Deep Learning Project 1

CNN & Data Augmentation Summary

Basic LeNet CNN

- LeNet architecture: 2 conv layers → 2 FC layers → output
- CIFAR dataset: 32x32 RGB images, 10 classes
- Standard CNN with convolution, pooling, fully connected layers

Data Augmentation Techniques

- **Translation**: ±5 pixel random shifts
- **Rotation**: ±20 degree random rotations
- **Horizontal Flip**: 50% probability
- **Noise Injection**: Gaussian/uniform noise (zero mean)

Advanced CNN Design

- **Target**: Achieve 80%+ accuracy on CIFAR
- **Techniques**: Batch normalization, dropout, ReLU, momentum optimization
- **Regularization**: L1/L2 penalties
- **Architecture**: Deeper network (5 FC layers vs 2)

Custom CNN Architecture

- Implement specific CNN design optimization

Skills Demonstrated

- Computer vision with CNNs
- Data augmentation
- Batch norm, dropout, momentum
- Performance optimization and regularization

```
Shuxin Tang Deep Learning Project 1
"""
import numpy
import numpy as np
from math import floor
from random import randrange

import theano
import theano.tensor as T
from theano.tensor.signal import downsample

from hw3_utils import shared_dataset, load_data
from hw3_nn import LogisticRegression, HiddenLayer, LeNetConvPoolLayer, train_nn
from hw3_nn import drop, DropoutHiddenLayer
from hw3_nn import BatchNormalization
from hw3_nn import ConvLayer, PoolLayer

import matplotlib.pyplot as plt

from scipy.ndimage import shift
from scipy.ndimage import rotate
from numpy import fliplr


#Problem 1
#Implement the convolutional neural network architecture depicted in HW3 problem 1
#Reference code can be found in http://deeplearning.net/tutorial/code/convolutional_mlp.py
def test_lenet(learning_rate=0.1, n_epochs=200, nkerns=[20, 50], batch_size=500):

    rng = numpy.random.RandomState(23455)

    datasets = load_data()
```

```
train_set_x, train_set_y = datasets[0]
valid_set_x, valid_set_y = datasets[1]
test_set_x, test_set_y = datasets[2]

# compute number of minibatches for training, validation and testing
n_train_batches = train_set_x.get_value(borrow=True).shape[0]
n_valid_batches = valid_set_x.get_value(borrow=True).shape[0]
n_test_batches = test_set_x.get_value(borrow=True).shape[0]
n_train_batches //= batch_size
n_valid_batches //= batch_size
n_test_batches //= batch_size

# allocate symbolic variables for the data
index = T.lscalar()  # index to a [mini]batch

# start-snippet-1
x = T.matrix('x')   # the data is presented as rasterized images
y = T.ivector('y')  # the labels are presented as 1D vector of
                    # [int] labels

######################
# BUILD ACTUAL MODEL #
######################
print('... building the model')

# Reshape matrix of rasterized images of shape (batch_size, 28 * 28)
# to a 4D tensor, compatible with our LeNetConvPoolLayer
# (28, 28) is the size of MNIST images.
layer0_input = x.reshape((batch_size, 3, 32, 32))

# Construct the first convolutional pooling layer:
# filtering reduces the image size to (28-5+1 , 28-5+1) = (24, 24)
# maxpooling reduces this further to (24/2, 24/2) = (12, 12)
# 4D output tensor is thus of shape (batch_size, nkerns[0], 12, 12)
layer0 = LeNetConvPoolLayer(
    rng,
    input=layer0_input,
```

```
    image_shape=(batch_size, 3, 32, 32),
    filter_shape=(nkerns[0], 3, 3, 3),
    poolsize=(2, 2)
)

# Construct the second convolutional pooling layer
# filtering reduces the image size to (12-5+1, 12-5+1) = (8, 8)
# maxpooling reduces this further to (8/2, 8/2) = (4, 4)
# 4D output tensor is thus of shape (batch_size, nkerns[1], 4, 4)
layer1 = LeNetConvPoolLayer(
    rng,
    input=layer0.output,
    image_shape=(batch_size, nkerns[0], 15, 15),
    filter_shape=(nkerns[1], nkerns[0], 3, 3),
    poolsize=(2, 2)
)

# the HiddenLayer being fully-connected, it operates on 2D matrices of
# shape (batch_size, num_pixels) (i.e matrix of rasterized images).
# This will generate a matrix of shape (batch_size, nkerns[1] * 4 * 4),
# or (500, 50 * 4 * 4) = (500, 800) with the default values.
layer2_input = layer1.output.flatten(2)

# construct a fully-connected sigmoidal layer
layer2 = HiddenLayer(
    rng,
    input=layer2_input,
    n_in=nkerns[1] * 6 * 6,
    n_out=4096,
    activation=T.tanh
)

# construct a fully-connected sigmoidal layer
layer3 = HiddenLayer(
    rng,
    input=layer2.output,
    n_in=4096,
```

```
    n_out=512,
    activation=T.tanh
)

# classify the values of the fully-connected sigmoidal layer
layer4 = LogisticRegression(input=layer3.output, n_in=512, n_out=10)

# the cost we minimize during training is the NLL of the model
cost = layer4.negative_log_likelihood(y)

# create a function to compute the mistakes that are made by the model
test_model = theano.function(
    [index],
    layer4.errors(y),
    givens={
        x: test_set_x[index * batch_size: (index + 1) * batch_size],
        y: test_set_y[index * batch_size: (index + 1) * batch_size]
    }
)

validate_model = theano.function(
    [index],
    layer4.errors(y),
    givens={
        x: valid_set_x[index * batch_size: (index + 1) * batch_size],
        y: valid_set_y[index * batch_size: (index + 1) * batch_size]
    }
)

# create a list of all model parameters to be fit by gradient descent
params = layer4.params + layer3.params + layer2.params + layer1.params + layer0.params

# create a list of gradients for all model parameters
grads = T.grad(cost, params)

# train_model is a function that updates the model parameters by
# SGD Since this model has many parameters, it would be tedious to
```

```
    # manually create an update rule for each model parameter. We thus
    # create the updates list by automatically looping over all
    # (params[i], grads[i]) pairs.
    updates = [
        (param_i, param_i - learning_rate * grad_i)
        for param_i, grad_i in zip(params, grads)
    ]

    train_model = theano.function(
        [index],
        cost,
        updates=updates,
        givens={
            x: train_set_x[index * batch_size: (index + 1) * batch_size],
            y: train_set_y[index * batch_size: (index + 1) * batch_size]
        }
    )
    # end-snippet-1

    ###############
    # TRAIN MODEL #
    ###############
    train_nn(train_model, validate_model, test_model,
            n_train_batches, n_valid_batches, n_test_batches, n_epochs,
            verbose = True)


#Problem 2.1
#Write a function to add translations
def translate_image( img ):

    m = int(randrange(-5, 5)) # m pixels to left
    n = int(randrange(-5, 5)) # n pixels to up
    img_shift = shift(input=img, shift=(-1*n, -1*m, 0))

    return img_shift
```

```
#Implement a convolutional neural network with the translation method for augmentation
def test_lenet_translation(learning_rate=0.1, n_epochs=200, nkerns=[20, 50], batch_size=500):

    rng = numpy.random.RandomState(23455)

    ds_rate=None
    datasets = load_data(ds_rate=ds_rate,theano_shared=False)

    train_set_x, train_set_y = datasets[0]
    train_size = train_set_x.shape
    n_train = train_size[0]

    train_set_x_aug = np.empty(train_size)

    print '... Translating images'

    for i in range(n_train):
        img = (np.reshape(train_set_x[i],(3,32,32))).transpose(1,2,0)
        img_tran = translate_image(img)
        train_set_x_aug[i] = np.reshape(img_tran.transpose(2,0,1),(3*32*32))

    train_set_x = np.append(train_set_x, train_set_x_aug, axis=0)
    train_set_y = np.append(train_set_y, train_set_y, axis=0)

    #train_set_x = train_set_x_aug

    datasets[0] = [train_set_x, train_set_y]

    train_set_x, train_set_y = shared_dataset(datasets[0])
    valid_set_x, valid_set_y = shared_dataset(datasets[1])
    test_set_x, test_set_y = shared_dataset(datasets[2])

    # compute number of minibatches for training, validation and testing
    n_train_batches = train_set_x.get_value(borrow=True).shape[0]
    n_valid_batches = valid_set_x.get_value(borrow=True).shape[0]
    n_test_batches = test_set_x.get_value(borrow=True).shape[0]
```

```
n_train_batches //= batch_size
n_valid_batches //= batch_size
n_test_batches //= batch_size

# allocate symbolic variables for the data
index = T.lscalar()  # index to a [mini]batch

# start-snippet-1
x = T.matrix('x')   # the data is presented as rasterized images
y = T.ivector('y')  # the labels are presented as 1D vector of
                    # [int] labels

######################
# BUILD ACTUAL MODEL #
######################
print('... building the model')

# Reshape matrix of rasterized images of shape (batch_size, 28 * 28)
# to a 4D tensor, compatible with our LeNetConvPoolLayer
# (28, 28) is the size of MNIST images.
layer0_input = x.reshape((batch_size, 3, 32, 32))

# Construct the first convolutional pooling layer:
# filtering reduces the image size to (28-5+1 , 28-5+1) = (24, 24)
# maxpooling reduces this further to (24/2, 24/2) = (12, 12)
# 4D output tensor is thus of shape (batch_size, nkerns[0], 12, 12)
layer0 = LeNetConvPoolLayer(
    rng,
    input=layer0_input,
    image_shape=(batch_size, 3, 32, 32),
    filter_shape=(nkerns[0], 3, 3, 3),
    poolsize=(2, 2)
)

# Construct the second convolutional pooling layer
# filtering reduces the image size to (12-5+1, 12-5+1) = (8, 8)
# maxpooling reduces this further to (8/2, 8/2) = (4, 4)
```

```
# 4D output tensor is thus of shape (batch_size, nkerns[1], 4, 4)
layer1 = LeNetConvPoolLayer(
    rng,
    input=layer0.output,
    image_shape=(batch_size, nkerns[0], 15, 15),
    filter_shape=(nkerns[1], nkerns[0], 3, 3),
    poolsize=(2, 2)
)

# the HiddenLayer being fully-connected, it operates on 2D matrices of
# shape (batch_size, num_pixels) (i.e matrix of rasterized images).
# This will generate a matrix of shape (batch_size, nkerns[1] * 4 * 4),
# or (500, 50 * 4 * 4) = (500, 800) with the default values.
layer2_input = layer1.output.flatten(2)

# construct a fully-connected sigmoidal layer
layer2 = HiddenLayer(
    rng,
    input=layer2_input,
    n_in=nkerns[1] * 6 * 6,
    n_out=4096,
    activation=T.tanh
)

# construct a fully-connected sigmoidal layer
layer3 = HiddenLayer(
    rng,
    input=layer2.output,
    n_in=4096,
    n_out=512,
    activation=T.tanh
)

# classify the values of the fully-connected sigmoidal layer
layer4 = LogisticRegression(input=layer3.output, n_in=512, n_out=10)

# the cost we minimize during training is the NLL of the model
```

```
cost = layer4.negative_log_likelihood(y)

# create a function to compute the mistakes that are made by the model
test_model = theano.function(
    [index],
    layer4.errors(y),
    givens={
        x: test_set_x[index * batch_size: (index + 1) * batch_size],
        y: test_set_y[index * batch_size: (index + 1) * batch_size]
    }
)

validate_model = theano.function(
    [index],
    layer4.errors(y),
    givens={
        x: valid_set_x[index * batch_size: (index + 1) * batch_size],
        y: valid_set_y[index * batch_size: (index + 1) * batch_size]
    }
)

# create a list of all model parameters to be fit by gradient descent
params = layer4.params + layer3.params + layer2.params + layer1.params + layer0.params

# create a list of gradients for all model parameters
grads = T.grad(cost, params)

# train_model is a function that updates the model parameters by
# SGD Since this model has many parameters, it would be tedious to
# manually create an update rule for each model parameter. We thus
# create the updates list by automatically looping over all
# (params[i], grads[i]) pairs.
updates = [
    (param_i, param_i - learning_rate * grad_i)
    for param_i, grad_i in zip(params, grads)
]
```

```
    train_model = theano.function(
        [index],
        cost,
        updates=updates,
        givens={
            x: train_set_x[index * batch_size: (index + 1) * batch_size],
            y: train_set_y[index * batch_size: (index + 1) * batch_size]
        }
    )
    # end-snippet-1

    ###############
    # TRAIN MODEL #
    ###############
    train_nn(train_model, validate_model, test_model,
            n_train_batches, n_valid_batches, n_test_batches, n_epochs,
            verbose = True)



#Problem 2.2
#Write a function to add roatations
def rotate_image( img ):

    img_shape = img.shape
    angle = randrange(-20, 20)
    img_rotate = rotate(input=img, angle=angle)
    rot_shape = img_rotate.shape
    x = int(floor(rot_shape[0]/2)) - int(floor(img_shape[0]/2))
    y = int(floor(rot_shape[1]/2)) - int(floor(img_shape[1]/2))
    img_rotate = img_rotate[x:x+img_shape[0],y:y+img_shape[1],:]

    return img_rotate



#Implement a convolutional neural network with the rotation method for augmentation
def test_lenet_rotation(learning_rate=0.1, n_epochs=200, nkerns=[20, 50], batch_size=500):
```

```
rng = numpy.random.RandomState(23455)

ds_rate=None
datasets = load_data(ds_rate=ds_rate,theano_shared=False)

train_set_x, train_set_y = datasets[0]
train_size = train_set_x.shape
n_train = train_size[0]

train_set_x_aug = np.empty(train_size)

print '... Rotating images'

for i in range(n_train):
    img = (np.reshape(train_set_x[i],(3,32,32))).transpose(1,2,0)
    img_tran = rotate_image(img)
    train_set_x_aug[i] = np.reshape(img_tran.transpose(2,0,1),(3*32*32))

train_set_x = np.append(train_set_x, train_set_x_aug, axis=0)
train_set_y = np.append(train_set_y, train_set_y, axis=0)

datasets[0] = [train_set_x, train_set_y]

train_set_x, train_set_y = shared_dataset(datasets[0])
valid_set_x, valid_set_y = shared_dataset(datasets[1])
test_set_x, test_set_y = shared_dataset(datasets[2])

# compute number of minibatches for training, validation and testing
n_train_batches = train_set_x.get_value(borrow=True).shape[0]
n_valid_batches = valid_set_x.get_value(borrow=True).shape[0]
n_test_batches = test_set_x.get_value(borrow=True).shape[0]
n_train_batches //= batch_size
n_valid_batches //= batch_size
n_test_batches //= batch_size

# allocate symbolic variables for the data
```

```
index = T.lscalar()  # index to a [mini]batch

# start-snippet-1
x = T.matrix('x')   # the data is presented as rasterized images
y = T.ivector('y')  # the labels are presented as 1D vector of
                    # [int] labels

######################
# BUILD ACTUAL MODEL #
######################
print('... building the model')

# Reshape matrix of rasterized images of shape (batch_size, 28 * 28)
# to a 4D tensor, compatible with our LeNetConvPoolLayer
# (28, 28) is the size of MNIST images.
layer0_input = x.reshape((batch_size, 3, 32, 32))

# Construct the first convolutional pooling layer:
# filtering reduces the image size to (28-5+1 , 28-5+1) = (24, 24)
# maxpooling reduces this further to (24/2, 24/2) = (12, 12)
# 4D output tensor is thus of shape (batch_size, nkerns[0], 12, 12)
layer0 = LeNetConvPoolLayer(
    rng,
    input=layer0_input,
    image_shape=(batch_size, 3, 32, 32),
    filter_shape=(nkerns[0], 3, 3, 3),
    poolsize=(2, 2)
)

# Construct the second convolutional pooling layer
# filtering reduces the image size to (12-5+1, 12-5+1) = (8, 8)
# maxpooling reduces this further to (8/2, 8/2) = (4, 4)
# 4D output tensor is thus of shape (batch_size, nkerns[1], 4, 4)
layer1 = LeNetConvPoolLayer(
    rng,
    input=layer0.output,
    image_shape=(batch_size, nkerns[0], 15, 15),
```

```
    filter_shape=(nkerns[1], nkerns[0], 3, 3),
    poolsize=(2, 2)
)

# the HiddenLayer being fully-connected, it operates on 2D matrices of
# shape (batch_size, num_pixels) (i.e matrix of rasterized images).
# This will generate a matrix of shape (batch_size, nkerns[1] * 4 * 4),
# or (500, 50 * 4 * 4) = (500, 800) with the default values.
layer2_input = layer1.output.flatten(2)

# construct a fully-connected sigmoidal layer
layer2 = HiddenLayer(
    rng,
    input=layer2_input,
    n_in=nkerns[1] * 6 * 6,
    n_out=4096,
    activation=T.tanh
)

# construct a fully-connected sigmoidal layer
layer3 = HiddenLayer(
    rng,
    input=layer2.output,
    n_in=4096,
    n_out=512,
    activation=T.tanh
)

# classify the values of the fully-connected sigmoidal layer
layer4 = LogisticRegression(input=layer3.output, n_in=512, n_out=10)

# the cost we minimize during training is the NLL of the model
cost = layer4.negative_log_likelihood(y)

# create a function to compute the mistakes that are made by the model
test_model = theano.function(
    [index],
```

```
    layer4.errors(y),
    givens={
        x: test_set_x[index * batch_size: (index + 1) * batch_size],
        y: test_set_y[index * batch_size: (index + 1) * batch_size]
    }
)

validate_model = theano.function(
    [index],
    layer4.errors(y),
    givens={
        x: valid_set_x[index * batch_size: (index + 1) * batch_size],
        y: valid_set_y[index * batch_size: (index + 1) * batch_size]
    }
)

# create a list of all model parameters to be fit by gradient descent
params = layer4.params + layer3.params + layer2.params + layer1.params + layer0.params

# create a list of gradients for all model parameters
grads = T.grad(cost, params)

# train_model is a function that updates the model parameters by
# SGD Since this model has many parameters, it would be tedious to
# manually create an update rule for each model parameter. We thus
# create the updates list by automatically looping over all
# (params[i], grads[i]) pairs.
updates = [
    (param_i, param_i - learning_rate * grad_i)
    for param_i, grad_i in zip(params, grads)
]

train_model = theano.function(
    [index],
    cost,
    updates=updates,
    givens={
```

```
            x: train_set_x[index * batch_size: (index + 1) * batch_size],
            y: train_set_y[index * batch_size: (index + 1) * batch_size]
        }
    )
    # end-snippet-1

    ###############
    # TRAIN MODEL #
    ###############
    train_nn(train_model, validate_model, test_model,
            n_train_batches, n_valid_batches, n_test_batches, n_epochs,
            verbose = True)



#Problem 2.3
#Write a function to flip images
def flip_image( img ):

    k = np.random.uniform(low=0.0, high=1.0)
    if k > 0.5:  # 50% of probability to flip horizontally
        img_flip = fliplr(img)

    else:
        img_flip = img

    return img_flip


#Implement a convolutional neural network with the flip method for augmentation
def test_lenet_flip(learning_rate=0.1, n_epochs=200, nkerns=[20, 50], batch_size=500):

    rng = numpy.random.RandomState(23455)

    ds_rate=None
    datasets = load_data(ds_rate=ds_rate,theano_shared=False)
```

```
train_set_x, train_set_y = datasets[0]
train_size = train_set_x.shape
n_train = train_size[0]

train_set_x_aug = np.empty(train_size)

print '... Fliping images'

for i in range(n_train):
    img = (np.reshape(train_set_x[i],(3,32,32))).transpose(1,2,0)
    img_tran = flip_image(img)
    train_set_x_aug[i] = np.reshape(img_tran.transpose(2,0,1),(3*32*32))

train_set_x = np.append(train_set_x, train_set_x_aug, axis=0)
train_set_y = np.append(train_set_y, train_set_y, axis=0)

datasets[0] = [train_set_x, train_set_y]

train_set_x, train_set_y = shared_dataset(datasets[0])
valid_set_x, valid_set_y = shared_dataset(datasets[1])
test_set_x, test_set_y = shared_dataset(datasets[2])

# compute number of minibatches for training, validation and testing
n_train_batches = train_set_x.get_value(borrow=True).shape[0]
n_valid_batches = valid_set_x.get_value(borrow=True).shape[0]
n_test_batches = test_set_x.get_value(borrow=True).shape[0]
n_train_batches //= batch_size
n_valid_batches //= batch_size
n_test_batches //= batch_size

# allocate symbolic variables for the data
index = T.lscalar()  # index to a [mini]batch

# start-snippet-1
x = T.matrix('x')   # the data is presented as rasterized images
y = T.ivector('y')  # the labels are presented as 1D vector of
                    # [int] labels
```

```
######################
# BUILD ACTUAL MODEL #
######################
print('... building the model')

# Reshape matrix of rasterized images of shape (batch_size, 28 * 28)
# to a 4D tensor, compatible with our LeNetConvPoolLayer
# (28, 28) is the size of MNIST images.
layer0_input = x.reshape((batch_size, 3, 32, 32))

# Construct the first convolutional pooling layer:
# filtering reduces the image size to (28-5+1 , 28-5+1) = (24, 24)
# maxpooling reduces this further to (24/2, 24/2) = (12, 12)
# 4D output tensor is thus of shape (batch_size, nkerns[0], 12, 12)
layer0 = LeNetConvPoolLayer(
    rng,
    input=layer0_input,
    image_shape=(batch_size, 3, 32, 32),
    filter_shape=(nkerns[0], 3, 3, 3),
    poolsize=(2, 2)
)

# Construct the second convolutional pooling layer
# filtering reduces the image size to (12-5+1, 12-5+1) = (8, 8)
# maxpooling reduces this further to (8/2, 8/2) = (4, 4)
# 4D output tensor is thus of shape (batch_size, nkerns[1], 4, 4)
layer1 = LeNetConvPoolLayer(
    rng,
    input=layer0.output,
    image_shape=(batch_size, nkerns[0], 15, 15),
    filter_shape=(nkerns[1], nkerns[0], 3, 3),
    poolsize=(2, 2)
)

# the HiddenLayer being fully-connected, it operates on 2D matrices of
# shape (batch_size, num_pixels) (i.e matrix of rasterized images).
```

```
# This will generate a matrix of shape (batch_size, nkerns[1] * 4 * 4),
# or (500, 50 * 4 * 4) = (500, 800) with the default values.
layer2_input = layer1.output.flatten(2)

# construct a fully-connected sigmoidal layer
layer2 = HiddenLayer(
    rng,
    input=layer2_input,
    n_in=nkerns[1] * 6 * 6,
    n_out=4096,
    activation=T.tanh
)

# construct a fully-connected sigmoidal layer
layer3 = HiddenLayer(
    rng,
    input=layer2.output,
    n_in=4096,
    n_out=512,
    activation=T.tanh
)

# classify the values of the fully-connected sigmoidal layer
layer4 = LogisticRegression(input=layer3.output, n_in=512, n_out=10)

# the cost we minimize during training is the NLL of the model
cost = layer4.negative_log_likelihood(y)

# create a function to compute the mistakes that are made by the model
test_model = theano.function(
    [index],
    layer4.errors(y),
    givens={
        x: test_set_x[index * batch_size: (index + 1) * batch_size],
        y: test_set_y[index * batch_size: (index + 1) * batch_size]
    }
)
```

```
validate_model = theano.function(
    [index],
    layer4.errors(y),
    givens={
        x: valid_set_x[index * batch_size: (index + 1) * batch_size],
        y: valid_set_y[index * batch_size: (index + 1) * batch_size]
    }
)

# create a list of all model parameters to be fit by gradient descent
params = layer4.params + layer3.params + layer2.params + layer1.params + layer0.params

# create a list of gradients for all model parameters
grads = T.grad(cost, params)

# train_model is a function that updates the model parameters by
# SGD Since this model has many parameters, it would be tedious to
# manually create an update rule for each model parameter. We thus
# create the updates list by automatically looping over all
# (params[i], grads[i]) pairs.
updates = [
    (param_i, param_i - learning_rate * grad_i)
    for param_i, grad_i in zip(params, grads)
]

train_model = theano.function(
    [index],
    cost,
    updates=updates,
    givens={
        x: train_set_x[index * batch_size: (index + 1) * batch_size],
        y: train_set_y[index * batch_size: (index + 1) * batch_size]
    }
)
# end-snippet-1
```

```
    ################
    # TRAIN MODEL #
    ################
    train_nn(train_model, validate_model, test_model,
            n_train_batches, n_valid_batches, n_test_batches, n_epochs,
            verbose = True)




#Problem 2.4
#Write a function to add noise, it should at least provide Gaussian-distributed and uniform-distributed noise with zero mean
def noise_injection( img ):

    k = np.random.uniform(low=0.0, high=1.0)
    if k > 0.5:  # 50% of probability to add Gaussian noise
        img_noise = img + np.random.normal(loc=0, scale=0.005, size=img.shape)
    else:       # 50% of probability to add Uniform noise
        img_noise = img + np.random.uniform(low=-0.005, high=0.005, size=img.shape)

    return img_noise



#Implement a convolutional neural network with the augmentation of injecting noise into input
def test_lenet_inject_noise_input(learning_rate=0.1, n_epochs=200, nkerns=[20, 50], batch_size=500):

    rng = numpy.random.RandomState(23455)

    ds_rate=None
    datasets = load_data(ds_rate=ds_rate,theano_shared=False)

    train_set_x, train_set_y = datasets[0]
    valid_set_x, valid_set_y = datasets[1]
    test_set_x, test_set_y = datasets[2]

    train_size = train_set_x.shape
```

```
valid_size = valid_set_x.shape
test_size = test_set_x.shape
n_train = train_size[0]
n_valid = valid_size[0]
n_test = test_size[0]

train_set_x_aug = np.empty(train_size)
valid_set_x_aug = np.empty(valid_size)
test_set_x_aug = np.empty(test_size)

print '... Ennoising images'

for i in range(n_train):
    img = (np.reshape(train_set_x[i],(3,32,32))).transpose(1,2,0)
    img_tran = noise_injection(img)
    train_set_x_aug[i] = np.reshape(img_tran.transpose(2,0,1),(3*32*32))

train_set_x = np.append(train_set_x, train_set_x_aug, axis=0)
train_set_y = np.append(train_set_y, train_set_y, axis=0)

datasets[0] = [train_set_x, train_set_y]

train_set_x, train_set_y = shared_dataset(datasets[0])
valid_set_x, valid_set_y = shared_dataset(datasets[1])
test_set_x, test_set_y = shared_dataset(datasets[2])

# compute number of minibatches for training, validation and testing
n_train_batches = train_set_x.get_value(borrow=True).shape[0]
n_valid_batches = valid_set_x.get_value(borrow=True).shape[0]
n_test_batches = test_set_x.get_value(borrow=True).shape[0]
n_train_batches //= batch_size
n_valid_batches //= batch_size
n_test_batches //= batch_size

# allocate symbolic variables for the data
index = T.lscalar()  # index to a [mini]batch
```

```
# start-snippet-1
x = T.matrix('x')   # the data is presented as rasterized images
y = T.ivector('y')  # the labels are presented as 1D vector of
                    # [int] labels

######################
# BUILD ACTUAL MODEL #
######################
print('... building the model')

# Reshape matrix of rasterized images of shape (batch_size, 28 * 28)
# to a 4D tensor, compatible with our LeNetConvPoolLayer
# (28, 28) is the size of MNIST images.
layer0_input = x.reshape((batch_size, 3, 32, 32))

# Construct the first convolutional pooling layer:
# filtering reduces the image size to (28-5+1 , 28-5+1) = (24, 24)
# maxpooling reduces this further to (24/2, 24/2) = (12, 12)
# 4D output tensor is thus of shape (batch_size, nkerns[0], 12, 12)
layer0 = LeNetConvPoolLayer(
    rng,
    input=layer0_input,
    image_shape=(batch_size, 3, 32, 32),
    filter_shape=(nkerns[0], 3, 3, 3),
    poolsize=(2, 2)
)

# Construct the second convolutional pooling layer
# filtering reduces the image size to (12-5+1, 12-5+1) = (8, 8)
# maxpooling reduces this further to (8/2, 8/2) = (4, 4)
# 4D output tensor is thus of shape (batch_size, nkerns[1], 4, 4)
layer1 = LeNetConvPoolLayer(
    rng,
    input=layer0.output,
    image_shape=(batch_size, nkerns[0], 15, 15),
    filter_shape=(nkerns[1], nkerns[0], 3, 3),
    poolsize=(2, 2)
```

```
)

# the HiddenLayer being fully-connected, it operates on 2D matrices of
# shape (batch_size, num_pixels) (i.e matrix of rasterized images).
# This will generate a matrix of shape (batch_size, nkerns[1] * 4 * 4),
# or (500, 50 * 4 * 4) = (500, 800) with the default values.
layer2_input = layer1.output.flatten(2)

# construct a fully-connected sigmoidal layer
layer2 = HiddenLayer(
    rng,
    input=layer2_input,
    n_in=nkerns[1] * 6 * 6,
    n_out=4096,
    activation=T.tanh
)

# construct a fully-connected sigmoidal layer
layer3 = HiddenLayer(
    rng,
    input=layer2.output,
    n_in=4096,
    n_out=512,
    activation=T.tanh
)

# classify the values of the fully-connected sigmoidal layer
layer4 = LogisticRegression(input=layer3.output, n_in=512, n_out=10)

# the cost we minimize during training is the NLL of the model
cost = layer4.negative_log_likelihood(y)

# create a function to compute the mistakes that are made by the model
test_model = theano.function(
    [index],
    layer4.errors(y),
    givens={
```

```
        x: test_set_x[index * batch_size: (index + 1) * batch_size],
        y: test_set_y[index * batch_size: (index + 1) * batch_size]
    }
)

validate_model = theano.function(
    [index],
    layer4.errors(y),
    givens={
        x: valid_set_x[index * batch_size: (index + 1) * batch_size],
        y: valid_set_y[index * batch_size: (index + 1) * batch_size]
    }
)

# create a list of all model parameters to be fit by gradient descent
params = layer4.params + layer3.params + layer2.params + layer1.params + layer0.params

# create a list of gradients for all model parameters
grads = T.grad(cost, params)

# train_model is a function that updates the model parameters by
# SGD Since this model has many parameters, it would be tedious to
# manually create an update rule for each model parameter. We thus
# create the updates list by automatically looping over all
# (params[i], grads[i]) pairs.
updates = [
    (param_i, param_i - learning_rate * grad_i)
    for param_i, grad_i in zip(params, grads)
]

train_model = theano.function(
    [index],
    cost,
    updates=updates,
    givens={
        x: train_set_x[index * batch_size: (index + 1) * batch_size],
        y: train_set_y[index * batch_size: (index + 1) * batch_size]
```

```
        }
    )
    # end-snippet-1

    ################
    # TRAIN MODEL #
    ################
    train_nn(train_model, validate_model, test_model,
            n_train_batches, n_valid_batches, n_test_batches, n_epochs,
            verbose = True)



#Problem 3
#Implement a convolutional neural network to achieve at least 80% testing accuracy on CIFAR-dataset
def MY_lenet(learning_rate=0.1, n_epochs=200, nkerns=[20, 50], batch_size=500, L1_reg=0.00, L2_reg=0.0001):

    rng = numpy.random.RandomState(23455)

    ds_rate=None
    datasets = load_data(ds_rate=ds_rate,theano_shared=False)

    train_set_x, train_set_y = datasets[0]
    train_size = train_set_x.shape
    n_train = train_size[0]

    '''
    print '... Translating images'
    train_set_x_tran = np.empty(train_size)
    for i in range(n_train):
        img = (np.reshape(train_set_x[i],(3,32,32))).transpose(1,2,0)
        img_tran = translate_image(img)
        train_set_x_tran[i] = np.reshape(img_tran.transpose(2,0,1),(3*32*32))

    print '... Rotating images'
    train_set_x_rota = np.empty(train_size)
    for i in range(n_train):
```

```
    img = (np.reshape(train_set_x[i],(3,32,32))).transpose(1,2,0)
    img_tran = rotate_image(img)
    train_set_x_rota[i] = np.reshape(img_tran.transpose(2,0,1),(3*32*32))
'''
print '... Fliping images'
train_set_x_flip = np.empty(train_size)
for i in range(n_train):
    img = (np.reshape(train_set_x[i],(3,32,32))).transpose(1,2,0)
    img_tran = flip_image(img)
    train_set_x_flip[i] = np.reshape(img_tran.transpose(2,0,1),(3*32*32))
'''
print '... Ennoising images'
train_set_x_nois = np.empty(train_size)
for i in range(n_train):
    img = (np.reshape(train_set_x[i],(3,32,32))).transpose(1,2,0)
    img_tran = noise_injection(img)
    train_set_x_aug[i] = np.reshape(img_tran.transpose(2,0,1),(3*32*32))
'''


train_set_x = np.concatenate((
    train_set_x,
    #train_set_x_tran,
    #train_set_x_rota,
    train_set_x_flip),
    axis=0
)
train_set_y = np.concatenate(
    (train_set_y,
    #train_set_y,
    #train_set_y,
    train_set_y),
    axis=0
)

datasets[0] = [train_set_x, train_set_y]

train_set_x, train_set_y = shared_dataset(datasets[0])
```

```
valid_set_x, valid_set_y = shared_dataset(datasets[1])
test_set_x, test_set_y = shared_dataset(datasets[2])

# compute number of minibatches for training, validation and testing
n_train_batches = train_set_x.get_value(borrow=True).shape[0]
n_valid_batches = valid_set_x.get_value(borrow=True).shape[0]
n_test_batches = test_set_x.get_value(borrow=True).shape[0]
n_train_batches //= batch_size
n_valid_batches //= batch_size
n_test_batches //= batch_size

# allocate symbolic variables for the data
index = T.lscalar()  # index to a [mini]batch

# start-snippet-1
x = T.matrix('x')   # the data is presented as rasterized images
y = T.ivector('y')  # the labels are presented as 1D vector of
                    # [int] labels
training_enabled = T.iscalar('training_enabled') # pseudo boolean for switching between training and prediction

######################
# BUILD ACTUAL MODEL #
######################
print('... building the model')

# Reshape matrix of rasterized images of shape (batch_size, 28 * 28)
# to a 4D tensor, compatible with our LeNetConvPoolLayer
# (28, 28) is the size of MNIST images.
layer0_input = x.reshape((batch_size, 3, 32, 32))

# Construct the first convolutional pooling layer:
# filtering reduces the image size to (28-5+1 , 28-5+1) = (24, 24)
# maxpooling reduces this further to (24/2, 24/2) = (12, 12)
# 4D output tensor is thus of shape (batch_size, nkerns[0], 12, 12)
layer0 = LeNetConvPoolLayer(
    rng,
    input=layer0_input,
```

```
    image_shape=(batch_size, 3, 32, 32),
    filter_shape=(nkerns[0], 3, 3, 3),
    poolsize=(2, 2)
)
#print 'layer0.output.shape ='
#print layer0.output.shape.eval({x: np.random.rand(2,2).astype(dtype=theano.config.floatX)})

layerbn = BatchNormalization(
    input_shape=(batch_size, nkerns[0], 15, 15),
    mode=1,
    momentum=0.9
)
layerbn_output = layerbn.get_result(layer0.output)
#print 'layerbn_output.shape ='
#print layerbn_output.shape.eval({x: np.random.rand(2,2).astype(dtype=theano.config.floatX)})

# Construct the second convolutional pooling layer
# filtering reduces the image size to (12-5+1, 12-5+1) = (8, 8)
# maxpooling reduces this further to (8/2, 8/2) = (4, 4)
# 4D output tensor is thus of shape (batch_size, nkerns[1], 4, 4)
layer1 = LeNetConvPoolLayer(
    rng,
    input=layerbn_output,
    image_shape=(batch_size, nkerns[0], 15, 15),
    filter_shape=(nkerns[1], nkerns[0], 3, 3),
    poolsize=(2, 2)
)

# the HiddenLayer being fully-connected, it operates on 2D matrices of
# shape (batch_size, num_pixels) (i.e matrix of rasterized images).
# This will generate a matrix of shape (batch_size, nkerns[1] * 4 * 4),
# or (500, 50 * 4 * 4) = (500, 800) with the default values.
layer2_input = layer1.output.flatten(2)

# construct a fully-connected sigmoidal layer
layer2 = DropoutHiddenLayer(
    rng,
```

```
    is_train=training_enabled,
    input=layer2_input,
    n_in=nkerns[1] * 6 * 6,
    n_out=4096,
    activation=T.nnet.relu
)

# construct a fully-connected sigmoidal layer
layer3 = DropoutHiddenLayer(
    rng,
    is_train=training_enabled,
    input=layer2.output,
    n_in=4096,
    n_out=2048,
    activation=T.nnet.relu
)

# construct a fully-connected sigmoidal layer
layer4 = DropoutHiddenLayer(
    rng,
    is_train=training_enabled,
    input=layer3.output,
    n_in=2048,
    n_out=1024,
    activation=T.nnet.relu
)

# construct a fully-connected sigmoidal layer
layer5 = DropoutHiddenLayer(
    rng,
    is_train=training_enabled,
    input=layer4.output,
    n_in=1024,
    n_out=512,
    activation=T.nnet.relu
)
```

```
# classify the values of the fully-connected sigmoidal layer
layer6 = LogisticRegression(input=layer5.output, n_in=512, n_out=10)

# L1 norm ; one regularization option is to enforce L1 norm to
# be small
L1 = (
    abs(layer2.W).sum()
    + abs(layer3.W).sum()
    + abs(layer4.W).sum()
    + abs(layer5.W).sum()
    + abs(layer6.W).sum()
)

# square of L2 norm ; one regularization option is to enforce
# square of L2 norm to be small
L2_sqr = (
    (layer2.W ** 2).sum()
    + (layer3.W ** 2).sum()
    + (layer4.W ** 2).sum()
    + (layer5.W ** 2).sum()
    + (layer6.W ** 2).sum()
)

# the cost we minimize during training is the negative log likelihood of
# the model plus the regularization terms (L1 and L2); cost is expressed
# here symbolically
cost = (
    layer6.negative_log_likelihood(y)
    + L1_reg * L1
    + L2_reg * L2_sqr
)

# create a function to compute the mistakes that are made by the model
test_model = theano.function(
    [index],
    layer6.errors(y),
    givens={
```

```
            x: test_set_x[index * batch_size: (index + 1) * batch_size],
            y: test_set_y[index * batch_size: (index + 1) * batch_size],
            training_enabled: numpy.cast['int32'](0)
        }
    )

    validate_model = theano.function(
        [index],
        layer6.errors(y),
        givens={
            x: valid_set_x[index * batch_size: (index + 1) * batch_size],
            y: valid_set_y[index * batch_size: (index + 1) * batch_size],
            training_enabled: numpy.cast['int32'](0)
        }
    )

    # create a list of all model parameters to be fit by gradient descent
    params = layer6.params + layer5.params + layer4.params + layer3.params + layer2.params + layer1.params +
layer0.params

    '''
    # create a list of gradients for all model parameters
    grads = T.grad(cost, params)

    # train_model is a function that updates the model parameters by
    # SGD Since this model has many parameters, it would be tedious to
    # manually create an update rule for each model parameter. We thus
    # create the updates list by automatically looping over all
    # (params[i], grads[i]) pairs.
    updates = [
        (param_i, param_i - learning_rate * grad_i)
        for param_i, grad_i in zip(params, grads)
    ]
    '''

    # compute the gradient of cost with respect to theta (sotred in params)
    # the resulting gradients will be stored in a list gparams
```

```
#gparams = [T.grad(cost, param) for param in params]

# specify how to update the parameters of the model as a list of
# (variable, update expression) pairs
momentum =theano.shared(numpy.cast[theano.config.floatX](0.5), name='momentum')
updates = []
for param in  params:
    param_update = theano.shared(param.get_value()*numpy.cast[theano.config.floatX](0.))
    updates.append((param, param - learning_rate*param_update))
    updates.append(
        (param_update,
         momentum*param_update + (numpy.cast[theano.config.floatX](1.) - momentum)*T.grad(cost, param))
    )


train_model = theano.function(
    [index],
    cost,
    updates=updates,
    givens={
        x: train_set_x[index * batch_size: (index + 1) * batch_size],
        y: train_set_y[index * batch_size: (index + 1) * batch_size],
        training_enabled: numpy.cast['int32'](1)
    }
)
# end-snippet-1

################
# TRAIN MODEL #
################
train_nn(train_model, validate_model, test_model,
        n_train_batches, n_valid_batches, n_test_batches, n_epochs,
        verbose = True)
```

```python
#Problem4
#Implement the convolutional neural network depicted in problem4
def MY_CNN():
    # The code of this part is written in hw3p4.py
    pass
```