BMI / CS 771 Fall 2022:
Homework Assignment 2

Yin Li

Oct 2022

1 Overview

This assignment is about learning convolutional and Transformer neural networks for image classification. You will implement, design and train different types of deep networks for scene recognition using PyTorch — an open source deep learning package. Moreover, you will take a closer look at the learned networks by (1) identifying important image regions for classification; (2) generating adversarial samples to confuse your model; and (3) training models to defend against those adversarial samples (bonus). This assignment is team-based and requires cloud computing. A team can have up to 3 students. The assignment has a total of 12 points with 2 bonus points. Details and rubric are provided in Section 3.

2 Setup

- We recommend using Conda to manage your packages.
- The following packages are needed: PyTorch ($\geq$1.10 with GPU support), OpenCV ($\geq$3), NumPy, gdown, and Tensorboard. Again, you are in charge of installing them.
- You can debug your code and run experiments on CPUs. However, training a neural network is very expensive on CPUs. GPU computing is thus required for this project. Please setup your team's cloud instance. **Do remember to shutdown the instance when it is not used!**
- You will download the MiniPlaces dataset for Part II & III of the project. We have included a script for downloading and unpacking the dataset (assuming all dependencies are installed). Simply run
  sh ./download_dataset.sh
- You will need to fill in the missing code in:
  ./code/student_code.py

- Your submission should include the code, results and a writeup. The submission can be generated 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 (2 Pts)

In the first part, you will implement the 2D convolution operation — a 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. *We only consider squared filters.* $\mathcal{W}$ will be learned from data.
- **Stride:** $*_S$ denotes the convolution operation with stride S, where S is the step size of the sliding window when $\mathcal{W}$ convolves with $\mathcal{X}$. *We will only consider equal stride size along the height and width.*
- **Bias:** b is the bias term of size C_o, and is added to every spatial location $H \times W$ after the convolution. Again, b 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 some 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**. Please do not modify the code in the class **CustomConv2d**. This is the module wrapper for your code.

Requirements: You will implement both the forward pass and the 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 must 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 $\frac{\partial \mathcal{Y}}{\partial b}$ (implementation provided in helper code). In your writeup, please describe your implementation.

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

3.2 Design and Train a Deep Neural Network

In the second part, you will design and train convolutional and Transformer neural networks for scene classification using 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 will need to manually download this dataset. The images and annotations will be located under ./data. We will evaluate top-1/5 accuracy on the validation set as our performance metric. For more details about the dataset, please refer to their github page `https://github.com/CSAILVision/miniplaces`.

Downloading the Dataset: A script has been included to download and unpack the dataset. Please follow the instruction in Section 2 (Setup). If successful, the data folder will contain two sub-folders "images" and "objects," as well as two text files "train.txt" and "val.txt," in addition to data.tar.gz.

Helper Code: We have provided helper code for training and testing a deep model (./code/main.py). You will run this script many times but it is unlikely that you will need to modify this file. *If you do modify this file, please describe the modification and its justification in your writeup.* To check how to use this function, run
python ./main.py --help

Other helper code include

- Dataloader (./code/custom_dataloader.py) for MiniPlaces Dataset.
- Image augmentations (./code/custom_transforms.py), which also provide a reference solution for HW1.
- Transformer blocks (./code/custom_blocks.py) for the implementation of vision Transformer models.

Finally, a simple convolutional neural network is implemented by **SimpleNet** in ./code/student_code.py. You will likely need to modify this class for designing your own model.

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 after each run of the experiment.* These 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. Using labels of the validation set for training, or using ImageNet pre-trained weights is not allowed, unless otherwise specified.

A Simple Convolutional Network (1 Pt): 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 to start with. You can run
python ./main.py ../data --epochs=60

Importantly, GPU is needed for the training. The training might take a few hours and will 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 or screen, such that your process won't get killed when you SSH session expires.* You can also use
watch -n 0.1 nvidia-smi
to monitor GPU utilization and memory consumption. Once the training is done, the best model will be saved as ./models/model_best.pth.tar. *The saved models will be overwritten for each new experiment. Make sure you backup the models when necessary.* You can evaluate this model by
python ./main.py ../data --resume=../models/model_best.pth.tar -e

Train with Your Own Convolutions (1 Pt): As a step forward, we will 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? Please describe your findings in the writeup.

Implement a Vision Transformer (2 Pt): Beyond a convolutional network, let us now implement a Transformer based model [1]. Transformer blocks that

supports local window self-attention, are already implemented in our helper code (./code/custom_blocks.py). You will need to review the implementation and figure out how to use these blocks to build a vision Transformer using **SimpleViT** in ./code/student_code.py. To train the Transformer model, run
python ./main.py ../data --epochs=90 --wd 0.05 --lr 0.01 –use-vit

Here we decrease the learning rate and increase the weight decay and number of training epochs, partially due to the use of AdamW optimizer [5]. With our default parameters, the model should give a performance level similar to the simple convolutional network. You are welcome to play with the default parameters of **SimpleViT** as well as the hyperparameters for training, yet keep in mind that the model could take a much longer time to train. In your writeup, please describe the design of your vision Transformer model, its training scheme, and the results.

Design Your Own Network (1 Pts): Now let us try to improve the simple networks. You can choose to focus on either the convolutional network or the Transformer network. 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 [8], yet the model might start to diverge in the training. This divergence can be avoided by adding residual connections [3] and/or batch normalization [4]. You might also want to try different type of self-attention for the Transformer. You can also tweak the hyper-parameters for training, e.g., initial learning rate, weight decay, training epochs, type of data augmentations. Most of the hyper-parameters can be passed as an argument to main.py. *In all cases, you should implement your network in student_code.py and call main.py for training.* A good architecture strikes a balance between efficiency and accuracy.

Please describe and justify your design of the model and the training scheme, and present your results in the writeup (including training curves and training/validation accuracy).

Fine-Tune a Pre-trained Model (1 Pts): As the final step, we will fine-tune a residual network (18 layers) pre-trained on ImageNet [3]. 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? For a in-depth comparison, 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 us gain insight about these networks. For this Section,

we will only consider convolutional 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 fill in the missing code in class **GradAttention**. For adversarial samples, you need to complete the class **PGDAttack**. For adversarial training, your will have to modify part of the **SimpleNet**.

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

Saliency Maps (2 Pts): 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 the importance of the pixel 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 color channels. This method was discussed in [7]. Once you finished the coding, run
python ./main.py ../data --resume=../models/model_best.pth.tar -e -v

This command will evaluate your trained model (assuming model_best.pth.tar) and visualize the attention maps. All attention maps will be saved under ./logs. Again you can use Tensorboard to visualize the results
tensorboard --logdir=../logs

Now you will see a new tab named "Image", where you can scroll the slider 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 (2 Pts): Interestingly, by minimizing the loss of an incorrect label and compute the gradient of the loss w.r.t. the input, one can create adversarial samples that will confuse a model! This was first presented in [9] and further analyzed in [2]. We will consider the least confident label as a proxy for the incorrect label. And you will implement the Projected Gradient Descent under l_∞ norm in [6]. Specifically, PGD takes several steps of fast gradient sign method. At each time step, PGD also clips the result to the ϵ-neighborhood of the input. This implementation, however, requires some thoughts. The gradient operations should not be recorded by PyTorch, as doing so will create a computational 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? What if you increase the number of iterations and reduce the error bound (ϵ)? Please discuss your implementation of PGD and present the results (accuracy drop and adversarial samples) in your writeup.

Adversarial Training (Bonus +2 Pts): A deep model should be robust against adversarial samples. As we discussed in the lecture, a possible solution is using adversarial training, as described in [2, 6]. 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, we recommend to (1) reduce the number of steps in PGD and (2) reduce the number of epochs in training. The 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. For 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 other information you feel is relevant.

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.

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] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations*, 2021.

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

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

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

[5] I. Loshchilov and F. Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations*, 2018.

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

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

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

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