Homework 3 — Convolutional Neural Networks

1 Scope

This assignment is on implementing a convolutional neural network, from its basic concept to practical usage.

2 Instructions

- Deadline for this assignment is **Feb 23** via Gradescope
- The submission portal will remain open till end of the semester. Do keep in mind the late submission policy (Check slides from first lecture)
- Your submission has two parts: (i) a PDF report that details all the requested deliverables and (ii) the code for the assignment. You will upload the PDF report in gradescope, marking which parts of the report corresponds to which deliverable. The code will be uploaded separately (instructions TBA), and we will run "software similarity" software on it to detect violations of the integrity policy.
- You can discuss this HW with others, but you are **not** allowed to share, borrow, copy or look at each other's codes.
- All code that you submit must be yours (except for the starter code that we provide). Specifically, you are **not** allowed to use any software packages or code from the internet or other resources, except for `numpy`, `torch`, `torchvision`, `scipy`, `PIL`, `matplotlib`, and other built-in packages such as `math`.

3 Problem Set

(Deliverable 1: Implement convolution and pooling.) You will now implement convolution and pooling layers in PyTorch by yourself.

- For this problem, you can use PyTorch built-in functions, but you *cannot* use existing convolution and pooling functions, such as `nn.Conv2d`, `nn.functional.conv2d`, `nn.MaxPool2d`, `nn.functional.max_pool2d`.
- Implement a customized version of `nn.functional.Conv2d`. It should support arguments including input, weight, stride, padding, and bias.
- Implement a customized version of `nn.Conv2d`. It should also support arguments including in_channels, out_channels, kernel_size, stride, padding, and bias.
- Implement a customized version of `nn.MaxPool2d`. It should also support arguments including kernel_size and stride.

We provided function handles in `codes/deliverable2-3/mytorch.py`. The implemented architecture will be used in Deliverable 3.

Deliverables for PDF report. Use your own implementation of nn.functional.Conv2d to repeat Deliverable 1.

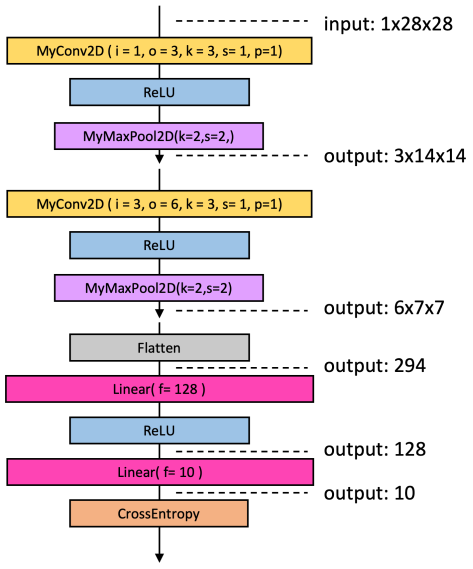

Figure 1: Architecture of a simple CNN.

(Deliverable 2: Classifying MNIST dataset with a CNN.) You will now use your convolution and pooling layers to build a simple CNN architecture for digits classification.

- Implement a small CNN with the architecture provided in Figure 1 using your own convolution and pooling layer.
- For arguments in convolution layers (yellow ones), i means in_channels, o means out_channels, k means kernel_size, s means stride, and p means padding. Also, set the bias argument to True.
- For arguments in pooling layers (purple ones), k means kernel_size and s means stride.
- Notice that we do not include softmax layer in Figure 1 as it has already been incorporated in `torch.nn.CrossEntropyLoss`. Please be careful with that.
- Uses 60,000 training images in torchvision.datasets.MNIST as the training dataset, and use the 10,000 test images as the validation dataset. Image sizes are 28×28.

We provided function handles in `codes/deliverable2-3/train.py`.

Deliverables for PDF report. Train the model for at least 5 epochs. Provide plots of loss and accuracy as a function of epochs in both training and validation datasets. Note that the validation accuracy is expected to reach at least 90%.

$$Accuracy = \frac{Number\ of\ correct\ predictions}{Total\ number\ of\ predictions}$$

(Deliverable 3: Implement ResNet in PyTorch) Now we are going to implement a signature CNN architecture ResNet. It is known for its residual blocks design, which adds a *short cut connection* between convolution layers to directly add input to later output layers as in Figure 2. It can counter the degradation problem caused by too many layers in the model.

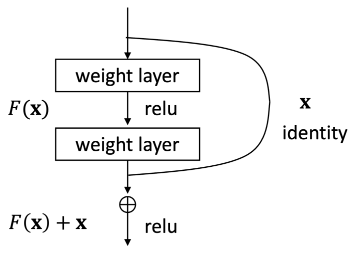

Figure 2: Design of a residual block as in the ResNet paper.

The full architecture of the ResNet you need to implement is provided in Figure 3. Here are some details:

- For this problem, you can use built-in function in PyTorch, such as `nn.Conv2d` or `nn.BatchNorm2d`.
- Do *not* use pre-trained ResNet model provided in PyTorch. Its architecture will be different from our instruction anyway.
- This architecture is tailored for the CIFAR-10 dataset as suggested in the original ResNet paper, with some minor modifications.
- Arguments in convolution blocks (yellow ones) corresponding to arguments in `nn.Conv2d`. Again, o means out_channels, k means kernel_size, and s means stride. Also, set the bias argument to False. You should figure out the rest of the arguments by yourself.
- Each convolution layer is first followed by a `BatchNorm2d` layer then a `ReLU` activation function.
- For the shortcut connection, since input and output dimensions are different in these residual blocks, we need to add 1x1 convolutions.
- The final residual blocks end with an average pooling. It means to average the 4×4 feature map in the final layer.
- Notice that we do not include softmax layer in Figure 3 as it has already been incorporated in `torch.nn.CrossEntropyLoss`. Please be careful with that.
- Use He initialization for each layer.
- Use an Adam optimizer.

The implemented architecture will be used in Deliverable 5. We provided function handles in `codes/deliverable4-6/model.py`. You do not need to include anything for the PDF report for this deliverable.

(Deliverable 4: Classifying CIFAR-10 with ResNet.) Now we are going to classify images in CIFAR-10 with the ResNet architecture implemented in the previous delivery.

- Directly uses 50,000 training images in torchvision.datasets.CIFAR10 as the training dataset, and use the 10,000 test images as the validation dataset. Image sizes are $3 \times 32 \times 32$.
- Apply data augmentation to the CIFAR10 dataset. Specifically, apply `train_transform` and `val_transform` when loading the training and validation dataset, respectively.

```
import torchvision.transforms as transforms
```

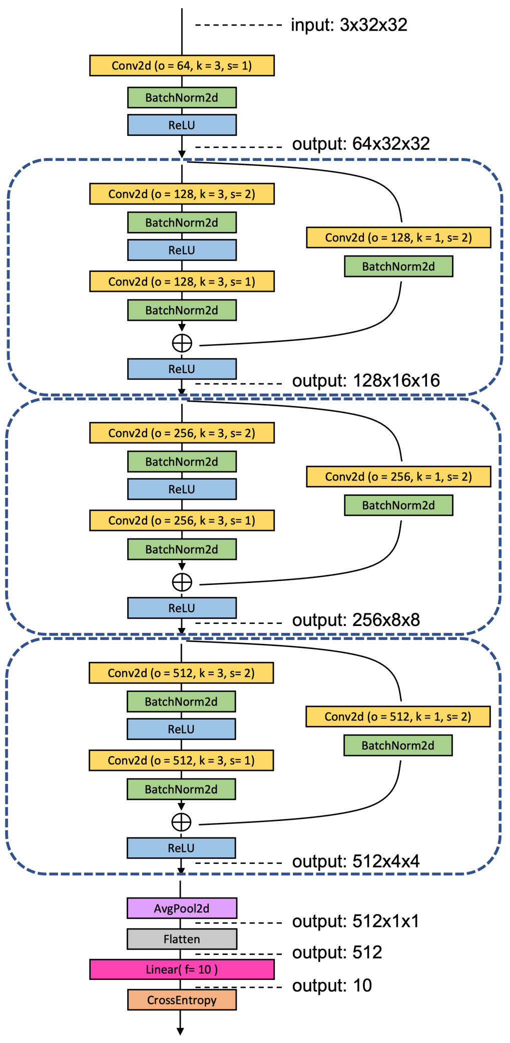

Figure 3: Architecture of a full ResNet.

```
normalize_param = dict(
        mean=[0.485, 0.456, 0.406],
        std=[0.229, 0.224, 0.225]
)
train_transform = transforms.Compose(
        [transforms.RandomResizedCrop(32, scale=(0.8, 1.0)),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize(**normalize_param,inplace=True)]
)
val_transform = transforms.Compose(
        [transforms.ToTensor(),
        transforms.Normalize(**normalize_param,inplace=True)]
)
```

We provided function handles in `codes/deliverable4-6/train.py`.

Deliverables for PDF report. Train the model for at least 30 epochs. Provide plots of loss and accuracy as a function of epochs in both training and validation datasets. Note that the validation accuracy is expected to reach at least 85%.

(Deliverable 5: Implementation of Class Activation Map (CAM).) Now we are going to implement an interesting method for model visualization. The overall mechanism and details could be found in this paper *Learning Deep Features for Discriminative Localization*. Implement the CAM technique and generate your results as shown in Figure 4 and Figure 5. We provided function handles in `codes/deliverable4-6/train.py`.

Deliverables for PDF report. Provide CAM results of at least 3 different images from validation set. You are expected to show the original/input images, CAM and the blending results, as shown in Figure 5. Hint: For blending results, you could check `alpha` argument in `matplotlib.pyplot.imshow`.

(Note: The CAM has a smaller size compared to your input image, so you will need to use interpolation or upsampling techniques to overlay it on your input image and perform blending. Depending on the interpolation techniques you use, you can get a smooth CAM visualization or a somewhat uneven looking one, as in Figure 6. However, as long as your CAM makes sense (the correct part of the image, i.e. the bird is being highlighted in both visualizations), don't worry too much about it. Feel free to try out interpolation functions like torch.nn.functional.interpolate and scipy.ndimage.zoom) and check what works best for you.)

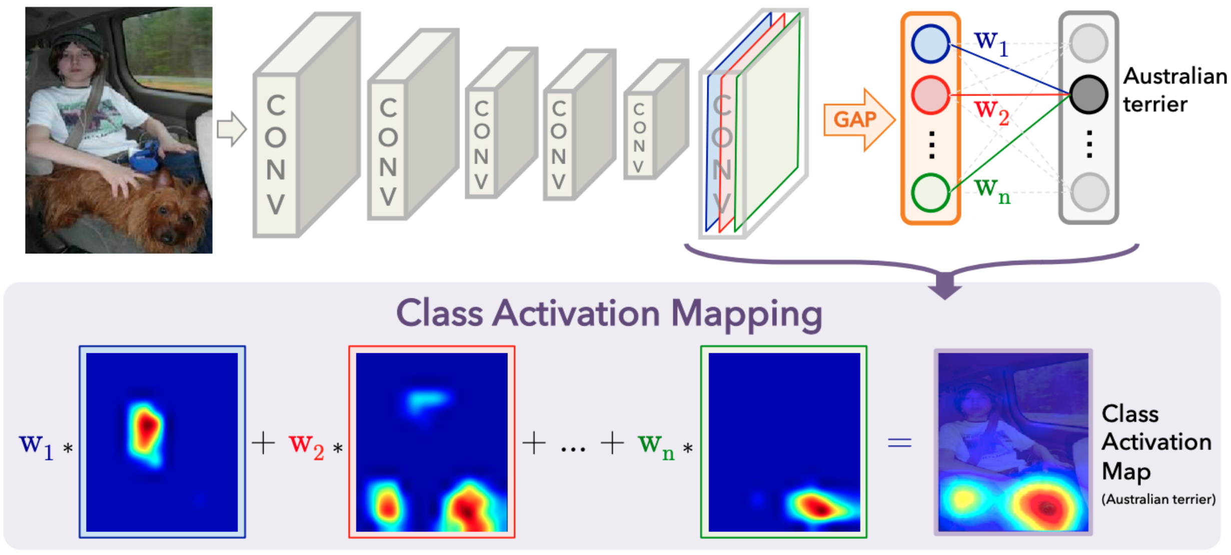

Figure 4: Class Activation Mapping pipeline from Bolei Zhou's paper.

(Deliverable 6: Evasion attacks) We have learned about evasion attacks in recitation. Alice and Bob, two machine learning practitioners, have trained their own MNIST classifiers, namely model A and model B. They would like to see how robust their models are against evasion attacks, specifically with L_∞ distance limit. They need your help. Feel free to use code from recitation, with citations.

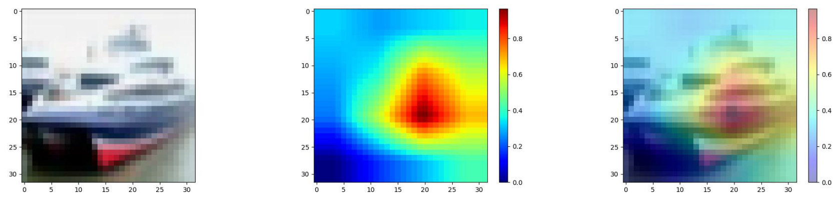

Figure 5: CAM results example (left: Input image. middle: CAM result. right: Blending result.

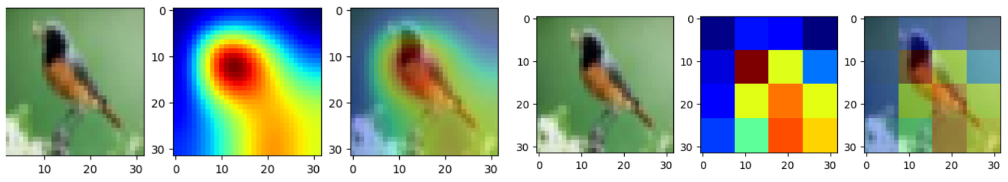

Figure 6: CAM results example depending on interpolation techniques

We have provided you starter code to load and evaluate the benign accuracy of the two models. Here are your tasks:

1. Task 1. Implement an untargeted attack as we did in recitation 3. You may compute backward propagation up to twenty times (less is fine) on each input instance.
2. Task 2. Implement an targeted attack that induces the model to predict all "1"s as "8". You may compute backward propagation up to 100 times (less is fine) on each input instance.
3. Task 3. Implement at least one improved version of the targeted attack in *Task 2*. You may use the approach we described in the recitation. Note: performing more backward propagation is *not* considered an improvement.
4. Evaluate your attacks implemented in Task 1, 2, and 3. The performance of attacks depends on the L_∞ distance limit ϵ, so you may want to plot the success rate against different choices of ϵ. Specifically, you need to
   - Plot and verify that the generated adversarial examples are inconspicuous to humans, especially at smaller choices of ϵ; and
   - Compute the probability that the adversarial examples actually induce desired misclassifications (a.k.a. success rate). Compare the success rates of the untargeted attack (Task 1) against models A and B. Compare the success rates of the targeted attacks (Task 2 and Task 3) against models A and B. What do you find?

When we evaluate the performance of these attacks, shall we use the training set or the testing set? Why?

Bonus: You may report your success rates of your improved targeted attack (Task 3), at $\epsilon = \{8, 16\}$, against {model A, model B}. That is being said, report four numbers (with two decimal places). We may run your code to verify if your numbers can be reproduced. Submissions with top 10 performance get extra credit.