BMI / CS 771 Fall 2025: Homework Assignment 2

Sep 2025

1 Overview

In this assignment, you will design, implement, train, and evaluate different types of deep neural networks for image classification using PyTorch. Moreover, you will take a closer look at the learned networks by (1) generating adversarial samples to confuse the models; and (2) training models to defend against those adversarial samples (bonus). This assignment is team-based and requires cloud computing. A team is expected to have 2-3 students, unless otherwise approved by the instructor. 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$2.0 with GPU support), OpenCV ($\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 need to download the MiniPlaces dataset for this project. We have included a script for downloading and unpacking the dataset (assuming all dependencies are installed). Simply run
  sh ./download_dataset.sh
- To complete the assignment, 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 file 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.**

Dataset: In this assignment, we will use MiniPlaces, 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 download this dataset under ./data (see details below). Top-1/5 accuracy on the validation set will be considered as the performance metric. For more details about the dataset, please refer to this github page `https://github.com/CSAILVision/miniplaces`.

Downloading the Dataset: We have included a script to download and unpack the dataset. Please follow the instruction in Section 2. 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

Hints: To test-run your code locally, you can set --gpu to -1.

Other helper code include

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

Please make sure that you review the helper code before attempting this assignment.

3.1 Convolutional Neural Networks

3.1.1 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 $W_o = \lfloor \frac{W_i+2P-K}{S} \rfloor + 1$.

Starter 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**. The implementation 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 pass for this 2D convolution operation. Your implementation is expected to 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 slides, except $\frac{\partial \mathcal{Y}}{\partial b}$ (whose implementation is provided in helper code).

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

3.1.2 Design and Training Convolutional Neural Networks

Requirements: You will design and train convolutional networks 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 here — 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 some time and will yield 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 your SSH session expires.* You can 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. You can evaluate this model by
*python ./main.py ../data \
--resume=../logs/exp_timestamp/models/model_best.pth.tar -e*

Monitor the Training: All intermediate results during training, including training loss, learning rate, train/validation accuracy are logged into ./logs. Specifically, each run will create a folder *exp_timestamp* in ./logs. You can monitor and visualize these variables by using
tensorboard --logdir=../logs/exp_timestamp

Resume Training from a Checkpoint: Under rare cases, the training might get interrupted. We have implemented checkpointing during training, and you can resume from a previous checkpoint using
*python ./main.py ../data --epochs=60
--resume=../logs/exp_timestamp/models/checkpoint.pth.tar*

We recommend copying the ../logs folder to a local machine and use Tensorboard locally for plotting curves that are needed for the writeup. Thus, you can avoid to setup a Tensorboard server on the cloud. *You will have to manually manage this log folder and record different runs of your experiments.* For example, older experiments can be removed to save some storage space. **Please include (1) training curves (loss, train/val accuracy, and learning rate) and (2) evaluation results in the writeup.**

Train with Your Own Convolutions (1 Pt): As a step forward, you will use your own convolution (Section 3.1.1) to replace PyTorch's version and train the model for 10 epochs, assuming that part one of the assignment was successfully completed. 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 theses differences? **Please describe your findings in**

the writeup.

Design Your Own Convolutional Network (1 Pt): Now let us try to improve the simple networks. The goal is to design a "better" network for this recognition task. There are a couple of things one can explore here. For example, you can add more layers [10], yet the model might start to diverge in the training. You might want to try batch normalization [6] and skip connections [4]. 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 (a) your design of the model; (b) the training scheme, and (3) your results in the writeup. The training curves and training/validation accuracy should also be included.**

Fine-Tune a Pre-trained Model (1 Pt): Next, we will fine-tune a residual network (18 layers) pre-trained on ImageNet [4]. The implementation is included in the helper code. You can run
python ./main.py ../data --epochs=60 --use-resnet18

How is your "best" 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.**

3.2 Vision Transformers

3.2.1 Understanding Self-Attention and Transformer Block

Self-Attention (2 Pts): Key to a Transformer lies in self-attention [12]. While self-attention has been discussed in our lecture, we briefly review the key idea here. Concretely, self-attention computes a weighted average of features with the weight proportional to a similarity score between pairs of input features. Given $\mathbf{Z}^0 \in \mathbb{R}^{T\times D}$ with T time steps of D dimensional features, $\mathbf{Z}^0$ is projected using $\mathbf{W}_Q \in \mathbb{R}^{D\times D_q}$, $\mathbf{W}_K \in \mathbb{R}^{D\times D_k}$, and $\mathbf{W}_V \in \mathbb{R}^{D\times D_v}$ to extract feature representations $\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$, referred to as query, key and value respectively with $D_k = D_q$. The outputs $\mathbf{Q}$, $\mathbf{K}$, $\mathbf{V}$ are computed as

$$\mathbf{Q} = \mathbf{Z^0}\mathbf{W}_Q, \quad \mathbf{K} = \mathbf{Z^0}\mathbf{W}_K, \quad \mathbf{V} = \mathbf{Z^0}\mathbf{W}_V. \tag{2}$$

The output of self-attention is given by

$$\mathbf{S} = \text{softmax}\left(\mathbf{Q}\mathbf{K}^T/\sqrt{D_q}\right)\mathbf{V}, \tag{3}$$

where $\mathbf{S} \in \mathbb{R}^{T\times D}$ and softmax is performed *row-wise.* A multihead self-attention (MSA) further adds several self-attention operations in parallel. **You will implement the MSA by filling in the missing code for the class Attention**. We have provided some helper code in this class to begin with.

Transformer Block with Local Self-Attention (1 Pt): A main advantage of MSA is the ability to integrate long-range context across the full sequence, yet such a benefit comes at the cost of computation. A vanilla MSA has a complexity of $O(T^2D + D^2T)$ in both memory and time, and is thus highly inefficient for long sequences derived from high-resolution images. A simple solution, as you will implement in this assignment, is to consider local window self-attention [1, 7]. This is done by limiting the self-attention within a local window. Such a local self-attention significantly reduces the complexity to $O(W^2TD + D^2T)$ with W the local window size ($\ll T$). **You will implement a Transformer block that supports local self-attention in the class TransformerBlock**. Specifically, this implementation will follow a Pre-LN Transformer block [13] and additionally support stochastic depth regularization [5]. To aide your implementation, we have provided some helper code, in particular *window_partition* and *window_unpartition*. See further details in the code and comments.

3.2.2 Design and Implement Vision Transformers (1 Pt)

With the implementation of self-attention and Transformer block, we are now ready to implement a Transformer model [2]. **You will implement a vision Transformer in the class SimpleViT**. 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 [8]. With our default parameters, the model should give a performance level similar to the simple convolutional network. To further optimize the model, you are encouraged to modify the default parameters of **SimpleViT** as well as the hyperparameters for training. A caveat is that the model may take much longer time to train in comparison to convolutional networks. Over-optimization is not necessary. **In the writeup, please describe the design of your Transformer model, its training scheme and results.**

3.3 Adversarial Samples

Finally, we will take a look at adversarial samples that closely related to the robustness of deep models. For this section, *we will only consider convolutional networks.* You will implement methods for generating adversarial samples, and optionally for adversarial training as a defense to adversarial samples.

Adversarial Samples (2 Pts): 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 look very similar to the original image yet are likely to confuse a deep model! This was first described in [11] and further analyzed in [3]. *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 [9] in the class **PGDAttack**. 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, assuming a trained model

```
python ./main.py ../data \
--resume=../logs/exp_timestamp/models/model_best.pth.tar -a -v
```

This command will generate adversarial samples on the validation set and try to attack a trained model. You can see how the accuracy drops (significantly!). Moreover, adversarial samples will be saved in the "logs" folder. You can use Tensorboard to check them. This time, you will find new tabs "Org_Image" and "Adv_Image". You can use the slide bar to navigate across different mini-batches. 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. One possible solution is using adversarial training, as described in [3, 9]. The key idea is to generate adversarial samples and feed these samples into the network during training. To implement adversarial training, your will have to modify part of the **SimpleNet**. 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

Please follow the following instructions, or otherwise you will lose 5% of points. 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 that are not included in the writeup.

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**. Training curves and results are expected to be embedded in the writeup. Hand in your project as a zip file through Canvas. You can create this zip file using *python zip_submission.py*.

References

[1] K. M. Choromanski, V. Likhosherstov, D. Dohan, X. Song, A. Gane, T. Sarlos, P. Hawkins, J. Q. Davis, A. Mohiuddin, L. Kaiser, et al. Rethinking attention with performers. In *ICLR*, 2020.

[2] 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 *ICLR*, 2021.

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

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

[5] G. Huang, Y. Sun, Z. Liu, D. Sedra, and K. Q. Weinberger. Deep networks with stochastic depth. In *ECCV*, 2016.

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

[7] Y. Li, H. Mao, R. Girshick, and K. He. Exploring plain vision transformer backbones for object detection. In *ECCV*, 2022.

[8] I. Loshchilov and F. Hutter. Decoupled weight decay regularization. In *ICLR*, 2018.

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

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

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

[12] A. Vaswani. Attention is all you need. *NeurIPS*, 2017.

[13] R. Xiong, Y. Yang, D. He, K. Zheng, S. Zheng, C. Xing, H. Zhang, Y. Lan, L. Wang, and T. Liu. On layer normalization in the transformer architecture. In *ICML*, 2020.