Project 4: Scene Recognition with Deep Learning

CS 4476

Fall 2023

Brief

- Due: Check Canvas for up to date information
- Project materials including report template: GitHub
- Hand-in: Gradescope
- Required files: `<your_gt_username>.zip`, `<your_gt_username>_proj4.pdf`

Overview

In this project, you will design and train deep convolutional networks for scene recognition. In Part 1, you will train a simple network from scratch. In Part 2, you will implement a few modifications on top of the base architecture from Part 1 to increase recognition accuracy to ∼55%. In Part 3, you will instead *fine-tune* a pre-trained deep network to achieve more than 80% accuracy on the task. We will use the pre-trained ResNet architecture which was not trained to recognize scenes at all. Finally, we will explore multi-label prediction of scene attributes in Part 4.

These different approaches (starting the training from scratch or fine-tuning) represent the most common approach to recognition problems in computer vision today–train a deep network from scratch if you have enough data (it's not always obvious whether or not you do), and if you cannot then fine-tune a pre-trained network instead. A GPU is not necessary for this project, but you can use Google Colab to help speed up training. Learn more about Colab here.

Setup

1. Check https://github.gatech.edu/cs4476/project-4 for environment installation.
2. If you choose to use Google Colab to train your models, run `python zip_for_colab.py` and required files will be written to `cv_proj4.zip`. Open `proj4.ipynb` in Colab, upload this zipped file and do not forget to **download** them after you have completed. You will **lose your files** if the runtime resets!
3. Run the notebook using `jupyter notebook ./project-4.ipynb`
4. After implementing all functions, ensure that all sanity checks are passing by running `pytest tests` inside the main folder.
5. Generate the zip folder for the code portion of your submission once you've finished the project using `python zip_submission.py --gt_username <your_gt_username>`

Dataset

The dataset to be used in this assignment is the 15-scene dataset, containing natural images in 15 possible scenarios like bedrooms and coasts. It was first introduced by Lazebnik et al, 2006 [1]. The images have a typical size of around 200 by 200 pixels, and serve as a good milestone for many vision tasks. A sample collection of the images can be found below:

Figure 1: Example scenes from each of the categories of the dataset.

Download the data (link at the top), unzip it and put the `data` folder in the `proj4` directory.

1 Part 1: SimpleNet

Introduction

In this project, scene recognition with deep learning, we are going to train a simple convolutional neural net from scratch. We'll be starting with some modification to the dataloader used in this project to include a few extra pre-processing steps. Subsequently, you will define your own model and optimization function. A trainer class will be provided to you, and you will be able to test out the performance of your model with this complete pipeline of classification problem.

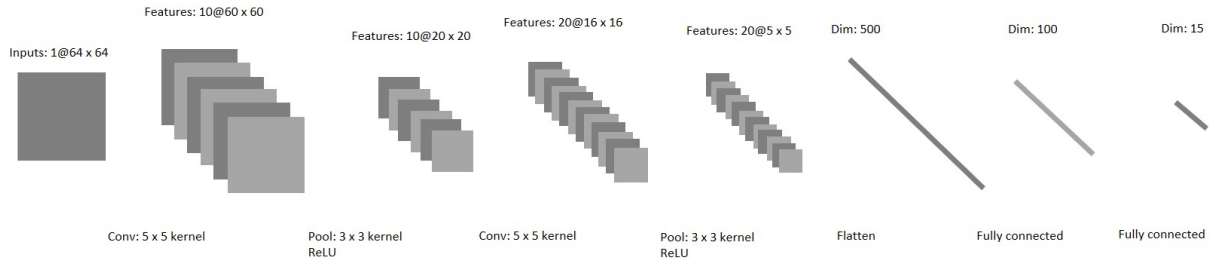


Figure 2: The base SimpleNet architecture for Part 1.

1.1 Dataset

In this part you'll be implementing the basic `Dataset` object, which helps to retrieve the data from your local data folder, and prepare to be used by your model. To start with, fill in the `ImageLoader` class in `image_loader.py`. Note that the `ImageLoader` class contains the paths to the dataset images, and should be able to return the expected element given an index. More details can be found in the file.

Additionally, in `data_transforms.py`, complete the function `get_fundamental_transforms()` to resize the input and convert it to a tensor.

Useful functions: `transforms.Resize`, `transforms.ToTensor`, `transforms.Compose`

1.2 Model

First, open `simple_net.py` and fill in the model definition. By now you should have a decent grasp on how to properly define a deep learning model using `nn.Conv2d`, `nn.ReLU`, `nn.MaxPool2d`, etc. For this part, define a convolutional neural network with 2 conv layers (and the corresponding ReLU, maxpool, and fully-connected layers) which aligns with our training dataset (15 classes) and mirrors Figure 2. After you have defined the proper model, fill in the `forward` function, which accepts the input (a batch of images), and generates the output classes (this should only take a couple of lines of code).

1.3 Trainer

We have provided you with a basic trainer, `runner.py`, where we load the dataset and train the network you have defined for some epochs. To complete this section, first assign proper values to the dict, `optimizer_config`, in the Jupyter notebook with reasonable learning rate and weight decay, and then fill in the `optimizer.py` with the default optimization function `Adam`.

Next, in `dl_utils.py`, complete `compute_accuracy()` and `compute_loss()`, where you are given an input model, and calculate the accuracy and loss given the prediction logits and the ground-truth labels respectively. These functions will be used in the training process later.

Lastly, train it using the starter code in Jupyter notebook, and report the training process according to the report. Take note that you may use these values (`lr` and `weight_decay`) as a starting point to tune your parameters to pass this part.

Note that although you are not required to modify anything in the `runner.py`, it is still highly recommended to read through the code and understand how to properly define the training process, as well as record the loss history.

To receive credit for the `SimpleNet` model in Part 1, you are required to get a validation accuracy of **45%**.

2 Part 2: SimpleNet with additional modifications

In Part 1 we implemented a basic CNN, but it doesn't perform very well. Let's try a few tricks to see if we can improve our model performance. You can start by copying your `SimpleNet` architecture from `simple_net.py` into `SimpleNetFinal` in `simple_net_final.py`

2.1 Problem 1: We don't have enough training data. Let's "jitter."

If you left-right flip (mirror) an image of a scene, it never changes categories. A kitchen doesn't become a forest when mirrored. This isn't true in all domains–a "d" becomes a "b" when mirrored, so you can't "jitter" character recognition training data in the same way. But we can synthetically increase our amount of training data by left-right mirroring training images during the learning process.

In `data_transforms.py`, implement `get_fundamental_augmentation_transforms()` to randomly flip some of the images (or entire batches) in the training dataset in addition to the transforms from Section 4.1 (you can start by copying your implementation of `get_fundamental_transforms()` into `get_fundamental_augmentation_transforms()`). You should also add some amount of color jittering (which in our case just means random pixel intensity shifts because we're working with grayscale images).

You can try more elaborate forms of augmentation (e.g., zooming in a random amount, rotating a random amount, taking a random crop, etc.). Mirroring and color jitter help quite a bit though (you should see a 5% to 10% increase in accuracy).

After you implement these, you should notice that your training error doesn't drop as quickly. That's actually a good thing because it means the network isn't overfitting to the 1,500 original training images as much (because it sees 3,000 training images now, although they're not as good as 3,000 truly independent samples). Because the training and test errors fall more slowly, you may need more training epochs or you may try modifying the learning rate.

Useful functions: `transforms.RandomHorizontalFlip, transforms.ColorJitter`

2.2 Problem 2: The images aren't zero-centered and variance-normalized.

One simple trick which can help a lot is to normalize the images by subtracting their mean and then dividing by their standard deviation. First `compute_mean_and_variance()` function in `stats_helper.py` to compute the mean and standard deviation. You should decide which mean or standard deviation you need to use for the training and test datasets. It would arguably be more proper to only compute the mean from the training images (since the test/validation images should be strictly held-out). Then, in `data_transforms.py`, implement the functions `get_fundamental_normalization_transforms()` and `get_all_transforms()` to normalize the input using the passed in mean and standard deviation in addition to the transforms from Section 4.1 and Section 2.1, respectively. Again, you can first copy over `get_fundamental_transforms()` and `get_fundamental_augmentation_transforms()`, respectively, and then add the normalization transform to both (the former is used for pre-processing the validation set, and the latter is used for pre-processing the training set). After doing this, you should see another few % increase in accuracy.

Useful function: `transforms.Normalize`

2.3 Problem 3: Our network isn't regularized.

If you train your network (especially for more than the default number of epochs) you'll see that the training accuracy can approach the 90s while the validation accuracy hovers at 45% to 55%. The network has learned weights which can perfectly recognize the training data, but those weights don't generalize to held-out test data. The best regularization would be more training data, but we don't have that. Instead we will use *dropout* regularization.

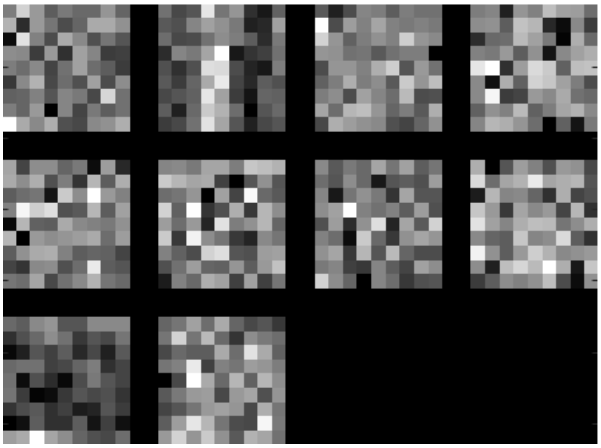
What does dropout regularization do? It randomly turns off network connections at training time to fight overfitting. This prevents a unit in one layer from relying too strongly on a single unit in the previous layer. Dropout regularization can be interpreted as simultaneously training many "thinned" versions of your network. At test time all connections are restored, which is analogous to taking an average prediction over all the "thinned" networks. You can see a more complete discussion of dropout regularization in this paper.

The dropout layer has only one free parameter (the dropout rate), which is the proportion of connections that are randomly deleted. The default of 0.5 should be fine. Insert a dropout layer between your convolutional layers in the `SimpleNetFinal` class in `simple_net_final.py`. In particular, insert it directly before your last convolutional layer. Your validation accuracy should increase by another 10%. Your train loss should decrease much more slowly. That's to be expected–you're making life much harder for the training algorithm

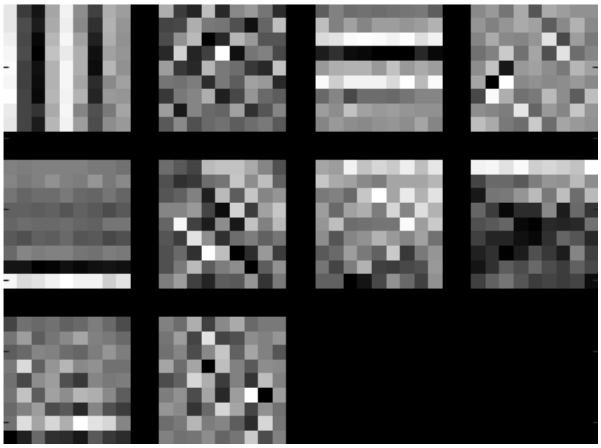
by cutting out connections randomly.

If you increase the number of training epochs (and maybe decrease the learning rate) you should be able to achieve around 55% validation accuracy. Notice how much more structured the learned filters are at this point compared to the initial network before we made improvements:

First layer filters learned by starter code



First layer filters learned with mirroring, center, and dropout regularization



2.4 Problem 4: Our network isn't deep.

Let's take a moment to reflect on what our convolutional network is actually doing. We learn filters which seem to be looking for horizontal, vertical, and parallel edges. Some of the filters have diagonal orientations, and some seem to be looking for high frequencies or center-surround. This learned filter bank is applied to each input image, the maximum response from each 7×7 block is taken by the max-pooling, and then the ReLU layer zeroes out negative values. The fully connected layer sees a 10 channel image with 8×8 spatial resolution. It learns 15 linear classifiers (a linear filter with a learned threshold is basically a linear classifier) on this 8×8 filter response map. This architecture is reminiscent of hand-crafted features like the gist scene descriptor developed precisely for scene recognition (on 8 scene categories which would later be incorporated into the 15 scene database). The gist descriptor actually works better than our learned feature. The gist descriptor with a non-linear classifier can achieve 74.7% accuracy on the 15 scene database.

Our convolutional network to this point isn't "deep." It has two layers with learned weights. Contrast this with the example networks for MNIST and CIFAR in PyTorch which contains 4 and 5 layers, respectively. AlexNet and VGG-F contain 8 layers, the VGG "very deep" networks contain 16 and 19 layers, and ResNet contains up to 150 layers.

One quite unsatisfying aspect of our current network architecture is that the max-pooling operation covers a window of 7×7 and then is subsampled with a stride of 7. That seems overly lossy and deep networks usually do not subsample by more than a factor 2 or 3 each layer.

Let's make our network deeper by adding an additional convolutional layer to `simple_net_final.py`. In fact, we probably don't want to add just a convolutional layer, but another max-pooling layer and ReLU layer as well. For example, you might insert a convolutional layer after the existing ReLU layer with a 5×5 spatial support, followed by a max-pool over a 3×3 window with a stride of 2. You can reduce the max-pool window in the previous layer, adjust padding, and reduce the spatial resolution of the final layer until your network's final layer (not counting the softmax) has a data size of 15. You also need to make sure that the data depth output by any channel matches the data depth input to the following channel. For instance,

maybe your new convolutional layer takes in the 10 channels of the first layer but outputs 15 channels. The final layer would then need to have its weights initialized accordingly to account for the fact that is operates on a 15-channel image instead of a 10-channel image.

Training deeper networks is tricky. The networks are slower to train and more sensitive to initialization and learning rate. For now, try to add one or two more blocks of "conv/pool/relu" and see if you can simply match your performance of the previous section.

2.5 Problem 5: Our "deep" network is slow to train and brittle

You might have noticed that your deeper network doesn't seem to learn very reasonable filters in the first layer. It is harder for the gradients to pass from the last layer all the way to the first in a deeper architecture. Normalization can help. In particular, let's add a batch normalization layer after each convolutional layer (except for the last) in `simple_net_final.py`. If you have 4 total convolutional layers we will add 3 batch normalization layers. You can check out `nn.BatchNorm2d(num_features=...)`. You will also need to initialize the weights of the batch normalization layers. Set weight to 1 and bias to 0.

Batch normalization by itself won't necessarily increase accuracy, but it will allow you to use *much* higher learning rates. Try increasing your learning rate by a factor of 10 or even 100, and now you can rapidly explore and train different network architectures. Notice how the first layer filters start to show structure quickly with batch normalization.

We leave it up to you to determine the specifics of your deeper network: number of layers, number of filters in each layer, size of filters in each layer, padding, max-pooling, stride, dropout factor, etc. It is *not* required that your deeper network increases accuracy over the shallow network (but it can with the right hyperparameters).

To receive credit for the `SimpleNetFinal` model in Part 2, you are required to get a validation accuracy of **55%**.

2.6 Analysis using confusion matrix

A confusion matrix is a helpful tool for visualizing the performance of classification algorithms. Each row of the matrix represents the instances in a predicted class, while each column represents the instances in an actual class. The confusion matrix counts the number of instances of a given (target, prediction) pair. We are able to use this to understand the classification behaviour.

A confusion matrix can also be normalized by dividing each row by the total number of instances of the target class. This is helpful for comparing between large and small datasets, as well as when there is significant class imbalance.

Complete the following in `confusion_matrix.py`:

1. Implement the code to extract the predictions and targets from a model and a dataset in `generate_confusion_data()`
2. Implement the code to generate the confusion matrix, and its normalized form in `generate_confusion_matrix()`
3. Plot the confusion matrix using `plot_confusion_matrix()` and try to understand how your model is performing, and where it falls short. We'll use this later on for the report.

3 Part 3: Fine-tuning a pre-trained network

Introduction

Since 2012, deep learning has revolutionized Computer Vision, improving the state-of-the-art performance significantly. Networks with residual connections, e.g., ResNet [2], were an important milestone in increasing the performance of image classification.

Training a relatively complex model like ResNet-1818 completely from scratch is sometimes overwhelming, especially when GPUs are not available. However, PyTorch has already incorporated the pre-trained weights of some influential models and provides you with the option of loading them directly.

3.1 ResNet model definition

In `my_resnet.py`, first load a pre-trained ResNet-18 using `model=torchvision.models.resnet18(pretrained=True)`.

Note that in the original ResNet model, there are 1000 output classes, while we have 15 for this project. We will need to retrieve some layers from the pre-trained model and concatenate them with our own custom layers. In `my_resnet.py`, when you have obtained the ResNet model, retrieve the convolutional layers (which helps you to detect the features) and the fully connected layers (which generates the classification scores), remove the last Linear layer, and replace it with a proper layer which could output the scores for 15 classes. You may find `module.children()` helpful here.

3.2 Fine-tuning ResNet

Notice that what we have done is merely defining the final layer with the correct dimensions, but the weights of the layer are just some random values and it won't be able to produce the desired scores. In the Jupyter Notebook, follow the starter code and train the newly defined network. Note that ResNet has millions of parameters, so training it from scratch could cause problems for your laptop with no GPU support, hence here we are only training it for 5 epochs and the expected running time is around 10 minutes. In addition, note that you do **NOT** need to update the weights for earlier layers of the model, therefore, freeze the weights of the convolution layers of the model, and then train the network following the template code in the notebook, observe the performance and answer the questions in the report. You may find `weight.requires_grad` and `bias.requires_grad` helpful here.

For Part 2, note that you are required to get a final testing accuracy of **85%** to receive full credit.

3.3 Analysis using confusion matrix

Once you have finished training your best model and have a good accuracy, we can analyze where it falls short. Find the top 3 most misclassified (prediction, target) class pairs, and provide examples of images that are misclassified as a result. Use the provided utilities from `confusion_matrix.py` to help you with this. You will need to include these, as well as some kind of reasoning why these might be misclassified.

4 Part 4: Multi-Label Classification

*This section is **optional**.*

Introduction

In this part you will use ResNet to perform multi-label classification on a subset of the dataset consisting of the *coast*, *highway*, *mountain*, *open country*, and *street* images. The goal of the model is to predict the attributes *clouds*, *water body*, *people*, *animals*, *natural*, *man-made*, and *vehicle* in a given image.

4.1 Dataset

You will implement the basic `Dataset` object, which helps to retrieve and prepare the data to be used by your model. Fill in the `MultiLabelImageLoader` class in `image_loader.py` similarly to how you did in Part 1.

4.2 Multi-Label ResNet model definition

In `multilabel_resnet.py`, you reuse parts of your code from `resnet.py`, except that the output layer should have a dimension of 7 (corresponding to the number of attributes) as opposed to 15. You should also change loss function to binary cross entropy and add a sigmoid activation function to the model.

4.3 Trainer/Accuracy

In `runner.py`, the multi-label ResNet utilizes a separate trainer class, `MultiLabelTrainer`, to train the data. This has been provided to you, and all you have to do is modify the accuracy function used in the training process.

In `dl_utils.py`, complete `compute_multilabel_accuracy`, where you are given an input model, and calculate the accuracy given the prediction logits and the ground-truth labels respectively. These functions will be used in the training process later.

Lastly, train the model using the starter code in Jupyter notebook, and modify the values of `lr` and `weight_decay` as needed.

Note that although you are not required to modify anything in the `runner.py`, it is still highly recommended to read through the code and understand how to properly define the training process, as well as record the loss history.

To receive credit for the `MultilabelResNet18` model in Part 4, you are required to get a validation accuracy of **90%**.

5 Writeup

For this project (and all other projects), you must do a project report using the template slides provided to you. Do ***not*** change the order of the slides or remove any slides, as this will affect the grading process on Gradescope and you will be deducted points. In the report you will describe your algorithm and any decisions you made to write your algorithm a particular way. Then you will show and discuss the results of your algorithm. The template slides provide guidance for what you should include in your report. A good writeup doesn't just show results–it tries to draw some conclusions from the experiments. You must convert the slide deck into a PDF for your submission, and then assign each PDF page to the relevant question number on Gradescope.

If you choose to do anything extra, add slides *after the slides given in the template deck* to describe your implementation, results, and analysis. You will not receive full credit for your extra credit implementations if they are not described adequately in your writeup.

Bells & whistles (extra credit)

Implementation of bells whistles can increase your grade on this project by up to 10 points (potentially over 100). The max score is 110.

For all extra credit, be sure to include quantitative analysis showing the impact of the particular method you've implemented. Each item is "up to" some amount of points because trivial implementations may not be worthy of full extra credit. Some ideas:

- Up to 5 pts: Sort the dataset by loss, and find the 10 examples with highest loss. Explain why these have highest loss.
- Up to 5 pts: Implement the basic residual block (referred to as "BasicBlock") of the ResNet-18, add it to your SimpleNet, and compare performance of your SimpleNet with this residual connection added. Does performance improve? Count the number of learnable parameters. What are the kernel sizes that are used? (See Figure 5, left, of ResNet paper for implementation details. Figures 2 and 3 and Table 1 may also be helpful).
- Up to 5 pts: Implement a network as small as possible that can still reach 80% accuracy (small as measured by learnable parameters). Note: you cannot simply re-use the pre-trained ResNet-18 or code for ResNet-18 (you need to find the minimum yourself via experimentation).

Rubric

- +30 pts: Part 1 SimpleNet
- +20 pts: Part 2 SimpleNetFinal
- +20 pts: Part 3 ResNet
- +10 pts (extra credit): Part 4 Multi-Label ResNet
- +30 pts: Report
- -5*n pts: Lose 5 points for every time you do not follow the instructions for the hand-in format

Submission format

This is very important as you will lose 5 points for every time you do not follow the instructions. You will submit two items to Gradescope:

1. `<your_gt_username>.zip` containing:
   (a) `src/vision/` - directory containing all your code for this assignment
   (b) `setup.cfg`: setup file for environment, no need to change this file
   (c) `additional_data/` - (optional) if you use any data other than the images we provide you, please include them here
   (d) `README.txt`: (optional) if you implement any new functions other than the ones we define in the skeleton code (e.g., any extra credit implementations), please describe what you did and how we can run the code. We will not award any extra credit if we can't run your code and verify the results.
2. `<your_gt_usernamme>_proj4.pdf` - your report

Do ***not*** install any additional packages inside the conda environment. The TAs will use the same environment as defined in the config files we provide you, so anything that's not in there by default will probably cause your code to break during grading. Do ***not*** use absolute paths in your code or your code will break. Use relative paths like the starter code already does. Failure to follow any of these instructions will lead to point deductions. Create the zip file using `python zip_submission.py --gt_username <your_gt_username>` (it will zip up the appropriate directories/files for you!) and hand it in with your report PDF through Gradescope (please remember to mark which parts of your report correspond to each part of the rubric).

Credits

Assignment developed by Esther Gu, Anant Joshi, Arvind Krishnakumar, Cusuh Ham, John Lambert, Vijay Upadhya, Wei Xiong Toh and James Hays.

References

[1] Svetlana Lazebnik, Cordelia Schmid, and Jean Ponce. "Beyond bags of features: spatial pyramid matching for recognizing natural scene categories". In: *IEEE Conference on Computer Vision & Pattern Recognition (CVPR '06)*. June 2006.

[2] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep Residual Learning for Image Recognition". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2016.