Advanced Machine Learning Methods
Project CNN

Thomas Astad Sve thomaasv@stud.ntnu.no
Computer Science, Artificial Intelligence
Norwegian University of Science & Technology
University & Professional Studies ax004206@acsmail.ucsd.edu
University of California, San Diego

Abstract

In this work, I use Convolutional Neural Networks trained on a GPU for classifying images in a tiny imagenet dataset. In particulay I have used the python library lasagne [1] to built on the googlenet [2] architecture.

1. Introduction

I use Convolutional Neural Networks for image classifications on the tiny ImageNet dataset, a smaller version of the ImageNet [4] challenge dataset.

2. Method

2.1 Model

The model used to build the classifier is a model based on the googlenet architecture. I used the library lasagne [1] which is built on theano for deep learning. I also found the recipie for the model online [3]. The network has 22 layers where 9 of the layers is inception layers with dimensionality reductions.

2.2 Dataset

The tiny ImageNet is a smaller version of the ImageNet dataset. The tiny versions consist of total 200 classes

3. Experiment

4. Results

In order to test my program to see if it would run, I first classified a smaller version of the dataset with only 20 classes. This took around 3 hours running 500 epochs. After having confirmed everything worked with a low number of classes, I wanted to test if I could run for the whole dataset with 200 classes. Starting to train the classes with 200 classes took me around 380seconds per epoch, and I realized I would have to run for many days to complete it. I therefore decided to reduce down to 100 classes with epoch of 250. With 100 classes it now took me 190 seconds per epoch, and running 250 epochs it took me around 14 hours to complete it. I could have let the network run at 200 classes and rather reduce the layers to make the network less deep, or significantly reduce number of epochs. I chose not to do this to keep the depth of the network.

Table 1: Shows results for running classification on Googlenet

# classes	# training	# validation	# testing	# epoch	test-loss	Accuracy
20	6903	1480	1480	500	10.9	39.30
100	39340	4928	9835	250		

On epoch 50 it was actually lower than epoch 30. Looking closer at the runs, it shows that on epoch 49 and 51 it was around 28 %, meaning it hasnt really changed since epoch 30, it just dropped a little on the actual round 50.

Table 2: Shows results for validation accuracy for Googlenet on tiny ImageNet with 100 classes on some epochs

Epoch	Val accuracy
1	1.51 %
5	7.29 %
10	16.53 %
20	25.89 %
30	27.38 %
50	22.64 %
100	
150	
200	
250	

5. Conclusion

References

[1] Sander Dieleman, Lasagne is a lightweight library to build and train neural networks in Theano. `https://github.com/Lasagne/Lasagne`

[2] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich. Going Deeper with Convolutions, 2014. `http://arxiv.org/abs/1409.4842`

[3] `https://github.com/Lasagne/Recipes/blob/master/modelzoo/googlenet.py`

[4] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. FeiFei. ImageNet: A large-scale hierarchical image database. In CVPR, 2009

Code

main.py

```
import numpy as np
import lasagne
import theano
import theano.tensor as T
import vgg16
import googlenet
import pickle
import time

from preprocess import load_dataset, generate_url_zip

def iterate_minibatches(inputs, targets, batchsize, shuffle=False):
    assert len(inputs) == len(targets)
    if shuffle:
        indices = np.arange(len(inputs))
        np.random.shuffle(indices)
    for start_idx in range(0, len(inputs) - batchsize + 1, batchsize):
        if shuffle:
            excerpt = indices[start_idx:start_idx + batchsize]
        else:
            excerpt = slice(start_idx, start_idx + batchsize)
        yield inputs[excerpt], targets[excerpt]

def train_network(num_epochs, X_train, y_train, X_val, y_val, train_fn, val_fn):
    # We iterate over epochs:
    for epoch in range(num_epochs):
        # In each epoch, we do a full pass over the training data:
        train_err = 0
        train_batches = 0
        start_time = time.time()
        for batch in iterate_minibatches(X_train, y_train, 500, shuffle=True):
            inputs, targets = batch
            train_err += train_fn(inputs, targets)
            train_batches += 1

        # And a full pass over the validation data:
        val_err = 0
        val_acc = 0
        val_batches = 0
        for batch in iterate_minibatches(X_val, y_val, 500, shuffle=False):
            inputs, targets = batch
            err, acc = val_fn(inputs, targets)
            val_err += err
            val_acc += acc
            val_batches += 1

        # Then we print the results for this epoch:
        print("Epoch {} of {} took {:.3f}s".format(
            epoch + 1, num_epochs, time.time() - start_time))
        print("  training loss:\t\t{:.6f}".format(train_err / train_batches))
        print("  validation loss:\t\t{:.6f}".format(val_err / val_batches))
        print("  validation accuracy:\t\t{:.2f} %".format(
            val_acc / val_batches * 100))


def test_network(X_test, y_test, val_fn):
    test_err = 0
    test_acc = 0
    test_batches = 0
    for batch in iterate_minibatches(X_test, y_test, 500, shuffle=False):
        inputs, targets = batch
        err, acc = val_fn(inputs, targets)
        test_err += err
        test_acc += acc
        test_batches += 1
    print("Final results:")
    print("  test loss:\t\t\t{:.6f}".format(test_err / test_batches))
    print("  test accuracy:\t\t{:.2f} %".format(
        test_acc / test_batches * 100))

def build_parameter_update(network, loss):
    # create parameter update expressions
    params = lasagne.layers.get_all_params(network, trainable=True)
    updates = lasagne.updates.nesterov_momentum(loss, params, learning_rate=0.01,
                                                momentum=0.9)
    return updates


def build_loss(network, target_var):
    # Create a loss expression for training, i.e., a scalar objective we want
    # to minimize (for our multi-class problem, it is the cross-entropy loss):
    prediction = lasagne.layers.get_output(network)
    loss = lasagne.objectives.categorical_crossentropy(prediction, target_var)
    loss = loss.mean()
    return loss

def build_test_loss(network, target_var):
    # Create a loss expression for validation/testing. The crucial difference
    # here is that we do a deterministic forward pass through the network,
    # disabling dropout layers.
    test_prediction = lasagne.layers.get_output(network, deterministic=True)
    test_loss = lasagne.objectives.categorical_crossentropy(test_prediction,
                                                            target_var)
    test_loss = test_loss.mean()
    # As a bonus, also create an expression for the classification accuracy:
    test_acc = T.mean(T.eq(T.argmax(test_prediction, axis=1), target_var),
                      dtype=theano.config.floatX)

    return test_loss, test_acc

def main(num_epochs=250):
    print("Loading data...")
    X_train, y_train, X_val, y_val, X_test, y_test = generate_url_zip()

    print "X_train: ", X_train.shape, " y_train: ", y_train.shape
    print "X_val: ", X_val.shape, " y_val: ", y_val.shape

    input_var = T.tensor4('inputs')
    target_var = T.ivector('targets')

    print("Building network...")
    #network = vgg16.build_model(input_var)
    network = googlenet.build_model(input_var)
    # Create a loss expression for training
    loss = build_loss(network, target_var)

    # create parameter update expressions
    updates = build_parameter_update(network, loss)

    # Create a loss expression for validation/testing.
    test_loss, test_acc = build_test_loss(network, target_var)

    # Compile a function performing a training step on a mini-batch (by giving
    # the updates dictionary) and returning the corresponding training loss:
    print("Setting training function...")
    train_fn = theano.function([input_var, target_var], loss, updates=updates)

    # Compile a second function computing the validation loss and accuracy:
    print("Setting validation function for loss and accuracy...")
    val_fn = theano.function([input_var, target_var], [test_loss, test_acc])

    # Finally, launch the training loop.
    print("Starting training...")
    train_network(num_epochs, X_train, y_train, X_val, y_val, train_fn, val_fn)

    # After training, we compute and print the test error:
    print("Starting testing...")
    test_network(X_test, y_test, val_fn)

    # Save network
    np.savez('trained_alexnet_200.npz', *lasagne.layers.get_all_param_values(network))

    # And load them again later on like this:
    # with np.load('model.npz') as f:
    #     param_values = [f['arr_%d' % i] for i in range(len(f.files))]
    # lasagne.layers.set_all_param_values(network, param_values)


if __name__ == "__main__":
    main()
```

googlenet.py

```
# BLVC Googlenet, model from the paper:
# "Going Deeper with Convolutions"
# Original source:
# https://github.com/BVLC/caffe/tree/master/models/bvlc_googlenet
# License: unrestricted use

# Download pretrained weights from:
# https://s3.amazonaws.com/lasagne/recipes/pretrained/imagenet/blvc_googlenet.pkl

from lasagne.layers import InputLayer
from lasagne.layers import DenseLayer
from lasagne.layers import ConcatLayer
from lasagne.layers import NonlinearityLayer
from lasagne.layers import GlobalPoolLayer
from lasagne.layers.dnn import Conv2DDNNLayer as ConvLayer
from lasagne.layers.dnn import MaxPool2DDNNLayer as PoolLayerDNN
from lasagne.layers import MaxPool2DLayer as PoolLayer
```

```python
from lasagne.layers import LocalResponseNormalization2DLayer as LRNLayer
from lasagne.nonlinearities import softmax, linear


def build_inception_module(name, input_layer, nfilters):
    # nfilters: (pool_proj, 1x1, 3x3_reduce, 3x3, 5x5_reduce, 5x5)
    net = {}
    net['pool'] = PoolLayerDNN(input_layer, pool_size=3, stride=1, pad=1)
    net['pool_proj'] = ConvLayer(net['pool'], nfilters[0], 1, flip_filters=False)

    net['1x1'] = ConvLayer(input_layer, nfilters[1], 1, flip_filters=False)

    net['3x3_reduce'] = ConvLayer(input_layer, nfilters[2], 1, flip_filters=False)
    net['3x3'] = ConvLayer(net['3x3_reduce'], nfilters[3], 3, pad=1, flip_filters=False)

    net['5x5_reduce'] = ConvLayer(input_layer, nfilters[4], 1, flip_filters=False)
    net['5x5'] = ConvLayer(net['5x5_reduce'], nfilters[5], 5, pad=2, flip_filters=False)

    net['output'] = ConcatLayer([
        net['1x1'],
        net['3x3'],
        net['5x5'],
        net['pool_proj'],
        ])

    return {'{}/{}'.format(name, k): v for k, v in net.items()}


def build_model(input_var = None):
    net = {}
    net['input'] = InputLayer((None, 3, None, None), input_var = input_var)
    net['conv1/7x7_s2'] = ConvLayer(net['input'], 64, 7, stride=2, pad=3, flip_filters=False)
    net['pool1/3x3_s2'] = PoolLayer(net['conv1/7x7_s2'], pool_size=3, stride=2, ignore_border=False)
    net['pool1/norm1'] = LRNLayer(net['pool1/3x3_s2'], alpha=0.00002, k=1)
    net['conv2/3x3_reduce'] = ConvLayer(net['pool1/norm1'], 64, 1, flip_filters=False)
    net['conv2/3x3'] = ConvLayer(net['conv2/3x3_reduce'], 192, 3, pad=1, flip_filters=False)
    net['conv2/norm2'] = LRNLayer(net['conv2/3x3'], alpha=0.00002, k=1)
    net['pool2/3x3_s2'] = PoolLayer(net['conv2/norm2'], pool_size=3, stride=2, ignore_border=False)

    net.update(build_inception_module('inception_3a',
                                      net['pool2/3x3_s2'],
                                      [32, 64, 96, 128, 16, 32]))
    net.update(build_inception_module('inception_3b',
                                      net['inception_3a/output'],
                                      [64, 128, 128, 192, 32, 96]))
    net['pool3/3x3_s2'] = PoolLayer(net['inception_3b/output'], pool_size=3, stride=2, ignore_border=False)

    net.update(build_inception_module('inception_4a',
                                      net['pool3/3x3_s2'],
                                      [64, 192, 96, 208, 16, 48]))
    net.update(build_inception_module('inception_4b',
                                      net['inception_4a/output'],
                                      [64, 160, 112, 224, 24, 64]))
    net.update(build_inception_module('inception_4c',
                                      net['inception_4b/output'],
                                      [64, 128, 128, 256, 24, 64]))
    net.update(build_inception_module('inception_4d',
                                      net['inception_4c/output'],
                                      [64, 112, 144, 288, 32, 64]))
    net.update(build_inception_module('inception_4e',
                                      net['inception_4d/output'],
                                      [128, 256, 160, 320, 32, 128]))
    net['pool4/3x3_s2'] = PoolLayer(net['inception_4e/output'], pool_size=3, stride=2, ignore_border=False)

    net.update(build_inception_module('inception_5a',
                                      net['pool4/3x3_s2'],
                                      [128, 256, 160, 320, 32, 128]))
    net.update(build_inception_module('inception_5b',
                                      net['inception_5a/output'],
                                      [128, 384, 192, 384, 48, 128]))

    net['pool5/7x7_s1'] = GlobalPoolLayer(net['inception_5b/output'])
    net['loss3/classifier'] = DenseLayer(net['pool5/7x7_s1'],
                                         num_units=1000,
                                         nonlinearity=linear)
    net['prob'] = NonlinearityLayer(net['loss3/classifier'],
                                    nonlinearity=softmax)
    return net['prob']
```

preprocess_zip.py

```python
import numpy as np
import h5py
import zipfile
from random import shuffle
from math import floor
from PIL import Image
from StringIO import StringIO

def split_dataset(X, y, test_size = 0.2, val = False):
    print len(X), len(y)
    data = zip(X, y)
    shuffle(data)
    X, y = zip(*data)

    if val:
        split_point = int(floor(len(X)*(1 - test_size * 2)))
        X_train, y_train = X[:split_point], y[:split_point]
        X_test_val, y_test_val = X[split_point:], y[split_point:]
        split_point = int(floor(len(X_test_val)*0.5))
        X_test, y_test, X_val, y_val = X_test_val[:split_point], y_test_val[:split_point], \
                                       X_test_val[split_point:], y_test_val[split_point:]

        return np.array(X_train), np.array(y_train), np.array(X_test)\
            , np.array(y_test), np.array(X_val), np.array(y_val)
    else:
        split_point = int(floor(len(X)*(1 - test_size)))
        return np.array(X[:split_point]), np.array(y[:split_point]), \
                                                   np.array(X[split_point:]), np.array(y[split_point:])


def load_zip_training_set(path, wnids, archive):
    X = []
    y = []
    i = 0
    for class_id in wnids:
        bbox_file = path + class_id + "/" + class_id + "_boxes.txt"
        for line  in archive.open(bbox_file):
            words = line.split()
            img = archive.read(path + class_id + "/images/" + words[0])
            img = Image.open(StringIO(img))
            image = np.array(img)
            if image.ndim == 3:
                image = np.rollaxis(image, 2)
                X.append(image) # Append image to dataset
                y.append(i)
        i = i + 1

    return np.array(X, dtype=np.uint8), np.array(y, dtype=np.uint8)

def find_label(wnids, wnid):
    i = 0
    for line in wnids:
        if line == wnid:
            return i
        i = i + 1
    return None

def load_zip_val_set(path, wnids, archive):
    X = []
    y = []
    val_annotations = path + "val_annotations.txt"
    for line in archive.open(val_annotations):
        words = line.split()
        img = archive.read(path + "images/" + words[0])
        img = Image.open(StringIO(img))
        image = np.array(img)
        label = find_label(wnids, words[1])
        if image.ndim == 3 and label != None:
            image = np.rollaxis(image, 2)
            X.append(image) # Append image to dataset
            y.append(find_label(wnids, words[1]))

    return np.array(X, dtype=np.uint8), np.array(y, dtype=np.uint8)

def generate_url_zip():
    zip_url = "tiny-imagenet-200.zip"
    train_path = "tiny-imagenet-200/train/"
    val_path = "tiny-imagenet-200/val/"
    test_path = "tiny-imagenet-200/test/"
    wnid_file = "tiny-imagenet-200/wnids.txt"


    print "Reading from zip..."
    archive = zipfile.ZipFile(zip_url, 'r') # Open the archive
    wnids = [line.strip() for line in archive.open(wnid_file)] # Load list over classes
    wnids = wnids[:100] # Load only the 100 first classes

    print "Loading training set..."
    X, y = load_zip_training_set(train_path, wnids, archive)

    print "Loading validation set.."
    X_val, y_val = load_zip_val_set(val_path, wnids, archive)
    X_train, y_train, X_test, y_test =  split_dataset(X, y, test_size = 0.2)
    print(X_train.shape, y_train.shape, X_val.shape, y_val.shape, X_test.shape, y_test.shape)

    return X_train.astype(np.uint8), y_train.astype(np.uint8), X_val.astype(np.uint8), y_val.astype(np.uint8), X_test.a

if __name__ == "__main__":
    generate_url_zip()
```

classify a image

```
import numpy as np
import matplotlib.pyplot as plt
import lasagne
import googlenet

def load_pickle_googlenet():
    import pickle
    model = pickle.load(open('vgg_cnn_s.pkl'))
    CLASSES = model['synset words']
    MEAN_IMAGE = model['mean image']
    lasagne.layers.set_all_param_values(output_layer, model['values'])

    return output_layer

def load_network():
    network = googlenet.build_model()
    with np.load('trained_alexnet_200.npz') as f:
        param_values = [f['arr_%d' % i] for i in range(len(f.files))]
    lasagne.layers.set_all_param_values(network, param_values)

    return network

def load_test_images(image_urls):
    images = []
    images_raw = []

    for url in image_urls:
        img = Image.open(url)
        image = np.array(img)
        images_raw.append(image)
        image = np.rollaxis(image)
        images.append(image)

    return images, images_raw

def random_test_images(image_urls, num_samples = 5):
    np.random.seed(23)
    image_urls = image_urls[:num_samples]
    images, images_raw = load_test_images(image_urls)

    return images, images_raw

def load_classes(wnid_file):
    return [line.strip() for line in open(wnid_file)]

def print_predictions(images, images_raw, network, classes):
    for i in range(len(images)):
            prob = np.array(lasagne.layers.get_output(network, images[i], deterministic=True).eval())
            top5 = np.argsort(prob[0])[-1:-6:-1]

            plt.figure()
            plt.imread(images_raw[i].astype('uint8'))
            plt.axis('off')
            for n, label in enumerate(top5):
                plt.text(250, 70 + n * 20, '{}. {}'.format(n+1, classes[label]), fontsize=14)

            plt.save("predicted_" + str(i) + ".JPEG")
            print "Saved plot: predicted_" + str(i) + ".JPEG"
            i = i + 1

def main():
    wnid_file = "/home/thomas/data/dataset/tiny-imagenet-200/wnids.txt"
    test_path = "tiny-imagenet-200/test/"

    network = load_network()
    #network = load_pickle_googlenet()
    images, images_raw = random_test_images()
    classes = load_classes(wnid_file)
    print_predictions(images, images_raw, network, classes)

if __name__=="__main__":
    main()
```