Programming Assignment 2: Convolutional Neural Networks

Version: 1.2
Changes by Version:

- **(v1.2)** Fixed a few typos.
- **(v1.1)** Updated to new due date Feb. 18th.

Version Release Date: 2022-02-05
Due Date: Friday, Feb. 18th, at 11:59pm
Based on an assignment by Lisa Zhang

Submission: You must submit 2 files through MarkUs[1]: a PDF file containing your writeup, titled `a2-writeup.pdf`, and your code file `a2-code.ipynb`. Your writeup must be typed.

The programming assignments are individual work. See the Course Information handout[2] for detailed policies.

You should attempt all questions for this assignment. Most of them can be answered at least partially even if you were unable to finish earlier questions. If you think your computational results are incorrect, please say so; that may help you get partial credit.

The teaching assistants for this assignment are Stephen Zhao and Yun-Chun Chen. Send your email with subject "*[CSC413] PA2 ...*" to csc413-2022-01-tas@cs.toronto.edu or post on Piazza with the tag `pa2`.

Introduction

This assignment will focus on the applications of convolutional neural networks in various image processing tasks. The starter code is provided as a Python Notebook on Colab (`https://colab.research.google.com/github/uoft-csc413/2022/blob/master/assets/assignments/a2-code.ipynb`). First, we will train a convolutional neural network for a task known as image colourization. Given a greyscale image, we will predict the colour at each pixel. This is a difficult problem for many reasons, one of which being that it is ill-posed: for a single greyscale image, there can be multiple, equally valid colourings. In the second half of the assignment, we will perform fine-tuning on a pre-trained object detection model.

[1]`https://markus.teach.cs.toronto.edu/2022-01/courses/16`
[2]`https://uoft-csc413.github.io/2022/assets/misc/syllabus.pdf`

Image Colourization as Classification

In this section, we will perform image colourization using three convolutional neural networks (Figure 1). Given a grayscale image, we wish to predict the color of each pixel. We have provided a subset of 24 output colours, selected using k-means clustering[3]. The colourization task will be framed as a pixel-wise classification problem, where we will label each pixel with one of the 24 colours. For simplicity, we measure distance in RGB space. This is not ideal but reduces the software dependencies for this assignment.

We will use the CIFAR-10 data set, which consists of images of size 32x32 pixels. For most of the questions we will use a subset of the dataset. The data loading script is included with the notebooks, and should download automatically the first time it is loaded.

Helper code for Part A is provided in `a2-code.ipynb`, which will define the main training loop as well as utilities for data manipulation. Run the helper code to setup for this question and answer the following questions.

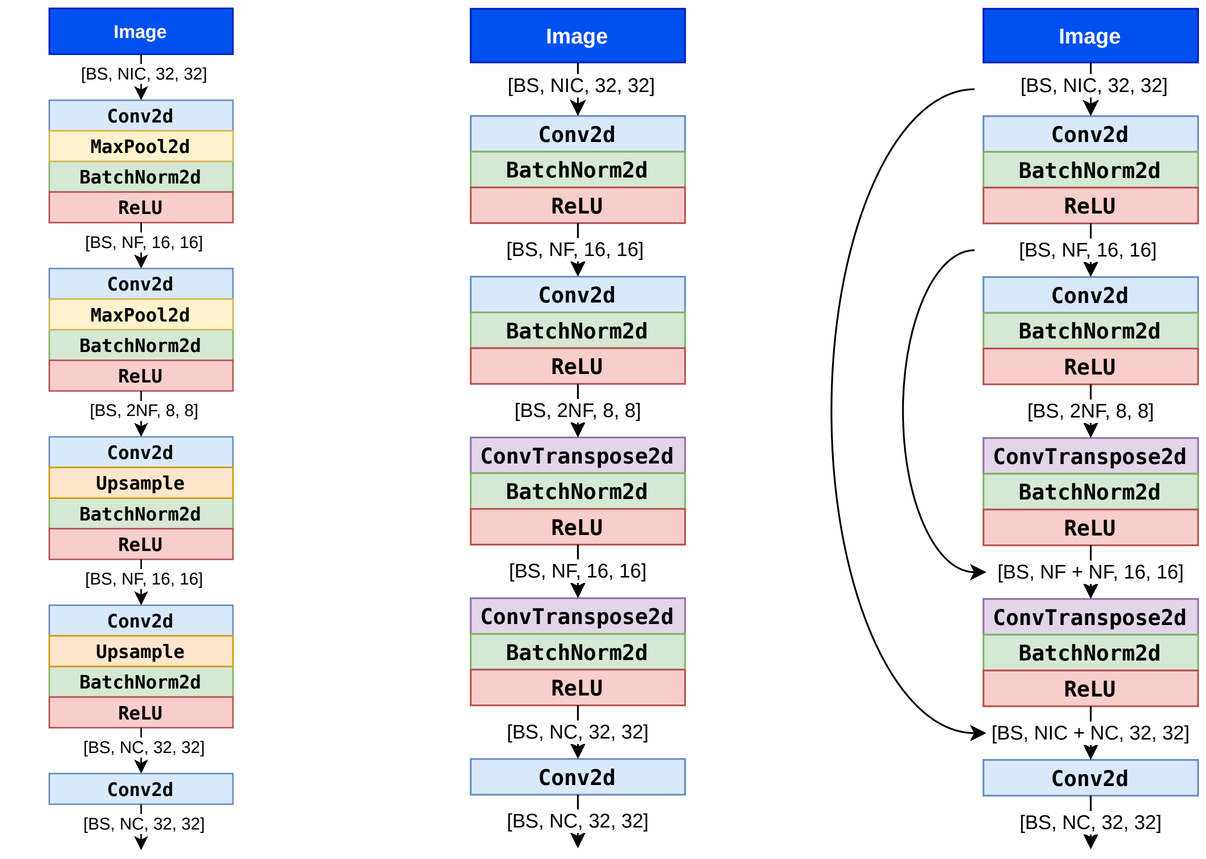
Part A: Pooling and Upsampling (2 pts)

1. Complete the model `PoolUpsampleNet`, following the diagram in Figure 1a. Use the PyTorch layers `nn.Conv2d`, `nn.ReLU`, `nn.BatchNorm2d`, `nn.Upsample`, and `nn.MaxPool2d`. Your CNN should be configurable by parameters `kernel`, `num_in_channels`, `num_filters`, and `num_colours`. In the diagram, `num_in_channels`, `num_filters` and `num_colours` are denoted **NIC**, **NF** and **NC** respectively. Use the following parameterizations (if not specified, assume default parameters):

   - `nn.Conv2d`: The number of input filters should match the second dimension of the *input* tensor (e.g. the first `nn.Conv2d` layer has **NIC** input filters). The number of output filters should match the second dimension of the *output* tensor (e.g. the first `nn.Conv2d` layer has **NF** output filters). Set kernel size to parameter `kernel`. Set padding to the `padding` variable included in the starter code.
   - `nn.BatchNorm2d`: The number of features should match the second dimension of the output tensor (e.g. the first `nn.BatchNorm2d` layer has **NF** features).
   - `nn.Upsample`: Use `scaling_factor = 2`.
   - `nn.MaxPool2d`: Use `kernel_size = 2`.

   Note: grouping layers according to the diagram (those not separated by white space) using the `nn.Sequential` containers will aid implementation of the `forward` method.

[3]`https://en.wikipedia.org/wiki/K-means_clustering`



(a) `PoolUpsampleNet` (b) `ConvTransposeNet` (c) `UNet`

Figure 1: Three network architectures that we will be using for image colourization. Numbers inside square brackets denote the shape of the tensor produced by each layer: **BS**: batch size, **NIC**: `num_in_channels`, **NF**: `num_filters`, **NC**: `num_colours`.

2. Run main training loop of `PoolUpsampleNet`. This will train the CNN for a few epochs using the cross-entropy objective. It will generate some images showing the trained result at the end. Do these results look good to you? Why or why not?

   Answer

   Fill in

3. Compute the number of weights, outputs, and connections in the model, as a function of **NIC**, **NF** and **NC**. Compute these values when each input dimension (width/height) is doubled. Report all 6 values.

   Answer

   Fill in

Part B: Strided and Transposed Dilated Convolutions (3 pts)

For this part, instead of using `nn.MaxPool2d` layers to reduce the dimensionality of the tensors, we will increase the step size of the preceding `nn.Conv2d` layers, and instead of using `nn.Upsample` layers to increase the dimensionality of the tensors, we will use *transposed* convolutions. Transposed convolutions aim to apply the same operations as convolutions but in the opposite direction. For example, while increasing the stride from 1 to 2 in a convolution forces the filters to skip over every other position as they slide across the input tensor, increasing the stride from 1 to 2 in a transposed convolution adds "empty" space around each element of the input tensor, as if reversing the skipping over every other position done by the convolution. We will be using a `dilation rate of 1` for the transposed convolution. Excellent visualizations of convolutions and transposed convolutions have been developed by Dumoulin and Visin [2018] and can be found on their GitHub page[4].

1. Complete the model `ConvTransposeNet`, following the diagram in Figure 1b. Use the PyTorch layers `nn.Conv2d`, `nn.ReLU`, `nn.BatchNorm2d` and `nn.ConvTranspose2d`. As before, your CNN should be configurable by parameters `kernel`, `dilation`, `num_in_channels`, `num_filters`, and `num_colours`. Use the following parameterizations (if not specified, assume default parameters):

   - `nn.Conv2d`: The number of input and output filters, and the kernel size, should be set in the same way as Part A. For the first two `nn.Conv2d` layers, set `stride` to 2 and set `padding` to 1.
   - `nn.BatchNorm2d`: The number of features should be specified in the same way as for Part A.

[4] `https://github.com/vdumoulin/conv_arithmetic`

- `nn.ConvTranspose2d`: The number of input filters should match the second dimension of the *input* tensor. The number of output filters should match the second dimension of the *output* tensor. Set `kernel_size` to parameter `kernel`. Set `stride` to 2, set `dilation` to 1, and set both `padding` and `output_padding` to 1.

2. Train the model for at least 25 epochs using a batch size of 100 and a kernel size of 3. Plot the training curve, and include this plot in your write-up.

3. How do the result compare to Part A? Does the `ConvTransposeNet` model result in lower validation loss than the `PoolUpsampleNet`? Why may this be the case?

   The validation accuracy should be higher and validation loss should be lower. Some possible reasons include:

   -

4. How would the `padding` parameter passed to the first two `nn.Conv2d` layers, and the `padding` and `output_padding` parameters passed to the `nn.ConvTranspose2d` layers, need to be modified if we were to use a kernel size of 4 or 5 (assuming we want to maintain the shapes of all tensors shown in Figure 1b)?

   Note: PyTorch documentation for `nn.Conv2d`[5] and `nn.ConvTranspose2d`[6] includes equations that can be used to calculate the shape of the output tensors given the parameters.

   - `kernel_size = 4, padding = 1, output_padding = 0`
   - `kernel_size = 5, padding = 2, output_padding = 1`
   - `kernel_size = x, padding = (x - 1) // 2, output_padding = int(x % 2 == 1)`

5. Re-train a few more `ConvTransposeNet` models using different batch sizes (e.g., 32, 64, 128, 256, 512) with a fixed number of epochs. Describe the effect of batch sizes on the training/validation loss, and the final image output quality. You do *not* need to attach the final output images.

   As batch size increases, validation loss increases (given the same number of epochs).

Part C: Skip Connections (1 pts)

A skip connection in a neural network is a connection which skips one or more layer and connects to a later layer. We will introduce skip connections to the model we implemented in Part B.

[5]`https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html`
[6]`https://pytorch.org/docs/stable/generated/torch.nn.ConvTranspose2d.html`

1. Add a skip connection from the first layer to the last, second layer to the second last, etc. That is, the final convolution should have both the output of the previous layer and the initial greyscale input as input (see Figure 1c). This type of skip-connection is introduced by Ronneberger et al. [2015], and is called a "UNet". Following the `ConvTransposeNet` class that you have completed, complete the `__init__` and `forward` methods of the `UNet` class in Part C of the notebook.

   Hint: You will need to use the function `torch.cat`.

2. Train the model for at least 25 epochs using a batch size of 100 and a kernel size of 3. Plot the training curve, and include this plot in your write-up.

3. How does the result compare to the previous model? Did skip connections improve the validation loss and accuracy? Did the skip connections improve the output qualitatively? How? Give at least two reasons why skip connections might improve the performance of our CNN models.

   - Skip connections improve the vanishing gradient problem.
   - Skip connections allow some of the information that was lost in the pooling or strided convolution layers to be reintroduced to the final output layers.
   - The models with skip connections have more parameters due to increased dimensionality of the transposed convolution layers.

Object Detection

In the previous two parts, we worked on training models for image colourization. Now we will switch gears and perform object detection by fine-tuning a pre-trained model.

Object detection is a task of detecting instances of semantic objects of a certain class in images. Fine-tuning is often used when you only have limited labeled data.

For the following, you are not expected to read the referenced papers, though the writing is very entertaining (by academic paper standards) and it may help provide additional context.

We use the YOLO (You Only Look Once) approach, as laid out in the the original paper by Redmon et al. [2016]. YOLO uses a single neural network to predict bounding boxes (4 coordinates describing the corners of the box bounding a particular object) and class probabilities (what object is in the bounding box) based on a single pass over an image. It first divides the image into a grid, and for each grid cell predicts bounding boxes, confidence for those boxes, and conditional class probabilities.

Here, we take a pre-trained model, YOLOv3 [Redmon and Farhadi, 2018] and fine-tune it to perform detection on the COCO dataset Lin et al. [2014].

Part D.1: Fine-tuning from pre-trained models for object detection (2 pts)

1. A common practice in computer vision tasks is to take a pre-trained model trained on a large datset and finetune only parts of the model for a specific usecase. This can be helpful, for example, for preventing overfitting if the dataset we fine-tune on is small.

   To keep track of which weights we want to update, we use the PyTorch utility `Model.named_parameters()`[7], which returns an iterator over all the weight matrices of the model.

   Complete the model parameter freezing part in `train.py` by adding 3-4 lines of code where indicated. See also the notebook for further instruction.

Part D.2: Implement the classification loss (2 pts)

1. See the notebook for instructions for this part. You will fill in the BCEcls loss used for the classification loss.

2. Visualize the predictions on 2 sample images by running the helper code provided.

What you have to submit

For reference, here is everything you need to hand in. See the top of this handout for submission directions.

- A PDF file titled `a2-writeup.pdf` containing only the following:
  - Answers to questions from Part A
    - Q1 code for model `PoolUpsampleNet` (screenshot or text)
    - Q2 visualizations and your commentary
    - Q3 answer (6 values as function of **NIC**, **NF**, **NC**)
  - Answers to questions from Part B
    - Q1 code for model `ConvTransposeNet` (screenshot or text)
    - Q2 answer: 1 plot figure (training/validation curves)
    - Q3 answer
    - Q4 answer
    - Q5 answer
  - Answers to questions from Part C

[7]See examples at `https://pytorch.org/docs/stable/nn.html`

        - Q1 code for model `UNet` (screenshot or text)
        - Q2 answer: 1 plot figure (training/validation curves)
        - Q3 answer
    - Answers to questions from Part D.1
        - Q1 code for freezing layers in `train.py` (screenshot or text; you may include only the lines you added)
    - Answers to questions from Part D.2
        - Q1 code for classification loss in `utils/loss.py` (screenshot or text; you may include only the lines you added)
        - Q2 answer: visualization of predictions (screenshot or image files)

- Your code file `a2-code.ipynb`

References

Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning, 2018.

Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.

Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.

Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer, 2014.