Advanced Machine Learning Methods
Project CNN GoogleNet on tiny ImageNet

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

I trained a Convolutional Neural Network on GPU for classifying images using the tiny ImageNet data set. The model is built after googlenet's [4] architecture using the python library lasagne [3], which is built on top of theano [2]. The CNN network consist of 22 layers and achived a total accuracy of 30 % when having 100 different classes. A reduced version with 20 different classes got a total of 37 % accuracy on the test-set. More in depth analysis would have been considered if I had more time available to test several CNN networks. I also cut the original data set of 200 to 100 classes to reduce training time as I calculated it would take me up to three days to train with the 200 classes.

1. Introduction

A common problem in machine learning is to make effective representation of complex inputs such as image and video. Over the past years deep learning algorithms have proven to perform well on such complex problems. In particular a version of deep learning called Convolutional Neural Network (CNN) have had great results with classifying images. CNN architecture make the assumption that the inputs are images, which allows us to encode certain properties into the architecture. The first successful applicants of convolutional networks were developed by Yann LeCun in 1990's. One which is known as the LeNet [7] architecture. Other known arcitectures are AlexNet [8], GoogLeNet [4] and VGGNet [9]. I have decided for my project to build an Convolutional Neural Networks based on the GoogLeNet arcitecture using the python library lasagne [3]. I will use the network for image classifications on the tiny ImageNet dataset, a dataset which looks like a tiny version of the ImageNet [6] 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 [3] which is built on theano for deep learning. I also found the recipie for the model online [5] which I cloned and used in my implementations. The network has 22 layers where 9 of the layers is inception layers with dimensionality reductions. A more in depth explanation of all the 22 layers in the googlenet architecture I highly suggest reading their paper ”Going deeper with convolutions” [4] or watch a interesting youtube video that represents an image on each layer [1].

2.2 Data Set

Tiny ImageNet is a dataset with 120,000 labeled images into 200 different categories. The dataset is similar to the ImageNet used in the ILSVRC benchmark, known as the ImageNet challenge. The images in the tiny dataset have lower resolution and it's an smaller dataset in total. Each of the 200 categories consists of 500 training samples with a 64x64 resolution. The raw RGB pixel-values of these images are extracted and fed to the CNN network. In the dataset there were textfiles that listed x-axis and y-axis values that marked boxes on the images. These boxes I did not use in my implementation, and this may have led to a smaller precision. In my implementation I reduced the numbers of classes to 100 to reduce training time, as I did not have time to wait a 2-3 days for the network to train a network with all the 200 classes.

Figure 1: TSNE Embedding of Tiny Imagenet CNN Codes

2.3 Lasagne / Theano

Lasagne is a lightweight library to build and train neural networks in Theano. The simplicity of Lasagne makes it great to use in a project like this where it's easy to build complex and deep models relatively easily. Lasagne is built on top of Theano which uses CUDA technologies to boost the performance by using GPU when training the neural network. In particular I have used the NVIDIA cuDNN techonology to speed up my training of the convolutional network.

3. Experiment

In order to test the network I started and classified a smaller version of the dataset with only 20 classes. This took me around 3 hours with 500 epochs. I saved the results and wanted to go bigger with the full dataset on all 200 classes. After seeing that one epoch took me 380 seconds, I decided to scale the dataset down to 100 classes. After scaling it down, it now took me 190 seconds per epoch, and I also reduced the numbers of epoch to 250 so it would finish on around 13 hours. I could have chosen a different architecture with fewer layers and trained on the full dataset, but I wanted to see how well the results would be training on a deep CNN with the GoogLeNet architecture.

I used 20 % of the training folder as test-set, meaning 20 % of the training data was reduced (so less than 500 samples in each class now). For validation I sorted out the images in the validation folder in the dataset and picked out the images that was labeled with one of the classes I wanted to train. To reduce the number of samples in the training set to have it in a test-set could potentially effect the results in a large scale, considering there only were originally 500 samples in each class which is very low compared to other datasets for image classification. For example, the CIFAR-10 data set contains more than 5,000 images for each of the ten classes compared to the 500 for each of the 200 tiny ImageNet classes.

After the 13 hours it took to train the network I saved all the parameters and results so I could reload the network later to do predictions on images and play with the already trained network. This way I wouldn't have to wait another 13 hours if I wanted to test something new on the trained network, which could have been even worse if I ended up training with more classes. I also loaded the network and predicted 5 random images from the test-set and printed the top-5 results. These results you can see in the end of the section.

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	9.95	29.31

From the table 1 above, it shows that I got an 39 % accuracy for the dataset with 20 classes and a 29 % accuracy for the 100 classes dataset. This result is rather poor. This may be due to the low number of samples in each class, that I pointed out earlier or the low resolution on the actual images. As I never figured out how to use the boxes I had at my disposal in the dataset may also be a reason where I did not preprocess the images properly before starting training. I could maybe have used the boxes to add a extra feature to the image. If I figured out how to use the boxes it would take me another 13 hours to train the network, which I did not have the time to do.

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

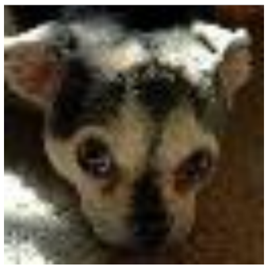
Epoch	Val accuracy
1	2.24 %
5	7.29 %
10	18.4 %
20	26.87 %
30	27.38 %
50	28.48 %
100	30.00 %
150	29.20 %
200	28.93 %
250	29.69 %

As we can see on the validation accuracy after numbers of epochs (table 2), it stopped improved very little after 20 epochs, making it stuck at around 28 % accuracy. This could have been improved by having many more samples pr classes, as it may be overfitting on those 500 images for each class. In highsight it seems like it was a waste training 250 epochs as it really didn't improve that much on the last 200. I do also think a even deeper network could help make the network improve more after a numerous of epochs.

3.1 Some Top-5 classification examples

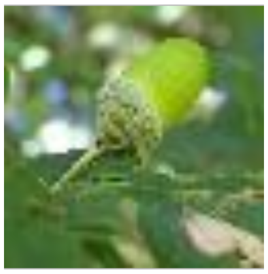
I took 5 random images from the test set and printed out top 4 predictions on each of these which you can see on next page.

1. Tarantula
2. German shepherd, German shepherd dog, German police dog, alsatian
3. Candle, taper, wax light
4. Labrador retriever
5. Egyptian cat



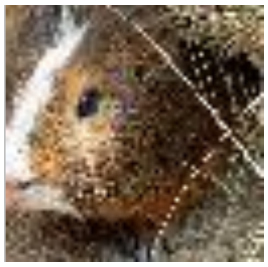
---

1. Lemon
2. Lakeside, lakeshore
3. Sulphur butterfly, sulfur butterfly
4. Spider web, spider's web
5. Snail



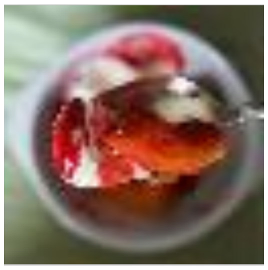
---

1. Snail
2. Arabian camel, dromedary, Camelus dromedarius
3. Cockroach, roach
4. Bullfrog, Rana catesbeiana
5. Koala, koala bear, kangaroo bear, native bear, Phascolarctos cinereus



---

1. Pretzel
2. Tractor
3. Monarch, monarch butterfly, milkweed butterfly, Danaus plexippus
4. Candle, taper, wax light
5. Lemon



It is interesting to see what is being predicted on the images, but it's difficult to understand why it predicts as it does. The samples from the previous page shows how uncertain and inprecise the trained network is. Still, on some of the images I am myself unsure on what exactly is shown as the images have very low resolution. For example on the first image, I feel confident it is a puppy or a dog but do not know what breed of dog it is as I dont know everything about dog breeds.

4. Conclusion

In this work, I used Convolutional Neural Network for performing image classification on the tiny ImageNet dataset. By building my network after the GoogLeNet model with 22 layers I was expecting to get a similar results as others using the same architecture

A accuracy of 30 % on 100 classes on the tiny ImageNet was much lower than I expected, considering the caffe model [11] had top-1 accuracy of 68.7%, tho not on the same data set. One idea could be that I did not preprocess my images correctly, not using the boxes given to me. Or that the data set I used did not have enough samples per classes.

Given my limited time and resources for the project, there is significant room for improvements. For example changing the model for a deeper neural network, or having images with higher resolution could improve the accuracy. In the trained network I have now I did not use the boxes at my disposal, and with more time I would include these boxes in the preprocessing and retrained the network.

The complexity of a Convolutional Neural Network is clearly one of the major bottlenecks, making it not easy to retrain a trained network over the full dataset without having to wait hours if not days for it to finish. So therefore it is hard to come with a solution with what to do to improve my network.

References

[1] DeepDream through all the layers of GoogleNet `https://www.youtube.com/watch?v=w5U7EL72ngI`

[2] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley and Y. Bengio. Theano: A CPU and GPU Math Expression Compiler. Proceedings of the Python for Scientific Computing Conference (SciPy) 2010. June 30 - July 3, Austin, TX

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

[4] 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`

[5] Lasagne Recipe for the GoogLeNet-model `https://github.com/Lasagne/Recipes/blob/master/modelzoo/googlenet.py`

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

[7] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. Proceedings of the IEEE, november 1998 `http://yann.lecun.com/exdb/publis/pdf/lecun-98.pdf`

[8] Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks, 2012. `http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks`

[9] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich. Going Deeper with Convolutions, 2014. `http://arxiv.org/abs/1409.4842`

[10] Andrej Karpathy, Convolutional Neural Networks (CNNs / ConvNets) `http://cs231n.github.io/convolutional-networks/`

[11] Caffe implementation of the GoogleNet model. `https://github.com/BVLC/caffe/tree/master/models/bvlc_googlenet`

Code

main.py

```python
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, network):
    save_iterals = {0, 49, 99, 199, 249}
    results = []
    # 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()

        print("   pass over training data...")
        for batch in iterate_minibatches(X_train, y_train, 500, shuffle=True):
            inputs, targets = batch
            train_err += train_fn(inputs, targets)
            train_batches += 1

        print("   pass over validation data...")
        # 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))

        results.append(val_acc / val_batches * 100)

        if epoch in save_iterals: # Store the network while training
            np.savez("epoch_googlenet_100_" + str(epoch + 1) +".npz", *lasagne.layers.get_all_param_values(network))


    np.savez('googlenet_epochs.npz', results=results)



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, network)

    # 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_googlenet_100.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
```

```python
# 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
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

```
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 lasagne
from lasagne.utils import floatX
import googlenet
import zipfile
from random import shuffle
from math import floor
from preprocess_zip import load_zip_val_set
from PIL import Image
from StringIO import StringIO


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_googlenet_100.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)
        rawim = np.copy(img).astype('uint8')

        images_raw.append(rawim)

        image = np.rollaxis(image)
        images.append(floatX(image[np.newaxis]))


    return images, images_raw

def load_images(path, wnids, archive):
    X = []
    X_raw = []
    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)
        if image.ndim == 3:
            X_raw.append(np.copy(image).astype('uint8'))
            image = np.rollaxis(image, 2)
            X.append(floatX(image[np.newaxis])) # Append image to dataset

    return np.array(X), np.array(X_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_name(wnids, archive):
    words = "tiny-imagenet-200/words.txt"
    classes_words = {}
    for line in archive.open(words):
        words = line.split()
        classes_words[words[0]] = words[1]

    return classes_words

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

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

    np.savez("predictions.npz", top=top5, images=images, images_raw=images_raw, classes=classes, classes_words=classes_w
    print "Predictions saved as predictions.npz"


def main():
    #with np.load('googlenet_epochs.npz') as data:
    #    results = data['results']

    #epoch_print = [1, 5, 10, 20, 30, 50, 100, 150, 200, 250]
    #for i in epoch_print:
    #    print "Results, epoch " + str(i) + ": " + str(results[i - 1])


    zip_url = "tiny-imagenet-200.zip"
    wnid_file = "tiny-imagenet-200/wnids.txt"
    test_path = "tiny-imagenet-200/test/"
    val_path = "tiny-imagenet-200/val/"


    print "Reading from zip..."
    archive = zipfile.ZipFile(zip_url, 'r')
    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 classes"
    classes_words = load_classes_name(wnids, archive)

    print "Loading network"
    network = load_network()
    #network = load_pickle_googlenet()
    #images, images_raw = random_test_images()

    print "Loading images"
    X, X_raw = load_images(val_path, wnids, archive)

    data = zip(X, X_raw)
    np.random.shuffle(data)
    data = data[:5]
    X, X_raw = zip(*data)

    classes = load_classes(wnid_file, archive)

    print "Making predictions"
    save_predictions(X, X_raw, network, classes, classes_words)



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