

# Homework 2 Part 1

## An Introduction to Convolutional Neural Networks

11-785: INTRODUCTION TO DEEP LEARNING (FALL 2021)

OUT: **September 30, 2021**  
DUE: **October 21, 2021, 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:**

- **Multiple Choice:** These are a series of multiple choice questions (autograded) which will speedup your ability to complete this homework.
- **NumPy Based Convolutional Neural Networks:** implement the forward and backward passes of a 1D & 2D convolutional layer and a flattening layer. All of the problems in this will be graded on Autolab. You can download the starter code from Autolab as well.
- **CNNs as Scanning MLPs:** Two questions on converting a linear scanning MLP to a CNN.
- **Implementing a CNN Model:** Combine all the pieces to build a CNN model.
- **Appendix:** This contains information on some theory that will be helpful in understanding the homework.

- **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 (One much simple way, just use print()! Print the shape or anything else that can help you find the bug!)and please PLEASE Google your error messages before posting on Piazza.

# 1 Introduction

In this assignment, you will continue to develop your own version of PyTorch, which is of course called MyTorch (still a brilliant name; a master stroke. Well done!). In addition, you'll convert two scanning MLPs to CNNs and build a CNN model.

## 2 MyTorch Structure

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 (Copy your file from HW1P1)
  - linear.py (Copy your file from HW1P1)
  - conv.py
  - HINT: You can use `np.tanh()` in `Tanh()` class if there is related error raising.
- hw2
  - hw2.py
  - mlp\_scan.py
  - mlp.py
  - mc.py
- autograder
  - hw2.autograder
    - \* runner.py
- toyproblem
  - toy.py
  - value.py
  - Toyproblem.pdf
- create\_tarball.sh
- exclude.txt

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

```
pip install numpy
pip 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
```

- **Toy problem** we have toy problem for you to test the code in `conv.py`, which is the core part of this homework. You can learn more detail in the document.

- **DO:**

- We strongly recommend that you understand the Numpy functions `reshape`, `transpose`, and `tensordot` as they will be essential 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.

### 3 Multiple Choice

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 network for 1 dimensional input. (Help<sup>1</sup>)(More help<sup>2</sup>)[1 point]

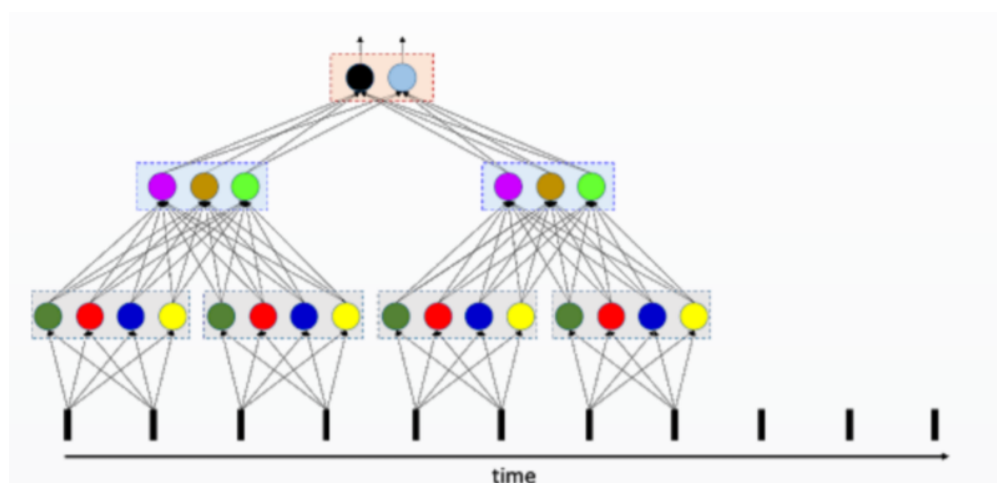


Figure 1: The architecture of a scanning MLP

- (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
- (2) **Question 2: Considering 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/generated/torch.nn.Conv1d.html#torch.nn.Conv1d>) (// is integer division)? [1 point]**

```
eq 1: out_width = [in_width + 2 * padding - dilation * (kernel - 1) - 1] // stride + 1
eq 2: out_width = [(in_width_padded - kernel_dilated) // stride] + 1, where in_width_padded
      = in_width + 2 * padding, kernel_dilated = (kernel - 1) * (dilation - 1) + kernel
eq 3: great_baked_potato = (2*potato + 1*onion + celery//3 + love)**(sour_cream)
```

<sup>1</sup>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...

<sup>2</sup>Understanding this question will help you with 3.3 and 3.4.

- (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]**

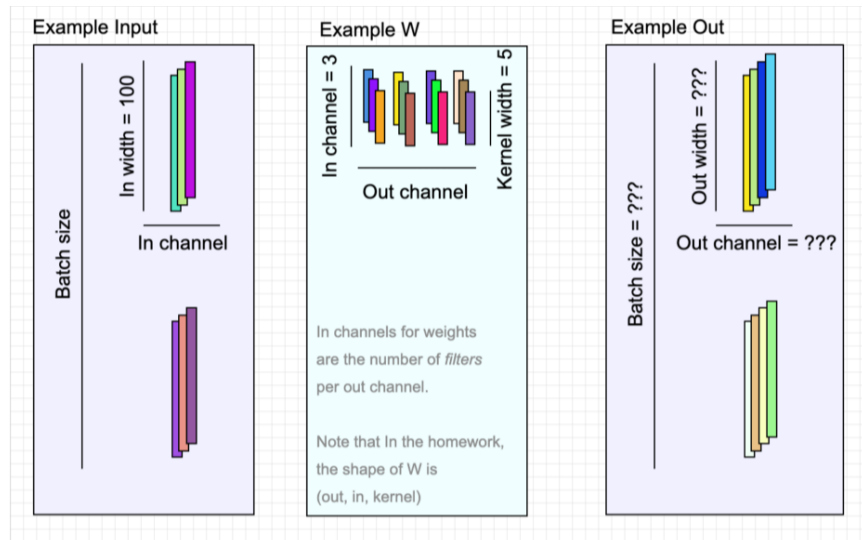


Figure 2: Example dimensions resulting from a 1D Convolutional layer

- (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) 

---

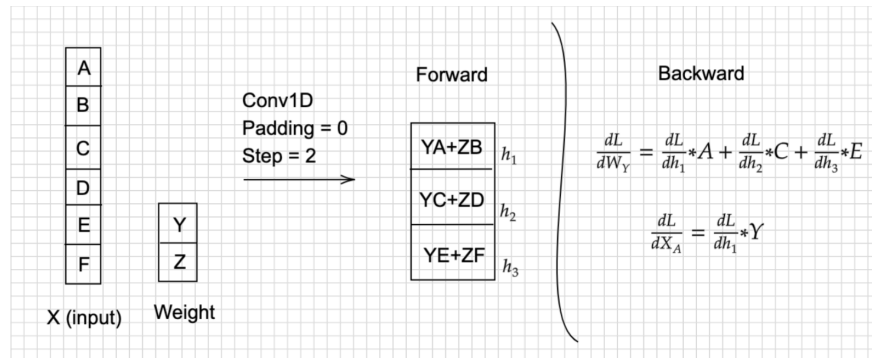
Example Input: Batch size = 2, In channel = 3, In width = 100  
Example W: Out channel = 4, In channel = 3, Kernel width = 5  
Example Out: Batch size = 2, Out channel = 4, Out width = 48

---
- (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?
- (6) **Question 6:** This question will help you understand why CNN is helpful and powerful. Please choose the listing properties of CNN.[0 Point]  
(For more details, you can refer to <https://www.deeplearningbook.org/contents/convnets.html>)(Typo<sup>3</sup>)
- (1) Sparse Interactions: It means every output interacts with every input units. This is accomplished by making the kernel smaller than the input. So that we only needs fewer parameters and fewer operations. Like processing images, we only needs to detect small features, like the edges.
- (2) Parameter Sharing: It refers to using the same parameters for more than one function in a model. One can say that the network has tied weights because the value of the weight applied to one input is tied to the value of a weight applied elsewhere
- (3) Equivalent Representation: In the case of convolution, the parameter sharing causes the layer to have a property called equivalent representation to translation. Equivalent means if the inputs changes, the output changes in the same way.
- (A) (1) is correct
- (B) (1) and (2) are correct
- (C) (2) and (3) are correct
- (D) all of them are correct

<sup>3</sup>There is a typo in the last sentence on Page 334 of deep learning book. It should be "applied convolution to I" instead of "I' "

- (7) **Question 7:** This question will help you visualize the CNN as a scanning MLP. Given a weight matrix for a scanning MLP, you want to determine the corresponding weights of the filter in its equivalent CNN.  $W_{MLP}(input\_size, output\_size)$  is the weight matrix for a layer in a MLP and  $W_{conv}(out\_channel, in\_channel, kernel\_size)$  is the equivalent convolutional filter. As discussed in class, you want to use the convolutional layer instead of scanning the input with an entire MLP, then how will you modify  $W_{MLP}$  to find  $W_{conv}$  [0 Point]

(To help you visualize this better, you can refer to Figure 3 and the lecture slides)

- (A)  $W_{conv} = W_{MLP}.reshape(input\_channel, kernel\_size, output\_channel).transpose()$   
where  $input\_channel * kernel\_size = input\_size$  and  $output\_size = output\_channel$
- (B)  $W_{conv} = W_{MLP}.reshape(kernel\_size, input\_channel, output\_channel).transpose()$   
where  $input\_channel * kernel\_size = input\_size$  and  $output\_size = output\_channel$
- (C)  $W_{conv} = W_{MLP}.reshape(output\_channel, input\_channel, kernel\_size)$   
where  $input\_channel * kernel\_size = input\_size$  and  $output\_size = output\_channel$
- (D) Am I supposed to answer A for this question, I really don't understand what is going on anymore?

## 4 NumPy Based Convolutional Neural Networks

In this section, you need to implement convolutional neural networks using the NumPy library only. Python 3, NumPy $\geq 1.16$  and PyTorch $\geq 1.0.0$  are suggested environment.

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

### 4.1 Convolutional layer : Conv1D [30 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 seven arguments: `in_channel`, `out_channel`, `kernel_size`, `stride`, `padding`, and `dilation`(default=1). They are all positive integers.
- A detailed explanation of the arguments has been discussed in class and you can also find them in the [torch documentation](#).
- **Note:** Changing the shape/name of the provided attributes is **not allowed**. Like in HW1P1 we will check the value of these attributes.

#### 4.1.1 Forward [10 points]

Calculate the return value for the `forward` method of `Conv1d`. The psuedocode for implementing the `conv1d` forward pass can be found in the slides. In the forward pass, you will need to calculate the size of the output data (`output_size`). Since we are not considering padding or dilation, we can calculate the output size using this:

$$\text{output\_size} = [(\text{input\_size} - \text{kernel\_size}) // \text{stride}] + 1$$

The shapes of the input and output should be as follows:

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

#### 4.1.2 Backward [20 points]

Write the code for the `backward` method of `Conv1d`.(Help<sup>4</sup>)

- 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.

---

<sup>4</sup>Please refer to Appendix for more information about how to deal with padding and dilation



## 4.2 Convolutional layer : Conv2D [20 points]

In this section you will implement a 2D convolutional layer, this time from scratch.

In Section 4.1, you implemented 1D convolutions, which involved the scanning over multiple channels each with a 1D sequence. In this section, you will consider a higher dimensional algorithm, the 2D convolution.

This task requires you to implement both forward propagation and backward propagation for 2D convolutions. Fill out the code for the rest of this class as you see fit. You are encouraged to reference the pseudocode presented in the lectures and consider the lecture notes on CNNs.

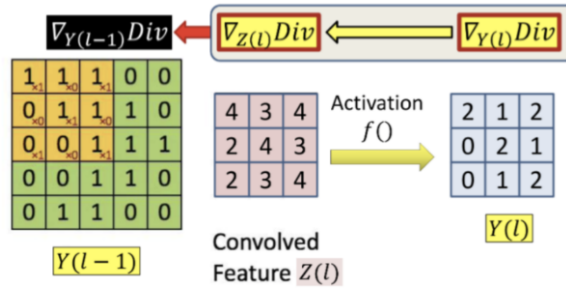


Figure 3: 2D Convolution Example

**First**, create the user-facing `Conv2d(Module)` class in `conv.py`.

During initialization, it should accept four args: **in\_channel**, **out\_channel**, **kernel\_size**, **stride=1**.

Use the following code for weight/bias initialization:

```
# Kaiming init (fan-in) (good init strategy)
bound = np.sqrt(1 / (in_channel * kernel_size * kernel_size))
weight = np.random.uniform(-bound, bound, size=(out_channel, in_channel,
                                                kernel_size, kernel_size))
self.weight = Tensor(weight, requires_grad=True, is_parameter=True)
bias = np.random.uniform(-bound, bound, size=(out_channel,))
self.bias = Tensor(bias, requires_grad=True, is_parameter=True)
```

## 4.3 Convolutional layer : Padding and dilation [10 points]

In the previous sections, we have not considered padding (pad values around the input) and dilation (dilate the kernels), which are of great importance to improve the performance of CNNs. In this section, we will implement and apply the padding and dilation in `Conv2d` class

With padding and dilation, we can calculate the output size using the following formulas:

$$\text{output\_size} = [(\text{input\_size} + 2 * \text{padding} - \text{dilation} * (\text{kernel\_size} - 1) - 1) // \text{stride}] + 1$$

Or in a much more understandable way:

$$\text{output\_size} = [(\text{input\_size\_padded} - \text{kernel\_dilated}) // \text{stride}] + 1$$

with  $\text{input\_size\_padded} = \text{input\_size} + 2 * \text{padding}$

$\text{kernel\_dilated} = (\text{kernel\_size} - 1) * (\text{dilation} - 1) + \text{kernel\_size}$  (Help<sup>5</sup>)

<sup>5</sup>If you don't understand what is dilation or padding, please refer to torch official documents or Appendix

## 4.4 Flatten layer

In `nn/conv.py`, complete `Flatten()`.

This layer is often used between `Conv` and `Linear` layers, in order to squish the high-dim convolutional outputs into a lower-dim shape for the linear layer. For more info, see the [torch documentation](#).

**Hint:** This can be done in one line of code, with no new operations or (horrible, evil) broadcasting needed.

**Bigger Hint:** Flattening is a subcase of reshaping. [numpy.prod](#) may be useful.

## 5 Converting Scanning MLPs to CNNs [Total: 20 points]

### 5.1 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  $128 \times 24$  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. Architecture is as follows:

```
[Flatten(), Linear(8 * 24, 8), ReLU(), Linear(8, 16), ReLU(), Linear(16, 4)]
```

- 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 ( $4 \times 31$ ) 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 the above sections to compose a Convolution Neural Network which will behave identically to scanning the input with the given MLP as explained above.

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.

*For more of a theoretical understanding of Simple Scanning MLPs, please refer to the Appendix section.*

## 5.2 CNN as a Distributed Scanning MLP [10 points]

Complete 5.2 in `hw2/mlp_scan.py` in the class `CNN_DistributedScanningMLP`. This section of the homework is very similar to 5.1, 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.**

**Architecture details:**

- 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 4. 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 activation 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**.

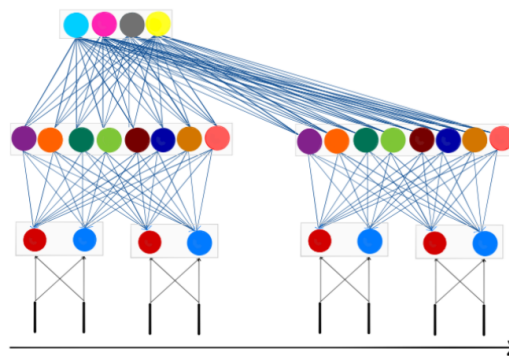


Figure 4: The Distributed Scanning MLP network architecture

**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`
    - (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.

*For more of a theoretical understanding of Distributed Scanning MLPs, please refer to the Appendix section.*

## 6 Build a CNN model [Total: 15 points]

Finally, in `hw2/hw2.py`, implement a CNN model.

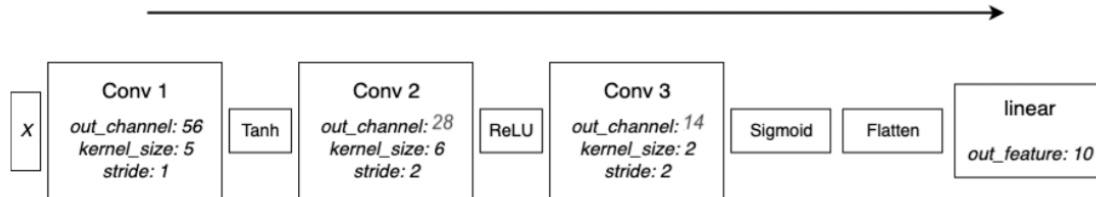


Figure 5: CNN Architecture to implement.

- 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. You can use some of the formulas referenced in the previous sections to calculate the output size of a layer.
- 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.
- Remember that we provided you the `Tanh` and `Sigmoid` code; if you haven't already, see earlier instructions for copying and pasting them in.
- Please refer to the lecture slides for pseudocodes.

We ask you to implement this because you may want to modify it and use it for HW2P2.

Great work as usual!! All the best for HW2P2!!

## 7 Appendix

### 7.1 Padding and Dilation: Forward and Backward

In this section, we will explain padding and dilation, and how the forward and backward pass changes with them. For a good visualization, you can refer to [this link](#).

#### 7.1.1 Padding

Padding controls the amount of padding applied to the input. When we have a deep neural network, we hope to keep enough information and padding will help. Padding will add zeros (right now we only consider constant padding value 0) at the edge of the input (Figure 6). So the padded input size will be:

$$\text{input\_size\_padded} = \text{input\_size} + 2 * \text{padding}$$

The forward padding is quite straightforward and you can use `numpy.pad`.

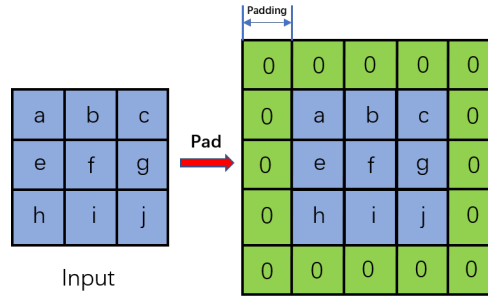


Figure 6: Padding Forward

However, will padding cause a huge difference when we do backpropagation? During backpropagation, we need to compute  $\frac{\delta L}{\delta w}$  and  $\frac{\delta L}{\delta x}$ , where  $x$  is the input and  $w$  are the kernel weights.

When computing  $\frac{\delta L}{\delta w}$ , we can assume the padded input as the original input. It won't affect the result of  $\frac{\delta L}{\delta w}$ . However, when computing  $\frac{\delta L}{\delta x}$ , we don't need the derivative of the pads (i.e. zero values inserted at the margin). What we need is the original  $\frac{\delta L}{\delta x}$  (without pads) for backpropagation.

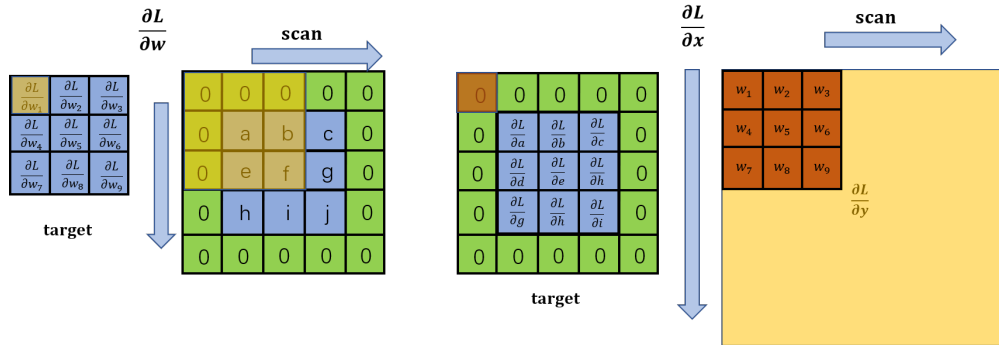


Figure 7: Padding Backward

#### 7.1.2 Dilation

Dilation controls the spacing between the weights in kernels. It aims at increasing the receptive field without increasing the parameters.

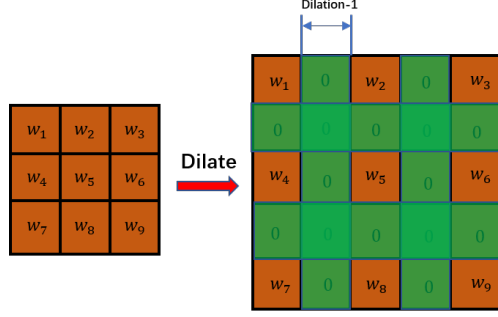


Figure 8: Dilation Forward

The default value of dilation is one, and Figure 8 shows the resulting matrix of applying dilation=1. We can apply backpropagation with dilation similar to how we dealt with padding.

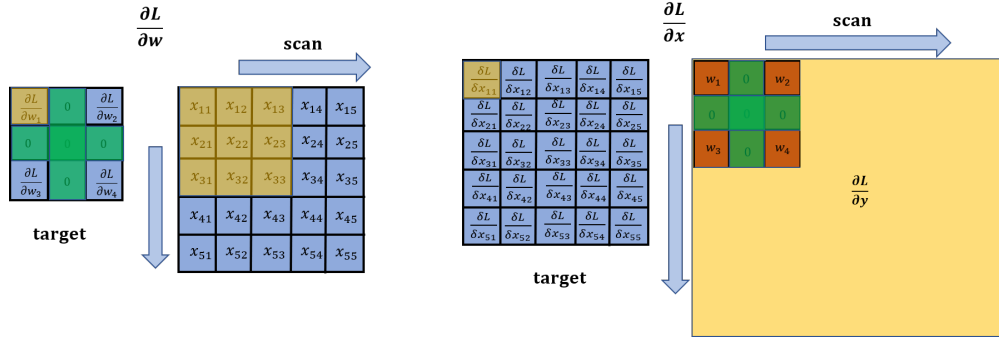


Figure 9: Dilation Backward

When we compute  $\frac{\partial L}{\partial x}$ , we can assume the dilated kernel as the original kernel. It won't affect the result of  $\frac{\partial L}{\partial x}$ . However, when we compute  $\frac{\partial L}{\partial w}$ , we don't need the derivative of the zero-dilations. What we need is the original  $\frac{\partial L}{\partial w}$  for backpropagation.

## 7.2 Scanning MLP : Illustration

Consider a 1-D CNN model (This explanation generalizes to any number of dimensions). Specifically consider a CNN with the following architecture:

- Layer 1: 2 filters of kernel width 2, with stride 2
- Layer 2: 3 filters of kernel width 3, with stride 3
- Layer 3: 2 filters of kernel width 3, with stride 3
- Finally a single softmax unit which combines all the outputs of the final layer.

This is a regular, if simple, 1-D CNN. The computation performed can be visualized by the figure below.

Input: The little black bars at the bottom represent the sequence of input vectors. There are two layer-1 filters of width 2.

Layer 1: The red and green circles just above the input represent these filters (each filter has one arrow going to each of the input vectors it analyzes, in the illustration; a more complete illustration would have as many arrows as the number of components in the vector).



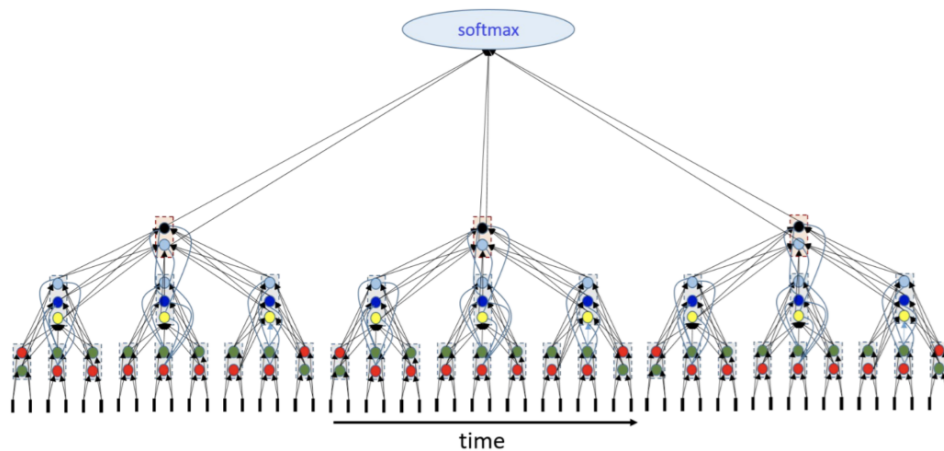


Figure 10: Scanning MLP Figure 1

Each filter (of width 2, stride 2) analyzes 2 inputs (that's the kernel width), then strides forward by 2 to the next step. The output is a sequence of output vectors, each with 2 components (one from each level-1 filter).

In the figure the little vertical rectangular bars shows the sequence of outputs computed by the two layer-1 filters.

The layer-1 outputs now form the sequence of output vectors that the second-layer filters operate on.

Layer-2: Layer 2 has 3 filters (shown by the dark and light blue circles and the yellow circle). Each of them gets inputs from three (kernel width) of the layer-1 bars. The figure shows the complete set of connections. The three filters compute 3 outputs, which can be viewed as one three-component output illustrated by the vertical second-level rectangles in the figure.

The layer-2 filters then skip 3 layer-1 vectors (stride 3) and then repeat the computation. Thus we get one 3-component layer-2 output for every three layer-1 outputs.

Layer 3 works on the outputs of layer 2.

Layer-3: Layer 3 consists of two filters (the black and grey circles) that get inputs from three layer-2 vectors (kernel width 3). Each of the layer 3 filters computes an output, so we get one 2-component output (shown by the orange boxes). The layer-3 units then stride 3 steps over the layer-2 outputs and compute the next output.

Softmax: The outputs of the layer-3 units are all jointly then sent on to the softmax unit for the final classification.

Now note that this computation is identical to "scanning" the input sequence of vectors with the MLP below, where the MLP strides by 18 steps of input after each analysis. The outputs from all the individual evaluations by the scanning MLP are sent to a final softmax.

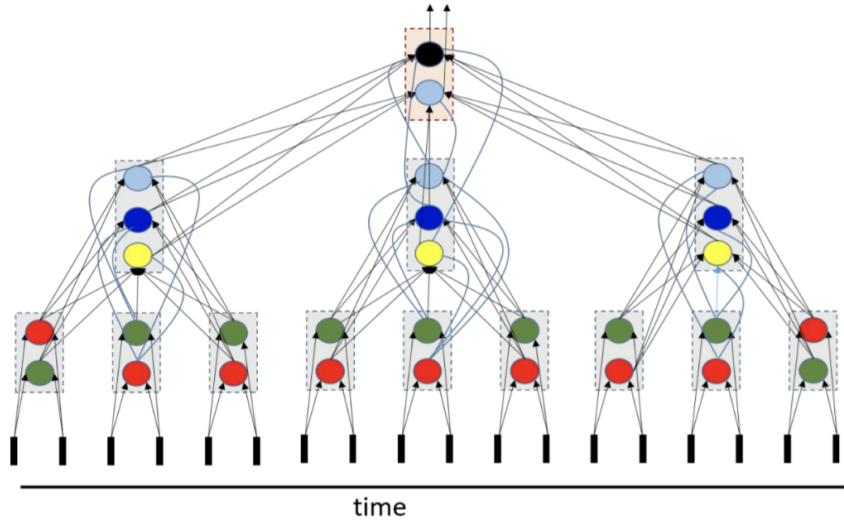


Figure 11: Scanning MLP Figure 2

The "scanning" MLP here has three layers. The first layer has 18 neurons, the second has 9 and the third has 2. So the CNN is actually equivalent to scanning with an MLP with 29 neurons.

Notably, since this is a distributed representation, and although the MLP has 29 neurons, it only has 7 unique neuron types. The 29 neurons share 7 shared sets of parameters.

It's sometimes more intuitive to use a horizontal representation of the arrangement of neurons, e.g.

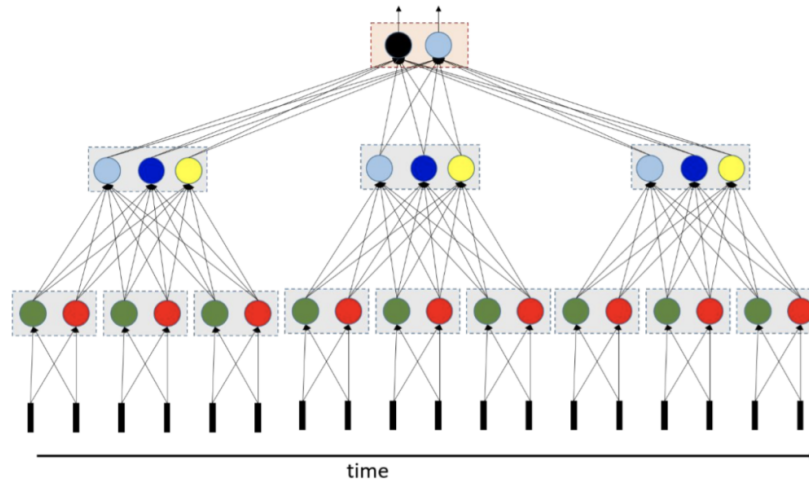


Figure 12: Scanning MLP Figure 3

Note that this figure is identical to the second figure shown. But it also leads to more intuitive questions such as "do the individual groups of neurons (shown in each rectangular bar) have to have scalar activations, or could they be grouped for vector activations. Such intuitions lead to other architectures.