CS 280 HW3 Mini Places Challenge

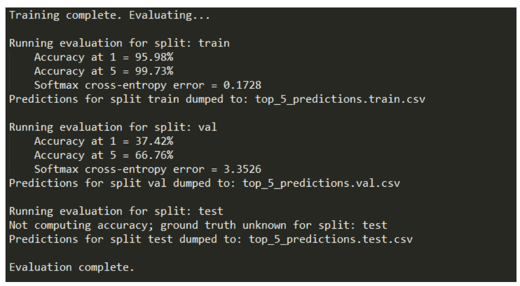
BetaGo: Jiaying Shi (24978491), Mo Zhou (21242515)

March 16, 2016

Best Model:

Our best model is a 17 layer convolutional neural network modified from VGG-16 network blablabla

The screenshot of the training log is shown as below:



```
Training complete. Evaluating...

Running evaluation for split: train
    Accuracy at 1 = 95.98%
    Accuracy at 5 = 99.73%
    Softmax cross-entropy error = 0.1728
Predictions for split train dumped to: top_5_predictions.train.csv

Running evaluation for split: val
    Accuracy at 1 = 37.42%
    Accuracy at 5 = 66.76%
    Softmax cross-entropy error = 3.3526
Predictions for split val dumped to: top_5_predictions.val.csv

Running evaluation for split: test
Not computing accuracy; ground truth unknown for split: test
Predictions for split test dumped to: top_5_predictions.test.csv

Evaluation complete.
```

From the above results we can see that in this case, the validation accuracy at 1 is about 37.42 while the accuracy at 5 is 66.76%. The training error is very small. That indicate in the best model we get, there are over fitting problems. We then tried to modified the network to reduce parameters but the accuracy on validation data set was not as good as this one. The Kaggle test score of this model is 0.37.

To analyze the results, we plotted the confusion matrices of of the validation data with the prediction with highest probability and the top 5 predictions. For the top 5 labels, if the top 5 predictions contains the true label, we regard it as an accurate prediction. Otherwise, we compare the true label with the prediction with highest probability.

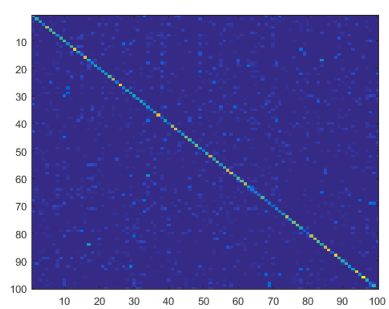


Figure 1: Confusion matrix of prediction with highest prediction

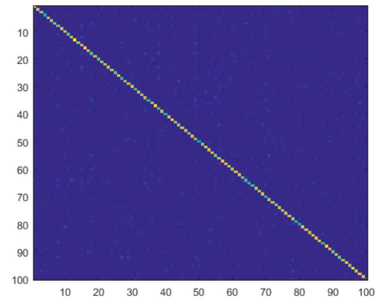


Figure 2: Confusion Matrix of top 5 predictions

From the above figures we can see that the top 5 predictions has much higher probability of covering the true label. For the first figure, the value of off-diagonal dots on row i column j represents the number images of class i are classified as class j. We take some of the wrong prediction and try to figure out what categories are easily misclassified. Taking category 29 labeled as caynon as an example, 13 images in the validation set is classified as "badlands". If we take a look at the two categories in the training data set we can see that the two categories are pretty similar. The following figures shows some images randomly chosen from the two categories in the training data set.

Figure 3: Examples of category "badlands"

Figure 4: Examples of category "canyon"

Even people may misclassify these two classes since they share similar "features". Different people may have different definitions for the two categories. And labels of the training data set are made up by people, we can totally rule out this effect. The misclassfication of the two categories actually reflects that the model can cluster the two categories together and it can actually identify that these two categories share common features.

Analysis:

The baseline train accuracy is obtained by running the original code, where we get:
train: Accuracy at 1 = 44.25%; Accuracy at 5 = 74.27%; Softmax cross-entropy error = 2.0991
validate: Accuracy at 1 = 34.25%; Accuracy at 5 = 64.27%; Softmax cross-entropy error = 2.6122

1 Layer Effect

1.1 Remove a convolution layer

Notice that the training accuracy is much better than the validation accuracy, we suspect that the model is overfitting. Thus we first tried to delete convolution layer 4 from the AlexNet. The resulting net has 4 convolution layers, all with ReLu and 3 with pooling, followed by 3 FC layers. This net, however, generates accuracy slightly worse than the baseline:
train: Accuracy at 1 = 41.63%; Accuracy at 5 = 72.08%; Softmax cross-entropy error = 2.2216
validate: Accuracy at 1 = 33.89%; Accuracy at 5 = 63.53%; Softmax cross-entropy error = 2.6298

1.2 Add a convolution layer

Observing that removing layers may not increase accuracy, we tried to add layers. We then add a convolution layer with ReLu and another pooling layer after the second pooling layer. The performance is however disappointing:
train: Accuracy at 1 = 37.25%; Accuracy at 5 = 67.90%; Softmax cross-entropy error = 2.3961
validate: Accuracy at 1 = 30.11%; Accuracy at 5 = 59.08%; Softmax cross-entropy error = 2.8144

1.3 Add a FC layer

We also tried adding a FC layer to the original network. It is added after FC7 and it also contains 1024 nodes. This net, however, generates accuracy slightly worse than the baseline:
train: Accuracy at 1 = 43.29%; Accuracy at 5 = 73.17%; Softmax cross-entropy error = 2.1340
validate: Accuracy at 1 = 33.29%; Accuracy at 5 = 63.14%; Softmax cross-entropy error = 2.6635
Notice that the training accuracy of this net is similar to the original net, however the validation error is greater. It might be due to overfitting of the added FC layer. Hence 3 FC layers seem to be appropriate for our dataset.

2 Parameter Effect

Notice that parameters can effect the outcomes greatly, we also experimented with different model parameters for better performance. By changing the crop size from 96 to 114, halfing the stepsize and setting the momentum from 0.9 to 0.8, our net generates accuracy slightly worse than the baseline:
train: Accuracy at 1 = 40.00%; Accuracy at 5 = 70.75%; Softmax cross-entropy error = 2.2839
validate: Accuracy at 1 = 31.42%; Accuracy at 5 = 61.60%; Softmax cross-entropy error = 2.7254

3 Fine-tuning

One effective approach to improve model accuracy is using fine-tuning because it has been demonstrated that the middle layers of a CNN usually holds general characteristics. Therefore, we used the resulting weights of a AlexNet trained on the ImageNet dataset as the intial weights of our model. The ImageNet model is trained on a outside dataset. Note that the FC layers in the two models have different number of nodes, so we discard the weights for the FC layers. As expected, Fine-tuning converges much faster than starting from scratch. We are able to achieve 40% training accuracy at 1 with only thousands of iterations. After 50000 iterations, we obtain:
train: Accuracy at 1 = 69.13%; Accuracy at 5 = 91.62%; Softmax cross-entropy error = 1.0797
validate: Accuracy at 1 = 44.34%; Accuracy at 5 = 73.68%; Softmax cross-entropy error = 2.2397
We observe that the validation error is much smaller than the other methods we tried, comfirming the effectiveness of fine-tuning. However, fine-tuning ues data ouside of the given train data set. We are not allowed to submit the test predictions from fine-tuning so we are not able to compare its performace on the test set ot the performance of other methods.

4 Deeper Nets

4.1 Variations of VGG

Observing that more convolution layers produces better result, we tried to use a deeper net to train our model. Three variations of the 16-layer VGG model is used: one orignal, one with 4 convolution layers less and one with 6

more convolution layers. We reduce the batch size and increase the step size to save memory. The performance of the original VGG has best performance after 15000 iterations and is shown below:
train: Accuracy at 1 = 95.98%; Accuracy at 5 = 99.73%; Softmax cross-entropy error = 0.1728
validate: Accuracy at 1 = 37.42%; Accuracy at 5 = 66.76%; Softmax cross-entropy error = 3.3526
We observe significant overfitting of the model. However, the validation error is still slightly better than the results from AlexNet.

4.2 Variations of GoogLeNet

We also implemented the GoogLeNet on our miniplaces dataset. GoogLeNet has 22 layers with more convolution layers than VGG. It requires more memory and time to run. To save on memory, we reduce the batch size to 64 and increase the iteration size to 100. It turns out the training error jumps greatly between consecutive iterations even after 30000 iterations. We therefore reduce the iteration size in the hope of a more robust model. The resulting model with 30000 iterations has the following performance:
train: Accuracy at 1 = 21.56% Accuracy at 5 = 51.17% Softmax cross-entropy error = 3.0915 Predictions for split train dumped to: top_5_predictions.train.csv
validate: Accuracy at 1 = 20.24% Accuracy at 5 = 49.99% Softmax cross-entropy error = 3.1386 Predictions for split val dumped to: top_5_predictions.val.csv
Since the loss improves pretty slow for a big model like this and we do not have good enough resource and time to train the model with more iterations, though we suspect that this model will perform better but the accuracy at 30000 iteration is the worst one.

5 Visualizations

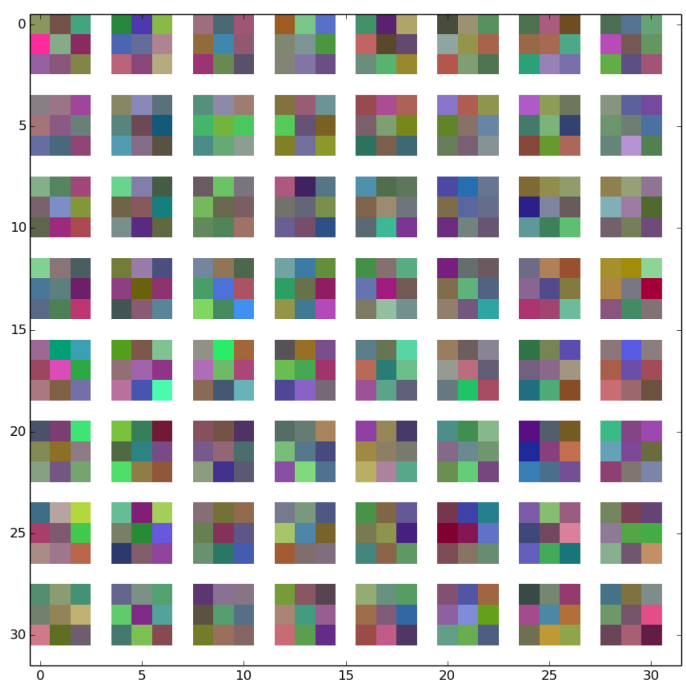


Figure 5: Filters in conv1_1

Figure 5 shows the first layer of the filters, conv1_1 in our best model. The filters are 3×3. They present very general features that are Gabor-like in some sense. We also observe that each subimage has a somewhat general shape and color which represent the building blocks of the images.

Appendix: Code

```
# train_places_net_conv.py
# Add one convolution layer
#!/usr/bin/env python

from __future__ import division
import argparse
import numpy as np
import os
import tempfile
import time

parser = argparse.ArgumentParser(
    description='Train and evaluate a net on the MI
T mini-places dataset.')
parser.add_argument('--image_root', default='./imag
es/',
    help='Directory where images are stored')
#crop size oritinally 96
parser.add_argument('--crop', type=int, default=114
,
    help=('The edge length of the random image crop
s'
          '(defaults to 96 for 96x96 crops)'))
parser.add_argument('--disp', type=int, default=10,

    help='Print loss/accuracy every --disp training
 iterations')
parser.add_argument('--snapshot_dir', default='./sn
apshot',
    help='Path to directory where snapshots are sav
ed')
parser.add_argument('--snapshot_prefix', default='p
lace_net',
    help='Snapshot filename prefix')
parser.add_argument('--iters', type=int, default=50
*1000,
    help='Total number of iterations to train the n
etwork')
parser.add_argument('--batch', type=int, default=25
6,
    help='The batch size to use for training')
parser.add_argument('--iter_size', type=int, defaul
```

```
t=1,
    help=('The number of iterations (batches) over
which to average the '
          'gradient computation. Effectively increa
ses the batch size '
          '(--batch) by this factor, but without in
creasing memory use '))
parser.add_argument('--lr', type=float, default=0.0
1,
    help='The initial learning rate')
parser.add_argument('--gamma', type=float, default=
0.1,
    help='Factor by which to drop the learning rate
')
# half of origianl stepsize
parser.add_argument('--stepsize', type=int, default
=50*100,
    help='Drop the learning rate every N iters -- t
his specifies N')
#momentum originally 0.9
parser.add_argument('--momentum', type=float, defau
lt=0.8,
    help='The momentum hyperparameter to use for mo
mentum SGD')
parser.add_argument('--decay', type=float, default=
5e-4,
    help='The L2 weight decay coefficient')
parser.add_argument('--seed', type=int, default=1,
    help='Seed for the random number generator')
parser.add_argument('--cudnn', action='store_true',

    help='Use CuDNN at training time -- usually fas
ter, but non-deterministic')
parser.add_argument('--gpu', type=int, default=0,
    help='GPU ID to use for training and inference
(-1 for CPU)')
args = parser.parse_args()

# disable most Caffe logging (unless env var $GLOG_
minloglevel is already set)
key = 'GLOG_minloglevel'
if not os.environ.get(key, ''):
```

```
    os.environ[key] = '3'

import caffe
from caffe.proto import caffe_pb2
from caffe import layers as L
from caffe import params as P

if args.gpu >= 0:
    caffe.set_mode_gpu()
    caffe.set_device(args.gpu)
else:
    caffe.set_mode_cpu()

def to_tempfile(file_content):
    """Serialize a Python protobuf object str(proto
), dump to a temporary file,
       and return its filename."""
    with tempfile.NamedTemporaryFile(delete=False)
as f:
        f.write(file_content)
        return f.name

weight_param = dict(lr_mult=1, decay_mult=1)
bias_param   = dict(lr_mult=2, decay_mult=0)
learned_param = [weight_param, bias_param]
frozen_param = [dict(lr_mult=0)] * 2

zero_filler     = dict(type='constant', value=0)
msra_filler     = dict(type='msra')
uniform_filler  = dict(type='uniform', min=-0.1, ma
x=0.1)
fc_filler       = dict(type='gaussian', std=0.005)
# Original AlexNet used the following commented out
 Gaussian initialization;
# we'll use the "MSRA" one instead, which scales th
e Gaussian initialization
# of a convolutional filter based on its receptive
field size.
# conv_filler     = dict(type='gaussian', std=0.01)

conv_filler     = dict(type='msra')
```

```
def conv_relu(bottom, ks, nout, stride=1, pad=0, gr
oup=1,
              param=learned_param,
              weight_filler=conv_filler, bias_fille
r=zero_filler,
              train=False):
    # set CAFFE engine to avoid CuDNN convolution -
- non-deterministic results
    engine = {}
    if train and not args.cudnn:
        engine.update(engine=P.Pooling.CAFFE)
    conv = L.Convolution(bottom, kernel_size=ks, st
ride=stride,
                         num_output=nout, pad=pad,
group=group, param=param,
                         weight_filler=weight_fille
r, bias_filler=bias_filler,
                         **engine)
    return conv, L.ReLU(conv, in_place=True)

def fc_relu(bottom, nout, param=learned_param,
            weight_filler=fc_filler, bias_filler=ze
ro_filler):
    fc = L.InnerProduct(bottom, num_output=nout, pa
ram=param,
                        weight_filler=weight_filler
, bias_filler=bias_filler)
    return fc, L.ReLU(fc, in_place=True)

def max_pool(bottom, ks, stride=1, train=False):
    # set CAFFE engine to avoid CuDNN pooling -- no
n-deterministic results
    engine = {}
    if train and not args.cudnn:
        engine.update(engine=P.Pooling.CAFFE)
    return L.Pooling(bottom, pool=P.Pooling.MAX, ke
rnel_size=ks, stride=stride,
                     **engine)

def minialexnet(data, labels=None, train=False, par
am=learned_param,
                num_classes=100, with_labels=True):
```

```
    """
    Returns a protobuf text file specifying a varia
nt of AlexNet, following the
    original specification (<caffe>/models/bvlc_ale
xnet/train_val.prototxt).
    The changes with respect to the original AlexNe
t are:
        - LRN (local response normalization) layers
 are not included
        - The Fully Connected (FC) layers (fc6 and
fc7) have smaller dimensions
          due to the lower resolution of mini-place
s images (128x128) compared
          with ImageNet images (usually resized to
256x256)
    """
    n = caffe.NetSpec()
    n.data = data
    conv_kwargs = dict(param=param, train=train)
    n.conv1, n.relu1 = conv_relu(n.data, 11, 96, st
ride=4, **conv_kwargs)
    n.pool1 = max_pool(n.relu1, 3, stride=2, train=
train)
    n.conv2, n.relu2 = conv_relu(n.pool1, 5, 256, p
ad=2, group=2, **conv_kwargs)
    n.pool2 = max_pool(n.relu2, 3, stride=2, train=
train)
    n.conv3, n.relu3 = conv_relu(n.pool2, 3, 384, p
ad=1, **conv_kwargs)
    n.conv3_1, n.relu3_1 = conv_relu(n.relu3, 3, 38
4, pad=1, **conv_kwargs)
    n.pool3 = max_pool(n.relu3_1, 3, stride=2, trai
n=train)


    n.conv4, n.relu4 = conv_relu(n.pool3, 3, 384, p
ad=1, group=2, **conv_kwargs)
    n.conv5, n.relu5 = conv_relu(n.relu4, 3, 256, p
ad=1, group=2, **conv_kwargs)
    n.pool5 = max_pool(n.relu5, 3, stride=2, train=
train)
```

```
    n.fc6, n.relu6 = fc_relu(n.pool5, 1024, param=p
aram)
    n.drop6 = L.Dropout(n.relu6, in_place=True)
    n.fc7, n.relu7 = fc_relu(n.drop6, 1024, param=p
aram)
    n.drop7 = L.Dropout(n.relu7, in_place=True)
    preds = n.fc8 = L.InnerProduct(n.drop7, num_out
put=num_classes, param=param)
    if not train:
        # Compute the per-label probabilities at te
st/inference time.
        preds = n.probs = L.Softmax(n.fc8)
    if with_labels:
        n.label = labels
        n.loss = L.SoftmaxWithLoss(n.fc8, n.label)
        n.accuracy_at_1 = L.Accuracy(preds, n.label
)
        n.accuracy_at_5 = L.Accuracy(preds, n.label
,
                                     accuracy_param
=dict(top_k=5))
    else:
        n.ignored_label = labels
        n.silence_label = L.Silence(n.ignored_label
, ntop=0)
    return to_tempfile(str(n.to_proto()))

def get_split(split):
    filename = './development_kit/data/%s.txt' % sp
lit
    if not os.path.exists(filename):
        raise IOError('Split data file not found: %
s' % split)
    return filename

def miniplaces_net(source, train=False, with_labels
=True):
    mean = [104, 117, 123]  # per-channel mean of t
he BGR image pixels
    transform_param = dict(mirror=train, crop_size=
args.crop, mean_value=mean)
    batch_size = args.batch if train else 100
```

```
    places_data, places_labels = L.ImageData(transf
orm_param=transform_param,
        source=source, root_folder=args.image_root,
 shuffle=train,
        batch_size=batch_size, ntop=2)
    return minialexnet(data=places_data, labels=pla
ces_labels, train=train,
                       with_labels=with_labels)

def snapshot_prefix():
    return os.path.join(args.snapshot_dir, args.sna
pshot_prefix)

def snapshot_at_iteration(iteration):
    return '%s_iter_%d.caffemodel' % (snapshot_pref
ix(), iteration)

def miniplaces_solver(train_net_path, test_net_path
=None):
    s = caffe_pb2.SolverParameter()

    # Specify locations of the train and (maybe) te
st networks.
    s.train_net = train_net_path
    if test_net_path is not None:
        s.test_net.append(test_net_path)
        # Test after every 1000 training iterations
.
        s.test_interval = 1000
        # Set `test_iter` to test on 100 batches ea
ch time we test.
        # With test batch size 100, this covers the
 entire validation set of
        # 10K images (100 * 100 = 10K).
        s.test_iter.append(100)
    else:
        s.test_interval = args.iters + 1  # don't t
est during training

    # The number of batches over which to average t
he gradient.
    # Effectively boosts the training batch size by
```

```
 the given factor, without
    # affecting memory utilization.
    s.iter_size = args.iter_size

    # Solve using the stochastic gradient descent (
SGD) algorithm.
    # Other choices include 'Adam' and 'RMSProp'.
    s.type = 'SGD'

    # The following settings (base_lr, lr_policy, g
amma, stepsize, and max_iter),
    # define the following learning rate schedule:
    #   Iterations [  0, 20K) -> learning rate 0.01
   = base_lr
    #   Iterations [20K, 40K) -> learning rate 0.00
1  = base_lr * gamma
    #   Iterations [40K, 50K) -> learning rate 0.00
01 = base_lr * gamma^2

    # Set the initial learning rate for SGD.
    s.base_lr = args.lr

    # Set `lr_policy` to define how the learning ra
te changes during training.
    # Here, we 'step' the learning rate by multiply
ing it by a factor `gamma`
    # every `stepsize` iterations.
    s.lr_policy = 'step'
    s.gamma = args.gamma
    s.stepsize = args.stepsize

    # `max_iter` is the number of times to update t
he net (training iterations).
    s.max_iter = args.iters

    # Set other SGD hyperparameters. Setting a non-
zero `momentum` takes a
    # weighted average of the current gradient and
previous gradients to make
    # learning more stable. L2 weight decay regular
izes learning, to help
    # prevent the model from overfitting.
```

```
    s.momentum = args.momentum
    s.weight_decay = args.decay

    # Display the current training loss and accurac
y every `display` iterations.
    # This doesn't have an effect for Python traini
ng here as logging is
    # disabled by this script (see the GLOG_minlogl
evel setting).
    s.display = args.disp

    # Number of training iterations over which to s
mooth the displayed loss.
    # The summed loss value (Iteration N, loss = X)
 will be averaged,
    # but individual loss values (Train net output
#K: my_loss = X) won't be.
    s.average_loss = 10

    # Seed the RNG for deterministic results.
    # (May not be so deterministic if using CuDNN.)

    s.random_seed = args.seed

    # Snapshots are files used to store networks we
've trained.  Here, we'll
    # snapshot twice per learning rate step to the
location specified by the
    # --snapshot_dir and --snapshot_prefix args.
    s.snapshot = args.stepsize // 2
    s.snapshot_prefix = snapshot_prefix()

    # Create snapshot dir if it doesn't already exi
st.
    if not os.path.exists(args.snapshot_dir):
        os.makedirs(args.snapshot_dir)

    return to_tempfile(str(s))

def train_net(with_val_net=False):
    train_net_file = miniplaces_net(get_split('trai
n'), train=True)
```

```
    # Set with_val_net=True to test during training
.
    # Environment variable GLOG_minloglevel should
be set to 0 to display
    # Caffe output in this case; otherwise, the tes
t result will not be
    # displayed.
    if with_val_net:
        val_net_file = miniplaces_net(get_split('va
l'), train=False)
    else:
        val_net_file = None
    solver_file = miniplaces_solver(train_net_file,
 val_net_file)
    solver = caffe.get_solver(solver_file)
    outputs = sorted(solver.net.outputs)
    def str_output(output):
        value = solver.net.blobs[output].data
        if output.startswith('accuracy'):
            valstr = '%5.2f%%' % (100 * value, )
        else:
            valstr = '%6f' % value
        return '%s = %s' % (output, valstr)
    def disp_outputs(iteration, iter_pad_len=len(st
r(args.iters))):
        metrics = '; '.join(str_output(o) for o in
outputs)
        return 'Iteration %*d: %s' % (iter_pad_len,
 iteration, metrics)
    # We could just call `solver.solve()` rather th
an `step()`ing in a loop.
    # (If we hadn't set GLOG_minloglevel = 3 at the
 top of this file, Caffe
    # would display loss/accuracy information durin
g training.)
    previous_time = None
    for iteration in xrange(args.iters):
        solver.step(1)
        if (args.disp > 0) and (iteration % args.di
sp == 0):
            current_time = time.clock()
            if previous_time is None:
```

```
                benchmark = ''
            else:
                time_per_iter = (current_time - pre
vious_time) / args.disp
                benchmark = ' (%5f s/it)' % time_pe
r_iter
            previous_time = current_time
            print disp_outputs(iteration), benchmar
k
    # Print accuracy for last iteration.
    solver.net.forward()
    disp_outputs(args.iters)
    solver.net.save(snapshot_at_iteration(args.iter
s))

def eval_net(split, K=5):
    print 'Running evaluation for split:', split
    filenames = []
    labels = []
    split_file = get_split(split)
    with open(split_file, 'r') as f:
        for line in f.readlines():
            parts = line.split()
            assert 1 <= len(parts) <= 2, 'malformed
 line'
            filenames.append(parts[0])
            if len(parts) > 1:
                labels.append(int(parts[1]))
    known_labels = (len(labels) > 0)
    if known_labels:
        assert len(labels) == len(filenames)
    else:
        # create file with 'dummy' labels (all 0s)
        split_file = to_tempfile(''.join('%s 0\n' %
 name for name in filenames))
    test_net_file = miniplaces_net(split_file, trai
n=False, with_labels=False)
    weights_file = snapshot_at_iteration(args.iters
)
    net = caffe.Net(test_net_file, weights_file, ca
ffe.TEST)
    top_k_predictions = np.zeros((len(filenames), K
```

```
), dtype=np.int32)
    if known_labels:
        correct_label_probs = np.zeros(len(filename
s))
    offset = 0
    while offset < len(filenames):
        probs = net.forward()['probs']
        for prob in probs:
            top_k_predictions[offset] = (-prob).arg
sort()[:K]
            if known_labels:
                correct_label_probs[offset] = prob[
labels[offset]]
            offset += 1
            if offset >= len(filenames):
                break
    if known_labels:
        def accuracy_at_k(preds, labels, k):
            assert len(preds) == len(labels)
            num_correct = sum(l in p[:k] for p, l i
n zip(preds, labels))
            return num_correct / len(preds)
        for k in [1, K]:
            accuracy = 100 * accuracy_at_k(top_k_pr
edictions, labels, k)
            print '\tAccuracy at %d = %4.2f%%' % (k
, accuracy)
        cross_ent_error = -np.log(correct_label_pro
bs).mean()
        print '\tSoftmax cross-entropy error = %.4f
' % (cross_ent_error, )
    else:
        print 'Not computing accuracy; ground truth
 unknown for split:', split
    filename = 'top_%d_predictions.%s.csv' % (K, sp
lit)
    with open(filename, 'w') as f:
        f.write(','.join(['image'] + ['label%d' % i
 for i in range(1, K+1)]))
        f.write('\n')
        f.write(''.join('%s,%s\n' % (image, ','.joi
n(str(p) for p in preds))
```

```
                        for image, preds in zip(fil
enames, top_k_predictions)))
    print 'Predictions for split %s dumped to: %s'
% (split, filename)

if __name__ == '__main__':
    print 'Training net...\n'
    train_net()

    print '\nTraining complete. Evaluating...\n'
    for split in ('train', 'val', 'test'):
        eval_net(split)
        print
    print 'Evaluation complete.'
```

```
# add_fc.py
# add a convolution layer
#!/usr/bin/env python

from __future__ import division

import argparse
import numpy as np
import os
import tempfile
import time

parser = argparse.ArgumentParser(
    description='Train and evaluate a net on the MI
T mini-places dataset.')
parser.add_argument('--image_root', default='./imag
es/',
    help='Directory where images are stored')
parser.add_argument('--crop', type=int, default=96,

    help=('The edge length of the random image crop
s'
          '(defaults to 96 for 96x96 crops)'))
parser.add_argument('--disp', type=int, default=10,

    help='Print loss/accuracy every --disp training
 iterations')
```

```
parser.add_argument('--snapshot_dir', default='./sn
apshot',
    help='Path to directory where snapshots are sav
ed')
parser.add_argument('--snapshot_prefix', default='p
lace_net',
    help='Snapshot filename prefix')
parser.add_argument('--iters', type=int, default=50
*1000,
    help='Total number of iterations to train the n
etwork')
parser.add_argument('--batch', type=int, default=25
6,
    help='The batch size to use for training')
parser.add_argument('--iter_size', type=int, defaul
t=1,
    help=('The number of iterations (batches) over
which to average the '
          'gradient computation. Effectively increa
ses the batch size '
          '(--batch) by this factor, but without in
creasing memory use '))
parser.add_argument('--lr', type=float, default=0.0
1,
    help='The initial learning rate')
parser.add_argument('--gamma', type=float, default=
0.1,
    help='Factor by which to drop the learning rate
')
parser.add_argument('--stepsize', type=int, default
=10*1000,
    help='Drop the learning rate every N iters -- t
his specifies N')
parser.add_argument('--momentum', type=float, defau
lt=0.9,
    help='The momentum hyperparameter to use for mo
mentum SGD')
parser.add_argument('--decay', type=float, default=
5e-4,
    help='The L2 weight decay coefficient')
parser.add_argument('--seed', type=int, default=1,
    help='Seed for the random number generator')
```

```
parser.add_argument('--cudnn', action='store_true',

    help='Use CuDNN at training time -- usually fas
ter, but non-deterministic')
parser.add_argument('--gpu', type=int, default=0,
    help='GPU ID to use for training and inference
(-1 for CPU)')
args = parser.parse_args()

# disable most Caffe logging (unless env var $GLOG_
minloglevel is already set)
key = 'GLOG_minloglevel'
if not os.environ.get(key, ''):
    os.environ[key] = '3'

import caffe
from caffe.proto import caffe_pb2
from caffe import layers as L
from caffe import params as P

if args.gpu >= 0:
    caffe.set_mode_gpu()
    caffe.set_device(args.gpu)
else:
    caffe.set_mode_cpu()

def to_tempfile(file_content):
    """Serialize a Python protobuf object str(proto
), dump to a temporary file,
       and return its filename."""
    with tempfile.NamedTemporaryFile(delete=False)
as f:
        f.write(file_content)
        return f.name

weight_param = dict(lr_mult=1, decay_mult=1)
bias_param   = dict(lr_mult=2, decay_mult=0)
learned_param = [weight_param, bias_param]
frozen_param = [dict(lr_mult=0)] * 2

zero_filler     = dict(type='constant', value=0)
msra_filler     = dict(type='msra')
```

```
uniform_filler  = dict(type='uniform', min=-0.1, ma
x=0.1)
fc_filler       = dict(type='gaussian', std=0.005)
# Original AlexNet used the following commented out
 Gaussian initialization;
# we'll use the "MSRA" one instead, which scales th
e Gaussian initialization
# of a convolutional filter based on its receptive
field size.
# conv_filler     = dict(type='gaussian', std=0.01)

conv_filler     = dict(type='msra')

def conv_relu(bottom, ks, nout, stride=1, pad=0, gr
oup=1,
              param=learned_param,
              weight_filler=conv_filler, bias_fille
r=zero_filler,
              train=False):
    # set CAFFE engine to avoid CuDNN convolution -
- non-deterministic results
    engine = {}
    if train and not args.cudnn:
        engine.update(engine=P.Pooling.CAFFE)
    conv = L.Convolution(bottom, kernel_size=ks, st
ride=stride,
                         num_output=nout, pad=pad,
group=group, param=param,
                         weight_filler=weight_fille
r, bias_filler=bias_filler,
                         **engine)
    return conv, L.ReLU(conv, in_place=True)

def fc_relu(bottom, nout, param=learned_param,
            weight_filler=fc_filler, bias_filler=ze
ro_filler):
    fc = L.InnerProduct(bottom, num_output=nout, pa
ram=param,
                        weight_filler=weight_filler
, bias_filler=bias_filler)
    return fc, L.ReLU(fc, in_place=True)
```

```
def max_pool(bottom, ks, stride=1, train=False):
    # set CAFFE engine to avoid CuDNN pooling -- no
n-deterministic results
    engine = {}
    if train and not args.cudnn:
        engine.update(engine=P.Pooling.CAFFE)
    return L.Pooling(bottom, pool=P.Pooling.MAX, ke
rnel_size=ks, stride=stride,
                     **engine)

def minialexnet(data, labels=None, train=False, par
am=learned_param,
                num_classes=100, with_labels=True):

    """
    Returns a protobuf text file specifying a varia
nt of AlexNet, following the
    original specification (<caffe>/models/bvlc_ale
xnet/train_val.prototxt).
    The changes with respect to the original AlexNe
t are:
        - LRN (local response normalization) layers
 are not included
        - The Fully Connected (FC) layers (fc6 and
fc7) have smaller dimensions
          due to the lower resolution of mini-place
s images (128x128) compared
          with ImageNet images (usually resized to
256x256)
    """
    n = caffe.NetSpec()
    n.data = data
    conv_kwargs = dict(param=param, train=train)
    n.conv1, n.relu1 = conv_relu(n.data, 11, 96, st
ride=4, **conv_kwargs)
    n.pool1 = max_pool(n.relu1, 3, stride=2, train=
train)
    n.conv2, n.relu2 = conv_relu(n.pool1, 5, 256, p
ad=2, group=2, **conv_kwargs)
    n.pool2 = max_pool(n.relu2, 3, stride=2, train=
train)
    n.conv3, n.relu3 = conv_relu(n.pool2, 3, 384, p
```

```
ad=1, **conv_kwargs)
    n.conv4, n.relu4 = conv_relu(n.relu3, 3, 384, p
ad=1, group=2, **conv_kwargs)
    n.conv5, n.relu5 = conv_relu(n.relu4, 3, 256, p
ad=1, group=2, **conv_kwargs)
    n.pool5 = max_pool(n.relu5, 3, stride=2, train=
train)
    n.fc6, n.relu6 = fc_relu(n.pool5, 1024, param=p
aram)
    n.drop6 = L.Dropout(n.relu6, in_place=True)
    n.fc7, n.relu7 = fc_relu(n.drop6, 1024, param=p
aram)
    n.drop7 = L.Dropout(n.relu7, in_place=True)
    n.fc77, n.relu77 = fc_relu(n.drop7, 1024, param
=param)
    n.drop7 = L.Dropout(n.relu77, in_place=True)

    preds = n.fc8 = L.InnerProduct(n.drop7, num_out
put=num_classes, param=param)
    if not train:
        # Compute the per-label probabilities at te
st/inference time.
        preds = n.probs = L.Softmax(n.fc8)
    if with_labels:
        n.label = labels
        n.loss = L.SoftmaxWithLoss(n.fc8, n.label)
        n.accuracy_at_1 = L.Accuracy(preds, n.label
)
        n.accuracy_at_5 = L.Accuracy(preds, n.label
,
                                     accuracy_param
=dict(top_k=5))
    else:
        n.ignored_label = labels
        n.silence_label = L.Silence(n.ignored_label
, ntop=0)
    return to_tempfile(str(n.to_proto()))

def get_split(split):
    filename = './development_kit/data/%s.txt' % sp
lit
    if not os.path.exists(filename):
```

```
        raise IOError('Split data file not found: %
s' % split)
    return filename

def miniplaces_net(source, train=False, with_labels
=True):
    mean = [104, 117, 123]  # per-channel mean of t
he BGR image pixels
    transform_param = dict(mirror=train, crop_size=
args.crop, mean_value=mean)
    batch_size = args.batch if train else 100
    places_data, places_labels = L.ImageData(transf
orm_param=transform_param,
        source=source, root_folder=args.image_root,
 shuffle=train,
        batch_size=batch_size, ntop=2)
    return minialexnet(data=places_data, labels=pla
ces_labels, train=train,
                       with_labels=with_labels)

def snapshot_prefix():
    return os.path.join(args.snapshot_dir, args.sna
pshot_prefix)

def snapshot_at_iteration(iteration):
    return '%s_iter_%d.caffemodel' % (snapshot_pref
ix(), iteration)

def miniplaces_solver(train_net_path, test_net_path
=None):
    s = caffe_pb2.SolverParameter()

    # Specify locations of the train and (maybe) te
st networks.
    s.train_net = train_net_path
    if test_net_path is not None:
        s.test_net.append(test_net_path)
        # Test after every 1000 training iterations
.
        s.test_interval = 1000
        # Set `test_iter` to test on 100 batches ea
ch time we test.
```

```
        # With test batch size 100, this covers the
 entire validation set of
        # 10K images (100 * 100 = 10K).
        s.test_iter.append(100)
    else:
        s.test_interval = args.iters + 1  # don't t
est during training

    # The number of batches over which to average t
he gradient.
    # Effectively boosts the training batch size by
 the given factor, without
    # affecting memory utilization.
    s.iter_size = args.iter_size

    # Solve using the stochastic gradient descent (
SGD) algorithm.
    # Other choices include 'Adam' and 'RMSProp'.
    s.type = 'SGD'

    # The following settings (base_lr, lr_policy, g
amma, stepsize, and max_iter),
    # define the following learning rate schedule:
    #   Iterations [  0, 20K) -> learning rate 0.01
   = base_lr
    #   Iterations [20K, 40K) -> learning rate 0.00
1  = base_lr * gamma
    #   Iterations [40K, 50K) -> learning rate 0.00
01 = base_lr * gamma^2

    # Set the initial learning rate for SGD.
    s.base_lr = args.lr

    # Set `lr_policy` to define how the learning ra
te changes during training.
    # Here, we 'step' the learning rate by multiply
ing it by a factor `gamma`
    # every `stepsize` iterations.
    s.lr_policy = 'step'
    s.gamma = args.gamma
    s.stepsize = args.stepsize
```

```
    # `max_iter` is the number of times to update t
he net (training iterations).
    s.max_iter = args.iters

    # Set other SGD hyperparameters. Setting a non-
zero `momentum` takes a
    # weighted average of the current gradient and 
previous gradients to make
    # learning more stable. L2 weight decay regular
izes learning, to help
    # prevent the model from overfitting.
    s.momentum = args.momentum
    s.weight_decay = args.decay

    # Display the current training loss and accurac
y every `display` iterations.
    # This doesn't have an effect for Python traini
ng here as logging is
    # disabled by this script (see the GLOG_minlogl
evel setting).
    s.display = args.disp

    # Number of training iterations over which to s
mooth the displayed loss.
    # The summed loss value (Iteration N, loss = X)
 will be averaged,
    # but individual loss values (Train net output 
#K: my_loss = X) won't be.
    s.average_loss = 10

    # Seed the RNG for deterministic results.
    # (May not be so deterministic if using CuDNN.)

    s.random_seed = args.seed

    # Snapshots are files used to store networks we
've trained.  Here, we'll
    # snapshot twice per learning rate step to the 
location specified by the
    # --snapshot_dir and --snapshot_prefix args.
    s.snapshot = args.stepsize // 2
    s.snapshot_prefix = snapshot_prefix()
```

```
    # Create snapshot dir if it doesn't already exi
st.
    if not os.path.exists(args.snapshot_dir):
        os.makedirs(args.snapshot_dir)

    return to_tempfile(str(s))

def train_net(with_val_net=False):
    train_net_file = miniplaces_net(get_split('trai
n'), train=True)
    # Set with_val_net=True to test during training
.
    # Environment variable GLOG_minloglevel should
be set to 0 to display
    # Caffe output in this case; otherwise, the tes
t result will not be
    # displayed.
    if with_val_net:
        val_net_file = miniplaces_net(get_split('va
l'), train=False)
    else:
        val_net_file = None
    solver_file = miniplaces_solver(train_net_file,
 val_net_file)
    solver = caffe.get_solver(solver_file)
    outputs = sorted(solver.net.outputs)
    def str_output(output):
        value = solver.net.blobs[output].data
        if output.startswith('accuracy'):
            valstr = '%5.2f%%' % (100 * value, )
        else:
            valstr = '%6f' % value
        return '%s = %s' % (output, valstr)
    def disp_outputs(iteration, iter_pad_len=len(st
r(args.iters))):
        metrics = '; '.join(str_output(o) for o in
outputs)
        return 'Iteration %*d: %s' % (iter_pad_len,
 iteration, metrics)
    # We could just call `solver.solve()` rather th
an `step()`ing in a loop.
```

```
    # (If we hadn't set GLOG_minloglevel = 3 at the
 top of this file, Caffe
    # would display loss/accuracy information durin
g training.)
    previous_time = None
    for iteration in xrange(args.iters):
        solver.step(1)
        if (args.disp > 0) and (iteration % args.di
sp == 0):
            current_time = time.clock()
            if previous_time is None:
                benchmark = ''
            else:
                time_per_iter = (current_time - pre
vious_time) / args.disp
                benchmark = ' (%5f s/it)' % time_pe
r_iter
            previous_time = current_time
            print disp_outputs(iteration), benchmar
k
    # Print accuracy for last iteration.
    solver.net.forward()
    disp_outputs(args.iters)
    solver.net.save(snapshot_at_iteration(args.iter
s))

def eval_net(split, K=5):
    print 'Running evaluation for split:', split
    filenames = []
    labels = []
    split_file = get_split(split)
    with open(split_file, 'r') as f:
        for line in f.readlines():
            parts = line.split()
            assert 1 <= len(parts) <= 2, 'malformed
 line'
            filenames.append(parts[0])
            if len(parts) > 1:
                labels.append(int(parts[1]))
    known_labels = (len(labels) > 0)
    if known_labels:
        assert len(labels) == len(filenames)
```

```
    else:
        # create file with 'dummy' labels (all 0s)
        split_file = to_tempfile(''.join('%s 0\n' %
 name for name in filenames))
    test_net_file = miniplaces_net(split_file, trai
n=False, with_labels=False)
    weights_file = snapshot_at_iteration(args.iters
)
    net = caffe.Net(test_net_file, weights_file, ca
ffe.TEST)
    top_k_predictions = np.zeros((len(filenames), K
), dtype=np.int32)
    if known_labels:
        correct_label_probs = np.zeros(len(filename
s))
    offset = 0
    while offset < len(filenames):
        probs = net.forward()['probs']
        for prob in probs:
            top_k_predictions[offset] = (-prob).arg
sort()[:K]
            if known_labels:
                correct_label_probs[offset] = prob[
labels[offset]]
            offset += 1
            if offset >= len(filenames):
                break
    if known_labels:
        def accuracy_at_k(preds, labels, k):
            assert len(preds) == len(labels)
            num_correct = sum(l in p[:k] for p, l i
n zip(preds, labels))
            return num_correct / len(preds)
        for k in [1, K]:
            accuracy = 100 * accuracy_at_k(top_k_pr
edictions, labels, k)
            print '\tAccuracy at %d = %4.2f%%' % (k
, accuracy)
        cross_ent_error = -np.log(correct_label_pro
bs).mean()
        print '\tSoftmax cross-entropy error = %.4f
' % (cross_ent_error, )
```

```
    else:
        print 'Not computing accuracy; ground truth
 unknown for split:', split
    filename = 'top_%d_predictions.%s.csv' % (K, sp
lit)
    with open(filename, 'w') as f:
        f.write(','.join(['image'] + ['label%d' % i
 for i in range(1, K+1)]))
        f.write('\n')
        f.write(''.join('%s,%s\n' % (image, ','.joi
n(str(p) for p in preds))
                        for image, preds in zip(fil
enames, top_k_predictions)))
    print 'Predictions for split %s dumped to: %s'
% (split, filename)

if __name__ == '__main__':
    print 'Training net...\n'
    train_net()

    print '\nTraining complete. Evaluating...\n'
    for split in ('train', 'val', 'test'):
        eval_net(split)
        print
    print 'Evaluation complete.'
```

```
# fine_tune.py
# fine-tuning using the imageNet weights
#!/usr/bin/env python

from __future__ import division

import argparse
import numpy as np
import os
import tempfile
import time


parser = argparse.ArgumentParser(
    description='Train and evaluate a net on the MI
T mini-places dataset.')
```

```
parser.add_argument('--image_root', default='./imag
es/',
    help='Directory where images are stored')
parser.add_argument('--crop', type=int, default=96,

    help=('The edge length of the random image crop
s'
          '(defaults to 96 for 96x96 crops)'))
parser.add_argument('--disp', type=int, default=10,

    help='Print loss/accuracy every --disp training
 iterations')
parser.add_argument('--snapshot_dir', default='./sn
apshot',
    help='Path to directory where snapshots are sav
ed')
parser.add_argument('--snapshot_prefix', default='p
lace_net',
    help='Snapshot filename prefix')
parser.add_argument('--iters', type=int, default=50
*1000,
    help='Total number of iterations to train the n
etwork')
parser.add_argument('--batch', type=int, default=25
6,
    help='The batch size to use for training')
parser.add_argument('--iter_size', type=int, defaul
t=1,
    help=('The number of iterations (batches) over
which to average the '
          'gradient computation. Effectively increa
ses the batch size '
          '(--batch) by this factor, but without in
creasing memory use '))
parser.add_argument('--lr', type=float, default=0.0
1,
    help='The initial learning rate')
parser.add_argument('--gamma', type=float, default=
0.1,
    help='Factor by which to drop the learning rate
')
parser.add_argument('--stepsize', type=int, default
```

```
=10*1000,
    help='Drop the learning rate every N iters -- t
his specifies N')
parser.add_argument('--momentum', type=float, defau
lt=0.9,
    help='The momentum hyperparameter to use for mo
mentum SGD')
parser.add_argument('--decay', type=float, default=
5e-4,
    help='The L2 weight decay coefficient')
parser.add_argument('--seed', type=int, default=1,
    help='Seed for the random number generator')
parser.add_argument('--cudnn', action='store_true',

    help='Use CuDNN at training time -- usually fas
ter, but non-deterministic')
parser.add_argument('--gpu', type=int, default=0,
    help='GPU ID to use for training and inference
(-1 for CPU)')
args = parser.parse_args()

# disable most Caffe logging (unless env var $GLOG_
minloglevel is already set)
key = 'GLOG_minloglevel'
if not os.environ.get(key, ''):
    os.environ[key] = '3'

import os
weights = 'bvlc_reference_caffenet.caffemodel'
assert os.path.exists(weights)


import caffe
from caffe.proto import caffe_pb2
from caffe import layers as L
from caffe import params as P

if args.gpu >= 0:
    caffe.set_mode_gpu()
    caffe.set_device(args.gpu)
else:
    caffe.set_mode_cpu()
```

```
def to_tempfile(file_content):
    """Serialize a Python protobuf object str(proto
), dump to a temporary file,
       and return its filename."""
    with tempfile.NamedTemporaryFile(delete=False)
as f:
        f.write(file_content)
        return f.name

weight_param = dict(lr_mult=1, decay_mult=1)
bias_param   = dict(lr_mult=2, decay_mult=0)
learned_param = [weight_param, bias_param]
frozen_param = [dict(lr_mult=0)] * 2

zero_filler     = dict(type='constant', value=0)
msra_filler     = dict(type='msra')
uniform_filler  = dict(type='uniform', min=-0.1, ma
x=0.1)
fc_filler       = dict(type='gaussian', std=0.005)
# Original AlexNet used the following commented out
 Gaussian initialization;
# we'll use the "MSRA" one instead, which scales th
e Gaussian initialization
# of a convolutional filter based on its receptive
field size.
# conv_filler     = dict(type='gaussian', std=0.01)

conv_filler     = dict(type='msra')

def conv_relu(bottom, ks, nout, stride=1, pad=0, gr
oup=1,
              param=learned_param,
              weight_filler=conv_filler, bias_fille
r=zero_filler,
              train=False):
    # set CAFFE engine to avoid CuDNN convolution -
- non-deterministic results
    engine = {}
    if train and not args.cudnn:
        engine.update(engine=P.Pooling.CAFFE)
    conv = L.Convolution(bottom, kernel_size=ks, st
```

```
ride=stride,
                         num_output=nout, pad=pad,
group=group, param=param,
                         weight_filler=weight_fille
r, bias_filler=bias_filler,
                         **engine)
    return conv, L.ReLU(conv, in_place=True)

def fc_relu(bottom, nout, param=learned_param,
            weight_filler=fc_filler, bias_filler=ze
ro_filler):
    fc = L.InnerProduct(bottom, num_output=nout, pa
ram=param,
                        weight_filler=weight_filler
, bias_filler=bias_filler)
    return fc, L.ReLU(fc, in_place=True)

def max_pool(bottom, ks, stride=1, train=False):
    # set CAFFE engine to avoid CuDNN pooling -- no
n-deterministic results
    engine = {}
    if train and not args.cudnn:
        engine.update(engine=P.Pooling.CAFFE)
    return L.Pooling(bottom, pool=P.Pooling.MAX, ke
rnel_size=ks, stride=stride,
                     **engine)

def minialexnet(data, labels=None, train=False, par
am=learned_param,
                num_classes=100, with_labels=True):

    """
    Returns a protobuf text file specifying a varia
nt of AlexNet, following the
    original specification (<caffe>/models/bvlc_ale
xnet/train_val.prototxt).
    The changes with respect to the original AlexNe
t are:
        - LRN (local response normalization) layers
 are not included
        - The Fully Connected (FC) layers (fc6 and
fc7) have smaller dimensions
```

```
        due to the lower resolution of mini-place
s images (128x128) compared
        with ImageNet images (usually resized to
256x256)
    """
    n = caffe.NetSpec()
    n.data = data
    conv_kwargs = dict(param=param, train=train)
    n.conv1_1, n.relu1_1 = conv_relu(n.data, 3, 64,
 stride=1, **conv_kwargs)
    n.conv1_2, n.relu1_2 = conv_relu(n.relu1_1, 3,
64, stride=1, **conv_kwargs)
    n.pool1 = max_pool(n.relu1_2, 2, stride=2, trai
n=train)
    n.conv2_1, n.relu2_1 = conv_relu(n.pool1, 3, 12
8, pad=1, group=2, **conv_kwargs)
    n.conv2_2, n.relu2_2 = conv_relu(n.relu2_1, 3,
128, pad=1, group=2, **conv_kwargs)
    n.pool2 = max_pool(n.relu2_2, 2, stride=2, trai
n=train)
    n.conv3_1, n.relu3_1 = conv_relu(n.pool2, 3, 25
6, pad=1, **conv_kwargs)
    n.conv3_2, n.relu3_2 = conv_relu(n.relu3_1, 3,
256, pad=1, **conv_kwargs)
    n.conv3_3, n.relu3_3 = conv_relu(n.relu3_2, 3,
256, pad=1, **conv_kwargs)
    n.pool3 = max_pool(n.relu3_3, 2, stride=2, trai
n=train)
    n.conv4_1, n.relu4_1 = conv_relu(n.pool3, 3, 51
2, pad=1, group=2, **conv_kwargs)
    n.conv4_2, n.relu4_2 = conv_relu(n.relu4_1, 3,
512, pad=1, group=2, **conv_kwargs)
    n.conv4_3, n.relu4_3 = conv_relu(n.relu4_2, 3,
512, pad=1, group=2, **conv_kwargs)
    n.pool4 = max_pool(n.relu4_3, 2, stride=2, trai
n=train)
    n.conv5_1, n.relu5_1 = conv_relu(n.pool3, 3, 51
2, pad=1, group=2, **conv_kwargs)
    n.conv5_2, n.relu5_2 = conv_relu(n.relu5_1, 3,
512, pad=1, group=2, **conv_kwargs)
    n.conv5_3, n.relu5_3 = conv_relu(n.relu5_2, 3,
512, pad=1, group=2, **conv_kwargs)
```

```
    n.pool5 = max_pool(n.relu5_5, 2, stride=2, trai
n=train)
    n.fc6, n.relu6 = fc_relu(n.pool5, 4096, param=p
aram)
    n.drop6 = L.Dropout(n.relu6, in_place=True)
    n.fc7, n.relu7 = fc_relu(n.drop6, 4096, param=p
aram)
    n.drop7 = L.Dropout(n.relu7, in_place=True)
    preds = n.fc8 = L.InnerProduct(n.drop7, num_out
put=num_classes, param=param)
    if not train:
        # Compute the per-label probabilities at te
st/inference time.
        preds = n.probs = L.Softmax(n.fc8)
    if with_labels:
        n.label = labels
        n.loss = L.SoftmaxWithLoss(n.fc8, n.label)
        n.accuracy_at_1 = L.Accuracy(preds, n.label
)
        n.accuracy_at_5 = L.Accuracy(preds, n.label
,
                                     accuracy_param
=dict(top_k=5))
    else:
        n.ignored_label = labels
        n.silence_label = L.Silence(n.ignored_label
, ntop=0)
    return to_tempfile(str(n.to_proto()))

def get_split(split):
    filename = './development_kit/data/%s.txt' % sp
lit
    if not os.path.exists(filename):
        raise IOError('Split data file not found: %
s' % split)
    return filename

def miniplaces_net(source, train=False, with_labels
=True):
    mean = [104, 117, 123]  # per-channel mean of t
he BGR image pixels
    transform_param = dict(mirror=train, crop_size=
```

```
args.crop, mean_value=mean)
    batch_size = args.batch if train else 100
    places_data, places_labels = L.ImageData(transf
orm_param=transform_param,
        source=source, root_folder=args.image_root,
 shuffle=train,
        batch_size=batch_size, ntop=2)
    return minialexnet(data=places_data, labels=pla
ces_labels, train=train,
                       with_labels=with_labels)

def snapshot_prefix():
    return os.path.join(args.snapshot_dir, args.sna
pshot_prefix)

def snapshot_at_iteration(iteration):
    return '%s_iter_%d.caffemodel' % (snapshot_pref
ix(), iteration)

def miniplaces_solver(train_net_path, test_net_path
=None):
    s = caffe_pb2.SolverParameter()

    # Specify locations of the train and (maybe) te
st networks.
    s.train_net = train_net_path
    if test_net_path is not None:
        s.test_net.append(test_net_path)
        # Test after every 1000 training iterations
.
        s.test_interval = 1000
        # Set `test_iter` to test on 100 batches ea
ch time we test.
        # With test batch size 100, this covers the
 entire validation set of
        # 10K images (100 * 100 = 10K).
        s.test_iter.append(100)
    else:
        s.test_interval = args.iters + 1  # don't t
est during training

    # The number of batches over which to average t
```

```
he gradient.
    # Effectively boosts the training batch size by
 the given factor, without
    # affecting memory utilization.
    s.iter_size = args.iter_size

    # Solve using the stochastic gradient descent (
SGD) algorithm.
    # Other choices include 'Adam' and 'RMSProp'.
    s.type = 'SGD'

    # The following settings (base_lr, lr_policy, g
amma, stepsize, and max_iter),
    # define the following learning rate schedule:
    #   Iterations [  0, 20K) -> learning rate 0.01
   = base_lr
    #   Iterations [20K, 40K) -> learning rate 0.00
1  = base_lr * gamma
    #   Iterations [40K, 50K) -> learning rate 0.00
01 = base_lr * gamma^2

    # Set the initial learning rate for SGD.
    s.base_lr = args.lr

    # Set `lr_policy` to define how the learning ra
te changes during training.
    # Here, we 'step' the learning rate by multiply
ing it by a factor `gamma`
    # every `stepsize` iterations.
    s.lr_policy = 'step'
    s.gamma = args.gamma
    s.stepsize = args.stepsize

    # `max_iter` is the number of times to update t
he net (training iterations).
    s.max_iter = args.iters

    # Set other SGD hyperparameters. Setting a non-
zero `momentum` takes a
    # weighted average of the current gradient and
previous gradients to make
    # learning more stable. L2 weight decay regular
```

```
izes learning, to help
    # prevent the model from overfitting.
    s.momentum = args.momentum
    s.weight_decay = args.decay

    # Display the current training loss and accurac
y every `display` iterations.
    # This doesn't have an effect for Python traini
ng here as logging is
    # disabled by this script (see the GLOG_minlogl
evel setting).
    s.display = args.disp

    # Number of training iterations over which to s
mooth the displayed loss.
    # The summed loss value (Iteration N, loss = X)
 will be averaged,
    # but individual loss values (Train net output
#K: my_loss = X) won't be.
    s.average_loss = 10

    # Seed the RNG for deterministic results.
    # (May not be so deterministic if using CuDNN.)

    s.random_seed = args.seed

    # Snapshots are files used to store networks we
've trained.  Here, we'll
    # snapshot twice per learning rate step to the
location specified by the
    # --snapshot_dir and --snapshot_prefix args.
    s.snapshot = args.stepsize // 2
    s.snapshot_prefix = snapshot_prefix()

    # Create snapshot dir if it doesn't already exi
st.
    if not os.path.exists(args.snapshot_dir):
        os.makedirs(args.snapshot_dir)

    return to_tempfile(str(s))

def vis_square(data):
```

```
    """Take an array of shape (n, height, width) or
 (n, height, width, 3)
       and visualize each (height, width) thing in
 a grid of size approx. sqrt(n) by sqrt(n)"""

    # normalize data for display
    data = (data - data.min()) / (data.max() - data
.min())

    # force the number of filters to be square
    n = int(np.ceil(np.sqrt(data.shape[0])))
    padding = (((0, n ** 2 - data.shape[0]),
               (0, 1), (0, 1))                # ad
d some space between filters
               + ((0, 0),) * (data.ndim - 3))  # d
on't pad the last dimension (if there is one)
    data = np.pad(data, padding, mode='constant', c
onstant_values=1)  # pad with ones (white)

    # tile the filters into an image
    data = data.reshape((n, n) + data.shape[1:]).t
ranspose((0, 2, 1, 3) + tuple(range(4, data.ndim +
 1)))
    data = data.reshape((n * data.shape[1], n * dat
a.shape[3]) + data.shape[4:])

    plt.imsave('pool5.png',data); plt.axis('off')


def train_net(with_val_net=False):
    train_net_file = miniplaces_net(get_split('trai
n'), train=True)
    # Set with_val_net=True to test during training
.
    # Environment variable GLOG_minloglevel should
be set to 0 to display
    # Caffe output in this case; otherwise, the tes
t result will not be
    # displayed.
    if with_val_net:
        val_net_file = miniplaces_net(get_split('va
l'), train=False)
```

```
    else:
        val_net_file = None
    solver_file = miniplaces_solver(train_net_file,
 val_net_file)
    solver = caffe.get_solver(solver_file)
    solver.net.copy_from(weights)


    outputs = sorted(solver.net.outputs)
    def str_output(output):
        value = solver.net.blobs[output].data
        if output.startswith('accuracy'):
            valstr = '%5.2f%%' % (100 * value, )
        else:
            valstr = '%6f' % value
        return '%s = %s' % (output, valstr)
    def disp_outputs(iteration, iter_pad_len=len(st
r(args.iters))):
        metrics = '; '.join(str_output(o) for o in
outputs)
        return 'Iteration %*d: %s' % (iter_pad_len,
 iteration, metrics)
    # We could just call `solver.solve()` rather th
an `step()`ing in a loop.
    # (If we hadn't set GLOG_minloglevel = 3 at the
 top of this file, Caffe
    # would display loss/accuracy information durin
g training.)
    previous_time = None
    for iteration in xrange(args.iters):
        solver.step(1)
        if (args.disp > 0) and (iteration % args.di
sp == 0):
            current_time = time.clock()
            if previous_time is None:
                benchmark = ''
            else:
                time_per_iter = (current_time - pre
vious_time) / args.disp
                benchmark = ' (%5f s/it)' % time_pe
r_iter
            previous_time = current_time
```

```
            print disp_outputs(iteration), benchmar
k
    # Print accuracy for last iteration.
    solver.net.forward()
    disp_outputs(args.iters)
    solver.net.save(snapshot_at_iteration(args.iter
s))


def eval_net(split, K=5):
    print 'Running evaluation for split:', split
    filenames = []
    labels = []
    split_file = get_split(split)
    with open(split_file, 'r') as f:
        for line in f.readlines():
            parts = line.split()
            assert 1 <= len(parts) <= 2, 'malformed
 line'
            filenames.append(parts[0])
            if len(parts) > 1:
                labels.append(int(parts[1]))
    known_labels = (len(labels) > 0)
    if known_labels:
        assert len(labels) == len(filenames)
    else:
        # create file with 'dummy' labels (all 0s)
        split_file = to_tempfile(''.join('%s 0\n' %
 name for name in filenames))
    test_net_file = miniplaces_net(split_file, trai
n=False, with_labels=False)
    weights_file = snapshot_at_iteration(args.iters
)
    net = caffe.Net(test_net_file, weights_file, ca
ffe.TEST)

    top_k_predictions = np.zeros((len(filenames), K
), dtype=np.int32)
    if known_labels:
        correct_label_probs = np.zeros(len(filename
s))
    offset = 0
```

```
    while offset < len(filenames):
        probs = net.forward()['probs']
        for prob in probs:
            top_k_predictions[offset] = (-prob).arg
sort()[:K]
            if known_labels:
                correct_label_probs[offset] = prob[
labels[offset]]
            offset += 1
            if offset >= len(filenames):
                break
    if known_labels:
        def accuracy_at_k(preds, labels, k):
            assert len(preds) == len(labels)
            num_correct = sum(l in p[:k] for p, l i
n zip(preds, labels))
            return num_correct / len(preds)
        for k in [1, K]:
            accuracy = 100 * accuracy_at_k(top_k_pr
edictions, labels, k)
            print '\tAccuracy at %d = %4.2f%%' % (k
, accuracy)
        cross_ent_error = -np.log(correct_label_pro
bs).mean()
        print '\tSoftmax cross-entropy error = %.4f
' % (cross_ent_error, )
    else:
        print 'Not computing accuracy; ground truth
 unknown for split:', split
    filename = 'top_%d_predictions.%s.csv' % (K, sp
lit)
    with open(filename, 'w') as f:
        f.write(','.join(['image'] + ['label%d' % i
 for i in range(1, K+1)]))
        f.write('\n')
        f.write(''.join('%s,%s\n' % (image, ','.joi
n(str(p) for p in preds))
                        for image, preds in zip(fil
enames, top_k_predictions)))
    print 'Predictions for split %s dumped to: %s'
% (split, filename)
```

```
if __name__ == '__main__':
    print 'Training net...\n'
    train_net()

    print '\nTraining complete. Evaluating...\n'
    for split in ('train', 'val', 'test'):
        eval_net(split)
        print
    print 'Evaluation complete.'
```