1 Outline

In this assignment, you are asked to design CIFAR-10 classifier using PyTorch library.

2 Specification

In this assignment, you are asked to write a Python code to implement a classifier for CIFAR-10 dataset. Specifically, you need to implement a variant of ResNet whose blocks can model identity mapping. The implementation details are provided in the following sections.

3 Motivation

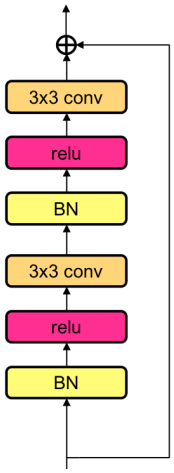

Figure 1: CNN architecture.

3.1 Network Architecture

We consider the architecture from paper "Identity Mappings in Deep Residual Networks" link: `https://arxiv.org/pdf/1603.05027.pdf`. The architecture uses and improves residual network which uses skip connections. The improved ResNet emphasizes the ability to model identity function. The basic residual block is given in

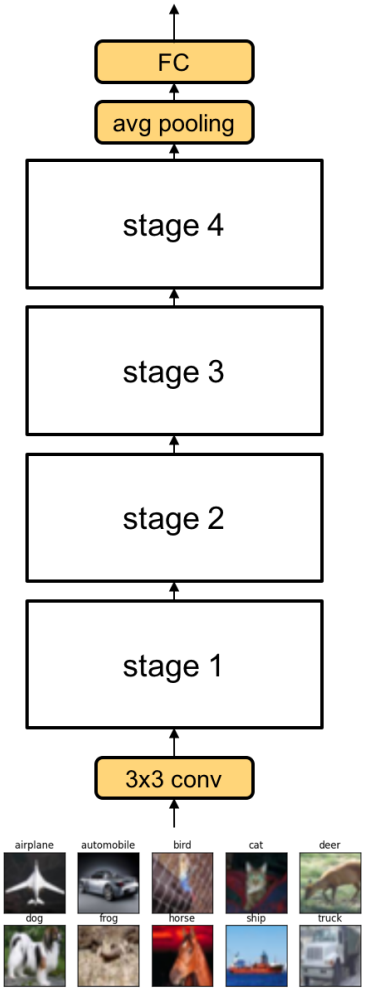

Figure 2: Network

Fig. 1; by letting all the weights in the convolutional layers to 0, the block can model identity, that is, $F(x) = x$. The overall network has following structure (shown in Fig. 2), in order:

1. convolutional layer
2. 1st stage: n_1 blocks
3. 2nd stage: n_2 blocks
4. 3rd stage: n_3 blocks

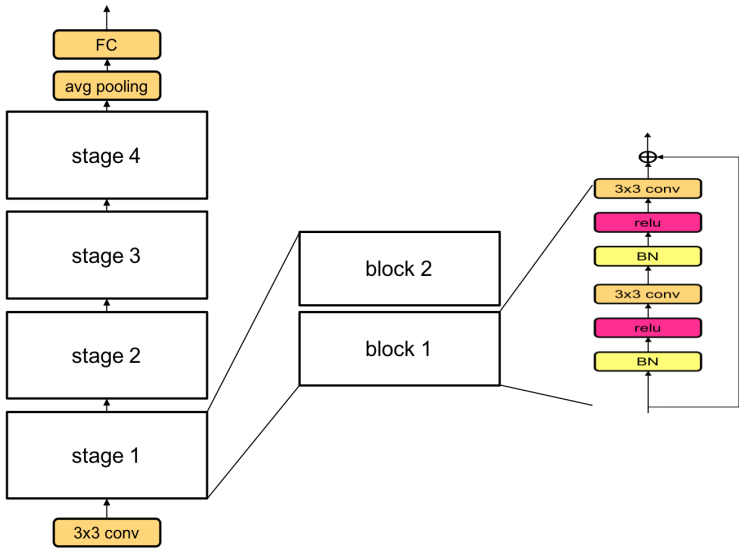

Figure 3: Stage 1 architecture.

5. 4th stage: n_4 blocks
6. average pooling layer
7. fully connected layer (input to 10 classes)

We inspect the network components one-by-one.

3.1.1 convolutional layer

The input to the network is a (batch of) CIFAR-10 image, which has 3 channels and spatial size of 32x32. The first layer is 3x3 convolutional layer. Details of the layer is as follows:

- Filter size: 3×3, stride: 1, padding: 1
- input channel size: 3
- output channel size: 64

You can implement it with `nn.Conv2d` in PyTorch.

3.1.2 1st stage

The 1st stage consists of n_1 residual blocks. The network for stage 1 is given in Fig. 3 which uses the basic residual block in Fig. 1. All the blocks have the same structure, where the convolutional layer is given by

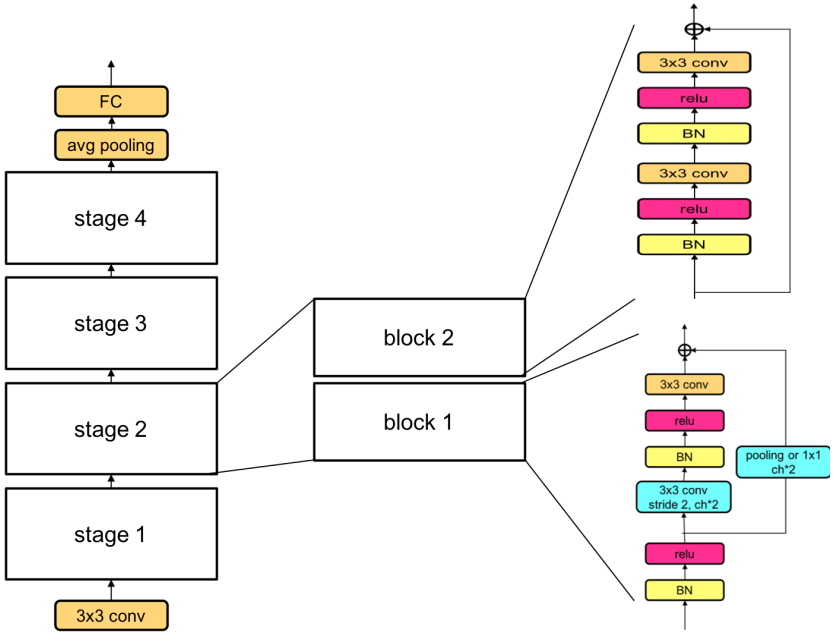

Figure 4: Stage 2 architecture.

- Filter size: 3×3, stride: 1, padding: 1
- input, output channel size: 64

Also the blocks uses batch normalization (BN) and ReLU activation, as given in Fig. 1.

Note: In the homework, we will use $n_1 = 2$.

3.1.3 2nd stage

The 2nd stage consists of n_2 residual blocks. The network for stage 2 is given in Fig. 4. In the 2nd stage, **the spatial dimension is cut by half, but the channel size doubles.** Thus, **the first block of layer 2 slightly differs from the normal residual block due to change in the number of channels.**

The input tensor has shape (B, C, H, W) where B stands for batch size, C stands for channel size, H and W are height and width. Suppose the batch size is 4. The input to stage 2 has shape (4,64,32,32); the input to has 64 channels, with 32x32 map. After going through the first block (see Fig. 4), the shape changes to (4,128,16,16), i.e., 128 channels of spatial map of size 16 by 16.

The first block of stage 2 (block 1 in Fig. 4) consists of two convolutional layers, where the first one is given by

- Filter size: 3×3

- **stride: 2**
- padding: 1
- input channel size: 64
- **output channel size: 128**

and the second convolutional layer is

- Filter size: 3×3 stride: 1, padding: 1
- input and output channel size: 128

In block 1, the skip connection goes through the following layer to reduce spatial dimension by half and increase channel dimension by 2. We have two options:

1. Option A: uses pooling
    - max pooling, 2×2 filter size with stride of 2
    - double the channel size by padding zeros along the channel dimension.
2. Option B: uses 1-by-1 convolution
    - `Conv2d` module with filter size 1-by-1, **stride=2, input channel size=64, output channel size = 128**

For Option A: after applying stride-2 `maxpool` to input (4, 64, 32, 32), the output will be (4,64,16,16) – let's say the output is y. Here y needs to have the same shape with the output from convolutional layers, which is (4, 128, 16, 16), because they need to be added together. Thus, we concatenate y with the zeros of shape (4,64,16,16) along C (channel dimension). By concatenating two (4,64,16,16) tensors along C dimension, we create the tensor of shape (4, 128, 16, 16), which now can be added.

For Option B: by using 1-by-1 convolution with stride 2, we can directly reduce spatial dimension by half and increase channel size from 64 to 128.

Note: You can use either option in the homework.

The convolutional layers of the rest of blocks ($n_2 - 1$ blocks) have the following structure:

- Filter size: 3×3 stride: 1, padding: 1
- input and output channel size: 128

Note: In the homework, we will use $n_2 = 2$.

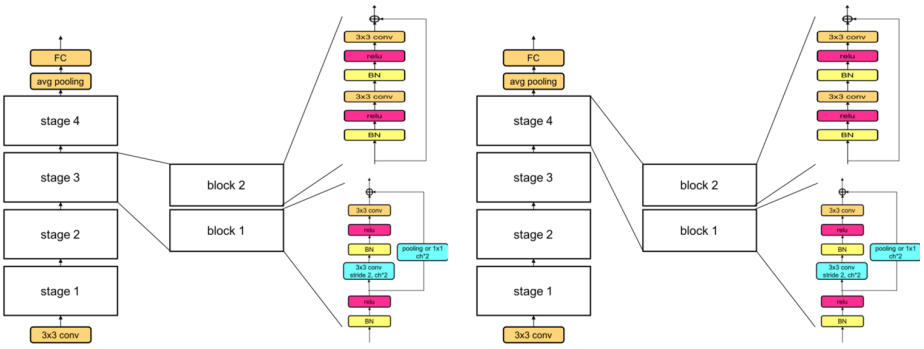

Figure 5: Stage 3 and 4 architecture.

3.1.4 3rd stage and 4th stage

The 3rd and 4th stage contains n_3 and n_4 blocks, and they have essentially the same structure as 2nd stage (see Fig 5): the first convolution of the first block will have stride 2 with skip connections having pooling or 1-by-1 conv. The number of channels differs from 2nd stage: in 3rd stage, channel number doubles from 128 to 256, in 4th stage, increases from 256 to 512. That is, the first convolution layer of the first block in each stage is

- Filter size: 3×3, **stride: 2**, padding: 1
- input channel size: 128 (stage 3), 256 (stage 4)
- **output channel size: 256 (stage 3), 512 (stage 4)**

The second conv layer is given by

- Filter size: 3×3, stride: 1, padding: 1
- input, output channel size: 256 (stage 3), 512 (stage 4)

Also the rest of blocks ($n_3 - 1$ blocks in stage 3) will have the same convolutional layer structure (3x3 stride 1 padding 1, input/output 256) as above. Similar applies to $n_4 - 1$ blocks in stage 4.

Note: In the homework, we will use $n_3 = n_4 = 2$.

3.1.5 average pooling layer

The average pooling layer uses 4×4 filter with stride 4 and no padding. Since the spatial dimension to this layer is 4-by-4 with channel size 512, or (B, 512, 4, 4), the output after pooling layer would be (B, 512, 1, 1).

3.1.6 fully connected layer

This is the linear layer with input vector size 512, and output vector size 10. This is the final output of the whole network.

3.2 class `IdentityResNet`

This class defines the overall neural network. **In implementing this class, you can define any other auxiliary classes, or declare variables or methods.**

The constructor (`__init__` method) takes 4 parameters:

- `nblk_stage1`: the number of blocks in stage 1
- `nblk_stage2`: the number of blocks in stage 2
- `nblk_stage3`: the number of blocks in stage 3
- `nblk_stage4`: the number of blocks in stage 4

Ideally, your class should able to create the network with varying number of blocks per stage, specified by these parameters. In this homework, all the parameters are fixed to 2. **You can assume that when your homework is graded, these parameters will be also fixed to 2. But I recommend you try to implement neural net supporting variable number of blocks per stage (this is optional).**

Varying the number of layers in a neural net: it may be helpful to use `nn.ModuleList` in `https://pytorch.org/docs/stable/generated/torch.nn.ModuleList.html`.

In particular, the examples of using `append` method is well shown in the first answer of the following question:

`https://discuss.pytorch.org/t/append-for-nn-sequential-or-directly-converting-7104`

Also, some other references are:

`https://discuss.pytorch.org/t/nn-with-configurable-number-of-layers/5202`

`https://discuss.pytorch.org/t/how-to-create-mlp-model-with-arbitrary-number-o-13124`

3.2.1 `forward` method

This method performs the forward pass of a batch of CIFAR-10 images. The output is the same as the output from the last fully connected (linear) layer.

Parameters:

- `x`: *n-D Tensor*.

  The input with type 4-dimensional PyTorch tensor of shape (B, C, H, W) where B stands for batch size, C stands for channel size, H and W are height and width.

Returns:

- `out`: *n-D Tensor*

  The result of forward pass, i.e., 10-dimensional output from the linear layer.

4 Tips

You can use following `nn` modules (search them on the web)

- `nn.Conv2d`
- `nn.BatchNorm2d`
- `nn.Linear`
- `nn.Sequential`

Also you can use the following useful functionals (after `import torch.nn.functional as F`) for layers which do not need learnable parameters.

- `F.relu`
- `F.max_pool2d`
- `F.avg_pool2d`

If needed, use other modules on your own.

For further information about the residual architecture, you can refer to the previously mentioned paper ("Identity Mappings in Deep Residual Networks" link: `https://arxiv.org/pdf/1603.05027.pdf`.), and also the original ResNet paper "Deep Residual Learning for Image Recognition" at `https://openaccess.thecvf.com/content_cvpr_2016/papers/He_Deep_Residual_Learning_CVPR_2016_paper.pdf`.

5 Training and Testing

You need the script for training and testing in `hw6.py`. Specifically, complete Q1–Q9. Determine proper hyperparameters like learning rate. The following tutorial will be very helpful in completing the questions:

`https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html`

Note: Make sure you use GPU in your training. Otherwise training may be very slow. In Colab environment, make sure go to the menu `RunTime->Change runtime type`, and selecting GPU.

6 What to submit

You will be given file `hw6.py`.

1. **You are asked to complete `IdentityResNet` class.**
2. **You are asked to complete Q1–Q9 in `hw6.py`. Fill in the part with '...' for Q1–Q9.**

Submission instructions are

- Submit modified Python file `hw6.py`.
- Upload your files at Blackboard before deadline. (Please submit the file in time, no late submission will be accepted).

7 How your module will be graded

If done correctly, you will be able to achieve accuracy close to 80% **within 5 epochs** of training with proper batch size and learning rate. **The goal is to achieve over 70% of accuracy under the following condition:**

Before testing, your model will be trained until one of the following two events occur, whichever comes earlier.

1. Your model will be trained for **5** epochs.
2. Your model will be trained for **two hours** under the standard Google Colab compute engine environment.

So before submitting your model, please try to train your model on the standard Google Colab environment, and check whether your model can achieve over 70% accuracy under 2 hours of training. **If you properly use GPU runtime environment by CoLab, it should take 4–8 minutes**[1] **per training of 1 epoch.**

8 Grading

- 20 points if your classifier achieves total accuracy more than 70%.
- 6 points if your classifier achieves total accuracy between 30–70%.
- 2 points if your classifier achieves total accuracy below 30%.
- 0 point if you do not submit the file by deadline.

In the blackboard, you can upload your files as many times as you like, before the deadline. The last uploaded file will be used for grading. After deadline, the submission menu will be closed and you will not be able to make submission.

[1] In case you open or run multiple Colab windows, you may get slower performance due to resource allocation policy by Google cloud. So try to use one new Colab window if experiment is expected to take long time.