CS78/178, Winter 2019, Problem Set # 2

October 17, 2019

Due: February 6th 2020, 11:59pm

This problem set introduces you to the task of image categorization. It requires you to to implement PyTorch functions for creating and training a model that predicts the scene category for an image from a pre-defined set of classes. Section 5 lists the deliverables of this assignment, which include code and data. The code ('*.py') and data ('*.pt') files must be submitted electronically via Canvas as a single zipped directory. The directory must not contain any subdirectories. The name of the directory should be in the format 'First_Middle_Last_HW2', where 'First', 'Middle' (if it exists), and 'Last' match your student name in Canvas. Your zipped directory should therefore have the format 'First_Middle_Last_HW2.zip'.

In this assignment, we provide detailed specifications of a base model for image categorization, which will produce good (but not great) results on the given dataset. You will receive regular points for implementing and training the base model so as to reproduce the expected results. Furthermore, you will receive bonus points if you are able to improve the base model. In order to receive bonus points you must submit the test set results achieved with your improved model to our evaluation software on `flume.cs.dartmouth.edu`. You may only make 5 submissions during the assignment period. Your bonus score will be a function of the highest test set accuracy over all your submissions to our evaluation software. The deadline for this assignment is February 6, 2020. As you may remember, you are given a total of 4 free late days to be used for homework assignments. You may decide to use the free late days for the base model submission, the bonus part, or for both. But regardless of what you use them for, the late days that you use will be deducted from your total of 4 days. Once these 4 days are used up, any homework turned in late will receive a penalty of 25% per late day on your final score (regular points plus bonus points). Make sure to upload to Canvas all of your code, the base model as well as your best submission for the bonus part.

1 Overview

In this assignment you are provided with detailed instructions on how to create a model for solving an image categorization problem. Your task is to implement the functions described in the text in order to create and train the model (referred to as the 'base model') and replicate the performance reported within the text. This part is worth 100 points. You may additionally score up to 100 **bonus points** for modifying the proposed 'base model' so as to further improve its performance on the test set. We provide some guidelines on how the model may be improved but leave all actual design decisions to you. We strongly encourage you to attempt the bonus part.

By solving this problem set you will learn how to

- extract information from data with grid topology (images);
- determine how to properly train deep networks;
- identify the network characteristics that lead to improvements in performance.

2 Image Categorization

Image categorization involves predicting the correct class label for a given image from a set of predefined classes. We have provided you with a file called `image_categorization_dataset.pt` which contains the variables `data_tr, data_te, label_tr, sets_tr` and `class_names`.

- `data_tr` contains the training set, which consists of $32,000$ RGB images of size (32×32). Each images belongs to one of 16 possible image categories. Each category has 2000 examples in this training set. `data_tr` also contains $6,400$ images which form the validation set. Each class has 400 image examples in the validation set.
- `sets_tr` tells you which examples in `data_tr` belong to the training set and which examples belong to the validation set. This is a tensor with the i^{th} element denoting which set (training or validation) the example i belongs to. Examples from the training set have a value of 1 and examples from the validation set have a value of 2 in the `sets_tr` tensor.
- `label_tr` is a 1-D tensor with the same number of elements as the number of examples in `data_tr`. Each entry in this tensor corresponds to the class label of a given example.
- `data_te` is a tensor of $9,600$ images, corresponding to the test set. The labels of the test set are not given to you, to avoid the risk of overfitting your models to the test set.
- We provide the class names in a variable called `class_names`. Images are drawn from the following classes: `bathroom, bedroom, bowling_alley, castle, classroom, clothing_store, dining_hall, golf_course, hospital, library, office, restaurant, shopping_mall, swimming_pool, train_station, volleyball_court`.

Your first task will be to preprocess and organize the dataset into `TensorDatsets`. Next, you will implement and train the base model (described in detail in section 2.2.1). We provide detailed instructions on how to implement your first convolutional network for image categorization. The assignment asks you to perform these tasks by writing code in the following files (provided to you) along with the assignment handout.

1. `create_dataset`: This function preprocess the data as well as create training and validation `TensorDatasets`. Note that the function `create_dataset` that you will write for this assignment differs from the one you wrote for the previous assignment.

2. `cnn_categorization`: This script specifies the architecture of the convolutional network for the categorization task, e.g., how many hidden layers, what non-linearities to use at which hidden layer, what loss function to use for training, etc. It also specifies meta-details such as the training policy to be used for training your base model. This function will build the data structures (datasets and the network object) for training.

3. `cnn_categorization_base`: This function constructs the base model. It is invoked in `cnn_categorization.py` to create a network based on the architecture you specify in that file. It returns a model that can be trained by PyTorch.

2.1 The Dataset

Complete the function `create_dataset` in `create_dataset.py` which have the signature:

```
def create_dataset(data_path, output_path, contrast_normalization, whiten)
```

Your function must load the dataset file that `data_path` points to. The dataset is a Python dictionary containing the keys `data_tr`, `data_te`, `sets_tr`, `label_tr`, and `class_names`. The values referenced by these keys have been explained above.

If `data_path` is not the string `'image_categorization_dataset.pt'`, it is assumed that the user is simply reading a saved preprocessed data from an earlier call. In that case, the function skips the preprocessing steps. However, if `data_path` is the string `'image_categorization_dataset.pt'`, preprocess the data as follows.

1. Zero-center the input images by subtracting the per-pixel mean computed from the *training* set from all of the examples (both training and validation) in the variable `data_tr`. Note that the image mean should be of the same size as the original examples ($3 \times 32 \times 32$).

2. Zero-center the test images, by subtracting the per-pexel mean computed on the *training* set. (Typically, preprocessing data statistics such as the per-pixel mean on the training are saved for use on future test datapoints. However, since our test data is included in the dataset, we will just preprocess it and save the preprocessed data.)

3. Your function should also allow for optional preprocessing of the data. The boolean input arguments `contrast_normalization` and `whiten` indicate whether or not to do the corresponding preprocesing.

   - `contrast_normalization` (Boolean): Contrast normalization performs element-wise standardization (division by the standard deviation).
   - `whiten` (Boolean): Whitening is an operation that removes correlation between the pixels in the image.

   Code for these preprocessing operations has been provided to you in the function. The pre-processing steps are performed after you have zero-centred the images in `data_tr` and `data_te`.

4. Save the preprocessed data in `output_path` so that it can be retrieved efficiently in future (if the same preprocessing steps are desired). Thus, after you have called the function once, you should set the value of `data_path` to the output file name in subsequent calls unless you are changing the optional preprocessing options. Like the optional preprocessing steps, code for saving the preprocessed data has been provided in the function.

2.2 Designing the base model

2.2.1 Base model details

Table 1 contains the specifications of the base model that you must implement in the file `cnn_categorization_base.py`.

Layer	1	2	3	4	5	6	7	8	9	10	11
Name:	`conv_1`	`bn_1`	`relu_1`	`conv_2`	`bn_2`	`relu_2`	`conv_3`	`bn_3`	`relu_3`	`pool_3`	`pred`
Type:	Conv	BN	ReLU	Conv	BN	ReLU	Conv	BN	ReLU	Pool	Conv
Kernel:	3			3			3			8	1
Pad:	1			1			1			0	0
Stride:	1			2			2			1	1
Filters:	16	16		32	32		64	64			16
Output:	`conv_1`	`bn_1`	`relu_1`	`conv_2`	`bn_2`	`relu_2`	`conv_3`	`bn_3`	`relu_3`	`pool_3`	`pred`

Table 1: This table describes the base model architecture for the categorization task. BN stands for Batch Normalization. You have not yet studied the concept of batch normalization but you will be introduced to it at a later point in the course. Batch Normalization provides significant advantages in training. Thus we use this layer within our network architecture. Name stands for the layer name.

Your function takes as input a dictionary `netspec_opts`. You are to specify this dictionary in the file `cnn_categorization.py`. Your code in `cnn_categorization_base.py` is supposed to read data from this dictionary and construct a network accordingly. A well written `cnn_categorization_base.py` can greatly help you with the bonus part of this assignment.

The dictionary `netspec_opts` contains the following keys:

- `kernel_size` is a list of length (L) where L is the total number of layers in the network. Each entry of the list is a *tuple* of the form (x, y). For all convolutional and pooling layers the tuple describes the kernel size where x is the kernel height and y is the kernel width. For all other layers, both x and y should be set to 0.

- `num_filters` is a list of length (L). For a "Conv2d" layer it specifies the number of filters you want to learn at that layer. For Batch Normalization (BN) layers this should be set to the number of filters in the preceding convolutional layer. For all other layers this should be set to 0.

- `stride` is a list of length (L) which contains the stride for the convolution and pooling operation at each layer.

- `layer_type` is a list of strings describing the type of each layer. You should use `'conv'` for convolutional layers, `'bn'` for batch normalization layers, `'relu'` for Rectified Linear Unit layers and `'pool'` for pooling layers.

Just like in the previous assignment, you should add a convolutional layer that produces an output of size 16 (the number of classes you are trying to predict). The output of this layer should be called `'pred'`. Also like the previous assignment, we have provided you with the `categorical cross-entropy` loss as the objective so you do not need to specify one.

2.2.2 Guide for implementing the base model

This subsection will guide you in setting up the base model architecture for image categorization using PyTorch's `Sequential` interface. The code discussed in this subsection must be inserted in the file `cnn_categorization_base.py`. The code will require creating the layers of your network using the field values that you already set in the dictionary `netspec_opts` defined above.

Start as before by constructing an instance of a `Sequential` class as shown below.

```
sequential_net = nn.Sequential();
```

To start constructing an image classification network, add a convolutional layer to your `sequential_net` network by using the `add_module` method and the `Conv2d` module of the `nn` interface:

```
sequential_net.module(name, nn.Conv2d(in_channels, num_filters,
                                      kernel_size, stride, padding))
```

We will now explain in detail these parameters.

- `name`: This refers to the name of this layer. The layer names can be read from row 2 of Table 1.
- `in_channels` is the number of filters of the previous convolutional layer. The value of this parameter for the first convolutional layer is 3 since our data is RGB images. If our images were grayscale images, `in_channels` would have been 1.
- `num_filters`: The is the number of filters in the convolutional layer being defined. This is also the number of feature maps that the current layer will create based on its input. The i^{th} layer's `num_filters` should be set to `num_filters[i]`.
- `kernel_size`: This corresponds to the spatial dimensions of your convolutional filter for this layer. For the i^{th} layer, this value should be set to `kernel_size[i]`.
- `stride`: This determines how many pixels are evaluated in convolution. A striding factor of 1 means all pixels are convolved with each filter and a striding factor of 2 makes the convolutional layer skip every other pixel and produce an output that is half the size of the input. Striding is used to increase the effective receptive field size and to reduce the dimensionality of the data passed to the next layer.
- `padding`: As seen in class, padding is used in convolutional layers to allow the filters to be applied to pixels near the border. The padding factor depends on the size of the convolutional kernel. Typically, if your convolutional kernel has a size of 3, you will pad by 1. If your kernel has a size of 5, you should pad your input by 2 pixels on each side. In general, if your convolutional kernel has size k (where k is assumed to be an odd integer number), then you

should set the padding factor to be $\frac{(k-1)}{2}$ in order to obtain an output of the same size as the input (assuming no stride or pooling).

Now that we have shown you how to add a convolutional layer, we will show you how to add a `'BatchNormalization'` layer. We have not yet discussed the method of batch normalization in class. Yet, we recommend using this layer for this assignment as it renders the training more stable and it prevents problems of vanishing and exploding gradients. A batch normalization layer is typically included after a convolutional layer (see Table 1 for our recommended placement of batch normalization layers in the base model). To add a batch normalization layer, you should adapt the following code

```
sequential_net.add_module(name, nn.BatchNorm2d(num_features))
```

We now explain the parameters of adding the batch normalization layer:

- `name`: Like for the convolutional layer, `name` is the name of the batch normalization layer.
- `nn.BatchNorm2d`: This is the `nn` module for a $2d$ batch normalization layer. It must be initialized with the attributes `'num_features'` corresponding to the number of channels (filters) of the preceding convolution layer (whose output is being batch normalized).

For image categorization architectures it is common to use the Rectified Linear Unit (`relu`) non-linearity for all hidden layers. `relu` is better than the `sigmoid` and `tanH` non-linearities that you implemented in the previous assignment as the responses produced by the `relu` function for positive input do not saturate.

To add a `relu` layer you should use the following code.

```
sequential_net.add_module(name, nn.ReLU())
```

In the base model a `relu` layer is always used after a batch normalization layer as shown in Table 1.

At a certain depth in your architecture you will want to capture greater spatial context via a pooling layer. The following code shows how to add a pooling layer.

```
sequential_net.add_module(name, nn.AvgPool2d(kernel_size, stride, padding))
```

We now explain the parameters of adding a pooling layer.

- `name`: This is the name of the pooling layer being added.
- `nn.AvgPool2d`: This creates an instance of a $2d$ pooling layer which has arguments `'kernel_size'`, `'stride'` and `'padding'`. The value for `'kernel_size'` determines the extent of the pooling region. This is specified as the kernel size in the base model description. Both `'stride'` and `'padding'` have the same meaning as explained above. There are other pooling options like `nn.MaxPool2d` besides `nn.AvgPool2d`. We recommend that you use average pooling for this dataset.

2.3 Training policy for base model

Now we specify the policy to be used for training the base model. The training policy must be stored in the dictionary `train_opts` defined in `cnn_categorization.py`. This dictionary has keys: `lr`, `weight_decay`, `batch_size`, `momentum`, `num_epochs`, `step_size`, and `gamma`. The function `train` in `train.py` depends on a well defined `train_opts` to work.

We ask that for the base network you specify the value of `num_epochs` to be 25. Set the learning rate `lr` to 0.1 and the `momentum` value to 0.9. We ask that you set `weight_decay` to be 0.0001 and the `batch_size` to be 128. The value of `step_size` should be 20 and that of `gamma` should be 0.1.

It is always a good idea to save your models at various checkpoints. Specify a variable in `cnn_categorization.py` called `exp_dir`. This variable will instruct the `train` function on where to save the model checkpoints. If the value for `exp_dir` is not specified, `train` will skip saving checkpoints of your model.

2.4 Training and evaluation

Once you have implemented the base model according to the architecture specified in Table 1 and the guide provided in the previous subsections, you will be able to train the base model by running the script `cnn_categorization.py`.

Our implementation of the base model produces an accuracy of 55.61% on the validation set. If you have implemented everything correctly, your model should produce a validation accuracy within 1% of this value.

3 Bonus: improving the base model

The bonus task for this assignment is to improve the performance of the base model described in the previous section. For this, you must implement a function called `cnn_categorization_advanced.py` (similar to `cnn_categorization_base.py`) which takes as input a dictionary `netspec_opts`. The function returns as output a model. You should definte a dictionary `train_opts`, the training policy for the advanced model. You may want to vary both the architecture and the training policy. Your goal is improve test performance over the base model as much as possible. Since we do not give you access to the labels for the test examples, you will evaluate your model on the test set by making a submission to our evaluation software. Section 5.1 provides details on how to make a submission to the evaluation software.

We now give some potential suggestions on how to improve the base model.

- You should consider varying the number of filters. Note that the first few convolutional layers in the network have a small effective receptive field and therefore are unable to capture high-level features. Conversely, deeper layers have bigger effective receptive field and build on top of the low-level features computed in the early layers. Thus, the deeper layers are potentially able to capture semantic features (e.g., features corresponding to objects or parts of objects in the scene). Think about this when choosing how to vary the number of filters in your network.

- Do you think the model is overfitting or underfitting? To answer this question look at the results of the train-validation plot produced during the training of the base model.

- Do you think the base model has enough convolutional layers? Think about the receptive field size for the model at the last layer (before pooling). Does the receptive field size cover the entire image?
- You may choose to perform data preprocessing steps such as contrast normalization and/or data whitening to improve performance, if you want.
- By looking at the train-validation plot from training your base model, do you think the model has converged? Should you vary the training policy in some way?
- Consider using early stopping if your model yields increasing validation errors at the end of the training stage.
- You may want to try data-augmentation techniques such as cropping. Also consider over-sampling at testing time if you use cropping.
- You may want to remove the randomization seed from the file `cnn_categorization.py` if you want to try out different random initializations for the same model.

You must include in your final submission a `submission_details.txt` file describing the modifications you made and why you think these modifications might help improve the performance of your model. Your description should be no be longer than $1,000$ characters (including spaces and punctuations). However, you must give a complete description of your modifications with respect to the base model.

You will be graded based on the performance of your model on the top-1 accuracy metric on the testing set. The prediction with the highest probability will be considered your predicted class label and compared with the ground-truth class.

Please note: bonus points will be given only if we can reproduce your best results using the training code that you submitted to Canvas. If the code that you submit to Canvas does not run or produces a different result, you will receive 0 bonus points.

4 Academic integrity

This homework assignment must be done individually. Sharing code or model specifications is strictly prohibited. Homework discussions are allowed only on Piazza, according to the policy outlined on the course Web page: `http://www.cs.dartmouth.edu/~lorenzo/teaching/cs178/`. You are not allowed to search online for auxiliary software, reference models, architecture specifications or additional data to solve the homework assignment. Your submission must be entirely your own work. That is, the code and the answers that you submit must be created, typed, and documented by you alone, based exclusively on the materials discussed in class and released with the homework assignment. You can obviously consult the class slides posted in Canvas, your lectures notes and the textbook. **Important**: the models for this homework assignment must be trained **exclusively** on the data provided with this assignment. The improvements made to the base model must be your own. These rules will be strictly enforced and any violation will be treated seriously.

5 Submission instructions

You must submit to our evaluation software the archive prepared by the provided script `create_submission_base.py`. If you attempt the bonus section (you should!), you must also run the script `create_submission_advanced.py` and submit the resulting archive to the evaluation software. You must also upload to Canvas a zip file containing both submission archives (created by our scripts) **as well as all your code**.

Read the scripts (`create_submission_base.py` and `create_submission_advanced.py`) very carefully to understand how the evaluation is performed. You are required to provide the path to your final models and the path to the preprocessed datasets you created for running these scripts along with other important pieces of information. These scripts perform evaluation on the validation set and the testing set and save the probability values produced by each of your models to disk. Then they create an archive containing your model file, the probability values produced on the validation set and the probability values produced on the testing set (and your model description for the bonus task).

5.1 Getting feedback for the bonus task

5.1.1 Setting up submission files

For this class we set up a real-time feedback mechanism for you to test the performance of your advanced model on actual testing data. Start by logging

For a base model submission, our system computes the prediction accuracy on the validation set and sends you an email telling you the accuracy of your predictions on the validation set only. This is to allow you to get familiar with the submission format and procedure and also to double-check that the validation performance returned by our evaluation system matches the one returned by your code. You may make as many submissions as you like for the base model.

For the advanced model, our system evaluates your predictions on the validation set and the test set. The feedback system will send an email to your account with the results of your current model and your best submission to date. The feedback system will not accept more than 5 submissions. Thus make sure to submit an advanced model only when you think it performs well. At the same time, you don't want to wait to use your 5 submissions until the last minute only to discover that they all heavily overfit the training data and produce poor performance on the test set!

5.1.2 Making the submission

If your submission is successful you will get a prompt on your terminal stating that your submission has been processed successfully and an email has been sent to your account. Please make sure to use the submissions generated by the scripts we provide (`create_submission_base.m` and `create_submission_advanced.m`). If you attempt to archive the files yourself the created archive may not work with the feedback system.

The email generated contains the score of your last submission, your best evaluation score so far and the number of attempts you have left.

5.2 Submission instructions for Canvas

You must also upload your code, models and submission archives to Canvas. Your submission to Canvas should be a zip file that contains the following files. Please only provide the best performing `advanced_categorization.zip` in your final submission to Canvas (keep track of them). Please also provide the `*_mod.py` files if you modified the original files provided for your advanced submission. Please make sure that in addition to these mandatory files you include any and all code files that are needed for replicating your results.

- ☐ `base_submission.zip` [**100 points**]
- ☐ `advanced_categorization.zip` [**100 bonus points**]
- ☐ `create_dataset.py`
- ☐ `create_dataset_mod.py`
- ☐ `cnn_categorization.py`
- ☐ `cnn_categorization_mod.py`
- ☐ `cnn_categorization_base.py`
- ☐ `cnn_categorization_advanced.py train.py`

You must upload a zip file containing all these files to Canvas.