HW2P1 Bootcamp

Convolutional Neural Networks with Numpy

John Liu, Michael Kireeff, Sardhishya Agrawal

Acknowledgment: ***Denish, Pavitra, Harshith, Shreyan, Aparajith for the slide content.***

Resampling

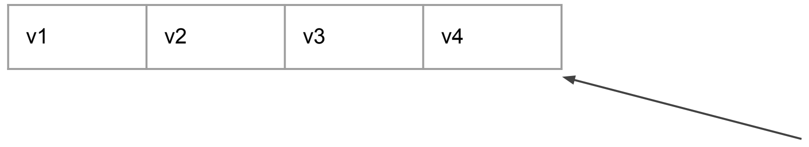
- For loop is not required in *python*
  - *Look up np.kron*
  - *Array slicing:* `[start:end:step]`
- Things to remember
  - Trying to compute the required shape while up sampling (some simple formula you can think of?)
  - Computing and storing the shape in forward.
    - This is because the gradient should be the same shape as the input.

Convolutions

- You can perform convolutions in 2 ways:
  - The Loopy way (Bad)
  - Tensordot (Good)
- The more for loops you use for your questions, the more time it takes to run.
- With tensordot, you don’t have to do all those broadcasting and everything given in the write-up

Dimensionality - 1D

Let's start with a 1-dimensional array



This is a 1d array of intensity values.
So, it's a 1d array of gray-scale pixels

Current dimensions:
(width)

Dimensionality - 2D

v1	v2	v3	v4
v5	v6	v7	v8
v9	v10	v11	v12
v13	v14	v15	v16

This is a 1d array of intensity values.
So, it’s a 1d array of gray-scale pixels

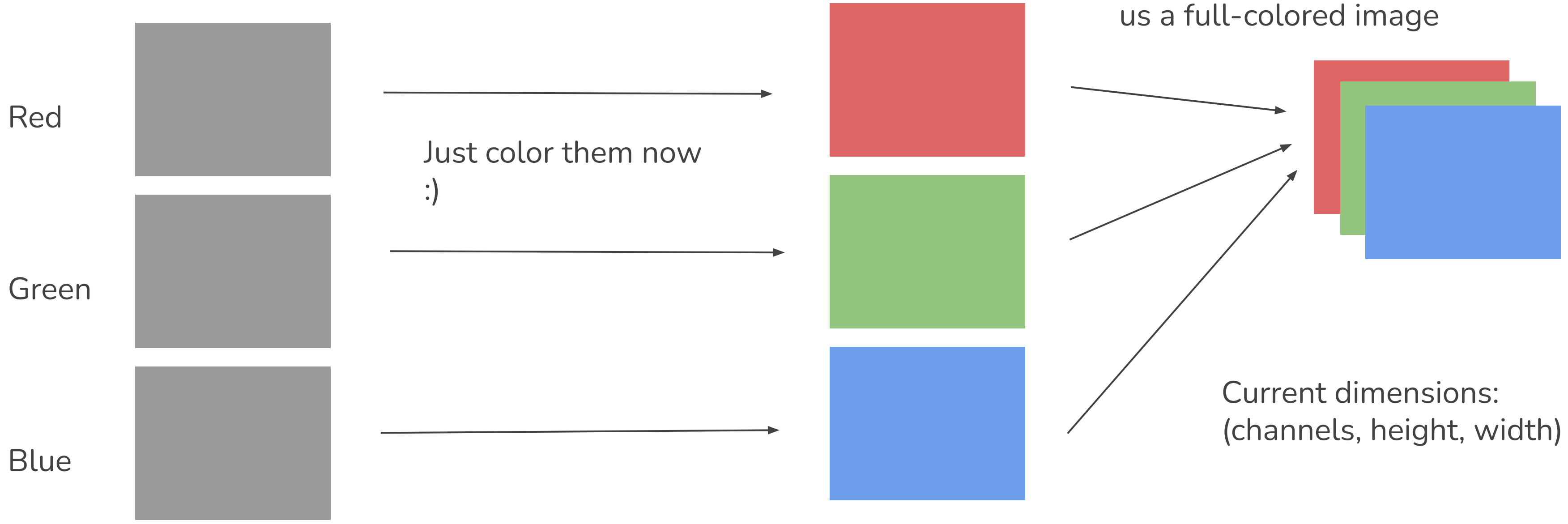
Let’s simplify the image going forward....

Current dimensions:
(height, width)

Dimensionality - 2D-RGB

Let’s have 3 blocks now,

- one representing Red light intensities
- one representing blue light intensities
- one representing Green light intensities



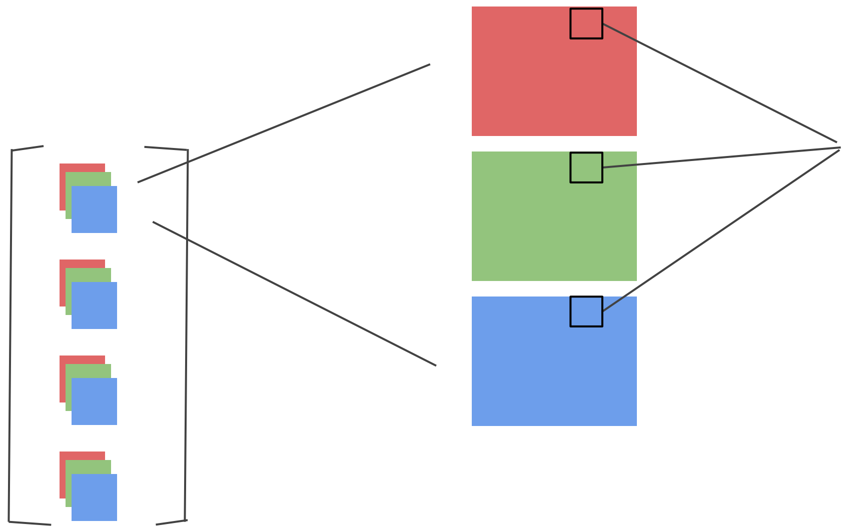
Just color them now :)

Stacking them will give us a full-colored image

Current dimensions: (channels, height, width)

Dimensionality - Batch of 2D RGB

Let's have a bunch of images for a batch now...



Apply a filters of (kernel_height x kernel_width)

Current dimensions:
(batch size, channels, height, width)

Tensordot

- Ref: https://numpy.org/doc/stable/reference/generated/numpy.tensordot.html
- Appendix of the write up has amazing documentation of it
- Don’t use for loops for convolution even though everything is given in the lecture slides
- Tensordot is faster and helps you (also TAs) to debug easily
- You only need ***1 for loop*** for conv1d and ***2 for loops*** for conv2d. If you are using more, then your implementation of *tensordot* is wrong even if you get the answer right

Tensordot

- Before starting ConvXd.py, open a notebook and try to understand tensordot with random examples
- Consider the shapes:
  - Input: X(A, B, C); Weight: W(P, Q, R)
  - You can do tensordot when you have matched shapes
  - If B = Q and C = R,
    - Tensordot(X, W, matched axes) -> Output(A, P)
    - You can think that the output shape will be the shape of the unmatched axes in that order
  - Make sure inputs (input and weight) to tensordot have some matching axes. Why do you need matching axes in convolution? (Hint: A filter only looks at a segment of input)
- Tip: Print shapes in your code to understand

Tensordot

$$\begin{tikzcd}(A, X, C, Y) \arrow[r, bend left] \arrow[r, bend right] & (X, P, Q, Y) \arrow[r] & (A, C, P, Q)\end{tikzcd}$$

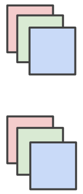
Can you find the output pattern?

(X, Y) from input 1 matches to (X, Y) from input 2

Can you think in terms of axes?

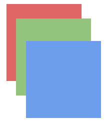
Should match all the axes that you think needs to be matched. Not restricted to 2 axes

Multiplying Two Arrays (A basic example of tensordot)



(filters, channels, height, width)

X



(channels, height, width)

= [f1 score, f2 score]

Resulting in output of dimension: (filters,).

- We took our two image filters,
- Applied our filters across each channel in the image,
- And summed up the responses to get a single score per filter.

- Let’s say our filters are looking for a cat and a dog
- Our resulting values indicate how strongly each filter detects a cat or a dog in the entire image.

```
np.tensordot(filters, images, axes=[[1, 2, 3][0, 1, 2]])
```

Match the dimensions channel to channel, height to height, width to width

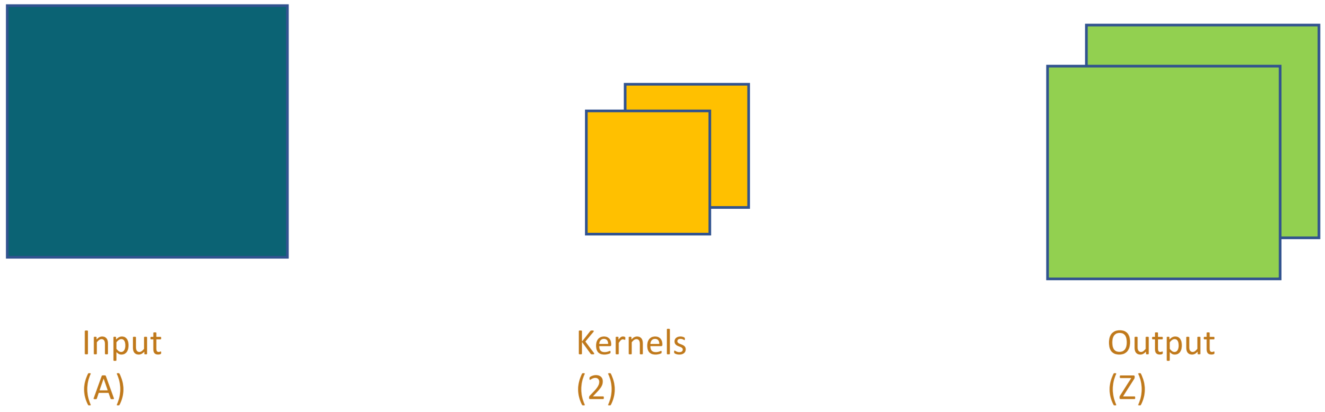
Conv1d to Conv2d

- Try to understand each step while coding conv1d
- Every step between Conv1d and Conv2d (forward and backward) are identical
- While transitioning from Conv1d to Conv2d, you just need to account for the extra dimension and do an **extra something**

Pooling

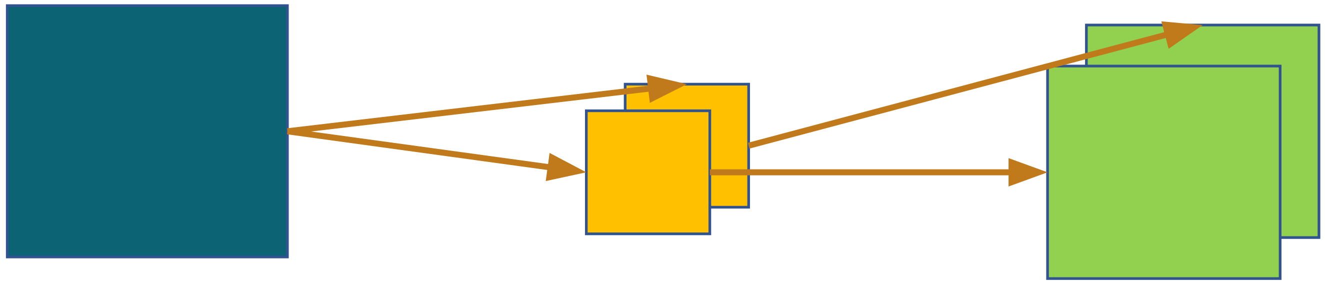
- Lectures have a basic pseudocode which can be developed
- You might need many loops for this task
    - Np.max and np.unravel_index might be useful if you want to reduce the number of loops
    - But multiple loops are acceptable for this particular task
- Backprop in both might is harder than forward, but if you know the concept behind it, it will not be that hard.
- Look at the write up for images.

Easy way to understand gradient propagation



We get 2 maps in backward for dLdZ. After some process for finding dLdA, you again get 2 maps. But A has 1 map and dLdA will also have the same shape. How to understand gradient propagation?

Easy way to understand gradient propagation



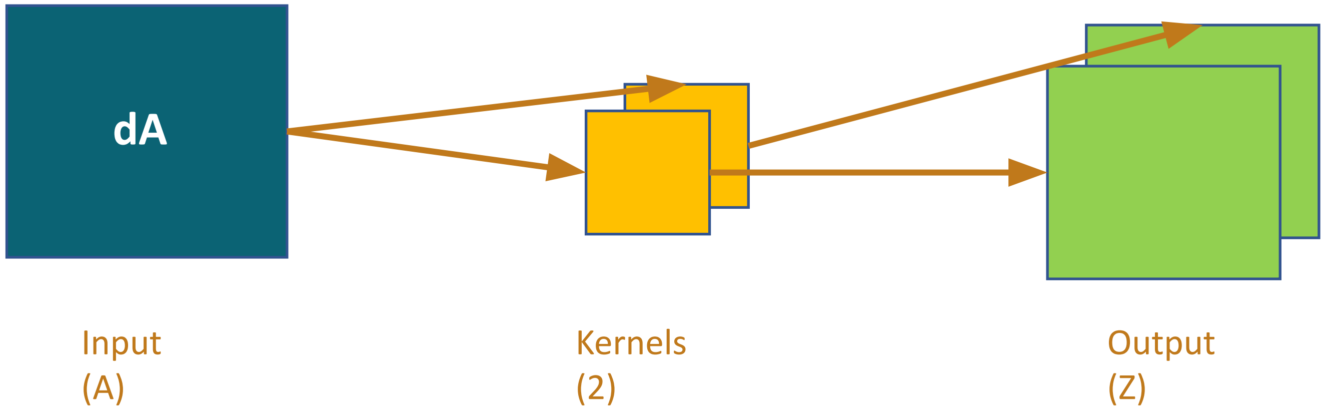
Input
(A)

Kernels
(2)

Output
(Z)

Draw the influence diagram.

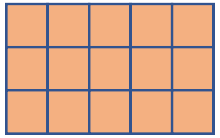
Easy way to understand gradient propagation



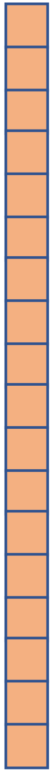
Any small change dA will cause a change in both maps of Z.

Scanning MLP

- Appendix of HW2P1
- Tips to understand better: Draw everything

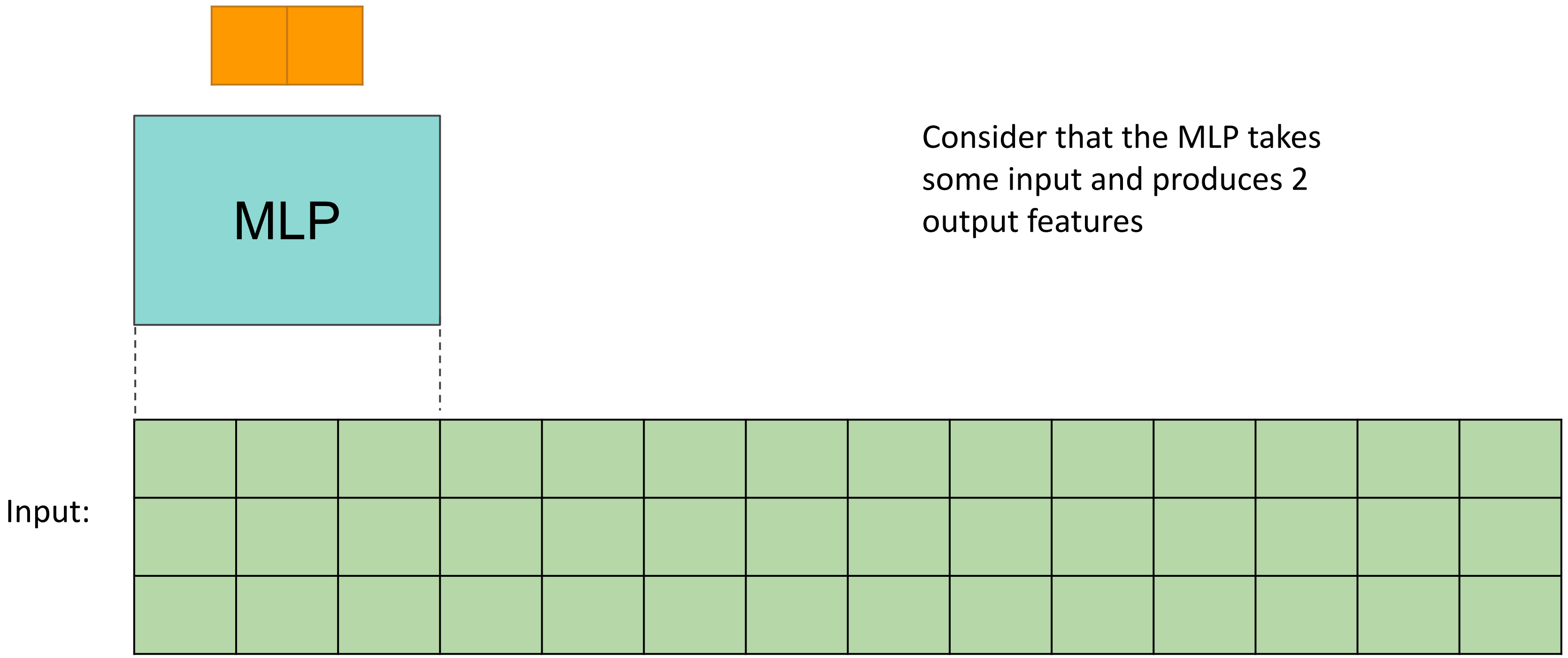


How Conv1d sees the input



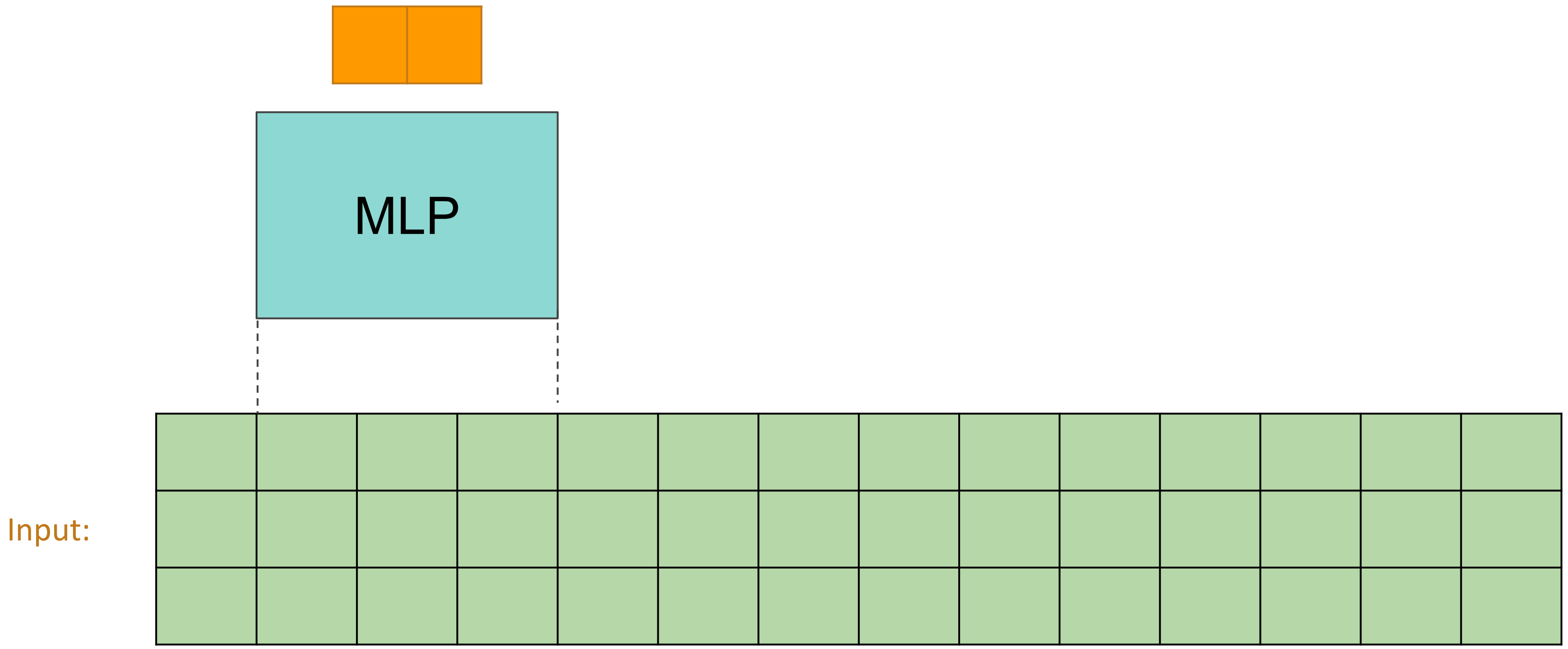
How Linear sees the input

Scanning MLP

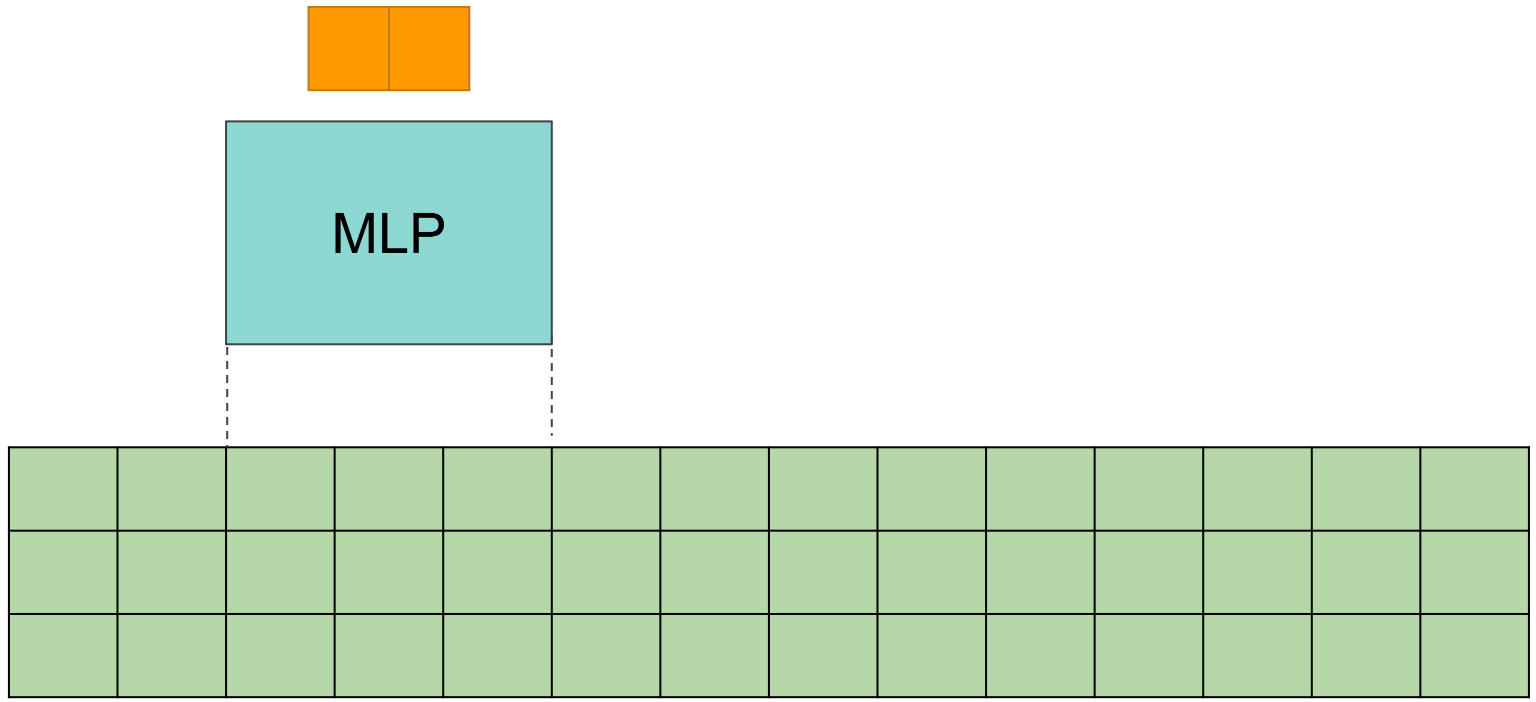


Consider that the MLP takes some input and produces 2 output features

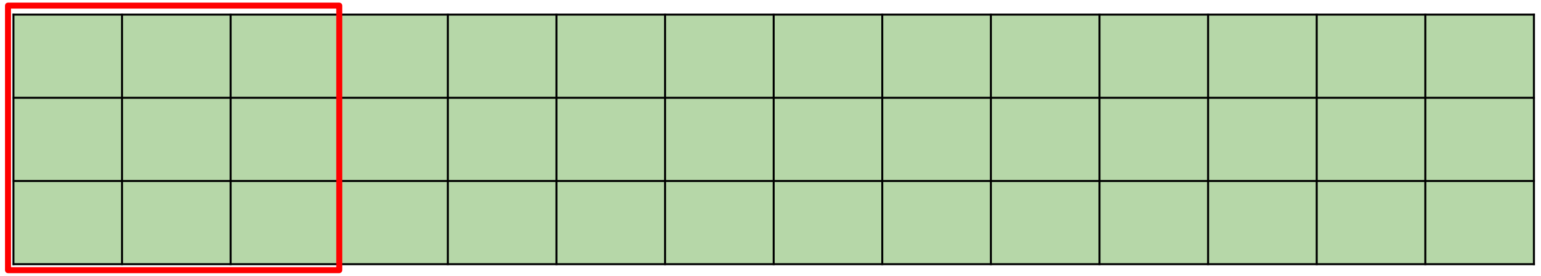
Scanning MLP



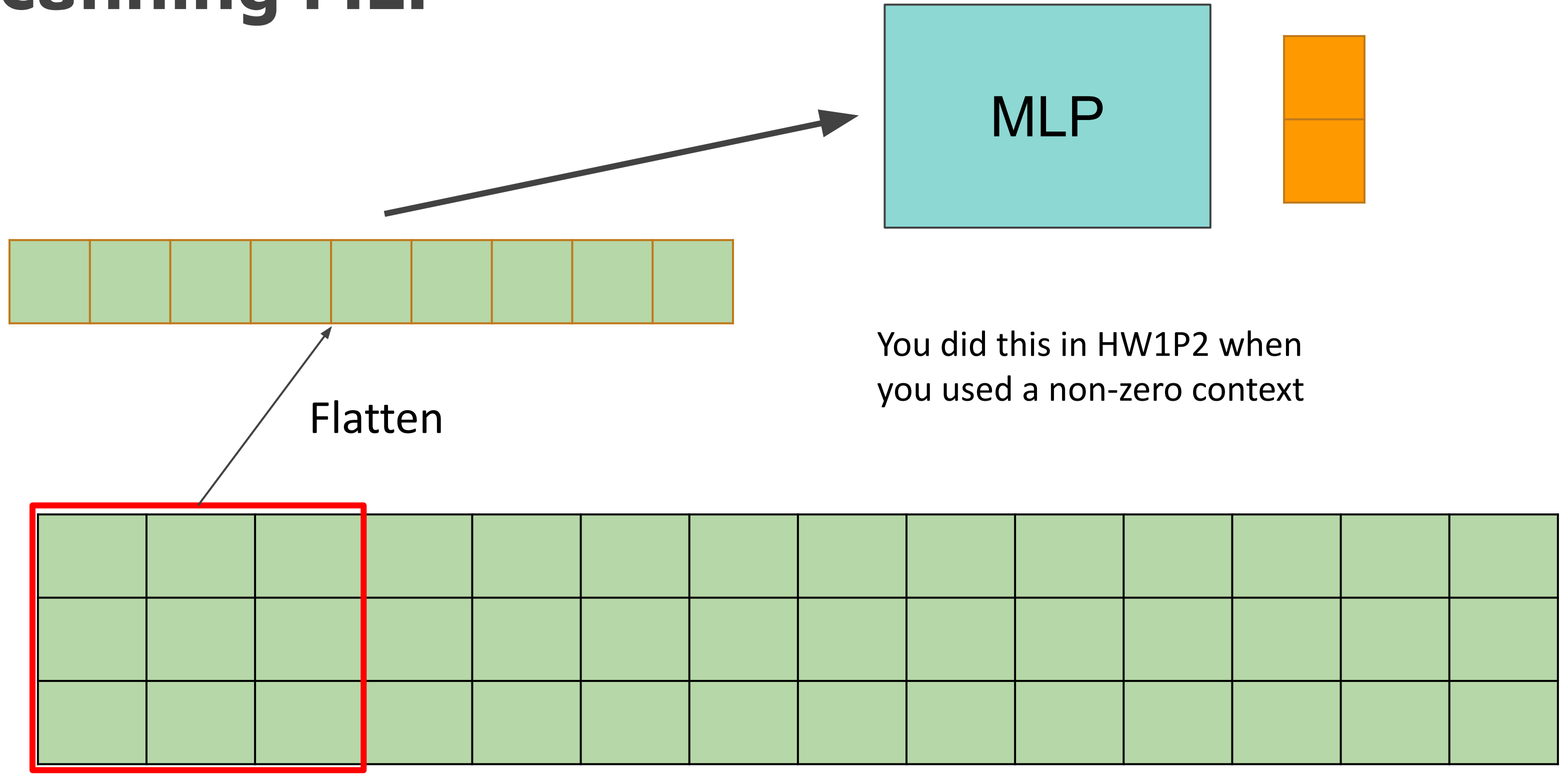
Scanning MLP



Scanning MLP

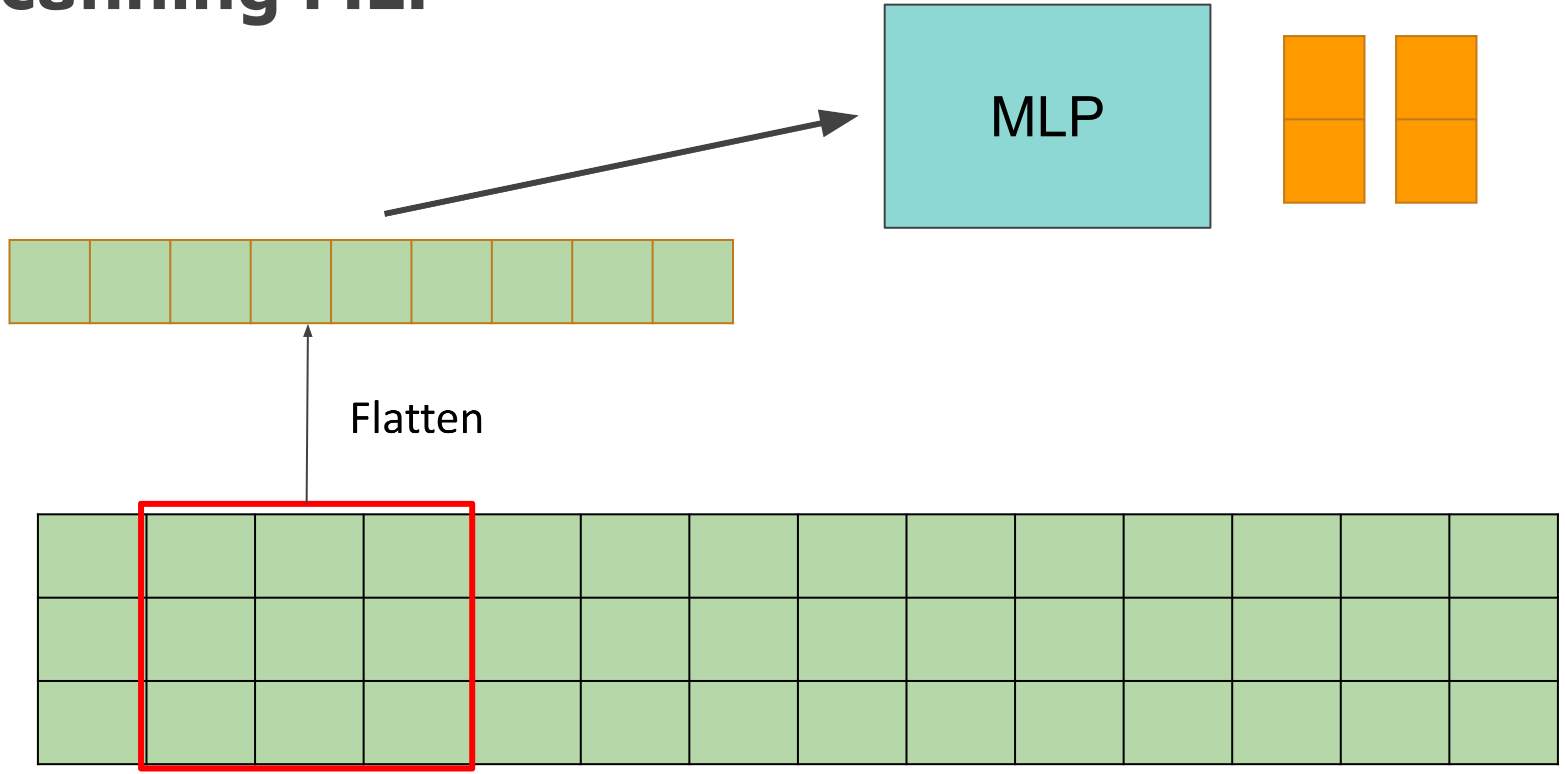

Kernel size=3

Scanning MLP

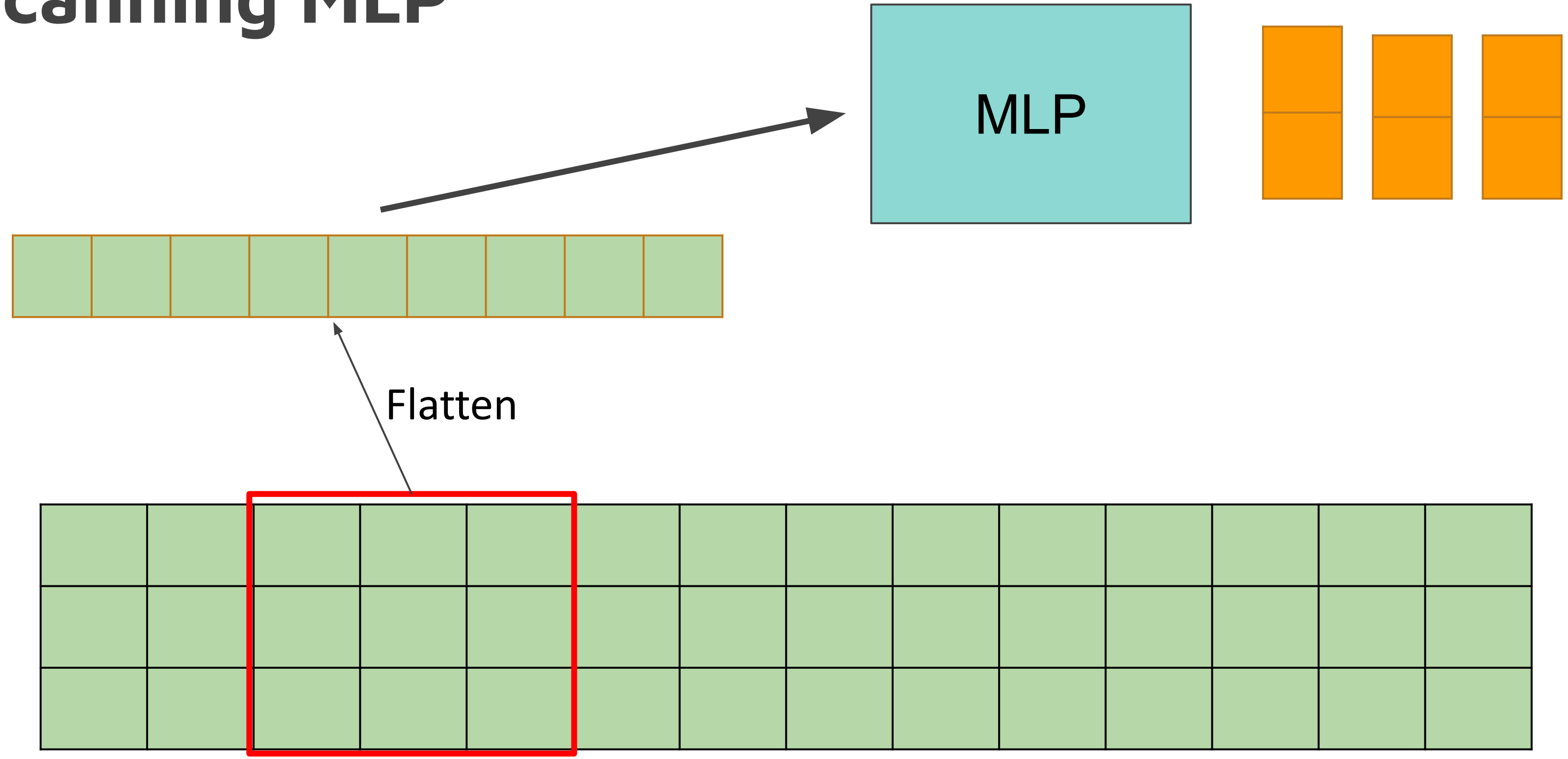


You did this in HW1P2 when you used a non-zero context

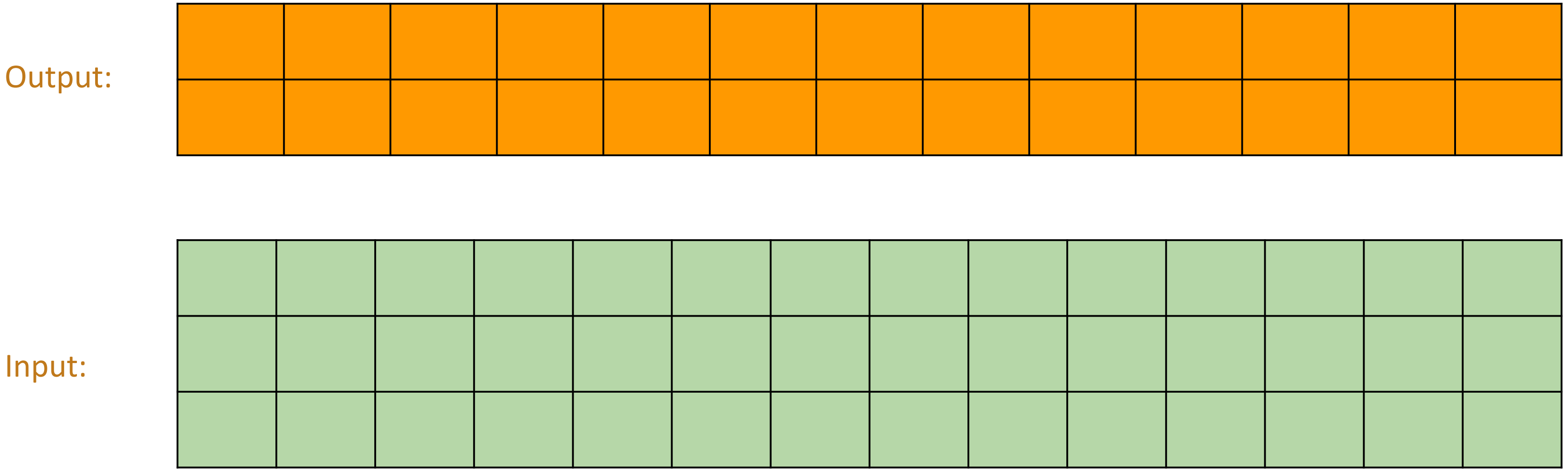
Scanning MLP



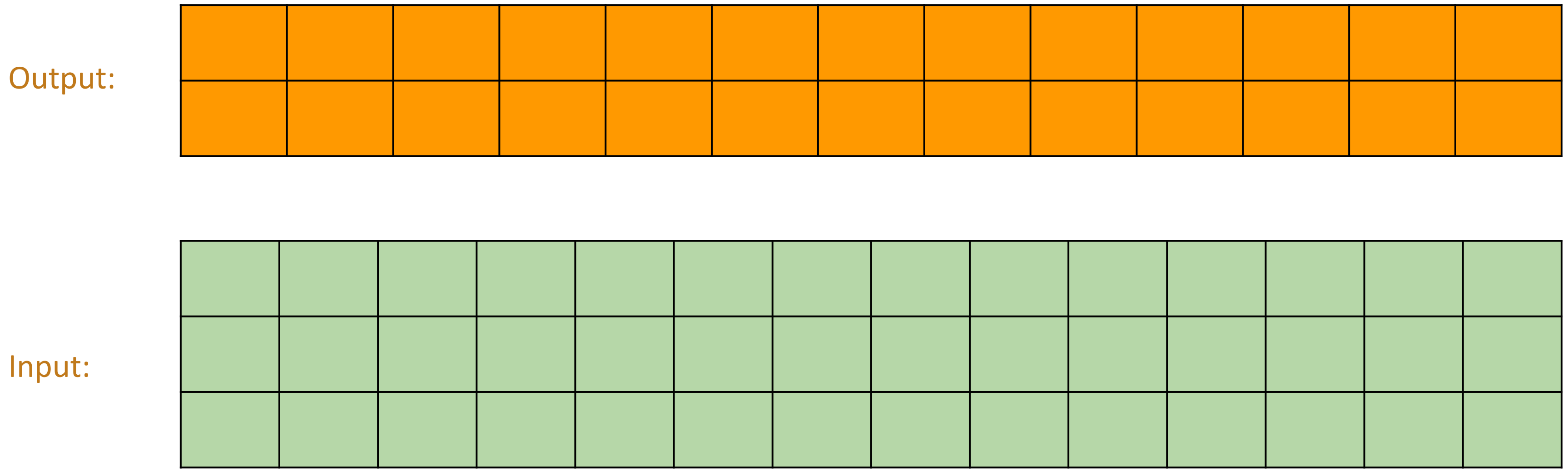
Scanning MLP



Scanning MLP



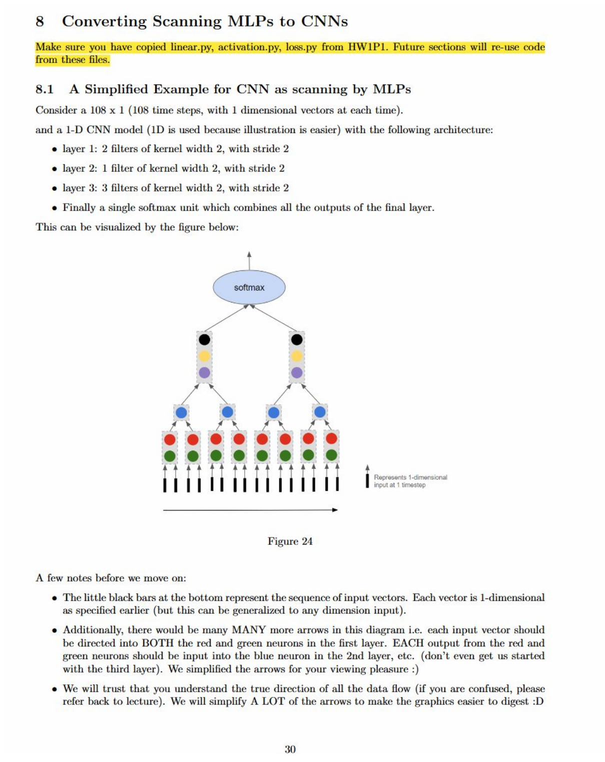
Scanning MLP



Which gives in_channels = 3, out_channels = 2, kernel_size = 3, stride = 1

We transformed ***Linear(9, 2)*** to ***Conv1d(3, 2, kernel_size= 3, stride= 1)***

Dive into the Writeup



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:

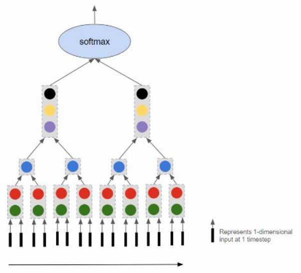
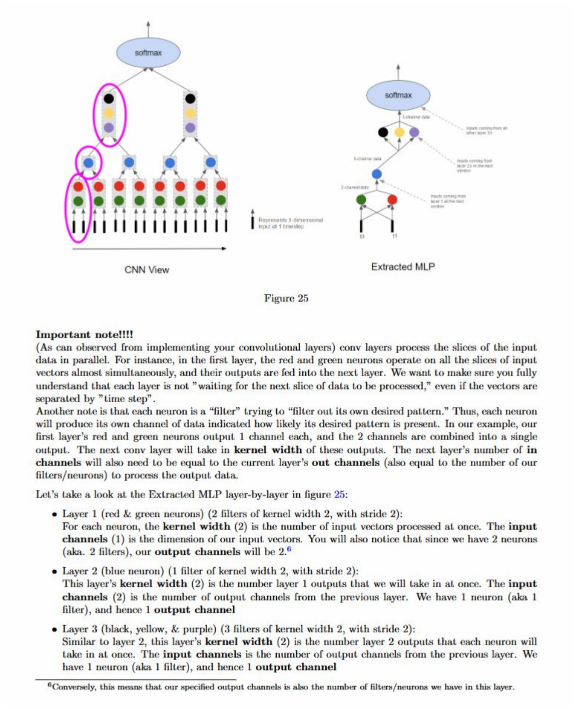


Figure 24

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

Dive into the Writeup



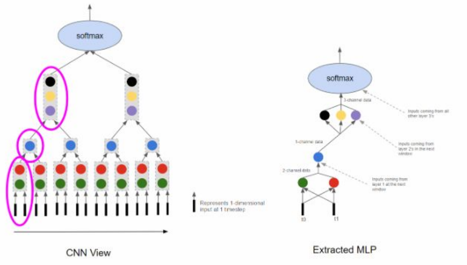

softmax
Represents 1-dimensional input at 1 timestep
CNN View
softmax
3-channel data
Inputs coming from all other layer 3's
1-channel data
Inputs coming from layer 2's in the next window
2-channel data
Inputs coming from layer 1 at the next window
t0
t1
Extracted MLP

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.

Dive into the Writeup

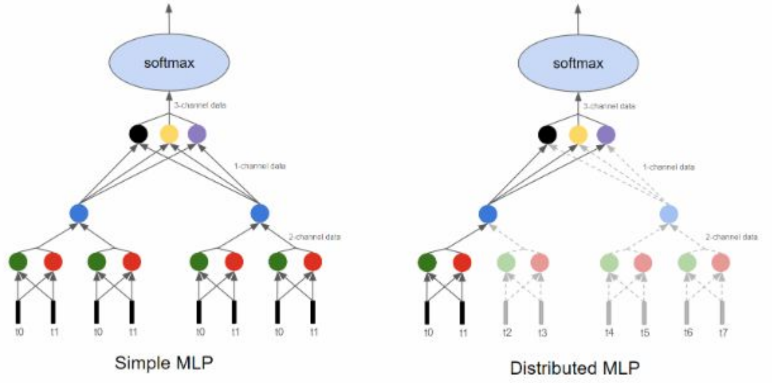


Figure 26

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".

CNN Model

- Just calling all the layers which you implemented previously
- Only thing to think about: Initialization size of the final Linear Layer?
- Errors which you may get:
    - If you have a closeness error (*true_divide error*), change to ***np.tanh()***

THANK YOU