Homework 2 Part 1

An Introduction to Convolutional Neural Networks

11-785: Introduction to Deep Learning (FALL 2022)

OUT: **September 30, 2022**
EARLY SUBMISSION BONUS: **October 14th, 2022**
DUE: **October 27, 2022, 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 help your friends debug
  - You are allowed to look at your friends code
  - You are allowed to copy math equations from any source that are not in code form
  - You are not allowed to type code for your friend
  - You are not allowed to look at your friends code while typing your solution
  - You are not allowed to copy and paste solutions off the internet
  - You are not allowed to import pre-built or pre-trained models
  - 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.

  We encourage you to meet regularly with your study group to discuss and work on the homework. You will not only learn more, you will also be more efficient that way. However, as noted above, the actual code used to obtain the final submission must be entirely your own.

- **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 layers, transpose convolution layers, flattening layer and pooling layers. 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.

- **Early submission bonus deadline:**
  - If you complete this assignment successfully and achieve full marks on Autolab before **October 14th, 2022, 11:59 PM, Eastern Time**, you will receive **5** point bonus for this assignment.

Homework objectives

If you complete this homework successfully, you would ideally have learned:

- How to write code to implement a CNN from scratch
  - How to write code to implement all components of a CNN from scratch
  - How to implement convolutional layers
  - How to implement pooling layers
  - How to implement downsampling and upsampling layers
  - How to chain these up, along with components you have already implemented in HW1P1 to compose a CNN of any size
- Your code will be able to perform forward inference through the CNNs
- How to write code to implement training of your CNN
  - How to write code to implement training of your CNN
  - How to perform a forward pass through your network
  - How to implement backpropagation through the convolutional layers
  - How to implement backpropagation through the pooling layers
  - How to implement backpropagation through resampling layers
  - How to combine these to perform backpropagation through an entire CNN to compute gradients (to train all parameters of the network)

Contents

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.

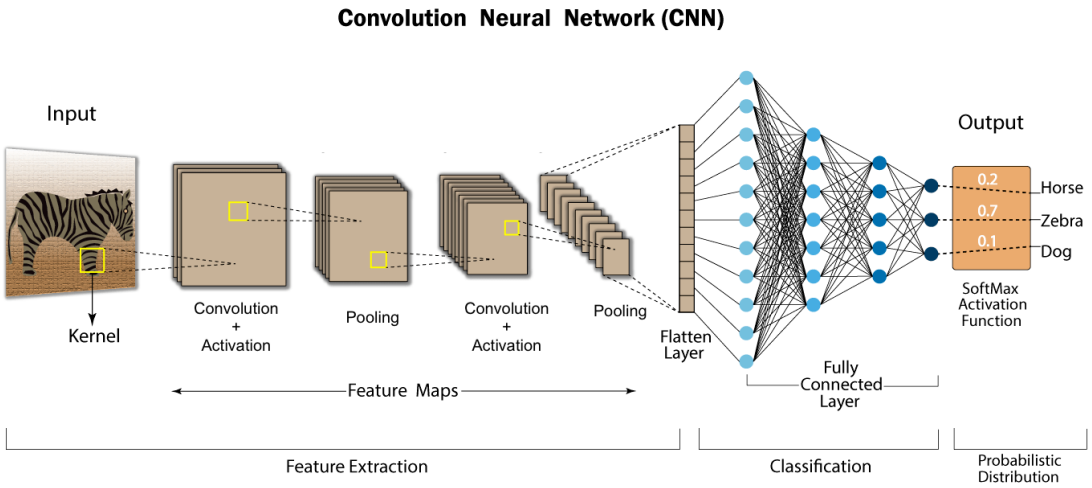


Figure 1: Standard 2d CNN

1.1 The Story:

What is a Convolutional Neural Network (CNN)? What is it comprised of? And why does it matter? How I can build one of my own?

If you have any of these questions, you will likely be pleasantly surprised by this homework. :-)

A Convolutional Neural Network is a position-invariant pattern detector. CNNs have wide-ranging use-cases in image classification, object detection, semantic segmentation, image captioning, natural language processing, forecasting, and more.

Before we dive into an overview of a CNN and why each part of this homework matters, a quick PSA: several parts of this homework have been implemented for you, or will be reused from the previous homework.

A Convolutional Neural Network consists of the following parts:

1.2 Convolutional Layer

- This first layer is the Convolutional Neural Network's namesake. A convolutional layer consists of many filters, where each filter is responsible for capturing a different pattern. The filters in this layer are used to extract features from the input. For example, in images of cars, a convolutional layer might extract the "wheel" patterns. Another convolutional layer's filters might extract the patterns formed by the frame of the car images. Yet another convolutional layer's filters might extract the patterns formed by the bumper/license plate in the front of the car images. Check out https://poloclub.github.io/cnn-explainer/ for a more visual explanation about CNNs!
- It's important to note that a Convolutional Layer will consist of a "forward" and "backward" operation, as we saw with MLPs in HW 1. As a quick recap: the "forward" operation is used to figure out what our layer can produce. We can think of this as the inference or prediction part of the layer, since we will be using what was learned with the "backward" operation to make a prediction. The "backward"

operation is used to train our layer. Please see later parts of this writeup, as well as the lectures on CNNs for more details about how to write these forward and backward operations.

1.3 Activation

- As we've seen extensively in HW 1, we will now apply an activation to the output of our convolutional layer.

1.4 Resampling (Upsampling and Downsampling)

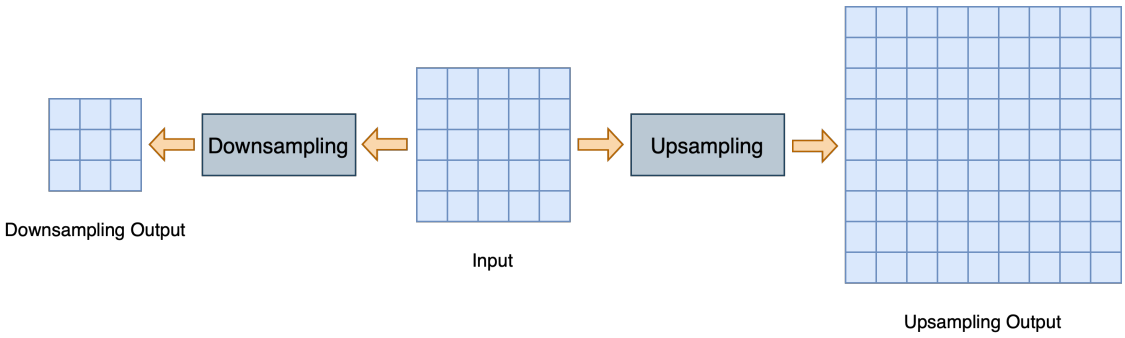


Figure 2: Resampling 2d input

- Now, let's take a quick pause to talk about resampling, which consists of upsampling and downsampling. To put it simply, upsampling is a way to "bloat" or fill up our input and make it bigger. Conversely, downsampling is a way to reduce the size of our input and make it smaller, while preserving the information that we need.
- Why are these necessary? Let's say you have a stride != 1. (Stride is the number of pixel the filter passes in each step). In that case, you can either modify your convolutional layer implementation to work for stride != 1, OR you can leave your convolutional layer implementation untouched, and tack on resampling to the OUTPUT of the convolutional layer to "simulate" the stride, so to speak. To understand resampling, let's concretely break up the convolutional layer into 3 main cases:
  - Stride = 1
    - In this case, after our convolutional layer, we will simply proceed on to Pooling. Nice! :)
  - Stride > 1
    - In this case, we will DOWNSAMPLE the output of our convolutional layer so that we can simulate a stride > 1.
  - Stride < 1
    - In this case, we will UPSAMPLE the output of our convolutional layer so that we can simulate a stride < 1.
- When you STRIDE this path (pun intended), you will need to implement the 1d and 2d versions of Upsampling and Downsampling. Note that Upsampling and Downsampling are direct inverse operations. So the forward of Upsampling is THE SAME as the backward of Downsampling. And the backward of Upsampling is THE SAME as the forward of Downsampling.
- Please refer to the lecture slides for more info. You may also find this link helpful: https://stats.stackexchange.com/quest vs-stride-for-downsampling/387522

1.5 Pooling

- Next, we have a pooling layer. Pooling is used for jitter invariance. Conventionally, pooling is done with strides > 1 so that there is also a reduction in size of the output map. But why do we need to decrease the size? This reduces computational costs, and can help increase speed of the network. A decrease in size means less work for the machine, right? :)
- There are several different types of pooling operations, but in this homework, we'll look at the 2 most popular: max pooling and mean pooling. Again, just like Convolutional Layers, we have forward (inference) and backward (training) operations for these Pooling Layers. Please refer to the rest of this writeup, as well as the lecture slides on CNNs for more details about these operations.

1.6 Flatten

- We will first need to flatten our pooled feature map(s) before passing into the classification layers. This part is quite simple; just flatten our output so far into a single 1-dimensional array.

1.7 Linear Layer

- Now, we'll insert this long vector of input data (consisting of flattened, pooled feature maps) into the linear layer. Now the fun stuff starts — this fully connected linear layer will allow our network to move towards performing a classification. It will help to map the representation between the input and the output.

1.8 Softmax Layer

- Victory at last. We have reached the softmax layer, which will allow for us to output a probability distribution over our output classes. We will use this to allow our CNN to perform a classification.

1.9 Putting it all together

- Finally, we'll put everything that we learned together, in hw2.py.

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)
  - linear.py (Copy your file from HW1P1)
  - conv.py
  - batchnorm.py
  - HINT: You can use np.tanh() in Tanh() class if there is related error raising.
- hw2
  - hw2.py
  - mlp_scan.py
  - mc.py
- autograder
  - hw2_autograder
    - runner.py
- 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:

  ```
  pip3 install -r requirements.txt
  ```

- **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`, `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 [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]

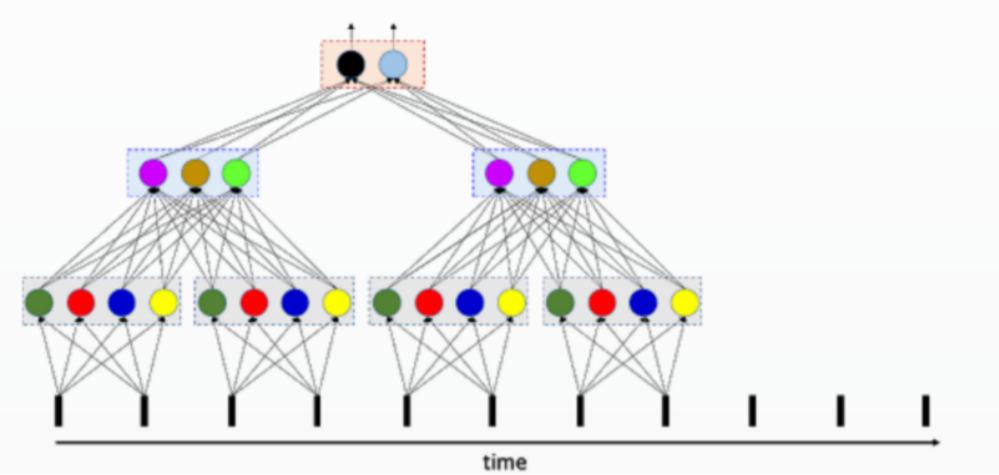


Figure 3: The architecture of a scanning MLP

(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_wdith_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)
```

[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.

(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]**

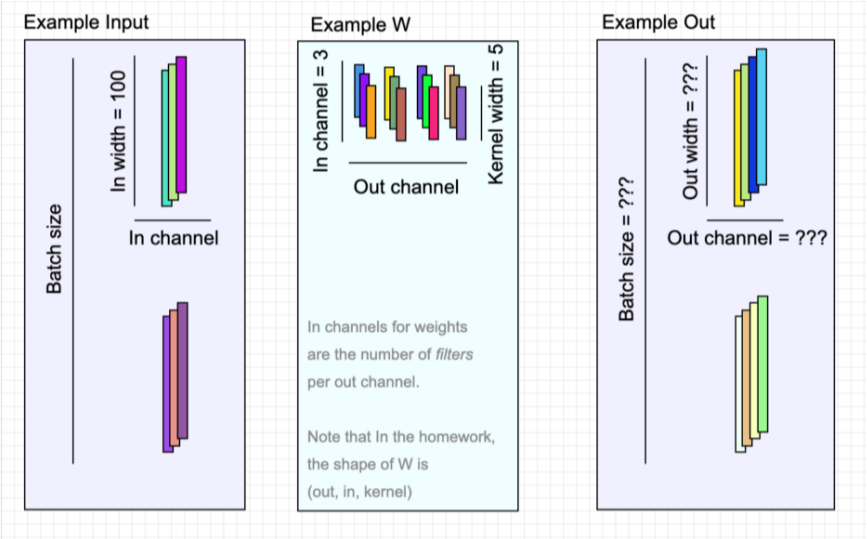


Figure 4: Example dimensions resulting form a 1d Convolutional layer

(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
```

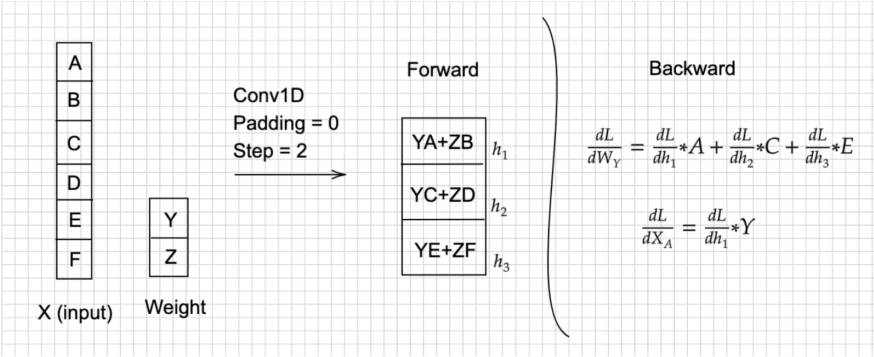
(4) **Question 4: Explore the usage of the command numpy.tensordot.** Run the following example and verify how this command combines broadcasting and inner products efficiently. What is the shape of C and the value of C[0,0]? [1 point]

```
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?

(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[3])

(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

[3]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 Resampling [10 Points]

Before moving to convolution, we will take a look at resampling operations which will make convolutions with strides ! = 1 more intuitive.

4.1 Upsampling1d

Upsamping as the name suggests is used to increase the size of input by a simple operation of adding intermediate 0s. The figure 5 shows how Upsampling is done. The output will be of size

```
input_size_upsampled = input_size * k - (k - 1)
```

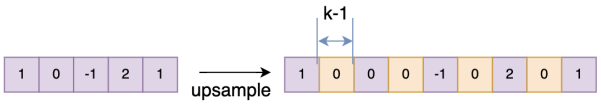


Figure 5: Upsampling1d Forward Example

where k is the *upsampling_factor*. For the given example, k = 2, so that only one zero is inserted between the elemenets. The example here is also for a single channel input. The backward of Upsampling is the equivalent to dropping the intermediate elements (which is same as Downsampling1d).

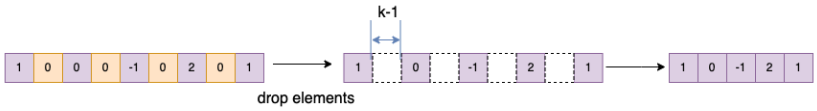


Figure 6: Upsampling1d Backward Example

In this section, your task is to implement the forward and backward attribute functions of `Upsample1d` class. The Please consider the following class structure.

```
class Upsample1d:

    def __init__(self, upsampling_factor):

        self.upsampling_factor = upsampling_factor

    def forward(self, ):

        Z = # TODO

        return Z

    def backward(self, dLdZ):

        dLdA = # TODO

        return dLdA
```

As you can see, the `Upsample1d` class has initialization, forward, and backward attribute functions. Immediately once the class is instantiated, the code in `__init__` is run. The initialization phase using `__init__` includes just the `upsampling_factor` as an argument.
In forward, we calculate `Z`. The attribute function `forward` include:

- As an argument, forward expects `A` as input.
- As an attribute, forward stores no variables.
- As an output, forward returns variable `Z`

In backward, we calculate gradient changes and store values needed for optimization. The attribute function `backward` includes:

- As arguments, backward expects inputs `dLdZ`.
- As attributes, backward stores no variables.
- As an output, backward returns `dLdA`.

Table 1: Upsample1d Class Components

Code Name	**Math**	**Type**	**Shape**	**Meaning**
`upsampling_factor`	$upsampling\_factor$	scalar	-	upsampling factor
`A`	A	matrix	$N \times C \times W_0$	pre-upsampling values
`Z`	Z	matrix	$N \times C \times W_1$	post-upsampling features
`dLdZ`	$\partial L/\partial Z$	matrix	$N \times C \times W_1$	gradient of Loss wrt Z
`dLdA`	$\partial L/\partial A$	matrix	$N \times C \times W_0$	gradient of Loss wrt A

where N is the batch size, C is the number of channels and W is the respective widths.

4.2 Downsampling1d

Downsampling1d is just "dropping" off elements with a factor k.

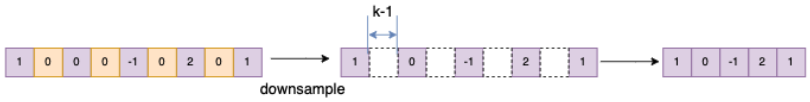


Figure 7: Downsampling1d Forward Example

For the given example, k = 2.

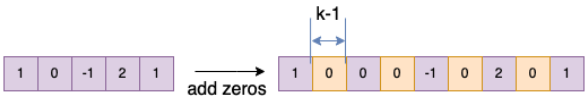


Figure 8: Downsampling1d Backward Example

As you may have figured it out, Upsampling and Downsampling are inverse operations of one another. Upsampling forward is downsampling backward and vice-versa.

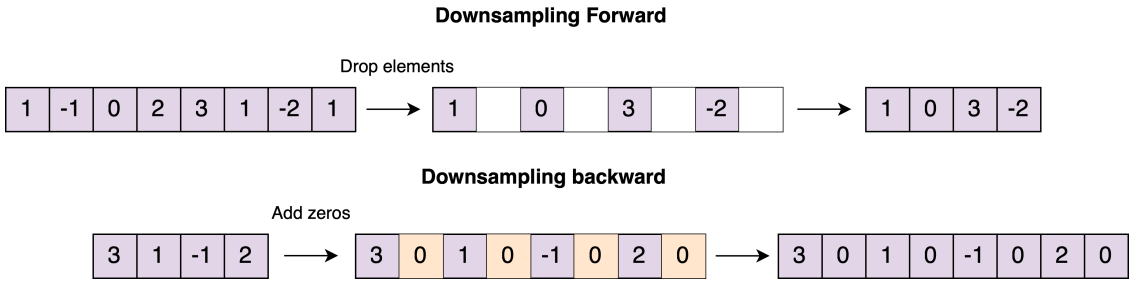


Figure 9: Downsampling1d Even Example (k=2)

In downsample, one thing to note is that the sizes may not match during backward. **We need the gradient of input to be the same size as the input in backward.** Think about what happens when the input size is even/odd and what you might want to add.

In this section,your task is to implement the forward and backward attribute functions of `Downsample1d` class. The Please consider the following class structure.

```
class Downsample1d:

    def __init__(self, stride):

        self.downsampling_factor = downsampling_factor

    def forward(self, A):

        Z = # TODO

        return Z

    def backward(self, dLdZ):

        dLdA = # TODO

        return dLdA
```

As you can see, the `Downsample1d` class has initialization, forward, and backward attribute functions. Immediately once the class is instantiated, the code in `__init__` is run. The initialization phase using `__init__` includes just the `downsampling_factor` as an argument.
In forward, we calculate `Z`. The attribute function `forward` include:

- As an argument, forward expects `A` as input.
- As an attribute, forward stores no attributes.
- As an output, forward returns variable `Z`

In backward, we calculate gradient changes and store values needed for optimization. The attribute function `backward` includes:

- As arguments, backward expects inputs `dLdZ`.
- As attributes, backward stores no variables.
- As an output, backward returns `dLdA`.

where W_0 is the size before downsampling while W_1 is the shape after downsampling.

Table 2: Downsample1d Class Components

Code Name	Math	Type	Shape	Meaning
`downsampling_factor`	$downsampling\_factor$	scalar	-	downsampling factor
`A`	A	matrix	$N \times C \times W_0$	pre-downsampling features
`Z`	Z	matrix	$N \times C \times W_1$	post-downsampling features
`dLdZ`	$\partial L/\partial Z$	matrix	$N \times C \times W_1$	gradient of Loss wrt Z
`dLdA`	$\partial L/\partial A$	matrix	$N \times C \times W_0$	gradient of Loss wrt A

4.3 Upsampling2d

2d upsampling is used for inputs like images where upsampling is performed in both the x and y direction. Upsampling an image is a simple operation where, zeros are added inbetween pixels of the input map. The number of zeros equals to the sampling rate k - 1. (Some may know this as dilation). The following diagram gives an intuitive explanation.

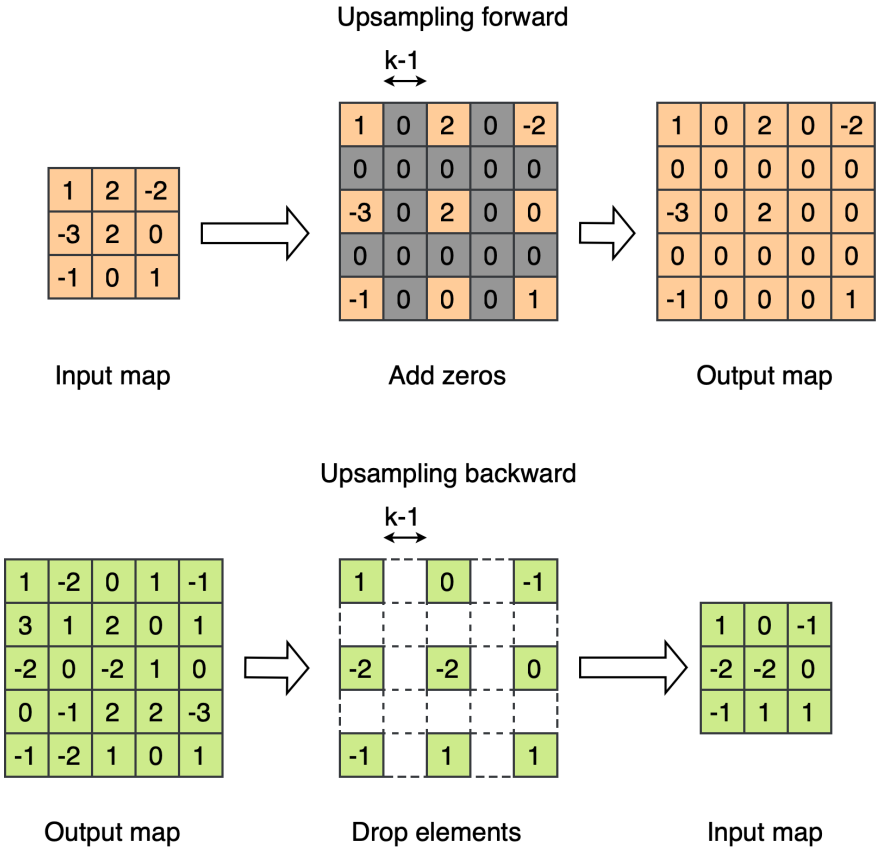


Figure 10: Upsampling 2d

In the forward pass of Upsampling2d, the size of input will be:

```
input_size_dilated = input_size * k - (k - 1)
```

Adjust factor k and set correct output size, or you may need to crop output map to ensure it is the right size. In the above example, k = 2.

The exact opposite takes place in backward where intermediate k-1 elements are dropped. In this section, you will implement forward and backward of the `Upsample2d` class. Please consider the following class structure.

```
class Upsample2d:

    def __init__(self, upsampling_factor):

        self.upsampling_factor = upsampling_factor

    def forward(self, A):

        Z = # TODO

        return Z

    def backward(self, dLdZ):

        dLdA = # TODO

        return dLdA
```

As you can see, the `Upsample2d` class has initialization, forward, and backward attribute functions. Immediately once the class is instantiated, the code in `__init__` is run. The initialization phase using `__init__` includes just the `upsampling_factor` as an argument.
In forward, we calculate `Z`. The attribute function `forward` include:

- As an argument, forward expects `A` as input.
- As an attribute, forward stores no attributes.
- As an output, forward returns variable `Z`

In backward, we calculate gradient changes and store values needed for optimization. The attribute function `backward` includes:

- As arguments, backward expects inputs `dLdZ`.
- As attributes, backward stores no variables.
- As an output, backward returns `dLdA`.

Table 3: Upsample2d Class Components

Code Name	**Math**	**Type**	**Shape**	**Meaning**
`upsampling_factor`	$upsampling\_factor$	scalar	-	upsampling factor
`A`	A	matrix	$N \times C \times H_0 \times W_0$	pre-upsampling features
`Z`	Z	matrix	$N \times C \times H_1 \times W_1$	post-upsampling features
`dLdZ`	$\partial L / \partial Z$	matrix	$N \times C \times H_1 \times W_1$	gradient of Loss wrt Z
`dLdA`	$\partial L / \partial A$	matrix	$N \times C \times H_0 \times W_0$	gradient of Loss wrt A

where H_0, W_0 are the sizes before upsampling while H_1, W_1 are the sizes after upsampling.

4.4 Downsampling2d

In downsampling, the input features are reduced. Downsampling forward is exactly similar to upsampling backward and vise versa. The following diagram gives an intuitive explanation.

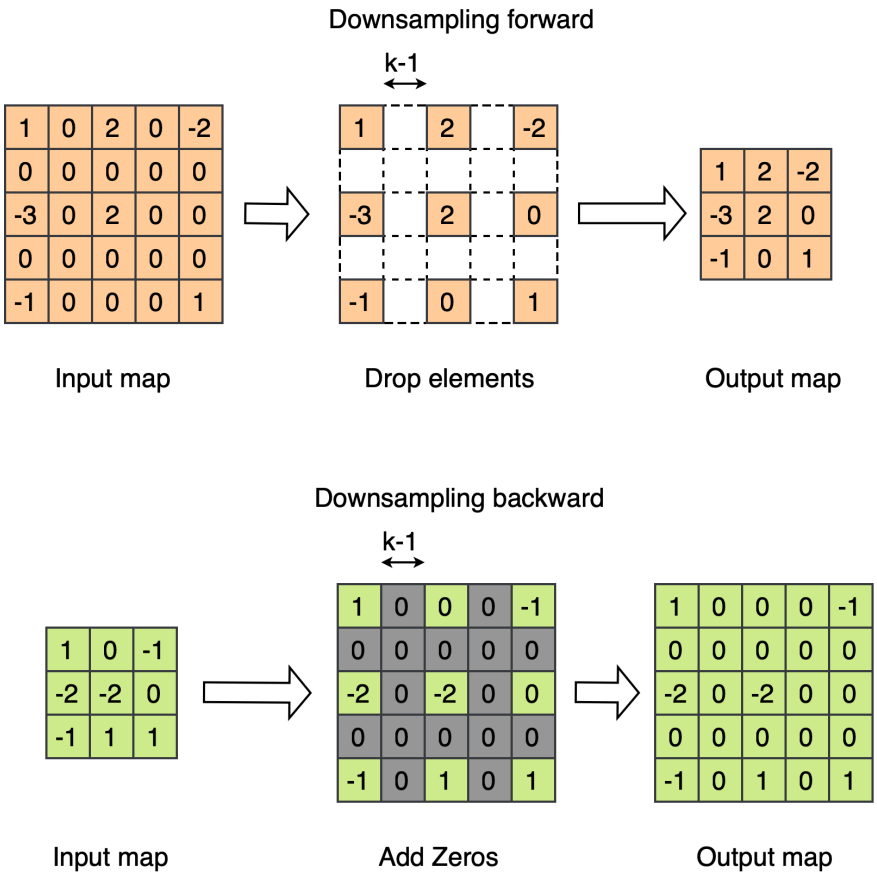


Figure 11: Downsampling 2d

In the forward pass of Downsampling2d, the size of input will be:

```
input_size_downsampled = (input_size - 1) // k + 1
```

Adjust factor k and set correct output size, or you may need to pad the output map to ensure it is the right size. In the above example, k = 2.
In this section, you will implement forward and backward of the `Downsample2d` class. Please consider the following class structure.

```
class Downsample2d:

    def __init__(self, downsampling_factor):

        self.downsampling_factor = downsampling_factor

    def forward(self, A):

        Z = # TODO

        return Z

    def backward(self, dLdZ):

        dLdA = # TODO
```

```
        return dLdA
```

As you can see, the `Downsample2d` class has initialization, forward, and backward attribute functions. Immediately once the class is instantiated, the code in `__init__` is run. The initialization phase using `__init__` includes just the `downsampling_factor` as an argument.
In forward, we calculate `Z`. The attribute function `forward` include:

- As an argument, forward expects `A` as input.
- As an attribute, forward stores no attributes.
- As an output, forward returns variable `Z`

In backward, we calculate gradient changes and store values needed for optimization. The attribute function `backward` includes:

- As arguments, backward expects inputs `dLdZ`.
- As attributes, backward stores no variables.
- As an output, backward returns `dLdA`.

Table 4: Downsample2d Class Components

Code Name	**Math**	**Type**	**Shape**	**Meaning**
`downsampling_factor`	$downsampling\_factor$	scalar	-	downsampling factor
`A`	A	matrix	$N \times C \times H_0 \times W_0$	pre-downsampling features
`Z`	Z	matrix	$N \times C \times H_1 \times W_1$	post-downsampling features
`dLdZ`	$\partial L / \partial Z$	matrix	$N \times C \times H_1 \times W_1$	gradient of Loss wrt Z
`dLdA`	$\partial L / \partial A$	matrix	$N \times C \times H_0 \times W_0$	gradient of Loss wrt A

where H_0, W_0 are the sizes before downsampling while H_1, W_1 is the shape after downsampling.

5 Convolutional Neural Networks

Congrats on completing the Resampling layers. Now come the interesting portions of this homework. Convolutional Neural Networks are one of the most widespread model architectures which are currently in use. From its inception, this model has undergone various transformations in a non-exhausive range of applications such as Image Classification, Image Segmentation, Object Detection, Speech Classification and so on. We will look into 1d and 2d convolutions and how to implement them from scratch.
Apart from previous editions of the homework, we will be using a different approach for CNNs. For strides greater than 1, we will be using the Resampling and Convolutional layers sequentially. You will get a clear idea about this after completion.
In this section, your task is to implement CNNs using only the NumPy library. 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**.

5.1 Convolutional layer : Conv1d [15 points]

Convolution 1d involves convolving the input with a kernel in just 1 direction. An example illustrating the same is presented in figure 12. A single channel input with 7 features is convolved with a single channel filter with kernel_size = 3. Convolution is basically an element wise multiplication and summation. As shown, when the filter scans through the input, at each step, there is an element wise multiplication between the patch of input elements (where the filter is on top of) and the filter elements. The output for a single convolutional step is a single scalar (orange). The filter can also take bigger steps for consecutive convolution steps. In the current example, the step/stride = 1 which is the number of pixels the filter passes after each convolution. There is also a bias which is added to the output (broadcast addition). It is a single scalar

per output channel that is added to all the elements of that channel. The size of the output is given by the formula:

```
output_size = [(input_size - kernel_size)//stride] + 1
```

Figure 15 explains multi-channel convolutions. The input has 3 channels with 5 features and the kernel has 3 channels (same as input) with kernel_size = 3. Similar to single channel, the filter convolves the input and performs an element-wise multiplication and addition. It should be noted that in the multi-channel case, **output of element wise multiplication and addition from all the 3 channels are added to produce a single scalar for a convolution step.** It can be observed that convolution of a single filter produces a single channel output. If we use N filters, we get N different outputs, producing an N channel output. Therefore, for a convolution with N filters, there would be N real numbers as bias for N output channels. We recommend you to understand the process and try the convolutions by hand before proceeding to code.

Convolution backward is almost exactly the same operation done in the reverse order. Consider the figure shown in 13 for single channel backward and 16 for multichannel backward. For the backward operation, we use the gradient of loss wrt to the output of convolution `dLdZ`. With this, we need to find the the gradient of the loss wrt to the kernels `dLdW` and gradient of loss wrt to the bias `dLdb`.

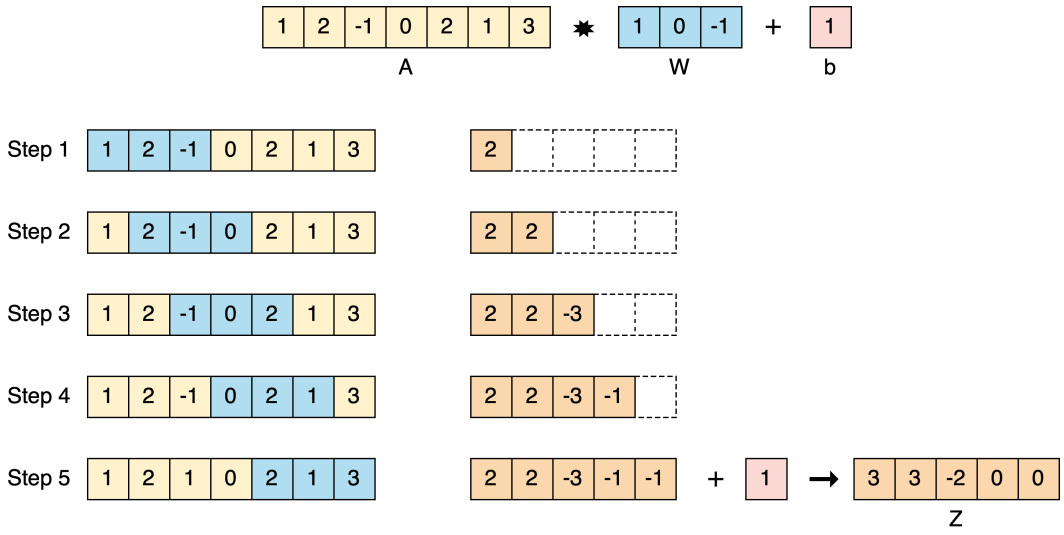


Figure 12: Conv1d with single channel input

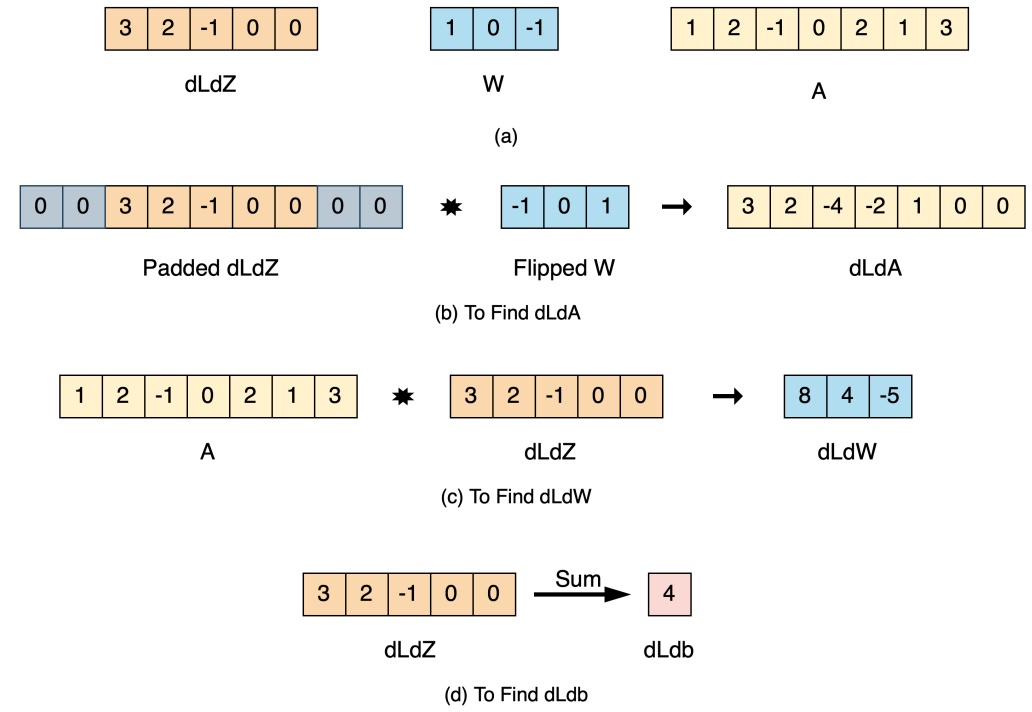


Figure 13: Conv1d backward with single channel input

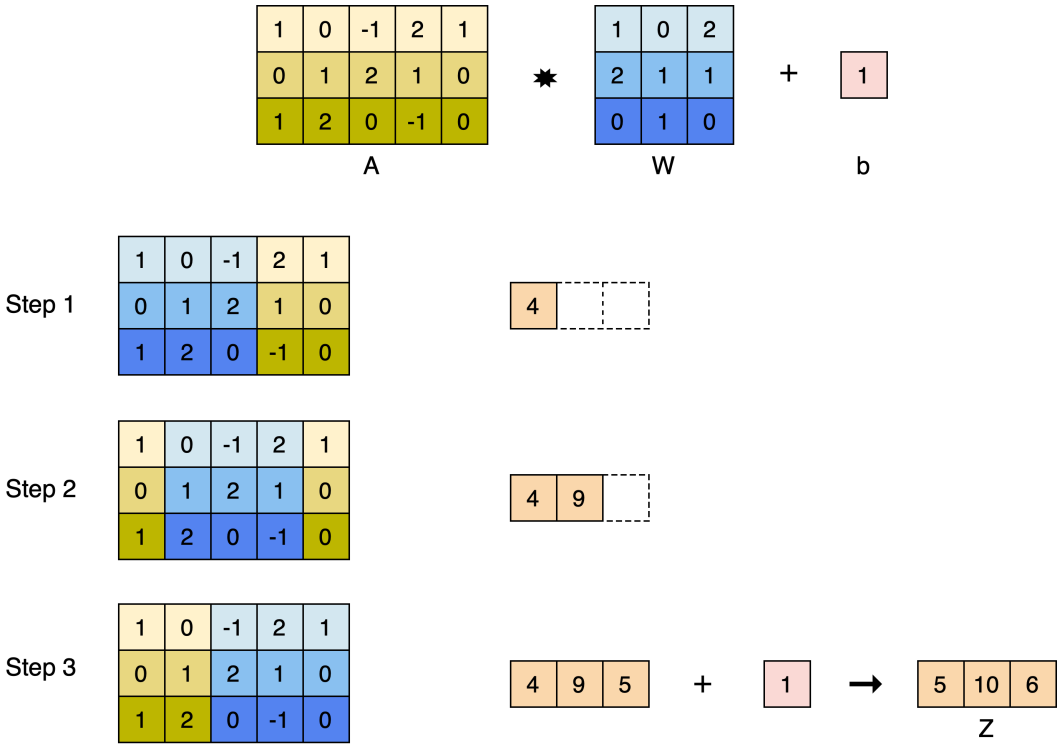


Figure 14: Conv1d with multichannel input

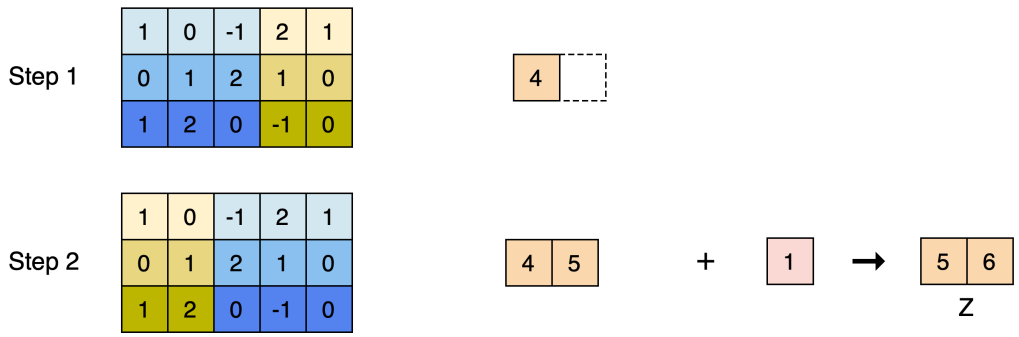


Figure 15: Conv1d with multichannel input and stride 2

Finding dLdb: As you know from the forward, bias is a single scalar for each output channel. Hence, we get `dLdb` by just summing the elements of `dLdZ`, channel wise to produce a vector of shape equal to the number of output channels.
Finding dLdW: Simply convolve the dLdZ with the input map A to get dLdW. Since one output map is produced from one filter, we can obtain derivatives w.r.t. all filters by doing the same process with different output maps. In addition, we can get derivatives for multiple channels of the same filter by convoluving dLdZ with the corresponding channel in input.

Finding dLdA:

- Broadcast `dLdZ` M times as shown (M being the number of channels of the input/kernel) This is done because, no matter how many channels the input has, a single filter produces only one output channel in forward. For single channel case 13, it is not required as M = 1.
- Pad this with kernel_size - 1 zeros on both sides (This is done as we would require the output to be larger than that of the input as convolution reduces the size in forward)
- Flip each channel of the filter left to right
- Convolve each flipped channel of the filter with the broadcasted and padded `dLdZ` to get `dLdA`

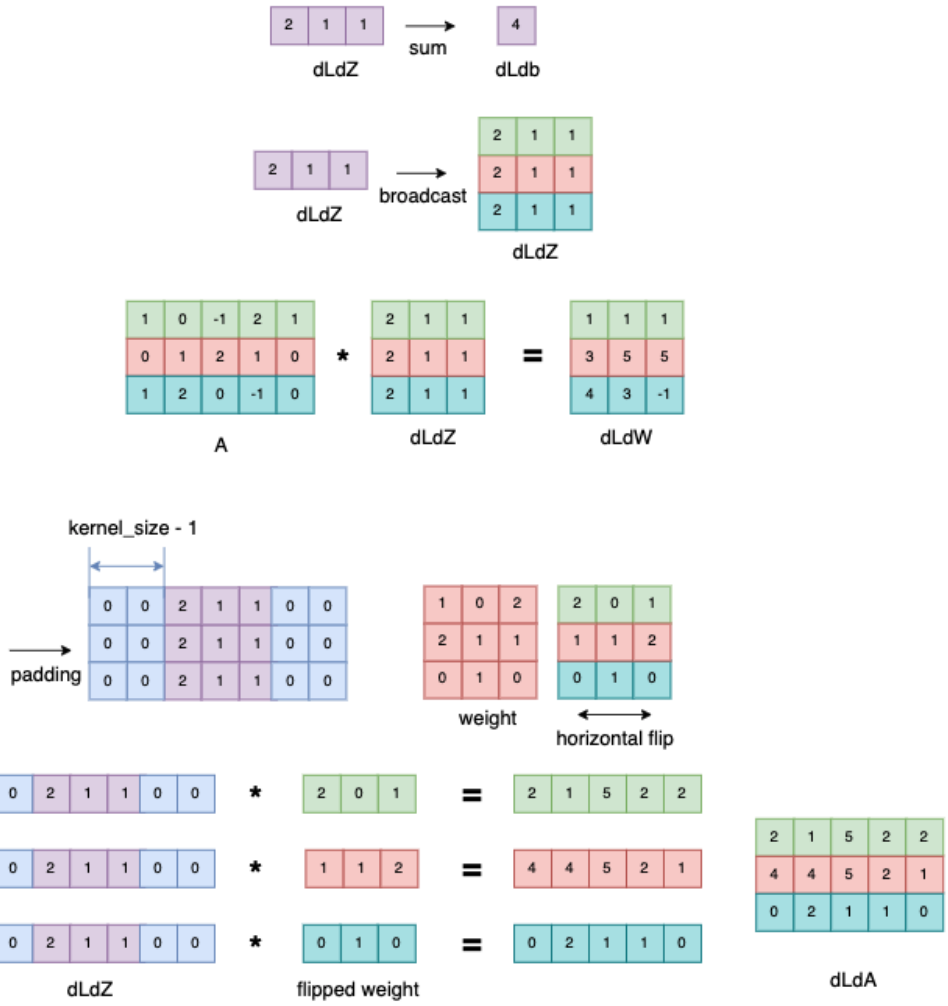


Figure 16: Conv1d_stride1 Backward Example

5.1.1 Conv1d_stride1

As you may have observed, the above example is for a stride of 1. Stride is the number of pixels by which the kernel moves at each step. In this section, you will implement forward and backward of the `Conv1d_stride1` class. Please consider the following class structure.

```
class Conv1d_stride1:

    def __init__(self, in_channels, out_channels, kernel_size, weight_init_fn, bias_init_fn):

    self.in_channels = in_channels
    self.out_channels = out_channels
    self.kernel_size = kernel_size

    self.W = weight_init_fn(out_channels, in_channels, kernel_size)
    self.b = bias_init_fn(out_channels)
```

```
        self.dLdW = np.zeros(self.W.shape)
        self.dLdb = np.zeros(self.b.shape)

        def forward(self, A):

            self.A = A
            Z = # TODO

            return Z

        def backward(self, dLdZ):
            self.dLdW = #TODO
            self.dLdb = #TODO

            dLdA = # TODO

            return dLdA
```

As you can see, the `Conv1d_stride1` class has initialization, forward, and backward attribute functions. Immediately once the class is instantiated, the code in `__init__` is run. The initialization phase using `__init__` includes:

- As arguments, Conv1d_stride1 will be specified using `in_channel`, `out_channel`, `kernel_size`, and `padding`. They are all positive integers.
- As attributes, `Conv1d_stride1` will be initialized with `W`, `dLdW`, `b`, and `dLdb`.

In forward, we calculate `Z` and store values needed for backward. The attribute function `forward` includes:

- As an argument, forward expects input `A`.
- As an attribute, forward stores variable `A`
- As an output, forward returns variable `Z`.

In backward, we calculate multiple gradient changes and store values needed for optimization. The attribute function `backward` includes:

- As an argument, backward expects input `dLdZ`.
- As attributes, backward stores variables `dLdW` and `dLdb`.
- As an output, backward returns variable `dLdA`.

Table 5: Conv1d_stride1 Class Components

Code Name	**Math**	**Type**	**Shape**	**Meaning**
`A`	A	matrix	$N \times C_0 \times W_0$	data input for convolution
`Z`	Z	matrix	$N \times C_1 \times W_1$	features after conv1d with stride 1
`W`	W	matrix	$C_1 \times C_0 \times K$	weight parameters
`b`	b	matrix	$C_1 \times 1$	bias parameters
`dLdZ`	$\partial L/\partial Z$	matrix	$N \times C_1 \times W_1$	how changes in outputs affect loss
`dLdA`	$\partial L/\partial A$	matrix	$N \times C_0 \times W_0$	how changes in inputs affect loss
`dLdW`	$\partial L/\partial W$	matrix	$C_1 \times C_0 \times K$	how changes in weights affect loss
`dLdb`	$\partial L/\partial b$	matrix	$C_1 \times 1$	how changes in bias affect loss

where W_0 is the size before convolution while W_1 are the size after convolution.

5.1.2 Conv1d

In this section, we would be implementing `Conv1d` which works for any stride > 1. We will reuse the code from conv1d_stride1 5.1.1 implementation to make it work for any stride > 1.
Specifically, a conv1d_stride1 layer followed by a downsample1d layer with factor k=2 is equivalent to a conv1d_stride2. More generally, a conv1d_stride1 layer followed by a downsample1d layer with factor k is equivalent to a conv1d with stride k.

$$\text{conv1d (stride=1)} \xrightarrow[\text{k}]{\text{downsample1d}} \text{conv1d (stride=k)}$$

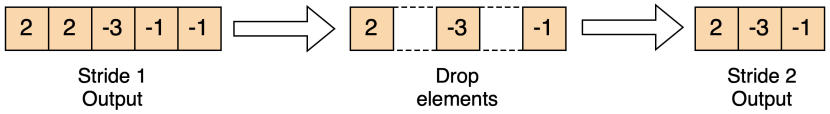


Figure 17: Convolution with stride 2 as convolution with stride 1 + downsampling

That's it! Now we can implement a conv1d operation for any value of stride with using just a combination of conv1d_stride1 and downsample layer with factor k (= stride).

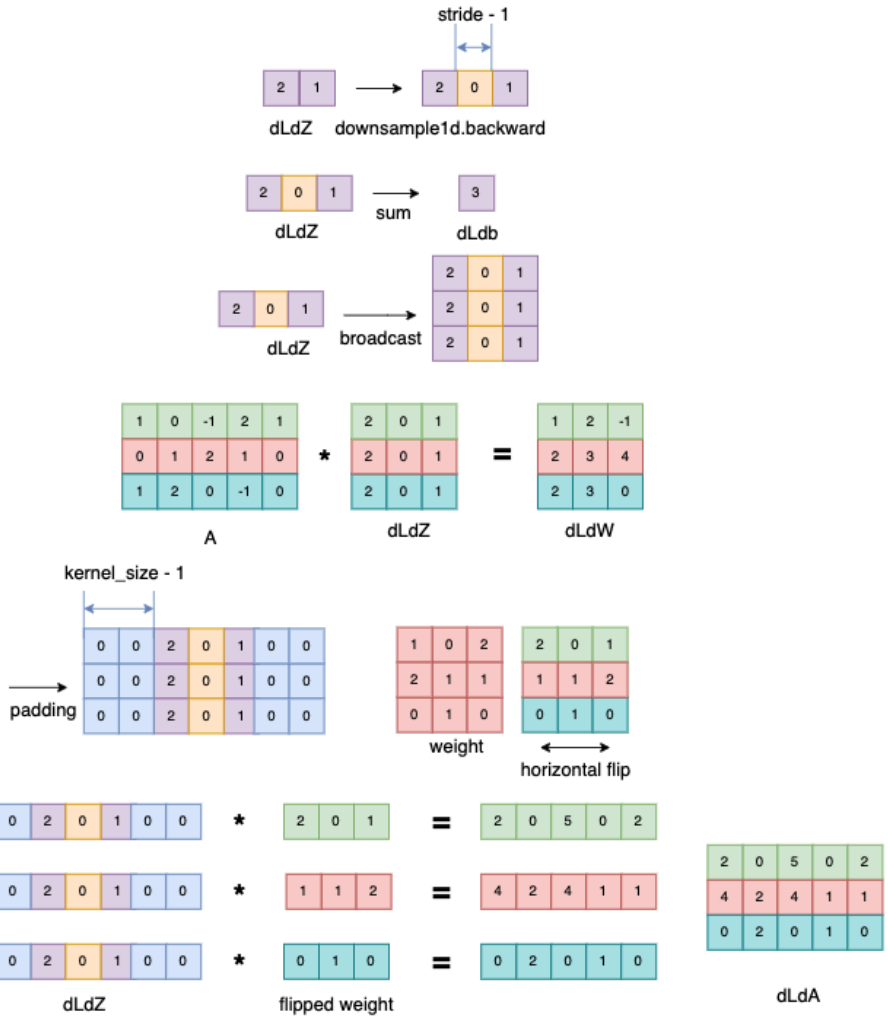


Figure 18: Conv1d Backward Example with stride 2

Your task is to implement forward and backward of the `Conv1d` class. Please consider the following class structure.

```
class Conv1d:

    def __init__(self, in_channels, out_channels, kernel_size, stride,
    weight_init_fn, bias_init_fn):

    self.stride = stride
    self.conv1d_stride1 = # TODO
    self.downsample1d = # TODO

    def forward(self, A):

        Z = # TODO (<= 2 lines of code)
        # Line 1: Conv1d forward
        # Line 2: Downsample1d forward
```

```
        return Z

    def backward(self, dLdZ):

        dLdA = # TODO (<= 2 lines of code)
        # Line 1: Downsample1d backward
        # Line 2: Conv1d backward

        return dLdA
```

As you can see, the `Conv1d` class has initialization, forward, and backward attribute functions. Immediately once the class is instantiated, the code in `__init__` is run. The initialization phase using `__init__` includes:

- As arguments, Conv1d will be specified using `in_channels`, `out_channels`, `kernel_size`, `stride`, `weight_init_fn` and `bias_init_fn`. They are all positive integers.
- As attributes, `Conv1d` will be initialized with `stride`, `Conv1d_stride1`, and `Downsample1d`.

In forward, we calculate `Z` and store values needed for backward. The attribute function `forward` includes:

- As an argument, forward expects input `A`.
- As an attribute, forward stores no variables.
- As an output, forward returns variable `Z`.

The attribute function `backward` includes:

- As an argument, backward expects input `dLdZ`.
- As attributes, `Conv1d` backward stores no variables.
- As an output, backward returns variable `dLdA`.

Table 6: Conv1d Class Components

Code Name	**Math**	**Type**	**Shape**	**Meaning**
`stride`	*stride*	scalar	-	downsampling factor
`A`	A	matrix	$N \times C_0 \times W_0$	data input for convolution
`Z`	Z	matrix	$N \times C_1 \times W_1$	features after conv1d with stride
`dLdZ`	$\partial L/\partial Z$	matrix	$N \times C_1 \times W_1$	how changes in outputs affect loss
`dLdA`	$\partial L/\partial A$	matrix	$N \times C_0 \times W_0$	how changes in inputs affect loss

where W_0 is the size before convolution while W_1 are the size after convolution.

5.2 Convolutional layer : Conv2d [15 points]

1Two dimensional convolution layers play a major role in today's image based intelligence tasks. From the inception of CNNs in the 80s by Yann LeCun, this model has undergone various transformations in a wide range of applications such as Image Classification, Image Segmentation, Object Detection and so on.
Let us understand 2d convolutions with an example. Figure 19 shows the convolution of a 3x3 kernel on a 5x5 input image. This example is for a single channel input.
Each step of convolution is depicted in the figure. Similar to the case of 1d, in a 2d convolution, an element wise multiplication of the filter with the image patch is done and then a sum is taken to produce a single output element (Image patch here means the portion of the image below the kernel). In step one, the filter starts from the top left corner of the image. After performing an element wise multiplication and a summation, we get the result to be 3 as shown in the output (blue colored). In step 2, the kernel moves towards the right by a distance of 1 pixel. Therefore, the stride of this convolution is 1. As the filter scans

through the image, the output map gets formed. It is intuitively observed that the position of the output element is influenced by where the kernel is placed on the image. The final step is to add a bias per channel for the output. We recommend to perform the convolution operation by hand to get a good grasp on the concept.

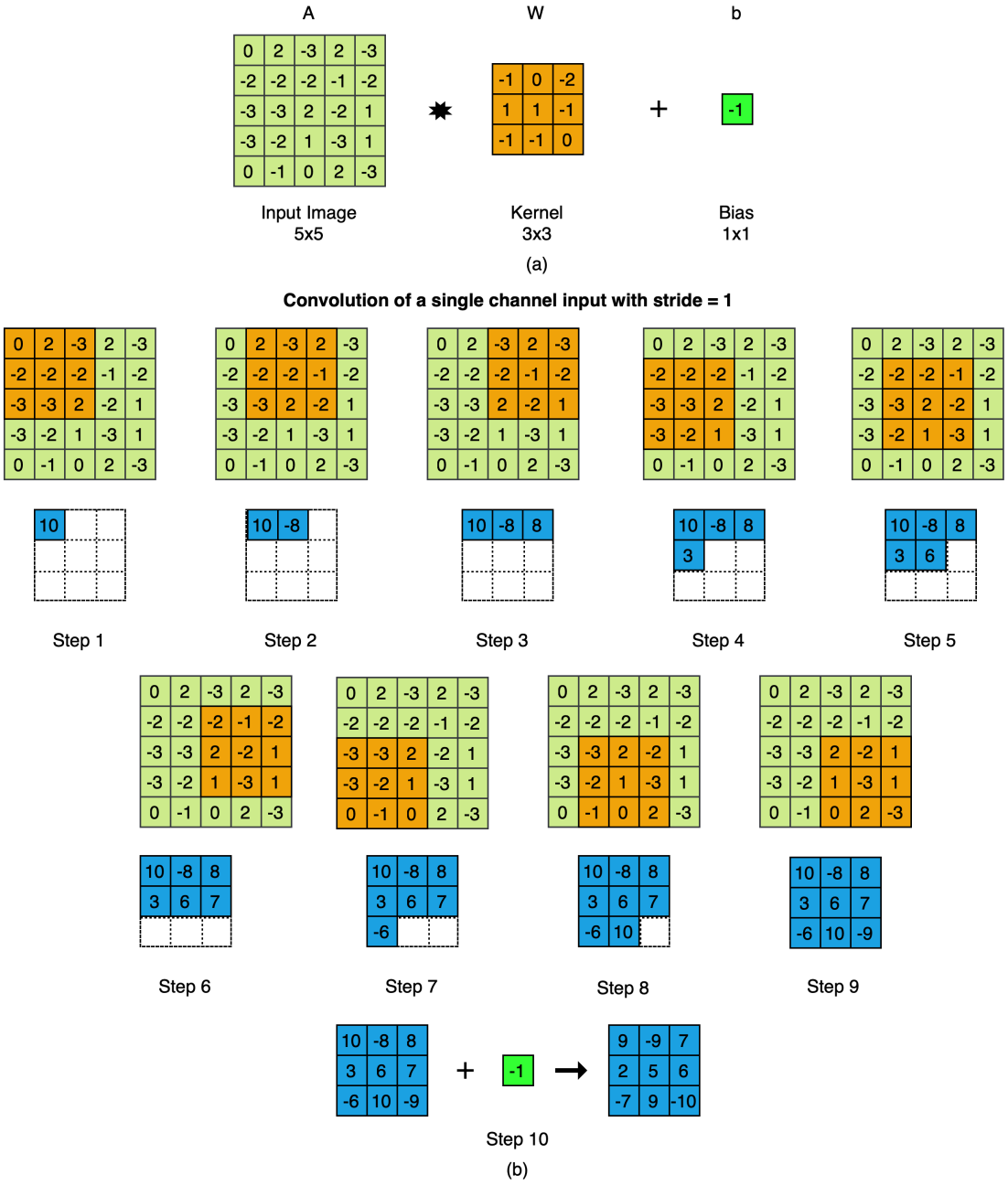


Figure 19: 2d Convolution Example stride 1

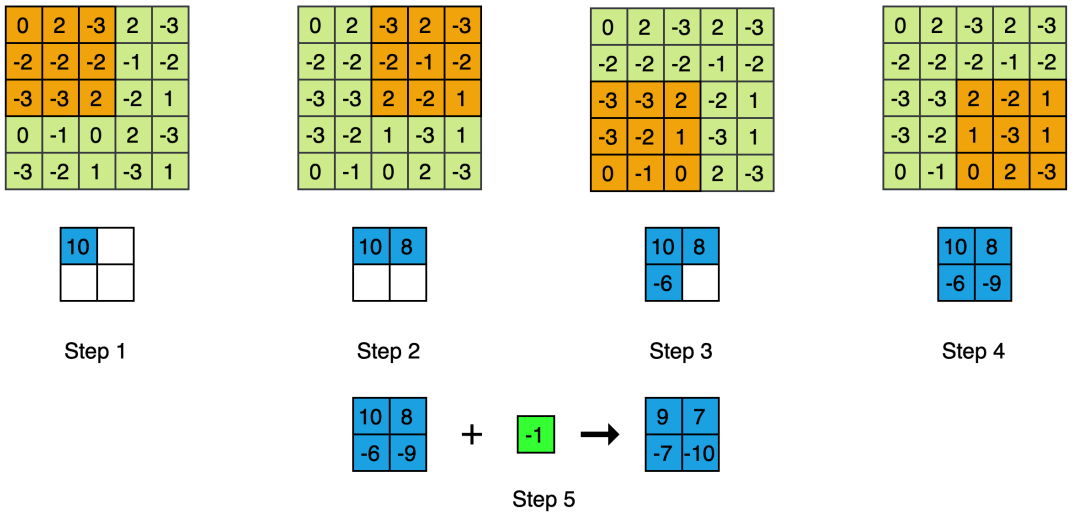


Figure 20: 2d Convolution Example stride 2

Figure 20 shows the steps relating to convolution with stride = 2. As you can see, the filter moves by 2 pixels left to right and top to bottom. With greater strides, the number of computations performed reduces and the size of input compared to stride = 1 is reduced.
Similar to 1d convolutions, the general formula for the size of output is given as:

```
output_size = [(input_size - kernel_size)//stride] + 1
```

We can extend the same idea stated above for a single channel image to a multi-channel image. Since a single channel filter convolves with a single channel image as explained previously, we would require a multi channel kernel to convolve a multi channel image. Therefore, the number of channels of the convolving kernel should be **equal** to the number of channels of the input image.

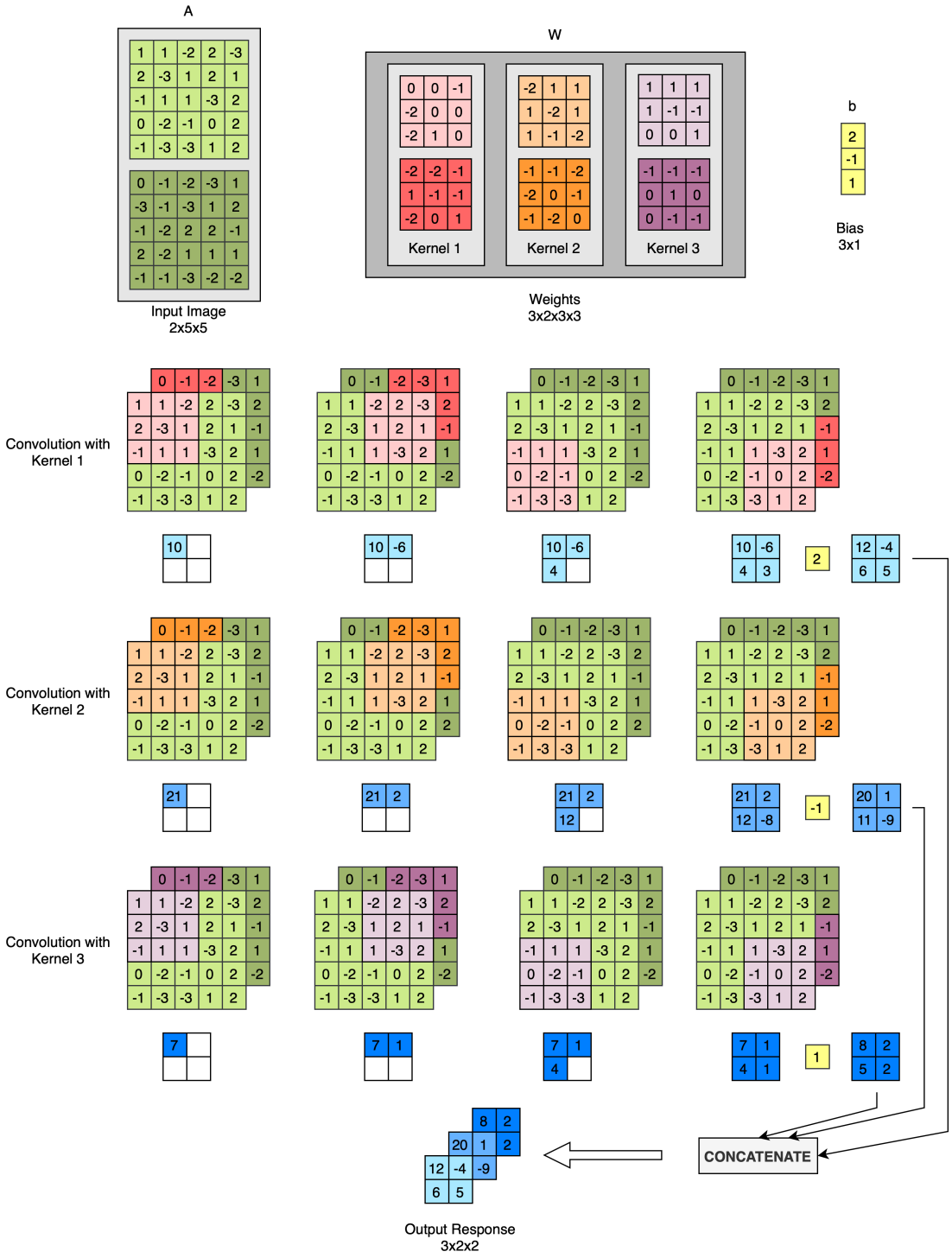


Figure 21: Multi channel convolution example

As shown in figure 21, A has 2 channels and so do all the kernels in W. Consider convolution with Kernel 1. Applying the same idea from the previous paragraphs, 2 input patches with 2 kernel channels would produce 2 output scalars. Then the outputs are summed to get a final single output scalar. The filter then takes 2 pixel steps (in this example stride = 2) and computes the same. After the 4th step, we get a single channel output. The take away is that, convoluving a filter with one input, produces a single channel output

irrespective of the number of input/kernel channels. When we use a different filter, we get a different output channel. That's what happens in convolution with Kernel 2 and Kernel 3. From 3 filters convolving with the input image, we get 3 output channels as shown. These channels are concatenated to produce a single 3 channel output image. The bias for this 3 channel output will be a 3x1 vector with each element as a bias for each channel.
The summary is that:

- `no. input channels = no. kernels channels`
- `no. output channels = no. kernels`

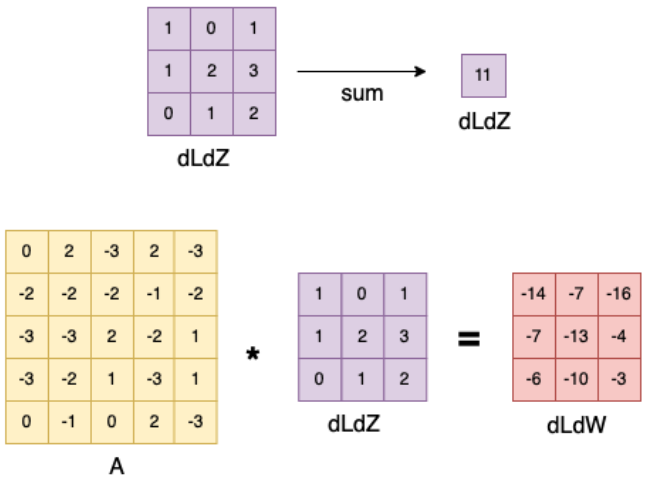
Now we will see how backward in convolution is performed. Backpropagation in 2d Convolution is not as hard at it sounds. Turns out, it employs that same process of convolution. Given an output map Z, in backprop, we get the gradient of the output map Z wrt the loss L `dLdZ` ($\nabla_Z L$) from the previous layers. This gradient map will be of the same shape as Z. With ($\nabla_Z L$), we need to find the gradient of the Loss wrt to weights ($\nabla_W L$), bias ($\nabla_b L$) and input ($\nabla_A L$).
The steps which are done to calculate ($\nabla_A L$) are as follows.

- Pad the `dLdZ` map with $kernel\_size - 1$ zeros as shown in Figure 22. We do this because, in forward, convolution reduces the size
- Flip the filter top to bottom and left to right.
- Convolve the flipped kernel over the padded ($\nabla_Z L$) to get ($\nabla_A L$) as the convolution output.

The above process gives us `self.dLdA` ($\nabla_A L$) for one input channel. To get the same for other input channels, we flip the filter channel corresponding to the input channel and convolve with the padded ($\nabla_Z L$).

To calculate `self.dLdW`: Simply convolve the ($\nabla_Z L$) with the input map A to get ($\nabla_W L$). Since one output map is produced from one filter, we can obtain gradients w.r.t. all filters by doing the same process with different output maps. In addition, we can get gradients for multiple channels of the same filter by convoluving ($\nabla_A L$) with the corresponding channel in input.

To calculate `self.dLdb`: As you know from the forward, bias is a single scalar for each output channel. Hence, we get `dLdb` by just summing the elements of `dLdZ` channelwise to produce a vector of shape equal to the number of output channels.
`self.dLdW` and `self.dLdb` 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`.



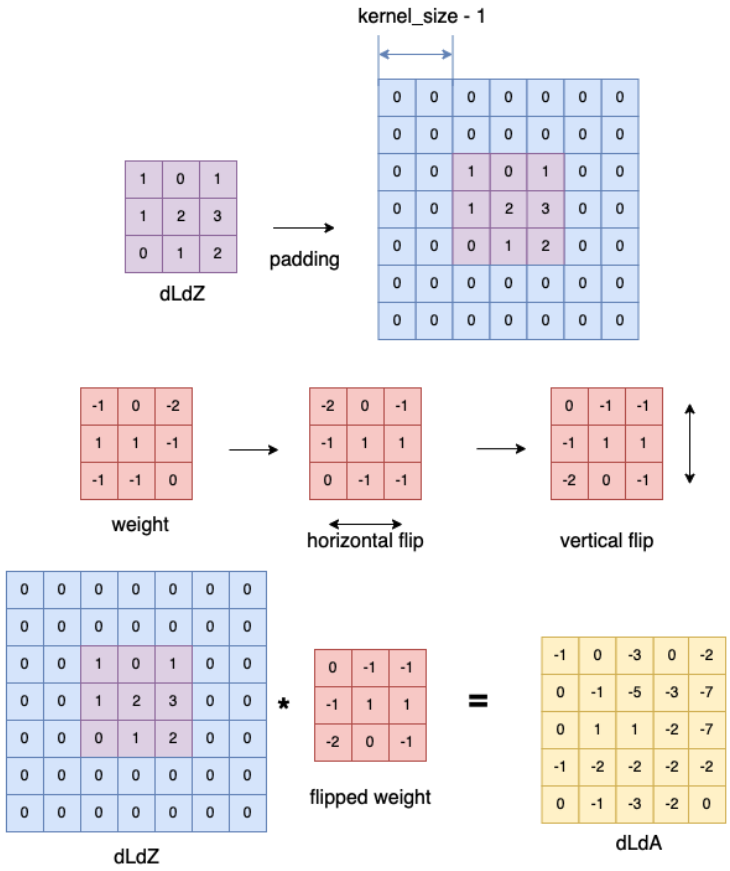


Figure 22: Conv2d_stride1 Backward: Convolution

5.2.1 Conv2d_stride1

As you may have observed, the above example is for a stride of 1. In this section, you will implement forward and backward of the `Conv2d_stride1` class. Make sure that you have understood it from the previous section before starting to code.
Please consider the following class structure.

```
class Conv2d_stride1:

    def __init__(self, in_channels, out_channels, kernel_size, weight_init_fn, bias_init_fn):

        self.in_channels = in_channels
        self.out_channels = out_channels
        self.kernel_size = kernel_size

        self.W = weight_init_fn(out_channels, in_channels, kernel_size)
        self.b = bias_init_fn(out_channels)

        self.dLdW = np.zeros(self.W.shape)
        self.dLdb = np.zeros(self.b.shape)

    def forward(self, A):

        self.A = A
```

```
        Z = # TODO

        return Z

    def backward(self, dLdZ):
        self.dLdW = #TODO
        self.dLdb = #TODO

        dLdA = # TODO

        return dLdA
```

As you can see, the `Conv2d_stride1` class has initialization, forward, and backward attribute functions. Immediately once the class is instantiated, the code in `__init__` is run. The initialization phase using `__init__` includes:

- As arguments, Conv2d_stride1 will be specified using `in_channel`, `out_channel`, `kernel_size`, and initialization functions.
- As attributes, `Conv2d_stride1` will be initialized with `W`, `dLdW`, `b`, and `dLdb`.

In forward, we calculate `Z` and store values needed for backward. The attribute function `forward` includes:

- As an argument, forward expects input `A`.
- As an attribute, forward stores variables `A`
- As an output, forward returns variable `Z`.

In backward, we calculate multiple gradient changes and store values needed for optimization. The attribute function `backward` includes:

- As an argument, backward expects input `dLdZ`.
- As attributes, backward stores variables `dLdW` and `dLdb`.
- As an output, backward returns variable `dLdA`.

Table 7: Conv2d_stride1 Class Components

Code Name	**Math**	**Type**	**Shape**	**Meaning**
`A`	A	matrix	$N \times C_0 \times H_0 \times W_0$	data input for convolution
`Z`	Z	matrix	$N \times C_1 \times H_1 \times W_1$	features after conv2d with stride 1
`W`	W	matrix	$C_1 \times C_0 \times K \times K$	weight parameters
`b`	b	matrix	$C_1 \times 1$	bias parameters
`dLdZ`	$\partial L/\partial Z$	matrix	$N \times C_1 \times H_1 \times W_1$	how changes in outputs affect loss
`dLdA`	$\partial L/\partial A$	matrix	$N \times C_0 \times H_0 \times W_0$	how changes in inputs affect loss
`dLdW`	$\partial L/\partial W$	matrix	$C_1 \times C_0 \times K \times K$	how changes in weights affect loss
`dLdb`	$\partial L/\partial b$	matrix	$C_1 \times 1$	how changes in bias affect loss

Table 7 shows the class components. H_0, W_0 are the shape before convolution while H_1, W_1 are the shape after convolution.

5.2.2 Conv2d

In this section, we would be implementing `Conv2d` which works for any stride > 1. We will reuse the code from conv1d_stride1 5.2.1 implementation to make it work for any stride > 1.
Specifically, a conv2d_stride1 layer followed by a downsample2d layer with factor k=2 is equivalent to a conv2d_stride2. More generally, a conv2d_stride1 layer followed by a downsample1d layer with factor k is

equivalent to a conv1d_stride with stride k. Figure 23 shows this.

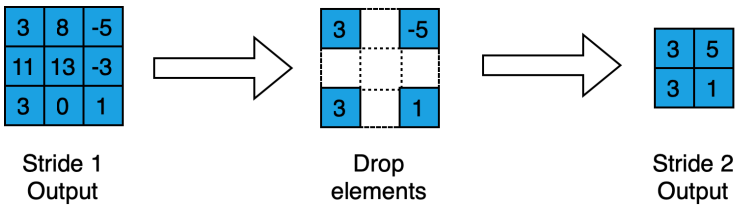


Figure 23: Stride 2 convolution as stride 1 convolution and downsampling with factor = 2

That's it! Now we can implement a conv2d operation for any value of stride with using just a combination of conv2d_stride1 and downsample layer with factor k (= stride).

For backward, the steps in forward are reversed in the same order. For stride > 1, downsampling backward is called as shown in Figure 24 and then convolution stride 1 backward is called as shown in Figure 25.

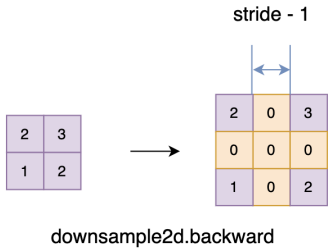
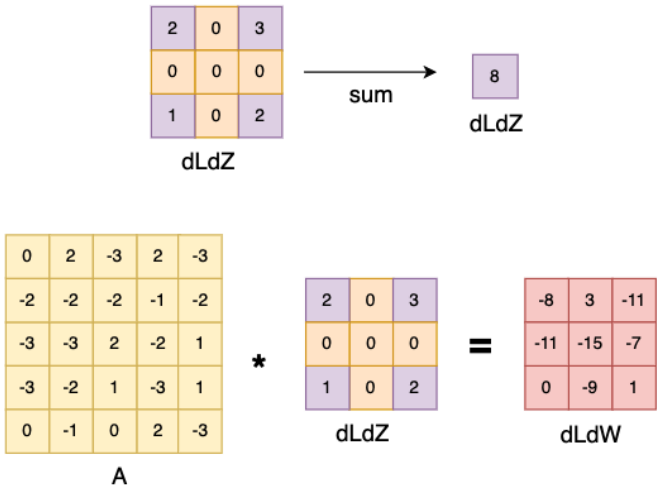


Figure 24: Downsample2d Backward: Conv2d Example



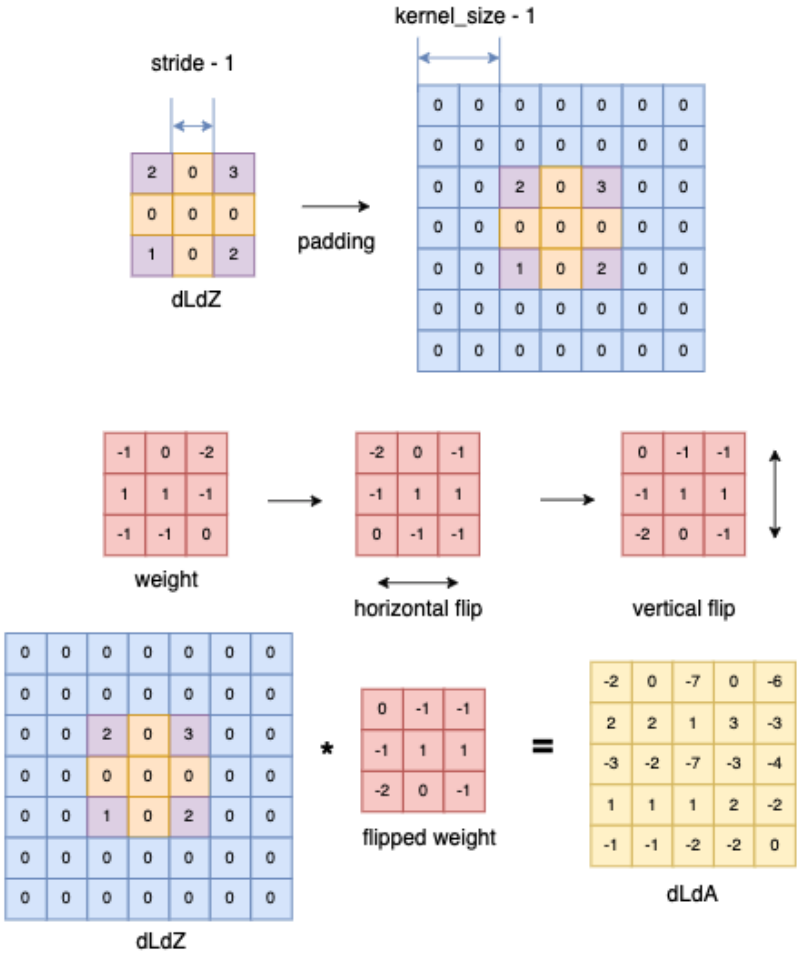


Figure 25: Conv2d Backward Example (stride¿1)

Make sure that you have understood that strided convolutions are just a combination of convolution with stride 1 and downsampling. In this section, your task is to implement the `forward` and backward attributes of the Conv2d class. Please consider the following class structure.

```
class Conv2d():
    def __init__(self, in_channels, out_channels, kernel_size, stride, weight_init_fn, bias_init_fn
        self.stride = stride
        self.conv2d_stride1 = None # TODO
        self.downsample2d = None # TODO

    def forward(self, A):
        # Call Conv2d_stride1 forward
        # TODO

        # Call downsample2d forward
        Z = # TODO

        return Z

    def backward(self, dLdZ):
```

```
        # Call downsample2d backward
        # TODO

        # Call Conv2d_stride1 backward
        dLdA = # TODO

        return dLdA
```

As you can see, the `Conv2d` class has initialization, forward, and backward attribute functions. Immediately once the class is instantiated, the code in `__init__` is run. The initialization phase using `__init__` includes:

- As arguments, `Conv2d` will be specified using `in_channels`, `out_channels`, `kernel_size` and `stride`. They are all positive integers.
- As attributes, `Conv2d` will be initialized with `Conv2d_stride1`, `Downsample2d`, `W`, `dLdW`, `b`, and `dLdb`.

In forward, we calculate `Z` and store values needed for backward. The attribute function `forward` includes:

- As an argument, forward expects input `A`.
- As an attribute, forward stores no variables.
- As an output, forward returns variable `Z`.

In backward, we calculate multiple gradient changes and store values needed for optimization. The attribute function `backward` includes:

- As an argument, backward expects input `dLdZ`.
- As attributes, backward stores no variables.
- As an output, backward returns variable `dLdA`.

Table 8: Conv2d Class Components

Code Name	**Math**	**Type**	**Shape**	**Meaning**
`stride`	*stride*	scalar	-	downsampling factor
`A`	A	matrix	$N \times C_0 \times H_0 \times W_0$	data input for convolution
`Z`	Z	matrix	$N \times C_1 \times H_1 \times W_1$	features after conv2d with stride
`dLdZ`	$\partial L / \partial Z$	matrix	$N \times C_1 \times H_1 \times W_1$	how changes in outputs affect loss
`dLdA`	$\partial L / \partial A$	matrix	$N \times C_0 \times H_0 \times W_0$	how changes in inputs affect loss

where H_0, W_0 are the shape before convolution while H_1, W_1 are the shape after convolution.

5.3 Transposed Convolution [10 points]

5.3.1 ConvTranspose1d

In regular convolutions, the affine value Z for a layer "pulls" input values A from the previous layer which causes in reduction of the output size. However, in an Transposed Convolution layer, the input values A are "pushed" to the next layer Z such that the output map's size is increased. The primary operation of ConvTransposed1d is to upsample the input and then convolve (with stride 1) to get the output. In your implementation, you will pass *upsampling_factor* to the ConvTransposed1d class, which is the same in definition of convolution with a fractional stride ($\frac{1}{upsampling\_factor}$). Please consider the Figure 26 for better intuition.

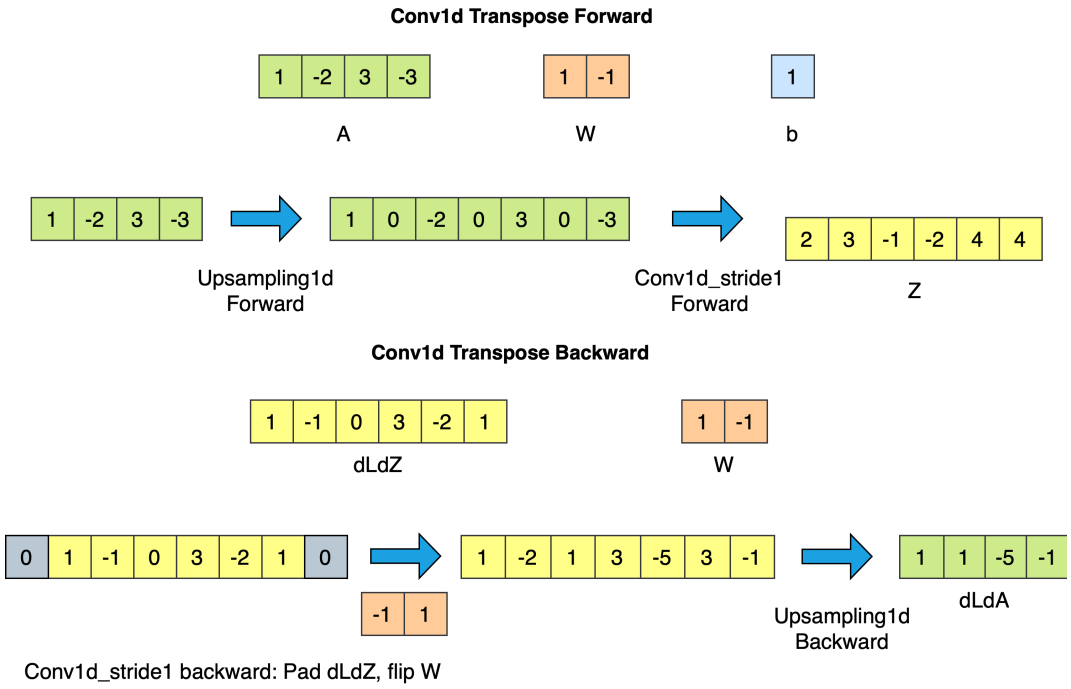


Figure 26: 1d Transpose Convolution

In this section, your task is to implement the `forward` and `backward` attributes of the ConvTransposed1d class. Please consider the following class structure.

```
class ConvTranspose1d():

    def __init__(self, in_channels, out_channels, kernel_size, upsampling_factor,
    weight_init_fn, bias_init_fn):

    self.upsampling_factor = upsampling_factor
    self.upsample1d = # TODO
    self.conv1d_stride1 = # TODO

    def forward(self, A):

        Z = # TODO (<= 2 lines of code)
        # Line 1: Upsampling1d forward
        # Line 2: Conv1d_stride1 forward

        return Z

    def backward(self, dLdZ):

        dLdA = # TODO (<= 2 lines of code)
        # Line 1: Conv1d_stride1 backward
        # Line 2: Upsampling1d backward

        return dLdA
```

As you can see, the `ConvTransposed1d` class has initialization, forward, and backward attribute functions. Immediately once the class is instantiated, the code in `__init__` is run. The initialization phase using `__init__` includes:

- As arguments, ConvTransposed1d will be specified using `in_channel`, `out_channel`, `kernel_size`, `upsampling_factor`, `weight_init_fn` and `bias_init_fn`. They are all positive integers.
- As attributes, `ConvTransposed1d` will be initialized with `upsampling_factor`, `conv1d_stride1`, and `Upsample1d`.

In forward, we calculate `Z` and store values needed for backward. The attribute function `forward` includes:

- As an argument, forward expects input `A`.
- As an attribute, forward stores no variables.
- As an output, forward returns variable `Z`.

The attribute function `backward` includes:

- As an argument, backward expects input `dLdZ`.
- As attributes, `ConvTransposed1d` backward stores no variables.
- As an output, backward returns variable `dLdA`.

Table 9: ConvTransposed1d Class Components

Code Name	**Math**	**Type**	**Shape**	**Meaning**
`upsampling_factor`	$upsampling\_factor$	scalar	-	upsampling factor
`A`	A	matrix	$N \times C_0 \times W_0$	data input for ConvTransposed1d
`Z`	Z	matrix	$N \times C_1 \times W_1$	features after ConvTransposed1d
`dLdZ`	$\partial L/\partial Z$	matrix	$N \times C_1 \times W_1$	how changes in outputs affect loss
`dLdA`	$\partial L/\partial A$	matrix	$N \times C_0 \times W_0$	how changes in inputs affect loss

where W_0 is the shape before convolution while W_1 is the shape after convolution.

5.3.2 ConvTranspose2d

Similar to ConvTranspose1d, ConvTranspose2d is a combination of Upsample2d and convolution with stride 1. Please consider the Figure 27 for better intuition. The example uses as convolution with stride = 1/3. The downsampling factor k = 3 in the forward example. In the backward example, we have used an example from stride = 1/2 convolution where the downsampling factor = 2.

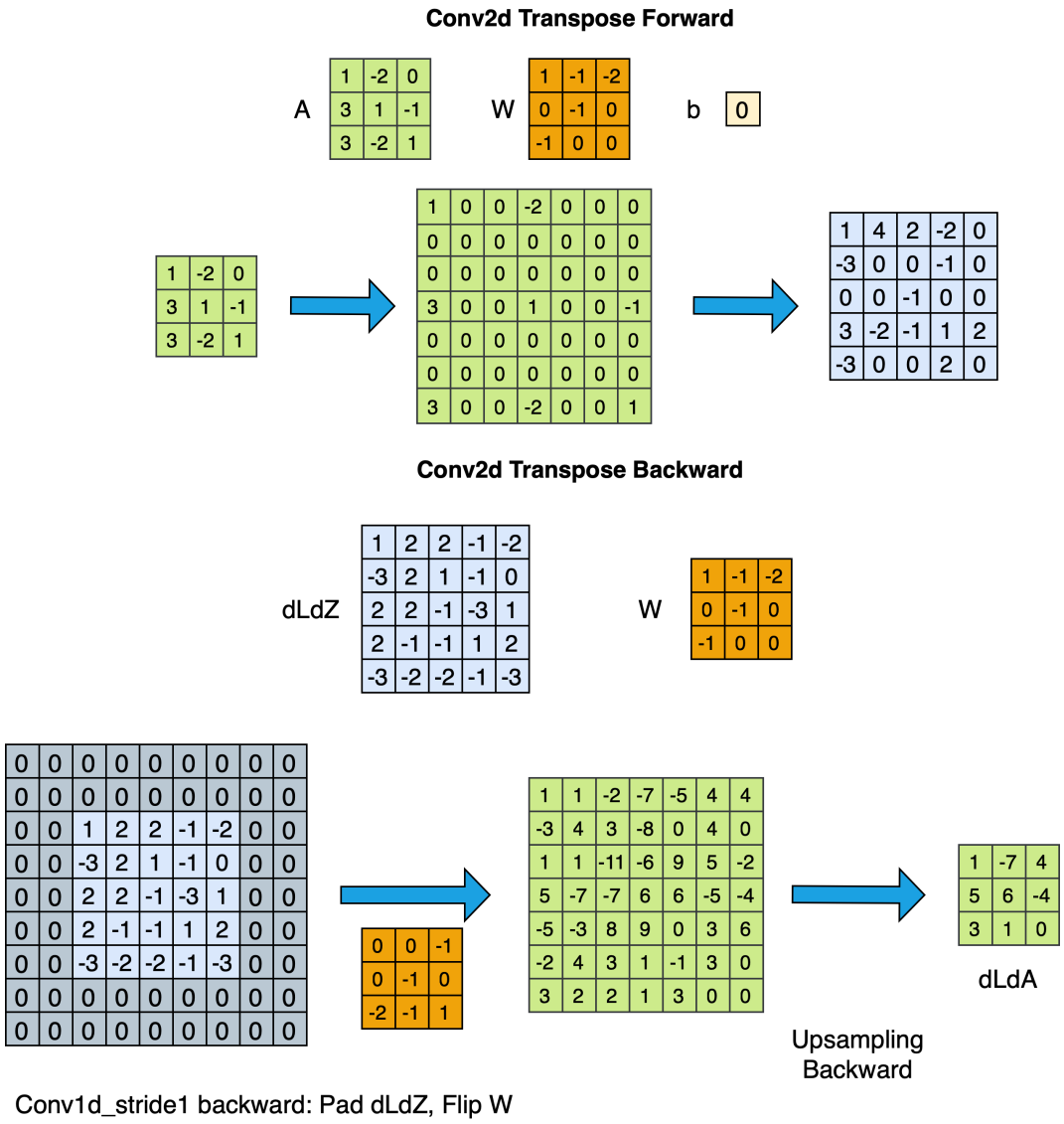


Figure 27: 2d Transpose Convolution

In your implementation, you are going to pass a *upsampling_factor* to the ConvTranspose2d class, which is the inverse of the stride(in this case, stride is fractional). Your task is to implement the `forward` and `backward` attributes of the ConvTranspose2d class. Please consider the following class structure.

In this section, your task is to implement the `forward` and `backward` attributes of the ConvTranspose2d class. Please consider the following class structure.

```
class ConvTranspose2d():

    def __init__(self, in_channels, out_channels, kernel_size, upsampling_factor,
    weight_init_fn, bias_init_fn):

    self.upsampling_factor = upsampling_factor
    self.upsample2d = # TODO
    self.conv2d_stride1 = # TODO
```

```
def forward(self, A):

    Z = # TODO (<= 2 lines of code)
    # Line 1: Upsampling2d forward
    # Line 2: Conv2d_stride1 forward

    return Z

def backward(self, dLdZ):

    dLdA = # TODO (<= 2 lines of code)
    # Line 1: Conv2d_stride1 backward
    # Line 2: Upsampling2d backward

    return dLdA
```

As you can see, the `ConvTranspose2d` class has initialization, forward, and backward attribute functions. Immediately once the class is instantiated, the code in `__init__` is run. The initialization phase using `__init__` includes:

- As arguments, ConvTranspose2d will be specified using `in_channel`, `out_channel`, `kernel_size`, `upsampling_factor`, `weight_init_fn` and `bias_init_fn`. They are all positive integers.
- As attributes, `ConvTranspose2d` will be initialized with `upsampling_factor`, `conv2d_stride1`, and `Upsample2d`.

In forward, we calculate `Z` and store values needed for backward. The attribute function `forward` includes:

- As an argument, forward expects input `A`.
- As an attribute, forward stores no variables.
- As an output, forward returns variable `Z`.

The attribute function `backward` includes:

- As an argument, backward expects input `dLdZ`.
- As attributes, `ConvTranspose2d` backward stores no variables.
- As an output, backward returns variable `dLdA`.

Table 10: ConvTranspose2d Class Components

Code Name	**Math**	**Type**	**Shape**	**Meaning**
`upsampling_factor`	$upsampling\_factor$	scalar	-	upsampling factor
`A`	A	matrix	$N \times C_0 \times H_0 \times W_0$	data input for ConvTranspose2d
`Z`	Z	matrix	$N \times C_1 \times H_1 \times W_1$	features after ConvTranspose2d
`dLdZ`	$\partial L / \partial Z$	matrix	$N \times C_1 \times H_1 \times W_1$	how changes in outputs affect loss
`dLdA`	$\partial L / \partial A$	matrix	$N \times C_0 \times H_0 \times W_0$	how changes in inputs affect loss

where H_0, W_0 are the sizes before transposed convolution while H_1, W_1 are the sizes after transposed convolution.

5.4 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. In forward, a multi-dimensional input is reshaped into a single dimensional output. In backward, a single dimensional input is reshaped into a multi-dimensional output.

Figure 28: Flatten Forward Example

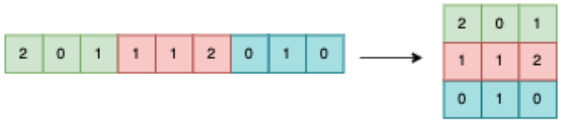


Figure 29: Flatten Backward Example

```
class Flatten():
    def forward(self, A):
        #TODO save shape of A
        Z = # TODO # reshape

    def backward(self, dLdZ):
        dLdA = # TODO # reshape
```

As you can see, the `Flatten` class has forward, and backward attribute functions. In forward, we calculate `Z` and store values needed for backward. The attribute function `forward` includes:

- As an argument, forward expects input `A`.
- As an attribute, forward stores the shape of input.
- As an output, forward returns variable `Z`.

The attribute function `backward` includes:

- As an argument, backward expects input `dLdZ`.
- As attributes, `Flatten` backward stores no variables.
- As an output, backward returns variable `dLdA`.

Table 11: Flatten Class Components

Code Name	Math	Type	Shape	Meaning
`A`	A	matrix	$N \times C_0 \times W_0$	data input for Flatten
`Z`	Z	matrix	$N \times (C_0 * W_0)$	features after Flatten
`dLdZ`	$\partial L / \partial Z$	matrix	$N \times (C_0 * W_0)$	how changes in outputs affect loss
`dLdA`	$\partial L / \partial A$	matrix	$N \times C_0 \times W_0$	how changes in inputs affect loss

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.

5.5 Pooling [30 Points]

Different from convolution, the operation in pooling is fixed and no parameters to learn.

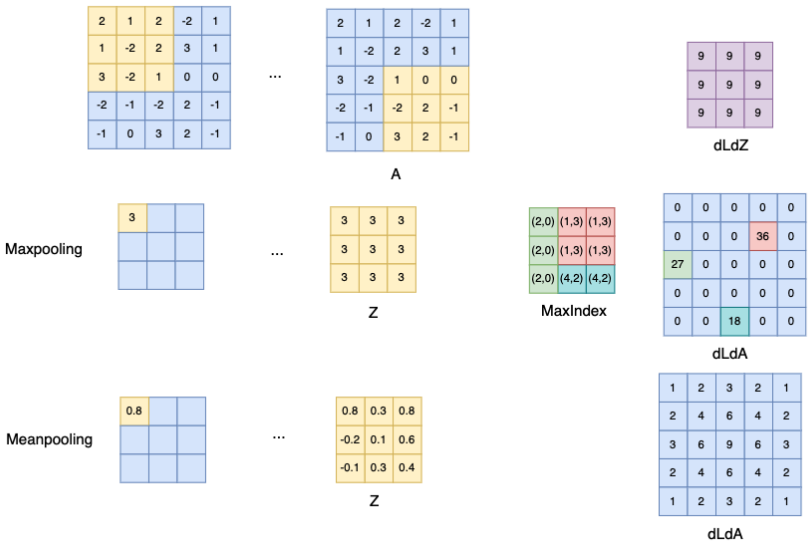


Figure 30: Pooling2d Example

Max pooling selects the largest from a pool of elements and is performed by "scanning" the input. Your task is to implement both forward propagation and backward propagation for 1d and 2d max pooling. In case you find two max values in a kernel patch, you can use the first occurrence (consider row major ordering) of max value as the output for that kernel patch. This also can be seen from Figure 30.

Mean pooling takes the arithmetic mean of elements and is performed by "scanning" the input. Your task is to implement both forward propagation and backward propagation for 1d and 2d mean pooling.

Similar to the previous layers, you will implement a stride 1 pooling and then a combination of pooling and downsampling for higher strides. The pooling operation itself is just a jitter invariant operation. Pooling typically uses a *stride* > 1, which is the same as convolution followed by downsampling. You can take maxpooling as a normal convolution with standard filter and max activation. As to the meanpooling, it can be viewed as a convolution with a special filter, where each element in the filter is $\frac{1}{filter\_size}$. Consider the class structure given below.

```
class MaxPool2d_stride1():

    def __init__(self, kernel):
        self.kernel = kernel

    def forward(self, A):
        Z = # TODO
        return Z

    def backward(self, dLdZ):
        dLdA = # TODO
```

```
        return dLdA

class MeanPool2d_stride1():

    def __init__(self, kernel):
        self.kernel = kernel

    def forward(self, A):
        Z = # TODO
        return Z

    def backward(self, dLdZ):
        dLdA = # TODO
        return dLdA
```

As you can see, each `Pool2d_stride1` class has forward, and backward attribute functions. In forward, we calculate `Z` and store values needed for backward. The attribute function `forward` includes:

- As an argument, forward expects input `A`.
- As an attribute, forward of the MaxPool2d_stride1 stores the index of the max values and the MeanPool2d_stride1 stores the shape of input.
- As an output, forward returns variable `Z`.

The attribute function `backward` includes:

- As an argument, backward expects input `dLdZ`.
- As attributes, `Pool2d_stride1` backward stores no variables.
- As an output, backward returns variable `dLdA`.

You are expected to complete pooling operation in **MaxPool2d_stride1** and **MeanPool2d_stride1**. Use **MaxPool2d** and **MeanPool2d** as a wrapper class to implement downsampling after the max or mean operations.

```
class MaxPool2d():

    def __init__(self, kernel, stride):
        self.kernel = kernel
        self.stride = stride

        #Create an instance of MaxPool2d_stride1
        self.maxpool2d_stride1 = MaxPool2d_stride1(kernel)
        self.downsample2d = Downsample2d(stride)

    def forward(self, A):
        Z = # TODO
        return Z

    def backward(self, dLdZ):
        dLdA = # TODO
        return dLdA

class MeanPool2d():

    def __init__(self, kernel, stride):
```

```
        self.kernel = kernel
        self.stride = stride

    def forward(self, A):
        Z = # TODO
        return Z

    def backward(self, dLdZ):
        dLdA = # TODO
        return dLdA
```

As you can see, each `Pool2d` class has forward, and backward attribute functions. In forward, we calculate `Z` and store values needed for backward. The attribute function `forward` includes:

- As an argument, forward expects input `A`.
- As an attribute, forward stores the nothing.
- As an output, forward returns variable `Z`.

The attribute function `backward` includes:

- As an argument, backward expects input `dLdZ`.
- As attributes, `Pooling` backward stores no variables.
- As an output, backward returns variable `dLdA`.

Table 12: Pooling Class Components

Code Name	**Math**	**Type**	**Shape**	**Meaning**
`stride`	*stride*	scalar	-	downsampling factor
`kernel`	*kernel_size*	scalar	-	kernel_size
`A`	A	matrix	$N \times C_0 \times H_0 \times W_0$	data input for pooling
`Z`	Z	matrix	$N \times C_0 \times H_1 \times W_1$	features after pooling
`dLdZ`	$\partial L / \partial Z$	matrix	$N \times C_0 \times H_1 \times W_1$	how changes in outputs affect loss
`dLdA`	$\partial L / \partial A$	matrix	$N \times C_0 \times H_0 \times W_0$	how changes in inputs affect loss

6 Converting Scanning MLPs to CNNs [10 Points]

6.1 CNN as a Simple Scanning MLP

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. 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** (4×31) **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_channels`, `out_channels`, `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_stride1 layers.
  - This will involve (1) reshaping a transposed weight matrix into `out_channels, kernel_size, in_channels` and (2) transposing the weights back into the correct shape for the weights of a Conv1d_stride1 instance, `out_channels, in_channels, 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_stride1 layer. I.e. weights[0] are the weights for the first Conv1d_stride1 instance.

```
class CNN_SimpleScanningMLP():
    def __init__(self):
        self.conv1 = # TODO
        self.conv2 = # TODO
        self.conv3 = # TODO
        self.layers = []

    def init_weights(self, weights):
        w1,w2,w3 = weights
        self.conv1.conv1d_stride1.W = # TODO
        self.conv2.conv1d_stride1.W = # TODO
        self.conv3.conv1d_stride1.W = # TODO

    def forward(self, A):
        Z = A
        for layer in self.layers:
            Z = layer(Z)
        return Z

    def backward(self, dLdZ):
        dLdA = dLdZ
        for layer in self.layers[::-1]:
            dLdA = layer.backward(dLdA)
        return dLdA
```

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. We will not give you tables of variables in this section. Please figure out the shape and complete this section.

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

6.2 CNN as a Distributed Scanning MLP

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.**

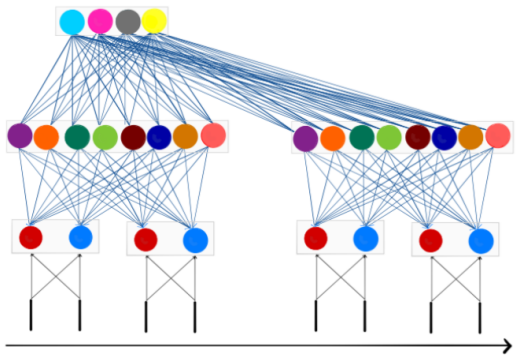


Figure 31: The Distributed Scanning MLP network architecture

```
class CNN_DistributedScanningMLP():
    def __init__(self):
        self.conv1 = # TODO
        self.conv2 = # TODO
        self.conv3 = # TODO
        self.layers = []

    def init_weights(self, weights):
        w1, w2, w3 = weights
        self.conv1.conv1d_stride1.W = # TODO
        self.conv2.conv1d_stride1.W = # TODO
        self.conv3.conv1d_stride1.W = # TODO

    def forward(self, A):
        Z = A
        for layer in self.layers:
            Z = layer(Z)
        return Z

    def backward(self, dLdZ):
        dLdA = dLdZ
        for layer in self.layers[::-1]:
            dLdA = layer.backward(dLdA)
        return dLdA
```

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_channels`, `out_channels`, `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_stride1` layers. This will involve:

  (1) Reshaping the transposed weight matrix into `out_channels, kernel_size, in_channels`

  (2) Transposing the weights again into the correct shape for the weights of a `Conv1d_stride1` 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.

7 Build a CNN model [5 Points]

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

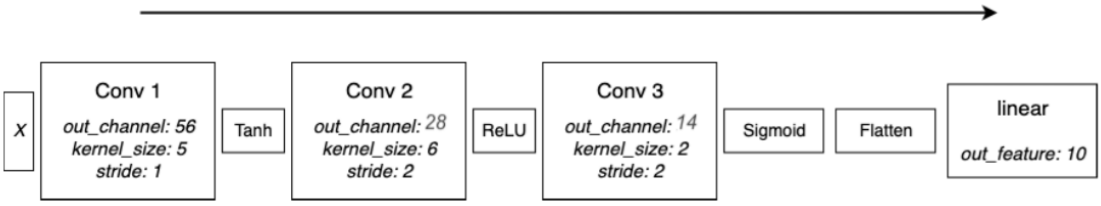


Figure 32: CNN Architecture to implement.

- 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!!

8 Appendix

8.1 Scanning MLP : Illustration

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

- 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.

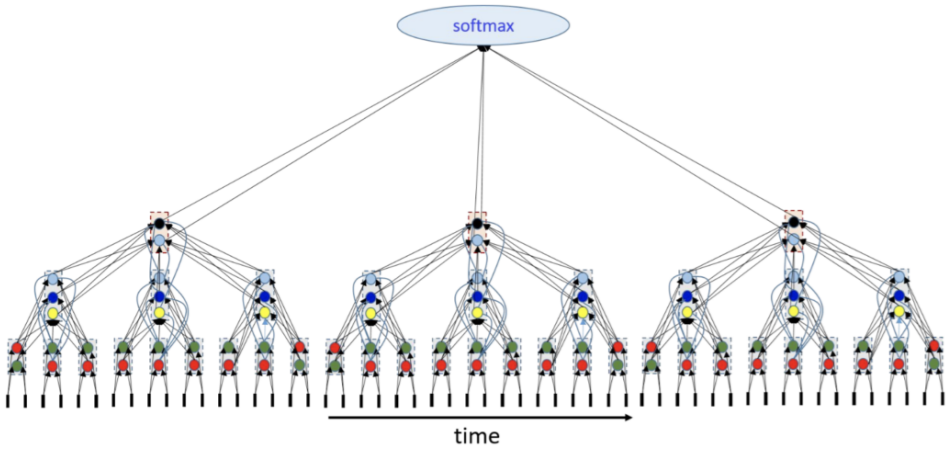


Figure 33: Scanning MLP Figure 1

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).

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 the 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 analaysis. The outputs from all the individual evaluations by the scanning MLP are sent to a final softmax.

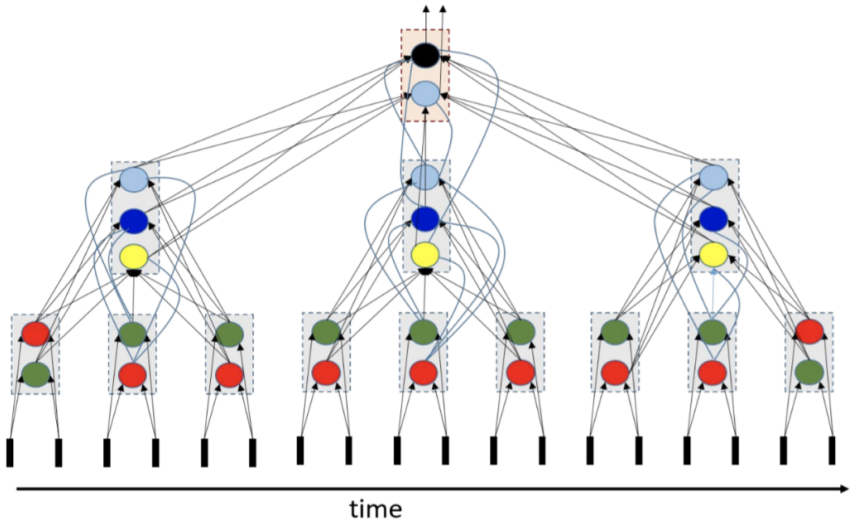


Figure 34: Scanning MLP Figure 2

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.

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,

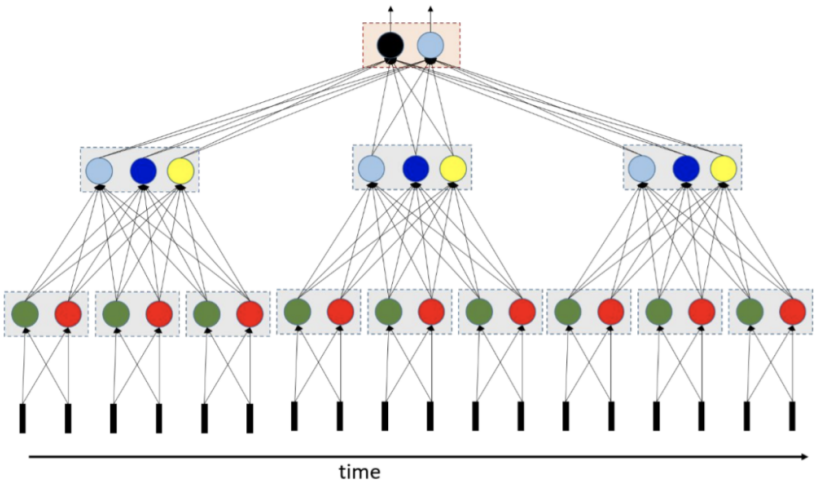


Figure 35: Scanning MLP Figure 3

or could they be grouped for vector activations. Such intuitions lead to other architectures.