eBook Gratuit

APPRENEZ caffe

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#caffe

Table des matières

À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: caffe

It is an unofficial and free caffe ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official caffe.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapitre 1: Commencer avec caffe

Remarques

Caffe est une bibliothèque écrite en C ++, pour faciliter l'expérimentation et l'utilisation des réseaux de neurones à convolution (CNN). Caffe a été développé par Berkeley Vision and Learning Center (BVLC).

Caffe est en fait une abréviation faisant référence à "Architectures convolutionnelles pour l'extraction rapide de caractéristiques". Cet acronyme englobe une partie importante de la bibliothèque. Caffe sous la forme d'une bibliothèque offre un cadre / une architecture de programmation générale qui peut être utilisé pour effectuer une formation et des tests efficaces des CNN. "Efficiency" est une caractéristique majeure de caffe, et constitue un objectif de design majeur de Caffe.

Caffe est une bibliothèque open source publiée sous licence BSD 2 Clause.

Caffe est maintenu sur GitHub

Caffe peut être utilisé pour:

- Entraînez et testez efficacement plusieurs architectures CNN, en particulier toute architecture pouvant être représentée sous forme de graphe acyclique dirigé (DAG).
- Utilisez plusieurs GPU (jusqu'à 4) pour la formation et les tests. Il est recommandé que tous les GPU soient du même type. Sinon, les performances sont limitées par les limites du processeur graphique le plus lent du système. Par exemple, dans le cas de TitanX et GTX 980, les performances seront limitées par ce dernier. Le mélange de plusieurs architectures n'est pas pris en charge, par exemple Kepler et Fermi 3 .

Caffe a été écrit selon des principes efficaces de programmation orientée objet (OOP).

Un bon point de départ pour commencer une introduction au café consiste à avoir une vue d'ensemble de la façon dont la café fonctionne à travers ses objets fondamentaux.

Versions

Version	Date de sortie
1.0	2017-04-19

Examples

Installation et configuration

Ubuntu

Vous trouverez ci-dessous des instructions détaillées pour installer Caffe, pycaffe ainsi que ses dépendances, sur Ubuntu 14.04 x64 ou 14.10 x64.

Exécutez le script suivant, par exemple "bash compile_caffe_ubuntu_14.sh" (~ 30 à 60 minutes sur un nouvel Ubuntu).

```
# This script installs Caffe and pycaffe.
# CPU only, multi-threaded Caffe.

# Usage:
# 0. Set up here how many cores you want to use during the installation:
# By default Caffe will use all these cores.
NUMBER_OF_CORES=4

sudo apt-get install -y libprotobuf-dev libleveldb-dev libsnappy-dev
sudo apt-get install -y libopencv-dev libhdf5-serial-dev
sudo apt-get install -y --no-install-recommends libboost-all-dev
sudo apt-get install -y libatlas-base-dev
sudo apt-get install -y python-dev
sudo apt-get install -y python-pip git

# For Ubuntu 14.04
sudo apt-get install -y libgflags-dev libgoogle-glog-dev liblmdb-dev protobuf-compiler

# Install LMDB
git clone https://github.com/LMDB/lmdb.git
cd lmdb/libraries/liblmdb
sudo make
sudo make install

# More pre-requisites
sudo apt-get install -y cmake unzip doxygen
sudo apt-get install -y protobuf-compiler
sudo apt-get install -y libffi-dev python-pip python-dev build-essential
sudo pip install lmdb
sudo pip install numpy
sudo apt-get install -y python-numpy
sudo apt-get install -y gfortran # required by scipy
sudo pip install scipy # required by scikit-image
sudo apt-get install -y python-scipy # in case pip failed
sudo apt-get install -y python-nose
sudo pip install scikit-image # to fix https://github.com/BVLC/caffe/issues/50


# Get caffe (http://caffe.berkeleyvision.org/installation.html#compilation)
cd
mkdir caffe
cd caffe
wget https://github.com/BVLC/caffe/archive/master.zip
unzip -o master.zip
cd caffe-master

# Prepare Python binding (pycaffe)
cd python
for req in $(cat requirements.txt); do sudo pip install $req; done
```

```
# to be able to call "import caffe" from Python after reboot:
echo "export PYTHONPATH=$(pwd):$PYTHONPATH " >> ~/.bash_profile
source ~/.bash_profile # Update shell
cd ..

# Compile caffe and pycaffe
cp Makefile.config.example Makefile.config
sed -i '8s/.*/CPU_ONLY := 1/' Makefile.config # Line 8: CPU only
sudo apt-get install -y libopenblas-dev
sed -i '33s/.*/BLAS := open/' Makefile.config # Line 33: to use OpenBLAS
# Note that if one day the Makefile.config changes and these line numbers may change
echo "export OPENBLAS_NUM_THREADS=($NUMBER_OF_CORES)" >> ~/.bash_profile
mkdir build
cd build
cmake ..
cd ..
make all -j$NUMBER_OF_CORES # 4 is the number of parallel threads for compilation: typically
equal to number of physical cores
make pycaffe -j$NUMBER_OF_CORES
make test
make runtest
#make matcaffe
make distribute

# Afew few more dependencies for pycaffe
sudo pip install pydot
sudo apt-get install -y graphviz
sudo pip install scikit-learn
```

A la fin, vous devez lancer "source ~ / .bash_profile" manuellement ou démarrer un nouveau shell pour pouvoir faire "python import caffe".

Activer le multithreading avec Caffe

Caffe peut fonctionner sur plusieurs cœurs. L'une des méthodes consiste à permettre au multithreading d'utiliser Caffe pour utiliser OpenBLAS au lieu de l'ATLAS par défaut. Pour ce faire, vous pouvez suivre ces trois étapes:

1. `sudo apt-get install -y libopenblas-dev`
2. Avant de compiler Caffe, éditez `Makefile.config` , remplacez `BLAS := atlas` par `BLAS := open`
3. Après la compilation de Caffe, l'exécution de l' `export OPENBLAS_NUM_THREADS=4` entraînera l'utilisation de 4 cœurs par Caffe.

Perte de régularisation (perte de poids) chez Caffe

Dans le fichier du solveur , nous pouvons définir une perte de régularisation globale à l'aide des options `weight_decay` et `regularization_type` .

Dans de nombreux cas, nous souhaitons des taux de décroissance du poids différents pour différentes couches. Cela peut être fait en réglant la `decay_mult` option pour chaque couche dans le fichier de définition du réseau, où `decay_mult` est le multiplicateur sur le taux de décroissance globale de poids, de sorte que le taux de décroissance du poids réel appliqué pour une couche est `decay_mult*weight_decay` .

Par exemple, ce qui suit définit une couche de convolution sans perte de poids, quelles que soient les options du fichier de résolution.

```
layer {
  name: "Convolution1"
  type: "Convolution"
  bottom: "data"
  top: "Convolution1"
  param {
    decay_mult: 0
  }
  convolution_param {
    num_output: 32
    pad: 0
    kernel_size: 3
    stride: 1
    weight_filler {
      type: "xavier"
    }
  }
}
```

Voir ce fil pour plus d'informations.

Lire Commencer avec caffe en ligne: https://riptutorial.com/fr/caffe/topic/4382/commencer-avec-caffe

Chapitre 2: Calques Python personnalisés

Introduction

Ce tutoriel vous guidera à travers les étapes pour créer un simple calque personnalisé pour Caffe en utilisant python. À la fin, il existe des exemples de couches personnalisées. Généralement, vous créez une couche personnalisée pour implémenter une fonctionnalité qui n'est pas disponible dans Caffe, en l'adaptant à vos besoins.

La création d'une couche personnalisée python ajoute un peu de temps à votre réseau et n'est probablement pas aussi efficace qu'une couche personnalisée C ++. Cependant, de cette façon, vous n'aurez pas à compiler tout le caffe avec votre nouveau calque.

Paramètres

Paramètre	Détails
Haut	Un tableau avec les blobs supérieurs de votre couche. Accéder aux données qui lui sont transmises en utilisant *top [i] .data* , où **i** est l'index d'un blob spécifique
bas	Un tableau avec les taches de fond de votre couche. Accédez aux données qui lui sont transmises en utilisant les données de *base [i]* , où **i** est l'index d'un objet blob spécifique

Remarques

- Caffe construit avec la couche Python

Caffe doit être compilé avec l'option `WITH_PYTHON_LAYER` :

```
WITH_PYTHON_LAYER=1 make && make pycaffe
```

- Où dois-je enregistrer le fichier de classe?

Vous avez deux options (du moins que je connais). Soit vous pouvez enregistrer le fichier de couche personnalisé dans le même dossier que vous allez exécuter la commande caffe (probablement là où se trouveraient vos fichiers prototxt). Une autre méthode, également préférée, consiste à enregistrer toutes vos couches personnalisées dans un dossier et à ajouter ce dossier à votre PYTHONPATH.

Les références

1. Le blog de Christopher Bourez
2. Caffe Github
3. StackOverflow

Examples

Modèle de calque

```
import caffe

class My_Custom_Layer(caffe.Layer):
    def setup(self, bottom, top):
        pass

    def forward(self, bottom, top):
        pass

    def reshape(self, bottom, top):
        pass

    def backward(self, bottom, top):
        pass
```

Des choses si importantes à retenir:

- Votre couche personnalisée doit hériter de **caffe.Layer** (n'oubliez donc pas d' *importer caffe*);
- Vous devez définir les quatre méthodes suivantes: **setup** , **forward** , **reshape** et **backward** ;
- Toutes les méthodes ont un *haut* et un *bas* paramètres, qui sont les blobs qui stockent l'entrée et la sortie passée à votre couche. Vous pouvez y accéder en utilisant les données *supérieures [i] .data* ou en *bas [i].* , Où *i* est l'index du blob au cas où vous auriez plus d'un blob supérieur ou inférieur.

- Méthode de configuration

La méthode Setup est appelée une fois au cours de l'exécution, lorsque Caffe instancie toutes les couches. C'est là que vous allez lire les paramètres, instancier des tampons de taille fixe.

- Méthode de remodelage

Utilisez la méthode de remise en forme pour l'initialisation / la configuration qui dépend de la taille du blob inférieur (entrée de couche). Il est appelé une fois lorsque le réseau est instancié.

- méthode de transfert

La méthode Forward est appelée pour chaque lot d'entrée et correspond à la plupart de vos

logiques.

- méthode arrière

La méthode Backward est appelée lors du retour en arrière du réseau. Par exemple, dans une couche de type convolution, ce serait l'endroit où calculer les dégradés. Ceci est facultatif (un calque ne peut être que vers l'avant).

Modèle Prototxt

Ok, maintenant vous avez votre couche conçue! Voici comment vous le définissez dans votre fichier *.prototxt* :

```
layer {
  name: "LayerName"
  type: "Python"
  top: "TopBlobName"
  bottom: "BottomBlobName"
  python_param {
    module: "My_Custom_Layer_File"
    layer: "My_Custom_Layer_Class"
    param_str: '{"param1": 1,"param2":True, "param3":"some string"}'
  }
  include{
        phase: TRAIN
  }
}
```

Remarques importantes:

- **le type** doit être **Python** ;
- Vous devez avoir un dictionnaire **python_param** avec au moins les paramètres de **module** et de **couche** ;
- **module** fait référence au fichier où vous avez implémenté votre couche (sans le *.py*);
- **le calque** fait référence au nom de votre classe;
- Vous pouvez passer des paramètres à la couche en utilisant **param_str** (plus sur leur accès ci-dessous);
- Tout comme n'importe quel autre calque, vous pouvez définir dans quelle phase vous souhaitez qu'il soit actif (voir les exemples pour voir comment vérifier la phase en cours);

Passer des paramètres à la couche

Vous pouvez définir les paramètres de couche dans le prototxt en utilisant *param_str* . Une fois que vous l'avez fait, voici un exemple de la manière dont vous accédez à ces paramètres dans la classe de couche:

```
def setup(self, bottom, top):
    params = eval(self.param_str)
    param1 = params["param1"]
    param2 = params.get('param2', False) #I usually use this when fetching a bool
    param3 = params["param3"]
```

```
    #Continue with the setup
    # ...
```

Mesurer la couche

Dans cet exemple, nous allons concevoir un calque «mesure», qui affiche la précision et une matrice de confusion pour un problème binaire pendant l'entraînement et la précision, le taux de faux positifs et le taux de faux négatifs pendant le test / validation. Bien que Caffe ait déjà une couche de précision, vous voulez parfois quelque chose de plus, comme une mesure F.

Ceci est mon *measureLayer.py* avec ma définition de classe:

```
#Remark: This class is designed for a binary problem, where the first class would be the
'negative'
# and the second class would be 'positive'

import caffe
TRAIN = 0
TEST = 1

class Measure_Layer(caffe.Layer):
    #Setup method
    def setup(self, bottom, top):
        #We want two bottom blobs, the labels and the predictions
        if len(bottom) != 2:
            raise Exception("Wrong number of bottom blobs (prediction and label)")

        #And some top blobs, depending on the phase
        if self.phase = TEST and len(top) != 3:
            raise Exception("Wrong number of top blobs (acc, FPR, FNR)")
        if self.phase = TRAIN and len(top) != 5:
            raise Exception("Wrong number of top blobs (acc, tp, tn, fp and fn)")

        #Initialize some attributes
        self.TPs = 0.0
        self.TNs = 0.0
        self.FPs = 0.0
        self.FNs = 0.0
        self.totalImgs = 0

    #Forward method
    def forward(self, bottom, top):
        #The order of these depends on the prototxt definition
        predictions = bottom[0].data
        labels = bottom[1].data

        self.totalImgs += len(labels)

        for i in range(len(labels)): #len(labels) is equal to the batch size
                pred = predictions[i]   #pred is a tuple with the normalized probability
                                        #of a sample i.r.t. two classes
                lab = labels[i]

                if pred[0] > pred[1]:
                        if lab == 1.0:
                                self.FNs += 1.0
                        else:
```

```
                                self.TNs += 1.0
                        else:
                                if lab == 1.0:
                                        self.TPs += 1.0
                                else:
                                        self.FPs += 1.0

                acc = (self.TPs + self.TNs) / self.totalImgs

                try: #just assuring we don't divide by 0
                        fpr = self.FPs / (self.FPs + self.TNs)
                except:
                        fpr = -1.0

                try: #just assuring we don't divide by 0
                        fnr = self.FNs / (self.FNs + self.TPs)
                except:
                        fnr = -1.0

               #output data to top blob
               top[0].data = acc
               if self.phase == TRAIN:
                   top[1].data = self.TPs
                   top[2].data = self.TNs
                   top[3].data = self.FPs
                   top[4].data = self.FNs
               elif self.phase == TEST:
                   top[1].data = fpr
                   top[2].data = fnr

    def reshape(self, bottom, top):
        """
        We don't need to reshape or instantiate anything that is input-size sensitive
        """
        pass

    def backward(self, bottom, top):
        """
        This layer does not back propagate
        """
        pass
```

Et *voici* un exemple de *prototxt* avec:

```
layer {
  name: "metrics"
  type: "Python"
  top: "Acc"
  top: "TPs"
  top: "TNs"
  top: "FPs"
  top: "FNs"

  bottom: "prediction"   #let's supose we have these two bottom blobs
  bottom: "label"

  python_param {
    module: "measureLayer"
    layer: "Measure_Layer"
  }
```

```
  include {
    phase: TRAIN
  }
}

layer {
  name: "metrics"
  type: "Python"
  top: "Acc"
  top: "FPR"
  top: "FNR"

  bottom: "prediction"   #let's supose we have these two bottom blobs
  bottom: "label"

  python_param {
    module: "measureLayer"
    layer: "Measure_Layer"
  }
  include {
    phase: TEST
  }
}
```

Couche de données

Cet exemple est une couche de données personnalisée qui reçoit un fichier texte avec des chemins d'image, charge un lot d'images et les pré-traite. Juste un petit conseil, Caffe a déjà une grande gamme de couches de données et probablement un calque personnalisé n'est pas le moyen le plus efficace si vous voulez juste quelque chose de simple.

Mon *dataLayer.py* pourrait être quelque chose comme:

```
import caffe

class Custom_Data_Layer(caffe.Layer):
    def setup(self, bottom, top):
        # Check top shape
        if len(top) != 2:
                raise Exception("Need to define tops (data and label)")

        #Check bottom shape
        if len(bottom) != 0:
            raise Exception("Do not define a bottom.")

        #Read parameters
        params = eval(self.param_str)
        src_file = params["src_file"]
        self.batch_size = params["batch_size"]
        self.im_shape = params["im_shape"]
        self.crop_size = params.get("crop_size", False)

        #Reshape top
        if self.crop_size:
            top[0].reshape(self.batch_size, 3, self.crop_size, self.crop_size)
        else:
            top[0].reshape(self.batch_size, 3, self.im_shape, self.im_shape)
```

```
        top[1].reshape(self.batch_size)

        #Read source file
        #I'm just assuming we have this method that reads the source file
        #and returns a list of tuples in the form of (img, label)
        self.imgTuples = readSrcFile(src_file)

        self._cur = 0 #use this to check if we need to restart the list of imgs

    def forward(self, bottom, top):
        for itt in range(self.batch_size):
            # Use the batch loader to load the next image.
            im, label = self.load_next_image()

            #Here we could preprocess the image
            # ...

            # Add directly to the top blob
            top[0].data[itt, ...] = im
            top[1].data[itt, ...] = label

    def load_next_img(self):
        #If we have finished forwarding all images, then an epoch has finished
        #and it is time to start a new one
        if self._cur == len(self.imgTuples):
            self._cur = 0
            shuffle(self.imgTuples)

        im, label = self.imgTuples[self._cur]
        self._cur += 1

        return im, label

    def reshape(self, bottom, top):
        """
        There is no need to reshape the data, since the input is of fixed size
        (img shape and batch size)
        """
        pass

    def backward(self, bottom, top):
        """
        This layer does not back propagate
        """
        pass
```

Et le *prototxt* serait comme:

```
layer {
  name: "Data"
  type: "Python"
  top: "data"
  top: "label"

  python_param {
    module: "dataLayer"
    layer: "Custom_Data_Layer"
    param_str: '{"batch_size": 126,"im_shape":256, "crop_size":224, "src_file":
"path_to_TRAIN_file.txt"}'
  }
```

```
}
```

Lire Calques Python personnalisés en ligne: https://riptutorial.com/fr/caffe/topic/10535/calques-python-personnalises

Chapitre 3: Entraînement d'un modèle Caffe avec pycaffe

Examples

Former un réseau sur le jeu de données Iris

Ci-dessous est un exemple simple pour entraîner un modèle Caffe sur le jeu de données Iris en Python, en utilisant PyCaffe. Il donne également les résultats prévus à partir de certaines entrées définies par l'utilisateur.

`iris_tuto.py`

```
import subprocess
import platform
import copy

from sklearn.datasets import load_iris
import sklearn.metrics
import numpy as np
from sklearn.cross_validation import StratifiedShuffleSplit
import matplotlib.pyplot as plt
import h5py
import caffe
import caffe.draw


def load_data():
    '''
    Load Iris Data set
    '''
    data = load_iris()
    print(data.data)
    print(data.target)
    targets = np.zeros((len(data.target), 3))
    for count, target in enumerate(data.target):
        targets[count][target]= 1
    print(targets)

    new_data = {}
    #new_data['input'] = data.data
    new_data['input'] = np.reshape(data.data, (150,1,1,4))
    new_data['output'] = targets
    #print(new_data['input'].shape)
    #new_data['input'] = np.random.random((150, 1, 1, 4))
    #print(new_data['input'].shape)
    #new_data['output'] = np.random.random_integers(0, 1, size=(150,3))
    #print(new_data['input'])

    return new_data

def save_data_as_hdf5(hdf5_data_filename, data):
    '''
    HDF5 is one of the data formats Caffe accepts
```

```
    '''
    with h5py.File(hdf5_data_filename, 'w') as f:
        f['data'] = data['input'].astype(np.float32)
        f['label'] = data['output'].astype(np.float32)


def train(solver_prototxt_filename):
    '''
    Train the ANN
    '''
    caffe.set_mode_cpu()
    solver = caffe.get_solver(solver_prototxt_filename)
    solver.solve()


def print_network_parameters(net):
    '''
    Print the parameters of the network
    '''
    print(net)
    print('net.inputs: {0}'.format(net.inputs))
    print('net.outputs: {0}'.format(net.outputs))
    print('net.blobs: {0}'.format(net.blobs))
    print('net.params: {0}'.format(net.params))

def get_predicted_output(deploy_prototxt_filename, caffemodel_filename, input, net = None):
    '''
    Get the predicted output, i.e. perform a forward pass
    '''
    if net is None:
        net = caffe.Net(deploy_prototxt_filename,caffemodel_filename, caffe.TEST)

    #input = np.array([[ 5.1,  3.5,  1.4,  0.2]])
    #input = np.random.random((1, 1, 1))
    #print(input)
    #print(input.shape)
    out = net.forward(data=input)
    #print('out: {0}'.format(out))
    return out[net.outputs[0]]


import google.protobuf
def print_network(prototxt_filename, caffemodel_filename):
    '''
    Draw the ANN architecture
    '''
    _net = caffe.proto.caffe_pb2.NetParameter()
    f = open(prototxt_filename)
    google.protobuf.text_format.Merge(f.read(), _net)
    caffe.draw.draw_net_to_file(_net, prototxt_filename + '.png' )
    print('Draw ANN done!')


def print_network_weights(prototxt_filename, caffemodel_filename):
    '''
    For each ANN layer, print weight heatmap and weight histogram
    '''
    net = caffe.Net(prototxt_filename,caffemodel_filename, caffe.TEST)
    for layer_name in net.params:
        # weights heatmap
        arr = net.params[layer_name][0].data
```

```
        plt.clf()
        fig = plt.figure(figsize=(10,10))
        ax = fig.add_subplot(111)
        cax = ax.matshow(arr, interpolation='none')
        fig.colorbar(cax, orientation="horizontal")
        plt.savefig('{0}_weights_{1}.png'.format(caffemodel_filename, layer_name), dpi=100,
format='png', bbox_inches='tight') # use format='svg' or 'pdf' for vectorial pictures
        plt.close()

        # weights histogram
        plt.clf()
        plt.hist(arr.tolist(), bins=20)
        plt.savefig('{0}_weights_hist_{1}.png'.format(caffemodel_filename, layer_name),
dpi=100, format='png', bbox_inches='tight') # use format='svg' or 'pdf' for vectorial pictures
        plt.close()


def get_predicted_outputs(deploy_prototxt_filename, caffemodel_filename, inputs):
    '''
    Get several predicted outputs
    '''
    outputs = []
    net = caffe.Net(deploy_prototxt_filename,caffemodel_filename, caffe.TEST)
    for input in inputs:
        #print(input)
        outputs.append(copy.deepcopy(get_predicted_output(deploy_prototxt_filename,
caffemodel_filename, input, net)))
    return outputs


def get_accuracy(true_outputs, predicted_outputs):
    '''

    '''
    number_of_samples = true_outputs.shape[0]
    number_of_outputs = true_outputs.shape[1]
    threshold = 0.0 # 0 if SigmoidCrossEntropyLoss ; 0.5 if EuclideanLoss
    for output_number in range(number_of_outputs):
        predicted_output_binary = []
        for sample_number in range(number_of_samples):
            #print(predicted_outputs)
            #print(predicted_outputs[sample_number][output_number])
            if predicted_outputs[sample_number][0][output_number] < threshold:
                predicted_output = 0
            else:
                predicted_output = 1
            predicted_output_binary.append(predicted_output)

        print('accuracy: {0}'.format(sklearn.metrics.accuracy_score(true_outputs[:,
output_number], predicted_output_binary)))
        print(sklearn.metrics.confusion_matrix(true_outputs[:, output_number],
predicted_output_binary))


def main():
    '''
    This is the main function
    '''

    # Set parameters
    solver_prototxt_filename = 'iris_solver.prototxt'
```

```
    train_test_prototxt_filename = 'iris_train_test.prototxt'
    deploy_prototxt_filename  = 'iris_deploy.prototxt'
    deploy_prototxt_filename  = 'iris_deploy.prototxt'
    deploy_prototxt_batch2_filename  = 'iris_deploy_batchsize2.prototxt'
    hdf5_train_data_filename = 'iris_train_data.hdf5'
    hdf5_test_data_filename = 'iris_test_data.hdf5'
    caffemodel_filename = 'iris__iter_5000.caffemodel' # generated by train()

    # Prepare data
    data = load_data()
    print(data)
    train_data = data
    test_data = data
    save_data_as_hdf5(hdf5_train_data_filename, data)
    save_data_as_hdf5(hdf5_test_data_filename, data)

    # Train network
    train(solver_prototxt_filename)

    # Print network
    print_network(deploy_prototxt_filename, caffemodel_filename)
    print_network(train_test_prototxt_filename, caffemodel_filename)
    print_network_weights(train_test_prototxt_filename, caffemodel_filename)

    # Compute performance metrics
    #inputs = input = np.array([[[[ 5.1,  3.5,  1.4,  0.2]]],[[[ 5.9,  3. ,  5.1,  1.8]]]])
    inputs = data['input']
    outputs = get_predicted_outputs(deploy_prototxt_filename, caffemodel_filename, inputs)
    get_accuracy(data['output'], outputs)


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

Il faut que les deux suivants `iris_train_test.prototxt` et `iris_deploy.prototxt` soient dans le même dossier.

`iris_train_test.prototxt` :

```
name: "IrisNet"
layer {
  name: "iris"
  type: "HDF5Data"
  top: "data"
  top: "label"
  include {
    phase: TRAIN
  }
  hdf5_data_param {
    source: "iris_train_data.txt"
    batch_size: 1

  }
}

layer {
  name: "iris"
  type: "HDF5Data"
  top: "data"
```

```
  top: "label"
  include {
    phase: TEST
  }
  hdf5_data_param {
    source: "iris_test_data.txt"
    batch_size: 1

  }
}




layer {
  name: "ip1"
  type: "InnerProduct"
  bottom: "data"
  top: "ip1"
  param {
    lr_mult: 1  # the learning rate multiplier for weights
  }
  param {
    lr_mult: 2  # the learning rate multiplier for biases
  }
  inner_product_param {
    num_output: 50
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}
layer {
  name: "relu1"
  type: "ReLU"
  bottom: "ip1"
  top: "ip1"
}
layer {
  name: "drop1"
  type: "Dropout"
  bottom: "ip1"
  top: "ip1"
  dropout_param {
    dropout_ratio: 0.5
  }
}


layer {
  name: "ip2"
  type: "InnerProduct"
  bottom: "ip1"
  top: "ip2"
  param {
    lr_mult: 1
  }
  param {
```

```
    lr_mult: 2
  }
  inner_product_param {
    num_output: 50
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}
layer {
  name: "drop2"
  type: "Dropout"
  bottom: "ip2"
  top: "ip2"
  dropout_param {
    dropout_ratio: 0.4
  }
}



layer {
  name: "ip3"
  type: "InnerProduct"
  bottom: "ip2"
  top: "ip3"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  inner_product_param {
    num_output: 3
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}

layer {
  name: "drop3"
  type: "Dropout"
  bottom: "ip3"
  top: "ip3"
  dropout_param {
    dropout_ratio: 0.3
  }
}

layer {
  name: "loss"
  type: "SigmoidCrossEntropyLoss"
  # type: "EuclideanLoss"
  # type: "HingeLoss"
```

```
  bottom: "ip3"
  bottom: "label"
  top: "loss"
}
```

`iris_deploy.prototxt` :

```
name: "IrisNet"
input: "data"
input_dim: 1 # batch size
input_dim: 1
input_dim: 1
input_dim: 4


layer {
  name: "ip1"
  type: "InnerProduct"
  bottom: "data"
  top: "ip1"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  inner_product_param {
    num_output: 50
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}
layer {
  name: "relu1"
  type: "ReLU"
  bottom: "ip1"
  top: "ip1"
}
layer {
  name: "drop1"
  type: "Dropout"
  bottom: "ip1"
  top: "ip1"
  dropout_param {
    dropout_ratio: 0.5
  }
}


layer {
  name: "ip2"
  type: "InnerProduct"
  bottom: "ip1"
  top: "ip2"
  param {
    lr_mult: 1
```

```
  }
  param {
    lr_mult: 2
  }
  inner_product_param {
    num_output: 50
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}
layer {
  name: "drop2"
  type: "Dropout"
  bottom: "ip2"
  top: "ip2"
  dropout_param {
    dropout_ratio: 0.4
  }
}


layer {
  name: "ip3"
  type: "InnerProduct"
  bottom: "ip2"
  top: "ip3"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  inner_product_param {
    num_output: 3
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}

layer {
  name: "drop3"
  type: "Dropout"
  bottom: "ip3"
  top: "ip3"
  dropout_param {
    dropout_ratio: 0.3
  }
}
```

`iris_solver.prototxt` :

```
# The train/test net protocol buffer definition
```

```
net: "iris_train_test.prototxt"
# test_iter specifies how many forward passes the test should carry out.
test_iter: 1
# Carry out testing every test_interval training iterations.
test_interval: 1000
# The base learning rate, momentum and the weight decay of the network.
base_lr: 0.0001
momentum: 0.001
weight_decay: 0.0005
# The learning rate policy
lr_policy: "inv"
gamma: 0.0001
power: 0.75
# Display every 100 iterations
display: 1000
# The maximum number of iterations
max_iter: 5000
# snapshot intermediate results
snapshot: 5000
snapshot_prefix: "iris_"
# solver mode: CPU or GPU
solver_mode: CPU # GPU
```

Lire Entraînement d'un modèle Caffe avec pycaffe en ligne:
https://riptutorial.com/fr/caffe/topic/4618/entrainement-d-un-modele-caffe-avec-pycaffe

Chapitre 4: Normalisation par lots

Introduction

De la documentation :

"Normalise l'entrée pour obtenir une variance moyenne et / ou unitaire (1) dans le lot.

Cette couche calcule la normalisation par lots comme décrit dans [1].

[...]

[1] S. Ioffe et C. Szegedy, "Normalisation par lots: Accélération de la formation en réseau profond en réduisant le décalage interne des covariables". arXiv preprint arXiv: 1502.03167 (2015). "

Paramètres

Paramètre	Détails
use_global_stats	D'après le post de rohrbach du 2 mars 2016 - peut-être qu'il sait:
(use_global_stats)	"Par défaut, pendant le temps d'entraînement, le réseau calcule des statistiques de moyenne / variance globales via une moyenne mobile, qui est ensuite utilisée au moment du test pour autoriser des sorties déterministes pour chaque entrée. via l'option use_global_stats IMPORTANT: pour que cette fonctionnalité fonctionne, vous DEVEZ définir le taux d'apprentissage à zéro pour les trois blobs de paramètres, c'est-à-dire param {lr_mult: 0} trois fois dans la définition de la couche.
(use_global_stats)	Cela signifie que par défaut (comme suit est défini dans batch_norm_layer.cpp), vous n'avez pas besoin de définir use_global_stats dans le prototxt. use_global_stats_ = this-> phase_ == TEST; "

Examples

Prototxt pour la formation

Vous trouverez ci-dessous un exemple de définition pour l'entraînement d'une couche BatchNorm avec une échelle et un biais au niveau des canaux. Généralement, une couche BatchNorm est insérée entre les couches de convolution et de rectification. Dans cet exemple, la convolution produirait le `layerx` et la rectification recevrait le blob `layerx-bn` .

```
layer { bottom: 'layerx' top: 'layerx-bn' name: 'layerx-bn' type: 'BatchNorm'
  batch_norm_param {
```

```
    use_global_stats: false  # calculate the mean and variance for each mini-batch
    moving_average_fraction: .999  # doesn't effect training
  }
  param { lr_mult: 0 }
  param { lr_mult: 0 }
  param { lr_mult: 0 }}
# channel-wise scale and bias are separate
layer { bottom: 'layerx-bn' top: 'layerx-bn' name: 'layerx-bn-scale' type: 'Scale',
  scale_param {
    bias_term: true
    axis: 1      # scale separately for each channel
    num_axes: 1  # ... but not spatially (default)
    filler { type: 'constant' value: 1 }           # initialize scaling to 1
    bias_filler { type: 'constant' value: 0.001 }  # initialize bias
}}
```

Plus d'informations peuvent être trouvées dans ce fil .

Prototxt pour le déploiement

Le principal changement nécessaire consiste à basculer `use_global_stats` sur `true` . Cela passe en utilisant la moyenne mobile.

```
layer { bottom: 'layerx' top: 'layerx-bn' name: 'layerx-bn' type: 'BatchNorm'
  batch_norm_param {
    use_global_stats: true  # use pre-calculated average and variance
  }
  param { lr_mult: 0 }
  param { lr_mult: 0 }
  param { lr_mult: 0 }}
# channel-wise scale and bias are separate
layer { bottom: 'layerx-bn' top: 'layerx-bn' name: 'layerx-bn-scale' type: 'Scale',
  scale_param {
    bias_term: true
    axis: 1      # scale separately for each channel
    num_axes: 1  # ... but not spatially (default)
}}
```

Lire Normalisation par lots en ligne: https://riptutorial.com/fr/caffe/topic/6575/normalisation-par-lots

Chapitre 5: Objets de base Caffe - Solver, Net, Layer et Blob

Remarques

Un utilisateur caffe envoie des instructions pour effectuer des opérations spécifiques sur des objets Caffe. Ces objets interagissent les uns avec les autres en fonction de leurs spécifications de conception et exécutent les opérations. C'est un principe de base de la POO.

Bien qu'il existe de nombreux types d'objet caffe (ou classes C ++), pour commencer, nous nous concentrons sur 4 objets caffe importants. Notre objectif à ce stade est d'observer simplement l'interaction entre ces objets à un niveau très abstrait où les détails spécifiques de la mise en œuvre et de la conception sont obscurcis.

Les 4 objets caffe de base sont:

- **Solveur**
- **Net**
- **Couche**
- **Goutte**

Une introduction très basique et une vue d'ensemble de leur rôle dans le travail de Caffe est présentée dans des points concis dans la section des exemples.

Après avoir lu et obtenu une idée de base de la manière dont ces objets Caffé interagissent, chaque type d'objet peut être lu en détail dans leurs rubriques dédiées.

Examples

Comment ces objets interagissent ensemble.

- Un utilisateur cherche à utiliser caffe pour la formation et les tests CNN. L'utilisateur décide de la conception de l'architecture CNN (par exemple: nombre de couches, nombre de filtres et leurs détails, etc.). L'utilisateur décide également de la technique d'optimisation des paramètres de formation et d'apprentissage au cas où une formation doit être effectuée. Si l'opération est un test simple, un modèle pré-formé est spécifié par l'utilisateur. En utilisant toutes ces informations, l'utilisateur instancie un objet Solver et fournit à l'objet Solver une instruction (qui détermine la ou les opérations telles que l'entraînement et les tests).

- **Solver** : Cet objet peut être considéré comme une entité qui supervise la formation et les tests d'un CNN. C'est l'entrepreneur réel qui obtient un CNN sur le processeur et en cours d'exécution. Il est spécialisé dans la réalisation des optimisations spécifiques conduisant à la formation de CNN.

- **Net** : Net peut être considéré comme un objet spécialisé qui représente le CNN réel sur

lequel des opérations sont effectuées. Solver demande à Net d’allouer de la mémoire pour le CNN et de l’instancier. Net est également responsable de donner des instructions qui conduisent effectivement à la transmission ou à la propagande sur le réseau CNN.

- **Couche** : C'est un objet qui représente une couche particulière d'un CNN. Ainsi, un CNN est constitué de couches. En ce qui concerne caffe, l'objet **Net** instancie chaque type de " **couche** " spécifié dans la définition de l'architecture et connecte également les différentes couches. Une couche spécifique exécute un ensemble spécifique d'opérations (p. Ex., Mise en commun maximale, regroupement min, convolution 2D, etc.).
- **Blob** : les données transitent par un CNN pendant la formation et les tests. Outre les données utilisateur, ces données comprennent également plusieurs calculs intermédiaires exécutés sur CNN. Ces données sont encapsulées dans un objet appelé Blob.

Lire Objets de base Caffe - Solver, Net, Layer et Blob en ligne:
https://riptutorial.com/fr/caffe/topic/5810/objets-de-base-caffe---solver--net--layer-et-blob

Chapitre 6: Préparer des données pour la formation

Examples

Préparer un dataset d'image pour une tâche de classification d'image

Caffe possède une couche d'entrée intégrée adaptée aux tâches de classification des images (c.-à-d. Une seule étiquette entière par image d'entrée). Cette couche d'entrée `"Data"` est construite sur une structure de données lmdb ou leveldb . Pour utiliser `"Data"` couche `"Data"` , il faut construire la structure de données avec toutes les données de formation.

Un guide rapide de `convert_imageset` de Caffe

Construire

La première chose à faire est de créer les outils de caffe et de caffe (`convert_imageset` est l'un de ces outils).
Après avoir installé caffe et `make` ing - il que vous avez exécuté `make tools` des `make tools` aussi bien.
Vérifiez qu'un fichier binaire `convert_imageset` est créé dans `$CAFFE_ROOT/build/tools` .

Préparez vos données

Images: placez toutes les images dans un dossier (je l'appellerai ici `/path/to/jpegs/`).
Étiquettes: créez un fichier texte (par exemple, `/path/to/labels/train.txt`) avec une ligne par image d'entrée <path / to / file>. Par exemple:

> img_0000.jpeg 1
> img_0001.jpeg 0
> img_0002.jpeg 0

Dans cet exemple, la première image est étiquetée `1` tandis que les deux autres sont étiquetées `0` .

Convertir le jeu de données

Exécuter le binaire en shell

```
~$ GLOG_logtostderr=1 $CAFFE_ROOT/build/tools/convert_imageset \
    --resize_height=200 --resize_width=200 --shuffle  \
    /path/to/jpegs/ \
    /path/to/labels/train.txt \
    /path/to/lmdb/train_lmdb
```

Ligne de commande expliquée:

- `GLOG_logtostderr` indicateur `GLOG_logtostderr` est défini sur 1 *avant d'* appeler `convert_imageset` indique le mécanisme de journalisation pour rediriger les messages de journal vers stderr.
- `--resize_height` et `--resize_width` redimensionnent **toutes** les images d'entrée de la même taille `200x200` .
- `--shuffle` change aléatoirement l'ordre des images et ne conserve pas la commande dans le fichier `/path/to/labels/train.txt` .
- Vous trouverez ci-dessous le chemin d'accès au dossier images, au fichier texte des étiquettes et au nom de la sortie. Notez que le nom de sortie ne doit pas exister avant d'appeler `convert_imageset` sinon vous aurez un message d'erreur effrayant.

Autres indicateurs pouvant être utiles:

- `--backend` - vous permet de choisir entre un `lmdb` données `levelDB` ou `levelDB` .
- `--gray` - convertit toutes les images en `--gray` de gris.
- `--encoded` et `--encoded_type` - conservent les données d'image sous forme compressée (jpg / png) compressée dans la base de données.
- `--help` - affiche de l'aide, voir tous les indicateurs pertinents sous *Drapeaux à partir d'outils / convert_imageset.cpp*

Vous pouvez consulter `$CAFFE_ROOT/examples/imagenet/convert_imagenet.sh` pour un exemple d'utilisation de `convert_imageset` .

voir ce fil pour plus d'informations.

Préparer des données arbitraires au format HDF5

En plus des jeux de données de classification des images , Caffe possède également une couche `"HDF5Data"` pour les entrées arbitraires. Cette couche nécessite que toutes les données de formation / validation soient stockées dans des fichiers au format hdf5 .
Cet exemple montre comment utiliser le module python `h5py` pour construire un tel fichier hdf5 et comment configurer la couche `"HDF5Data"` caffe pour lire ce fichier.

Construisez le fichier binaire hdf5

En supposant que vous avez un fichier texte `'train.txt'` avec chaque ligne avec un nom de fichier image et un seul nombre à virgule flottante à utiliser comme cible de régression.

```
import h5py, os
import caffe
import numpy as np

SIZE = 224 # fixed size to all images
with open( 'train.txt', 'r' ) as T :
    lines = T.readlines()
# If you do not have enough memory split data into
# multiple batches and generate multiple separate h5 files
X = np.zeros( (len(lines), 3, SIZE, SIZE), dtype='f4' )
y = np.zeros( (1,len(lines)), dtype='f4' )
```

```
for i,l in enumerate(lines):
    sp = l.split(' ')
    img = caffe.io.load_image( sp[0] )
    img = caffe.io.resize( img, (SIZE, SIZE, 3) ) # resize to fixed size
    # you may apply other input transformations here...
    # Note that the transformation should take img from size-by-size-by-3 and transpose it to
3-by-size-by-size
    X[i] = img
    y[i] = float(sp[1])
with h5py.File('train.h5','w') as H:
    H.create_dataset( 'X', data=X ) # note the name X given to the dataset!
    H.create_dataset( 'y', data=y ) # note the name y given to the dataset!
with open('train_h5_list.txt','w') as L:
    L.write( 'train.h5' ) # list all h5 files you are going to use
```

Configuration de la couche `"HDF5Data"`

Une fois que vous avez tous les fichiers `h5` et les fichiers de test correspondants, vous pouvez ajouter une couche d'entrée HDF5 à votre `train_val.prototxt` :

```
 layer {
   type: "HDF5Data"
   top: "X" # same name as given in create_dataset!
   top: "y"
   hdf5_data_param {
     source: "train_h5_list.txt" # do not give the h5 files directly, but the list.
     batch_size: 32
   }
   include { phase:TRAIN }
 }
```

Vous pouvez trouver plus d'informations ici et ici .

Comme indiqué ci-dessus, nous transmettons à Caffe une liste de fichiers HDF5. En effet, dans la version actuelle, il existe une limite de taille de 2 Go pour un seul fichier de données HDF5. Donc, si les données d'entraînement dépassent 2 Go, nous devrons les diviser en fichiers séparés.

Si un seul fichier de données HDF5 dépasse 2 Go, nous recevrons un message d'erreur tel que

```
Check failed: shape[i] <= 2147483647 / count_ (100 vs. 71) blob size exceeds INT_MAX
```

Si la quantité totale de données est inférieure à 2 Go, devons-nous diviser les données en fichiers séparés ou non?

Selon un commentaire du code source de Caffe , un seul fichier serait préférable,

> Si shuffle == true, l'ordre des fichiers HDF5 est mélangé et l'ordre des données dans un fichier HDF5 donné est mélangé, mais les données entre différents fichiers ne sont pas entrelacées.

Lire Préparer des données pour la formation en ligne:

https://riptutorial.com/fr/caffe/topic/5344/preparer-des-donnees-pour-la-formation

Crédits

S. No	Chapitres	Contributeurs
1	Commencer avec caffe	Community, dontloo, Franck Dernoncourt, GoodDeeds, Parag S. Chandakkar, Shai, Tahir Shahzad, Ujjwal Aryan
2	Calques Python personnalisés	Fernanda Andalo, rafaspadilha
3	Entraînement d'un modèle Caffe avec pycaffe	Franck Dernoncourt, Parag S. Chandakkar
4	Normalisation par lots	dasWesen, Jonathan, Shai
5	Objets de base Caffe - Solver, Net, Layer et Blob	Ujjwal Aryan
6	Préparer des données pour la formation	dontloo, malreddysid, Shai, Stephen Leppik