Computer Vision
Report 2
Zou Zijun – B4TB1709

Objective

To improve the object recognition more than 45% under the same conditions as used during the lecture and to describe the method and result for the program used

Environment

Python 2.7.13 :: Anaconda custom (64 bit)
OpenCV 2.4.11
Ubuntu 16.04 LTS

Background

CIFAR-10 contains 60000 labeled for 10 classes images 32x32 in size, with train set having 50000 and test set having 10000. Although this dataset is quite small by today's standards, but it still remains a good first step for machine learning algorithms.
Various techniques such as deep convolutional neural networks, cropping, affine transforms, horizontal flips etc to improve the accuracy of the program.

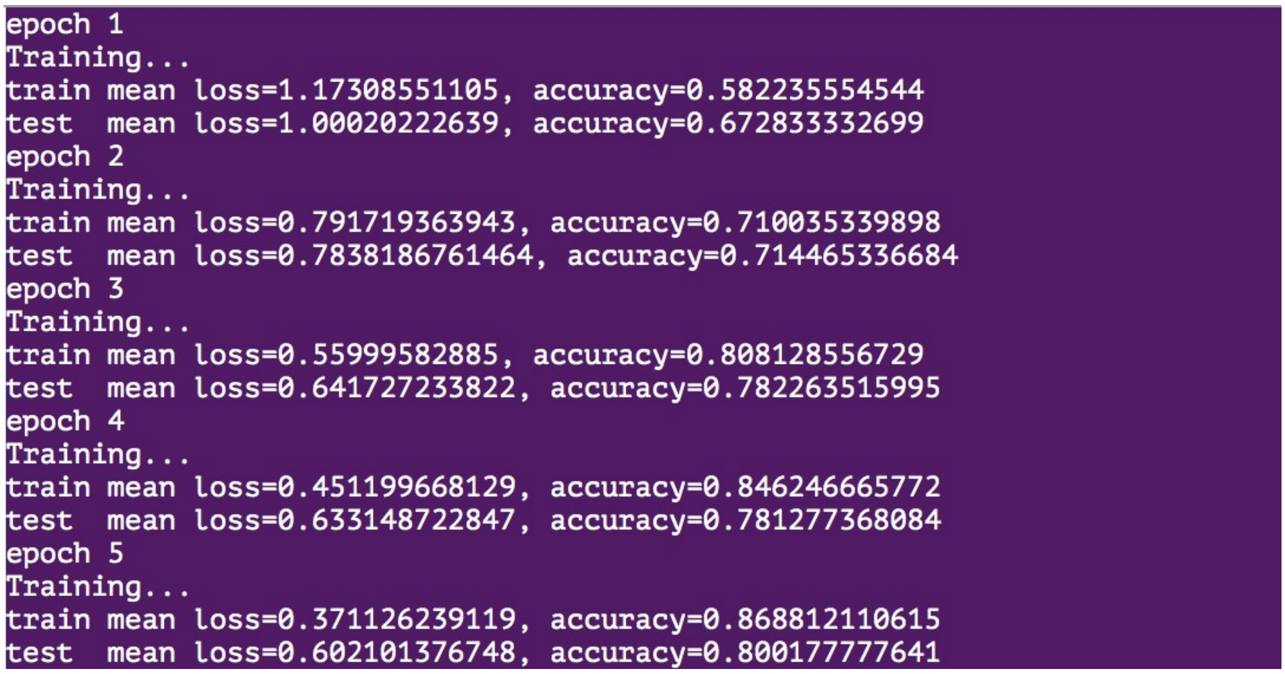
Approach

The dataset of CIFAR-10 is loaded and the data augmentation takes place which applies transformations to the training data thereby adding synthetic (artificial) data points to the data. This helps the model to add variations without using extra data for the same purpose. Cropping is used here to reduce the contribution of the background in the CNN (Convolutional Neural Network's) decision.
Then, CNN model is prepared for the dataset. During this process, each image is broken into overlapping image tiles. Convolutional operation is applied to the input by convolutional layers and the result is passed on to the next layer. Tiling helps CNNs to tolerate translation or rotation or perspective distortion of the input image. This improves generalization and helps to adjust its filter values through **backpropagation**. After this, **max pooling** is used to combine the outputs of filter clusters at one layer into a single filter in the next layer. These steps of repeated convolution and max pooling are iterated three more times to create a deep network which is able to recognize features more effectively.
Optimizers are then introduced to minimize the **loss function** which measures the quality of particular set of parameters based on how well the induced scores agreed with the ground truth labels in the training data.
The neural networks begin to form at this point as training data is used for learning.

Result

The program was run for 5 epoch duration in which the accuracy successively increased with each epoch. As evident from Figure 1, with each epoch, the image recognition got better and the loss started decreasing and accuracy increasing, which is expected to happen as more time(iteration) is given to the program, the better it is able to train and in turn the better it is able to perform.

Since each epoch took approximately 3 hours to complete, it was possible but not entirely feasible to let the program run to increase the accuracy even further. If the computer specification on which this program was run were better, the time taken would be much less. Moreover, the use of GPU (through Chainer CUDA) in computing the neural network also could have been a great help but it was not used in the making of the following results.



```
epoch 1
Training...
train mean loss=1.17308551105, accuracy=0.582235554544
test  mean loss=1.00020222639, accuracy=0.672833332699
epoch 2
Training...
train mean loss=0.791719363943, accuracy=0.710035339898
test  mean loss=0.7838186761464, accuracy=0.714465336684
epoch 3
Training...
train mean loss=0.55999582885, accuracy=0.808128556729
test  mean loss=0.641727233822, accuracy=0.782263515995
epoch 4
Training...
train mean loss=0.451199668129, accuracy=0.846246665772
test  mean loss=0.633148722847, accuracy=0.781277368084
epoch 5
Training...
train mean loss=0.371126239119, accuracy=0.868812110615
test  mean loss=0.602101376748, accuracy=0.800177777641
```

Figure 1 Train and test accuracy and loss

The final test accuracy and test loss at the time of termination of the program at epoch 5 were:

Test Accuracy = 80.02%
Test mean loss = 60.21%

It can be seen from Figure 2 that the accuracy of both train and test increases with each epoch iteration.

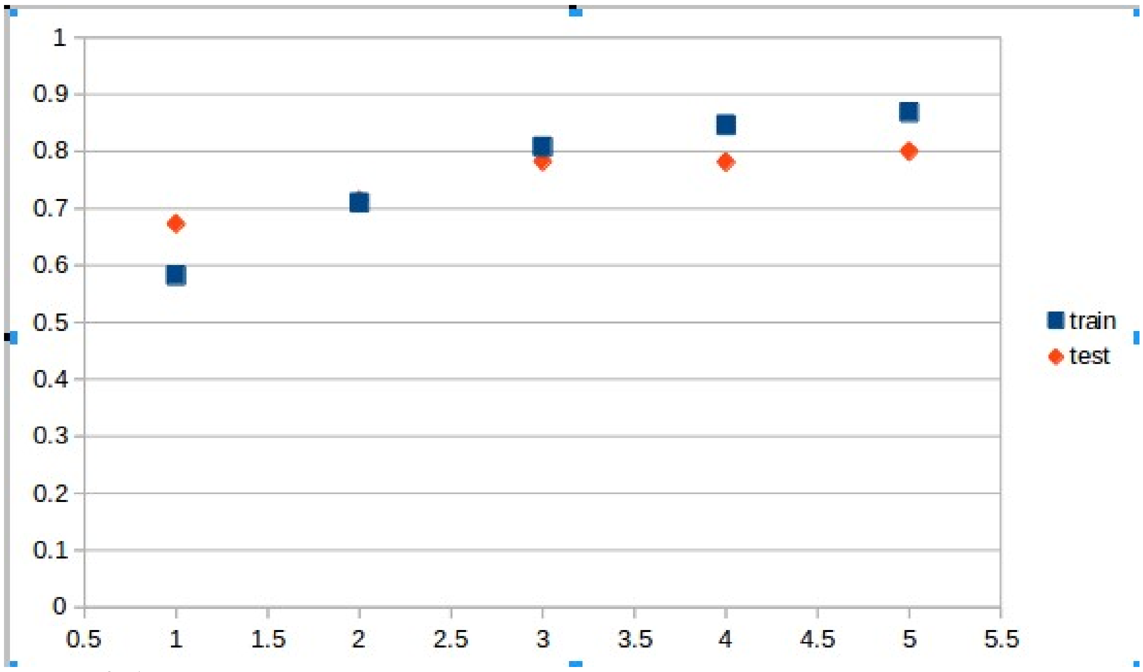


Figure 2 Accuracy curve

Program

Main program used is trainer.py, which imports from other modules like model_cnn and datahandler. Explanation about these is given after trainer.py

Chainer is a framework of neural networks which supports CUDA and multi-GPU capabilities. It supports almost arbitrary architectures and has many predefined functions which can be made use of.

Tqdm 4.15.0 is a python package to be used with iterables. As its overhead is low – 60ns per iteration – it is very useful compared to other similar packages. In addition, tqdm uses smart algorithms to predict the remaining time and to slip unnecessary iteration display. Also, tqdm does not require any dependencies.

In trainer.py, parser arguments are used to give some choices to the user regarding batchsize of CIFAR-10 to be used for training. Also, for data normalization and augmentation as well as to initialize the model. Then, the dataset from CIFAR-10 is loaded and then cropped to size (insize 24). Then, model CNN is imported to prepare Convolutional Neural Network model. The optimizer is then called for **gradient descent** regarding the loss function. After that, calculate class is defined to train further using tqdm thereby calculating sum loss and sum accuracy in the process. At the end, learning loop is then run to keep track of time and to publish the epoch number and progress in the terminal window.

#trainer.py

```
#-------------------------------------------------------------------------------------------------------------------
import argparse

import numpy as np
import six
from tqdm import tqdm

import chainer
import chainer.links as L
from chainer import optimizers
from chainer import serializers

import time
import datahandler as dh


parser = argparse.ArgumentParser(description='Example: cifar-10')
parser.add_argument('--augmentation', '-a', default=1.0, type=float,
                   help='The amount of data augmentation')
parser.add_argument('--batchsize', '-b', default=100, type=int,
                 help='Batch size of training')
parser.add_argument('--data', '-d', choices=('on', 'off'),
                 default='off', help='Data normalization & augmentation')
parser.add_argument('--initmodel', '-m', default='',
                 help='Initialize the model from given file')

args = parser.parse_args()

# Prepare dataset
```

```
print('load cifar-10 dataset')
if args.data == 'on': cifar = dh.process_data(augmentation=args.augmentation)
else: cifar = dh.load_data()

# Cropping => insize = 24
cifar['train']['x'], cifar['train']['y'] = dh.crop_data(cifar['train']['x'], cifar['train']['y'])
cifar['test']['x'], cifar['test']['y'] = dh.crop_data(cifar['test']['x'], cifar['test']['y'])

N, N_test = len(cifar['train']['x']), len(cifar['test']['x'])
print(N, N_test)
batchsize = args.batchsize
n_epoch = args.epoch
assert N % batchsize == 0
assert N_test % batchsize == 0


# Prepare Convolution NN model
    import model_cnn
    model = model_cnn.CifarCNN_bn_crop()


# Setup optimizer
    optimizer = optimizers.MomentumSGD(lr=0.01)
    optimizer.setup(model)

# Init/Resume
if args.initmodel:
    print('Load model from', args.initmodel)
    serializers.load_hdf5(args.initmodel, model)
if args.resume:
    print('Load optimizer state from', args.resume)
    serializers.load_hdf5(args.resume, optimizer)


def calculate(model, data, optimizer, mean_loss, ac, N, batchsize):
    sum_accuracy, sum_loss = 0, 0

    if model.train:
        perm = np.random.permutation(N)
        for i in tqdm(six.moves.range(0, N, batchsize)):
            x_batch = np.reshape(data['train']['x'][perm[i:i + batchsize]],
                            (batchsize, 3, model.insize, model.insize))
            y_batch = data['train']['y'][perm[i:i + batchsize]]

            x = chainer.Variable(xp.asarray(x_batch), volatile='off')
            t = chainer.Variable(xp.asarray(y_batch), volatile='off')

            optimizer.update(model, x, t)

            sum_loss += float(model.loss.data) * len(t.data)
            sum_accuracy += float(model.accuracy.data) * len(t.data)
        print('train mean loss={},
            accuracy={}'.format( sum_loss / N,
            sum_accuracy / N))
        mean_loss.append(sum_loss / N)
        ac.append(sum_accuracy / N)
        if len(mean_loss) > 1 and mean_loss[-2:-1] < 0.02:
            optimizer.lr = optimizer.lr / 2.0
    else:
        for i in tqdm(six.moves.range(0, N, batchsize)):
            val_x_batch = np.reshape(data['test']['x'][i:i + batchsize],
                                (batchsize, 3, model.insize, model.insize))
            val_y_batch = data['test']['y'][i:i + batchsize]

            x = chainer.Variable(xp.asarray(val_x_batch), volatile='on')
            t = chainer.Variable(xp.asarray(val_y_batch), volatile='on')

            loss = model(x, t)
            sum_loss += float(loss.data) * len(t.data)
            sum_accuracy += float(model.accuracy.data) * len(t.data)

        print('test mean loss={},
            accuracy={}'.format( sum_loss / N_test,
            sum_accuracy / N))
        mean_loss.append(sum_loss / N)
        ac.append(sum_accuracy / N)

    return model, optimizer, mean_loss, ac


# Learning loop
train_ac, test_ac, train_mean_loss, test_mean_loss = [], [], [], []
stime = time.clock()
for epoch in six.moves.range(1, n_epoch + 1):
    print('epoch', epoch)
    # training
    print('Training...')
    model.train = True
    model, optimizer, train_mean_loss, train_ac = calcNN(model, cifar, optimizer, train_mean_loss,
train_ac, N, batchsize)

    # evaluation
    model.train = False
    model, optimizer, test_mean_loss, testac = calcNN(model, cifar, optimizer, test_mean_loss,
test_ac, N_test, batchsize)
#-----------------------------------------------------------------------------------------------------------------
```

Next, in case of **datahandler.py**, it is used to normalize data to feed into the input. Contrast normalization helps in optimization and to avoid local minima.
This program is also used to add noise which can speed convergence in the backpropagation training of a convolutional neural network. The noisy CNN algorithm speeds training on average because the backpropagation algorithm is a special case of the generalized expectation maximization (EM) algorithm and because such carefully chosen noise always speeds up the EM algorithm on average.

The noise benefit is most pronounced for smaller data sets because the largest EM hill-climbing gains tend to occur in the first few iterations[1].
Also this program pads data allowing for deeper networks and improved performance

#datahandler.py

```
#-------------------------------------------------------------------------------------------------------------------
import numpy as np
from numpy import linalg as la
import os
import data
import six
from tqdm import tqdm


def normalize(data, M=None, Sd=None):
    print('Data normalization..... ')
    if M is None:
        M = np.mean(data, axis=0)        # mean
    if Sd is None:
        Sd = np.std(data, axis=0)        # Std

    stmat = np.zeros([len(Sd), len(Sd)])
    for i in range(0, len(Sd)):
        stmat[i][i] = Sd[i]
    S_inv = la.inv(np.matrix(stmat))
    data_n = S_inv.dot((np.matrix(data - M)).T)
    data_n = np.array(data_n.T, dtype=np.float32)
    return data_n, M, Sd


def pad_addnoise(data_x, data_y, mean=0.0, sd=1.0, mixratio=1.0, noiseratio=0.3):
    print('Data padding: adding noise to data......')
    col, row = np.int(len(data_x) * mixratio), len(data_x[0])
    noise = np.random.normal(mean, sd, (col, row)) * np.sqrt(noiseratio)
    noised = np.array(data_x[0:col] * np.sqrt(1 - noiseratio) + noise, dtype=np.float32)
    noised_x = np.append(data_x, noised, axis=0)
    noised_y = np.append(data_y, data_y[0:col], axis=0)
    return noised_x, noised_y


def pad_rightleft(data_x, data_y, mixratio=1.0, ch=3):
    print('Data padding: rightside left......')
    col, size = np.int(len(data_x) * mixratio), len(data_x[0]) / ch
    height, width = 32, 32

    rl = data_x.copy()
    for i in tqdm(range(0, ch)):
        for j in range(0, height):
            r = data_x[:, i * size + j * height:i * size + j * height + width]
            rl[:, i * size + j * height:i * size + j * height + width] = np.fliplr(r)
    rl_x = np.append(data_x, rl[0:col], axis=0)
    rl_y = np.append(data_y, data_y[0:col], axis=0)
    return rl_x, rl_y


def crop_data(data_x, data_y, imagesize=32, insize=24, stride=4, ch=3):
    print('Data cropping: images x 3 x 3......')
    ratio = (imagesize - insize) / stride + 1
    imagemat, labelvec = data_x, data_y
    nimage = len(imagemat)

    cropped_x = np.zeros([nimage * ratio ** 2, insize ** 2 * ch])
    cropped_y = np.zeros(nimage * ratio ** 2)
    for i in tqdm(range(0, nimage)):
        for c in range(0,ch):
            imagemat = data_x[i][c * imagesize ** 2:(c + 1) * imagesize ** 2]
            for j in range(0, ratio):
                xind = j * stride
                for k in range(0, ratio):
                    yind = k * stride
                    cropped_x[i * ratio**2 + j * ratio + k, c * insize ** 2:(c + 1) * insize ** 2] =
np.reshape(
                        np.reshape(imagemat, (imagesize, imagesize))[yind:yind + insize, xind:xind +
 insize],
                        (1, insize ** 2))
        cropped_y[i * ratio ** 2: (i + 1) * ratio ** 2] = np.ones(ratio ** 2).astype(np.int32) *
labelvec[i]

    data_x = cropped_x.astype(np.float32)
    data_y = np.array(cropped_y, dtype=np.int32)
    return data_x, data_y



def process_data(augmentation=2):
    cifar = load_data()
    cifar['train']['x'], m, sd = normalize(cifar['train']['x'])
    cifar['test']['x'], m, sd = normalize(cifar['test']['x'], M=m, Sd=sd)
    if augmentation > 0:
        cifar['train']['x'], cifar['train']['y'] = pad_rightleft(cifar['train']['x'], cifar['train']['y'],
                                                                  mixratio=augmentation)
    if augmentation > 1.0:
        cifar['train']['x'], cifar['train']['y'] = pad_addnoise(cifar['train']['x'], cifar['train']['y'],
                                                                 mixratio=augmentation - 1.0)
#    data.save_pkl(cifar, savename='cifar_processed.pkl')
    return cifar



def load_processed_data():
    if not os.path.exists('cifar_processed.pkl'):
        cifar = process_data()
        return cifar
    else:
        with open('cifar_processed.pkl', 'rb') as cifar_pickle:
```

```
        data = six.moves.cPickle.load(cifar_pickle)
    return data


def load_data():
    cifar = data.load_data()
    cifar['train']['x'] = cifar['train']['x'].astype(np.float32)
    cifar['test']['x'] = cifar['test']['x'].astype(np.float32)
    cifar['train']['x'] /= 255
    cifar['test']['x'] /= 255
    cifar['train']['y'] = np.array(cifar['train']['y'], dtype=np.int32)
    cifar['test']['y'] = np.array(cifar['test']['y'], dtype=np.int32)
    return cifar
#-----------------------------------------------------------------------------------------------------------------
```

Now, moving on to **model_cnn.py**, it is used for convolution and max pooling and repeated iterations of these two to make the network deeper and more efficient in learning. In this program, four instances of convolution and max pooling are used for the abovementioned goal.

#model_cnn.py

```
#-----------------------------------------------------------------------------------------------------------------
import chainer
import chainer.functions as F
import chainer.links as L
from chainer.utils import conv


class CifarCNN(chainer.Chain):

    """
    Single-GPU AlexNet without partition toward the channel axis.
    Number of units in each layer was arranged for cifar-10 example,
    because input image = 32 x 32 x 3.
    """

    insize = 32

    def_init__(self):
        super(CifarCNN, self)._init_(
            conv1=L.Convolution2D(3, 96, 3, stride=2),
            conv2=L.Convolution2D(96, 256, 3, pad=1),
            conv3=L.Convolution2D(256, 384,  3, pad=1),
            conv4=L.Convolution2D(384, 384,  2, pad=1),
            conv5=L.Convolution2D(384, 256,  2, pad=1),
            fc6=L.Linear(2304, 2304),
            fc7=L.Linear(2304, 512),
            fc8=L.Linear(512, 10),
            )
        self.train = True

    def_call_(self, x, t):
        h =
            F.max_pooling_2d(F.relu( F.local_response_normalization(
            self.conv1(x))), 3, stride=2)
        h =
            F.max_pooling_2d(F.relu( F.local_response_normalization(
            self.conv2(h))), 3, stride=2)
        h = F.relu(self.conv3(h))
        h = F.relu(self.conv4(h))
        h = F.max_pooling_2d(F.relu(self.conv5(h)), 3, stride=1)
        h = F.dropout(F.relu(self.fc6(h)), train=self.train)
        h = F.dropout(F.relu(self.fc7(h)), train=self.train)
        h = self.fc8(h)

        self.loss = F.softmax_cross_entropy(h, t)
        self.accuracy = F.accuracy(h, t)
        return self.loss


class CifarCNN_2(chainer.Chain):

    """
    Single-GPU AlexNet without partition toward the channel axis.
    Number of units in each layer was arranged for cifar-10 example,
    because input image = 32 x 32 x 3.
    """
    insize = 32

    def_init__(self):
        super(CifarCNN_2, self)._init_(
            conv1=L.Convolution2D(3, 96, 3, pad=1),
            conv2=L.Convolution2D(96, 256, 3, pad=1),
            conv3=L.Convolution2D(256, 384,  3, pad=1),
            conv4=L.Convolution2D(384, 384,  3, pad=1),
            conv5=L.Convolution2D(384, 256,  3, pad=1),
            fc6=L.Linear(1024, 1024),
            fc7=L.Linear(1024, 128),
            fc8=L.Linear(128, 10),
            )
        self.train = True

    def_call_(self, x, t):
        h =
            F.max_pooling_2d(F.relu( F.local_response_normalization(
            self.conv1(x))), 2, stride=2)
        h =
            F.max_pooling_2d(F.relu( F.local_response_normalization(
            self.conv2(h))), 2, stride=2)
        h = F.dropout(F.relu(self.conv3(h)), ratio=0.7, train=self.train)
        h = F.max_pooling_2d(F.relu(self.conv4(h)), 2, stride=2)
        h = F.max_pooling_2d(F.relu(self.conv5(h)), 2, stride=2, cover_all=True)
        h = F.dropout(F.relu(self.fc6(h)), ratio=0.7, train=self.train)
        h = F.dropout(F.relu(self.fc7(h)), ratio=0.7, train=self.train)
        h = self.fc8(h)

        self.loss = F.softmax_cross_entropy(h, t)
self.accuracy = F.accuracy(h, t)
```

```
        return self.loss


class CifarCNN_bn(chainer.Chain):
    """Single-GPU AlexNet with LRN layers replaced by BatchNormalization."""
    insize = 32

    def_init__(self):
        super(CifarCNN_bn, self)._init_(
            conv1=L.Convolution2D(3, 96, 3, pad=1),
            bn1=L.BatchNormalization(96),
            conv2=L.Convolution2D(96, 256, 3, pad=1),
            bn2=L.BatchNormalization(256),
            conv3=L.Convolution2D(256, 384,  3, pad=1),
            conv4=L.Convolution2D(384, 384,  3, pad=1),
            conv5=L.Convolution2D(384, 256,  3, pad=1),
            fc6=L.Linear(1024, 128),
            fc7=L.Linear(128, 10),
            )
        self.train = True

    def call_(self, x, t):
        h = self.bn1(self.conv1(x), test=not self.train)
        h = F.max_pooling_2d(F.relu(h), 2, stride=2)
        h = self.bn2(self.conv2(h), test=not self.train)
        h = F.max_pooling_2d(F.relu(h), 2, stride=2)
        h = F.dropout(F.relu(self.conv3(h)), ratio=0.6, train=self.train)
        h = F.max_pooling_2d(F.relu(self.conv4(h)), 2, stride=2)
        h = F.average_pooling_2d(F.relu(self.conv5(h)), 2, stride=2)
        h = F.dropout(F.relu(self.fc6(h)), ratio=0.6, train=self.train)
        h = self.fc7(h)

        self.loss = F.softmax_cross_entropy(h, t)
        self.accuracy = F.accuracy(h, t)
        return self.loss


class CifarCNN_bn_crop(chainer.Chain):
    """Single-GPU AlexNet with LRN layers replaced by BatchNormalization."""
    insize = 24

    def_init__(self):
        super(CifarCNN_bn_crop, self)._init_(
            conv1=L.Convolution2D(3, 96, 3, pad=1),
            bn1=L.BatchNormalization(96),
            conv2=L.Convolution2D(96, 256, 3, pad=1),
            bn2=L.BatchNormalization(256),
            conv3=L.Convolution2D(256, 384,  3, pad=1),
            conv4=L.Convolution2D(384, 384,  3, pad=1),
            conv5=L.Convolution2D(384, 256,  3, pad=1),
            fc6=L.Linear(256, 256),
            fc7=L.Linear(256, 10),
        )
        self.train = True

    def_call_(self, x, t, predict=False):
        h = self.bn1(self.conv1(x), test=not self.train)
        h = F.max_pooling_2d(F.relu(h), 2, stride=2)
        h = self.bn2(self.conv2(h), test=not self.train)
        h = F.max_pooling_2d(F.relu(h), 2, stride=2)
        h = F.dropout(F.relu(self.conv3(h)), ratio=0.6, train=self.train)
        h = F.max_pooling_2d(F.relu(self.conv4(h)), 2, stride=2)
        h = F.average_pooling_2d(F.relu(self.conv5(h)), 3, stride=1)
        h = F.dropout(F.relu(self.fc6(h)), ratio=0.6, train=self.train)
        h = self.fc7(h)

        self.loss = F.softmax_cross_entropy(h, t)
        self.accuracy = F.accuracy(h, t)
        if predict:
            return h
        else:
            return self.loss

#------------------------------------------------------------------------------------------------------------------
```

Reference

[1] Noise enhanced convolutional neural networks. Kartik Audhkasi, Osonde Osobe, Bart Kosko, University of Southern California