run_train_distracted_drivers-maxout8-lr.2.steps

August 4, 2016

The changes here are: - Orthogonal weight initialization - Decreasing learning rate by step size

In [1]:
```
from skimage import io, transform, exposure, color, util
import os, itertools, sys
from PIL import Image
%pylab inline
sys.setrecursionlimit(1000000)
```

```
Populating the interactive namespace from numpy and matplotlib
```

In [2]:
```
# data_dir = "/home/dylan/IdeaProjects/distracted_drivers/train/"
data_dir =  "/media/dylan/Science/Kaggle-Data/distracted_drivers/train/"
```

In [3]:
```
input_volume_shape = (128, 128)
```

In [4]:
```
def read_img_file_PIL(file_path, size=(32,32)):
    img = Image.open(file_path).convert('L')
    img.thumbnail(size, Image.NEAREST)
    data = np.array(img)
    shape = data.shape
    append_top = int(ceil(max(0, size[0] - shape[0])/2.0))
    append_bot = int(floor(max(0, size[0] - shape[0])/2.0))
    data = util.pad(data, ((append_top, append_bot),
                           (0,0)), mode='constant', constant_values=0)
    return data
```

In [5]:
```
def read_img_file(file_path, rescale=0.01):
    img = io.imread(file_path)
    img= color.rgb2gray(img)
    return transform.rescale(img, rescale)
```

In [6]:
```
def image_gen_from_dir(directory, batch_size, num_categories, size=input_volume_shape):
    result = {os.path.join(dp, f) : int(os.path.split(dp)[1]) for dp, dn, filenames in os.walk(
                for f in filenames if os.path.splitext(f)[1] == '.jpg'}
    # infinite loop
    while True:
        image_files = []
        labels = []
        # randomly choose batch size samples in result
        for category in range(num_categories):
            file_samples = np.random.choice([k for k, v in result.iteritems() if v == category]
                            size=batch_size, replace=False)
            for file_sample in file_samples:
                image_files.append(read_img_file_PIL(file_sample, size=size))
            labels.extend([v for v in itertools.repeat(category, batch_size)])
```

```
            # end category loop
            X = np.asarray(image_files, dtype=np.float32)
            # -1 to 1 range
            X = exposure.rescale_intensity(X, out_range=(-1,1))
            y = np.asarray(labels, dtype=np.int32)
            yield X, y
```

0.1 Another loader, augmentation time

We'll do 6 augmentations:

```
1.) Translation up to 10 pixels
2.) Rotation up to 15 degrees
3.) Zooming
4.) JPEG compression
5.) Sharpening
6.) Gamma correction
```

We won't do flips since the dataset only contains images from the passenger seat. Perhaps we can revisit this later.

In [7]:
```
from skimage.transform import rotate, warp, AffineTransform
from skimage import filters
from scipy import ndimage, misc
import StringIO
```

In [8]:
```
def random_translate(img):
    shift_random = AffineTransform(translation=(randint(-10, 10), randint(-10, 10)))
    min_value = 0 if min(img.ravel()) > 0 else min(img.ravel())
    return np.float32(warp(img, shift_random, mode='constant', cval=min_value))

def random_rotate(img):
    min_value = 0 if min(img.ravel()) > 0 else min(img.ravel())
    return np.float32(rotate(img, randint(-15, 15), mode='constant', cval=min_value))

def random_zoom(img):
    min_value = 0 if min(img.ravel()) > 0 else min(img.ravel())
    scale_random = AffineTransform(scale=(uniform(0.9, 1.1), uniform(0.9, 1.1)))
    return np.float32(warp(img, scale_random, mode='constant', cval=min_value))

def random_compress(img):
    max_v = np.ceil(img.max())
    min_v = np.floor(img.min())
    nd_im = exposure.rescale_intensity(img, out_range=(0, 1)).squeeze()
    nd_im = np.ndarray.astype(nd_im * 255, np.uint8)
    # nd_im = np.ndarray.astype(img * 255, np.uint8)
    im = Image.fromarray(nd_im)
    buf = StringIO.StringIO()
    im.save(buf, "JPEG", quality=np.random.randint(95, 99))
    buf.seek(0)
    im2 = Image.open(buf)
    x1 = exposure.rescale_intensity(np.ndarray.astype(np.array(im2), np.float32), out_range=(mi
    return x1
```

```
        def random_sharpening(img):
            blurred_f = ndimage.gaussian_filter(img, 0.5)
            filter_blurred_f = ndimage.gaussian_filter(blurred_f, 1)
            alpha = uniform(0.9, 1.2)
            img = blurred_f + alpha * (blurred_f - filter_blurred_f)
            return exposure.rescale_intensity(img, out_range=(-1 , 1))

        def random_gamma_correction(img):
            max_v = np.ceil(img.max())
            min_v = np.floor(img.min())
            img = exposure.rescale_intensity(img, out_range=(0,1))
            img = exposure.adjust_gamma(img, uniform(0.2, 0.8))
            return exposure.rescale_intensity(img, out_range=(-1, 1))
```

```
In [9]: def random_aug(img):
            choice = np.random.randint(0,6)
            # choose from 4 different augmentations!
            if choice == 0:
                return random_translate(img)
            elif choice == 1:
                return random_rotate(img)
            elif choice == 2:
                return random_zoom(img)
            elif choice == 3:
                return random_compress(img)
            elif choice == 4:
                return random_sharpening(img)
            else:
                return random_gamma_correction(img)
```

```
In [10]: def random_aug_batch(X, aug_algorithm):
             for i in range(X.shape[0]):
                 X[i] = aug_algorithm(X[i])
             return X
```

```
In [11]: def random_aug_gen(gen, aug_algorithm):
             for batchX, batchY in gen:
                 yield random_aug_batch(batchX, aug_algorithm), batchY
```

1 Process Generator with cached elements

```
In [12]: def threaded_generator(generator, num_cached=50):
             import Queue
             queue = Queue.Queue(maxsize=num_cached)
             sentinel = object()  # guaranteed unique reference

             # define producer (putting items into queue)
             def producer():
                 for item in generator:
                     queue.put(item)
                 queue.put(sentinel)

             # start producer (in a background thread)
             import threading
```

```
            thread = threading.Thread(target=producer)
            thread.daemon = True
            thread.start()

            # run as consumer (read items from queue, in current thread)
            item = queue.get()
            while item is not sentinel:
                yield item
                queue.task_done()
                item = queue.get()
```

In [13]:
```
from nolearn.lasagne import NeuralNet
from lasagne.layers import DenseLayer, ReshapeLayer, Upscale2DLayer, Conv2DLayer, InputLayer,
    MaxPool2DLayer, get_all_params, batch_norm, BatchNormLayer, FeaturePoolLayer
import numpy as np
from lasagne.nonlinearities import softmax, leaky_rectify, theano
from lasagne.updates import nesterov_momentum
from nolearn.lasagne import NeuralNet, BatchIterator, PrintLayerInfo, objective
from nolearn.lasagne import TrainSplit
from common import EarlyStopping, EndTrainingFromEarlyStopping
from lasagne.objectives import categorical_crossentropy, aggregate
import cPickle as pickle
from sklearn import metrics
import time, logging, logging.config, logging.handlers
from lasagne.init import Orthogonal
from notebook_functions import load_best_weights
```

```
Couldn't import dot_parser, loading of dot files will not be possible.

Using gpu device 0: GeForce GTX 960 (CNMeM is disabled, CuDNN 4004)
```

def batch_norm(s): return s

In [14]:
```
try:
    from lasagne.layers.dnn import Conv2DDNNLayer, MaxPool2DDNNLayer
    def conv_2_layer_stack(top, num_filters):
        conv1 = batch_norm(Conv2DDNNLayer(top, num_filters, (3, 3),
                stride=1, pad=1, nonlinearity=leaky_rectify, W=Orthogonal()))
        conv2 = batch_norm(Conv2DDNNLayer(conv1, num_filters, (3, 3),
                stride=1, pad=1, nonlinearity=leaky_rectify, W=Orthogonal()))
        return MaxPool2DDNNLayer(conv2, (2, 2), 2)

    def conv_3_layer_stack(top, num_filters):
        conv1 = batch_norm(Conv2DDNNLayer(top, num_filters, (3, 3),
                    stride=1, pad=1, nonlinearity=leaky_rectify, W=Orthogonal()))
        conv2 = batch_norm(Conv2DDNNLayer(conv1, num_filters, (3, 3),
                    stride=1, pad=1, nonlinearity=leaky_rectify, W=Orthogonal()))
        conv3 = batch_norm(Conv2DDNNLayer(conv2, num_filters, (3, 3),
                    stride=1, pad=1, nonlinearity=leaky_rectify, W=Orthogonal()))
        return MaxPool2DDNNLayer(conv3, (2, 2), 2)

    def conv_4_layer_stack(top, num_filters):
        conv1 = batch_norm(Conv2DDNNLayer(top, num_filters, (3, 3),
                    stride=1, pad=1, nonlinearity=leaky_rectify, W=Orthogonal()))
        conv2 = batch_norm(Conv2DDNNLayer(conv1, num_filters, (3, 3),
```

```
                stride=1, pad=1, nonlinearity=leaky_rectify, W=Orthogonal()))
        conv3 = batch_norm(Conv2DDNNLayer(conv2, num_filters, (3, 3),
                stride=1, pad=1, nonlinearity=leaky_rectify, W=Orthogonal()))
        conv4 = batch_norm(Conv2DDNNLayer(conv3, num_filters, (3, 3),
                stride=1, pad=1, nonlinearity=leaky_rectify, W=Orthogonal()))
        return MaxPool2DDNNLayer(conv4, (2, 2), 2)

    def conv_6_layer_stack(top, num_filters):
        conv1 = batch_norm(Conv2DDNNLayer(top, num_filters, (3, 3),
                stride=1, pad=1, nonlinearity=leaky_rectify, W=Orthogonal()))
        conv2 = batch_norm(Conv2DDNNLayer(conv1, num_filters, (3, 3),
                stride=1, pad=1, nonlinearity=leaky_rectify, W=Orthogonal()))
        conv3 = batch_norm(Conv2DDNNLayer(conv2, num_filters, (3, 3),
                stride=1, pad=1, nonlinearity=leaky_rectify, W=Orthogonal()))
        conv4 = batch_norm(Conv2DDNNLayer(conv3, num_filters, (3, 3),
                stride=1, pad=1, nonlinearity=leaky_rectify, W=Orthogonal()))
        conv5 = batch_norm(Conv2DDNNLayer(conv4, num_filters, (3, 3),
                stride=1, pad=1, nonlinearity=leaky_rectify, W=Orthogonal()))
        conv6 = batch_norm(Conv2DDNNLayer(conv5, num_filters, (3, 3),
                stride=1, pad=1, nonlinearity=leaky_rectify, W=Orthogonal()))
        return MaxPool2DLayer(conv6, (2, 2), 2)

except ImportError:
    def conv_2_layer_stack(top, num_filters):
        conv1 = batch_norm(Conv2DLayer(top, num_filters, (3, 3), stride=1, pad=1, nonlinearity=
        conv2 = batch_norm(Conv2DLayer(conv1, num_filters, (3, 3), stride=1, pad=1, nonlinearit
        return MaxPool2DLayer(conv2, (2, 2), 2)

    def conv_3_layer_stack(top, num_filters):
        conv1 = batch_norm(Conv2DLayer(top, num_filters, (3, 3), stride=1, pad=1, nonlinearity=
        conv2 = batch_norm(Conv2DLayer(conv1, num_filters, (3, 3), stride=1, pad=1, nonlinearit
        conv3 = batch_norm(Conv2DLayer(conv2, num_filters, (3, 3), stride=1, pad=1, nonlinearit
        return MaxPool2DLayer(conv3, (2, 2), 2)

    def conv_4_layer_stack(top, num_filters):
        conv1 = batch_norm(Conv2DLayer(top, num_filters, (3, 3), stride=1, pad=1, nonlinearity=
        conv2 = batch_norm(Conv2DLayer(conv1, num_filters, (3, 3), stride=1, pad=1, nonlinearit
        conv3 = batch_norm(Conv2DLayer(conv2, num_filters, (3, 3), stride=1, pad=1, nonlinearit
        conv4 = batch_norm(Conv2DLayer(conv3, num_filters, (3, 3), stride=1, pad=1, nonlinearit
        return MaxPool2DLayer(conv4, (2, 2), 2)

    def conv_6_layer_stack(top, num_filters):
        conv1 = batch_norm(Conv2DLayer(top, num_filters, (3, 3), stride=1, pad=1, nonlinearity=
        conv2 = batch_norm(Conv2DLayer(conv1, num_filters, (3, 3), stride=1, pad=1, nonlinearit
        conv3 = batch_norm(Conv2DLayer(conv2, num_filters, (3, 3), stride=1, pad=1, nonlinearit
        conv4 = batch_norm(Conv2DLayer(conv3, num_filters, (3, 3), stride=1, pad=1, nonlinearit
        conv5 = batch_norm(Conv2DLayer(conv4, num_filters, (3, 3), stride=1, pad=1, nonlinearit
        conv6 = batch_norm(Conv2DLayer(conv5, num_filters, (3, 3), stride=1, pad=1, nonlinearit
        return MaxPool2DLayer(conv6, (2, 2), 2)
```

In [15]:
```
k = 8
input_layer = InputLayer((None, 1, input_volume_shape[0], input_volume_shape[1]))
conv_stack_1 = conv_2_layer_stack(input_layer, 32)
dropout1 = DropoutLayer(conv_stack_1, p=0.1)
```

```
conv_stack_2 = conv_2_layer_stack(dropout1, 64)
dropout2 = DropoutLayer(conv_stack_2, p=0.2)

conv_stack_3 = conv_2_layer_stack(dropout2, 128)
dropout3 = DropoutLayer(conv_stack_3, p=0.3)

conv_stack_4 = conv_2_layer_stack(dropout3, 256)
dropout4 = DropoutLayer(conv_stack_4, p=0.4)

conv_stack_5 = conv_2_layer_stack(dropout4, 512)
dropout17 = DropoutLayer(conv_stack_5, p=0.5)

dense18 = DenseLayer(dropout17, 2048, nonlinearity=None)
norm1 = BatchNormLayer(dense18)
maxout1 = FeaturePoolLayer(norm1, k)
dropout19 = DropoutLayer(maxout1, p=0.5)

dense20 = DenseLayer(dropout19, 2048, nonlinearity=None)
norm2 = BatchNormLayer(dense20)
maxout2 = FeaturePoolLayer(norm2, k)

softmax21 = DenseLayer(maxout2, 10, nonlinearity=softmax)
```

1.1 Quality of Life Functions

```
In [16]: if not os.path.exists("logs"):
             os.mkdir("logs")
         logging.config.fileConfig("logging-training.conf")

         def regularization_objective(layers, lambda1=0., lambda2=0., *args, **kwargs):
             # default loss
             losses = objective(layers, *args, **kwargs)
             # get layer weights except for the biases
             weights = get_all_params(layers[-1], regularizable=True)
             regularization_term = 0.0
             # sum of abs weights for L1 regularization
             if lambda1 != 0.0:
                 sum_abs_weights = sum([abs(w).sum() for w in weights])
                 regularization_term += (lambda1 * sum_abs_weights)
             # sum of squares (sum(theta^2))
             if lambda2 != 0.0:
                 sum_squared_weights = (1 / 2.0) * sum([(w ** 2).sum() for w in weights])
                 regularization_term += (lambda2 * sum_squared_weights)
             # add weights to regular loss
             losses += regularization_term
             return losses

         def eval_regularization(net):
             if net.objective_lambda1 == 0 and net.objective_lambda2 == 0:
                 return 0
             # check the loss if the regularization term is not overpowering the loss
             weights = get_all_params(net.layers_[-1], regularizable=True)
             # sum of abs weights for L1 regularization
```

```
    sum_abs_weights = sum([abs(w).sum() for w in weights])
    # sum of squares (sum(theta^2))
    sum_squared_weights = (1 / 2.0) * sum([(w ** 2).sum() for w in weights])
    # add weights to regular loss
    regularization_term = (net.objective_lambda1 * sum_abs_weights) \
                          + (net.objective_lambda2 * sum_squared_weights)
    return regularization_term


def print_regularization_term(net):
    if net.objective_lambda1 > 0.0 or net.objective_lambda2 > 0.0:
        regularization_term = eval_regularization(net)
        print "Regularization term: {}".format(regularization_term.eval())

def validation_set_loss(_net, _X, _y):
    """We need this to track the validation loss"""
    _yb = _net.predict_proba(_X)
    _y_pred = np.argmax(_yb, axis=1)
    _acc = metrics.accuracy_score(_y, _y_pred)
    loss = aggregate(categorical_crossentropy(_yb, _y))
    loss += eval_regularization(_net)
    return loss, _acc


def store_model(model_file_name, net):
    directory_name = os.path.dirname(model_file_name)
    model_file_name = os.path.basename(model_file_name)
    if not os.path.exists(directory_name):
        os.makedirs(directory_name)
    # write model
    output_model_file_name = os.path.join(directory_name, model_file_name)
    start_write_time = time.time()
    if os.path.isfile(output_model_file_name):
        os.remove(output_model_file_name)
    with open(output_model_file_name, 'wb') as experiment_model:
        pickle.dump(net, experiment_model)
    total_write_time = time.time() - start_write_time
    m, s = divmod(total_write_time, 60)
    h, m = divmod(m, 60)
    logging.log(logging.INFO, "Duration of saving to disk: %0d:%02d:%02d", h, m, s)

def write_validation_loss_and_store_best(validation_file_name, best_weights_file_name,
                                         net, X_val, y_val, best_vloss, best_acc):
    # write validation loss
    start_validate_time = time.time()
    vLoss, vAcc = validation_set_loss(net, X_val, y_val)
    loss = vLoss.eval()
    current_epoch = net.train_history_[-1]['epoch']
    with open(validation_file_name, 'a') as validation_file:
        validation_file.write("{}, {}, {}\n".format(current_epoch, loss, vAcc))

    total_validate_time = time.time() - start_validate_time
    m, s = divmod(total_validate_time, 60)
    h, m = divmod(m, 60)
```

```
            logging.log(logging.INFO, "Duration of validation: %0d:%02d:%02d", h, m, s)

            # store best weights here
            if loss < best_vloss:
                start_bw_time = time.time()
                best_vloss = loss
                best_acc = vAcc
                with open(best_weights_file_name, 'wb') as best_model_file:
                    pickle.dump(net.get_all_params_values(), best_model_file, -1)

            return best_vloss, best_acc


        class AdjustVariableWithStepSize(object):
            """This class adjusts any variable during training
            """

            def __init__(self, name, start=0.03, steps=3, after_epochs=2000):
                self.name = name
                self.start = start
                self.steps=steps
                self.after_epochs=after_epochs
                self.ls = []

            def __call__(self, nn, train_history):
                if not self.ls:
                    for i in range(self.steps):
                        self.ls.extend(np.repeat(self.start/(np.power(10,i)), self.after_epochs))

                try:
                    epoch = train_history[-1]['epoch']
                    new_value = np.float32(self.ls[epoch - 1])
                    getattr(nn, self.name).set_value(new_value)
                except IndexError:
                    pass
```

1.2 CNN

```
In [17]: lambda1 = 0.0
         lambda2 = 5e-3

         net = NeuralNet(
             layers=softmax21,
             max_epochs=1,
             update=nesterov_momentum,
             update_learning_rate=theano.shared(np.float32(0.001)),
             update_momentum = 0.99,
             # update=adam,
             on_epoch_finished=[
                 EarlyStopping(patience=1000),
                 AdjustVariableWithStepSize('update_learning_rate', start=0.001, steps=2, after_epochs=8
             ],
             on_training_finished=[
                 EndTrainingFromEarlyStopping()
```

```
            ],
            objective=regularization_objective,
            objective_lambda2=lambda2,
            objective_lambda1=lambda1,
            batch_iterator_train=BatchIterator(batch_size=100),
            train_split=TrainSplit(
                eval_size=0.25),
            # train_split=TrainSplit(eval_size=0.0),
            verbose=3,
        )
```

In [18]:
```
p = PrintLayerInfo()
net.initialize()
# p(net)
```

1.2.1 load cnn instead

In [19]:
```
dir_name = 'net.vgg.large.l2.5e3.orthog-norm-maxout8-lr.2.steps'
validation_file_name = "{}/vloss-{}.txt".format(dir_name, dir_name)
model_file_name = "{}/{}.pickle".format(dir_name, dir_name)
best_weights_file_name = "{}/bw-{}.weights".format(dir_name, dir_name)
if os.path.exists(dir_name):
    print "Model exists. Loading {}.".format(dir_name)
    with open(model_file_name, 'rb') as reader:
        net = pickle.load(reader)
else:
    print "Training model from the beginning {}".format(dir_name)
```

```
Training model from the beginning net.vgg.large.l2.5e3.orthog-norm-maxout8-lr.2.steps
```

load_best_weights(best_weights_file_name, net) from nolearn.lasagne.visualize import plot_loss plt.figure(figsize=(15,9)) plt.ylim([0.1,0.5]) plt.plot([v['valid_loss'] for v in net.train_history_])

2 just this time.

net.on_epoch_finished.pop(1) print net.on_epoch_finished net.update_learning_rate=0.001

2.1 Define validation set

In [ ]:
```
val_dir =  "/media/dylan/Science/Kaggle-Data/distracted_drivers/val/"
X_val, y_val = image_gen_from_dir(val_dir, 40, 10, size=input_volume_shape).next()
X_val = X_val.reshape(-1, 1, input_volume_shape[0], input_volume_shape[1])
```

In [ ]:
```
image_gen = image_gen_from_dir(data_dir, 10, 10, size=input_volume_shape)
gen = random_aug_gen(image_gen, random_aug)
threaded_gen = threaded_generator(gen, num_cached=100)

ops_every = 500
best_acc = 0.0
best_vloss = np.inf

start_time = time.time()
try:
    for step, (inputs, targets) in enumerate(threaded_gen):
```

```
                shape = inputs.shape
                net.fit(inputs.reshape(shape[0],1, shape[1], shape[2]), targets)
                if (step + 1) % ops_every == 0:
                    print_regularization_term(net)
                    store_model(model_file_name, net)
                    # center validation
                    best_vloss, best_acc = write_validation_loss_and_store_best(
                        validation_file_name, best_weights_file_name, net, X_val, y_val, best_vloss, bes

        except StopIteration:
            # terminate if already early stopping
            with open(model_file_name, 'wb') as writer:
                pickle.dump(net, writer)
            total_time = time.time() - start_time
            print("Training successful by early stopping. Elapsed: {}".format(total_time))

# Neural Network with 22029994 learnable parameters

## Layer information

name                size          total    cap.Y    cap.X    cov.Y    cov.X    filter Y    filter X    fi
-----------------   ----------  -------  -------  -------  -------  -------  ----------  ----------  ----
InputLayer          1x128x128      16384   100.00   100.00   100.00   100.00         128         128
Conv2DDNNLayer      32x128x128    524288   100.00   100.00     2.34     2.34           3           3
BatchNormLayer      32x128x128    524288   100.00   100.00   100.00   100.00         128         128
NonlinearityLayer   32x128x128    524288   100.00   100.00   100.00   100.00         128         128
Conv2DDNNLayer      32x128x128    524288   100.00   100.00   100.00   100.00         128         128
BatchNormLayer      32x128x128    524288   100.00   100.00   100.00   100.00         128         128
NonlinearityLayer   32x128x128    524288   100.00   100.00   100.00   100.00         128         128
MaxPool2DDNNLayer   32x64x64      131072   100.00   100.00   100.00   100.00         128         128
DropoutLayer        32x64x64      131072   100.00   100.00   100.00   100.00         128         128
Conv2DDNNLayer      64x64x64      262144   100.00   100.00   100.00   100.00         128         128
BatchNormLayer      64x64x64      262144   100.00   100.00   100.00   100.00         128         128
NonlinearityLayer   64x64x64      262144   100.00   100.00   100.00   100.00         128         128
Conv2DDNNLayer      64x64x64      262144   100.00   100.00   100.00   100.00         128         128
BatchNormLayer      64x64x64      262144   100.00   100.00   100.00   100.00         128         128
NonlinearityLayer   64x64x64      262144   100.00   100.00   100.00   100.00         128         128
MaxPool2DDNNLayer   64x32x32       65536   100.00   100.00   100.00   100.00         128         128
DropoutLayer        64x32x32       65536   100.00   100.00   100.00   100.00         128         128
Conv2DDNNLayer      128x32x32     131072   100.00   100.00   100.00   100.00         128         128
BatchNormLayer      128x32x32     131072   100.00   100.00   100.00   100.00         128         128
NonlinearityLayer   128x32x32     131072   100.00   100.00   100.00   100.00         128         128
Conv2DDNNLayer      128x32x32     131072   100.00   100.00   100.00   100.00         128         128
BatchNormLayer      128x32x32     131072   100.00   100.00   100.00   100.00         128         128
NonlinearityLayer   128x32x32     131072   100.00   100.00   100.00   100.00         128         128
MaxPool2DDNNLayer   128x16x16      32768   100.00   100.00   100.00   100.00         128         128
DropoutLayer        128x16x16      32768   100.00   100.00   100.00   100.00         128         128
Conv2DDNNLayer      256x16x16      65536   100.00   100.00   100.00   100.00         128         128
BatchNormLayer      256x16x16      65536   100.00   100.00   100.00   100.00         128         128
NonlinearityLayer   256x16x16      65536   100.00   100.00   100.00   100.00         128         128
Conv2DDNNLayer      256x16x16      65536   100.00   100.00   100.00   100.00         128         128
BatchNormLayer      256x16x16      65536   100.00   100.00   100.00   100.00         128         128
NonlinearityLayer   256x16x16      65536   100.00   100.00   100.00   100.00         128         128
MaxPool2DDNNLayer   256x8x8        16384   100.00   100.00   100.00   100.00         128         128
```

```
DropoutLayer        256x8x8       16384   100.00   100.00   100.00   100.00          128          128
Conv2DDNNLayer      512x8x8       32768   100.00   100.00   100.00   100.00          128          128
BatchNormLayer      512x8x8       32768   100.00   100.00   100.00   100.00          128          128
NonlinearityLayer   512x8x8       32768   100.00   100.00   100.00   100.00          128          128
Conv2DDNNLayer      512x8x8       32768   100.00   100.00   100.00   100.00          128          128
BatchNormLayer      512x8x8       32768   100.00   100.00   100.00   100.00          128          128
NonlinearityLayer   512x8x8       32768   100.00   100.00   100.00   100.00          128          128
MaxPool2DDNNLayer   512x4x4        8192   100.00   100.00   100.00   100.00          128          128
DropoutLayer        512x4x4        8192   100.00   100.00   100.00   100.00          128          128
DenseLayer          2048           2048   100.00   100.00   100.00   100.00          128          128
BatchNormLayer      2048           2048   100.00   100.00   100.00   100.00          128          128
FeaturePoolLayer    256             256   100.00   100.00   100.00   100.00          128          128
DropoutLayer        256             256   100.00   100.00   100.00   100.00          128          128
DenseLayer          2048           2048   100.00   100.00   100.00   100.00          128          128
BatchNormLayer      2048           2048   100.00   100.00   100.00   100.00          128          128
FeaturePoolLayer    256             256   100.00   100.00   100.00   100.00          128          128
DenseLayer          10               10   100.00   100.00   100.00   100.00          128          128

Explanation
    X, Y:    image dimensions
    cap.:    learning capacity
    cov.:    coverage of image
    magenta: capacity too low (<1/6)
    cyan:    image coverage too high (>100%)
    red:     capacity too low and coverage too high
```

epoch	train loss	valid loss	train/val	valid acc	dur
1	33.25506	31.78047	1.04640	0.10000	1.03s
2	33.17589	31.78650	1.04371	0.06667	0.99s
3	32.99888	31.79005	1.03803	0.06667	0.99s
4	32.76854	31.79113	1.03074	0.06667	0.99s
5	32.49733	31.79183	1.02219	0.10000	0.99s
6	32.44246	31.78381	1.02072	0.16667	0.98s
7	32.39525	31.78540	1.01919	0.10000	0.99s
8	32.12280	31.78002	1.01079	0.03333	1.00s
9	32.20690	31.78030	1.01342	0.06667	0.99s
10	32.13252	31.78701	1.01087	0.13333	1.00s
11	32.00925	31.77393	1.00741	0.13333	1.01s
12	32.21346	31.79861	1.01305	0.10000	1.02s
13	32.43863	31.80940	1.01978	0.10000	0.99s
14	32.30440	31.82019	1.01522	0.10000	0.98s
15	32.48502	31.84547	1.02008	0.10000	0.99s
16	32.65747	31.85558	1.02517	0.10000	0.98s
17	32.57602	31.86535	1.02230	0.10000	0.98s
18	32.63330	31.84866	1.02464	0.10000	0.98s
19	32.59288	31.83547	1.02379	0.10000	0.99s
20	32.59788	31.78869	1.02546	0.10000	1.00s
21	32.40544	31.78866	1.01940	0.10000	1.01s
22	32.20409	31.75043	1.01429	0.10000	0.98s
23	32.16093	31.71952	1.01392	0.10000	0.95s
24	31.95499	31.72150	1.00736	0.16667	1.00s
25	32.12714	31.68436	1.01397	0.03333	0.99s

26	31.87635	31.70796	1.00531	0.06667	1.00s
27	31.97775	31.72711	1.00790	0.10000	1.00s
28	32.12251	31.69959	1.01334	0.06667	1.00s
29	32.03930	31.70591	1.01052	0.06667	0.99s
30	32.05185	31.72498	1.01030	0.13333	0.99s
31	32.19906	31.70462	1.01560	0.16667	0.97s
32	31.96948	31.68882	1.00886	0.10000	0.99s
33	32.12095	31.73195	1.01226	0.10000	1.01s
34	32.28888	31.70193	1.01851	0.10000	0.99s
35	32.10909	31.67241	1.01379	0.16667	1.00s
36	32.12541	31.62836	1.01572	0.13333	1.01s
37	31.98480	31.62747	1.01130	0.06667	1.00s
38	31.84256	31.59635	1.00779	0.06667	1.00s
39	31.76545	31.61071	1.00490	0.10000	1.01s
40	31.86823	31.60493	1.00833	0.10000	1.00s
41	31.93109	31.58243	1.01104	0.16667	1.00s
42	31.78740	31.59209	1.00618	0.10000	0.99s
43	31.86057	31.56188	1.00946	0.13333	1.01s
44	31.78907	31.59924	1.00601	0.10000	1.02s
45	31.66023	31.59312	1.00212	0.13333	1.00s
46	31.81987	31.58217	1.00753	0.10000	1.00s
47	31.84785	31.59037	1.00815	0.13333	1.00s
48	31.90572	31.59439	1.00985	0.06667	0.99s
49	31.86822	31.55051	1.01007	0.13333	0.99s
50	31.93616	31.57936	1.01130	0.06667	1.00s
51	31.87253	31.57030	1.00957	0.13333	1.01s
52	31.69820	31.50569	1.00611	0.06667	1.01s
53	31.68344	31.48365	1.00635	0.13333	1.01s
54	31.70494	31.49885	1.00654	0.10000	0.99s
55	31.65168	31.46290	1.00600	0.10000	0.99s
56	31.75206	31.44831	1.00966	0.10000	0.99s
57	31.68836	31.46900	1.00697	0.10000	1.00s
58	31.49895	31.40435	1.00301	0.10000	0.99s
59	31.52123	31.43587	1.00272	0.13333	1.00s
60	31.60312	31.42969	1.00552	0.13333	0.99s
61	31.63642	31.41870	1.00693	0.10000	1.01s
62	31.48268	31.42694	1.00177	0.10000	0.99s
63	31.53871	31.42119	1.00374	0.10000	1.00s
64	31.56316	31.41545	1.00470	0.13333	0.99s
65	31.49996	31.38778	1.00357	0.10000	1.00s
66	31.54456	31.33249	1.00677	0.13333	1.01s
67	31.62712	31.33192	1.00942	0.10000	1.00s
68	31.61530	31.33798	1.00885	0.16667	1.00s
69	31.53901	31.32602	1.00680	0.16667	0.99s
70	31.49148	31.28814	1.00650	0.06667	0.99s
71	31.39332	31.30571	1.00280	0.10000	0.97s
72	31.50287	31.30611	1.00629	0.03333	0.99s
73	31.47696	31.25891	1.00698	0.16667	0.99s
74	31.41952	31.31381	1.00338	0.10000	1.00s
75	31.59819	31.26338	1.01071	0.06667	1.00s
76	31.40242	31.27527	1.00407	0.10000	1.00s
77	31.26772	31.24176	1.00083	0.03333	1.00s
78	31.61490	31.18827	1.01368	0.10000	0.99s
79	31.52178	31.20059	1.01029	0.03333	1.00s

80	31.26742	31.21463	1.00169	0.06667	1.01s
81	31.32435	31.15311	1.00550	0.06667	1.00s
82	31.37941	31.17708	1.00649	0.06667	1.01s
83	31.31854	31.14687	1.00551	0.10000	0.99s
84	31.27228	31.12518	1.00473	0.06667	0.99s
85	31.32556	31.13277	1.00619	0.00000	1.00s
86	31.19910	31.01919	1.00580	0.20000	0.99s
87	31.38416	31.09854	1.00918	0.13333	1.01s
88	31.07056	31.13755	0.99785	0.03333	1.01s
89	31.23056	31.10376	1.00408	0.06667	1.01s
90	31.37108	31.08564	1.00918	0.03333	0.98s
91	31.15860	31.05631	1.00329	0.10000	0.96s
92	31.23090	31.03073	1.00645	0.06667	0.94s
93	31.04417	31.08680	0.99863	0.13333	0.96s
94	31.11362	31.01442	1.00320	0.16667	0.94s
95	30.97961	31.03494	0.99822	0.06667	0.98s
96	31.07029	30.95866	1.00361	0.13333	0.94s
97	31.05484	31.01039	1.00143	0.06667	0.95s
98	31.06017	30.98117	1.00255	0.13333	0.96s
99	30.98125	30.92967	1.00167	0.16667	1.01s
100	31.09165	30.94289	1.00481	0.10000	0.95s
101	30.93985	30.88714	1.00171	0.10000	0.99s
102	31.05118	30.92863	1.00396	0.10000	0.96s
103	31.07018	30.89117	1.00579	0.06667	0.95s
104	31.20908	30.89388	1.01020	0.16667	0.94s

2.2 Visualizations

In [ ]:
```
from notebook_functions import plot_validation_loss
```

In [ ]:
```
plot_validation_loss(net, validation_file_name, ylim=[0, 0.5])
```

In [ ]: