Homework 2 Part 1

An Introduction to Convolutional Neural Networks

11-785: Introduction to Deep Learning (Spring 2025)

OUT: **Feb 7th, 2025**
EARLY SUBMISSION BONUS: **Feb 21st, 2025, 11:59 PM, Eastern Time**
DUE: **Mar 1st, 2025, 11:59 PM, Eastern Time**

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.
- **Directions:**
  - 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.

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 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 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)

Checklist

Here is a checklist page that you can use to keep track of your progress as you go through the write-up and implement the corresponding sections in your starter notebook. As you complete each function in the notebook, you can check the corresponding boxes aligned with each section.

1. Getting Started
    - ☐ Download code handout and extract the file
    - ☐ Activate the conda environment created in HW1P1
    - ☐ Copy your `linear.py`, `activation.py` and `loss.py` from HW1P1 to `mytorch` fol.
    - ☐ Read the whole assignment write-up for an overview
    - ☐ Please review Recitations 0A (Python Fundamentals), 0C (Numpy Fundamentals), 0N (Debugging), 0O (What to do when you're struggling), 0P (Cheating), and 0Q (Workflow of homeworks) for supplementary material that could be helpful for this assignment.
2. Complete the Components of a CNN
    - ☐ Complete Resampling for 1D and 2D data [10 Points]
    - ☐ Complete the following Convolutional layers
        - ☐ Complete `Conv1d_stride1` and use it to build `Conv1d` [15 Points]
        - ☐ Complete `Conv2d_stride1` and use it to build `Conv2d` [15 Points]
        - ☐ Implement `ConvTranspose1d` and `ConvTranspose2d` [10 Points]
    - ☐ Complete `MaxPool2d` and `MeanPool2d` [30 Points]
    - ☐ Complete `Flatten` layer
3. Putting everything together to make CNN models
    - ☐ Convert Scanning MLPs to CNNs [15 Points]
    - ☐ Build a CNN model using the parts we coded in previous sections [5 Points]
4. Hand-in
    - ☐ Make sure to set all debug/autograde flags to true
    - ☐ Make sure you pass all test cases in the local autograder
    - ☐ Make the `handin.tar` file and submit to autolab

Contents

1 Setup and Submission

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. For Homework 2, *MyTorch* © will have the following file structure:

```
handout
├── mytorch
│   ├── nn
│   │   ├── Conv1d.py
│   │   ├── Conv2d.py
│   │   ├── ConvTranspose.py
│   │   ├── pool.py
│   │   ├── resampling.py
│   │   ├── linear.py (Copy your file from HW1P1)
│   │   ├── activation.py (Copy your file from HW1P1)
│   │   └── loss.py (Copy your file from HW1P1)
│   └── flatten.py
├── models
│   ├── mlp.py
│   ├── mlp_scan.py
│   └── cnn.py
├── tests
│   └── HW2P1_Tests
│       ├── helpers.py
│       ├── test.py
│       └── runner.py
└── create_tarball.sh
```

- **Environment** Activate the environment created in HW1P1 for all necessary packages:

  ```
  conda activate idlf24
  ```

- **(IMPORTANT) Copy** your completed `linear.py`, `activation.py`, `loss.py` from HW1P1, ensure that you received full marks for these sections. (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 :D.)

  (IMPORTANT!!!! Note: In the activation.py file, for the tanh activation, use self.A = np.tanh(Z), otherwise it can result in underflow issues.

- **Sandbox Code Testing** is available in the `sandbox` directory. There, you may experiment with trying out your own inputs. We have provided skeleton sandbox code so that you can quickly begin testing! All you need to do is run files from anywhere with

  ```
  python3 [path_to_sandbox_file].py
  ```

- **Autograde** and test your code by running the following command from the top level directory:
  - Step 1: Setting the flags in

    `tests/HW2P1\_Tests/hw2p1 autograder\_flags.py`

    to True to test any individual component on your local autograder.
  - Step 2: Running local autograder by: Confirm that you are the top-level directory and execute the following in anaconda prompt or terminal:

    ```
    python3 tests/HW2P1_Tests/runner.py
    ```

- **Hand-in** your code by first making sure to set all your autograde flags to true. Then run the following command from the top level directory, then **SUBMIT** the created *handin.tar* file to autolab:

  ```
  sh create_tarball.sh
  ```

- **Debugging Tool**
  Use the interactive python debugger pdb[1] to debug your code effectively.
  - In your code, set breakpoints by adding this line: `import pdb; pdb.set_trace()`
  - Run the following line from top level directory to turn on pdb with your local autograder:

    ```
    python3 -m pdb autograder/runner.py
    ```

[1] Useful commands: pdb cheatsheet

2 Notation

****Numpy Tips:**

- Use `A * B` for element-wise multiplication $A \odot B$.
- Use `A @ B` for m**AT**rix multiplication $A \cdot B$.
- Use `A / B` for element-wise division $A \oslash B$.

Linear Algebra Operations

A^T	Transpose of A
$A \odot B$	Element-wise (Hadamard) Product of A and B
$A \cdot B$	Matrix multiplication of A and B
$A \oslash B$	Element-wise division of A and B
$A \circledast B$	Matrix convolution of A and B

Functions and Operations

$\log(x)$	Natural logarithm of x
$\varsigma(x)$	Sigmoid, $\dfrac{1}{(1+\exp^{-x})}$
$\tanh(x)$	Hyperbolic tangent, $\dfrac{e^x - e^{-x}}{e^x + e^{-x}}$
$max_{\mathbb{S}} f$	The operator $max_{a \in \mathbb{S}} f(a)$ returns the highest value $f(a)$ for all elements in the set $\mathbb{S}$
$argmax_{\mathbb{S}} f$	The operator $argmax_{a \in \mathbb{S}} f(a)$ returns the element a of the set $\mathbb{S}$ that maximizes $f(a)$
$\sigma(x)$	Softmax function, $\sigma : \mathbb{R}^K \to (0,1)^K$ and $\sigma(x)_i = \dfrac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}}$ for $i = 1, ..., K$

Calculus

$\dfrac{dy}{dx}$	Derivative of scalar y with respect to scalar x
$\dfrac{\partial y}{\partial x}$	Partial derivative of scalar y with respect to scalar x
$\dfrac{\partial f(Z)}{\partial Z}$	Jacobian matrix $\mathbf{J} \in \mathbb{R}^{N \times M}$ of $f : \mathbb{R}^M \to \mathbb{R}^N$

3 Introduction

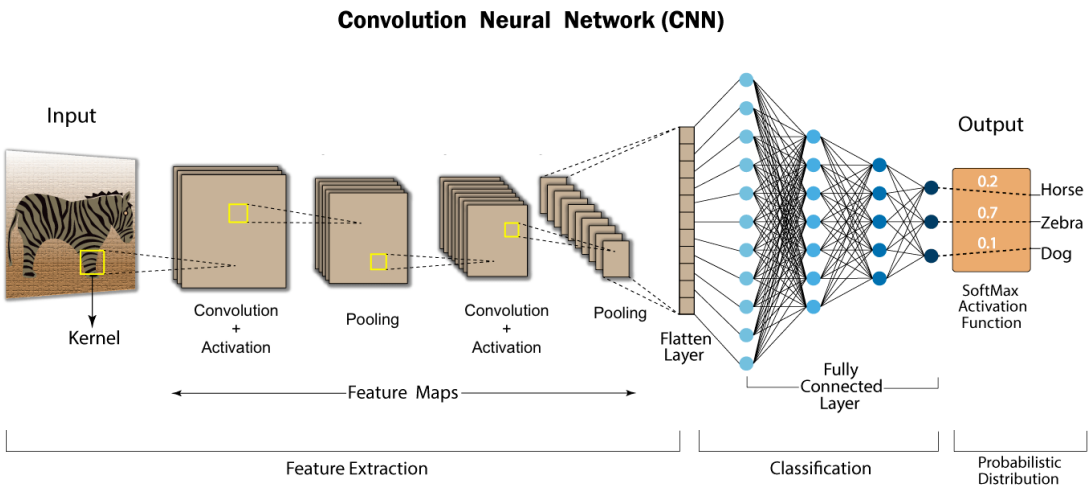


Figure 1: Standard 2d CNN for image classification

A Convolutional Neural Network is a **position-invariant** pattern detector and a special kind of FFNN (FeedForward Neural Network) that significantly reduces the number of parameters in a large deep neural network without losing too much in quality. CNNs have wide-ranging use-cases in image classification, object detection, semantic segmentation, image captioning, natural language processing, forecasting, and more. The foundational concept that it is based on is the concept of convolution. It is recommended to understand the convolution operation to fully appreciate the inspiration behind using CNNs. A good reference by 3B1B can be found here: But what is a convolution?

3.1 Structure of a CNN

A Convolutional Neural Network consists of the following parts:

- **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 extracting features and capturing patterns from the input. For example, in images of cars, one convolutional layer filter might extract "wheel" patterns while another filter might extract the patterns formed by the car frame. Check out CNN Explainer for a more visual explanation about CNNs!

- **Resampling (Upsampling and Downsampling)** Resampling consists of upsampling and downsampling. Upsampling expands and "bloats" our input to allow for finer detail. Conversely, downsampling reduces the size of our input (by discarding some data points), thereby reducing the needed computation on our input. (You will not have to do this, but anti-aliasing filters are typically placed to retain the most critical information if you were wondering!)

  Why are we doing this? Let's say you have implemented `Conv_stride1` class and want to generalize your implementation to work for `stride != 1`. You can tack on resampling to the output of the `Conv_stride1` to "simulate" the stride. To understand resampling, let's break up the Conv Layer into 3 main cases:

  - Stride $= 1$: In this case, after our `Conv_stride1` layer, we will simply proceed on to Pooling.
  - Stride > 1: we need to DOWNSAMPLE the output of our `Conv_stride1` layer.
  - Stride < 1: we need to UPSAMPLE the input and pass it to our `Conv_stride1` layer.

- **Activation**
  As we've seen extensively in HW 1, we will now apply an activation to the output of our convolutional layer.
- **Pooling**
  Next, we have a pooling layer, which generalizes the previous layer's output to maintain performance despite minor input variations.
- **Flatten**
  Finally, after obtaining all the feature maps, the flattening layer flattens quite simply flattens our pooled feature maps into a 1D array before passing them into the classification layers.
- **Classification Layer**
  - **Linear Layer:** Now, we'll insert this long vector of data (consisting of flattened, pooled feature maps) into a linear layer. This fully connected linear layer will allow our network to move towards performing a classification, i.e. map the representation between the input and the output.
  - **Softmax Layer:** [2] 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.

3.2 1D vs 2D

1D Convolutional Neural Networks (Conv1D) are used for processing sequences of one-dimensional signals, such as time-series data or text data represented as a sequence of word embeddings.

2D Convolutional Neural Networks (Conv2D) are used for processing two-dimensional data, such as images.

For Conv Layer class and Resampling classes, we will implement both the 1D and 2D versions.

[2]Note that Softmax layer is not always required, for HW2P2, if you are using CrossEntropy loss, it expects logits instead of probability.

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 Upsample1d

In this section, your task is to implement the `Upsample1d` class in `resampling.py` file. You may test your implementation in the `sandbox/resampling_sandbox.py` file.

Table 1: Upsample1d Class Components

Code Name	Math	Type	Shape	Meaning
`batch_size`	N	scalar	-	batch size
`in_channels`	C	scalar	-	Number of channels
`input_width`	W_{in}	scalar	-	Width of input channels
`output_width`	W_{out}	scalar	-	Width of output channels
`upsampling_factor`	k	scalar	-	upsampling_factor $\in \mathbb{Z}^+$
`A`	A	matrix	$N \times C \times W_{in}$	pre-upsampling values
`Z`	Z	matrix	$N \times C \times W_{out}$	post-upsampling values
`dLdZ`	$\partial L / \partial Z$	matrix	$N \times C \times W_{out}$	gradient of Loss wrt Z
`dLdA`	$\partial L / \partial A$	matrix	$N \times C \times W_{in}$	gradient of Loss wrt A

4.1.1 Upsample1d Forward

Upsampling as the name suggests is used to increase the size of input. In the `MyTorch` implementation, we do this by a simple operation of adding $(k-1)$ intermediate 0s. The Figure 2 shows how Upsampling is done.

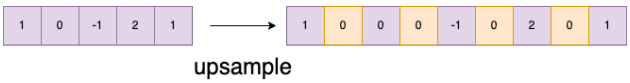


Figure 2: Upsampling1d Forward Example (k=2)

The output will be of width

$$W_{out} = k \times (W_{in} - 1) + 1$$

where k is the *upsampling_factor*. For this example, k = 2, so only 1 zero is inserted between the elements.
Hint: You will want to transfer your input data into a new array of 0's that has a width of W_{out}.
Hint: Did you know about Python's slice assignments :D

4.1.2 Upsample1d Backward

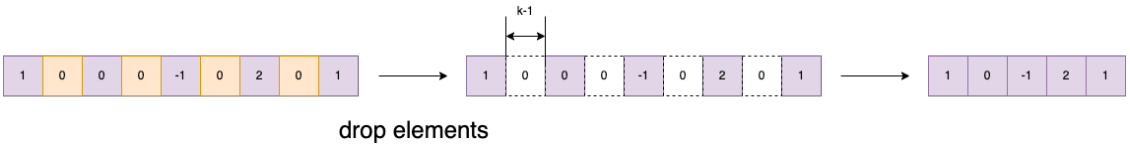


Figure 3: Upsampling1d Backward Example (k=2)

Refer to Figure 3, we do the reverse operation of forward. Only the `dLdZ` entries corresponding to the original input data should be kept, and the `dLdZ` entries corresponding to the padded 0s should have no effect on `dLdA` and hence should be dropped.

4.2 Downsampling1d

Downsampling1d is just "dropping" off elements with a factor of k.
In this section, your task is to implement `Downsample1d` class.

Table 2: Downsample1d Class Components

Code Name	Math	Type	Shape	Meaning
`downsampling_factor`	k	scalar	-	downsampling factor $\in \mathbb{Z}^+$
`A`	A	matrix	$N \times C \times W_{in}$	pre-downsampling features
`Z`	Z	matrix	$N \times C \times W_{out}$	post-downsampling features
`dLdZ`	$\partial L/\partial Z$	matrix	$N \times C \times W_{out}$	gradient of Loss wrt Z
`dLdA`	$\partial L/\partial A$	matrix	$N \times C \times W_{in}$	gradient of Loss wrt A

4.2.1 Downsample1d Forward

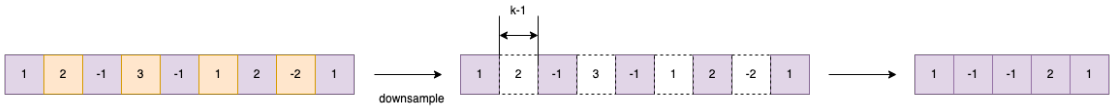


Figure 4: Downsampling1d Forward Example (k=2)

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

4.2.2 Downsample1d Backward

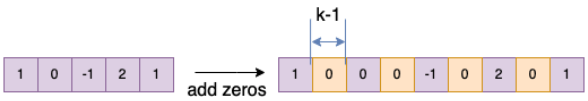


Figure 5: Downsampling1d Backward Example (k=2)

In `backward` function, one thing to note is that `dLdZ`, the gradient of input should be the **same size as the input**. We pad 0s in between because the gradient of Loss wrt the elements dropped in downsampling forward should be 0.

Hint: You might want to store W_{in} in `forward`. Think about what happens when the input size is even/odd.

4.3 Upsampling2d

2D upsampling is used for inputs like images where upsampling is performed in both the x and y direction. Your task is to implement the `Upsample2d` class in `resample.py`. You may test your implementation in the `sandbox/resampling_sandbox.py` file

Table 3: Upsample2d Class Components

Code Name	Math	Type	Shape	Meaning
`upsampling_factor`	k	scalar	-	upsampling factor $\in \mathbb{Z}^+$
`A`	A	matrix	$N \times C \times H_{in} \times W_{in}$	pre-upsampling features
`Z`	Z	matrix	$N \times C \times H_{out} \times W_{out}$	post-upsampling features
`dLdZ`	$\partial L / \partial Z$	matrix	$N \times C \times H_{out} \times W_{out}$	gradient of Loss wrt Z
`dLdA`	$\partial L / \partial A$	matrix	$N \times C \times H_{in} \times W_{in}$	gradient of Loss wrt A

4.3.1 Upsampling2d Forward

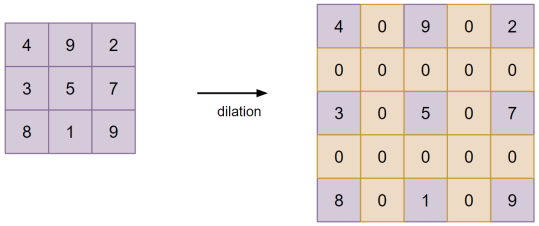


Figure 6: Upsampling 2d (k=2)

Upsampling an image is a simple operation where $(k-1)$ zeros are added in between pixels of the input map. (Some may know this as dilation). The above diagram gives an intuitive explanation.

- Can you deduce H_{out} based on H_{in} and k?
- Can you deduce W_{out} based on W_{in} and k?
- **Hint:** same logic as `Upsample1d`

4.3.2 Upsample2d Backward

The exact opposite takes place in backward where intermediate $k-1$ elements are dropped.

4.4 Downsampling2d

In downsampling, the input features are reduced by a factor of k in both x and y dimension.

In this section, you will implement the `Downsample2d` class. You may test your implementation in the `sandbox/resampling_sandbox.py` file

Table 4: Downsample2d Class Components

Code Name	Math	Type	Shape	Meaning
`downsampling_factor`	k	scalar	-	downsampling factor $\in \mathbb{Z}^+$
`A`	A	matrix	$N \times C \times H_{in} \times W_{in}$	pre-downsampling features
`Z`	Z	matrix	$N \times C \times H_{out} \times W_{out}$	post-downsampling features
`dLdZ`	$\partial L / \partial Z$	matrix	$N \times C \times H_{out} \times W_{out}$	gradient of Loss wrt Z
`dLdA`	$\partial L / \partial A$	matrix	$N \times C \times H_{in} \times W_{in}$	gradient of Loss wrt A

4.4.1 Downsample2d Forward

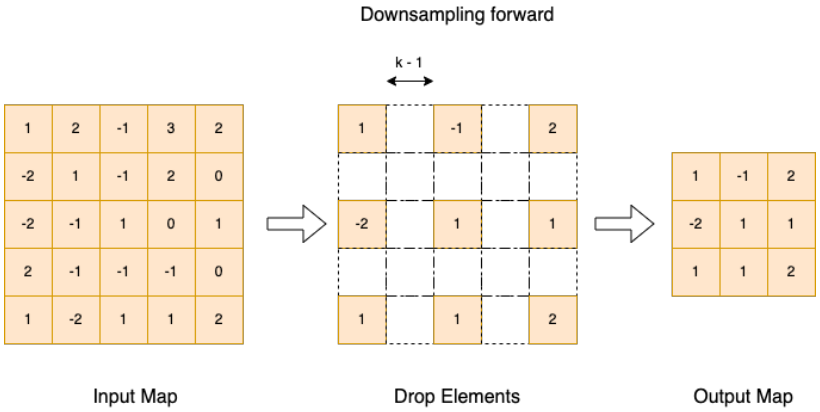


Figure 7: Downsample 2d Forward Example (k=2)

- Can you deduce H_{out} based on H_{in} and k?
- Can you deduce W_{out} based on W_{in} and k?
- **Hint:** same logic as `Downsample1d`

4.4.2 Downsample2d Backward

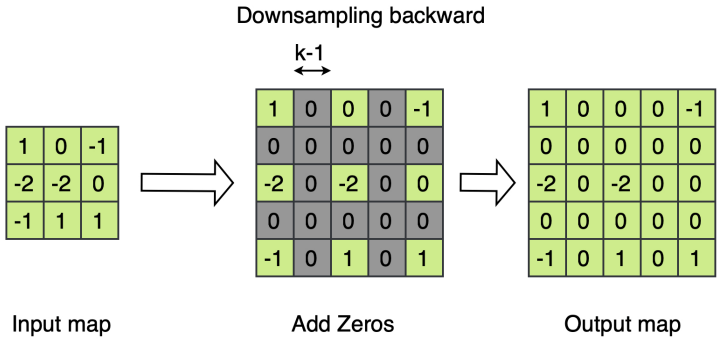


Figure 8: Downsample 2d Backward Example (k=2)

In `backward` function, we pad 0s in between because the gradient of Loss wrt the elements dropped in downsampling forward should be 0. Since the values of the dropped elements don't affect the downsampling output, they have zero gradients with respect to Loss.

5 Convolutional layer

Congrats on completing the Resampling layers. Now come the interesting portions of this homework. We will look into 1d and 2d convolutions and how to implement them from scratch.
But first of all, what is a convolution anyway and why does it help us capture the patterns?

Assume that we have a pattern which is 3 by 3 pixels that looks like this:

$$P_1 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

The above matrix represents a pattern that looks like a cross. The small regression model that will detect such patterns would need to learn a 3 by 3 parameter matrix F where parameters at positions corresponding to the 1s in the input patch would be positive numbers, while the parameters in positions corresponding to 0s would be close to zero. If we take the elements of F as the weight parameters for corresponding elements of P, the weighted sum we obtain is higher the more similar F is to P. For instance, assume that F looks like this:

$$F = \begin{bmatrix} 0 & 2 & 1 \\ 2 & 4 & 3 \\ 0 & 3 & 0 \end{bmatrix}$$

Convolving P_1 and F, we will get

$$P_1 \circledast F = 0 \cdot 0 + 1 \cdot 2 + 0 \cdot 1 + 1 \cdot 2 + 1 \cdot 4 + 1 \cdot 3 + 0 \cdot 0 + 1 \cdot 3 + 0 \cdot 0 = 14$$

If our input patch had a different pattern, for example, that of a letter T:

$$P_2 = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

The convolution result will be lower:

$$P_2 \circledast F = 10$$

So, you can see the more the patch "looks" like the filter, the higher the value of the convolution operation is.

Important note: For this section, please refer to Appendix 10.2 to learn about how to use tensordot in order to avoid multiple nested loops. It takes some time to learn, but it will save you a lot of time later.

5.1 Conv1d_stride1 [10 Points]

Convolution 1d involves convolving the input with a kernel in just 1 direction. We will first implement `Conv1d_stride1`[3], and use the resampling functions which we just built to generalize it to work for `stride != 1`.

In this section, you will implement the `Conv1d_stride1` class in `Conv1d.py` file. You may test your implementation in the `sandbox/conv1d_sandbox.py` file

Table 5: Conv1d_stride1 Class Components

Code Name	Math	Type	Shape	Meaning
`kernel_size`	K	scalar	-	kernel size
`A`	A	matrix	$N \times C_{in} \times W_{in}$	data input for convolution
`Z`	Z	matrix	$N \times C_{out} \times W_{out}$	features after conv1d with stride 1
`W`	W	matrix	$C_{out} \times C_{in} \times K$	weight parameters, also called kernels in CNN
`b`	b	vector	$C_{out} \times 1$	bias parameters
`dLdZ`	$\partial L/\partial Z$	matrix	$N \times C_{out} \times W_{out}$	how changes in outputs affect loss
`dLdA`	$\partial L/\partial A$	matrix	$N \times C_{in} \times W_{in}$	how changes in inputs affect loss
`dLdW`	$\partial L/\partial W$	matrix	$C_{out} \times C_{in} \times K$	how changes in weights affect loss
`dLdb`	$\partial L/\partial b$	vector	$C_{out} \times 1$	how changes in bias affect loss

5.1.1 Conv1d_stride1 Forward

When $C_{in} = 1$:

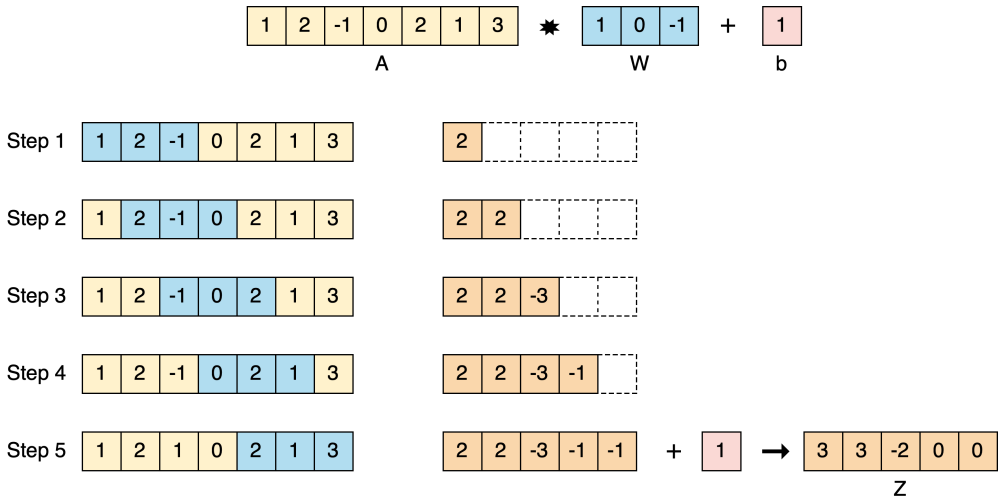


Figure 9: Conv1d with single channel input

An example illustrating the forward function is presented in Figure 9. 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 and the filter elements. The output for a single convolutional step is a single scalar (orange). In each step, the filter shifts to the right by stride = 1. 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.

Based on the example, can you deduce the value of W_{out} based on W_{in} and kernel_size?

[3]Stride is the number of pixels by which the kernel moves at each step.

When $C_{in} > 1$:

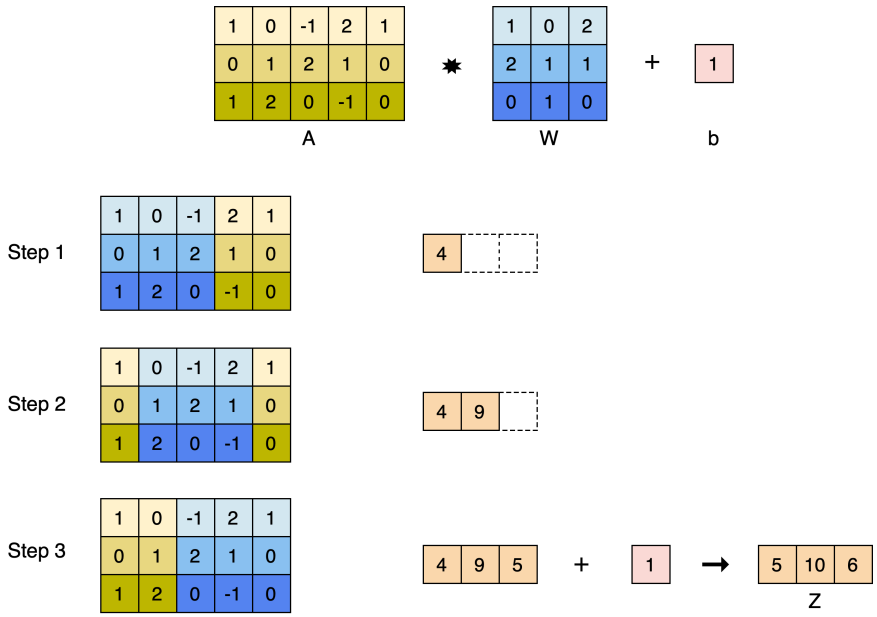


Figure 10: Conv1d with multichannel input

Figure 10 explains multi-channel convolutions. In our case, multiple channels may be used is to hold more information about the same pixel such as the intensities of the Red, Green, and Blue (RGB) colors. 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. The weight W has size $C_{out} \times C_{in} \times K$, meaning that we have C_{out} filters of size $C_{in} \times K$, each producing 1 different outputs with width W_{out}, hence producing an output of shape $C_{out} \times W_{out}$. We recommend you to understand the process and try the convolutions by hand before proceeding to code.

5.1.2 Conv_stride1 Backward

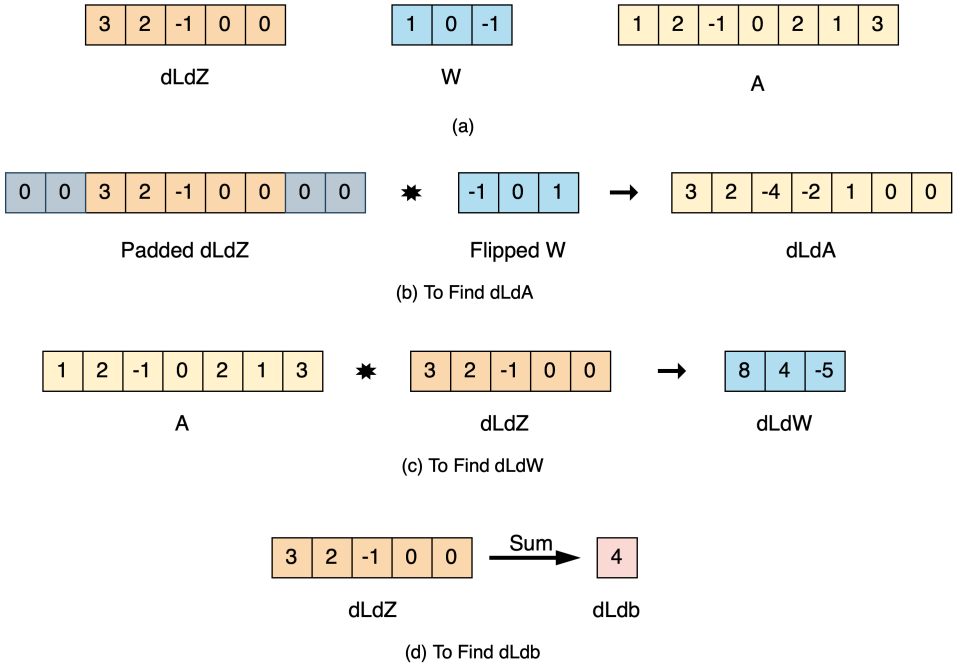


Figure 11: Conv1d backward with single channel input

Convolution backward is almost exactly the same operation done in the reverse order. Consider the figure shown in 11 for single channel backward and 12 for multi-channel 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`.

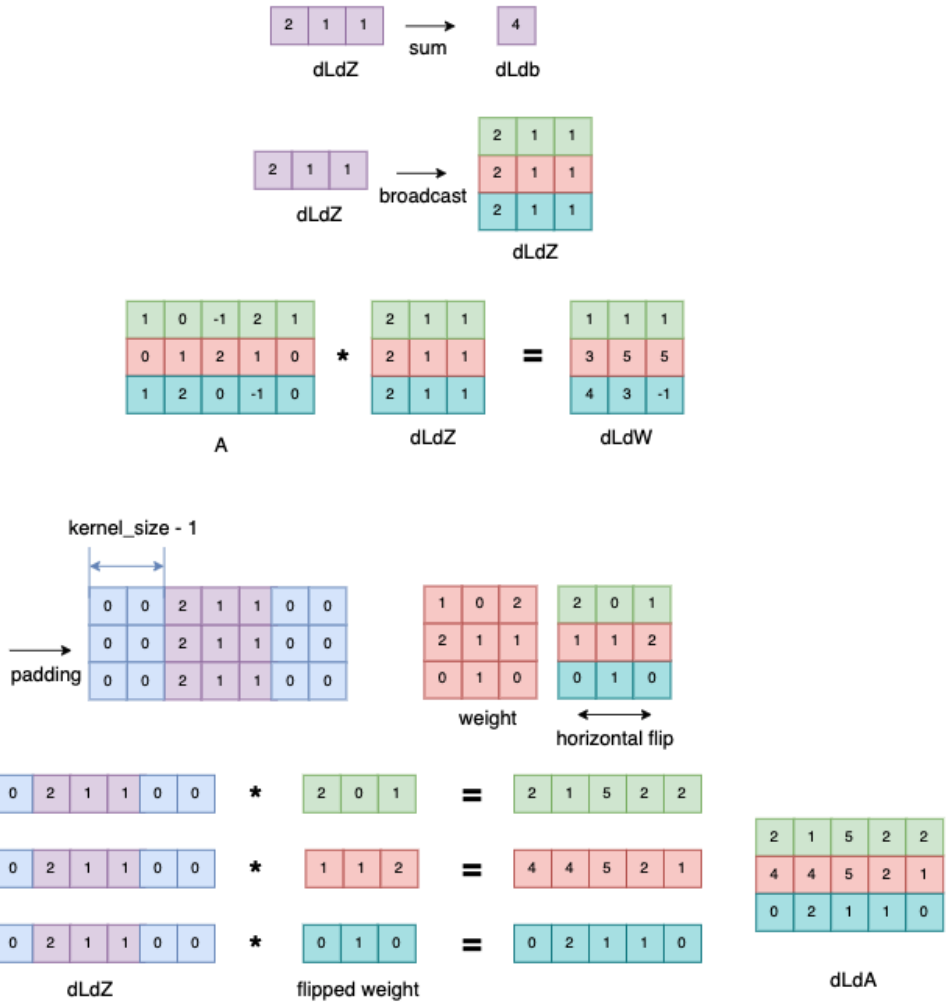


Figure 12: Conv1d_stride1 Multichannel Backward Example

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: Do a row-wise convolution between `dLdZ` and the input `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 convoluting `dLdZ` with the corresponding channel in input.

Finding dLdA:

- Broadcast[4] `dLdZ` C_{in} times as shown. 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 11, it is not required as $C_{in} = 1$.
- Pad this with `kernel_size - 1` zeros on both sides (This is done as we would require the output to

[4]You don't need to explicitly broadcast, tensordot would save you the trouble!

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`

5.2 Conv1d [5 points]

In this section, we would be implementing `Conv1d` class in `Conv1d.py` file which works for any stride > 1. We will reuse the code from conv1d_stride1 5.1 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. You may test your implementation in the `sandbox/conv1d_sandbox.py` file

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).

Your task is to implement forward and backward of the `Conv1d` class.

Table 6: Conv1d Class Components

Code Name	Math	Type	Shape	Meaning
`stride`	*stride*	scalar	-	equivalent to downsampling factor
`A`	A	matrix	$N \times C_{in} \times W_{in}$	data input for convolution
`Z`	Z	matrix	$N \times C_{out} \times W_{out}$	features after conv1d with stride
`dLdZ`	$\partial L/\partial Z$	matrix	$N \times C_{out} \times W_{out}$	how changes in outputs affect loss
`dLdA`	$\partial L/\partial A$	matrix	$N \times C_{in} \times W_{in}$	how changes in inputs affect loss

5.3 Conv2d_stride1 [10 Points]

2D 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.

In this section, you will implement the `Conv2d_stride1` class in `Conv2d.py` file. Similar to `Conv1d_stride1`, this class will be further generalized to work for `stride != 1` in the following sections. You may test your implementation in the `sandbox/conv2d_sandbox.py` file. (Note this sandbox file is a different file).

Table 7: Conv2d_stride1 Class Components

Code Name	Math	Type	Shape	Meaning
`kernel_size`	K	scalar	$N \times C_{in} \times H_{in} \times W_{in}$	kernel size
`A`	A	matrix	$N \times C_{in} \times H_{in} \times W_{in}$	data input for convolution
`Z`	Z	matrix	$N \times C_{out} \times H_{out} \times W_{out}$	features after conv2d with stride 1
`W`	W	matrix	$C_{out} \times C_{in} \times K \times K$	weight parameters, i.e. kernels
`b`	b	vector	$C_{out} \times 1$	bias parameters
`dLdZ`	$\partial L/\partial Z$	matrix	$N \times C_{out} \times H_{out} \times W_{out}$	how changes in outputs affect loss
`dLdA`	$\partial L/\partial A$	matrix	$N \times C_{in} \times H_{in} \times W_{in}$	how changes in inputs affect loss
`dLdW`	$\partial L/\partial W$	matrix	$C_{out} \times C_{in} \times K \times K$	how changes in weights affect loss
`dLdb`	$\partial L/\partial b$	vector	$C_{out} \times 1$	how changes in bias affect loss

Pay special attention to the order of the dimension of A and Z: batch size x channels x height x weight.

5.3.1 Conv2d_stride1 Forward

When $C_{in} = 1$

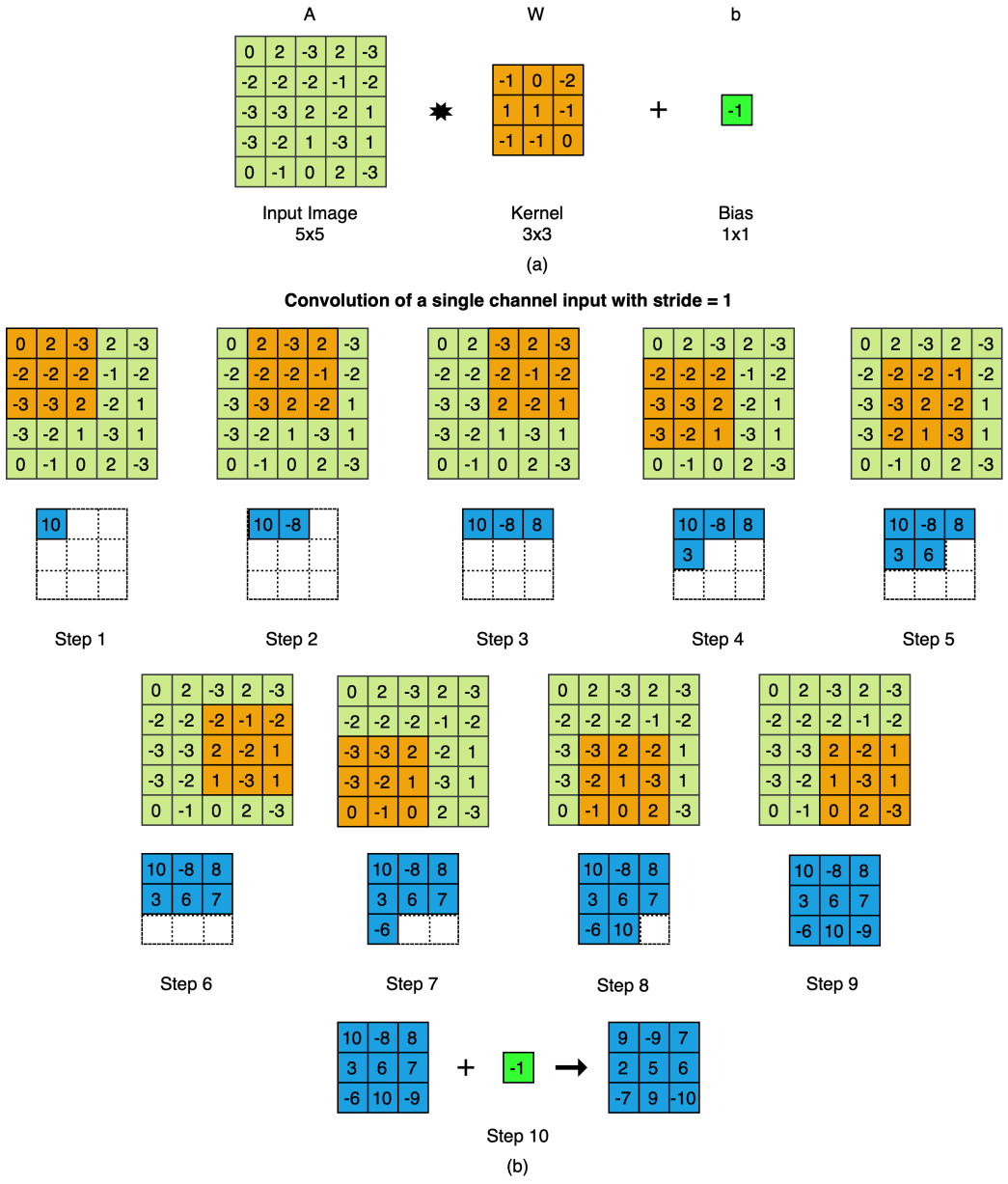


Figure 13: 2d Convolution Example stride=1 $C_{in} = 1$

Let us understand 2d convolutions with an example. Figure 13 shows the convolution of a 3x3 kernel on a 5x5 input image. This example is for a single channel input ($C_{in} = 1$).
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.

When $C_{in} = 2$

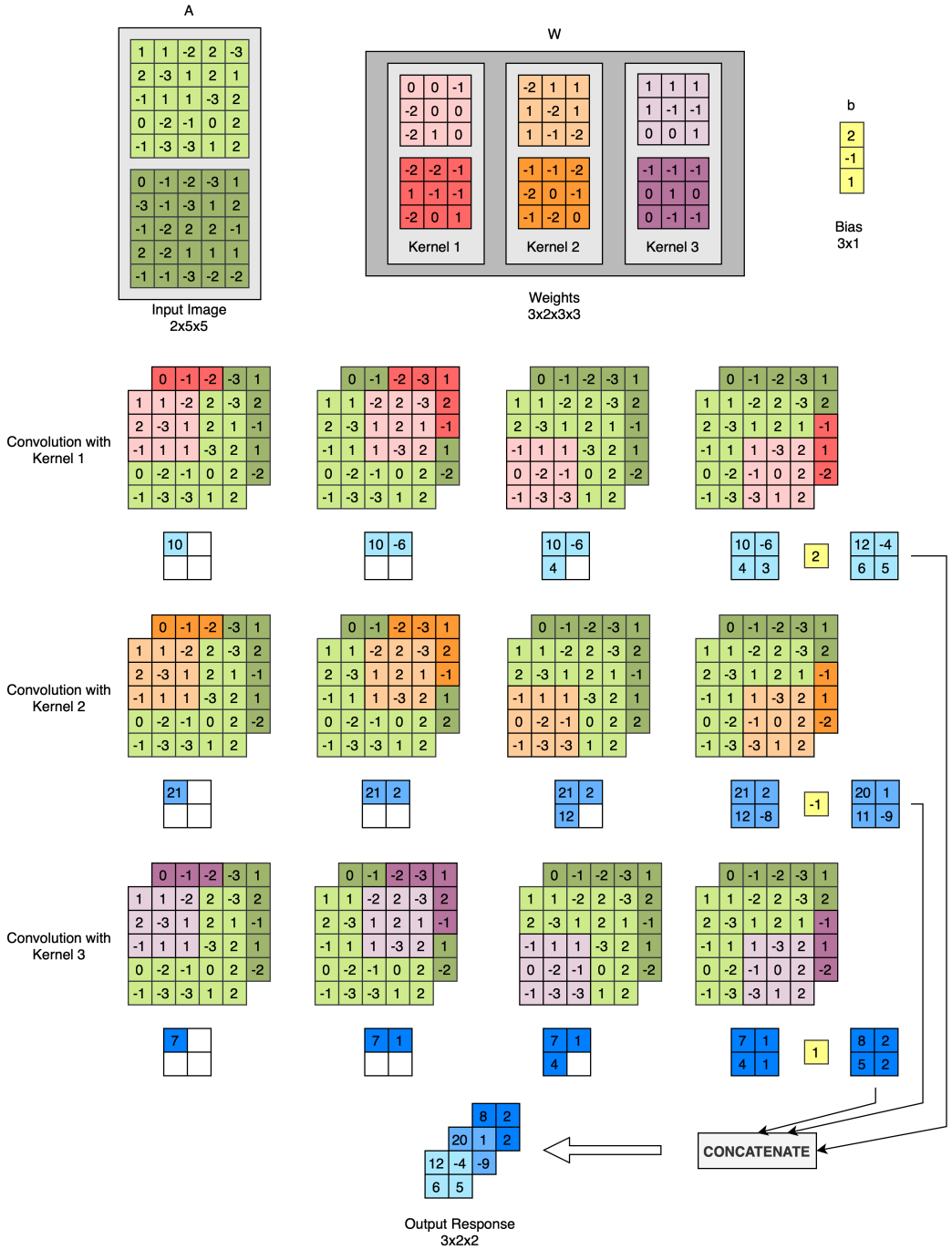


Figure 14: Multi channel convolution example $C_{in} = 2$

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.

As shown in figure 14, A has 2 channels and so do all the kernels in W. Consider convolution with Kernel 1. Starting from the top left, for channel 1 and channel 2 we have,

$$(1 \times 0) + (1 \times 0) + (-2 \times -1) + (2 \times -2) + (-3 \times 0) + (1 \times 0) + (-1 \times -2) + (1 \times 1) + (1 \times 0) = 1$$

$$(0 \times -2) + (-1 \times -2) + (-2 \times -1) + (-3 \times 1) + (-1 \times -1) + (-3 \times -1) + (-1 \times -2) + (-2 \times 0) + (2 \times 1) = 9$$

respectively. This sums to 10.

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, convoluting 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. kernel channels`
- `no. output channels = no. kernels`

5.3.2 Conv2d_stride1 Backward

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` from the previous layers. This gradient map will be of the same shape as Z. With `dLdZ`, we need to find the gradient of the Loss wrt to weights (`dLdW`), bias (`dLdb`) and input (`dLdA`).
To calculate `self.dLdA`:

- Pad the `dLdZ` map with $K-1$ zeros as shown in Figure 15. 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 (`dLdZ`) to get (`dLdA`) as the convolution output.

The above process gives us `self.dLdA` 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 (`dLdZ`).

To calculate `self.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 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 convoluting (`dLdZ`) 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`.

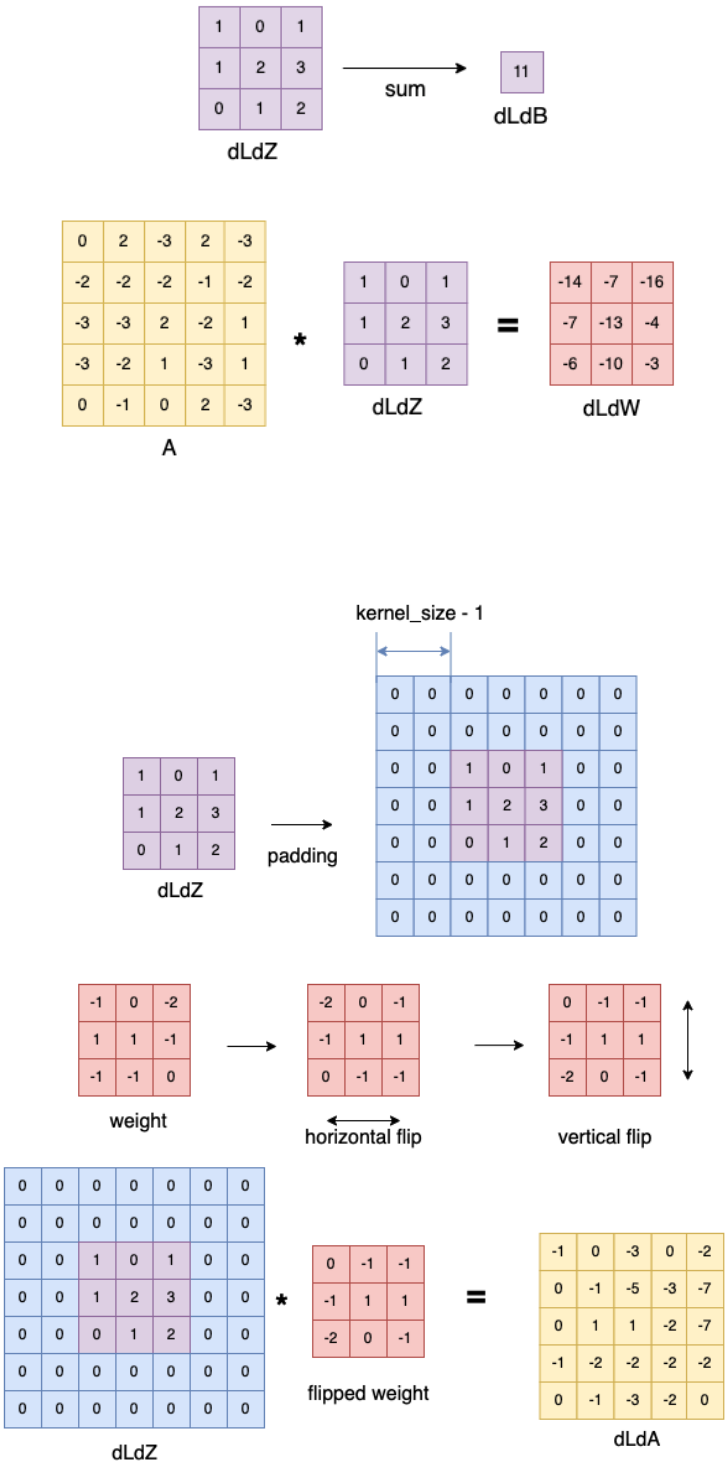


Figure 15: Conv2d_stride1 Backward: Convolution

5.4 Conv2d [5 points]

In this section, we would be implementing `Conv2d` which works for any stride > 1. We will reuse the code from conv1d_stride1 5.3 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 downsample2d layer with factor k is equivalent to a conv1d_stride with stride k. Figure 16 shows this.

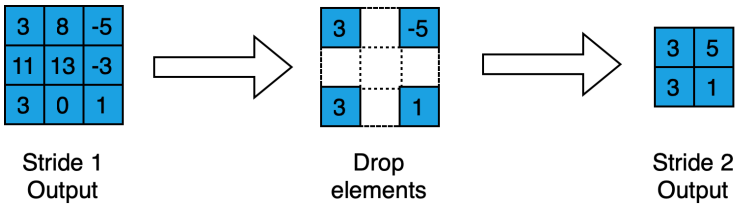


Figure 16: Stride 2 convolution as stride 1 convolution and downsampling with factor = 2

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 17 and then convolution stride 1 backward is called as shown in Figure 18.

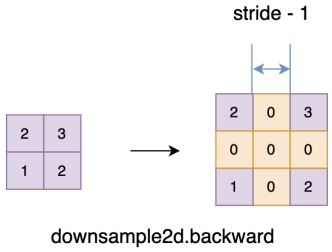
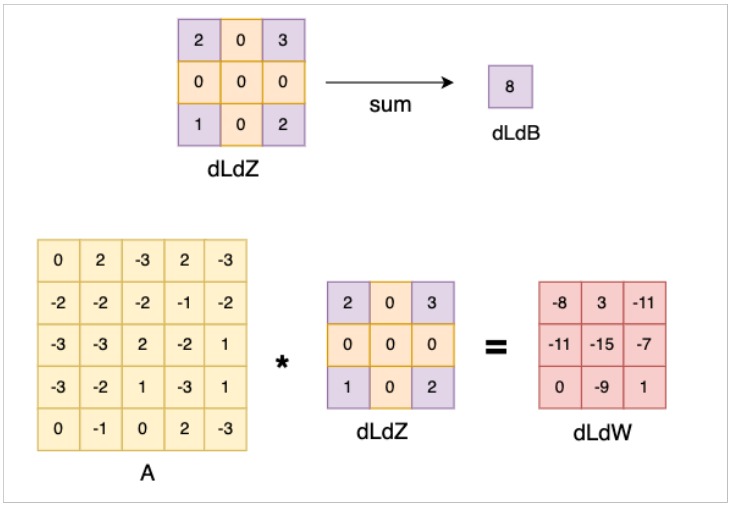


Figure 17: Downsample2d Backward: Conv2d Example



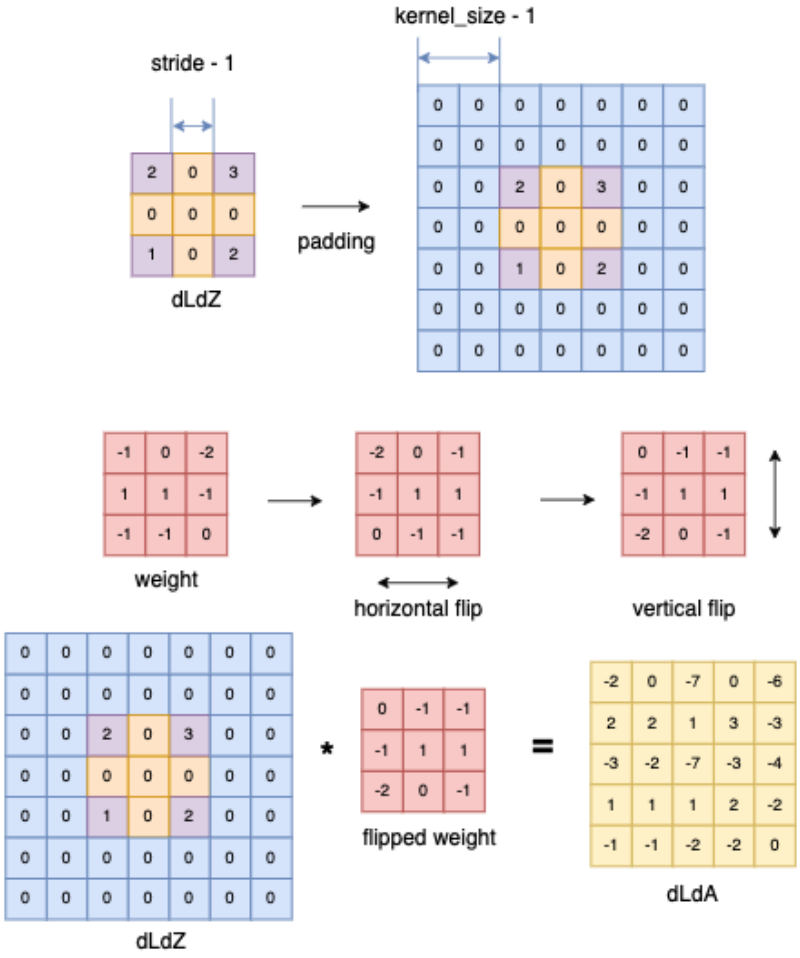


Figure 18: Conv2d Backward Example (stride > 1)

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. You may test your implementation in the `sandbox/resampling_sandbox.py` file

Table 8: Conv2d Class Components

Code Name	**Math**	**Type**	**Shape**	**Meaning**
`stride`	*stride*	scalar	-	Stride = downsampling factor
`A`	A	matrix	$N \times C_{in} \times H_{in} \times W_{in}$	data input for convolution
`Z`	Z	matrix	$N \times C_{out} \times H_{out} \times W_{out}$	features after conv2d with stride
`dLdZ`	$\partial L/\partial Z$	matrix	$N \times C_{out} \times H_{out} \times W_{out}$	how changes in outputs affect loss
`dLdA`	$\partial L/\partial A$	matrix	$N \times C_{in} \times H_{in} \times W_{in}$	how changes in inputs affect loss

5.5 Transposed Convolution [10 points]

5.5.1 ConvTranspose1d

In regular convolutions, the affine value Z for a layer are obtained by a direct convolution on the input values A in the previous layer. This causes a reduction of the output size. However, in an Transposed Convolution layer, the input values A are upsampled and then convolved, to be "pushed" to the next layer Z. Thus 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` k to the ConvTransposed1d class, which is the same in definition of convolution with a fractional stride ($\frac{1}{k}$). Please consider the Figure 19 for better intuition.

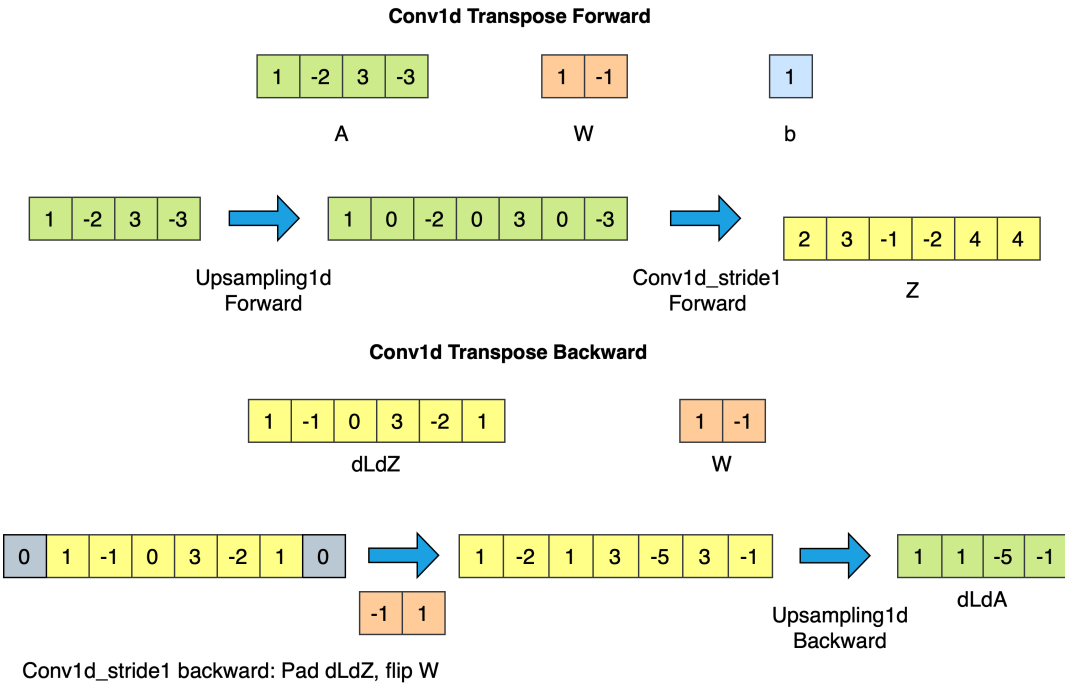


Figure 19: 1d Transpose Convolution

In this section, your task is to implement the `forward` and `backward` attributes of the `ConvTransposed1d` class in `ConvTranspose.py` file.

Table 9: ConvTransposed1d Class Components

Code Name	Math	Type	Shape	Meaning
`upsampling_factor`	$upsampling\_factor$	scalar	-	upsampling factor
`A`	A	matrix	$N \times C_{in} \times W_{in}$	data input for ConvTransposed1d
`Z`	Z	matrix	$N \times C_{out} \times W_{out}$	features after ConvTransposed1d
`dLdZ`	$\partial L/\partial Z$	matrix	$N \times C_{out} \times W_{out}$	how changes in outputs affect loss
`dLdA`	$\partial L/\partial A$	matrix	$N \times C_{in} \times W_{in}$	how changes in inputs affect loss

5.5.2 ConvTranspose2d

Similar to ConvTranspose1d, ConvTranspose2d is a combination of Upsample2d and convolution with stride 1. Please consider the Figure 20 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.

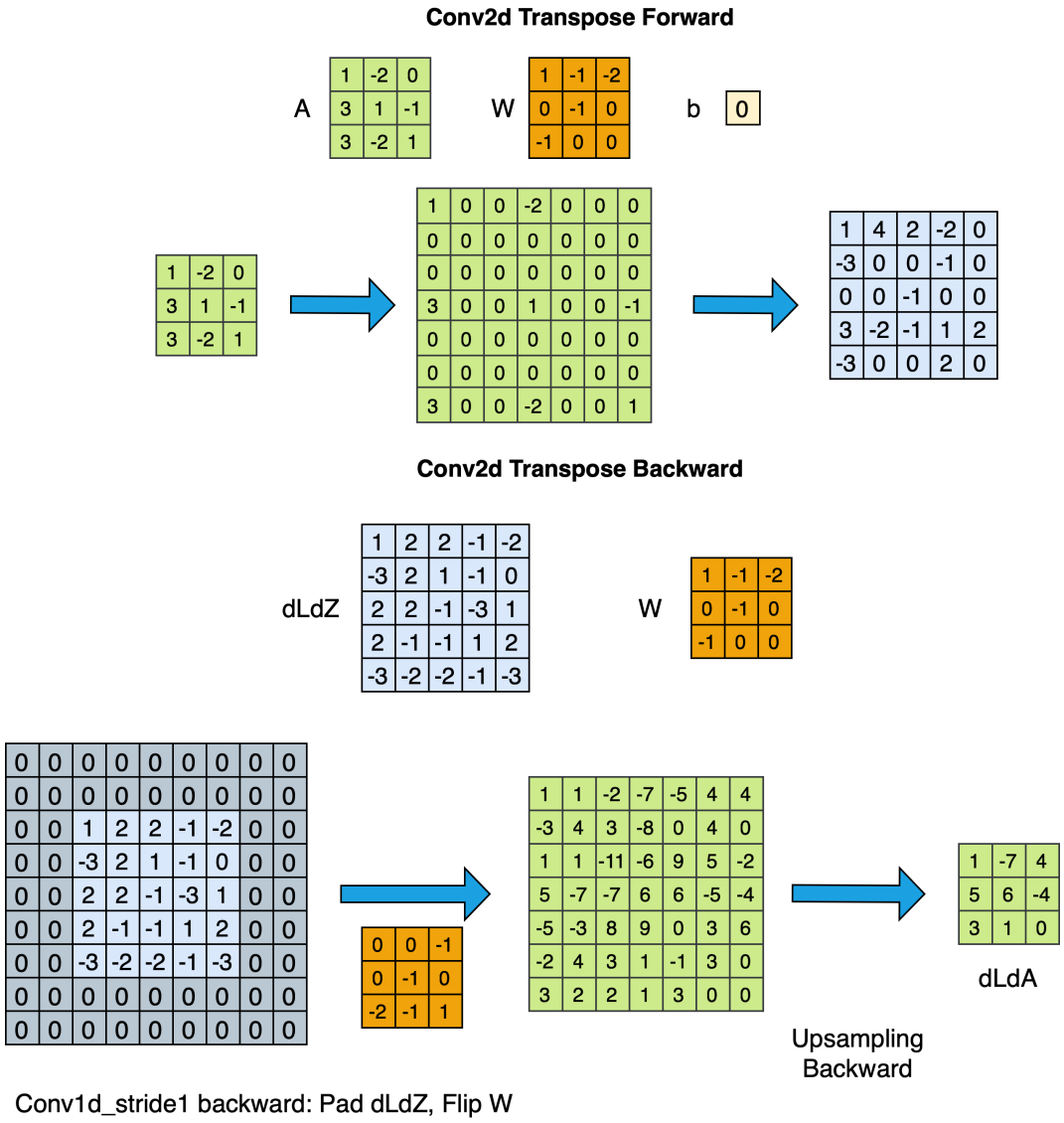


Figure 20: 2d Transpose Convolution

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.

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

Table 10: ConvTranspose2d Class Components

Code Name	**Math**	**Type**	**Shape**	**Meaning**
`upsampling_factor`	k	scalar	-	upsampling factor
`A`	A	matrix	$N \times C_{in} \times H_{in} \times W_{in}$	data input for ConvTranspose2d
`Z`	Z	matrix	$N \times C_{out} \times H_{out} \times W_{out}$	features after ConvTranspose2d
`dLdZ`	$\partial L/\partial Z$	matrix	$N \times C_{out} \times H_{out} \times W_{out}$	how changes in outputs affect loss
`dLdA`	$\partial L/\partial A$	matrix	$N \times C_{in} \times H_{in} \times W_{in}$	how changes in inputs affect loss

6 Pooling

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

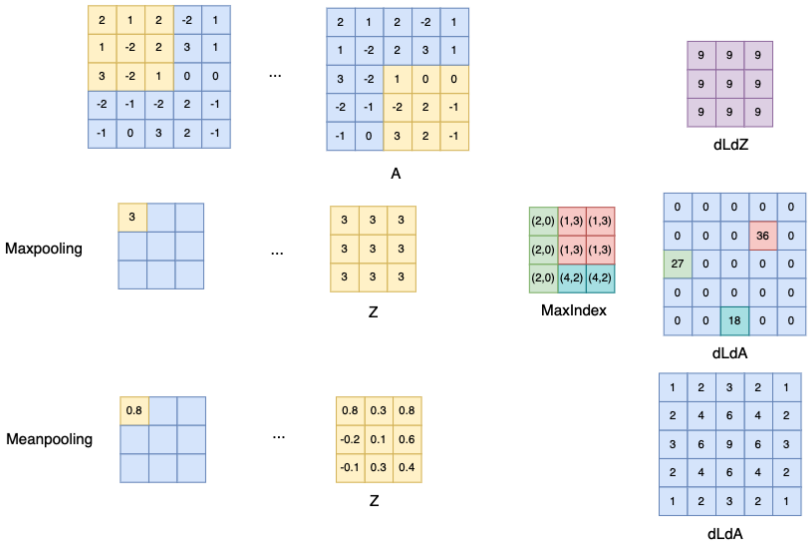


Figure 21: Pooling2d Example

Please complete the following classes in `mytorch/pool.py`
Max pooling selects the largest from a pool of elements and is performed by "scanning" the input. Implement both forward and backward methods 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 21.

Mean pooling takes the arithmetic mean of elements and is performed by "scanning" the input. Implement both forward and backward methods for 1d and 2d mean pooling.

6.1 MaxPool2d_stride1 and MeanPool2d_stride1 [20 Points]

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 $\frac{1}{K}$ where K is `filter_size`.
Hint: You might want to store the indices of the Max pool in `forward`.
Bigger Hint: You might want to look into NumPy's unravel index for Max pool

6.2 MaxPool2d and MeanPool2d [10 Points]

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

Table 11: Pooling Class Components

Code Name	Math	Type	Shape	Meaning
`stride`	*stride*	scalar	-	stride
`kernel_size`	K	scalar	-	kernel_size
`A`	A	matrix	$N \times C_{in} \times H_{in} \times W_{in}$	data input for pooling
`Z`	Z	matrix	$N \times C_{in} \times H_{out} \times W_{out}$	features after pooling
`dLdZ`	$\partial L/\partial Z$	matrix	$N \times C_{in} \times H_{out} \times W_{out}$	how changes in outputs affect loss
`dLdA`	$\partial L/\partial A$	matrix	$N \times C_{in} \times H_{in} \times W_{in}$	how changes in inputs affect loss

7 Flatten layer

In `flatten.py`, complete `Flatten()`[5].

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.

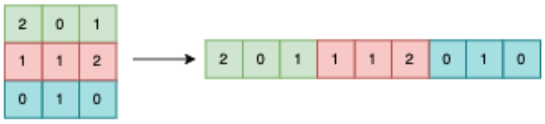


Figure 22: Flatten Forward Example

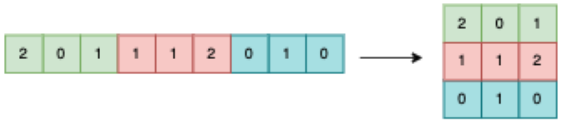


Figure 23: Flatten Backward Example

Table 12: Flatten Class Components

Code Name	Math	Type	Shape	Meaning
`A`	A	matrix	$N \times C_{in} \times W_{in}$	data input for Flatten
`Z`	Z	matrix	$N \times C_{in} \times W_{in}$	features after Flatten
`dLdZ`	$\partial L/\partial Z$	matrix	$N \times (C_{in} * W_{in})$	how changes in outputs affect loss
`dLdA`	$\partial L/\partial A$	matrix	$N \times C_{in} \times W_{in}$	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 just subcase of reshaping.

[5]No mark is allocated in this section, but you will need to implement it correctly for the next part.

8 Converting Scanning MLPs to CNNs

Make sure you have copied linear.py, activation.py, loss.py from HW1P1. Future sections will re-use code from these files.

8.1 A Simplified Example for CNN as scanning by MLPs

Consider a 108 x 1 (108 time steps, with 1 dimensional vectors at each time).

and a 1-D CNN model (1D is used because illustration is easier) with the following architecture:

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

This can be visualized by the figure below:

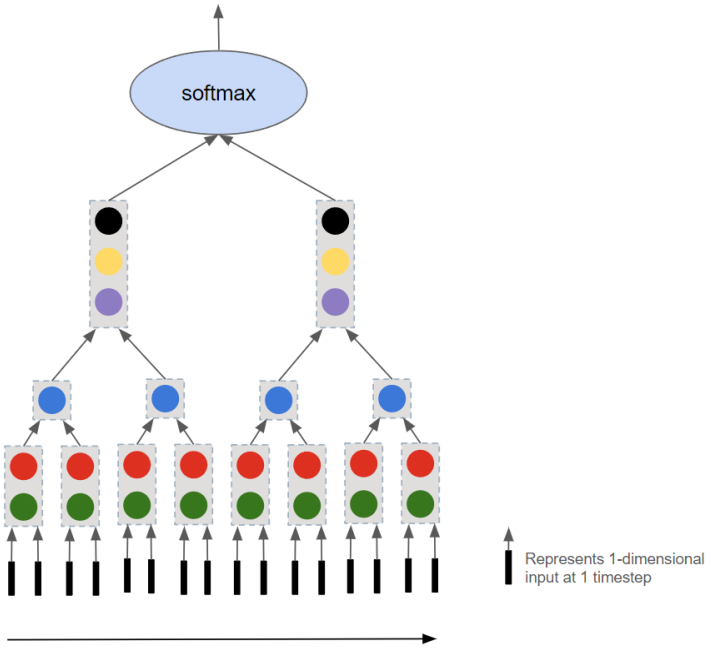


Figure 24

A few notes before we move on:

- The little black bars at the bottom represent the sequence of input vectors. Each vector is 1-dimensional as specified earlier (but this can be generalized to any dimension input).
- Additionally, there would be many MANY more arrows in this diagram i.e. each input vector should be directed into BOTH the red and green neurons in the first layer. EACH output from the red and green neurons should be input into the blue neuron in the 2nd layer, etc. (don't even get us started with the third layer). We simplified the arrows for your viewing pleasure :)
- We will trust that you understand the true direction of all the data flow (if you are confused, please refer back to lecture). We will simplify A LOT of the arrows to make the graphics easier to digest :D

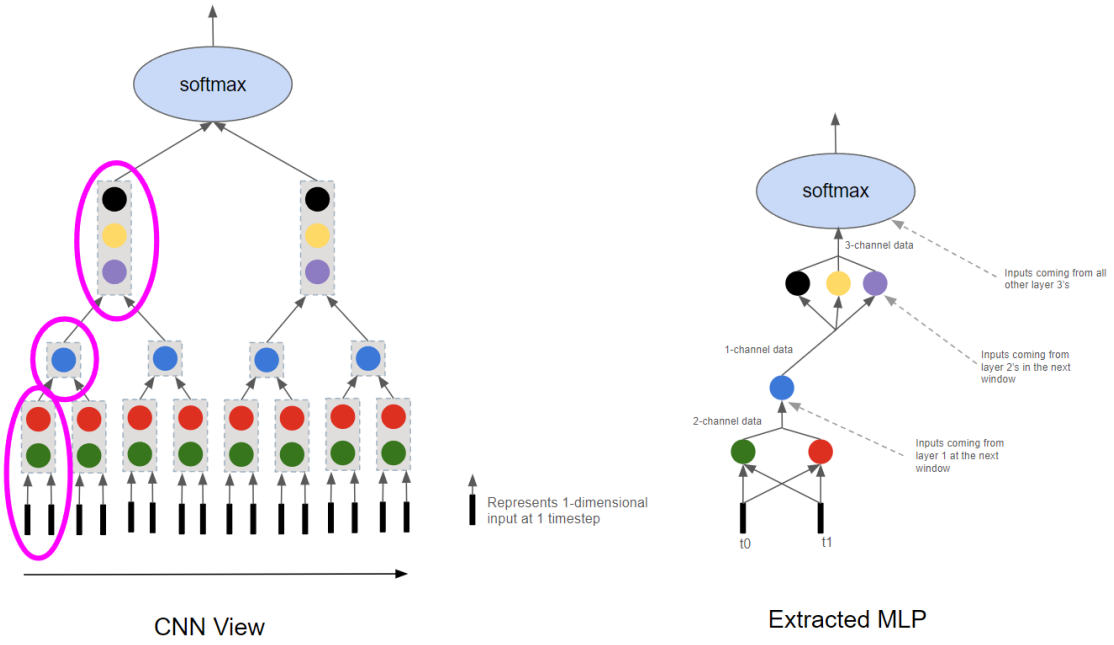


Figure 25

Important note!!!!
(As can observed from implementing your convolutional layers) conv layers process the slices of the input data in parallel. For instance, in the first layer, the red and green neurons operate on all the slices of input vectors almost simultaneously, and their outputs are fed into the next layer. We want to make sure you fully understand that each layer is not "waiting for the next slice of data to be processed," even if the vectors are separated by "time step".
Another note is that each neuron is a "filter" trying to "filter out its own desired pattern." Thus, each neuron will produce its own channel of data indicated how likely its desired pattern is present. In our example, our first layer's red and green neurons output 1 channel each, and the 2 channels are combined into a single output. The next conv layer will take in **kernel width** of these outputs. The next layer's number of **in channels** will also need to be equal to the current layer's **out channels** (also equal to the number of our filters/neurons) to process the output data.

Let's take a look at the Extracted MLP layer-by-layer in figure 25:

- Layer 1 (red & green neurons) (2 filters of kernel width 2, with stride 2):
  For each neuron, the **kernel width** (2) is the number of input vectors processed at once. The **input channels** (1) is the dimension of our input vectors. You will also notice that since we have 2 neurons (aka. 2 filters), our **output channels** will be 2.[6]
- Layer 2 (blue neuron) (1 filter of kernel width 2, with stride 2):
  This layer's **kernel width** (2) is the number layer 1 outputs that we will take in at once. The **input channels** (2) is the number of output channels from the previous layer. We have 1 neuron (aka 1 filter), and hence 1 **output channel**
- Layer 3 (black, yellow, & purple) (3 filters of kernel width 2, with stride 2):
  Similar to layer 2, this layer's **kernel width** (2) is the number layer 2 outputs that each neuron will take in at once. The **input channels** is the number of output channels from the previous layer. We have 1 neuron (aka 1 filter), and hence 1 **output channel**

[6]Conversely, this means that our specified output channels is also the number of filters/neurons we have in this layer.

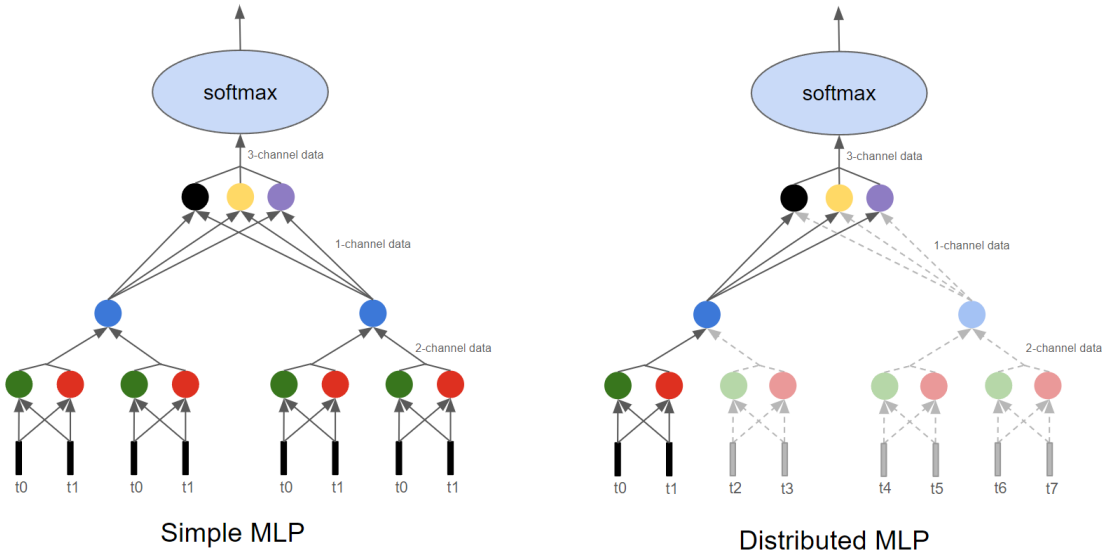


Figure 26

Let's take a step back out and complete the rest of our MLP. The "Simple Scanning MLP" on the left has three layers. If we naively count, we can see the first layer (in total) has 8 neurons, the second has 2 and the third has 3. So, if we implement our CNN naively, we can essentially mimic our CNN with an MLP that has 13 neurons.

Taking a look at the right "Distributed MLP" image, we can see more clearly that not all of the neurons are necessary. In the first layer, it is the same two neurons repeatedly operating on all the slices of our input data, and although the MLP has 13 neurons, it only has 6 unique neurons (this means 6 unique filters that need their weights to be adjusted to detect their desired patterns). As a result, we say that the 13 neurons "share 6 shared sets of parameters".

8.2 CNN as a Simple Scanning MLP [5 Points]

READ BEFORE PROCEEDING: Make sure you understand all of section 8.1 before proceeding. We also highly recommend drawing out the MLP and seeing how you can group neurons into a CNN
TAs will NOT help you debug your code until we have seen you attempt to draw out a diagram. All piazza posts about this problem must start with the phrase "I have read section 8.1"
In `models/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).
- The MLP evaluates **8 contiguous input vectors** at a time after flattening[7] them
- 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. (The input dimension to the first layer will be 8 * 24 neurons to process 8 vectors of dimension 24). Each neuron uses a `ReLU` activation except 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 `models/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-channel 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.**

Design 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`.
- In the `init_weights` method, you will receive `w1, w2,` and `w3` corresponding to layer 1, layer 2, layer 3 of the MLP. You must convert the initial weight matrices into the weights of the Conv1d_stride1 layers. (See code for TODO statements on how to do this)
- Use pdb to help you debug, by printing out what the initial input to this method is.

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.

[7] Remember that MLP only has linear layers, which are 1D.

8.3 CNN as a Distributed Scanning MLP [10 Points]

READ BEFORE PROCEEDING:
TAs will NOT help you debug your code until we have seen you attempt to draw out a diagram. All piazza posts about this problem must start with the phrase "I have read section 8.1"

In `models/mlp_scan.py` complete the `CNN_DistributedScanningMLP`. This section of the homework is very similar to Complete `CNN_SimpleScanningMLP`, except that the MLP provided to you is a shared-parameter network that captures a distributed representation of the input. (If you are struggling to understand this prolbem, make sure you try to fully understand section 8.1 and/or check out section 10.3 for additional help videos, or come to OH :)

Architecture details:

- The MLP 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 **??**. 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).

Design 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. (See code for TODO statements on how to do this)
- You will also need to slice the weights after reshaping them to reduce it to only the shared parameters.

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.

9 Build a CNN model [5 Points]

Finally, in `models/cnn.py`, implement a CNN model.

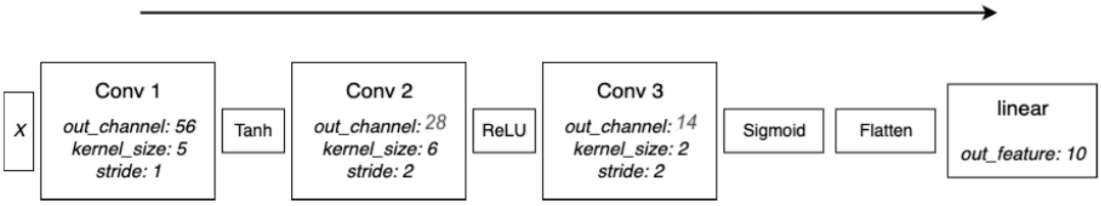


Figure 27: CNN Architecture to implement.

- 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.
- In activation.py, use NumPy's built-in `numpy.tanh()` function for `Tanh` class' forward function.
- 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!!

10 Appendix

10.1 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.

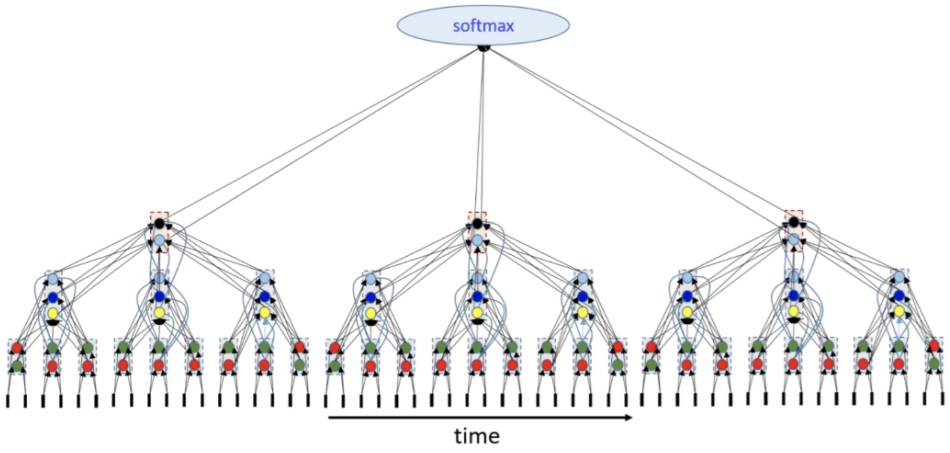


Figure 28: Scanning MLP Figure 1

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.

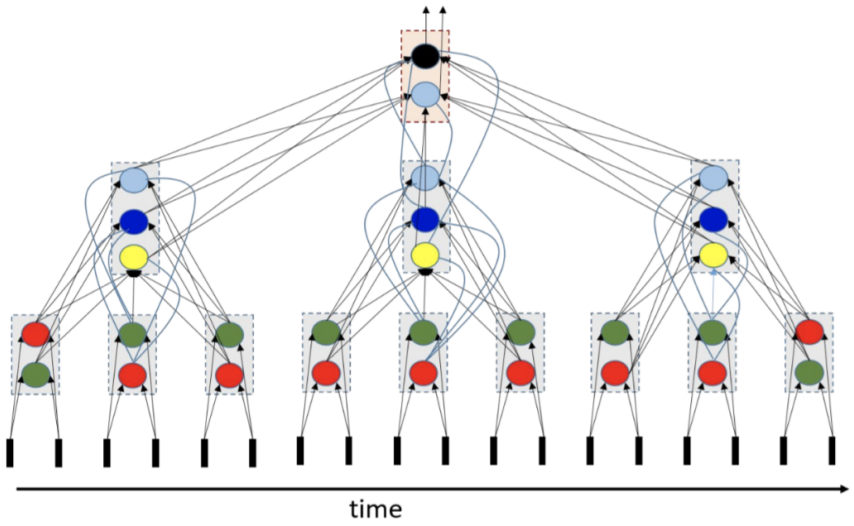


Figure 29: Scanning MLP Figure 2

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

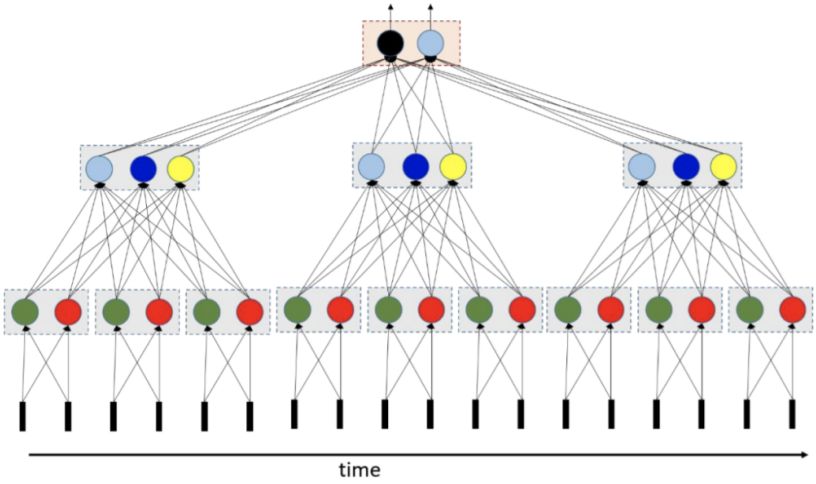


Figure 30: Scanning MLP Figure 3

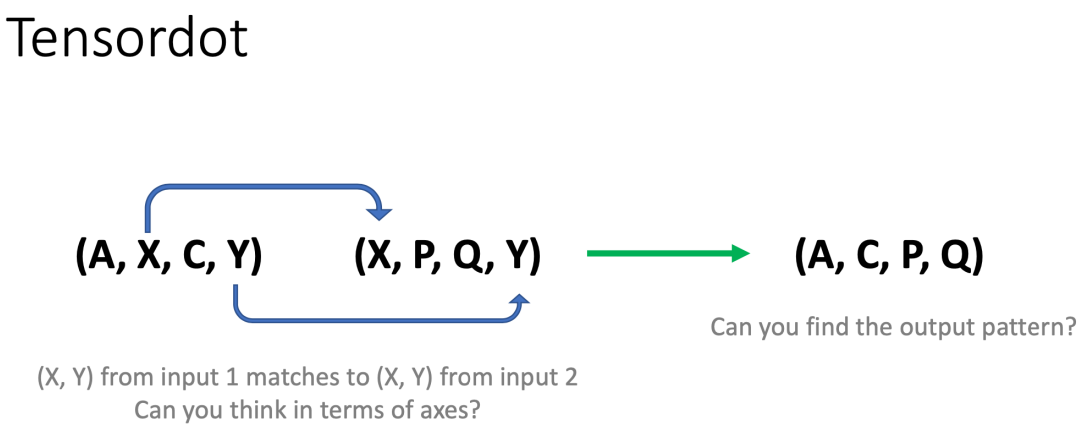
10.2 Numpy TensorDot

TensorDot is an amazing tool by numpy which allows you to simplify convolutions. It helps you to reduce the number of loops in your code.

So as the name suggests, tensordot is a function which helps you do a dot product. In the most basic level, it helps you do an elementwise multiplication and summation. It does not do the whole convolution, rather just a step in convolution. Where do you think there is an element wise multiplication and summation in convolution? When the kernel is over a patch/segment of the input image, we do an element-wise multiplication and summation to get a single number as output. After this operation, the kernel strides and does the same operation again. So tensordot does not make you escape a loop. It just reduces the number of loops.

So what constrain do you need 2 arrays to satisfy inorder to do a tensordot? Since its first step is an element-wise multiplication, the shapes of the 2 arrays should be the same. For example, you cannot do an element wise multiplication between a 3x3 array and a 4x4 array.

This begs the question, should the complete shape of the 2 arrays which we use in tensordot be the same? Not required. This is why we specify the axes parameter in tensordot. You can think of a big array as an array of small subarrays. The subarrays can have "matching shapes" to do a tensordot (or you have to make it match in convolution between a big input and a small kernel. Are you really using a big input in each step? Think about it). After doing a tensordot operation, the output which you get will be of shape consisting the unmatched axes in the same order. The below figure tries to explain this.



You could also refer to this blog which has a more detailed explanation of tensordot as well.

10.3 Additional Resources and Tips

- **HW2P1 — Helper Videos -**
  Resampling
  Convolution
  Pooling
  MLP Scanning

- **Tips for Tensordot and Common Mistakes -**
  - Open a new notebook, initialise random arrays with some matching axes and try doing tensordot.

Print the shape of the outputs

- As it was mentioned, tensordot does not do a convolution operation, rather just a step. Think about where this step is in the convolution operation
- Before coding, print out the shapes of your input, kernel and the expected output. Think about what you can do with tensordot along with the input and kernel to get the output.
- You can do a tensordot between input and kernel or kernel and input. The order depends on what shape your output needs to be.
- For a **conv1d_stride1()**, you only need 1 for loop along with tensordot and for **conv2d_stride1()**, you need 2. If you are using more, then your implementation of tensordot is wrong (You may get the correct answer in the more loopy way but you dont learn to use tensordot properly)
- **With tensordot, you dont need explicit broadcasting (important)**

- **Little Einsum (Tensordot Alternative) -**
  If you're feeling particularly saucy, you can also use this function called "einsum", that stands for Einstein Summation. It's more of a general case tensordot, though its syntax can be a bit hard to get immediately used to, but once you have it, it's rather intuitive.

  Here is an example:

  A: (d1,d2,d3)

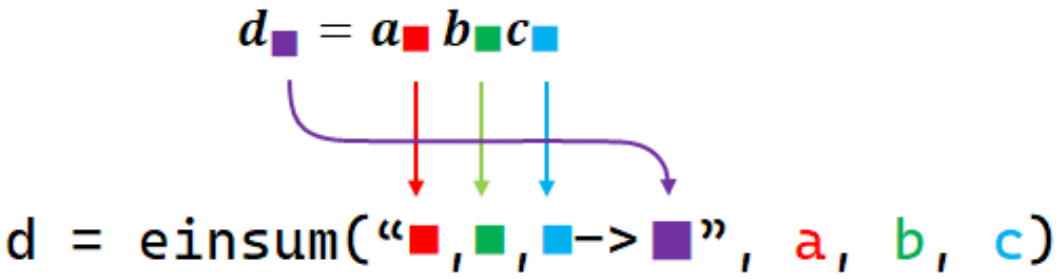
  B: (d1,d2,d4)

  Our goal is to multiply the two tensors along their second and third dimensions. The call to einsum would be the following:

  C = np.einsum("abc, abd → acd", A,B)

  The way this would be interpreted is: There are two inputs of dimensions (a,b,c) and (a,b,d), the output required is of shape (a,c,d). This must mean that the second dimension in both inputs has to be collapsed. To do this, we must transpose input_1 along dimensions (1,2), and conduct a batch-multiply with input_2, giving the desired shape.

  It's called einstein summation, precisely because it will figure out how to do what you need it to do without you having to explicitly state it.



Warning: It is easy to get undesired behavior in this function if not used carefully. Both einsum and tensordot are completely viable ways to go about solving the homework. Choose to your convenience and preference