HOMEWORK ASSIGNMENT #4

DUE: Tuesday, November 3, 2020, 2.00PM

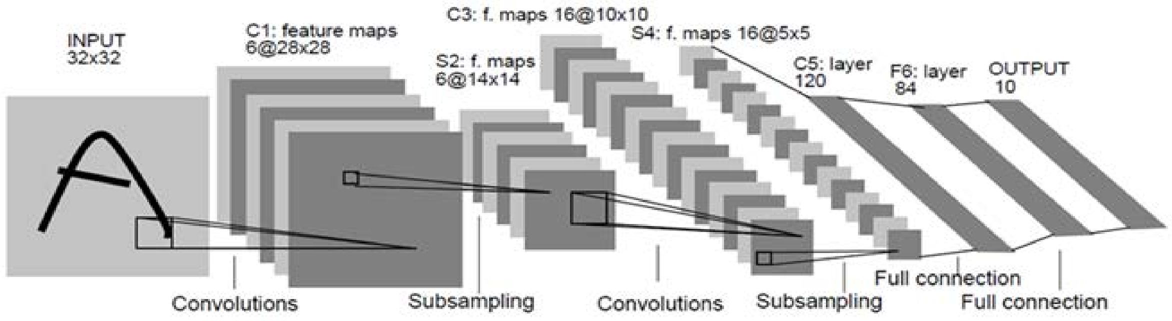
CSCI 677: Computer Vision, Prof. Nevatia

Fall Semester, 2020

This is a programming assignment to create, train and test a CNN for the task of image classification. To keep the task manageable, we will use a small dataset and a small network.

Architecture:

Construct a LeNet-5 style CNN network, using PyTorch functions. LeNet-5 is shown graphically below.



We ask you to experiment with varying the parameters of the network but use the following to start with:

a) The first layer has six, 5x5 convolution filters, stride =1, each followed by a max-pooling layer of 2x2 with stride = 2.

b) Second convolution layer has sixteen, 5x5 convolution filters, stride =1, each followed by 2x2 max pooling with stride =2.

c) Next is a fully connected layer of dimensions 120 followed by another fully connected layer of dimensions 84.

d) Finally, a SoftMax layer of dimension 10 provides the classification probabilities.

Note: All activation units should be ReLU.

Framework:

PyTorch is the required framework to use for this assignment. You are asked to define your model, “from scratch”, using available modules from PyTorch. Please do not import available definitions from the Internet though you are free to use them as a guide.

Dataset:

We will use the CIFAR-10 dataset. It consists of 10 mutually exclusive classes with 50,000 training images and 10000 test images, evenly distributed across the 10 classes. Each image is a 32x32 RGB image. You can download the Python version of the dataset from https://www.cs.toronto.edu/~kriz/cifar.html. It contains four files. Please use 40000 images from ***train*** file for training, the remaining 10000 images for validation, and ***test*** file for testing. You can select the images you use for training and validation on your own. You can use pickle package to load the files.

Each of the files contain a dictionary with several elements, below are the two elements we are going to use in this homework:

data -- a numpy array of uint8s. Each row of the array stores a 32x32 colour image

fine_labels -- a list of numbers in the range 0-99. The number at index *i* indicates the label of the *i*th image in the array **data**.

Training:

Train the network using the given training data. We suggest using a mini-batch size of 64, a learning rate of 0.001 and the ADAM optimizer, though you are free to experiment with other values. You do not need to write your own backpropagation algorithm, as it is implemented in PyTorch.

Record the error after each step (i.e. after each batch) so you can monitor it and plot it to show results. During training, you should test on the validation set at some regular intervals; say every 0.5 epochs, to check whether the model is overfitting.

Preprocessing:

You should normalize the images to zero mean and unit variance for pre-processing. An easy way to do this is to subtract the channel-wise mean and divide by the channel-wise standard division. Depends on the way you read your image, your image array might have a value range of (0, 255) or (0, 1). Please first normalize your image to (0,1) range, calculate the dataset mean/std values, and normalize the images to be zero mean and unit variance.

You can check torchvision.transforms.Normalize to apply the normalization in your pipeline.

By default PyTorch uses Kaiming (He) Uniform to initialize weights (see https://pytorch.org/docs/stable/nn.init.html#torch.nn.init.kaiming_uniform_).

You can try Xavier Uniform or Normal initializers for variations if you wish.

Augmentation:

You should augment the training data. Possible ways to augment are: enlarge the image by 10%, do a center crop by considering 90% of the image in the center, flip the image and add small amounts of Gaussian noise. You may use “resize” functions in the Python Image Library (PIL) to enlarge the image.

One caveat to note is that, PIL loads images in RGB format (OpenCV loads in BGR format by default). We briefly note how to read with PIL:

```
from PIL import Image
img = Image.open(path_to_img)
img_resized = img.resize(new_width, new_height)
# to further use as a numpy array
img_np = np.array(img_resized)
```

Test Results:
Test the trained network on the test data to obtain classification results and show the results in the form of confusion matrix and classification accuracies for each class. For the confusion matrix you could either write the code on your own, or use scikit-learn to to compute the confusion matrix. (See: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_matrix.html for more details).

System Variations:
Experiment with variations of the network with the aim of improving performance or ease of training the network without losing accuracy. Some suggestions are provided below but these are not requirements nor guaranteed to improve performance. Instead, you should be guided by the results of the variations you obtain.

i) Change the filter size (make them smaller) followed possibly by additional layers.
ii) Change the number of filters in the early layers.
iii) Change the size of the fully connected layers.
iv) Use batch normalization.
v) For inference, take multiple crops of the expanded test image and average the scores.

As the variations can be combined, the number of options to try can be very large, it is not expected that you would try all combinations but, instead, choose a small number of combinations that you expect will result in improved performance.

NOTE: **You may be able to find pre-trained models for LeNet and CIFAR-10 online. You may not use these and should train the network entirely from your own program.**

SUBMISSION:

You should turn in the following in a single PDF file:

1. A brief description of the programs you write, including the source listing.

2. A summary and discussion of the results, including:

   (i) Show the evolution of training losses and validation losses with multiple steps.
   (ii) Find the effects of parameter choices and try to find the best parameter settings. Show the confusion matrix and per-class classification accuracies on test dataset, for your best parameter settings: show the best few cases or all cases you have tried; visualize/present the metrics side-by-side to support your optimal parameters.
   (iii) Show some examples of failed cases. Discuss the limitation or weakness of the method or your choice of parameters.

Hints:

Following are some general hints on structuring your code, it is not required to follow this template.

1) You need to create:
   a) *Dataloader*: This produces batches of data points to be used for training
   b) *Model*: This is the main model. Data points would pass through the model in the forward pass. In the backward pass, using the backpropagation algorithm, gradients are stored.
   c) *Loss Function*: calculates the loss, given the outputs of the model and the ground-truth labels of the data.
   d) *Optimizer*: These are several optimization schemes: Standard ones are available in Pytorch; we suggest use of ADAM, though you could also try using the plain SGD. You would be calling these to update the model (after the gradients have been stored).
   e) *Evaluation*: Compute the predictions of the model and compare with the ground-truth labels of your data. In this homework, we are interested in the top-1 prediction accuracy
   f) *Training Loop*: This is the main function that will be called after the rest are initialized.
2) There is an official PyTorch tutorial online, please refer to this tutorial for constructing the above parts: https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html
3) Apart from the above, it is suggested to log your results. You could use TensorBoard (compatible with both PyTorch/TensorFlow) or simply use `logger` module to keep track of loss values, and metrics.
4) Note on creating batches: It is highly recommended to use the DataLoader class of PyTorch which uses multiprocessing to speed up the mini-batch creation.

Notes on using PyTorch Dataset Class:

For PyTorch, you would need to write a Dataset class as follows

```
from torch.utils.data import Dataset
class CIFAR10(Dataset):
    def __init__(self, *args):
        # Initialize some paths
    def __len__(self):
        # return the length of the full dataset.
        # You might want to compute this in
        # __init__ function and store it in a variable
        # and just return the variable, to avoid recomputation
        return len
    def __getitem__(self):
        # Somehow get the image and the label
        img_file, label = get_img_label()
        # read the image file
        img_pil = PIL.Image.open(img_file)
        img_transformed = transforms(img_pil)
```

Notes on using Transformations

Pytorch provides a nice API for the same via `torchvision.transforms`. Suppose you start with an image (H x W x C), two essential transforms needed are:

(i) Normalization: Need to subtract the mean and the standard deviation, computed on the training set.
(https://pytorch.org/docs/stable/torchvision/transforms.html#torchvision.transforms.Normalize)

(ii) ToTensor: Convert H x W x C -> C x H x W and also converts to the required types.
(https://pytorch.org/docs/stable/torchvision/transforms.html#torchvision.transforms.ToTensor)

You could compose the transforms using Compose:
(https://pytorch.org/docs/stable/torchvision/transforms.html#torchvision.transforms.Compose)

As an example, for random flip augmentation, you could do:
```
transforms_to_use = Compose([RandomFlip, ToTensor, Normalize])
new_img = transforms_to_use(img_pil)
```
Further, note that the suggested transformation cannot be done via existing transformations in `torchvision` and thus we would need to create our own `transformation`. For this, we need to create a new class as follows:
```
class SomeNewTransformation(object):
    def __init__(self, *args):
        ……
    def __call__(self, img):
        # do transformation
        return new_img
```