BMI 826 / CS 838 Homework Assignment 2

Oct 2019

1 Overview

This assignment focuses on convolutional neural networks for image classification. You will implement, design and train deep convolutional networks for scene recognition using PyTorch–an open source deep learning platform. Moreover, you will take a closer look at the learned network by (1) identifying important image regions for the classification; (2) generating adversarial samples that confuse your model; and (3) training models that can defend adversarial samples. This assignment is team-based. A team can have up to 3 students.

2 Setup

- Install Anaconda. We recommend using Conda to manage your packages.
- The following packages are needed: PyTorch ($\geq$1.2 with GPU support), OpenCV ($\geq$3), NumPy, Pillow and Tensorboard. And you are in charge of installing them.
- For the visualization of the results, you will need Tensorboard and TensorFlow (a dependency of Tensorboard). You don't need TensorFlow-gpu in this case.
- You can debug your code and run experiments on CPUs. However, training a neural network is very expensive on CPUs. We recommend using GPU computing for this project. Please setup your team's cloud instance. Do remember to shutdown the instance when it is not used!
- You will need to download the MiniPlaces dataset for Part II & III of the project. We have included the downloading script. Run download_dataset.sh in the assignment folder. All data will be downloaded under ./data/.
- You will need to fill in the missing code in:
  ./code/student_code.py
- You will need to submit your code, results and a writeup. You can generate the submission once you've finished the assignment using:
  python ./zip_submission.py

3 Details

This assignment has three parts. An autograder will be used to grade some parts of the assignment. Please follow the instructions closely.

3.1 Understand Convolutions

In this part, you will need to implement the 2D convolution operation–the fundamental component of deep convolutional neural networks. Specifically, a 2D convolution is defined as

$$\mathcal{Y} = \mathcal{W} *_S \mathcal{X} + b \tag{1}$$

- **Input:** $\mathcal{X}$ is a 2D feature map of size $C_i \times H_i \times W_i$ (following PyTorch's convention). H_i and W_i are the height and width of the 2D map and C_i is the input feature channels.
- **Weight:** $\mathcal{W}$ defines the convolution filters and is of size $C_o \times C_i \times K \times K$, where K is the kernel size. For this part, we only consider squared filters.
- **Stride:** $*_S$ is the convolution operation with stride S. S is the step size of the sliding window when $\mathcal{W}$ convolves with $\mathcal{X}$. For this part, we only consider equal stride size along the height and width. $\mathcal{W}$ is the parameter that will be learned from data.
- **Bias:** b is the bias term of size C_o. b is added to every spatial location $H \times W$ after the convolution. Again, b is the parameter that will be learned from data.
- **Padding:** Padding is often used before the convolution. We only consider equal padding along all sides of the feature map. A (zero) padding of size P adds zeros-valued features to each side of the 2D map.
- **Output:** Y is the output feature map of size $C_o \times H_o \times W_o$, where $H_o = \lfloor \frac{H_i + 2P - K}{S} \rfloor + 1$ and $W_o = \lfloor \frac{W_i + 2P - K}{S} \rfloor + 1$.

Helper Code: We have provided you helper functions for the implementation (./code/student_code.py). You will need to fill in the missing code in the class **CustomConv2DFunction**. You can use the fold / unfold functions and any matrix / tensor operations provided by PyTorch, except the convolution functions. You do not need to modify the code in the class **CustomConv2d**. This is the module wrapper for your code.

Requirements: You will need to implement both the forward and backward propagation for this 2D convolution operation. The implementation should work with any kernel size K, input and output feature channels C_i/C_o, stride S and padding P. Importantly, your implementation need to compute $\mathcal{Y}$ given input $\mathcal{X}$ and parameters $\mathcal{W}$ and b, and the gradients of $\frac{\partial \mathcal{Y}}{\partial \mathcal{X}}$, $\frac{\partial \mathcal{Y}}{\partial \mathcal{W}}$ and $\frac{\partial \mathcal{Y}}{\partial b}$. All derivations of the gradients can be found in our course material, except $\frac{\partial \mathcal{Y}}{\partial b}$ (provided in helper code). In your write up, please describe your implementation.

Testing Code: How can you make sure that your implementation is correct? You can compare your forward / backward propagation results with PyTorch's own Conv2d implementation. You can also compare your gradients with the numerical gradients. We included a sample testing code in ./code/test_conv.py. Please make sure your code can pass the test.

3.2 Design and Train a Convolutional Neural Network

In the second part, you will design and train a convolutional neural network for scene classification on MiniPlaces dataset.

MiniPlaces Dataset: MiniPlaces is a scene recognition dataset developed by MIT. This dataset has 120K images from 100 scene categories. The categories are mutually exclusive. The dataset is split into 100K images for training, 10K images for validation and 10K for testing. You can download the dataset by running download_dataset.sh in the assignment folder. The images and annotations will be located under ./data. We will evaluate top-1/5 accuracy for the performance metric. For more details about the dataset, please refer to their github page `https://github.com/CSAILVision/miniplaces`.

Helper Code: We have provided you helper code for training and testing a deep model (./code/main.py). You will have to run this script many times but you are unlikely to modify this file. For your reference, a simple neural network is implemented by the class **SimpleNet** in ./code/student_code.py. You will need to modify this class for this part of the project.

Monitor the Training: All intermediate results during training, including training loss, learning rate, train/validation accuracy are logged into files under ./logs. You can monitor and visualize these variables by using
tensorboard --logdir=../logs
We recommend copying the "logs" folder to a local machine and use Tensorboard locally for the curves. Thus, you can avoid to setup a Tensorboard server on the cloud. **Make sure you backup and clean the log folder for each run of the experiment.** The curves should be included in your writeup.

Requirements: You will design and train a deep network for scene recognition. You model must be trained using only the training set, e.g., using labels of the validation set for training, or using ImageNet pre-trained weights is not allowed, unless otherwise specified.

The Simple Network: Let us start by training our first deep network from scratch! No coding is needed in this section–we provide the dataloader and a simple network you can use. You can start by running
python ./main.py ../data --epochs=60
You will need to use GPU for this training. And it will take a few hours and give you a model with 40%+ top-1 accuracy on the validation set. Do remember

to put your training inside a container, e.g., tmux, such that your process won't get killed when you SSH session is disconnected. You can also use
watch -n 0.1 nvidia-smi
to monitor GPU utilization and memory consumption. Once the training is done, your best model will be saved as ./models/model_best.pth.tar. You can evaluate this model by
python ./main.py ../data --resume=../models/model_best.pth.tar -e

Dive into the Simple Network: While waiting for the training of the model, let us take a look at the code. Please describe the training process implemented in our code in your writeup. You will need to address the following questions: Which loss function/optimization method is used? How is the learning rate scheduled? Is there any regularization used?

An important aspect of learning based methods in computer vision is the design of proper evaluation metric. For MiniPlaces, we report top-k accuracy. Other possible metrics for image classification include average per-class accuracy, mean average precision (mAP).[1] Oftentimes, the choice of a metric will depend on the dataset, i.e., the data distribution. Please contrast these metrics and discuss when they should be used in your writeup.

Train with Your Own Convolutions: As a step forward, let us try to use our own convolution to replace PyTorch's version and train the model for 10 epochs. This can be done by
python ./main.py ../data --epochs=10 --use-custom-conv
How is your implementation different from PyTorch's version in terms of memory consumption, training speed and loss curve? What are the factors that might have produced the difference? Describe your findings in the writeup.

Design Your Own Network: Now let us try to improve our simple network. The current version is a combination of convolution, ReLU, max pooling and fully connected layers. Your goal is to design a better network for this recognition task. There are a couple of things you can explore here. For example, you can add more convolutional layers [6], yet the model might start to diverge in the training. This divergence can be avoided by adding residual connections [2] and/or batch normalization [3]. You might also want to try the multi-branch architecture in Google Inception networks [7]. You can also tweak the hyperparameters for training, e.g., learning rate, weight decay, training epochs, type of data augmentations. *In all cases, you should implement your network in student_code.py and call main.py for training.* A good architecture should balance between efficiency and accuracy. Please justify your design of the model and the training, and present your results in the writeup, including training curves and training/validation accuracy.

[1] Average Precision (AP) is the area under the precision-recall curve for a single class. mAP is the mean of per-class AP across all classes

Fine-Tune a Pre-trained Model: As the final step, we will fine-tune a residual network (18 layers) pre-trained on ImageNet [2]. The implementation is included in the helper code. And you can run
python ./main.py ../data --epochs=60 --use-resnet18
How is your model compared to this pre-trained ResNet18? You can look at the training curves and the training and validation accuracy. Please include the comparison in your writeup.

3.3 Attention and Adversarial Samples

In the final part, we will look at attention maps and adversarial samples. They present two critical aspects of deep neural networks: interpretation and robustness, and thus will help you gain insight about these networks.

Helper Code: Helper code is provided in ./code/main.py and student_code.py for visualizing attention maps and generating adversarial samples. For attention maps, you will need to fill in the missing code in class **GradAttention**. And for adversarial samples, you need to complete the class **PGDAttack**. For adversarial training, your need to modify our **SimpleNet**.

Requirements: You will implement methods for generating attention maps and adversarial samples. You will also need to implement adversarial training as a defense to adversarial samples.

Saliency Maps: Suppose you have a trained model. If you minimize the loss of the predicted label and compute the gradient of the loss w.r.t. the input, the magnitude of a pixel's gradient indicates how important that pixel is for the decision. You can create a 2D attention map by (1) computing the input gradient by minimizing the loss of the predicted label (most confident prediction); (2) taking the absolute values of the gradients; and (3) pick the maximum values across three channels. This method was discussed in [5]. Once you finished the coding, you can run
python ./main.py ../data --resume=../models/model_best.pth.tar -e -v
This command will evaluate your model using your trained model (assuming model_best.pth.tar) and visualize the attention maps. All attention maps will saved under ./logs. Again you can use Tensorboard
tensorboard --logdir=../logs
Now you will see a tab named "Image", where you can scroll the slide bar on top to see samples from different batches. You can also zoom in the image by clicking on it. Please include and discuss the visualization in your writeup.

Adversarial Samples: Interestingly, if you you minimize the loss of a wrong label and compute the gradient of the loss w.r.t. the input, you can create adversarial samples that will confuse the model! This was first presented in [8] and further analyzed in [1]. Let us use the least confident label as a proxy for the wrong label. And you will implement the Projected Gradient Descent under l_∞ norm in [4]. Specifically, PGD takes several steps of fast gradient sign method, and each time clip the result to the ϵ-neighborhood of the input. You will need to be a bit careful for this implementation. You do not want PyTorch to record your gradient operations in the computation graph. Otherwise, it will create a graph that grows indefinitely over time.

Again, you can call main.py once you complete the implementation
python ./main.py ../data --resume=../models/model_best.pth.tar -a -v
This command will generate adversarial samples on the validation set and try to attack your model. And you can see how the accuracy drops (significantly!). Moreover, adversarial samples will be saved in the "logs" folder. And you can use Tensorboard to check them. This time, you will find tabs "Org_Image" and "Adv_Image". Can you see the difference between the original images and the adversarial samples? Please discuss your implementation of PGD and present the results (accuracy drop and adversarial samples) in your writeup.

Adversarial Training: A deep model should be robust against adversarial samples. A possible solution is using adversarial training, as described in [1, 4]. The key idea is to generate adversarial samples and feed these samples into the network during training. To implement adversarial training, you can attach your PGD to the forward function in the **SimpleNet** (See the comments in the code for details). Unfortunately, this training can be 10x times more expansive than a normal training. To accelerate this process, you can (1) reduce the number of steps in PGD and (2) reduce the number of epochs in training. Your goal is to show that when compared to a model using normal training, your model using adversarial training has a better chance to defend adversarial attacks. Please discuss your experimental design, and present your results in the writeup.

4 Writeup

For this assignment, and all other assignments, you must submit a project report in PDF. Every team member should send the same copy of the report. For teams with more than one member, **please clearly identify the contributions of all members**. 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. In the case of this project, we have included detailed instructions for the writeup in each part of the project. You can also discuss anything extra you did. Feel free to add any other information you feel is relevant. A good writeup doesn't just show results, it tries to draw some conclusions from your experiments.

5 Handing in

This is very important as you will lose points if you do not follow instructions. Every time after the first that you do not follow instructions, you will lose 5%. The folder you hand in must contain the following:

- code/ - directory containing all your code for this assignment
- writeup/ - directory containing your report for this assignment.
- results/ - directory containing your results. Please include your model if you decide to participate in our challenge.

Do not use absolute paths in your code (e.g. /user/classes/proj1). Your code will break if you use absolute paths and you will lose points because of it. Simply use relative paths as the starter code already does. Do not turn in the data / logs / models folder. Hand in your project as a zip file through Canvas. You can create this zip file using *python zip_submission.py*.

References

[1] I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and harnessing adversarial examples. In *ICLR*, 2015.

[2] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *CVPR*, 2016.

[3] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *ICML*, 2015.

[4] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu. Towards deep learning models resistant to adversarial attacks. In *ICLR*, 2018.

[5] K. Simonyan, A. Vedaldi, and A. Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. In *ICLR*, 2014.

[6] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In *ICLR*, 2015.

[7] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *CVPR*, 2015.

[8] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing properties of neural networks. In *ICLR*, 2014.