Machine Learning for Computer Vision

Exercise 5

Kodai Matsuoka, Yuyan Li

26 May, 2017

1 Features

Our unary and Potts features are computed with the code in Listing 1.

Listing 1: Feature functions

```
def get_unary_features(raw):
    features = []

    ##############################################
    # ADD YOUR CODE HERE
    ##############################################

    raw = norm01(raw)
    sigmas = [0.5, 1.0, 1.5, 2.0, 3.0]
    for sigma in sigmas:
        feat = skimage.filters.gaussian(raw, sigma=sigma)
        features.append(feat[:,:,None])

    features.append(numpy.ones(raw.shape)[:,:,None])
    return numpy.concatenate(features, axis=2)


def get_potts_features(raw):
    features = []

    ##############################################
    # ADD YOUR CODE HERE
    ##############################################
    raw = norm01(raw)
    sigmas = [0.5, 1.0, 1.5, 2.0, 3.0]
    for sigma in sigmas:
        smooth = skimage.filters.gaussian(raw, sigma=sigma)
        edge = skimage.filters.laplace(smooth)
        feat = numpy.exp(-1.0*numpy.abs(edge))
```

```
        features.append(feat[:,:,None])

    # a constant feature is needed
    features.append(numpy.ones(raw.shape)[:,:,None])
    return numpy.concatenate(features, axis=2)
```

2 Test set performance with GraphCut

The full code is at the bottom in Listing 3. We have a variable `mode` for the different exercises.
For this part we chose `mode='gp'` to use graph cut.
The computed loss values for different noises and regularizers are:

```
[[[ 7 15  1 10  8]
  [ 7 13  2  8  8]
  [ 7 13  2  9  9]
  [ 7 14  1 10  9]
  [ 7 14  1 10  9]]

 [[16 14 17 15 19]
  [15 15 17 19 20]
  [16 15 12 19 20]
  [16 16 13 20 20]
  [16 15 13 20 20]]

 [[10 12 16 10 11]
  [17 14 22 10 15]
  [17 14 23 10 15]
  [17 14 22 10 15]
  [19 14 22 10 15]]

 [[22 18 16 21 15]
  [20 21 19 24 16]
  [20 23 19 22 18]
  [20 23 19 24 18]
  [20 23 19 24 18]]

 [[40 25 18 32 22]
  [40 30 18 41 32]
  [37 28 18 40 38]
  [41 30 17 42 43]
  [41 30 17 42 43]]]
```

The content of this list structure is:

- every row is the loss of the five test images of a certain regularizer C and noise
- every list of five rows belong to a certain noise

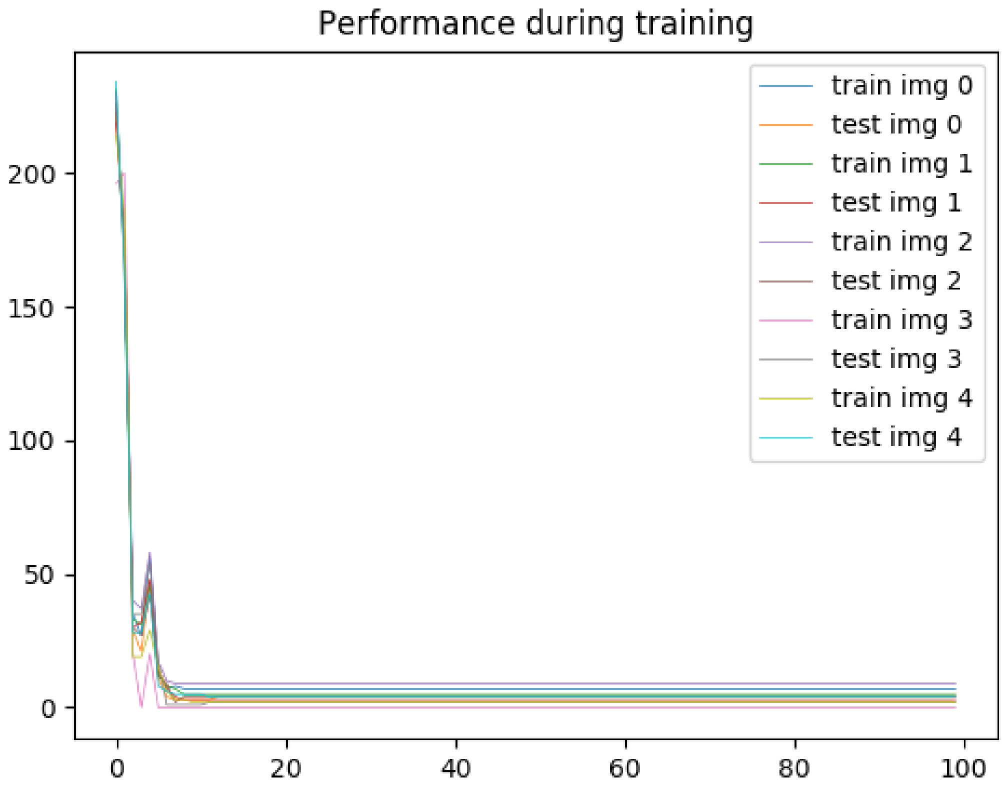
The noises and regularizer values are sorted as given in the exercise.
As expected, a bigger noise leads to worse predictions and therefore higher losses.

3 Test set performance with ICM

The same is done using an ICM solver instead of GraphCut by setting `mode='icm'`.

```
[[[ 5  5  6  9  5]
  [ 5  5  6  9  5]
  [ 5  5  6  8  5]
  [ 5  5  6  8  5]
  [ 5  5  6  8  5]]

 [[ 1 10 12 14  0]
  [ 5 11 12 15  0]
  [ 4 12 11 14  0]
  [ 5 11 13 15  0]
  [ 5 11 13 14  0]]

 [[16 12 12 17 15]
  [17 13 22 18 14]
  [17 13 21 19 14]
  [17 13 22 20 14]
  [17 13 22 20 14]]

 [[21 22 22 17 16]
  [22 25 20 22 17]
  [23 27 20 22 16]
  [22 24 19 22 16]
  [22 24 18 22 16]]

 [[56 46 21 26 34]
  [40 24 34 14 60]
  [46 24 43 17 60]
  [46 24 55 19 60]
  [44 26 44 21 60]]]
```

Qualitatively, there does not seem to be a difference to the GraphCut results.



4 Bonus: show performance during training

With the code in Listing 2 the training and test set performances are show after every iteration during training.
If `mode='bonus'` then after every training a plot with the losses is shown. One exemplary plot can be seen in Section 4.

Listing 2: Code to show performance after every iteration during training.

```
# ==========
# show performance during training
# ==========

losses = []
for gm,_,loss_function in dataset.models_train:
    gm.change_weights(weights)
    graphcut = GraphCut(model=gm)
    y_pred = graphcut.optimize()
    losses.append(loss_function(y_pred))
loss_train.append(losses)
print('training set performance', losses)
```

```
        losses = []
        for gm,_,loss_function in dataset.models_test:
            gm.change_weights(weights)
            graphcut = GraphCut(model=gm)
            y_pred = graphcut.optimize()
            losses.append(loss_function(y_pred))
        loss_test.append(losses)
        print('test set performance', losses)

    # in bonus mode: make a plot of performance
    if mode=='bonus':
        plottrain = numpy.array(loss_train).T
        plottest = numpy.array(loss_test).T
        for i in range(plottrain.shape[0]):
            plt.plot(range(n_iter),plottrain[i], lw=0.5, label=f'train img
                 {i}')
            plt.plot(range(n_iter),plottest[i], lw=0.5, label=f'test img {
                i}')
        plt.title('Performance during training')
        plt.legend()
```

Listing 3: StructLearn.py

```
# Structured Learning:
# ====================
#
# In this exercise we will implement a structured learning system for
# foreground background segmentation.
# We will learn the weights of a CRF Potts model.
#
# The first step is to  import all needed modules

# misc
import numpy
import sys

# visualization
import matplotlib.pyplot as plt
import pylab

# features
import skimage.filters

# discrete graphical model package
from dgm.models import *
from dgm.solvers import *
from dgm.value_tables import *
# misc. tools
from tools import make_toy_dataset, norm01

from skimage.morphology import disk

def get_unary_features(raw):
    features = []

    #############################################
    # ADD YOUR CODE HERE
    #############################################

    raw = norm01(raw)
    sigmas = [0.5, 1.0, 1.5, 2.0, 3.0]
    for sigma in sigmas:
        feat = skimage.filters.gaussian(raw, sigma=sigma)
        features.append(feat[:,:,None])

    features.append(numpy.ones(raw.shape)[:,:,None])
    return  numpy.concatenate(features, axis=2)


def get_potts_features(raw):
    features = []

    #############################################
    # ADD YOUR CODE HERE
```

```python
    ###############################################
    raw = norm01(raw)
    sigmas = [0.5, 1.0, 1.5, 2.0, 3.0]
    for sigma in sigmas:
        smooth = skimage.filters.gaussian(raw, sigma=sigma)
        edge = skimage.filters.laplace(smooth)
        feat = numpy.exp(-1.0*numpy.abs(edge))
        features.append(feat[:,:,None])

    # a constant feature is needed
    features.append(numpy.ones(raw.shape)[:,:,None])
    return  numpy.concatenate(features, axis=2)


class HammingLoss(object):
    def __init__(self, y_true):
        self.y_true = y_true.copy()

    def __call__(self, y_pred):
        """total loss"""
        return numpy.sum(self.y_true!=y_pred)


def build_model(raw_data, gt_image, weights):
    shape = raw_data.shape
    n_var = shape[0] * shape[1]
    n_labels = 2
    variable_space = numpy.ones(n_var)*n_labels

    # lets compute some filters for the uanry features
    unary_features = get_unary_features(raw_data)

    # lets compute some filters for the potts features
    potts_features = get_potts_features(raw_data)

    n_weights = potts_features.shape[2] + unary_features.shape[2]

    #assert n_weights == len(weights)

    # both graphical models
    gm = WeightedDiscreteGraphicalModel(variable_space=variable_space,
       weights=weights)
    loss_augmented_gm = WeightedDiscreteGraphicalModel(variable_space=
       variable_space, weights=weights)

    # convert coordinates to scalar
    def vi(x0,x1):
        return x1 + x0*shape[1]

    # weight ids for the unaries (just plain numbers to remeber which
       weights are associated with the unary features)
    weight_ids = numpy.arange(unary_features.shape[2])
```

```
for x0 in range(shape[0]):
    for x1 in range(shape[1]):

        pixel_val = raw_data[x0, x1]
        gt_label = gt_image[x0, x1]
        features = unary_features[x0, x1, :]

        unary_function = WeightedTwoClassUnary(features=features,
            weight_ids=weight_ids,
                                                weights=weights)

        if gt_label == 0:
            loss = numpy.array([0,1])
        else:
            loss = numpy.array([1,0])

        loss_augmented_unary_function = WeightedTwoClassUnary(features
            =features, weight_ids=weight_ids,
                                                weights=weights,
                                                    const_terms=-1.0*
                                                    loss)

        variables = vi(x0,x1)
        gm.add_factor(variables=variables, value_table=unary_function)
        loss_augmented_gm.add_factor(variables=variables, value_table=
            loss_augmented_unary_function)

# average over 2 coordinates to extract feature vectors for potts
    functins
def get_potts_feature_vec(coord_a, coord_b):

    fa = potts_features[coord_a[0],coord_a[1],:]
    fb = potts_features[coord_b[0],coord_b[1],:]
    return (fa+fb)/2.0

# weight ids for the potts functions (just plain numbers to remeber
    which weights are associated with the potts features)
weight_ids = numpy.arange(potts_features.shape[2]) + unary_features.
    shape[2]

for x0 in range(shape[0]):
    for x1 in range(shape[1]):

        # horizontal edge
        if x0 + 1 < shape[0]:
            variables = [vi(x0,x1),vi(x0+1,x1)]
            features = get_potts_feature_vec((x0,x1), (x0+1,x1))
            # the weighted potts function
            potts_function = WeightedPottsFunction(shape=[2,2],
                                                    features=features,
                                                    weight_ids=
                                                        weight_ids,
```

```
                                                    weights=weights)
                # add factors to both models
                gm.add_factor(variables=variables, value_table=
                    potts_function)
                loss_augmented_gm.add_factor(variables=variables,
                    value_table=potts_function)

            # vertical edge
            if x1 + 1 < shape[1]:
                variables = [vi(x0,x1),vi(x0, x1+1)]
                features = get_potts_feature_vec((x0,x1), (x0,x1+1))
                # the weighted potts function
                potts_function = WeightedPottsFunction(shape=[2,2],
                                                    features=features,
                                                    weight_ids=
                                                        weight_ids,
                                                    weights=weights)
                # add factors to both models
                gm.add_factor(variables=variables, value_table=
                    potts_function)
                loss_augmented_gm.add_factor(variables=variables,
                    value_table=potts_function)

    # gm, loss augmented and the loss
    return gm,loss_augmented_gm, HammingLoss(gt_image.ravel())

# very simple helper class to combine things
class Dataset(object):
    def __init__(self, models_train, models_test, weights):
        self.models_train = models_train
        self.models_test = models_test
        self.weights = weights

# Subgradient SSVM
# ========================
#
# Instead of a cutting plane approach, we use a subgradient decent to find
    the optimal weights
#

def subgradient_ssvm(dataset, mode='gp', n_iter=20, learning_rate=1.0, c
    =0.5, lower_bounds=None, upper_bounds=None, convergence=0.001):

    loss_train = []
    loss_test = []

    weights = dataset.weights
    n = len(dataset.models_train)

    if lower_bounds is None:
        lower_bounds = numpy.ones(len(weights))*-1.0*float('inf')
```

```
if upper_bounds is None:
    upper_bounds = numpy.ones(len(weights))*float('inf')

do_opt = True
for iteration in range(n_iter):

    effective_learning_rate = learning_rate*float(learning_rate)/(1.0+
        iteration)

    # compute gradient
    diff = numpy.zeros(weights.shape)
    for gm, gm_loss_augmented, loss_function in dataset.models_train:

        # update the weights to the current weight vector
        gm.change_weights(weights)
        gm_loss_augmented.change_weights(weights)

        # the gt vector
        y_true = loss_function.y_true


        # optimize loss augmented / find most violated constraint
        if mode == 'icm':
            icm = IteratedConditionalModes(model=gm_loss_augmented)
            y_hat = icm.optimize()
        else:
            graphcut = GraphCut(model=gm_loss_augmented)
            y_hat = graphcut.optimize()

        # compute joint feature vector
        phi_y_hat  = gm.phi(y_hat)
        phi_y_true = gm.phi(y_true)

        diff +=  phi_y_true - phi_y_hat



    new_weights = weights - effective_learning_rate*(c/n)*diff

    # project new weights
    where_to_large = numpy.where(new_weights>upper_bounds)
    new_weights[where_to_large]  = upper_bounds[where_to_large]
    where_to_small = numpy.where(new_weights<lower_bounds)
    new_weights[where_to_small]  = lower_bounds[where_to_small]


    delta = numpy.abs(new_weights-weights).sum()
    if(delta<convergence):
        print("converged")
        break
    print('iter',iteration, 'delta',delta,"  ",numpy.round(new_weights
        ,3))
```

```
        weights = new_weights


        # ==========
        # show performance during training
        # ==========

        losses = []
        for gm,_,loss_function in dataset.models_train:
            gm.change_weights(weights)
            graphcut = GraphCut(model=gm)
            y_pred = graphcut.optimize()
            losses.append(loss_function(y_pred))
        loss_train.append(losses)
        print('training set performance', losses)

        losses = []
        for gm,_,loss_function in dataset.models_test:
            gm.change_weights(weights)
            graphcut = GraphCut(model=gm)
            y_pred = graphcut.optimize()
            losses.append(loss_function(y_pred))
        loss_test.append(losses)
        print('test set performance', losses)

    # in bonus mode: make a plot of performance
    if mode=='bonus':
        plottrain = numpy.array(loss_train).T
        plottest = numpy.array(loss_test).T
        for i in range(plottrain.shape[0]):
            plt.plot(range(n_iter),plottrain[i], lw=0.5, label=f'train img
                {i}')
            plt.plot(range(n_iter),plottest[i], lw=0.5, label=f'test img {
                i}')
        plt.title('Performance during training')
        plt.legend()
        plt.show()


    return weights

noises = [1.5, 2.0, 2.5, 3.0, 3.5]
regularizers = [0.1, 0.5, 0.9, 5., 10.]
shape = (20,20)
# noise = 2.0

loss_train = []
loss_test = []

# ============
# CHOOSE MODE
```

```
# gp: use graphcut
# icm: use icm
# bonus: show a plot of the loss at every step of learning phase
# ============
modes = ['gp', 'icm', 'bonus']
mode = modes[2]

for noise in noises:

    l_noises = []

    # The Dataset
    # ============

    x_train , y_train =  make_toy_dataset(shape=shape, n_images=5, noise=
        noise)
    x_test  , y_test  =  make_toy_dataset(shape=shape, n_images=5, noise=
        noise)

    # Build the weighted models:
    # ============

    unary_features_0 = get_unary_features(x_train[0])
    potts_features_0 = get_potts_features(x_train[0])
    n_unary_features = unary_features_0.shape[2]
    n_potts_features = potts_features_0.shape[2]

    n_weights = n_unary_features + n_potts_features

    for c in regularizers:
        weights = numpy.zeros(n_weights)

        # build the graphical models
        models_train  = [build_model(x,y, weights) for x,y in zip(x_train,
             y_train)]
        models_test   = [build_model(x,y, weights) for x,y in zip(x_test,
            y_test)]


        # combine things in a dataset
        dset = Dataset(models_train, models_test, weights)


        # we want the regularizer 'beta' to be positive
        lower_bounds = numpy.ones(n_weights)*(-1.0*float('inf'))
        lower_bounds[n_unary_features:n_unary_features+n_potts_features] =
             0

        weights = subgradient_ssvm(dset,mode=mode,c=c,learning_rate=1.0,
            lower_bounds=lower_bounds, n_iter=100)
```

```
        # Training Set Performance:
        # =============

        print("learned weights",weights)

        for i,(gm,_,loss_function) in enumerate(models_train):
            gm.change_weights(weights)

            if mode == 'icm':
                icm = IteratedConditionalModes(model=gm)
                y_pred = icm.optimize()
            else:
                graphcut = GraphCut(model=gm)
                y_pred = graphcut.optimize()
            print('loss of train img ', i, ':', loss_function(y_pred))


        # Test set performance:
        # ============

        # losses for this c
        l_c = []

        for i,(gm,_,loss_function) in enumerate(models_test):
            gm.change_weights(weights)

            if mode == 'icm':
                icm = IteratedConditionalModes(model=gm)
                y_pred = icm.optimize()
            else:
                graphcut = GraphCut(model=gm)
                y_pred = graphcut.optimize()

            print('loss of test img', i, '=', loss_function(y_pred))
            l_c.append(loss_function(y_pred))

        l_noises.append(l_c)

    loss_test.append(l_noises)

print(numpy.array(loss_test))
```