Lab 3: CNN on Cifar-10

Lab Objective:

In this lab, you will be asked to build popular network architecture (*Network In Network*, NIN) [1], and train it on Cifar-10 dataset. Moreover, you need to use data augmentation and Dropout [2] during training.

Turn in:

1. Experiment Report (3/28(二))
2. Demo date (3/28(二))

Requirements:

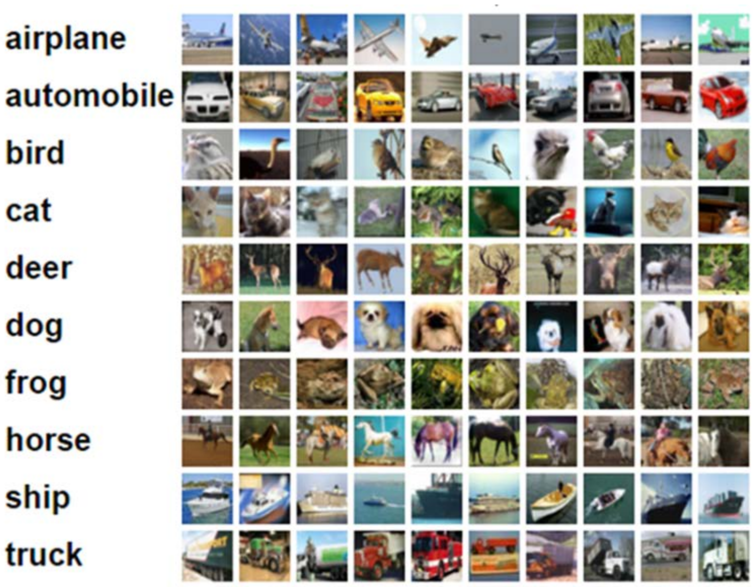
- Implement "**Network In Network**" (NIN) [1] convolutional architecture
- Implement data augmentation: translation and horizontal flipping
- Use "**Dropout**" [2] in NIN
- Train NIN+Dropout with/without data augmentation

Environment:

- Cifar-10 dataset

The CIFAR-10 dataset consists of 60000 32×32 color images (RGB) in **10** classes, with 6000 images per class. There are 50000 training images and 10000 test images.

Download: https://www.cs.toronto.edu/~kriz/cifar.html



Sample Code

There are many cifar-10 sample codes:

git clone https://github.com/tensorflow/models.git

Higher-level tensorflow API tflearn:

pip install git+https://github.com/tflearn/tflearn.git

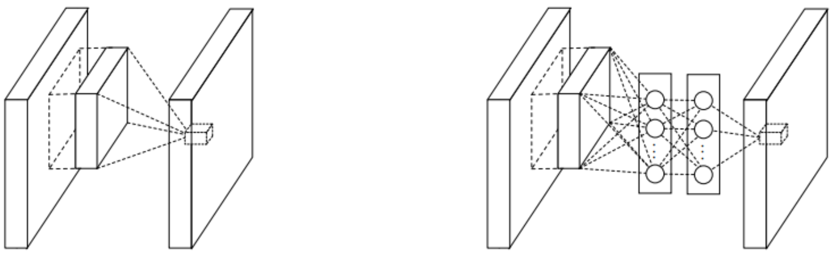
Or you can install other higher-level API, such as Keras.

Lab Description:

- Network In Network (NIN)
  - Enhance model discriminability for local patches within the receptive field

  Traditional CNN: 3x3 conv + ReLU

  NIN: 3x3 conv + ReLU + 1x1 conv + ReLU + 1x1 conv +ReLU



(a) Linear convolution layer

(b) Mlpconv layer

$$f_{i,j,k} = \max(w_k^T x_{i,j}, 0).$$

$$\begin{aligned} f^1_{i,j,k_1} &= \max({w^1_{k_1}}^T x_{i,j} + b_{k_1}, 0). \\ &\vdots \\ f^n_{i,j,k_n} &= \max({w^n_{k_n}}^T f^{n-1}_{i,j} + b_{k_n}, 0). \end{aligned}$$

- Full NIN architecture used in Cifar-10

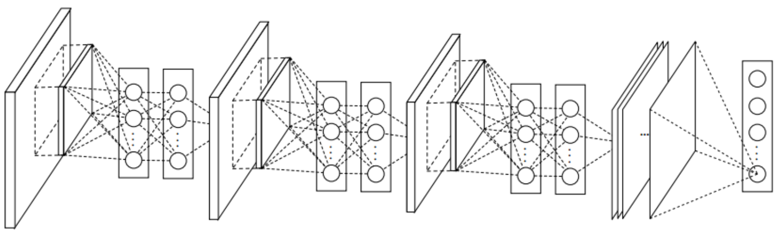
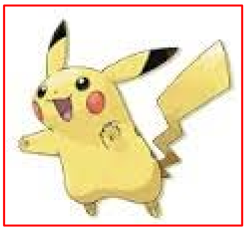


Figure 2: The overall structure of Network In Network. In this paper the NINs include the stacking of three mlpconv layers and one global average pooling layer.

- Architecture Details:

Conv1	filter size = 5x5, # of filter =192, pad = 2, stride = 1	Act.=ReLU
mlp 1	filter size = 1x1, # of filter =160, pad = 0, stride = 1	Act.=ReLU
mlp 2	filter size = 1x1, # of filter =96, pad = 0, stride = 1	Act.=ReLU
Pool 1	3x3 max pooling, stride = 2	
	Dropout 0.5	
Conv2	filter size = 5x5, # of filter =192, pad = 2, stride = 1	Act.=ReLU
mlp 2-1	filter size = 1x1, # of filter =192, pad = 0, stride = 1	Act.=ReLU
mlp 2-2	filter size = 1x1, # of filter =192, pad = 0, stride = 1	Act.=ReLU
Pool 2	3x3 max pooling, stride = 2	
	Dropout 0.5	
Conv3	filter size = 3x3, # of filter =192, pad = 1, stride = 1	Act.=ReLU
mlp 3-1	filter size = 1x1, # of filter =192, pad = 0, stride = 1	Act.=ReLU
mlp 3-2	filter size = 1x1, # of filter =10, pad = 0, stride = 1	Act.=ReLU
Global Pool	8x8 average pooling, stride =1	
	Softmax	

- Data augmentation: Translation and Horizontal flipping:



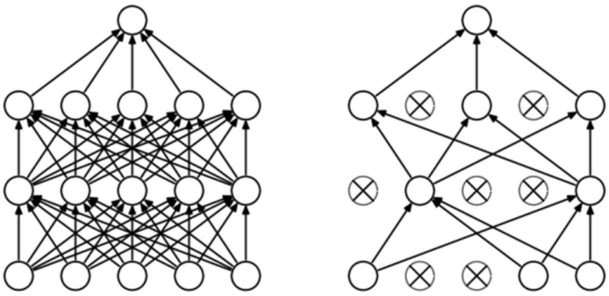
Original

Translation

Horizontal flipping

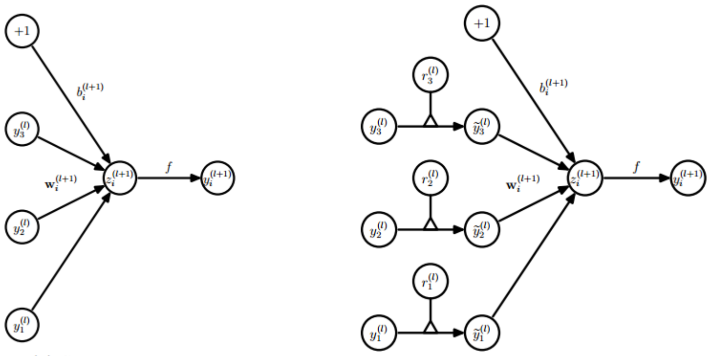
- Dropout

Dropout is a technique for addressing this problem. The key idea is to randomly drop units (along with their connections) from the neural network during training. This prevents units from **co-adapting** too much. During training, dropout samples from an exponential number of different "thinned" networks. At test time, it is easy to approximate the effect of averaging the predictions of all these thinned networks by simply using a single unthinned network that has **smaller weights**. This significantly reduces overfitting and gives major improvements over other regularization methods.



(a) Standard Neural Net (b) After applying dropout.

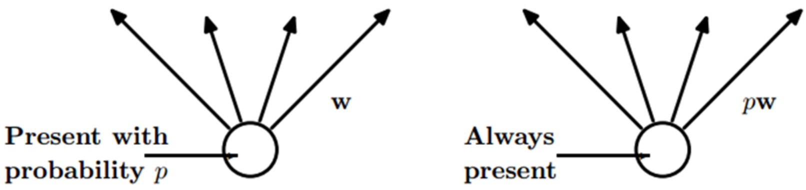
■ Network with Dropout



(a) Standard network (b) Dropout network

Figure 3: Comparison of the basic operations of a standard and dropout network.

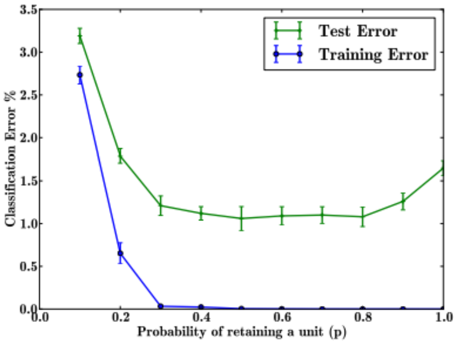
■ Dropout at testing



(a) At training time (b) At test time

■ The effect of Dropout rate

If the architecture is held constant, having a small p means very few units will turn on during training. It can be seen that this has led to *underfitting* since the training error is also high. We see that as p increases, the error goes down. It becomes flat when **$0.4 \le p \le 0.8$** and then increases as p becomes close to 1.



(a) Keeping n fixed.

Implementation Details:

- Training Hyperparameters:
  - Method: SGD with momentum
  - Mini-batch size: 128 (391 iterations for each epoch)
  - Total epochs: 164, momentum 0.9 (if you use momentum SGD)
  - Initial learning rate: 0.1, divide by 10 at 81, 122 epoch
  - Loss function: cross-entropy

- Data augmentation parameters:
  - Translation: Pad **4** zeros in each side and random cropping back to 32x32 size
  - Horizontal flipping: With probability **0.5**

Methodology:

- 91.10% accuracy with data augmentation in my implementation
- 89.23% accuracy without data augmentation

Extra Bonus:

- All convolutions NIN (remove pooling layers)
- Reproduce the experiment of Dropout rate in [2].

References:

[1] Lin, M., Chen, Q., & Yan, S. (2013). Network in network. *arXiv preprint arXiv:1312.4400*.

[2] Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, *15*(1), 1929-1958.

[3] https://www.tensorflow.org/tutorials/deep_cnn/

Report Spec: [black: Demo, Gray: No Demo]

1. Introduction (15%)
2. Experiment setup (15%)
   - The detail of your model
   - Report all your training hyper-parameters
3. Result
   - The comparison between with and without data augmentation
     - Final Test error (10%, 15%)
     - Training loss curve (you need to record training loss every epoch) (10%, 15%)
     - Test error curve (you need to record test error every epoch) (10%, 15%)
4. Discussion (20%, 25%)

Demo (20%) [抽 20 人 DEMO]

-------實驗結果標準 (with data augmentation)----

Accuracy: (92.0~90.0)% = 100%

Accuracy: (90.0~87.0)% = 90%

Accuracy: (87.0~84.0)% = 80%

Accuracy: below 84.0% = 70%

Accuracy: 10% = 0%

評分標準: 40%*實驗結果 + 60%*(報告+DEMO)

Data loading

https://www.cs.toronto.edu/~kriz/cifar.html

Python / Matlab versions

I will describe the layout of the Python version of the da

The archive contains the files `data_batch_1`, `data_batch_`
a file and return a dictionary:

```
def unpickle(file):
    import cPickle
    fo = open(file, 'rb')
    dict = cPickle.load(fo)
    fo.close()
    return dict
```

Loaded in this way, each of the batch files contains a di

- **data** -- a 10000x3072 numpy array of `uint8`s. Each
  image is stored in row-major order, so that the first
- **labels** -- a list of 10000 numbers in the range 0-9.

```
def unpickle(file):
  import cPickle
  fo = open(file, 'rb')
  dict = cPickle.load(fo)
  fo.close()
  if 'data' in dict:
    dict['data'] = dict['data'].reshape((-1, 3, 32, 32)).swapaxes(1, 3).swapaxes(1, 2).reshape(-1, 32*32*3)

  return dict

def load_data_one(f):
  batch = unpickle(f)
  data = batch['data']
  labels = batch['labels']
  print("Loading %s: %d" % (f, len(data)))
  return data, labels

def load_data(files, data_dir, label_count):
  data, labels = load_data_one(data_dir + '/' + files[0])
  for f in files[1:]:
    data_n, labels_n = load_data_one(data_dir + '/' + f)
    data = np.append(data, data_n, axis=0)
    labels = np.append(labels, labels_n, axis=0)
  labels = np.array([ [ float(i == label) for i in xrange(label_count) ] for label in labels ])
  return data, labels
```

```
###
data_dir = './cifar-10-batches-py'
image_size = 32
image_dim = image_size * image_size * 3
meta = unpickle(data_dir + '/batches.meta')
label_names = meta['label_names']
label_count = len(label_names)

train_files = [ 'data_batch_%d' % d for d in xrange(1, 6) ]
train_data, train_labels = load_data(train_files, data_dir, label_count)
#print("Train:", np.shape(train_data), np.shape(train_labels))
#print("Test:", np.shape(test_data), np.shape(test_labels))
test_data, test_labels = load_data([ 'test_batch' ], data_dir, label_count)
pi = np.random.permutation(len(train_data))
train_data, train_labels = train_data[pi], train_labels[pi]
train_data = train_data.reshape((-1, 32, 32, 3))
```

Save session

https://www.tensorflow.org/programmers_guide/variables

```
saver = tf.train.Saver()
save_path = saver.save(session, 'NIN_%d.ckpt' % epoch)
```

data augmentation

https://github.com/JiaRenChang/DLcourse_NCTU/blob/master/data_aug.py

Usage:

```
Import data_aug
cropped_data = data. _random_crop(batch_data)
flipped_data = data. _random_flip_leftright(cropped_data)
```

Tricks for LAB 1 & 2

1. Data preprocessing

- Color normalization

  Normalize each color channel (compute from entire CIFAR10 training set)

  Mean $\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{pmatrix} 125.3 \\ 123.0 \\ 113.9 \end{pmatrix}$

  Variance $\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{pmatrix} 63.0 \\ 62.1 \\ 66.7 \end{pmatrix}$

  This trick provides about 4~5% improvement

  Without data preprocessing:

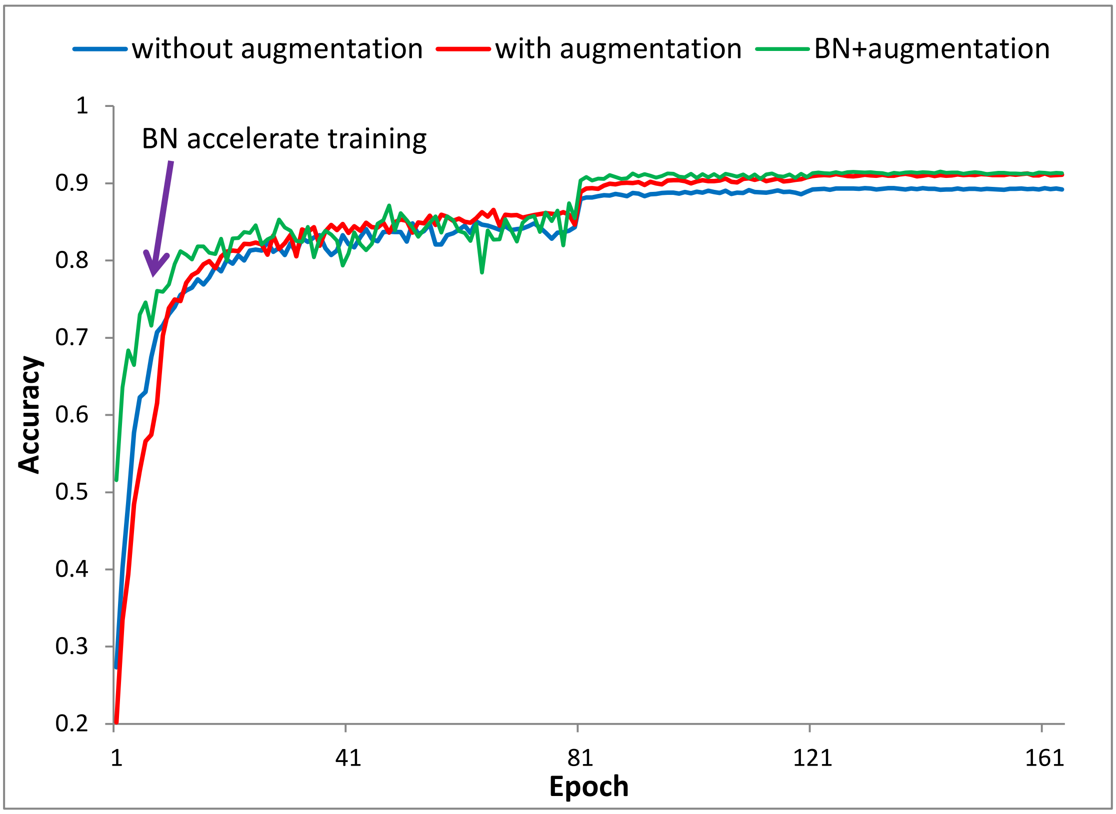
  Without data augmentation: ~84%

  With data augmentation: ~85%

  With data preprocessing:

  Without data augmentation: 89.23%

  With data augmentation: 91.10%



2. Learning rate schedule

If you use tflearn, try this simple way:

learner = tflearn.optimzer(0.1, lr_decay = 0.1,decay_step = 32000,staircase = True)
network_in_network = regression(out_network, optimizer = learner, loss='softmax_categorical_crossentropy')
model = tflearn.DNN(network_in_network)
model.fit(X, Y, n_epoch = 200, shuffle = True, validation_set = (X_test, Y_test), show_metric = True, batch_size = 128, run_id = 'cifar10_net_in_net')

0.1: 1~80 0.01: 81~160 0.001:161~200

Some problem in tflearn.momentum???

3. Weight Decay

Weight decay = 0.0001

http://stackoverflow.com/questions/38882629/how-to-implement-weight-decay-in-tensorflow-as-in-caffe

4. Weight initialization in LAB 1

First conv layer: Random_normal(stddev=0.01)
Others: Random_normal(stddev=0.05)

5. Use Nesterov momentum

tf.train.MomentumOptimizer(learning_rate, 0.9, use_nesterov=True)