Assignment 3: *Convolutional Neural Networks*

Date: *Mar 4, 2025* Due : *Mar 18, 2024*

Preface

This is the third series of assignments for the course *ECE1508: Applied Deep Learning*. The exercises are aimed to review convolutional neural networks studied in Chapter 4. Below, you can find the information about contents of these exercises, as well as instructions on how to submit them.

General Information The assignments are given in two sections. In the first section, you have written questions that you can answer in words or by derivation. The questions are consistent with the material of Chapter 4 and you do not need any further resources to answer them. The second section includes the programming assignments. For these assignments, you are assumed to have some very basic knowledge of programming in Python. For those who are beginners, a quick introduction to the install and use of Python libraries has been given in a separate file. In the case that a question is unclear or there is any flaws, please contact over Piazza. Also, in case that any particular assumption is required to solve the problem, feel free to consider the assumption and state it in your solution. The total mark of the assignments is **100 points** plus **5 extra points** with total mark of written questions adds to **35 points** and the total mark of the programming assignments adds to **65 points** plus **5 points bonus** for optional items.

How to Submit A notebook has been provided with basic code that you can complete. The submission is carried out through the Crowdmark. Please submit the answer of each question **separately**, following the **steps below**. Please note that *failure to meet the formatting can lead to mark deduction*.

1. For Written Questions, you can submit handwritten or typed answers as a `.pdf`, `.jpg` or `.png` file.
2. For Programming Questions, the answer should **complete the Python Notebook** shared with this assignment. For *each question*, please print the corresponding part of the notebook as a `.pdf` file and upload it in the corresponding field on Crowdmark. Your uploaded `.pdf` should contain **all figures, diagrams and codes requested** in the question.
3. The completed Notebook, i.e., the `.ipynb` file, including all the codes and the outputs should also be submitted as the attachment to the last item on Crowdmark.

When submitting your notebook, please pay attention to the following points:

1. The file should be named `Lastname_Firstname_Assgn3.ipynb`
2. Please make sure to name the files with the name that is displayed on the Quercus account

The deadline for your submission is on **March 18, 2025 at 11:59 PM**.

- You can delay up to two days. After this extended deadline no submission is accepted.
- For each day of delay, 5% of the mark after grading is deducted.

Please submit your assignment **only through Crowdmark, and not by email.**

1 Written Exercises

Question 1 [15 Points] **(One-dimensional Convolution)** For one-dimensional arrays, we can perform one-dimensional convolution: let $\mathbf{x} \in \mathbb{R}^N$ be a one-dimensional array, i.e., a vector of length N, and let $\mathbf{w} \in \mathbb{R}^F$ be a kernel of length $F < N$. The convolution of $\mathbf{x}$ with $\mathbf{w}$ with stride $S = 1$ is the array $\mathbf{z} \in \mathbb{R}^{N-F+1}$ whose entry i is computed as

$$z_i = \mathbf{w}^\mathsf{T} \mathbf{x}[i : F + i - 1], \tag{1.1}$$

where $\mathbf{x}[i : F + i - 1] = [x_i, x_{i+1}, \dots, x_{F+i-1}]$ with x_i denoting entry i of $\mathbf{x}$.

We intend to review what we learned in the lecture for this simple form of convolution. To this end, answer the following items

1. Extend the definition to the case with a general integer stride S.
2. Extend the definition to the case with stride $S = 1/m$ with m being an integer.
3. Explain how we could keep the dimension of the output array the same as the input.
4. Assume that we know $\nabla_{\mathbf{z}} \hat{R}$. Compute $\nabla_{\mathbf{x}} \hat{R}$ in terms of $\nabla_{\mathbf{z}} \hat{R}$ and the kernel $\mathbf{w}$.
5. Assume that we know $\nabla_{\mathbf{z}} \hat{R}$ and $\mathbf{x}$. Compute $\nabla_{\mathbf{w}} \hat{R}$ in terms of $\nabla_{\mathbf{z}} \hat{R}$ and the input $\mathbf{x}$.
6. Compare your result in Parts 4 and 5 with what we learned in Chapter 4 for multi-channel convolution. Are they consistent?

Question 2 [10 Points] **(Convolution as Layer of Neurons)** In Question 1, let $N = 8$ and $F = 3$ and assume that the convolution is performed with stride $S = 1$ and no zero padding. We further activate the output of this convolution layer via the ReLU function, i.e., we compute $\mathbf{y} = \mathrm{ReLU}(\mathbf{z})$. As mentioned in Chapter 4, we can look at the relation between the input $\mathbf{x}$ and the activated output $\mathbf{y}$ as a layer of neurons with *shared weights* and *local connectivity*.

Answer the following items.

1. Sketch the diagram of a layer of neurons that connects input array $\mathbf{x}$ to the activated outputs $\mathbf{y}$.
2. Explain how the neurons in this layer are *locally connected* to the inputs.
3. Assume that you replace this layer with a *fully-connected* layer. How many learnable parameters does this fully-connected layer have?
4. Compare the number of learnable parameters in Part 3 to that of the convolutional layer. What do you conclude from this comparison about a practical CNN architecture like VGG-16?

Question 3 [10 Points] **(Reducing Jitter via Max-Pooling)** Similar to convolution, we can apply max-pooling on one-dimensional arrays by moving only towards right. Max-pooling of one-dimensional array $\mathbf{x} = \mathbb{R}^N$ with a filter of length $F \leq N$ returns an array $\mathbf{y} \in \mathbb{R}^{N-F+1}$ whose entry i is

$$y_i = \max \{x_i, x_{i+1}, \dots, x_{i+F-1}\} \tag{1.2}$$

with x_i being entry i of $\mathbf{x}$.

Consider the following one-dimensional array

$$\mathbf{x} = [0.4,\ 0.1,\ 0.5,\ 0.9,\ 0.01,\ 0.3,\ 0.8]. \tag{1.3}$$

We intend to inspect the max-pooling on this array to better understand the properties of pooling layer in a CNN. To this end, answer the following items:

1. Plot $\mathbf{x}$ against its entry indices, i.e., sketch the points (i, x_i) in a two-dimensional plane for $i = 1, \dots, 7$ and connect each two neighboring points with a line.
2. Apply max-pooling with filter length $F = 2$ on array $\mathbf{x}$. To keep the dimension the same, apply zero padding, i.e., add appropriate number of zeros at the end of $\mathbf{x}$. Call the max-pooled array $\boldsymbol{y}$.
3. Plot $\mathbf{y}$ against its entry indices and compare it against Part 1.
4. Repeat Parts 2 and 3 for filter lengths $F = 3, 7$.
5. Explain briefly your observation.

2 Programming Questions

In this assignment, we are going to implement two target architectures: a *plain convolutional neural network (CNN)* and a *CNN with skip connection*. The architecture of each CNN is described below. Both of these NNs perform binary classifications on 32×32 RGB images. We then implement and train them on a subset of CIFAR-10 dataset.

2.1 Plain CNN

The architecture of the plain CNN is shown below.

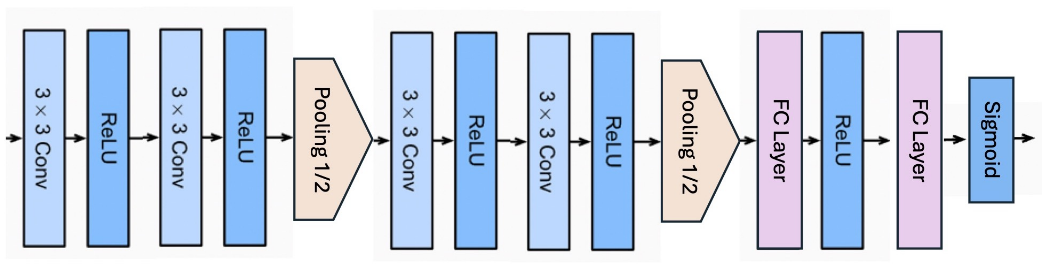

Figure 2.1: Plain CNN architecture we are to implement.

This architecture has 6 learnable layers and 2 pooling layers. The properties of each layer in the order of its depth (from left to right in Figure 2.1) are as follows:

1. *Convolutional Layer 1* has 3 input channels each of size 32×32 and computes 32 output channels using 3×3 filters with no padding and stride 1. It is activated by the ReLU function.
2. *Convolutional Layer 2* has 32 input channels and computes 32 output channels using 3×3 filters with no padding and stride 1. It is activated by the ReLU function.
3. *Pooling Layer 3* applies max-pooling using 2×2 filters with no padding and stride 2.
4. *Convolutional Layer 4* has 32 input channels and computes 64 output channels using 3×3 filters with no padding and stride 1. It is activated by the ReLU function.
5. *Convolutional Layer 5* has 64 input channels and computes 64 output channels using 3×3 filters with no padding and stride 1. It is activated by the ReLU function.
6. *Pooling Layer 6* applies max-pooling using 2×2 filters with no padding and stride 2.
7. *Fully-Connected Layer 7* computes 128 outputs all activated by ReLU.
8. *Fully-Connected Layer 8* computes a single output activated by sigmoid.

2.2 CNN with Skip Connection

Skip connection is studied in details in Chapter 5. Nevertheless, we can readily start implementing it even before we are over with Chapter 5. The architecture of the CNN with skip connection is shown below. This architecture has again 6 learnable layers and 2 pooling layers. The learnable layers are

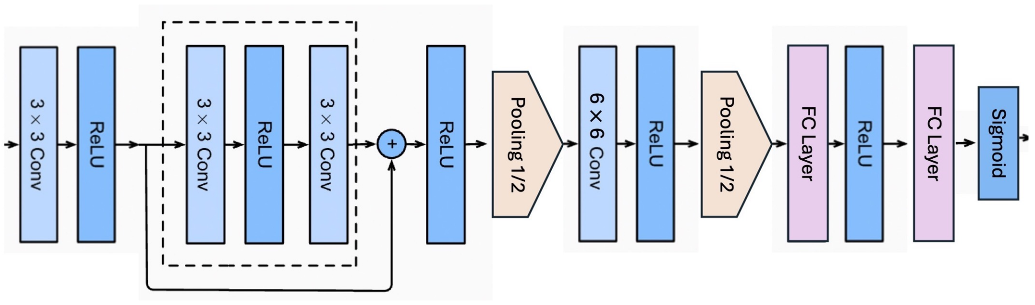

Figure 2.2: CNN with skip connection we are to implement.

arranged in the following form: one single convolutional layer, one residual unit with two convolutional layers and a *skip connection*, one single convolutional layer and 2 fully-connected layers. Properties of each block in the order of its depth (from left to right in Figure 2.2) are as follows:

1. *Convolutional Layer 1* has 3 input channels each of size 32×32 and computes 32 output channels using 3×3 filters with no padding and stride 1. It is activated by the ReLU function.

2. *Residual Unit* has two convolutional layers. We refer to them as *Convolutional Layer 2* and *Convolutional Layer 3*. Each of these layers has 32 input channels and computes 32 output channels. The layers use 3×3 filters with padding and stride 1 to *make sure that the input and output of the unit, i.e., input to Convolutional Layer 2 and output of Convolutional Layer 3, have the same dimensions*. They use ReLU activation. There is a simple skip connection that adds the input of the unit, i.e., the activated output of *Convolutional Layer 1*, to the output of the unit, i.e., output of *Convolutional Layer 3*, before activation.

3. *Pooling Layer 4* applies max-pooling using 2×2 filters with no padding and stride 2.

4. *Convolutional Layer 5* has 32 input channels and computes 64 output channels using 6×6 filters with no padding and stride 1. It is activated by the ReLU function.

5. *Pooling Layer 6* applies max-pooling using 2×2 filters with no padding and stride 2.

6. *Fully-Connected Layer 7* computes 128 outputs all activated by ReLU.

7. *Fully-Connected Layer 8* computes a single output activated by sigmoid.

Throughout programming tasks we mainly use `torch`. As we have seen in Assignment 2 and the Tutorials, module `torch.nn` gives us access to the components we need to build up our model. We further use `torchvision` functions and modules to load and process our dataset. The functionality of each item will be shortly clear.

Question 1 [15 Points] **(Making a Dataset)** We first build our training and test datasets which is a subset of CIFAR-10 dataset. To this end, we first load CIFAR-10. Read the following text to learn how to do it properly.

Import CIFAR-10 In practice, we apply some transforms on the dataset before importing it. This is done by functions in `torchvision.transforms`. You can check the reference code to learn the details. These transforms convert data samples into tensors that are understandable for PyTorch and normalize the values in these tensor. How do we know the normalization factors? Well, this is usually found out by experiment and are widely available online. You need to know your *dataset* and *model* to find proper values, most probably online.

1. Complete the reference code to import both training and test datasets from CIFAR-10.
2. Verify the size of `train_set` and `test_set` by printing their length. Use `len()` to print their length.
3. Print the first data-point by calling `train_set[0]`. This is a tuple with first entry being your pixel tensor and the second entry being the label. Print the label value.
4. Print all the classes by calling the attribute `.class_to_idx` of the loaded datasets.
5. Show the image of the first data-point and compare it to its label. To show the image, you may use the `imshow()` function in the reference file.

It would have been great to directly work with CIFAR-10 as the dataset. However, this may need too much computing power. To keep things simple for an assignment, we are going to extract a smaller dataset out of CIFAR-10. Namely, we collect the images of *cats* and *dogs* from it and train our NNs with this reduced dataset to perform binary classification. Read the following text to learn how to do it.

Making Smaller Dataset from CIFAR-10 To collect only images of *cats* and *dogs*, we use function `Subset()` from module `torch.utils.data`. This function gets a dataset and a list of indices and returns a subset that has the data-points whose indices are in the index list. For instance,

`Subset([data0, data1, data2, data3], [0,2]) = [data0, data2]`

Answer the following items.

6. Write the function `class_extract()` that gets a dataset and a list of classes and returns a subset of dataset that includes only the data-points of that class. You can do it by completing the provided reference code.
7. Use the attribute `.class_to_idx` to find out the numbers corresponding to classes "cat" and "dog". Save them in a list called `cls_list`.
8. Use the function `class_extract()` and make two subsets `train_subset` and `test_subset` as

   ```
   train_subset = class_extract(cls_list, train_set)
   test_subset = class_extract(cls_list, test_set)
   ```

   These subsets include only images of *cat* and *dog*. Check what the labels of *cat* and *dog* in these subsets are.
9. Print the sizes of the training and tests subsets.

The final part is to load the subsets as mini-batches. Read the following text to learn how to do this.

Load Training and Test Datasets as Mini-Batches To load datasets as mini-batches, we use the function `DataLoader` from module `torch.utils.data`. This function takes a dataset and batch size as inputs and returns an `iterator` that is a sequence of mini-batches.

10. Use `DataLoader` to load the training and test datasets as mini-batches with batch size 100.
    Hint: Do not forget to set option `shuffle = True` when you use `DataLoader`.

11*. (Optional) Confirm that the numbers of mini-batches in iterators `train_loader` and `test_loader` match what you expect.

Question 2 [35 Points] **(Implementing Plain CNN)** We now intend to implement and train the plain CNN. In case you are new with this in PyTorch, read the following text.

Convolutional and Pooling Layers in PyTorch A convolutional layer can be instantiated via the class `Conv2d()` from `torch.nn` module. Note that `Conv2d()` corresponds to multi-channel convolution we had in Chapter 4. This class gets the number of input and output channels as well as padding, stride and filter size. It then instantiates a convolutional layer with the corresponding parameters. For instance

```
nn.Conv2d(in_channels=3, out_channels=16, kernel_size=3)
```

returns one convolutional layer whose input has 3 channels and whose output has 16 channels. The filters in this layer are all 3×3. The padding and stride are also set to the default values 0 and 1, respectively. Note that *the height and width of the input and output maps are not specified in this layer.* So, this layer could get any 3-channel input and will return a 16-channel output with height and width that is computed from the input size, kernel size, stride and padding.

The pooling layer is further instantiated via its corresponding class: for instance, a max-pooling layer with 2×2 filter and stride 2 is realized using `MaxPool2d()` from `torch.nn` as follows

```
nn.MaxPool2d(kernel_size = 2, stride = 2)
```

Answer the following items.

1. Write the class `myCNN()` which implements the plain CNN in a simple form. You may complete the reference code.

2. Complete the forward pass of the neural network by adding the method `.forward()` to this class.

We next write the test function which gets an instant of the model and tests it over the test dataset. This function is very helpful when we complete the training loop, as it lets us test our trained model every couple of epochs. To write this function, we introduce two new options in PyTorch; namely, `device` and `torch.no_grad()`.

No-Grad and Device In the train loop, if we condition our code to `with torch.no_grad():`, we stop PyTorch from computing auto-gradient. This is important we test the code in the middle of training, as we do not need gradient when we are simply testing. The argument `device` further enables us to ask PyTorch performing the computation on a specific processor on our computer. If we have a GPU, we should use this option to speed up the computation. We can check the availability of GPU on our machine using `torch.backends` by looking into `cuda` or `mps` modules in it.

3. Complete the function `test` that takes a model, device and a loss function as input and returns the test risk and test accuracy of the model over the test set `test_subset`.

4. Use your implementation to test the *untrained* CNN via the binary cross-entropy function. Print the accuracy and loss value and explain your observation.

We now have all we need to implement the training loop. For training, we use

→ binary cross entropy as the loss function

→ Adam optimizer with learning rate $\eta = 0.001$.

5. Complete the training loop, which gets the model, device and number of epochs as inputs and returns the lists of training and test losses as well as test accuracy achieved at each epoch.

6. Instantiate `model = myCNN()` and run the training loop to train this model for 20 epochs.

7. Explain your observations.

We next include dropout and batch normalization to our implementation. Read the following text to learn how to do this.

Dropout and Batch Normalization in CNNs We know how to add dropout and batch normalization to fully-connected layers. For convolutional layers, we could use `Dropout2d()` and `BatchNorm2d()` from `torch.nn`, where in the input argument to the former class is the *dropout probability* and to the latter class is the number of channels. We can define them as separate objects in each layer and then use them in the forward pass. For instance, if `h` is one mini-batch at the output of a convolutional layer with 16 channels, we can apply dropout with probability 0.4 and apply batch normalization on `h` as follows:

```
dropout = nn.Dropout2d(p=0.4)
BN = nn.BatchNorm2d(16)
h_drop = dropout(h)
h_BN = BN(h_drop)
```

Answer the following items.

8. Revise your implementation `myCNN()` by adding batch normalization and dropout with probability 0.4 to *Convolutional layers 1 and 4* and the *Fully-connected Layer 7*. Add the dropout *before* the linear operation, and the batch normalization *after* activation.

9. Repeat the training for 20 epochs and observe the outcome.

Question 3 [15 + 5 (Optional) Points] **(Implementing CNN with Skip Connection)** In this question, we implement the CNN with skip connection and train it. Note that we could use all we have implemented in Programming Question 2. We only need to change our model.

1. Write the class `myResNet()` which implements the CNN with skip connection. You may complete the reference code.
   Hint: *Pay attention to the padding on Convolutional Layers 2 and 3.*

2. Complete the forward pass by adding the method `.forward()` to this class.
   Hint: *Pay attention to the skip connection.*

3. Instantiate `model = myResNet()` and run the training loop to train this model for 20 epochs. Explain your observations.

4. Revise the class `myResNet()` by adding batch normalization to all weighted layers, i.e., all convolutional and fully-connected layers, and dropout with probability 0.4 to *Convolutional layers 1 and 5* and the *Fully-connected Layer 7*. Add the dropout *before* the linear operation, and the batch normalization *after* activation.

5. Repeat the training for 20 epochs and observe the outcome.

6*. (Optional) Remove the dropout in the revised CNN, and repeat the training. What do you observe as compared to Part 5? Explain your observation.