Homework 2

An Introduction to Convolutional Neural Networks

11-785: Introduction to Deep Learning (Spring 2019)

OUT: **February 17, 2019**
DUE: **March 10, 2019, 11:59 PM**

Start Here

- **Collaboration policy:**
  - You are expected to comply with the University Policy on Academic Integrity and Plagiarism.
  - You are allowed to talk with / work with other students on homework assignments
  - You can share ideas but not code, you must submit your own code. All submitted code will be compared against all code submitted this semester and in previous semesters using MOSS.

- **Overview:**
  - **Part 1**: All of the problems in Part 1 will be graded on Autolab. You can download the starter code from Autolab as well. This assignment has 100 points total.
  - **Part 2**: This section of the homework is an open ended competition hosted on Kaggle.com, a popular service for hosting predictive modeling and data analytic competitions. All of the details for completing Part 2 can be found on the competition page.

- **Submission:**
  - **Part 1**: The compressed `handout` folder hosted on Autolab contains four python files, `layers.py`, `cnn.py`, `mlp.py` and `local_grader.py`, and some folders `data`, `autograde` and `weights`. File `layers.py` contains common layers in convolutional neural networks. File `cnn.py` contains outline class of two convolutional neural networks. File `mlp.py` contains a basic MLP network. File `local_grader.py` is foryou to evaluate your own code before submitting online. Folders contain some data to be used for section 1.3 and 1.4 of this homework.

    Make sure that all class and function definitions originally specified (as well as class attributes and methods) in the two files `layers.py` and `cnn.py` are fully implemented to the specification provided in this write-up.

    Your submission must be titled `handin.tar` (gzip format) and it is minimally required to contain a directory called `hw2`, which contains the `layers.py` and `cnn.py` files implementing the classes, functions, and methods specified in the write-up. Please do not import any other external libraries other than NumPy and the default python packages, as extra packages that do not exist in the autograder image will cause a submission failure.
  - **Part 2**: See the the competition page for details.

- **Local autograder:** To ease your life, we provide in the handout for part 1 a local autograder that you can use on your own code. It roughly has the same behavior as the one on Autolab, but the printouts are different. In theory, passing one means that you pass the other, but we recommend you also submit partial solutions on Autolab from time to time while completing the homework.

  To run the local autograder : run `local_grader.py`.

1 NumPy Based Convolutional Neural Networks

In this section, you need to implement convolutional neural networks using the NumPy library only. Python 3, NumPy>=1.16 and PyTorch>=1.0.0 are suggested environment. First implement forward and backward for linear layer, convolutional layer and flatten layer (Activation layers are already finished) in `layers.py` file. Your implementations will be compared with PyTorch.

Note : Contrary to HW1P1, derivatives `dW` and `db` are **not** averaged on the batch in the network classes. Consider that the update rule in an optimizer is `W -= lr * np.mean(dW,axis=0)`

Note : Also contrary to HW1P1, you will mostly code basic blocks (one CNN layer) and then use that in building bigger nets for which the overall structure is given.This should alleviate the "engineering" part of the homework so that you can focus on understanding what a CNN is.

1.1 Convolutional layer [60 points]

Implement the `Conv1D` class in `layers.py`. The class `Conv1D` has four arguments: `in_channel`, `out_channel`, `kernel_size` and `stride`. They are all positive integers. The detail meaning of the arguments has been discussed in class and you can also find them in PyTorch document: `https://pytorch.org/docs/stable/nn.html#torch.nn.Conv1d`. We do not consider other arguments such as `padding`.

Implement the `Conv1D` class so that it has similar usage and functionality to `torch.nn.Conv1d`. There are three parts to be implemented.

Note : changing the shape/name of the provided attributes is **not allowed**. Like in HW1P1 we will check the value of these attributes.

1.1.1 Forward [20 points]

Finish the returned value for the `forward` part.

The input `x` is the input of the convolutional layer and the shape of `x` is (`batch_size`, `in_channel`, `in_width`). The return value is the output of the convolutional layer and the shape is (`batch_size`, `out_channel`, `out_width`). Note that `in_width` is an arbitrary possitive integer and `out_width` is determined by `in_width`, `kernel_size` and `stride`.

1.1.2 Backward [40 points]

You will here write the code of the .ttbackward method. Its argument is the backpropagated `delta`, i.e. the derivative of the loss w.r.t the output of the current layer. From HW1P1 you should be familiar with this.

dW and db: Finish `self.dW` and `self.db` for the `backward` part.

The input `delta` is the derivative of the loss with respect to the convolutional layer output. It has the same shape as the convolutional layer output. `self.dW` and `self.db` represent the unaveraged gradients of the loss w.r.t `self.W` and `self.b`. Their shapes are the same as the weight `self.W` and the bias `self.b`. We have already initialized them for you.

dx: Finish the returned value for the `backward` part. It is the derivative of the loss with respect to the input of the convolutional layer and has the same shape as the input.

Test: To test your implementation, we will compare the results of the three parts with Pytorch function `torch.nn.Conv1d`. Details can be found in function `test_cnn_correctness_once` in the `local_grader.py` file. Here are some brief codes to show the testing.

```
import torch
import torch.nn as nn
import numpy as np
from torch.autograd import Variable
from layers import Conv1D

## initialize your layer and PyTorch layer
net1 = Conv1D(8, 12, 3, 2)
net2 = torch.nn.Conv1d(8, 12, 3, 2)

## initialize the inputs
x1 = np.random.rand(3, 8, 20)
x2 = Variable(torch.tensor(x1),requires_grad=True)

## Copy the parameters from the Conv1D class to PyTorch layer
net2.weight = nn.Parameter(torch.tensor(net1.W))
net2.bias = nn.Parameter(torch.tensor(net1.b))

## Your forward and backward
y1 = net1(x1)
b, c, w = y1.shape
delta = np.random.randn(b,c,w)
dx = net1.backward(delta)

## PyTorch forward and backward
y2 = net2(x2)
delta = torch.tensor(delta)
y2.backward(delta)

## Compare
def compare(x,y):
    y = y.detach().numpy()
    print(abs(x).max())
    return

compare(y1, y2)
compare(dx, x2.grad)
compare(net1.dW, net2.weight.grad)
compare(net1.db, net2.bias.grad)
```

As you can see in the codes, we check the maximum absolute error of the tensor elements. We ran our codes for 2,000 times and the maximum absolute error (MAE) is 5e-14. If the values your implementation returns can have an MAE smaller than 1e-12, then we think your implementation is right.

1.2 Flatten layer

After we finish linear layer and convolutional layer, we need one more layer to transform the output of convolutional layers to the input of linear layers since we need to use linear layers for many tasks such as classification. Finish the forward and backward part of the `Flatten` class.

In the forward part, the input `x` is the input of the flatten layer and its shape is (`batch_size`, `in_channel`, `in_width`). Just reshape it to get the returned value with a shape of (`batch_size`, `in_channel`*`in_width`).

In the backward part, the input `delta` is the derivative of the loss with respect to the flatten layer output. You need to return the derivative of the loss with respect to the flatten layer input. For this part, we do not provide an autograder. But you can check the correctness after you finish the convolution neural network in

Section 1.3.

1.3 CNN as a Simple Scanning MLP [20 points]

In this part you must compose a CNN that will perform the same computation as scanning a given input with a given multi-layer perceptron.

You are given a 128×24 input (128 time steps, with a 24-dimensional vector at each time). You are required to compute the result of scanning it with the given MLP. The MLP evaluates 8 contiguous input vectors at a time (so it is effectively scanning for 8-time-instant wide patterns). The MLP "strides" forward 4 time instants after each evaluation, during its scan. It only scans until the end of the input (so it does not pad the end of the input with zeros).

The MLP itself has three layers, with 8 neurons in the first layer (closest to the input), 16 in the second and 4 in the third. Each neuron uses a `ReLU` activation, except after the final output neurons. All bias values are 0.

Since the network has 4 neurons in the final layer and scans with a stride of 4, it produces one 4-component output every 4 time instants. Since there are 128 time instants in the inputs and no zero-padding is done, the network produces 31 outputs in all, one every 4 time instants. When flattened, this output will have 124 (4×31) values.

For this problem you are required to implement the above scan, but you must do so using a Convolutional neural network. You must use the implementation of your Convolutional layers in 1.1 to compose a Convolution Neural Network which will behave identically to scanning the input with the given MLP as explained above. You will be evaluated on the correctness of the output 124 values.

The Multi-layer Perceptron is composed of three layers and the architecture of the model is given in `mlp.py` included in the handout. Weights of the MLP are given as the `mlp_weights_part_b.npy` file.

Your task is merely to understand how the architecture (and operation) of the given MLP translates to a CNN. You will have to determine how many layers your CNN must have, how many filters in each layer, the kernel size of the filters, and their strides and their activations. You must decide (based on the given MLP) what the weights of each of the filters must be (i.e what are the values of each kernel). The final output (after flattening) must have 124 components.

To test your code locally you are given a sample weights file, a sample input and the correct 124 component output. The autograder uses those to check your implementation.

So your tasks include figuring out:

- How to initialize the given weights to your CNN module (how to construct the filters, and select their strides)
- Designing the CNN architecture to correspond to a Scanning MLP

You should require very little additional code over what you have already written for the previous questions, besides the portion required to read the provided weights file and convert it to the CNN, and creating a few Conv1D instances.

1.4 CNN as a Distributed Scanning MLP [20 points]

This section of the homework is very similar to 1.3, with the one difference that the MLP provided to you is a shared-parameter network that captures a distributed representation of the input.

You must compose a CNN that will perform the same computation as scanning a given input with a given multi-layer perceptron.

The network still has 8 first-layer neurons, 16 second-layer neurons and 4 third-layer neurons. However, many of the neurons have identical parameters. As before, the MLP scans the input with a stride of 4 time instants. The parameter-sharing pattern of the MLP is illustrated in Figure 1. As mentioned, the MLP is a 3 layer network with 28 neurons. Neurons with the same color in a layer share the same weights. You may find it useful to visualize the weights matrices to see how this symmetry translate to the weights.

You are required to identify the symmetry in this MLP and use that to come up with the architecture of the CNN (number of layers, number of filters in each layer and their kernel width and stride, and the activations in each layer).

The aim of this task is to understand how scanning with this distributed-representation MLP (with shared parameters) can be represented as a Convolutional Neural Network. The weights of the CNN should be initialized using the `mlp_weights_part_c.npy` file. The file consists of an NumPy array containing 3 NumPy arrays, each corresponding to a layer in the MLP. The dimensions of each layer-wise NumPy arrays are (`inchannel`, `outchannel`) corresponding to the given MLP. Make use of the dimensions of each layer to further understand the MLP.

Your tasks include figuring out:

- How to initialize the given weights to your CNN module
- Designing the CNN architecture to correspond to a Distributed Scanning MLP

You are provided with a sample weights file for this portion as well, as well as the sample output. You must use the same sample input as that provided in 1.3.

As in 1.3, the only additional code required will be the portion needed to read the weights file and convert it to a CNN.

The autograder will run your CNN with a different set of weights, on a different input (of the same size as the sample input provided to you). The output of your code will be compared to a gold standard. The MLP employed by the autograder will have the same parameter sharing structure as your sample MLP. The weights, however, will be different. We will test the 124-component output of your CNN against a gold standard, as before.

1.5 Submission of CNN architectures for 1.3 and 1.4

On autolab and the local autograder, section 1.3 is called the part B of this Homework and 1.4 is called the part C. The CNN architectures should be written in the file named cnn.py in class CNN_B and CNN_C respectively. (CNN_B for 1.3 and CNN_C for 1.4). This is the file that the autograder would use for evaluating your score for parts B and C (1.3 and 1.4). Include this file in the directory hw2, and tar the directory to create a handin.tar. Overall, your handin would include a directory called hw2 which would contain files layers.py (for part A) and cnn.py (for parts B and C).

2 Lorem Ipsum

That's all. As always, feel free to ask on Piazza if you have any questions.

Good luck and enjoy the challenge!

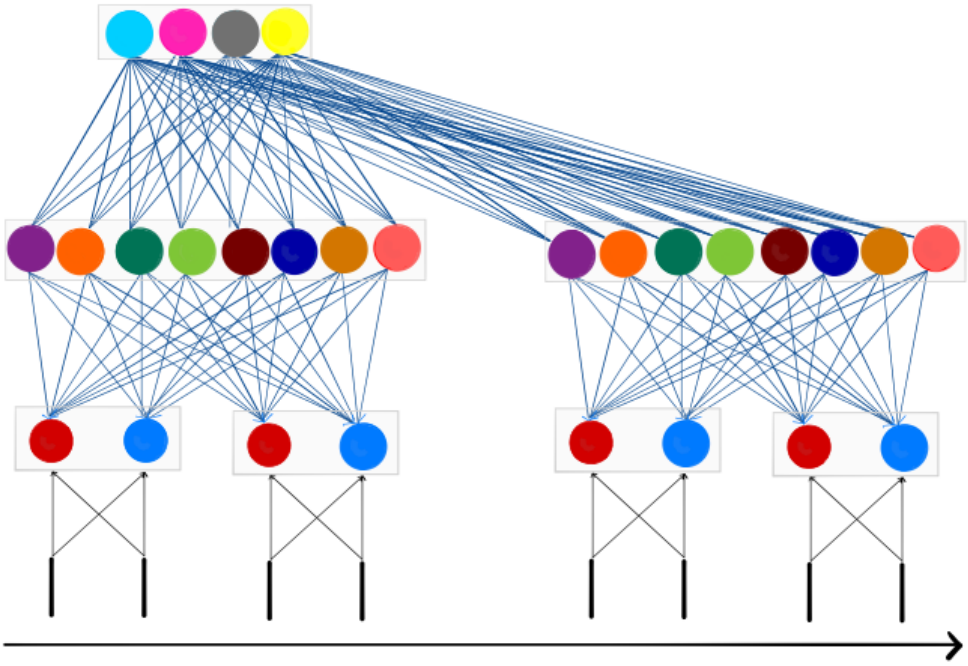

Figure 1: The MLP network architecture