Homework 2 Part 1

An Introduction to Convolutional Neural Networks

11-785: Introduction to Deep Learning (Spring 2020)

OUT: **February 9, 2020**
DUE: **March 7, 2020, 11:59 PM EST**

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:**
  - **MyTorch**: An introduction to the library structure... You probably don't read this anyway.
  - **Multiple Choice**: These are a series of multiple choice (autograded) questions which will speed up your ability to complete... Have you ever felt like a plastic bag?
  - **NumPy Based Convolutional Neural Networks**: All of the problems in this will be graded on Autolab. You can download the starter code/mytorch folder structure from Autolab as well... David Bowie was revolutionary.
- **Directions:**
  - You are required to do this assignment in the Python (version 3) programming language. Do not use any auto-differentiation toolboxes (PyTorch, TensorFlow, Keras, etc) - you are only permitted and recommended to vectorize your computation using the Numpy library.
  - We recommend that you look through all of the problems before attempting the first problem. However we do recommend you complete the problems in order, as the difficulty increases, and questions often rely on the completion of previous questions.
  - If you haven't done so, use pdb to debug your code effectively.

1 MyTorch

The culmination of all of the Homework Part 1's will be your own custom deep learning library, which we are calling *MyTorch* ©. It will act similar to other deep learning libraries like PyTorch or Tensorflow. The files in your homework are structured in such a way that you can easily import and reuse modules of code for your subsequent homeworks. For Homework 2, MyTorch will have the following structure:

- mytorch
  - loss.py (Copy your file from HW1P1)
  - activation.py (Copy your file from HW1P1)
  - batchnorm.py (Not necessary)
  - linear.py (Copy your file from HW1P1)
  - conv.py
- hw2
  - hw2.py
  - mlp_scan.py
  - mlp.py
  - mc.py
- autograder
  - hw2_autograder
    - runner.py
- create_tarball.sh
- exclude.txt

- **For** using code from Homework 1, ensure that you received all 95 autograded points.
- **Install** Python3, NumPy and PyTorch in order to run the local autograder on your machine:

```
pip3 install numpy
pip3 install torch
```

- **Hand-in** your code by running the following command from the top level directory, then **SUBMIT** the created *handin.tar* file to autolab:

```
sh create_tarball.sh
```

- **Autograde** your code by running the following command from the top level directory:

```
python3 autograder/hw2_autograder/runner.py
```

- **DO:**
  - We strongly recommend that you understand the Numpy functions `reshape` and `transpose` as they will be required in this homework.
- **DO NOT:**
  - Import any other external libraries other than numpy, as extra packages that do not exist in autolab will cause submission failures. Also do not add, move, or remove any files or change any file names.

2 Multiple Choice [5 points]

These questions are intended to give you major hints throughout the homework. Answer the questions by returning the correct letter as a string in the corresponding question function in **hw2/mc.py**. Each question has only a single correct answer. Verify your solutions by running the local **autograder**. To get credit (5 points), you must answer **all** questions correctly.

---

(1) **Question 1: Given the following architecture of a scanning MLP, what are the parameters of the equivalent Time Delay Neural Network which uses convolutional layers?**
As you have seen in the lectures, a convolutional layer can be viewed as an MLP which scans the input. This question illustrates an example of how the parameters are shared between a scanning MLP and an equivalent convolutional nework for 1 dimensional input.
(Help[1])(More help[2])[1 point]

The basic MLP is shown below. The sequence of little black bars shows the time sequence of input vectors. All blocks with the same background color are identical and share parameters (weights and biases). Neurons represented using the same color are identical and share parameters.

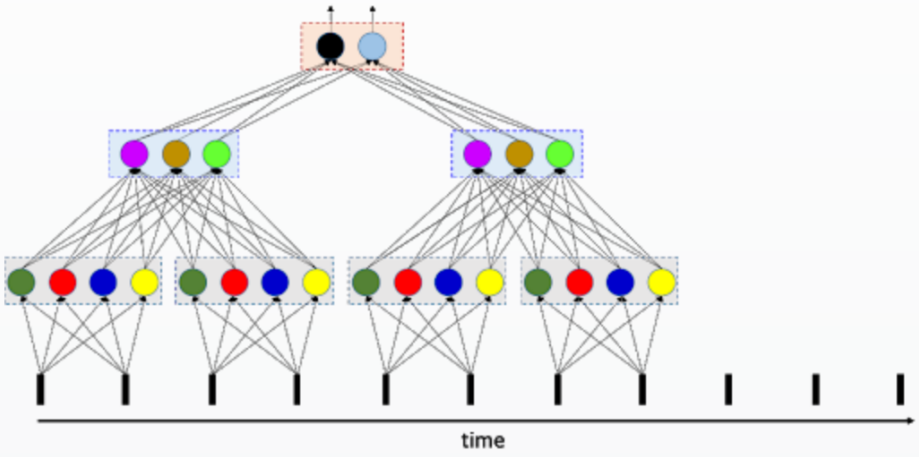


Figure 1: The architecture of a scanning MLP

(A) The first hidden layer has 4 filters of kernel-width 2 and stride 2; the second layer has 3 filters of kernel-width 8 and stride 2; the third layer has 2 filters of kernel-width 6 and stride 2

(B) The first hidden layer has 4 filters of kernel-width 2 and stride 2; the second layer has 3 filters of kernel-width 2 and stride 2; the third layer has 2 filters of kernel-width 2 and stride 2

(C) The first hidden layer has 2 filters of kernel-width 4 and stride 2; the second layer has 3 filters of kernel-width 2 and stride 2; the third layer has 2 filters of kernel-width 2 and stride 2

[1] Allow the input layer to be of an arbitrary length. The shared parameters should scan the entire length of the input with a certain repetition. In the first hidden layer, the horizontal gray boxes act as the black lines from the input layer. Why? Think...
[2] Understanding this question will help you with 3.3 and 3.4.

(2) **Question 2: Ignoring padding and dilation, which equations below are equivalent for calculating the out dimension (width) for a 1D convolution (L_{out} at `https://pytorch.org/docs/stable/nn.html#torch.nn.Conv1d`) (// is integer division)? [1 point]**

```
eq 1: out_width = (in_width - kernel + stride) // stride
eq 2: out_width = ((in_width - 1 * (kernel - 1) - 1) // stride) + 1
eq 3: great_baked_potato = (2*potato + 1*onion + celery//3 + love)**(sour cream)
```

(A) eq 1 is the one and only true equation

(B) eq 2 is the one and only true equation

(C) eq 1, 2, and 3 are all true equations

(D) eq 1 and 2 are both true equations

(3) **Question 3: In accordance with the image below, choose the correct values for the corresponding channels, width, and batch size given stride = 2 and padding = 0? [1 point]**

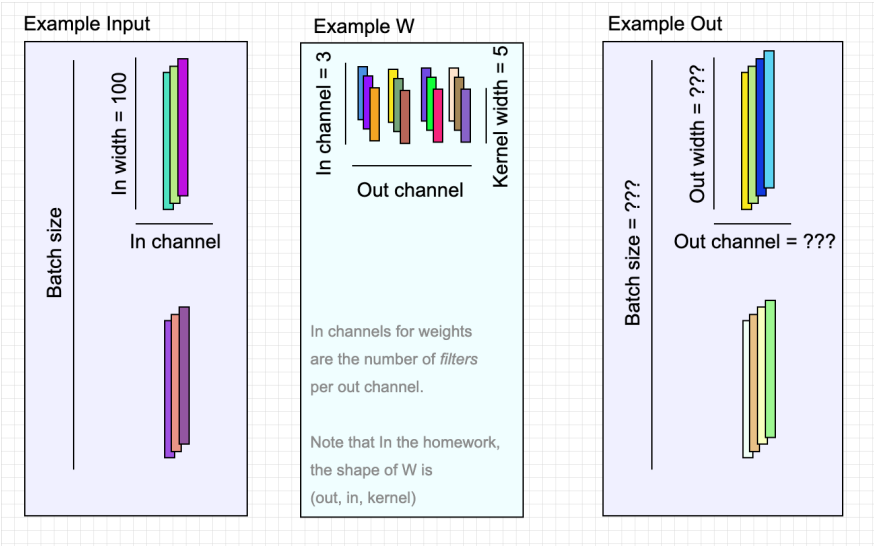


Figure 2: Example dimensions resulting form a 1D Convolutional layer

(A)

```
Example Input: Batch size = 2, In channel = 3, In width = 100
Example W: Out channel = 4, In channel = 3, Kernel width = 5
Example Out: Batch size = 2, Out channel = 4, Out width = 20
```

(B)

```
Example Input: Batch size = 2, In channel = 3, In width = 100
Example W: Out channel = 4, In channel = 3, Kernel width = 5
Example Out: Batch size = 2, Out channel = 4, Out width = 48
```

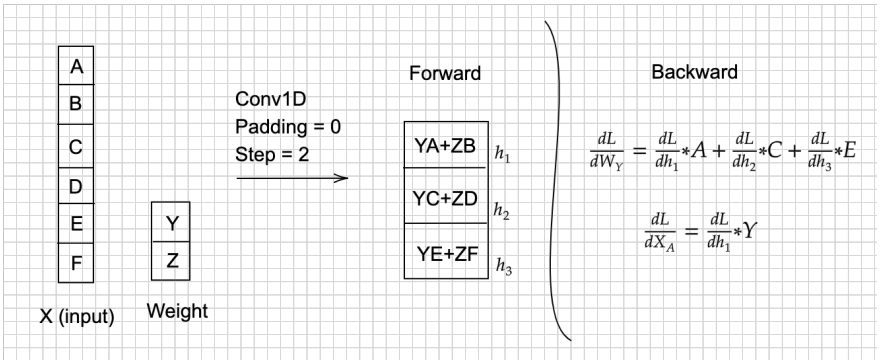
(4) **Question 4: Explore the usage of the command numpy.tensordot. Run the following example and verify how this command combines broadcasting and inner products efficiently. What is the shape of C and the value of C[0,0]? [1 point]**

```
A = np.arange(30.).reshape(2,3,5)
B = np.arange(24.).reshape(3,4,2)
C = np.tensordot(A,B, axes = ([0,1],[2,0]))
```

(A) `[5,4] and 820`

(B) `[4,5] and 1618`

(5) **Question 5: Given the toy example below, what are the correct values for the gradients? If you are still confused about backpropagation in CNNs watch the lectures or Google backpropagation with CNNs? [1 Point]**



(A) I have read through the toy example and now I understand backpropagation with Convolutional Neural Networks.

(B) This whole baked potato trend is really weird.

(C) Seriously, who is coming up with this stuff?

(D) Am I supposed to answer A for this question, I really don't understand what is going on anymore?

3 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.

Your implementations will be compared with PyTorch, **but you can only use NumPy in your code**.

3.1 Convolutional layer [60 points]

Implement the `Conv1D` class in `mytorch/conv.py` so that it has similar usage and functionality to `torch.nn.Conv1d`.

- The class `Conv1D` has four arguments: `in_channel`, `out_channel`, `kernel_size` and `stride`. They are all positive integers.
- A detailed explanation of the arguments has been discussed in class and you can also find them in the PyTorch docs: `https://pytorch.org/docs/stable/nn.html#torch.nn.Conv1d`
- **We do not consider other arguments such as `padding`.**
- **Note**: Changing the shape/name of the provided attributes is **not allowed**. Like in HW1P1 we will check the value of these attributes.

3.1.1 Forward [20 points]

Calculate the return value for the `forward` method of `Conv1d`.

- Input shape: (batch_size, in_channel, in_width)
- Output Shape: (batch_size, out_channel, out_width)

3.1.2 Backward [40 points]

Write the code for the `backward` method of `Conv1d`.

- The input `delta` is the derivative of the loss with respect to the output of the convolutional layer. It has the same shape as the convolutional layer output.
- `dW and db`: Calculate `self.dW` and `self.db` for the `backward` method. `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`: Calculate the return value for the `backward` method. `dx` is the derivative of the loss with respect to the input of the convolutional layer and has the same shape as the input.

3.2 Flatten layer

Implement the `forward` and `backward` methods of the `Flatten` class in `mytorch/conv.py`.

The autograder does not check your code for the Flatten layer as there are no points for it, but you will be required to implement it correctly to use it in the subsequent sections.

- Since we need to use linear layers for many tasks such as classification, we need one more layer to transform the output of convolutional layers into a legal input to the linear layers.
- In the forward method, the input `x` is the input of the flatten layer and its shape is (`batch_size`, `in_channel`, `in_width`). Just reshape the input to get the returned value with a shape of (`batch_size`, `in_channel`*`in_width`).
- In the backward method, 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.

3.3 CNN as a Simple Scanning MLP [10 points]

In `hw2/mlp_scan.py` for `CNN_SimpleScanningMLP` 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.

The Multi-layer Perceptron is composed of three layers and the architecture of the model is given in `hw2/mlp.py` included in the handout. You do not need to modify the code in this file, it is only for your reference.

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 3.1 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.

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. The final output (after flattening) must have 124 components.

Your tasks include:

- Designing the CNN architecture to correspond to a Scanning MLP
  - Create Conv1D objects defined as `self.conv1`, `self.conv2`, `self.conv3`, in the init method of `CNN_SimpleScanningMLP`.
  - For the Conv1D instances, you must specify the: `in_channel`, `out_channel`, `kernel_size`, and `stride`.
  - Add those layers along with the activation functions/ flatten layer to a class attribute you create called `self.layers`.
  - Initialize the weights for each convolutional layer, using the init_weights method. You must discover the orientation of the initial weight matrix(of the MLP) and convert it for the weights of the Conv1D layers.
  - This will involve (1) reshaping a transposed weight matrix into `out_channel, kernel_size, in_channel` and (2) transposing the weights back into the correct shape for the weights of a Conv1D instance, `out_channel, in_channel, kernel_size`.
  - Use pdb to help you debug, by printing out what the initial input to this method is. Each index into the given weight's list corresponds to the Conv1D layer. I.e. weights[0] are the weights for the first Conv1D instance.

The paths have been appended such that you can create layers with the calls to the class themselves, i.e. to make a ReLU layer, just use `ReLU()` . You have a weights file which will be used to autograde your network locally.

3.4 CNN as a Distributed Scanning MLP [10 points]

Complete 3.4 in `hw2/mlp_scan.py` in the class `CNN_DistributedScanningMLP`. This section of the homework is very similar to 3.3, except 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 MLP.

The network 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 3. 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 translates to 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, their kernel width, 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.

Your tasks include:

- Designing the CNN architecture to correspond to a Distributed Scanning MLP
  - Create Conv1D objects defined as `self.conv1`, `self.conv2`, `self.conv3`, in the init method of `CNN_DistributedScanningMLP` by defining the: `in_channel`, `out_channel`, `kernel_size`, and `stride` for each of the instances.
  - Then add those layers along with the activation functions/ flatten layer to a class attribute you create called `self.layers`.
  - Initialize the weights for each convolutional layer, using the init_weights method. Your job is to discover the orientation of the initial weight matrix and convert it for the weights of the Conv1D layers. This will involve (1) reshaping the transposed weight matrix into `out_channel, kernel_size, in_channel` and then (2) transposing the weights again into the correct shape for the weights of a Conv1D instance. You must **slice** the initial weight matrix to account for the shared weights.

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 MLP employed by the autograder will have the same parameter sharing structure as your sample MLP. The weights, however, will be different.

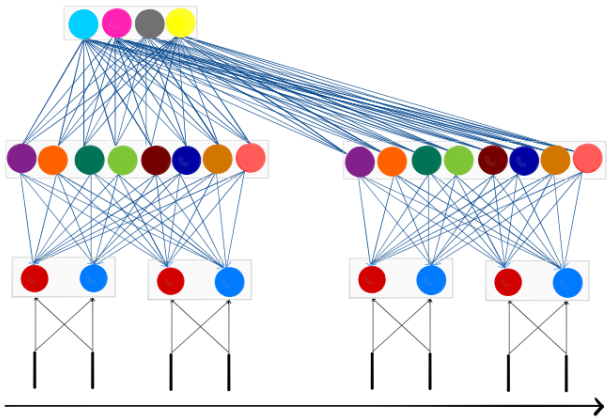


Figure 3: The MLP network architecture for 3.4

3.5 Build a CNN model [15 points]

- In `hw2/hw2.py` implement a CNN model.

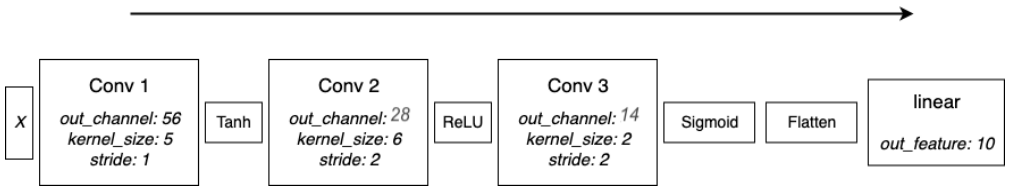


Figure 4: The architecture of a simple CNN that you will be implementing.

- First, initialize your `convolutional_layers` in the init function using `Conv1d` instances.
  - Then initialize your flatten and linear layers.
  - You have to calculate the `out_width` of the final CNN layer and use it to correctly give the linear layer the correct input shape.
- Now, implement the `forward` method, which is extremely similar to the MLP code from HW1.
- There are no batch norm layers.
- Remember to add the `Flatten` and `Linear` layer after the convolutional layers and activation functions.
- Next, implement the `backward` method which is extremely similar to what you did in HW1.
- The `step` function and `zero gradient` function are already implemented for you.