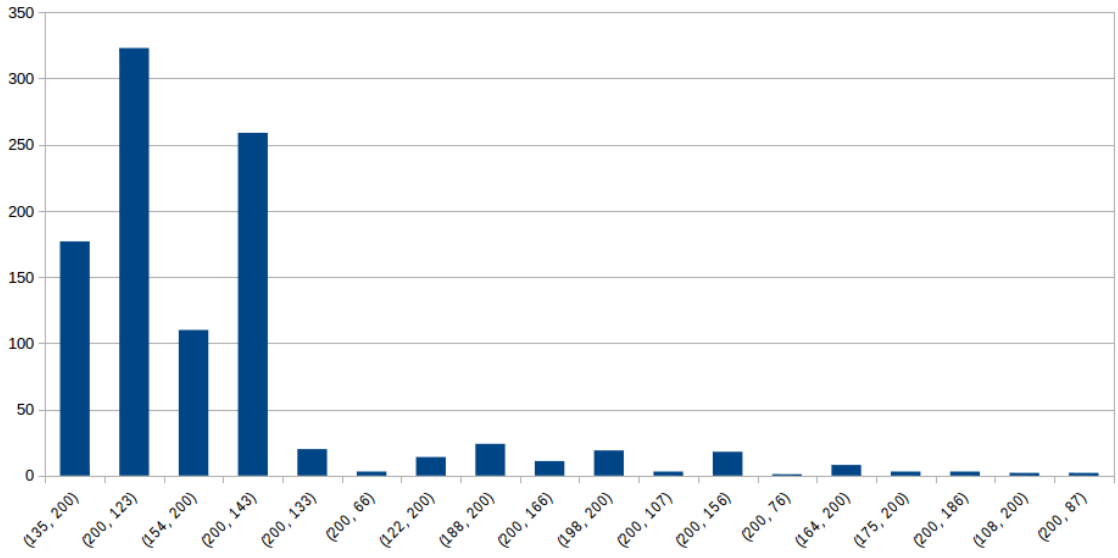

Figure 5: Test Dataset resolution summary

As the result above, we have many different resolutions. We can't just resize all of them into the smallest. So, we keep the images, which have resolution (width, height) from (200, 120) to (200, 165) and discard the others. Finally, we resize all images left to smallest resolution (200, 120) Below is the summary of each class in our dataset:

	Being	Heritage	Scenery	Other
count	140	160	127	189

Table 4: Test dataset summary

However, fixed this problem doesn't help us to reduce the loss when training the model. The lost still high, so we try to find another way to improve it.

5 Train our model with only one class

We have four classes includes being, heritage, scenery and other. We want to give a try to test and see how our model work when we train the model with just one class and ask whether this image belongs to this class or not.

5.1 Train only being class

With only being class, our dataset include:

- 140 images - being class
- 476 images - ”all other” class

5.1.1 Training progress

Figure 6: Being - training progress

5.1.2 Training result

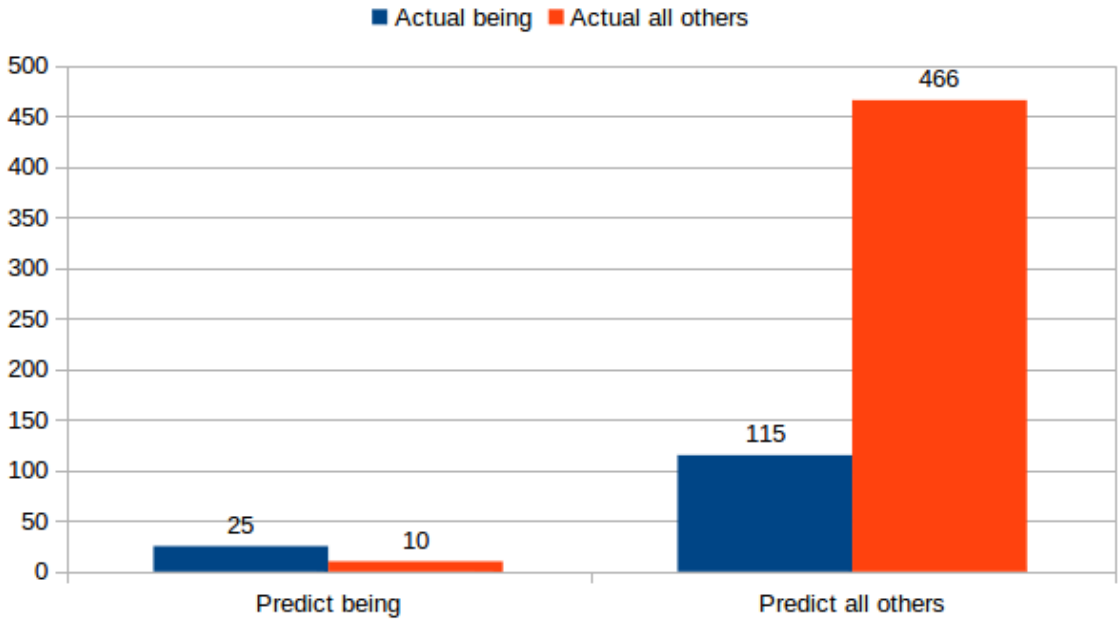

Figure 7: Being - training result

5.2 Train only heritage class

With only heritage class, our dataset include:

- 160 images - heritage class
- 456 images - ”all other” class

5.2.1 Training progress

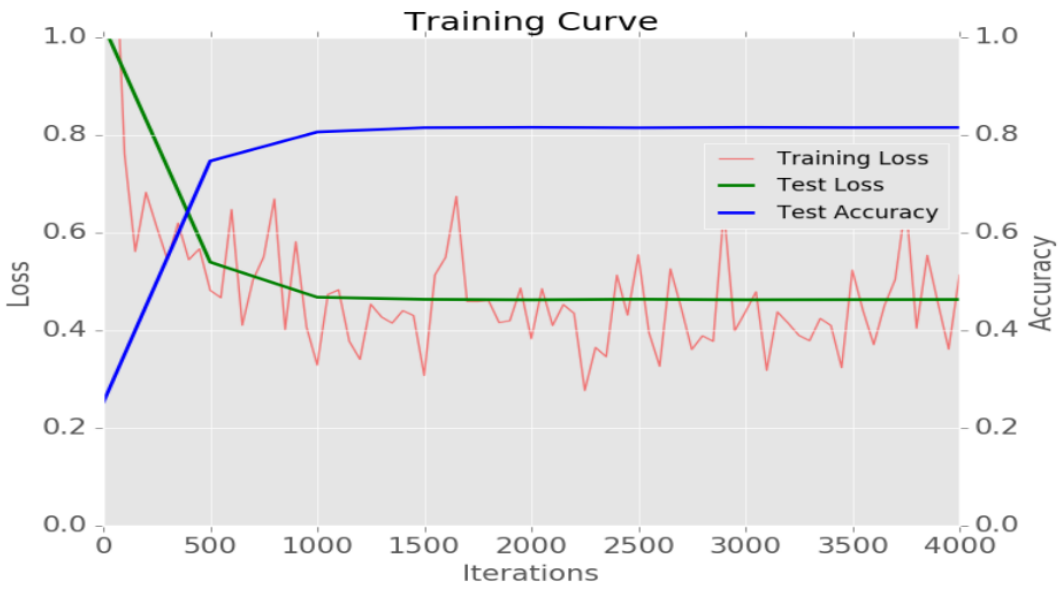

Figure 8: Heritage - training progress

5.2.2 Training result

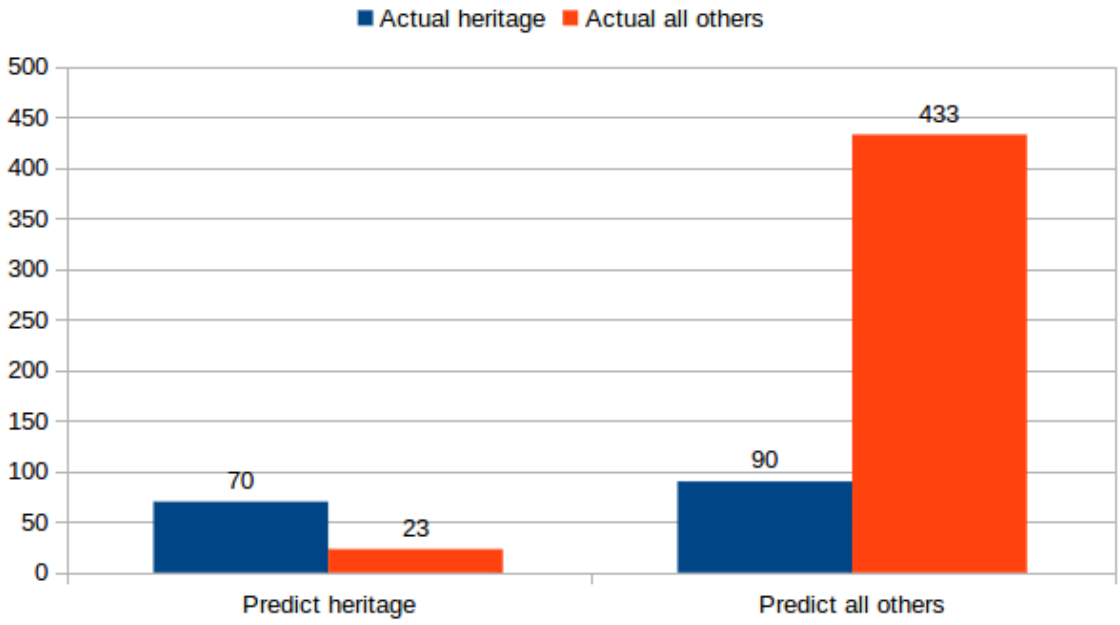

Figure 9: Heritage - training result

5.3 Train only scenery class

With only scenery class, our dataset include:

- 127 images - scenery class
- 489 images - ”all other” class

5.3.1 Training progress

Figure 10: Scenery - training progress

5.3.2 Training result

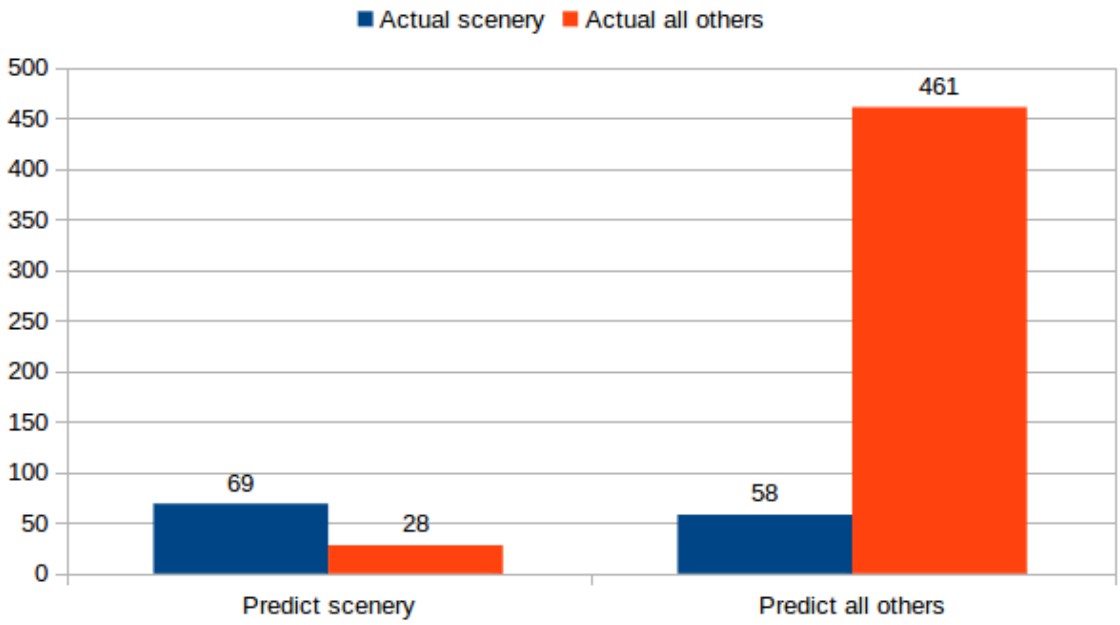

Figure 11: Scenery - training result

5.4 Train only other class

With only other class, our dataset include:

- 189 images - other class
- 427 images - ”all other” class

5.4.1 Training progress

Figure 12: Other - training progress

5.4.2 Training result

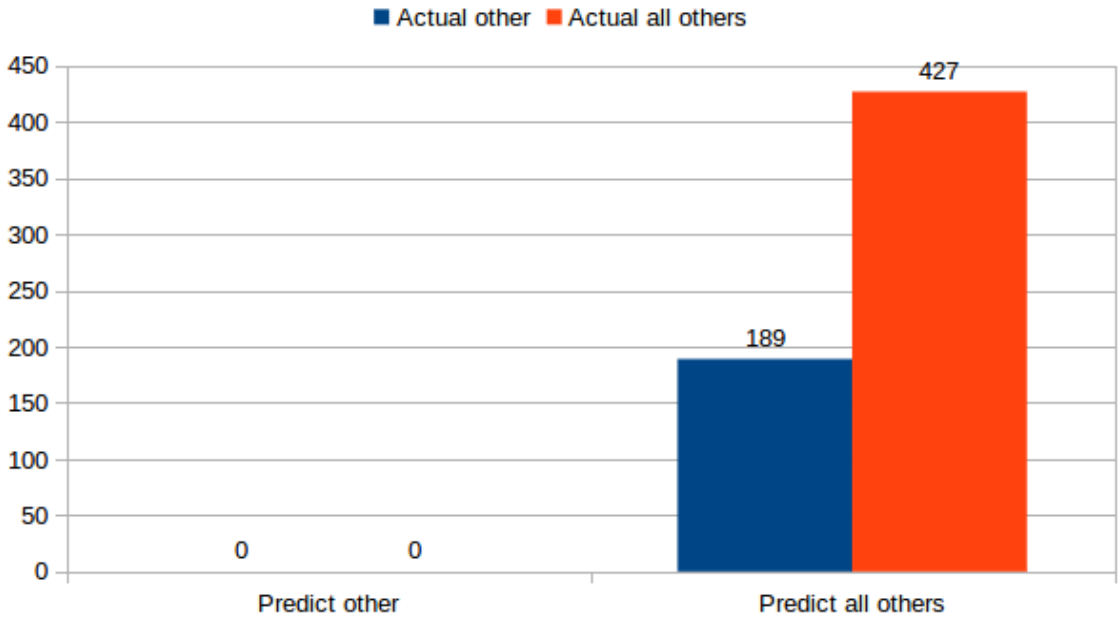

Figure 13: Other - training result

6 Omit other class

Because the other class, actually don't have any specialty. It just a place to keep all images, which don't belong to any other classes. So, It'll be a mistake if we continue to force our neural network to recognize it.

6.1 Train only being class

With only being class, our dataset include:

- 140 images - being class
- 287 images - "all other" class

6.1.1 Training progress

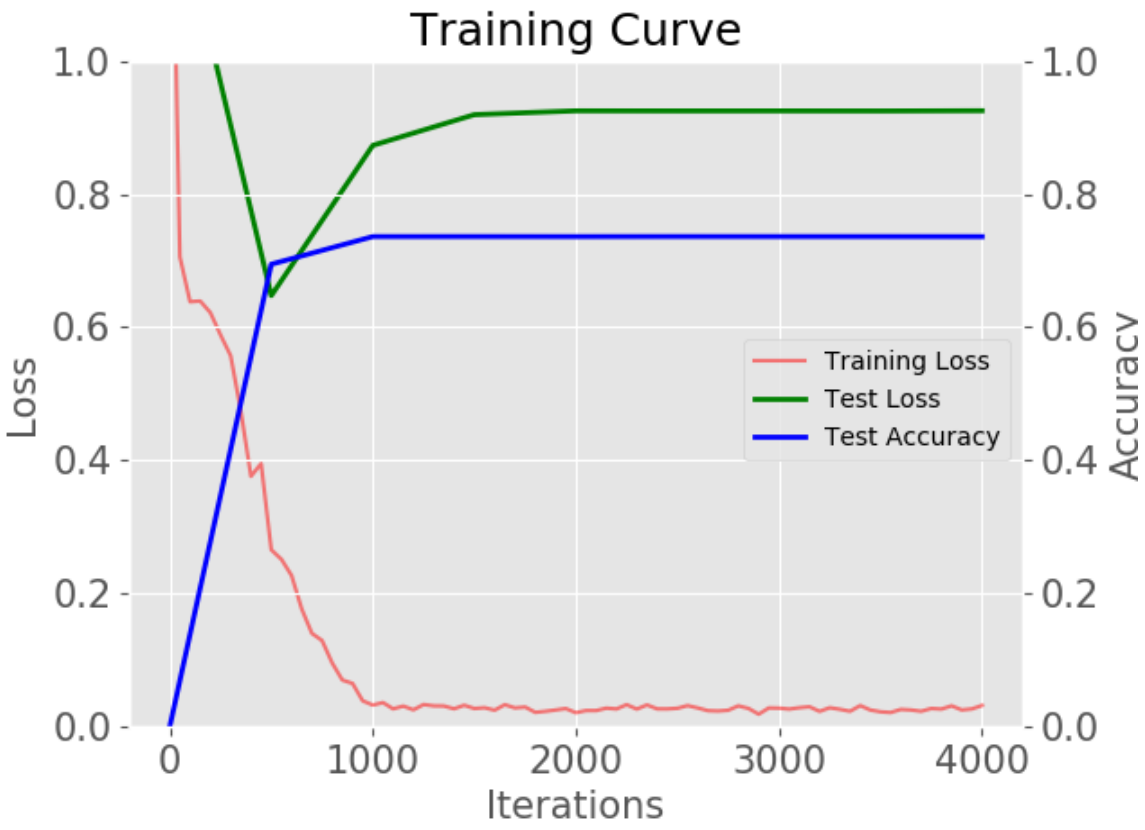

Figure 14: Being - training progress

6.1.2 Training result

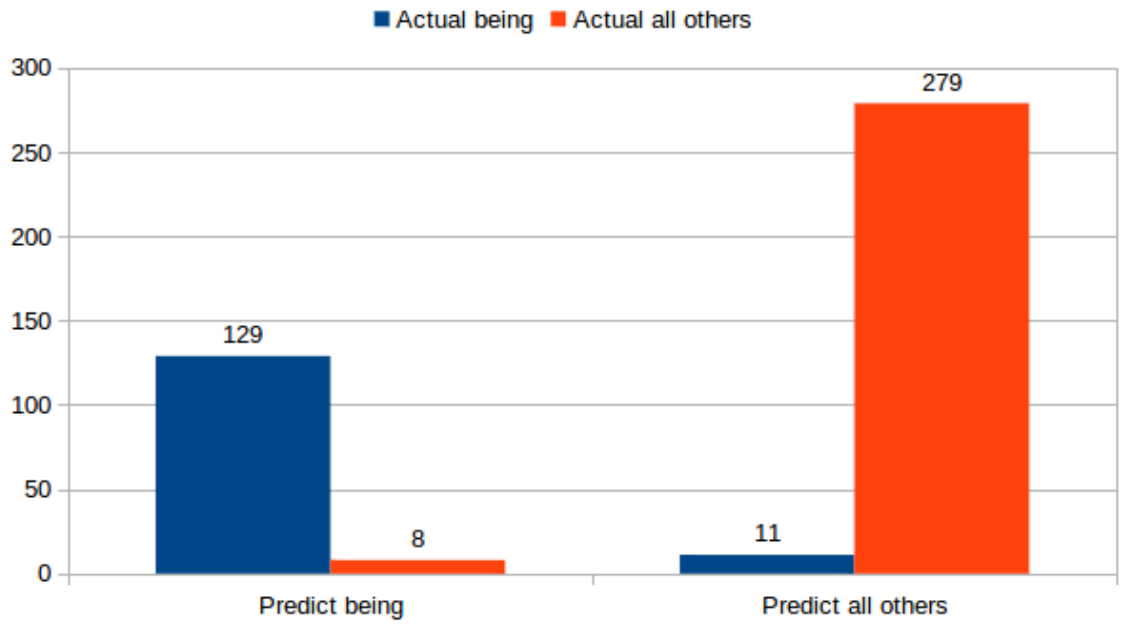

Figure 15: Being - training result

This result seem to be good. But we're using the dataset, which used to train to test the model. So, it doesn't seem to the correct way to test. Therefore, we split our dataset into three small one. First (80%) for training, second (10%) for validate and lately, (10%) for test.

7 Test result

After split our dataset, we have 43 images left for test.

7.0.1 Training progress

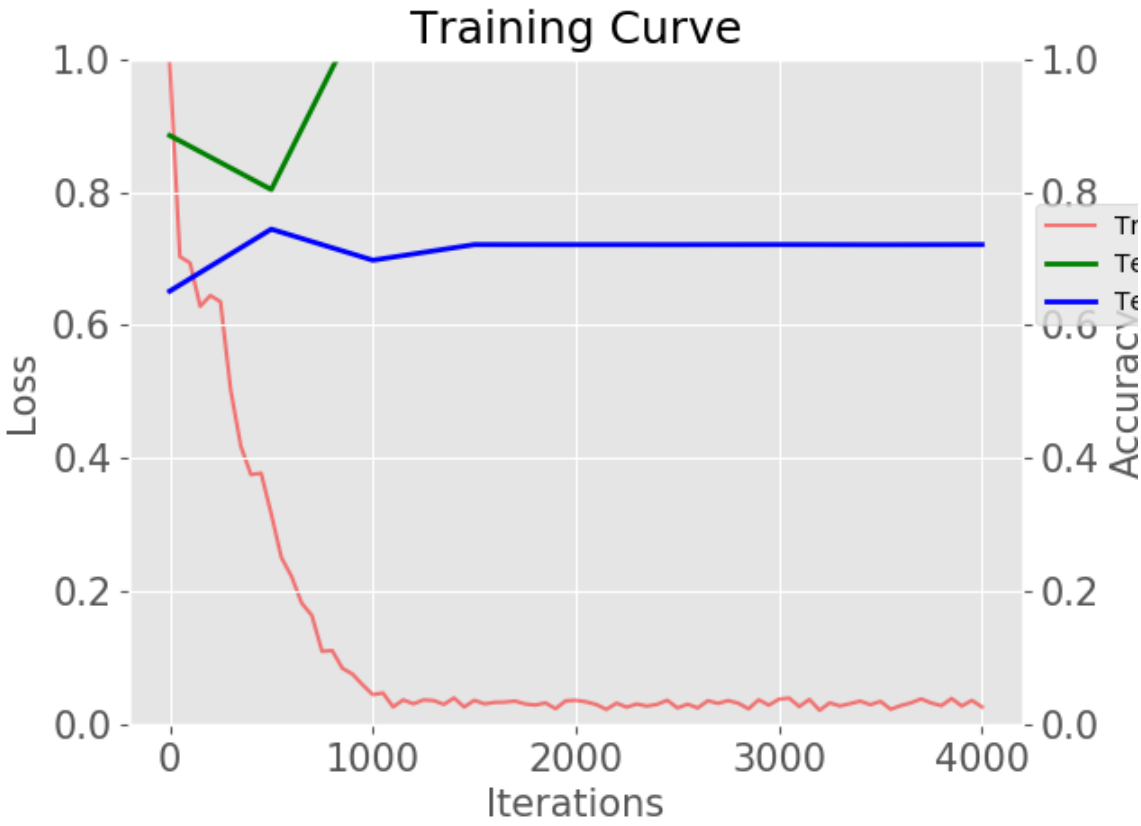

Figure 16: Being - training progress

7.0.2 Training result

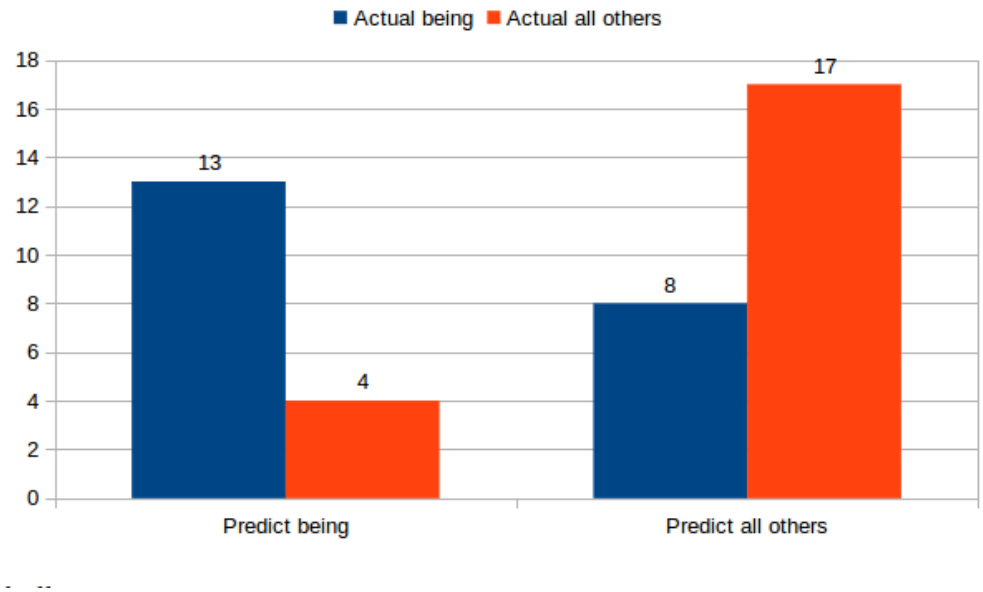

Figure 17: Being - training result

It's seem to we dont have enough images for both train & test purpose.

8 Improve our dataset

Because of lacking images so, we increase the size of our dataset by look back the original unlabeled images and get more images from them. We filter all images has too different size and then, we have 55477 images left. After that, we look through all images and pick all images, which we sure it belong to the class that we are looking for. Finally, we have 4245 images include:

	Being	Heritage	Scenery
count	1471	1832	942

Table 5: Test dataset summary

8.1 Training result

We train our new dataset with 3 classes

8.1.1 Training progress

Figure 18: Train process

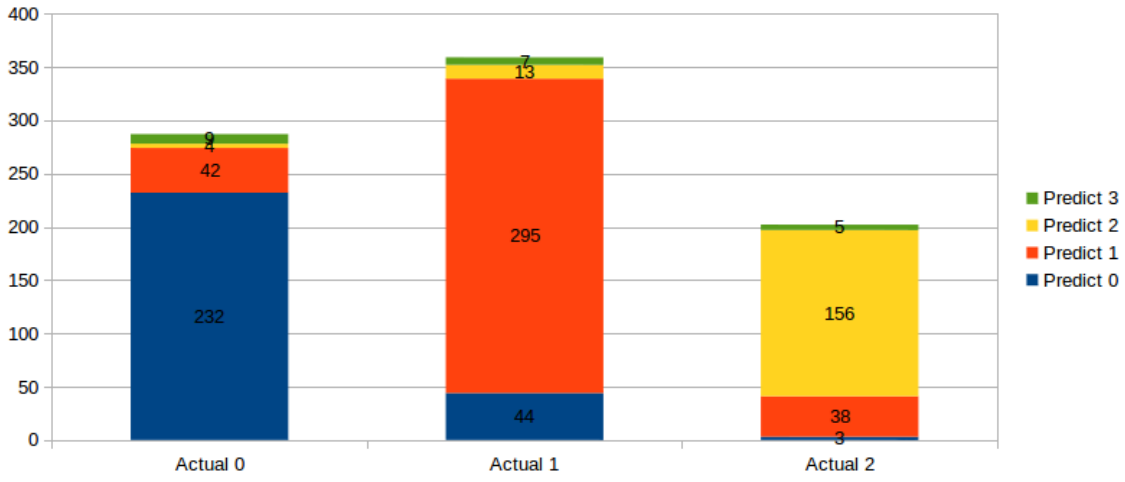

Figure 19: Test result

8.2 Reduce Gamma from 0.1 to 0.01

8.2.1 Training progress

Figure 20: Train process

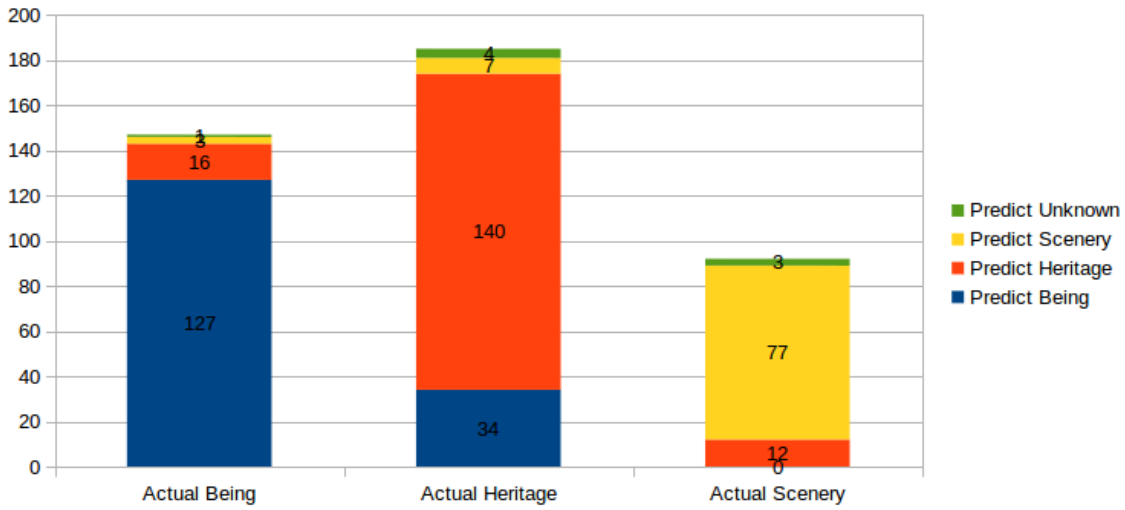

Figure 21: Test result

With 70% train, 10% validate and 20% test

8.2.2 Training progress

Figure 22: Train process

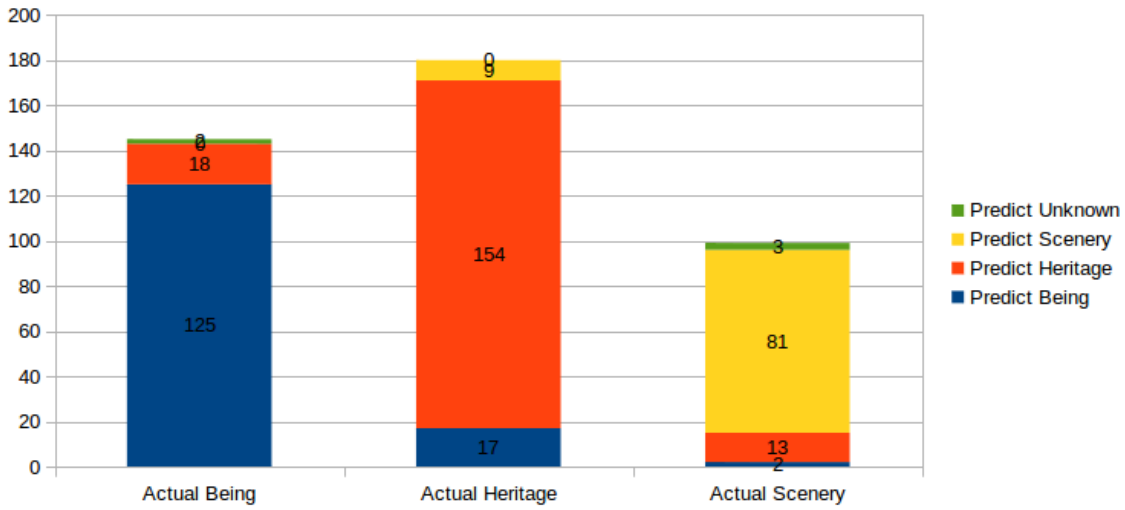

Figure 23: Test result

8.3 Train with dataset come from unsupervised

We using unsupervised to classify our images into 3 class. After that, we enter each class folder, remove all images, which not belong to current class.

8.3.1 Training progress

Figure 24: Train process

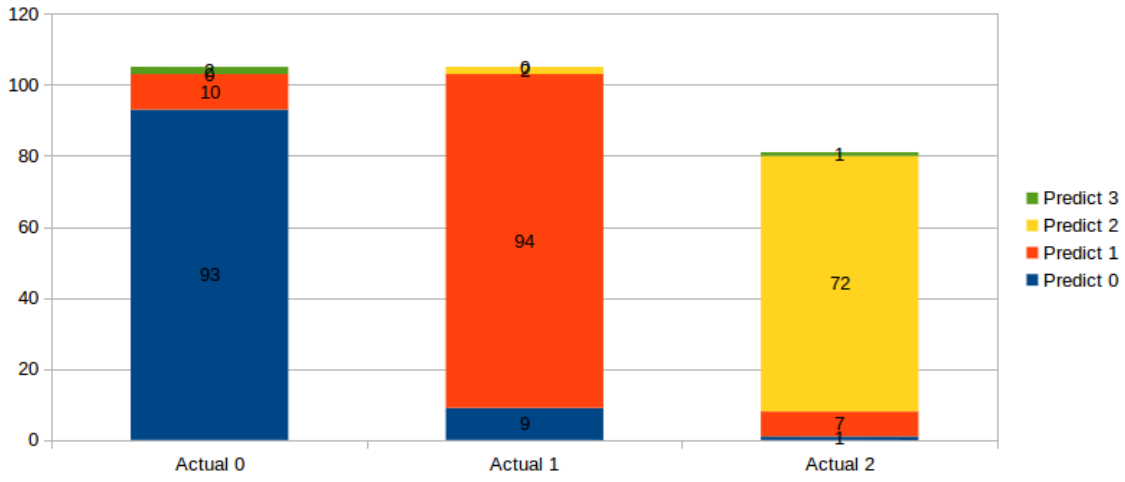

Figure 25: Test result

With 70% train, 10% validate and 20% test

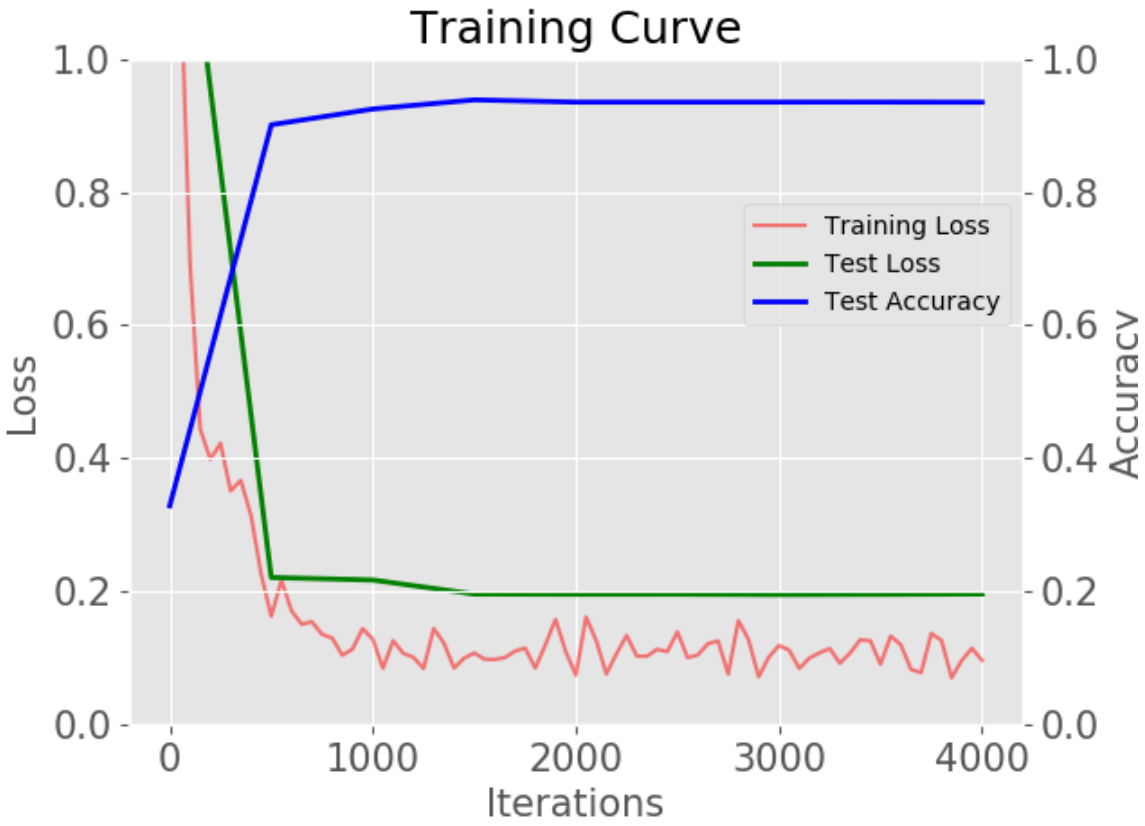

Figure 26: Train process

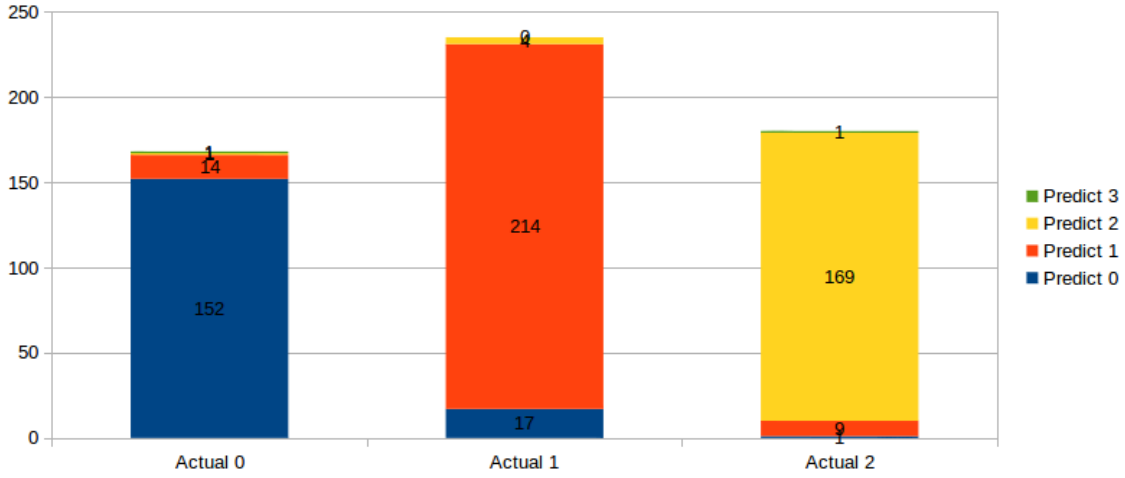

Figure 27: Test result

Without suppress incorrect images

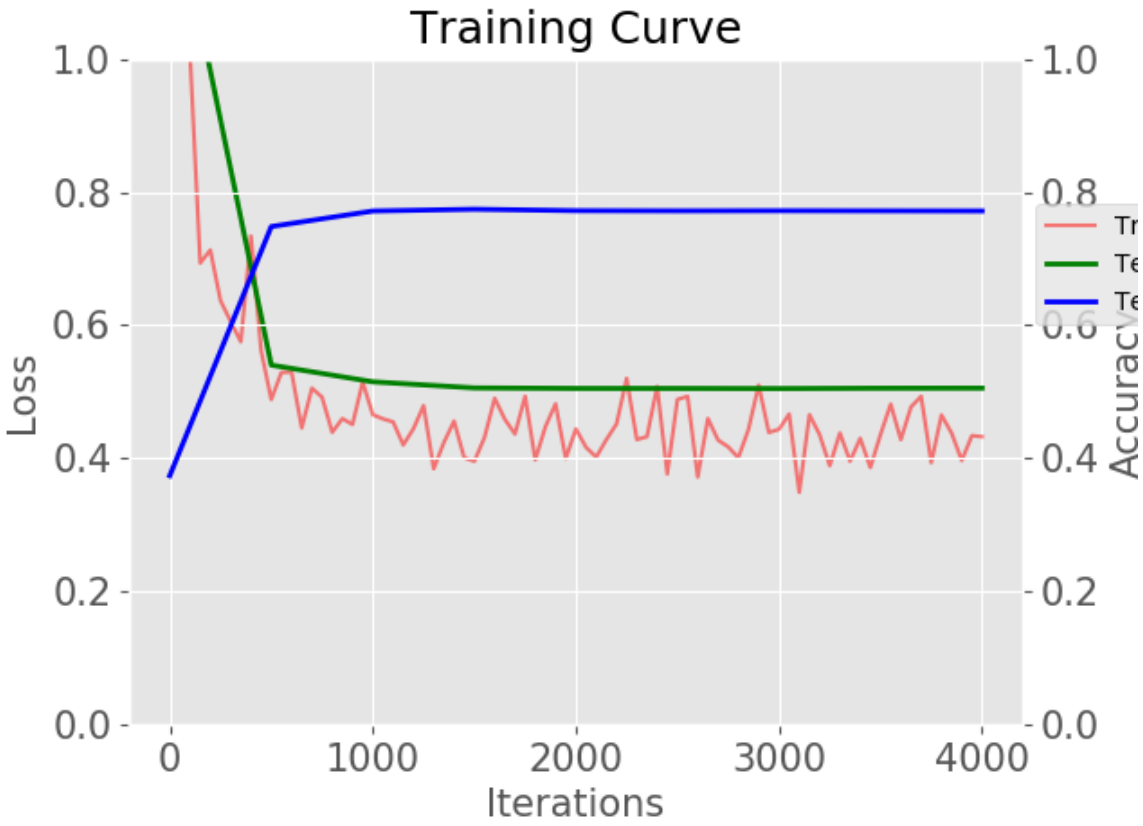

Figure 28: Train process

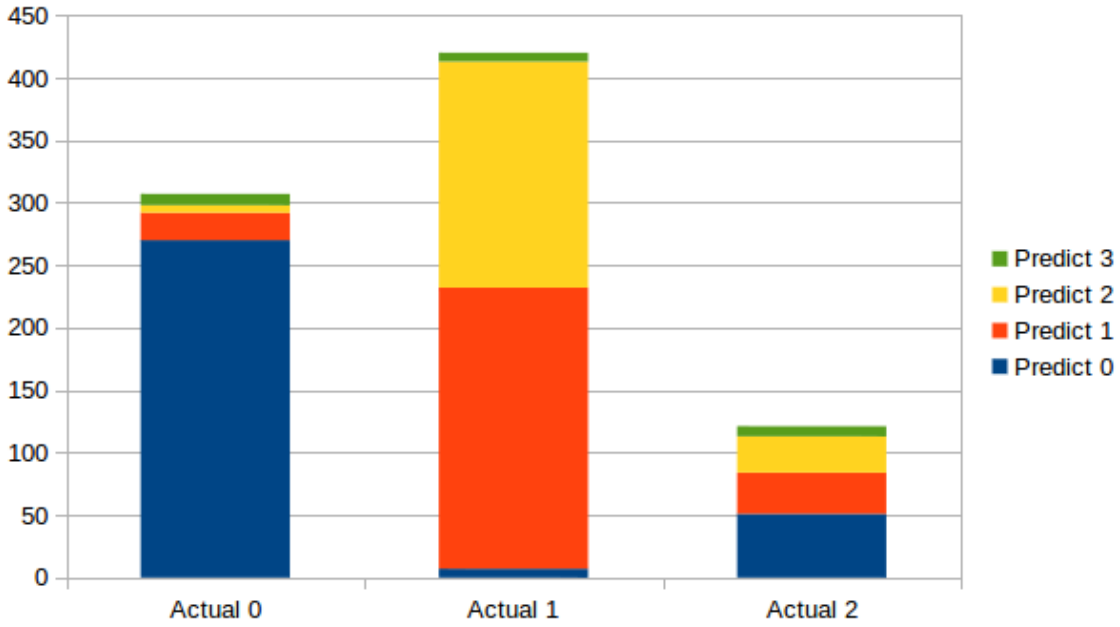

Figure 29: Test result

9 Caffe net layer description

9.1 Data layer

This layer read images from training dataset, validating dataset as input of CNN

- transform_param
  - mirror: when the sample is picked from the dataset, it'll be randomly flipped
    - activated value: true
    - possible value: true | false
  - crop_size: crop all images to specific resolution
    - activated value: unset
  - mean_file: Location of mean file, which we computed before
- data_param
  - source: Location of data source
  - batch_size: Number of sample images will be pick at each iterator to compute the (stochastic) gradient of the parameters in SGD
    - activated value: 256 (train) & 50 (validate)
    - note: depend on GPU
  - backend: Type of database (LevelDB or LMDB)
    - activated value: LMDB
    - possible value: LMDB | LevelDB

9.2 Convolution layer

The convolutional layer is the core building block of a CNN. The layer's parameters consist of a set of learnable filters (or kernels), which have a small receptive field, but extend through the full depth of the input volume. During the forward pass, each filter is convolved across the width and height of the input volume, computing the dot product between the entries of the filter and the input and producing a 2-dimensional activation map of that filter. As a result, the network learns filters that activate when it detects some specific type of feature at some spatial position in the input.

Stacking the activation maps for all filters along the depth dimension forms the full output volume of the convolution layer. Every entry in the output volume can thus also be interpreted as an output of a neuron that looks at a small region in the input and shares parameters with neurons in the same activation map.

- param (for the weight)
  - lr_mult: this value indicates what to multiply the learning rate (from slover) by for a particular layer
    - activated value: 1
    - note: *This is useful if we want to update some layers with a smaller learning rate (e.g. when finetuning some layers while training others from scratch) or if we do not want to update the weights for one layer (perhaps we keep all the conv layers the same and just retrain fully connected layers).*
  - decay_mult: this value indicates what to multiply the weight decay (from slover) by for a particular layer
    - activated value: 1
- param (for the bias)
  - lr_mult:
    - activated value: 2
  - decay_mult:
    - activated value: 0
- convolution_param
  - num_output: number of learnable filters
    - value [96, 256, 384, 384, 256]
  - kernel_size: size (resolution) of learnable filter
    - activated value: [11, 5, 3, 3, 3]
  - stride: controls how the filter convolves around the input volume
    - activated value: [4, 1, 1, 1, 1]
  - pad: pads the input volume with zeros around the border
    - activated value: [0, 2, 1, 1, 1]

  - group: [default 1]. If g > 1, we restrict the connectivity of each filter to a subset of the input. Specifically, the input and output channels are separated into g groups, and the iith output group channels will be only connected to the iith input group channels.
    * activated value: [1, 2, 1, 2, 2]
  - weight_filler: *initialize the filters*
    * type: algorithm type
      · activated value: gaussian
      · possible value: [gaussian, constant, positive_unitball, uniform, xavier, msra, bilinear]
    * other options for each type above
      · std: 0.01
  - bias_filler
    * type: algorithm type
      · activated value: constant
      · possible value: [gaussian, constant, positive_unitball, uniform, xavier, msra, bilinear]
    * other options for each type above
      · activated value: 1

9.3 ReLU (Rectified Linear Units) layer

The ReLU layer applies the function f(x) = max(0, x) to all of the values in the input volume. In basic terms, this layer just changes all the negative activations to 0.This layer increases the nonlinear properties of the model and the overall network without affecting the receptive fields of the convolution layer.

9.4 Dropout

During training only, sets a random portion of x to 0, adjusting the rest of the vector magnitude accordingly. Because a fully connected layer occupies most of the parameters, it is prone to overfitting. One method to reduce overfitting is dropout. At each training stage, individual nodes are either "dropped out" of the net with probability $1 - p$ or kept with probability p, so that a reduced network is left; incoming and outgoing edges to a dropped-out node are also removed. Only the reduced network is trained on the data in that stage. The removed nodes are then reinserted into the network with their original weights.

- dropout_param
  - dropout_ratio: the probability that any given unit is dropped.
    * activated value: 0.5

9.5 Pooling layer

Another important concept of CNNs is pooling, which is a form of non-linear down-sampling. There are several non-linear functions to implement pooling among which max pooling is the most common. It partitions the input image into a set of non-overlapping rectangles and, for each such sub-region, outputs the maximum. The intuition is that the exact location of a feature is less important than its rough location relative to other features. The pooling layer serves to progressively reduce the spatial size of the representation, to reduce the number of parameters and amount of computation in the network, and hence to also control overfitting. It is common to periodically insert a pooling layer between successive convolutional layers in a CNN architecture. The pooling operation provides another form of translation invariance.

The pooling layer operates independently on every depth slice of the input and resizes it spatially. The most common form is a pooling layer with filters of size 2x2 applied with a stride of 2 downsamples at every depth slice in the input by 2 along both width and height, discarding 75% of the activations. In this case, every max operation is over 4 numbers. The depth dimension remains unchanged.

In addition to max pooling, the pooling units can use other functions, such as average pooling or L2-norm pooling. Average pooling was often used historically but has recently fallen out of favor compared to max pooling, which works better in practice

- pooling_param
  - pool: type of pooling
    - activated value: MAX
    - possible value: [MAX, AVE, STOCHASTIC]
  - kernel_size: size (resolution) of pooling filter
    - activated value: 3
  - stride: controls how the filter slide around the input volume
    - activated value: 2
  - pad: pads the input volume with zeros around the border
    - activated value: 0

9.6 LRN (Local Response Normalization) layer

The local response normalization layer performs a kind of "lateral inhibition" by normalizing over local input regions. In ACROSS_CHANNELS mode, the local regions extend across nearby channels, but have no spatial extent (i.e., they have shape local_size x 1 x 1). In WITHIN_CHANNEL mode, the local regions extend spatially, but are in separate channels (i.e., they have shape 1 x local_size x local_size). Each input value is divided by $(1 + (\alpha/n)\sum_i x_i^2)^\beta$, where n is the size of each local region, and the sum is taken over the region centered at that value (zero padding is added where necessary).

- lrn_param

  - local_size: the number of channels to sum over (for cross channel LRN) or the side length of the square region to sum over (for within channel LRN)
    - activated value: 5
  - alpha: the scaling parameter
    - activated value: 0.0001
  - beta: the exponent
    - activated value: 0.75
  - norm_region: whether to sum over adjacent channels (ACROSS_CHANNELS) or nearby spatial locations (WITHIN_CHANNEL)
    - activated value: ACROSS_CHANNELS

9.7 InnerProduct layer

This layer basically takes an input volume and outputs an N dimensional vector where N is the number of classes that the program has to choose from. Neurons in a fully connected layer have connections to all activations in the previous layer, as seen in regular neural networks. Their activations can hence be computed with a matrix multiplication followed by a bias offset.

- param (for the weight)
  - lr_mult: this value indicates what to multiply the learning rate (from slover) by for a particular layer
    - activated value: 1
  - decay_mult: this value indicates what to multiply the weight decay (from slover) by for a particular layer
    - activated value: 1
- param (for the bias)
  - lr_mult:
    - activated value: 2
  - decay_mult:
    - activated value: 0
- inner_product_param
  - num_output: The number of outputs for the layer
    - value: [4096, 4096, 3]
  - weight_filler: *initialize the filters*
    - type: algorithm type
      - activated value: gaussian
      - possible value: [gaussian, constant, positive_unitball, uniform, xavier, msra, bilinear]

        * other options for each type above
            · std: 0.01
    - bias_filler
        * type: algorithm type
            · activated value: constant
            · possible value: [gaussian, constant, positive_unitball, uniform, xavier, msra, bilinear]
        * other options for each type above
            · activated value: 1

9.8 Accuracy layer

Accuracy scores the output as the accuracy of output with respect to target – it is not actually a loss and has no backward step.

9.9 SoftmaxWithLoss layer

The softmax loss layer computes the multinomial logistic loss of the softmax of its inputs. It's conceptually identical to a softmax layer followed by a multinomial logistic loss layer, but provides a more numerically stable gradient.

References

[1] A. F. Junyuan Xie, Ross Girshick, "Unsupervised deep embedding for clustering analysis," *ICML'16 Proceedings of the 33rd International Conference on International Conference on Machine Learning*, vol. 48, 2016.