Lecture 3: Convolutional Neural Networks

Michael Cochez

Deep Learning

dlvu.github.io

VRIJE UNIVERSITEIT AMSTERDAM

THE PLAN

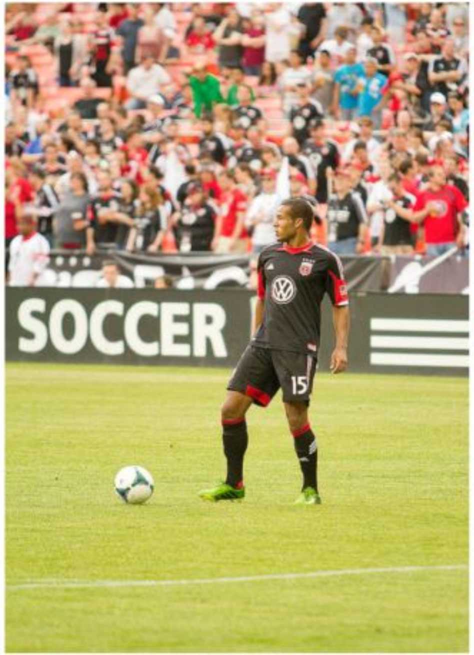
part 1: Introduction - why are convolutional architectures needed?

part 2: One-dimensional convolutional neural networks (conv1D)

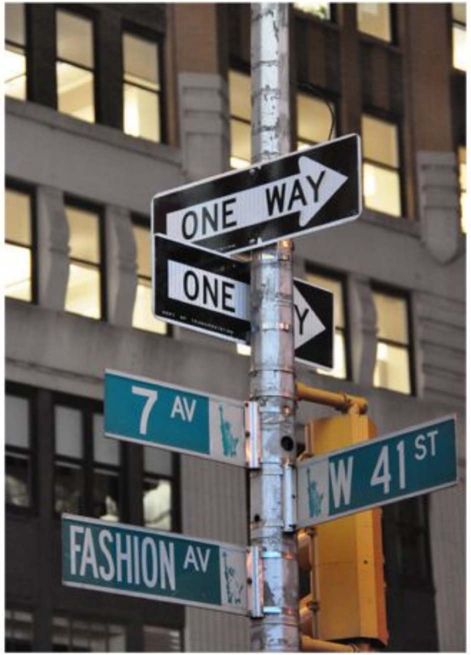
part 3: Two-dimensions and beyond (conv2D, conv3D, ...)

part 4: Example architecture

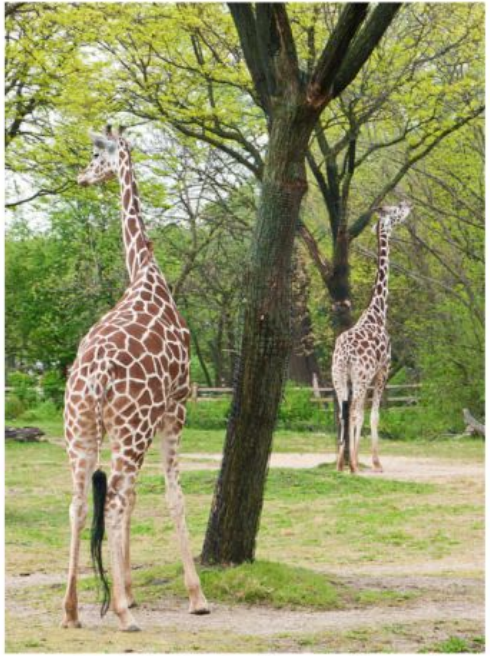
PART ONE: INTRODUCTION



a soccer player is kicking a soccer ball

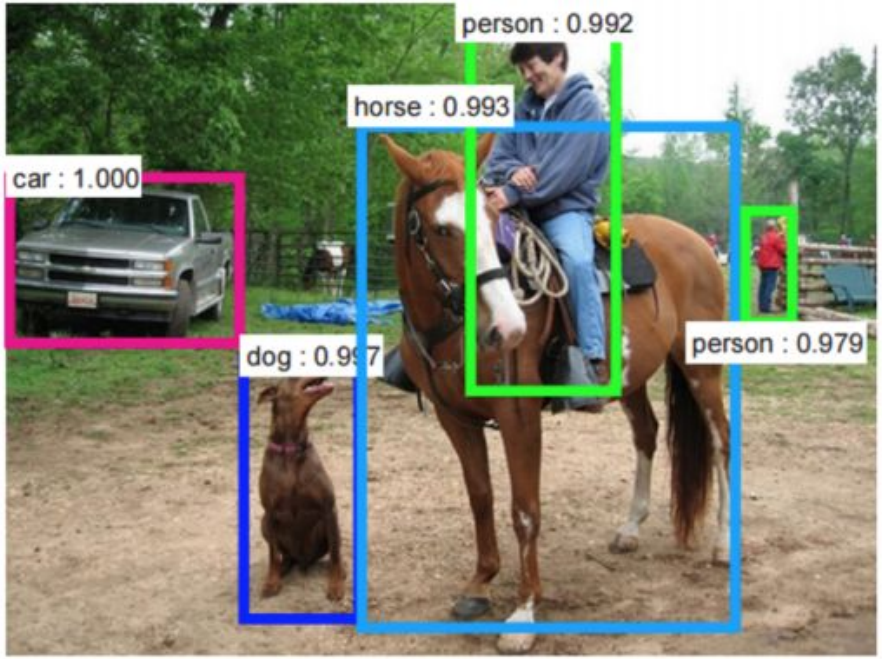


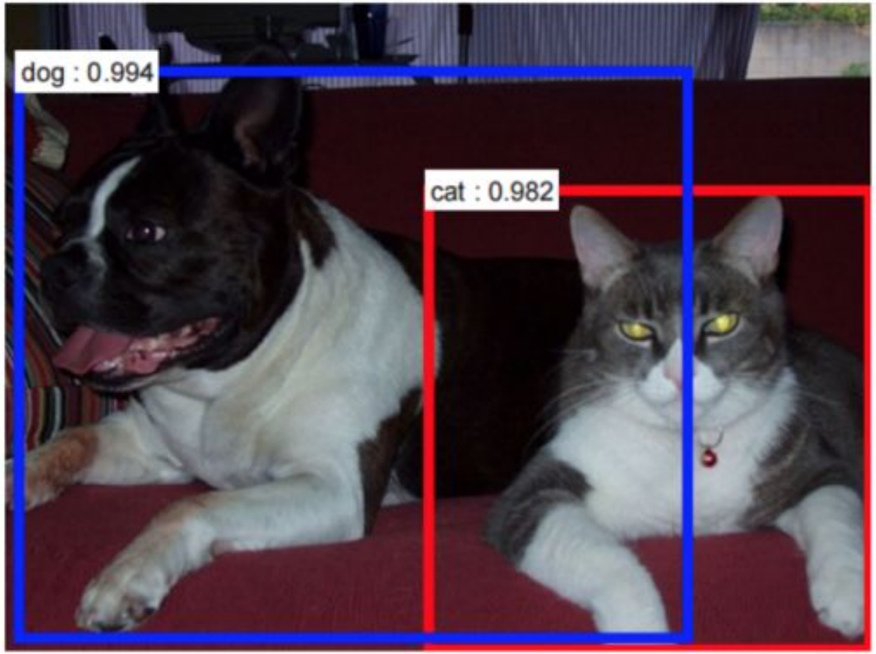
a street sign on a pole in front of a building



a couple of giraffe standing next to each other

source: https://cs.stanford.edu/people/karpathy/neuraltalk2/demo.html





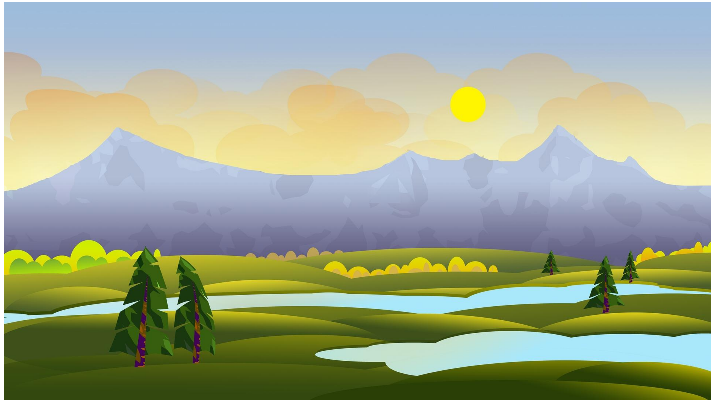
source: https://arxiv.org/pdf/1506.01497v3.pdf

INPUT DATA

- We need to get features!
- For tabular data, this is “simple”
- But what with more complex data?

INPUT DATA - IMAGES

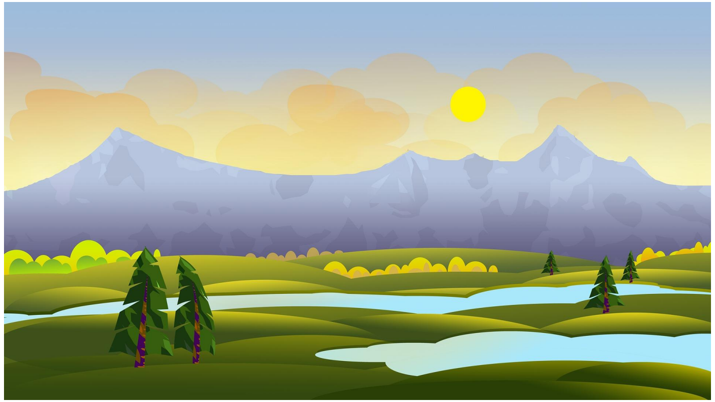
- Example

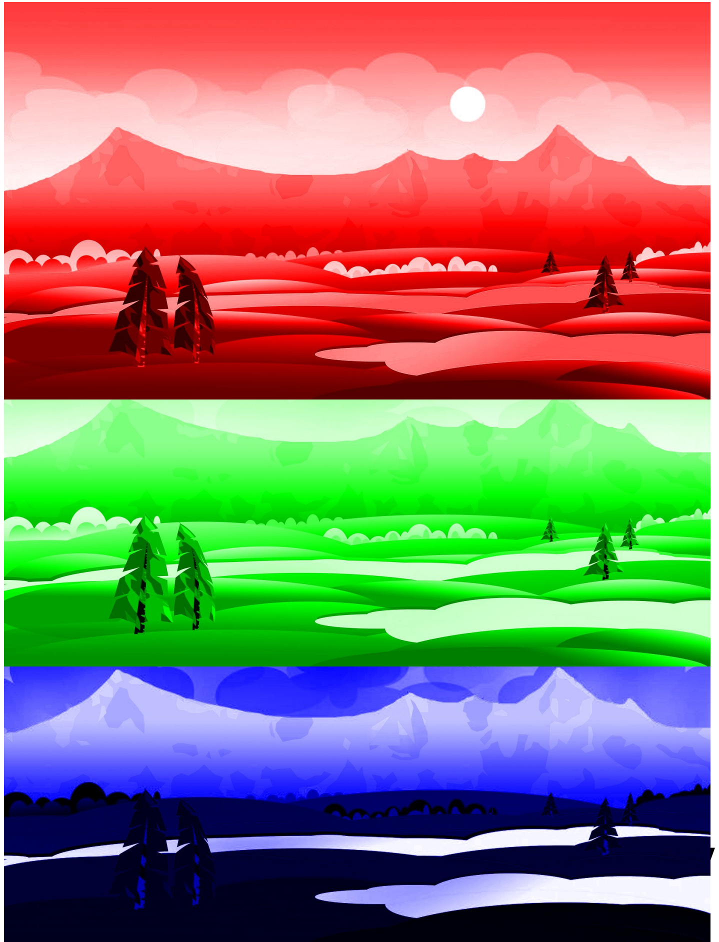


https://pixabay.com/nl/illustrations/vector-afbeelding-landschap-vector-3833815/

INPUT DATA - SOUND, IMAGES, VIDEO - ENCODING

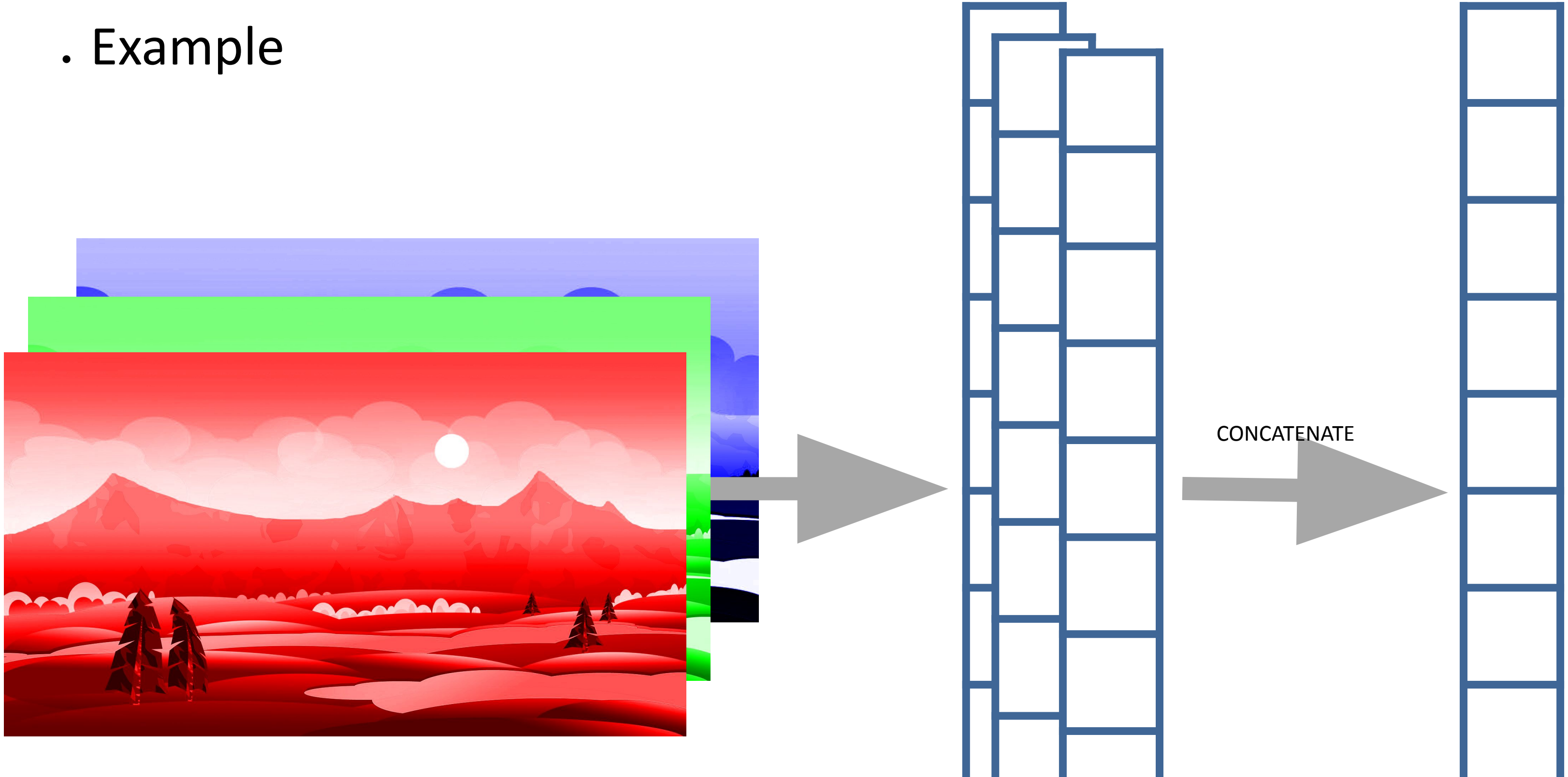
- Example





INPUT DATA - SOUND, IMAGES, VIDEO - ENCODING

- Example



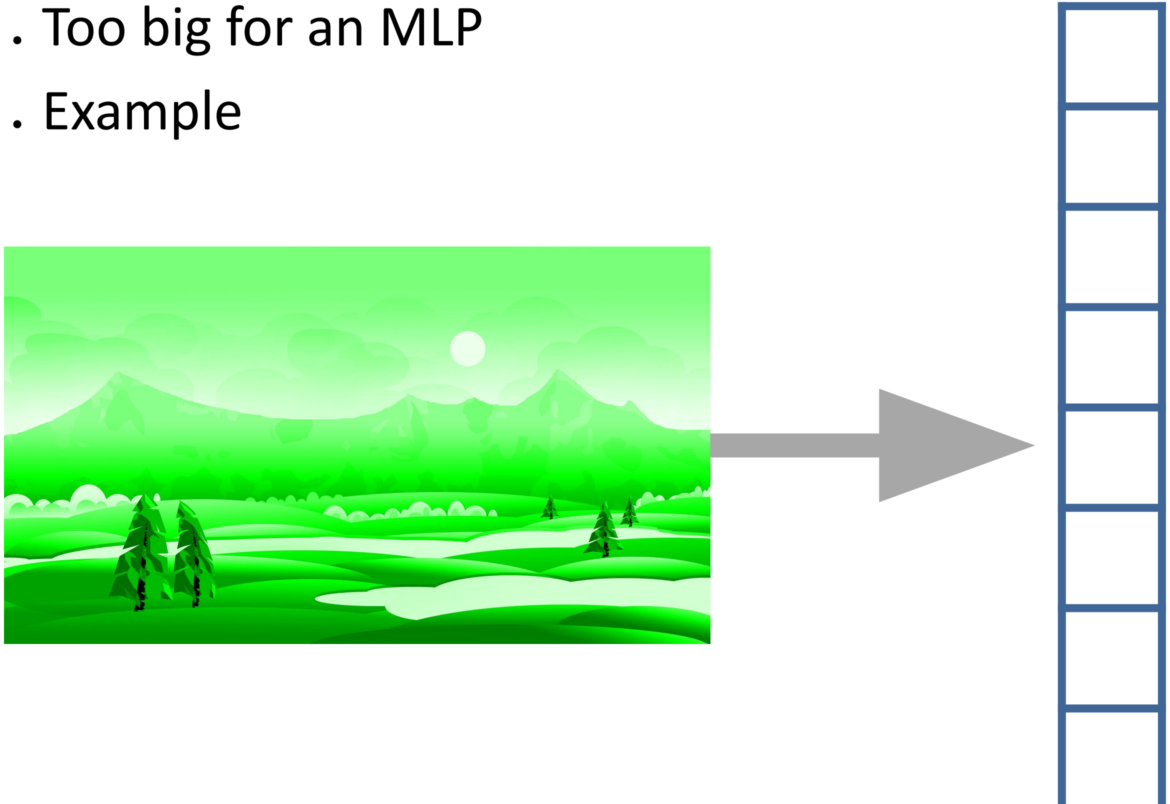
INPUT DATA - SOUND, IMAGES, VIDEO - ENCODING

- The input dimensions are **very** big

- One channel of an image of 1920x1080 ≃ 2M features
- 1 second of sound at 44kHz = 44k features
- A video: frame rate * image features + sound
  - 10 seconds => 10*(60fps*3*2M+44k)

INPUT DATA - SOUND, IMAGES, VIDEO - ENCODING

- The input dimensions are **very** big
- Too big for an MLP
- Example

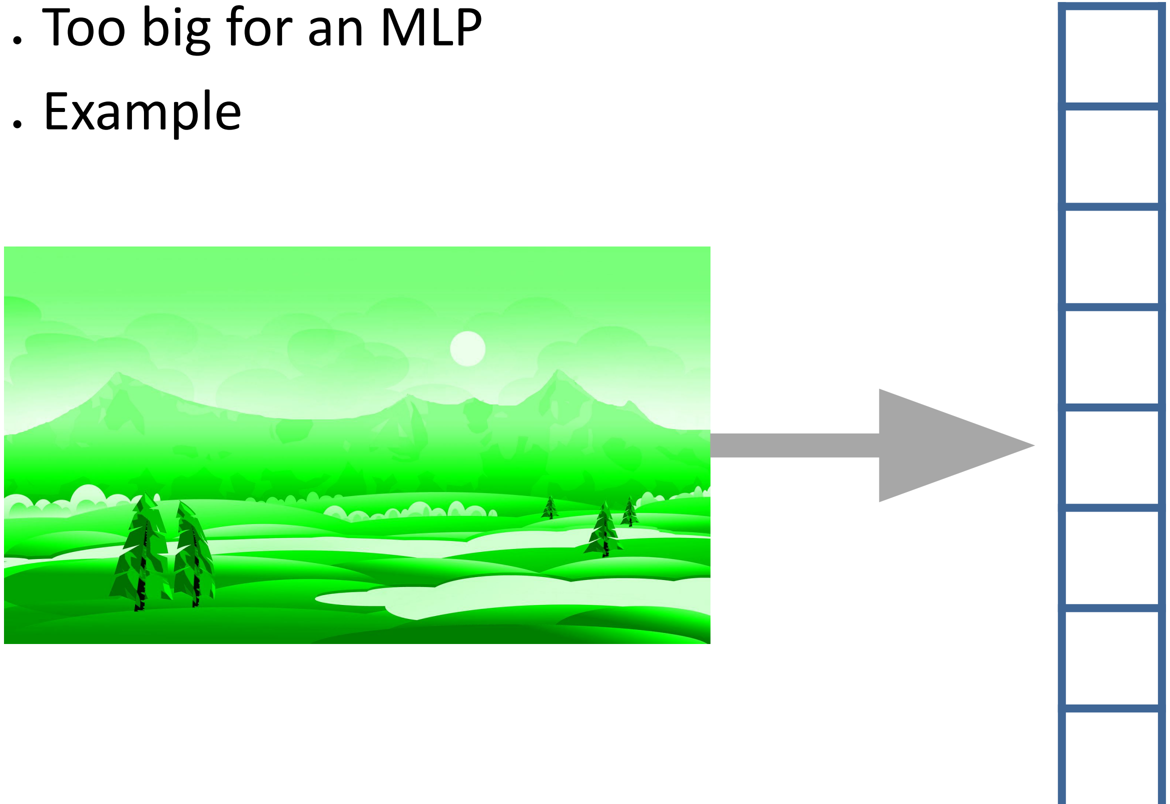


1920x1080 ≃ 2M

dimensional vector

INPUT DATA - SOUND, IMAGES, VIDEO - ENCODING - MLP

- The input dimensions are **very** big
- Too big for an MLP
- Example

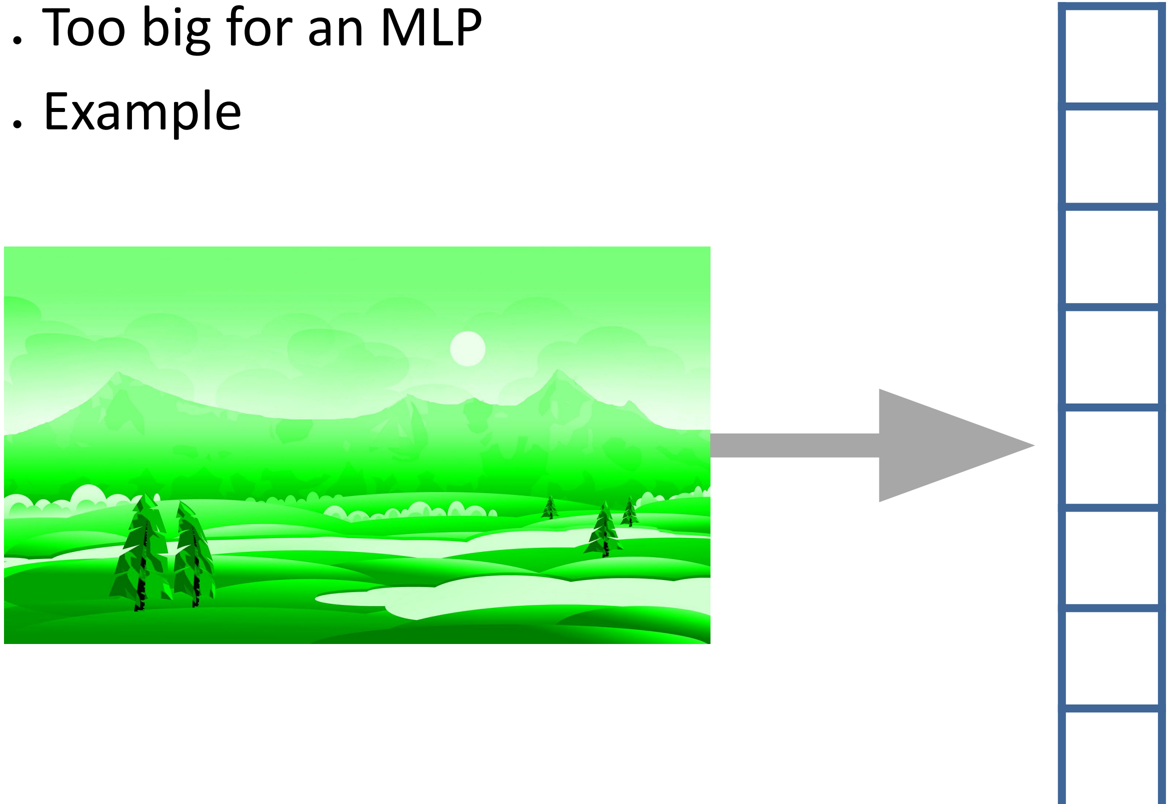


1920x1080 ≃ 2M dimensional vector

So, you need 2M weights for just 1 neuron!!!

INPUT DATA - SOUND, IMAGES, VIDEO - ENCODING - MLP

- The input dimensions are **very** big
- Too big for an MLP
- Example



1920x1080 ≃ 2M dimensional vector

So, you need 2M weights for just 1 neuron!!!

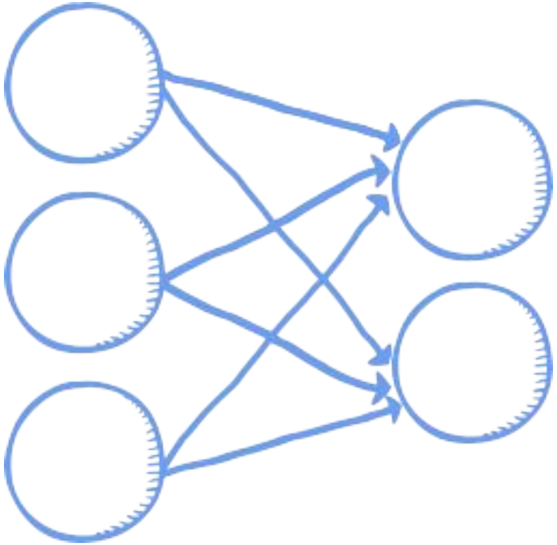
And you want more than 1

INPUT DATA - SOUND, IMAGES, VIDEO - ENCODING - MLP - LIMITATIONS

- The input dimensions are **very** big
- Too big for an MLP
  - Too many weights
  - Would not converge
  - would not fit in GPU memory
    - Especially when you also need to keep gradient information

INPUT DATA - SOUND, IMAGES, VIDEO - ENCODING

- The features in this kind of data are **not** independent
  - They have locality
- But, an MLP does not remember this ordering

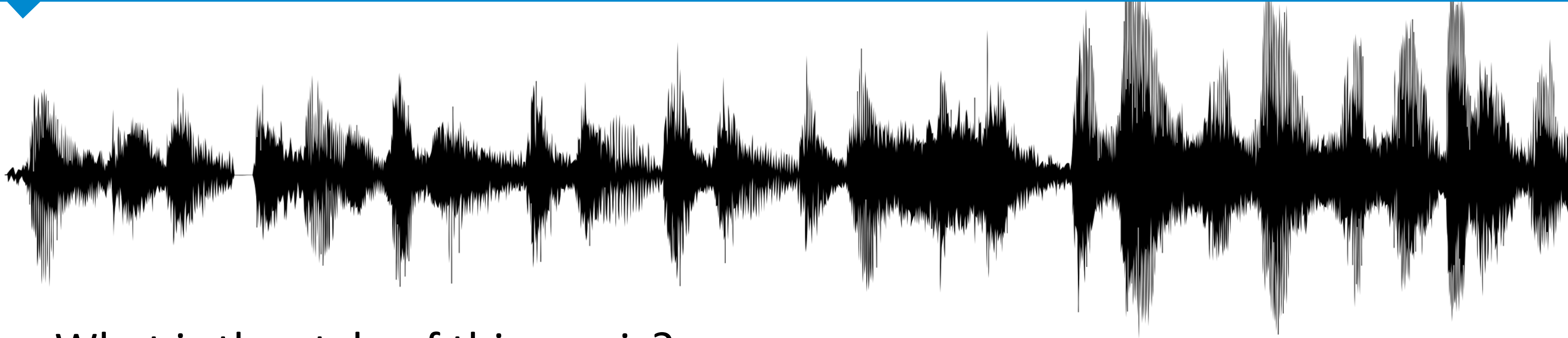


GOAL FOR TODAY

- We want to be able to do deep learning on this kind of data
- Steps
    - 1D
        - Build intuition
    - 2D
        - Do the same for images

PART TWO-a: conv1D

1D - SOUND - TASK - CLASSIFICATION / REGRESSION



- What is the style of this music?
- Does the user like this music (yes/no)?
  - A classifier
- What is the beat of this music?
- How pleasant is this music to listen to (1-100)?
  - Regression

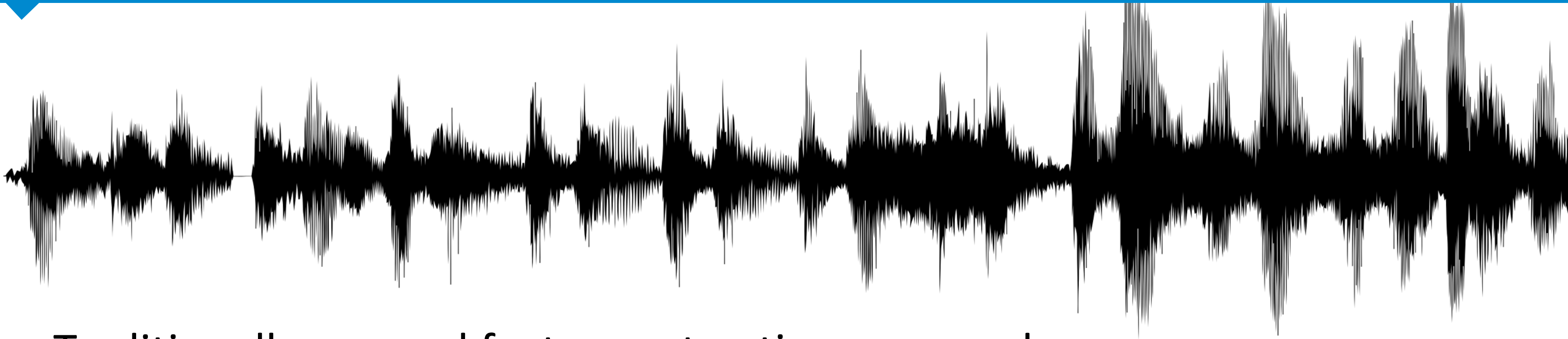
1D - SOUND - TASK - GENERATION

- What does a cleaned version of this audio signal look like?
- What audio would fit to these lyrics?
- How would this song continue?

1D - SOUND - FEATURES - TRADITIONAL

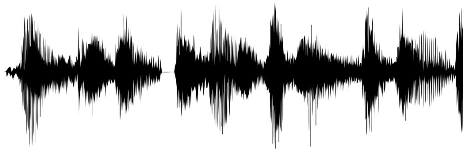
- Traditionally, manual feature extraction was used
  - (digital) signal processing with filters
  - Detecting beat
  - Finding manually crafted patterns
  - etc.

1D - SOUND - FEATURES - TRADITIONAL - ISSUES

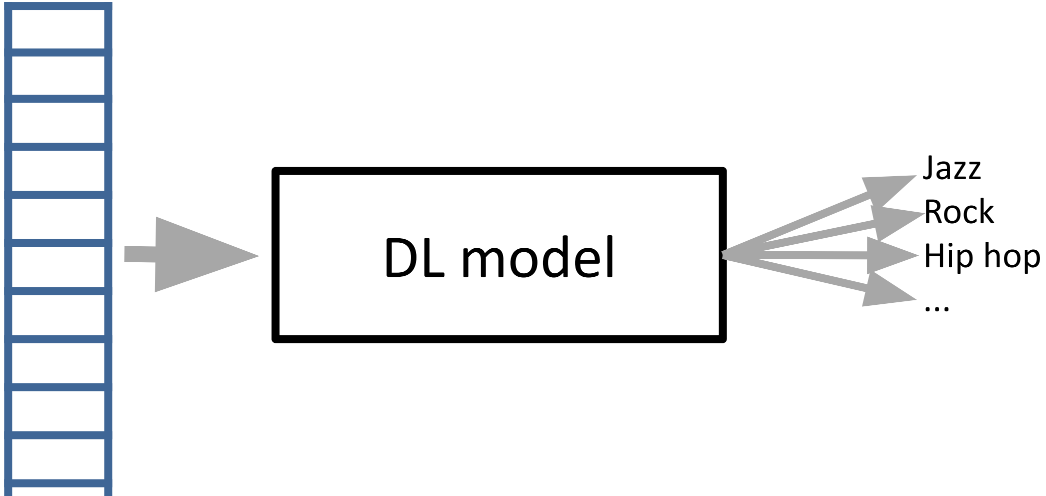


- Traditionally, manual feature extraction was used
  - Problems:
    - Noise
    - Variations
    - Fragments missing
    - etc.

1D - SOUND - FEATURES - DEEP LEARNING



- Feature Extraction in deep learning is dealt with by the model itself



DL model

Jazz

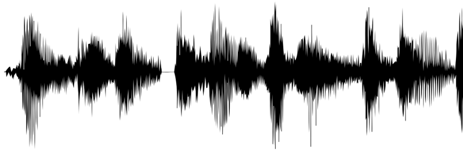
Rock

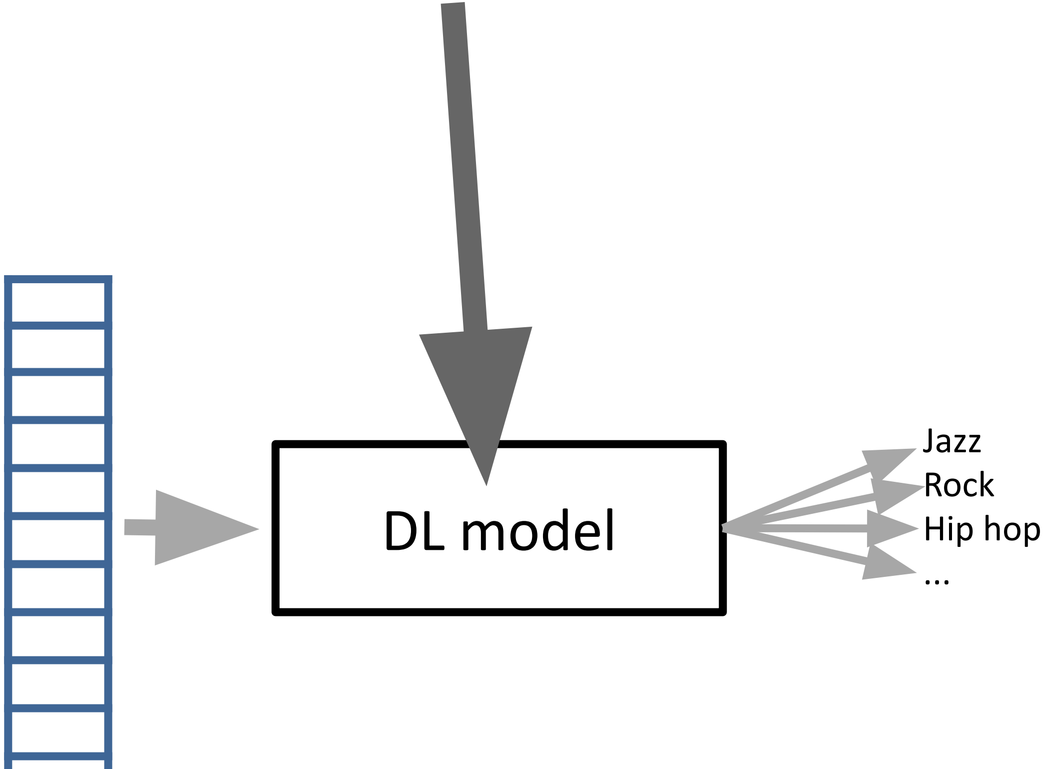
Hip hop

...

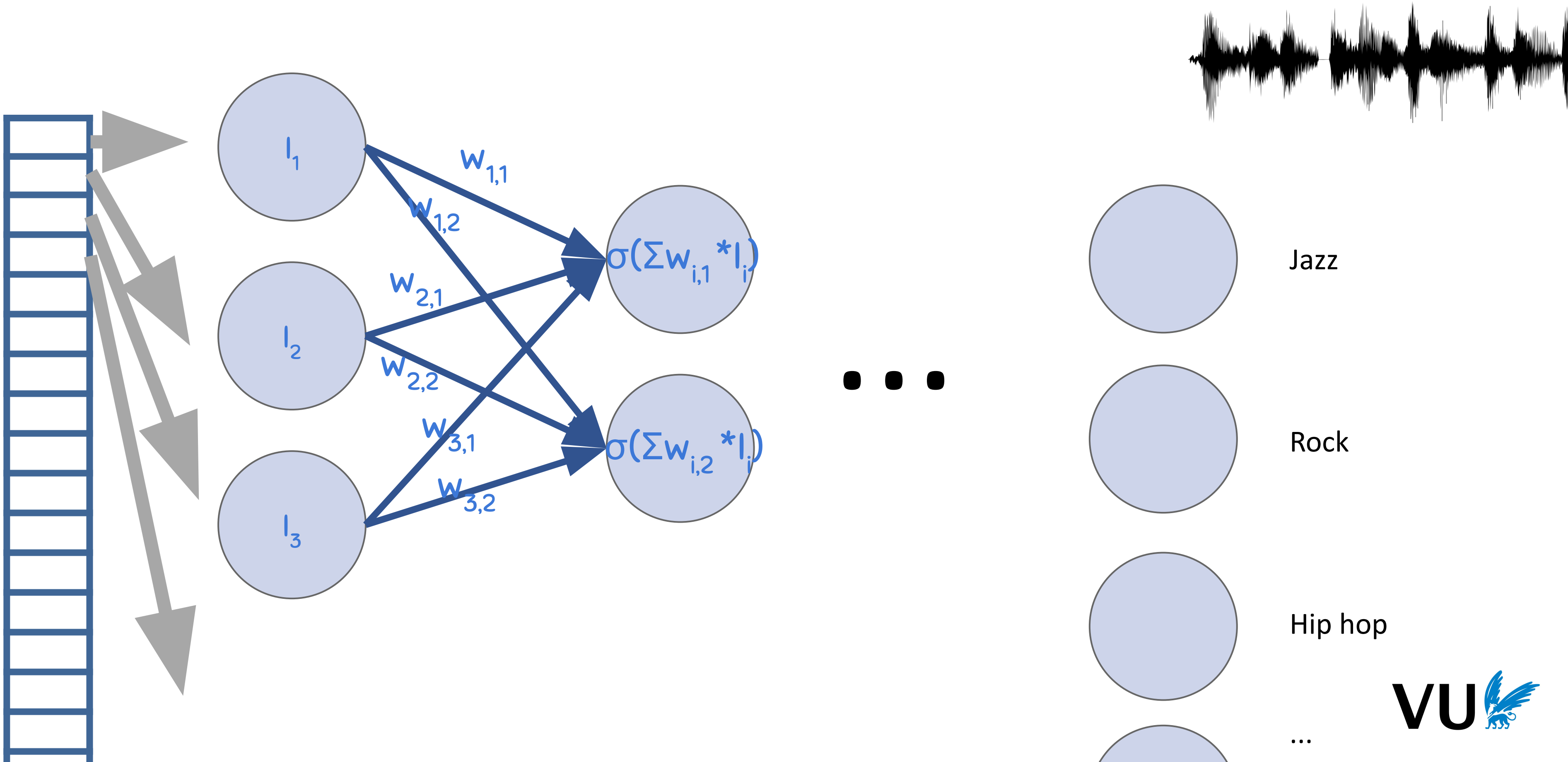
1D - SOUND - HOW? - MLP

- Let us try to use an MLP

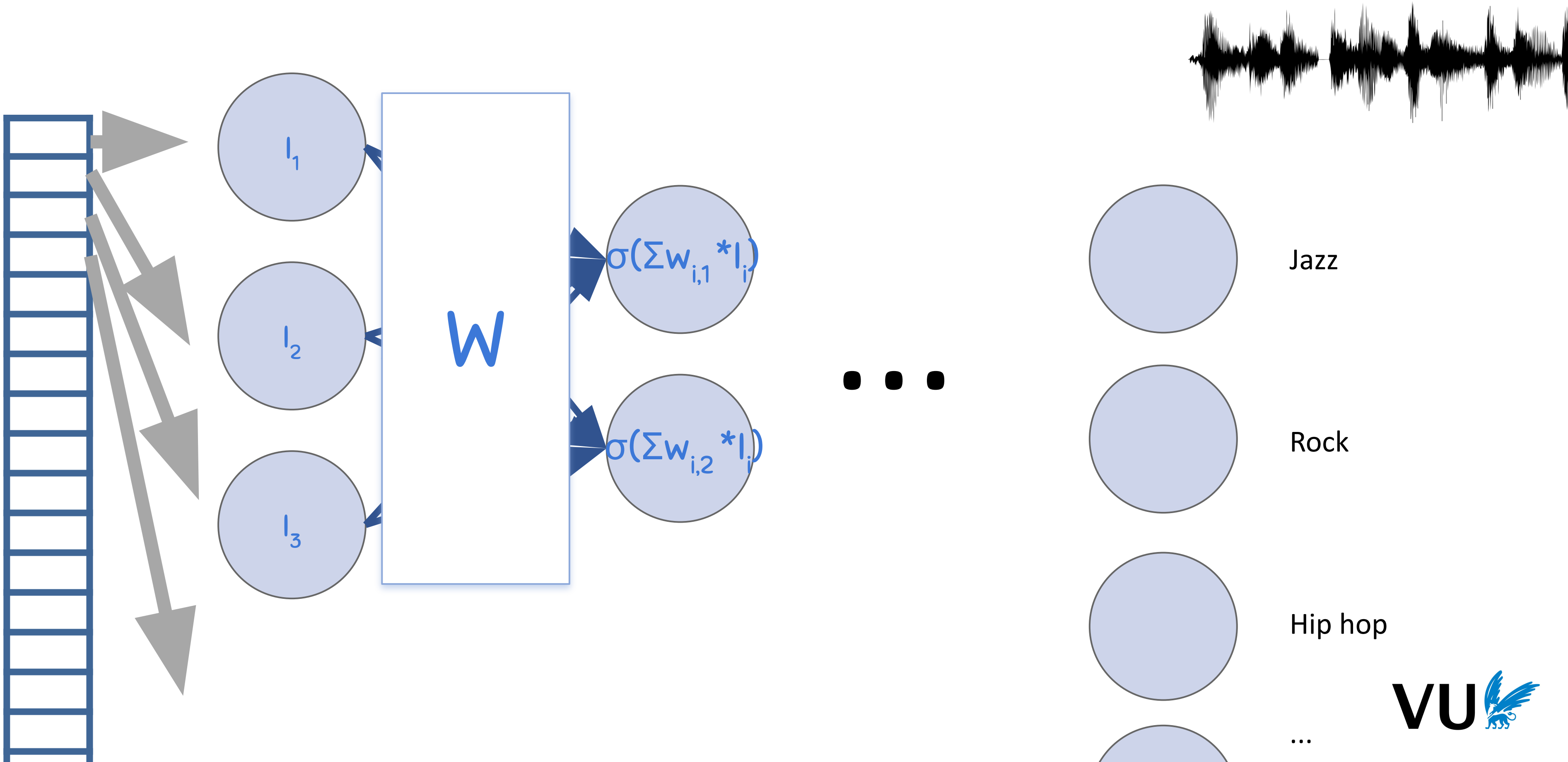




1D - SOUND - HOW? - MLP

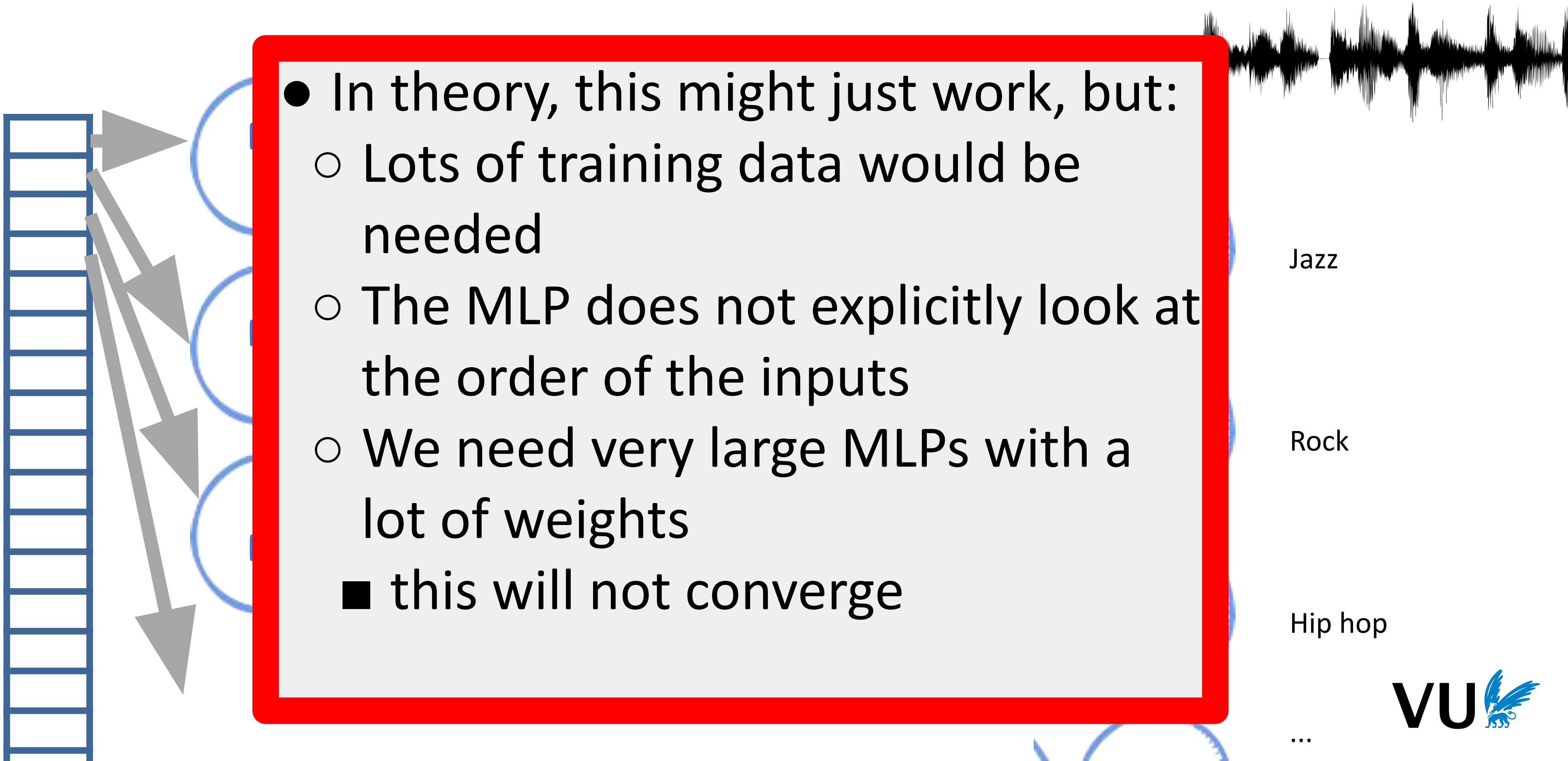


1D - SOUND - HOW? - MLP

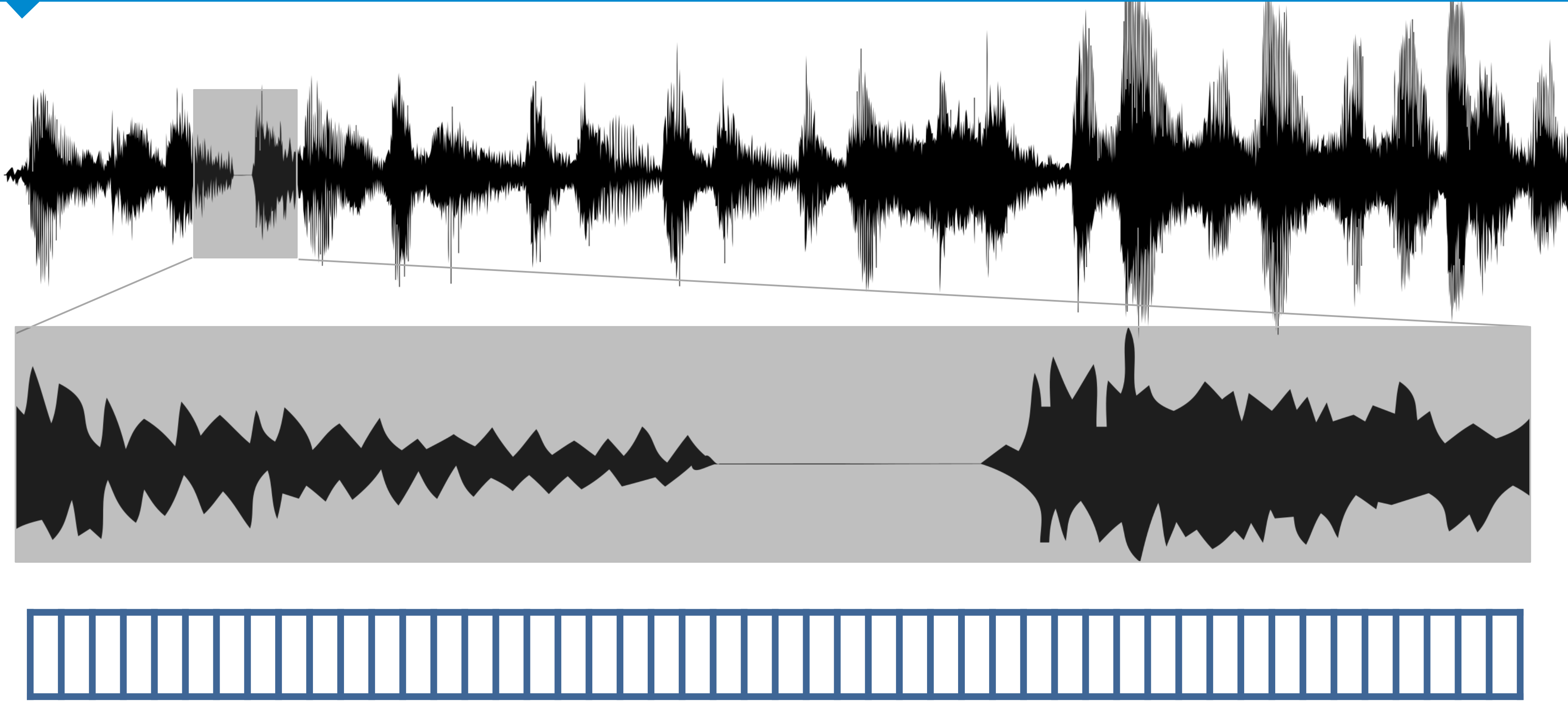

I_1
I_2
I_3
W
$\sigma(\Sigma w_{i,1} * I_i)$
$\sigma(\Sigma w_{i,2} * I_i)$
...
Jazz
Rock
Hip hop
...

1D - SOUND - HOW? - MLP

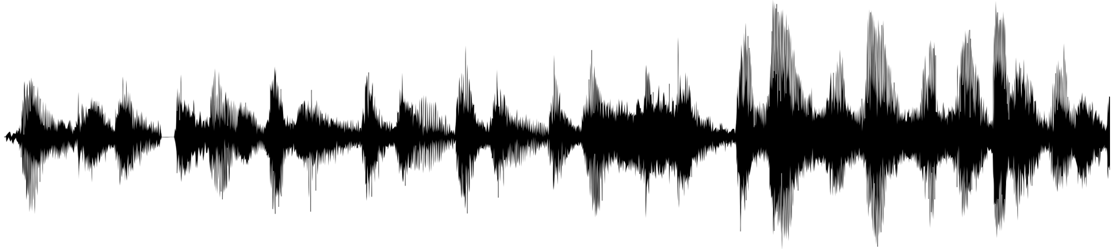
- In theory, this might just work, but:
  - Lots of training data would be needed
  - The MLP does not explicitly look at the order of the inputs
  - We need very large MLPs with a lot of weights
    - this will not converge



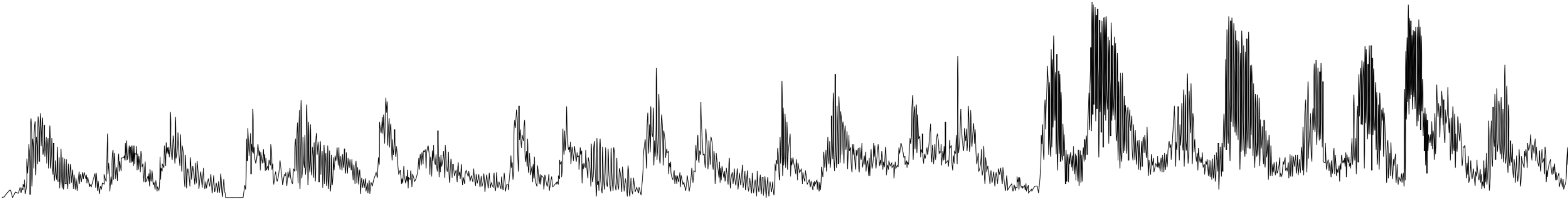
1D - SOUND - FEATURES - DIGITAL



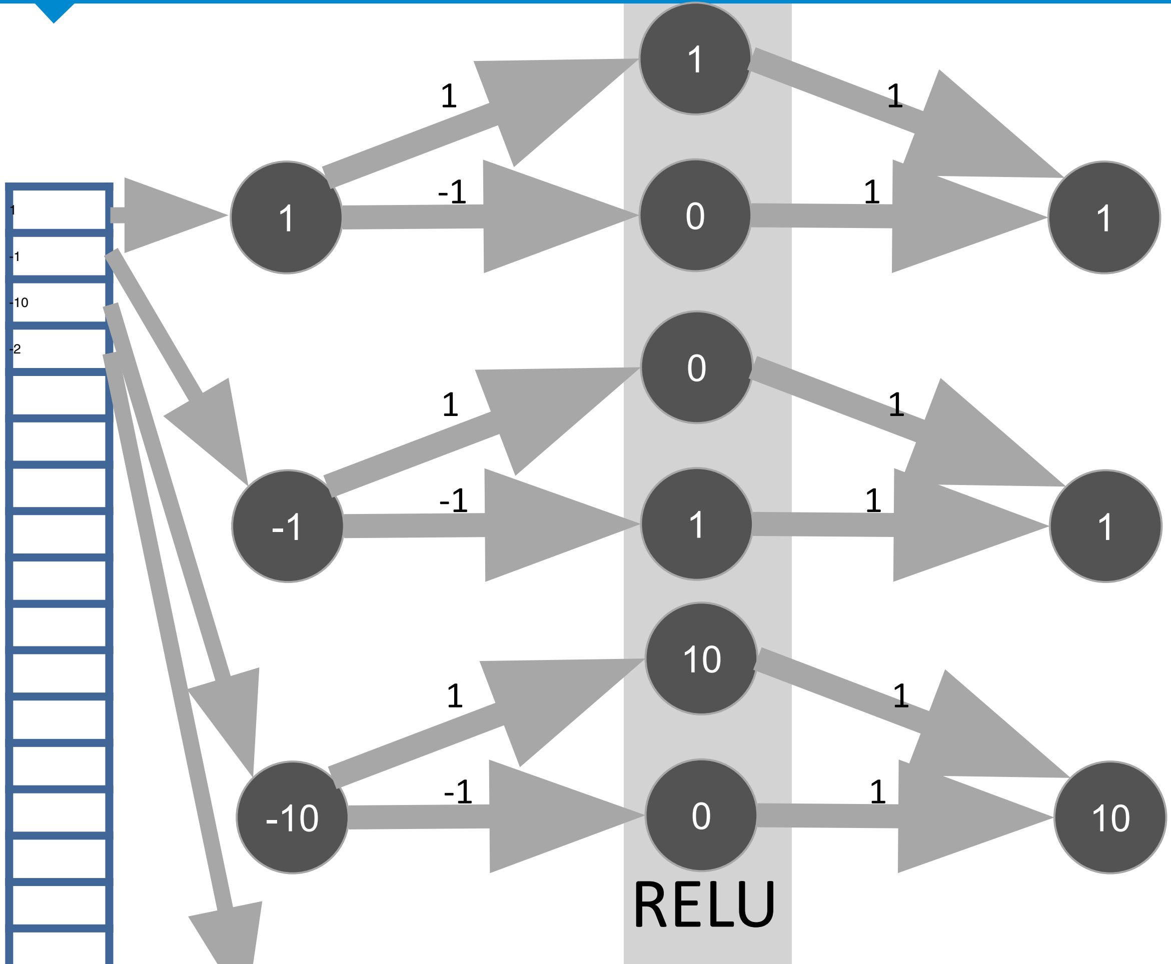
1D - SOUND - AMPLITUDE

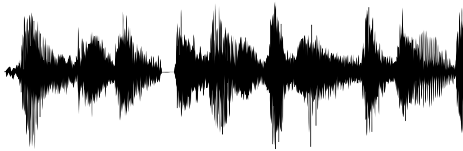


- In the context of the following, we will only use the amplitude of the soundwave:

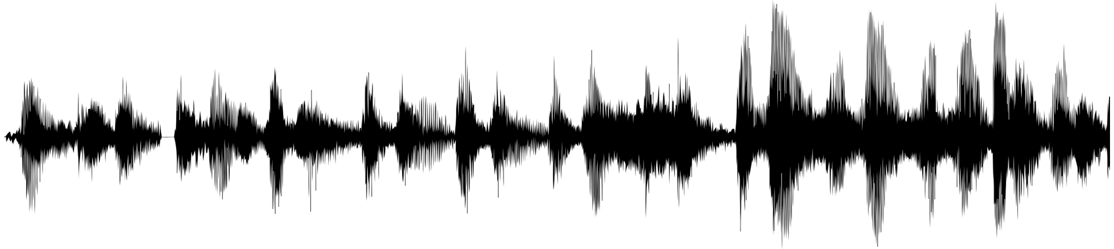


1D - SOUND - FILTERS - AMPLITUDE MLP

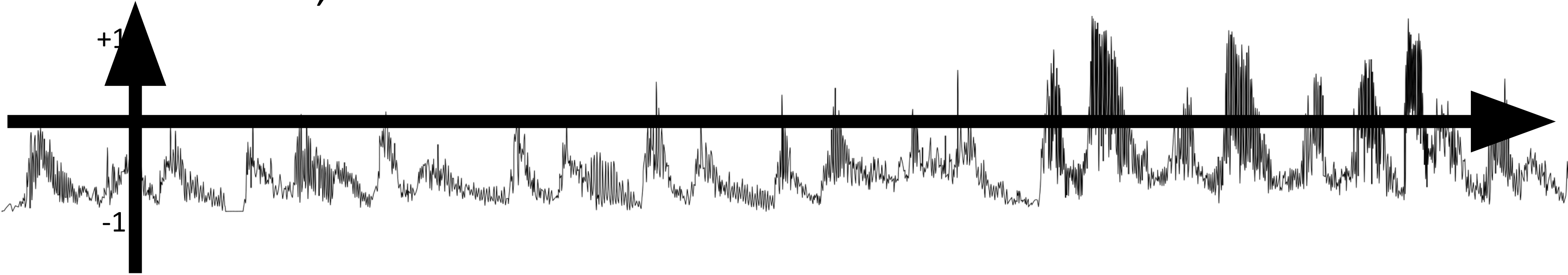




1D - SOUND - AMPLITUDE

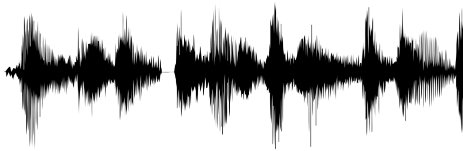


- In the context of the following, we will only use the amplitude of the soundwave, which we will normalize



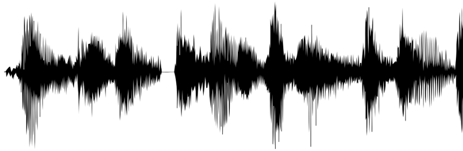
1D - SOUND - FILTERS

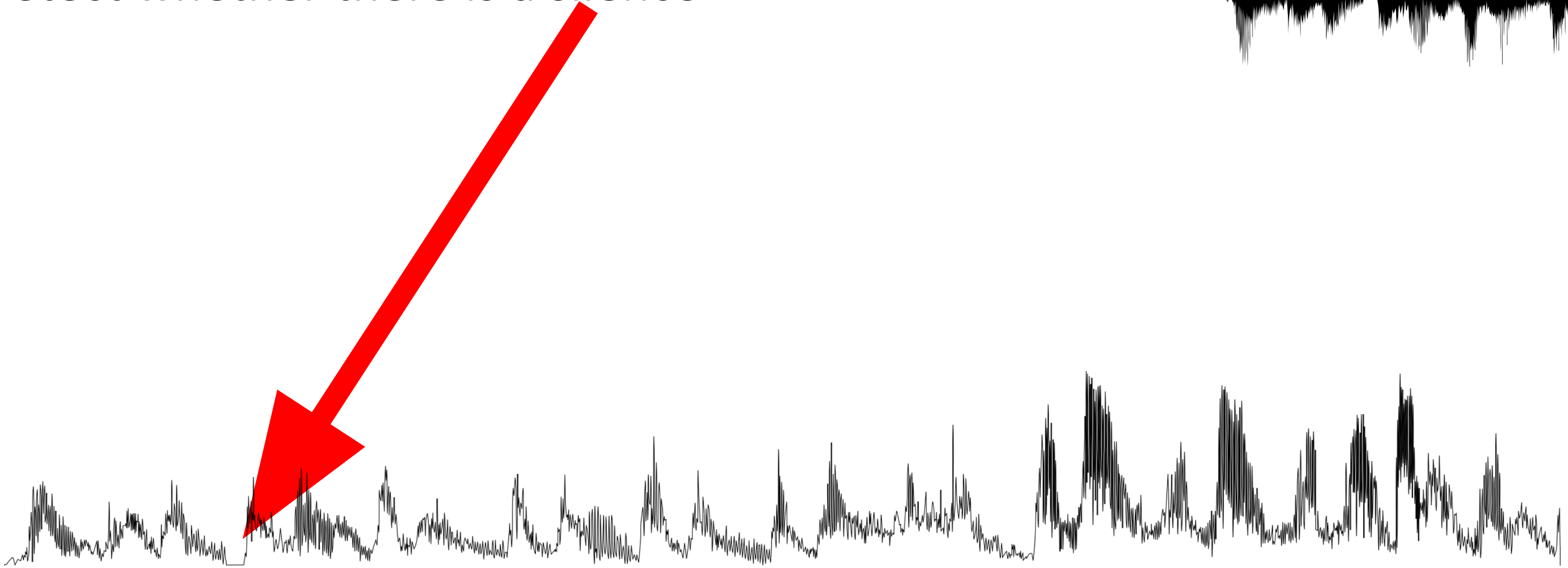
- The features in the soundwave are not independent
- Nearby features are more important as far away ones
- This idea can be used in filters



1D - SOUND - FILTERS - EXAMPLE

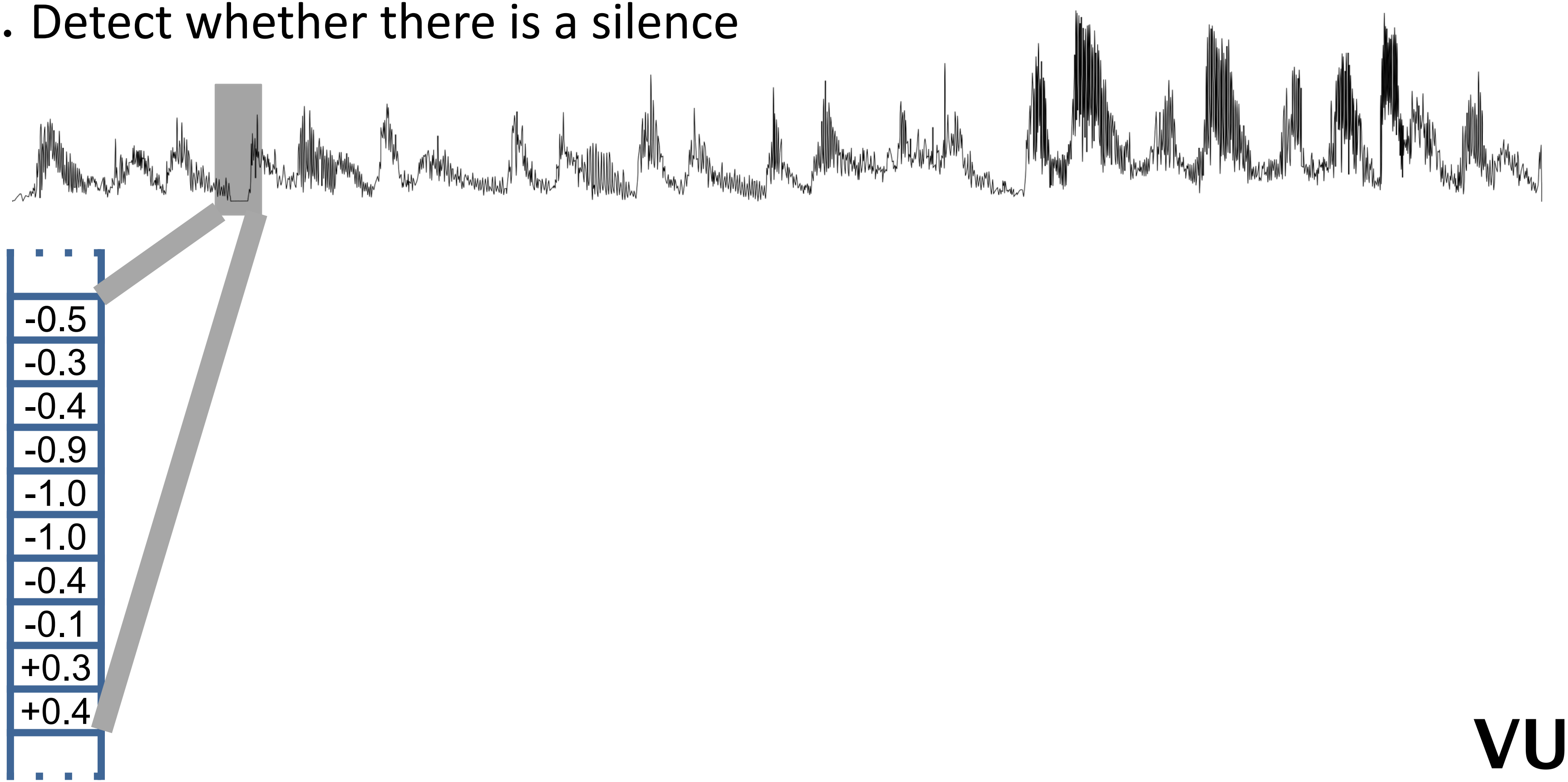
- Detect whether there is a silence





1D - SOUND - FILTERS - EXAMPLE

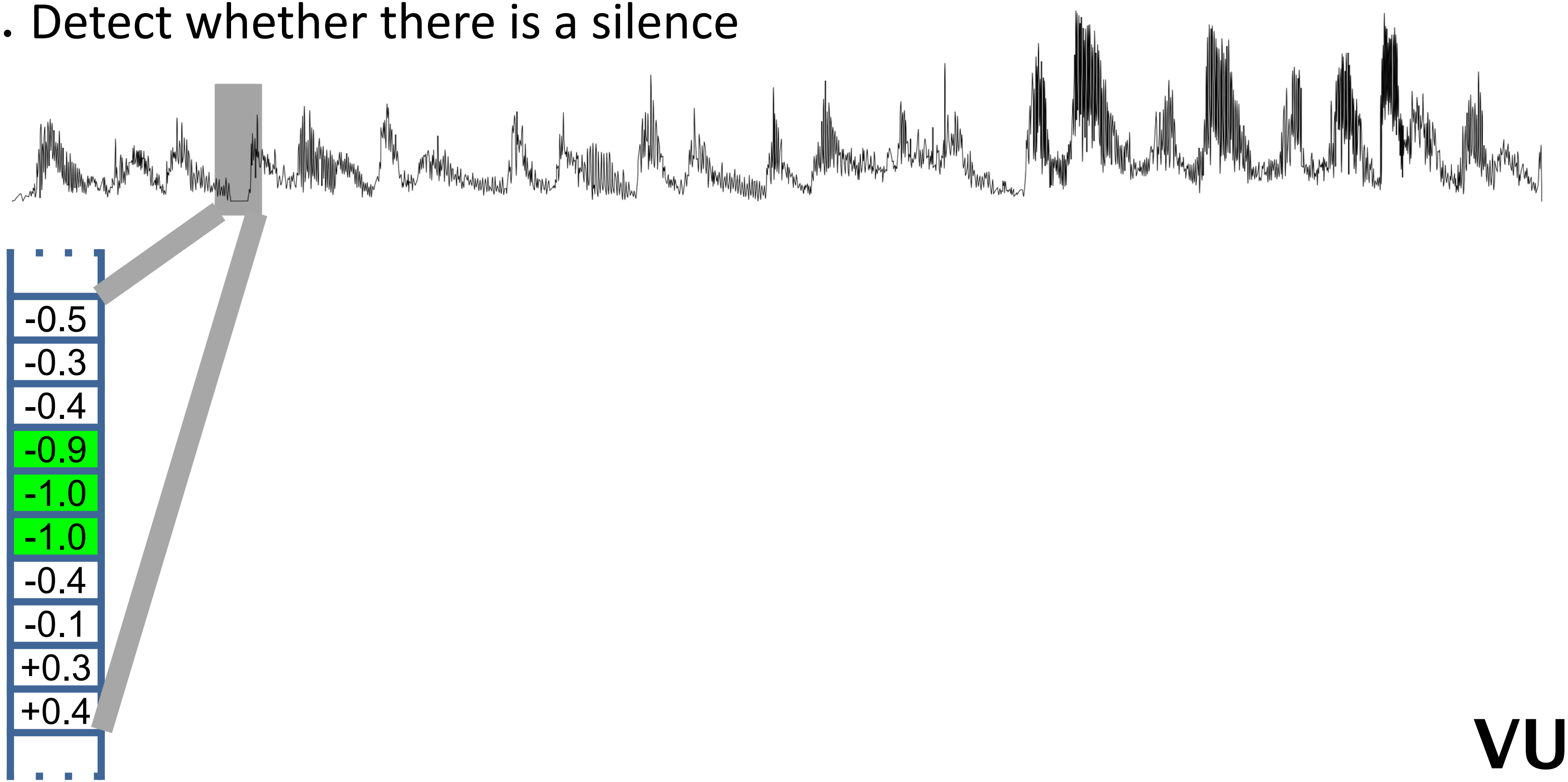
- Detect whether there is a silence



-0.5
-0.3
-0.4
-0.9
-1.0
-1.0
-0.4
-0.1
+0.3
+0.4

1D - SOUND - FILTERS - EXAMPLE

- Detect whether there is a silence



. .

-0.5

-0.3

-0.4

-0.9

-1.0

-1.0

-0.4

-0.1

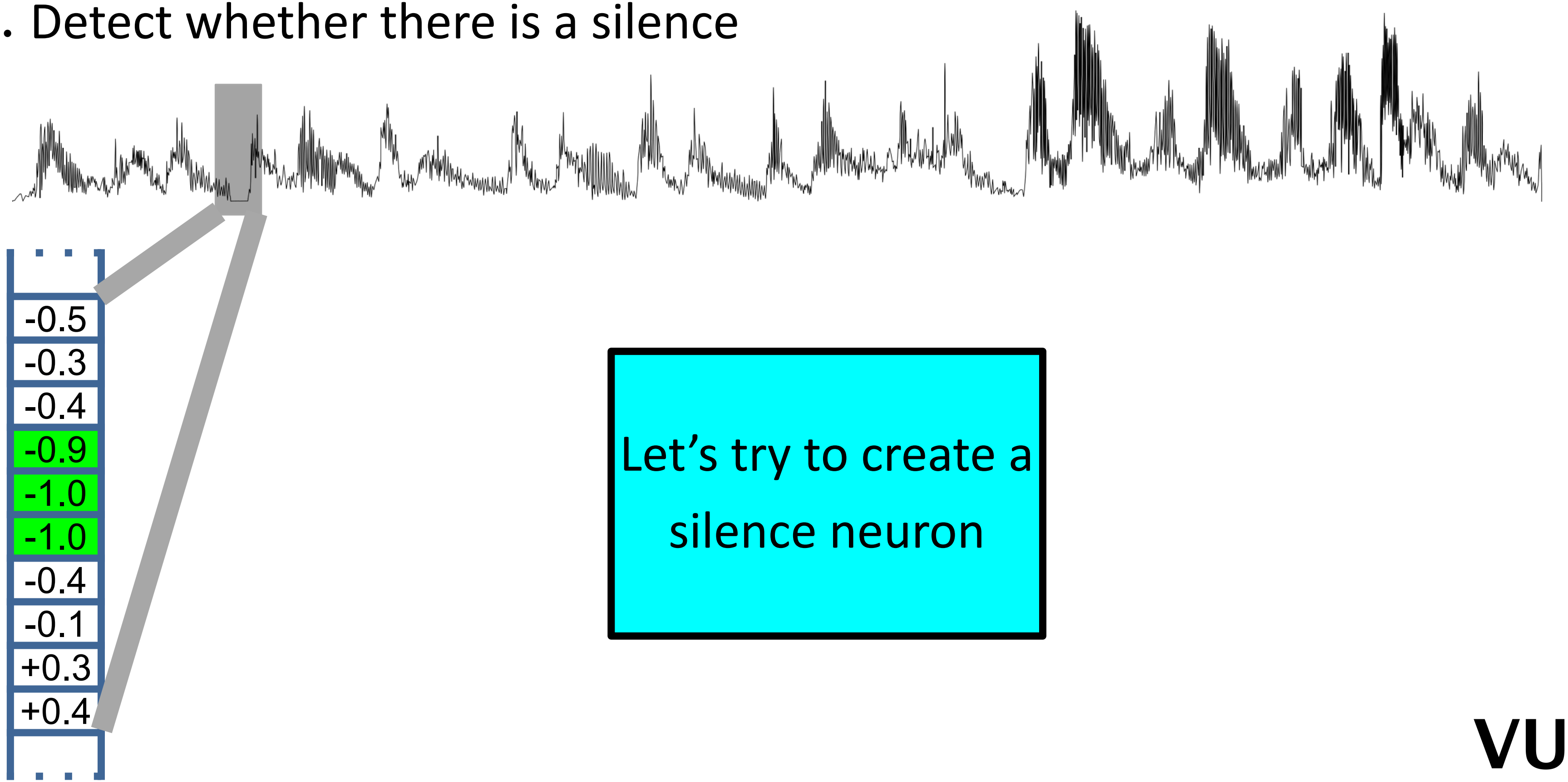
+0.3

+0.4

. .

1D - SOUND - FILTERS - EXAMPLE

- Detect whether there is a silence

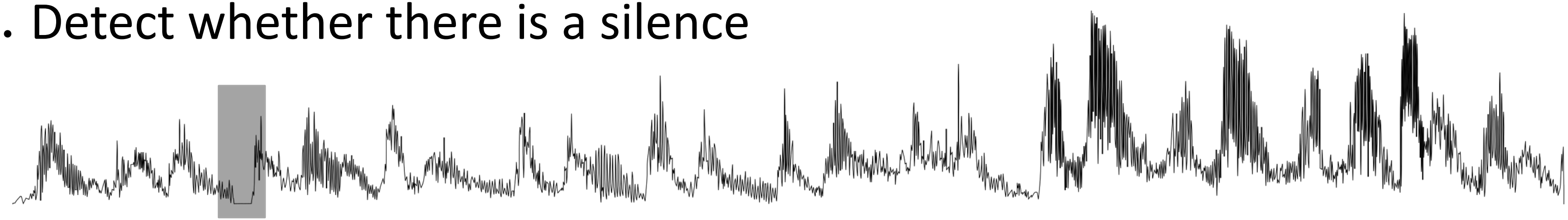


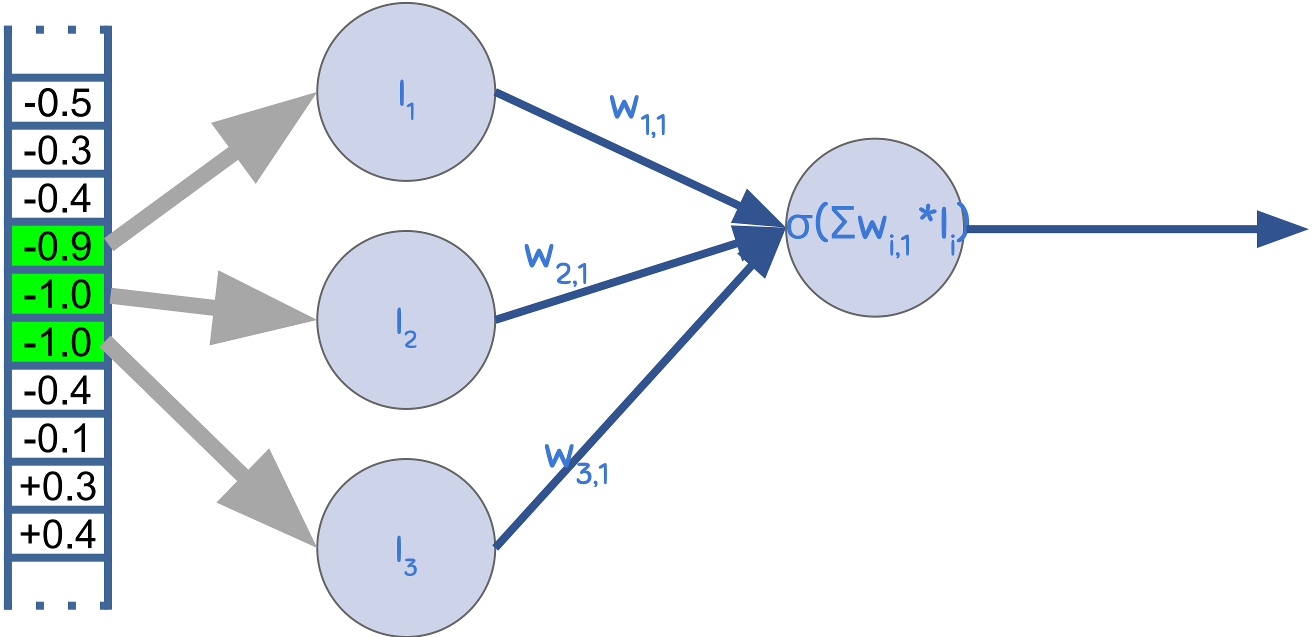
-0.5
-0.3
-0.4
-0.9
-1.0
-1.0
-0.4
-0.1
+0.3
+0.4

Let’s try to create a silence neuron

1D - SOUND - FILTERS - EXAMPLE

- Detect whether there is a silence

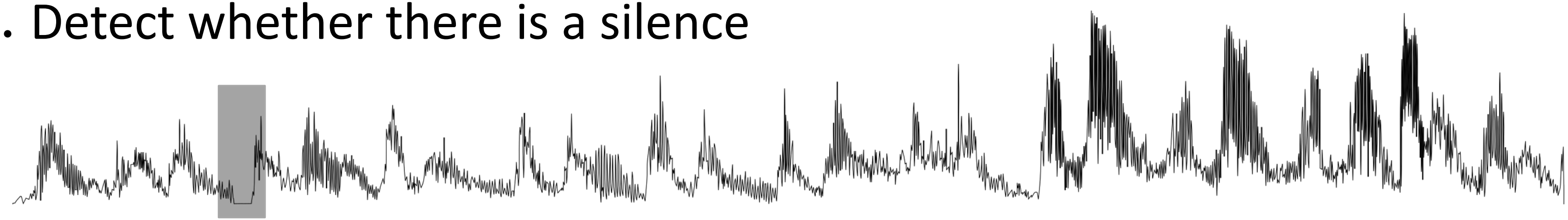


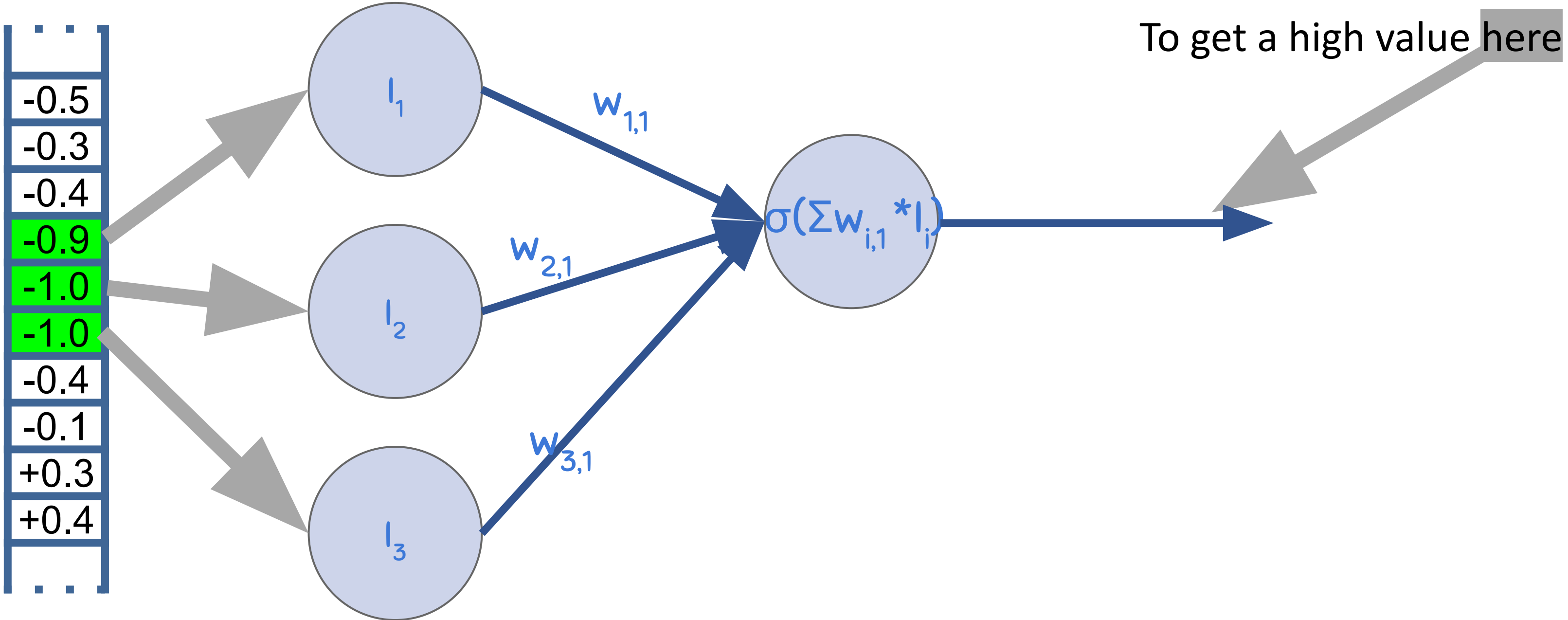


-0.5
-0.3
-0.4
-0.9
-1.0
-1.0
-0.4
-0.1
+0.3
+0.4

1D - SOUND - FILTERS - EXAMPLE

- Detect whether there is a silence





. .

-0.5

-0.3

-0.4

-0.9

-1.0

-1.0

-0.4

-0.1

+0.3

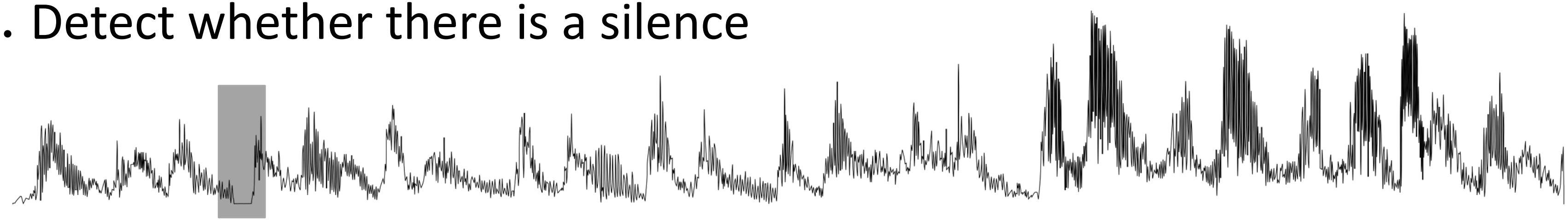
+0.4

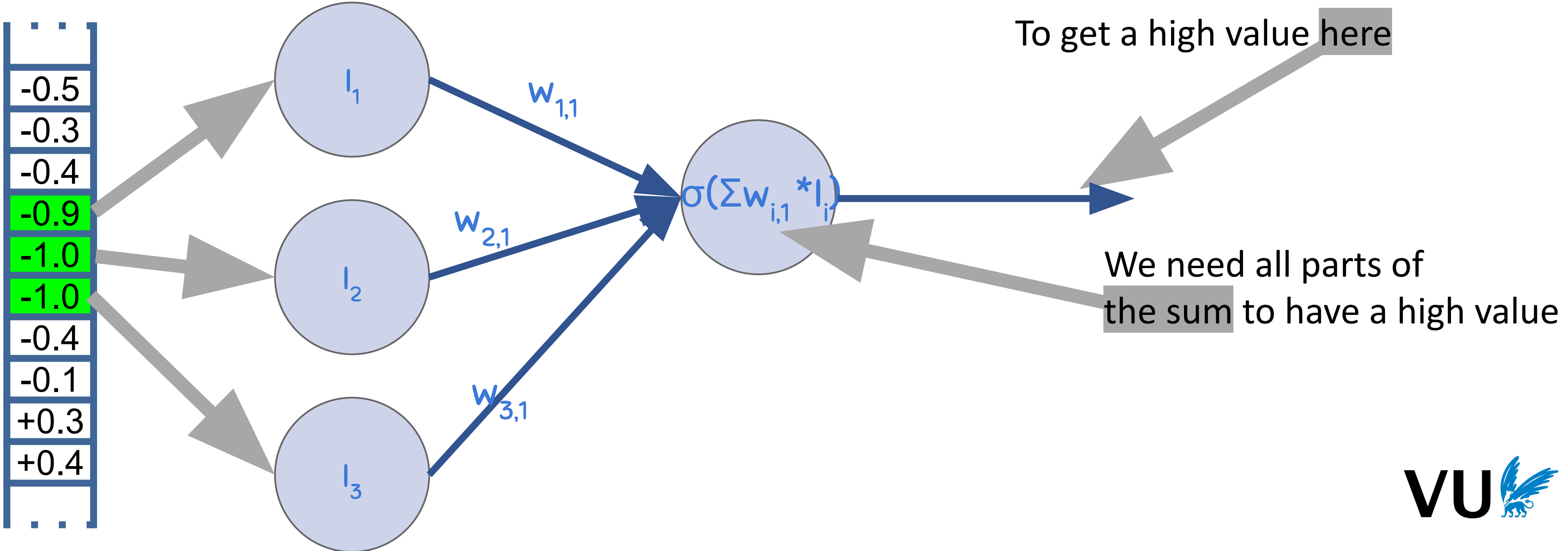
. .

To get a high value here

1D - SOUND - FILTERS - EXAMPLE

- Detect whether there is a silence





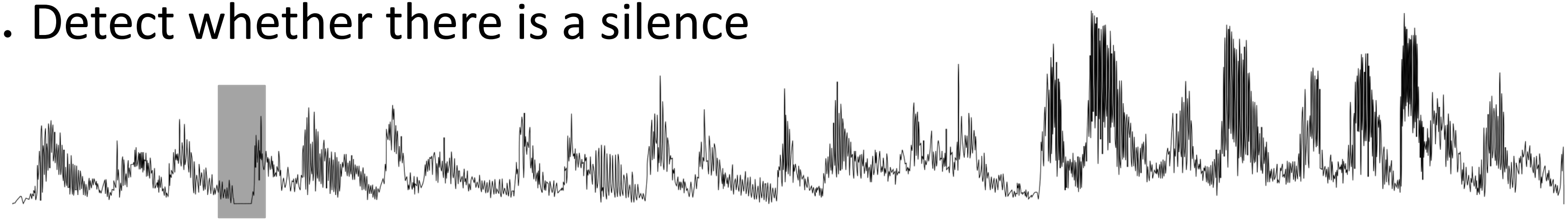
...
-0.5
-0.3
-0.4
-0.9
-1.0
-1.0
-0.4
-0.1
+0.3
+0.4
...

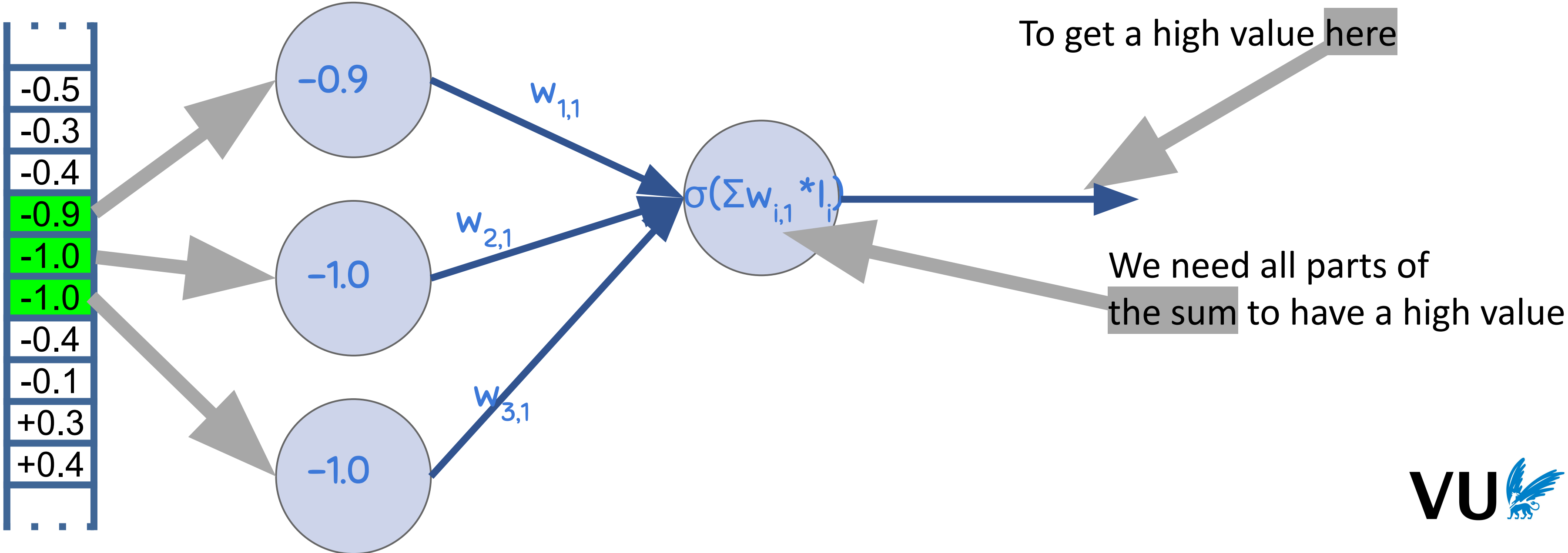
To get a high value here

We need all parts of the sum to have a high value

1D - SOUND - FILTERS - EXAMPLE

- Detect whether there is a silence



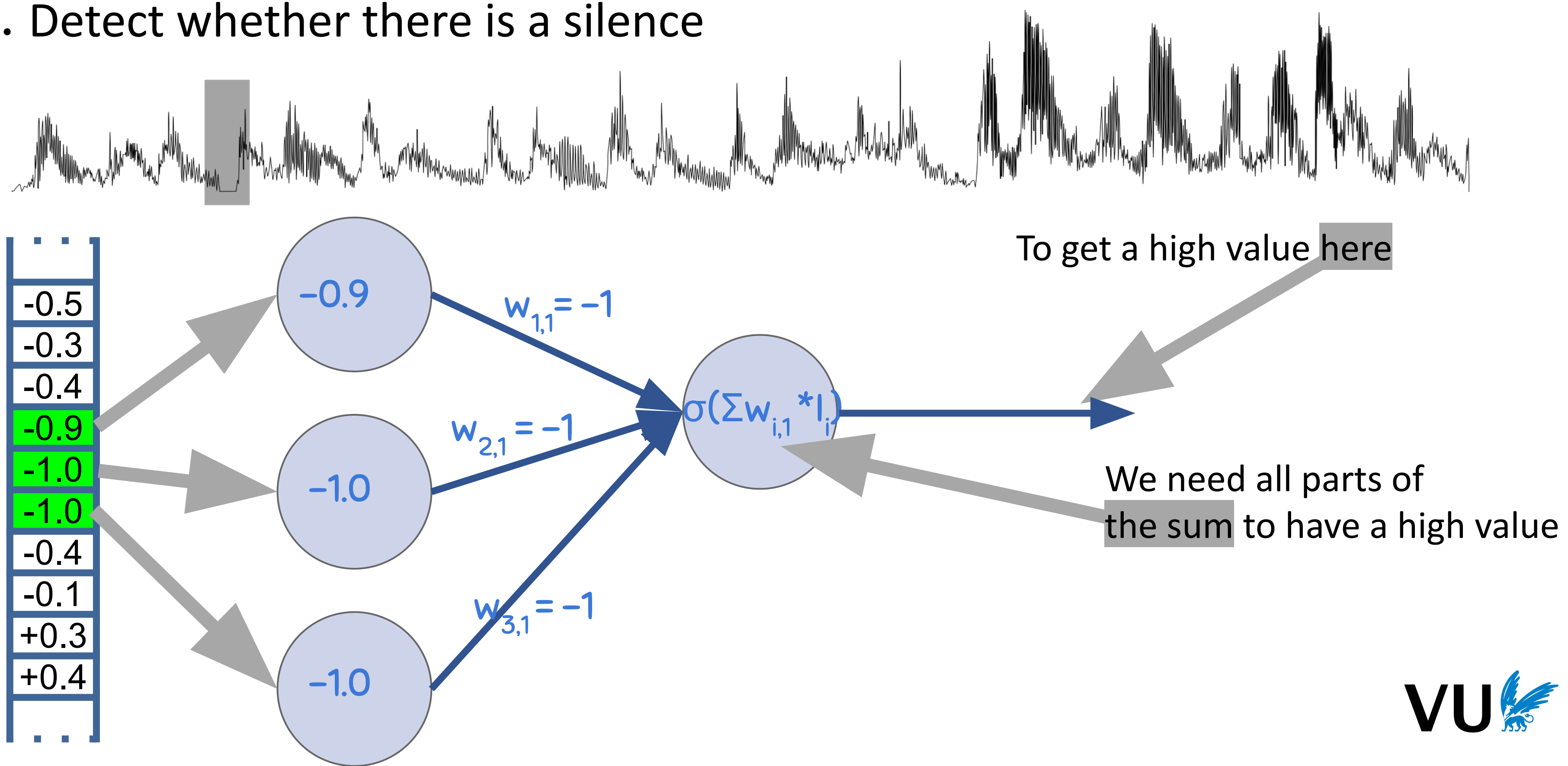


To get a high value here

We need all parts of the sum to have a high value

1D - SOUND - FILTERS - EXAMPLE

- Detect whether there is a silence

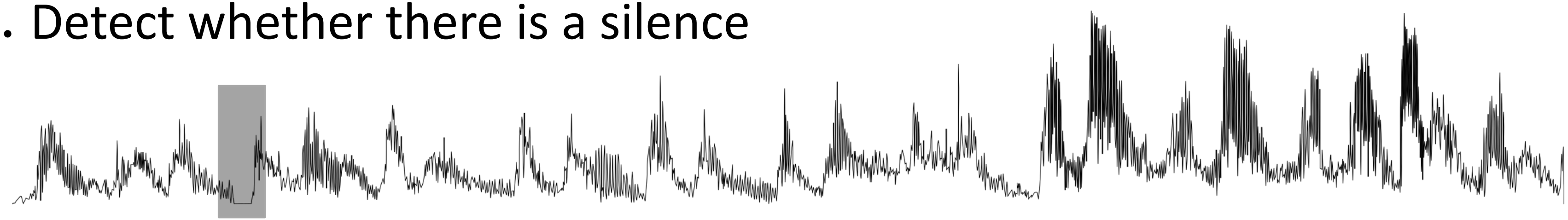


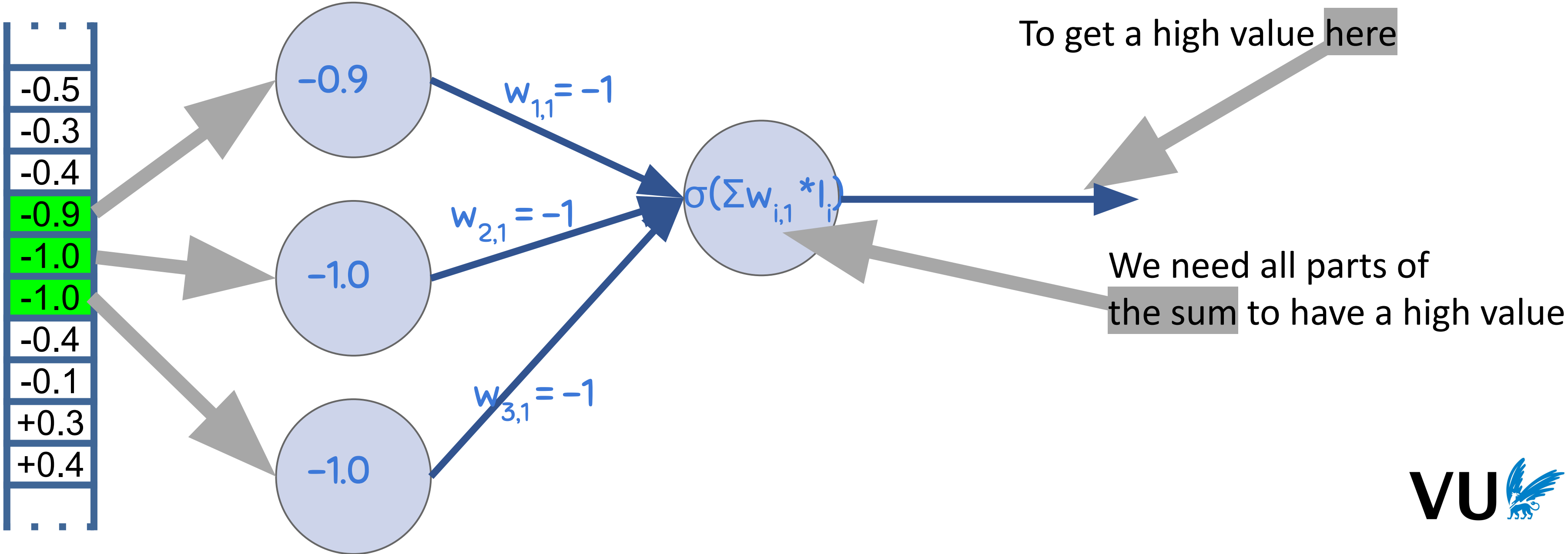
To get a high value here

We need all parts of the sum to have a high value

1D - SOUND - FILTERS - EXAMPLE

- Detect whether there is a silence





. . .

-0.5

-0.3

-0.4

-0.9

-1.0

-1.0

-0.4

-0.1

+0.3

+0.4

. . .

−0.9

−1.0

−1.0

$w_{1,1} = -1$

$w_{2,1} = -1$

$w_{3,1} = -1$

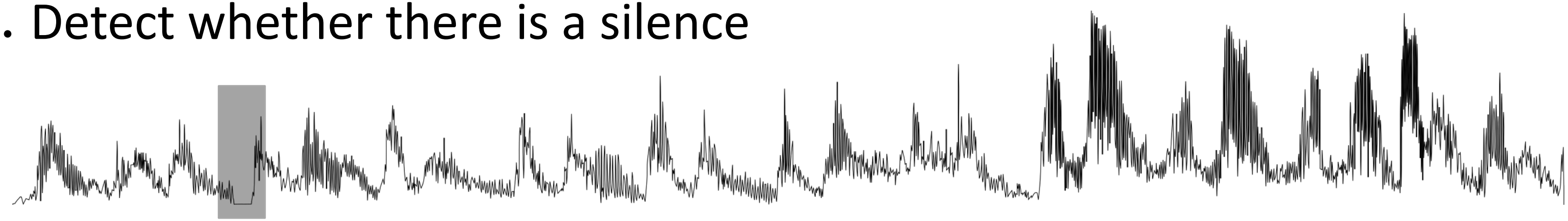
$\sigma(\Sigma w_{i,1} * I_i)$

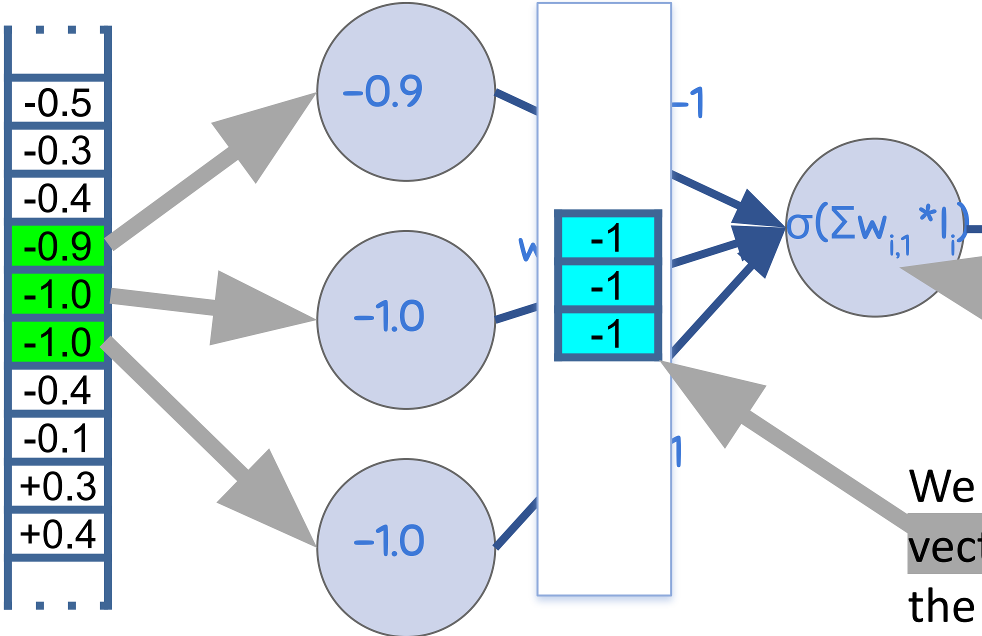
To get a high value here

We need all parts of the sum to have a high value

1D - SOUND - FILTERS - EXAMPLE

- Detect whether there is a silence





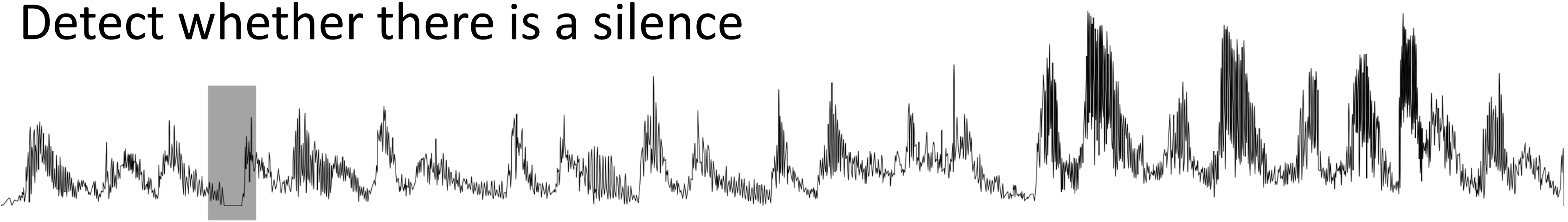
To get a high value here

We need all parts of the sum to have a high value

We represent our filter as a vector (which is equivalent to the weight matrix of an MLP.

1D - SOUND - FILTERS - EXAMPLE

- Detect whether there is a silence



-0.5
-0.3
-0.4
-0.9
-1.0
-1.0
-0.4
-0.1
+0.3
+0.4

Cleaning up

h value here

ed all parts of

m to have a high value

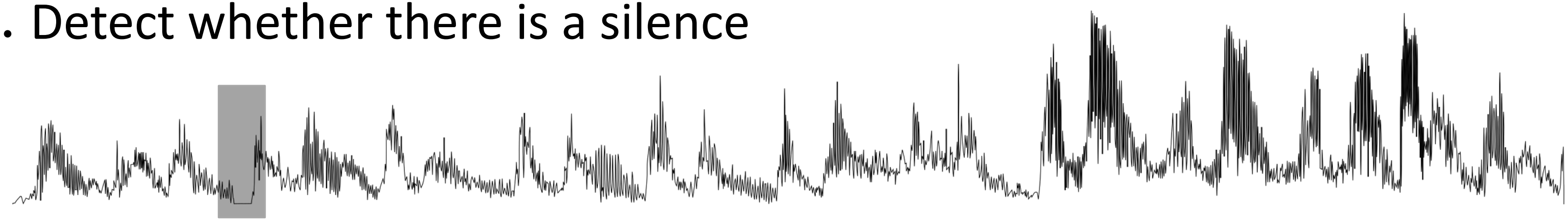
as a

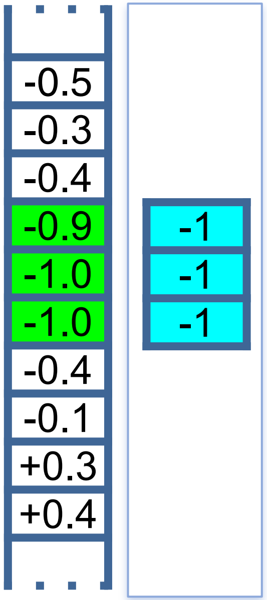
ent to

the weight matrix of an MLP.

1D - SOUND - FILTERS - CONVOLUTIONS

- Detect whether there is a silence



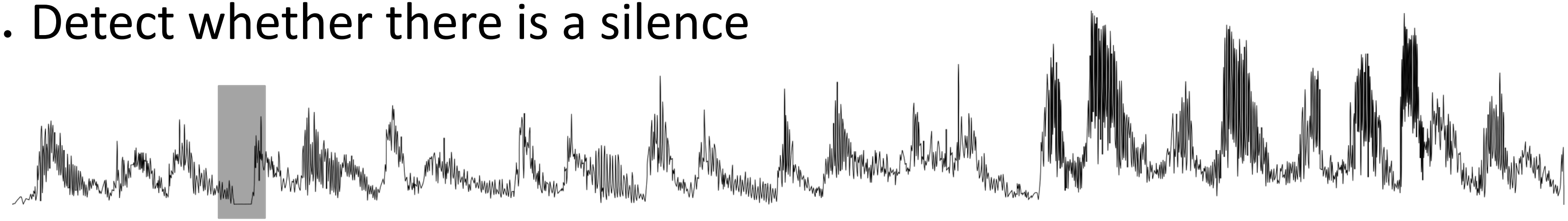


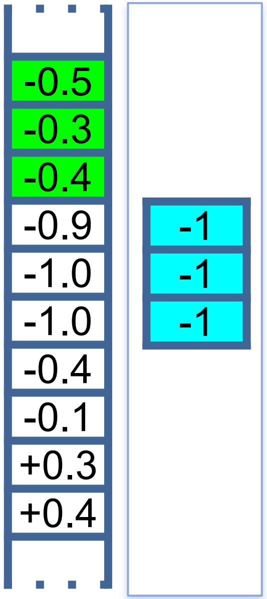
$\sigma(\Sigma w_i * I_i)$

$\Sigma w_i * I_i = 2.9$

1D - SOUND - FILTERS - CONVOLUTIONS

- Detect whether there is a silence





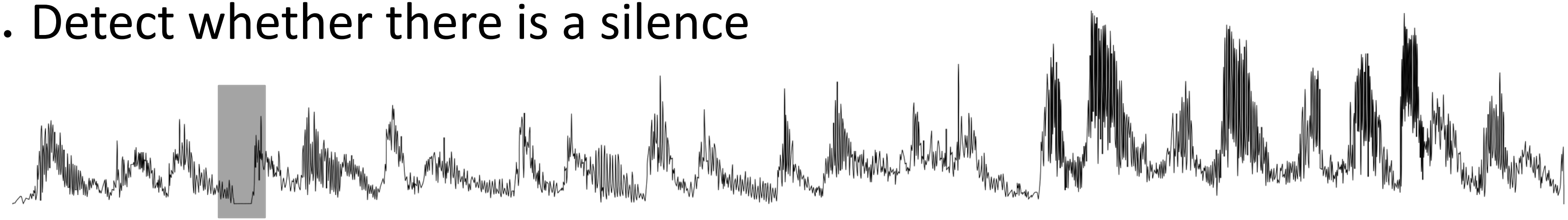
-0.5	
-0.3	
-0.4	
-0.9	-1
-1.0	-1
-1.0	-1
-0.4	
-0.1	
+0.3	
+0.4	

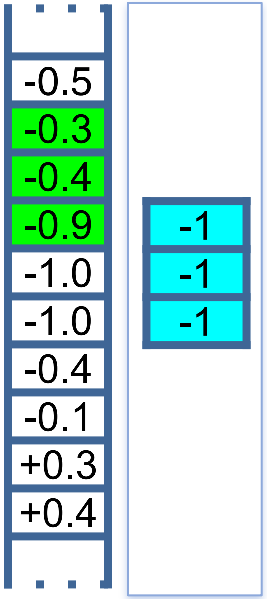
$\sigma(\Sigma w_i * I_i)$

$\Sigma w_i * I_i = 1.2$

1D - SOUND - FILTERS - CONVOLUTIONS

- Detect whether there is a silence



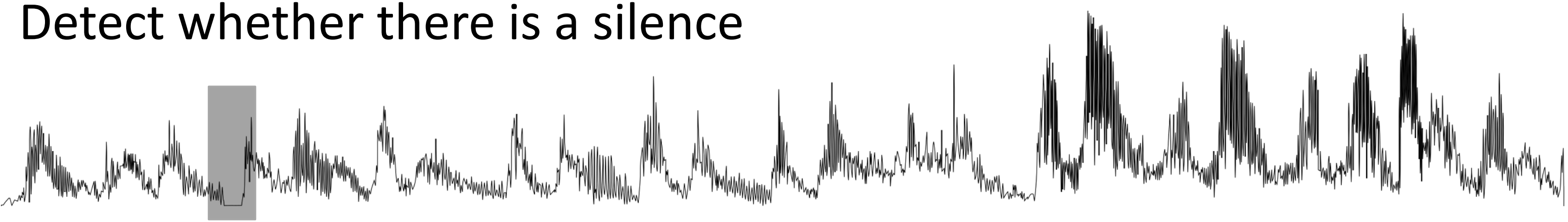


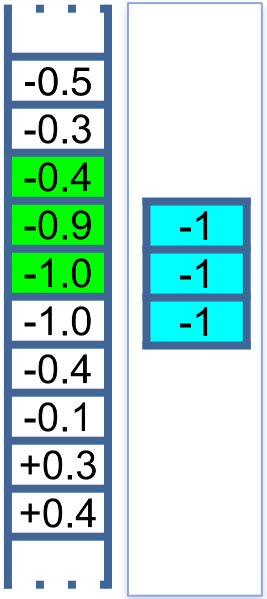
$\sigma(\Sigma w_i * I_i)$

$\Sigma w_i * I_i = 1.6$

1D - SOUND - FILTERS - CONVOLUTIONS

- Detect whether there is a silence





...

-0.5

-0.3

-0.4

-0.9

-1.0

-1.0

-0.4

-0.1

+0.3

+0.4

...

-1

-1

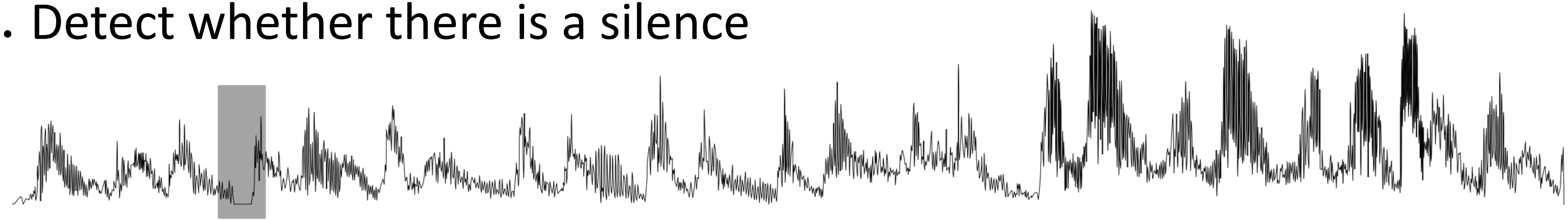
-1

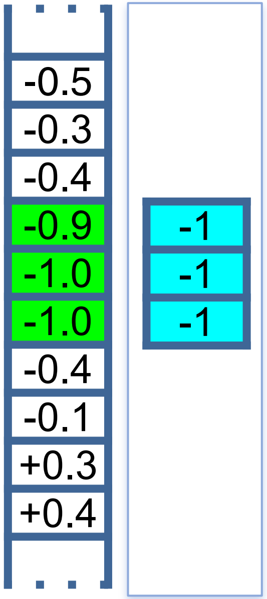
$\sigma(\Sigma w_i * I_i)$

$\Sigma w_i * I_i = 2.3$

1D - SOUND - FILTERS - CONVOLUTIONS

- Detect whether there is a silence





-0.5
-0.3
-0.4
-0.9
-1.0
-1.0
-0.4
-0.1
+0.3
+0.4

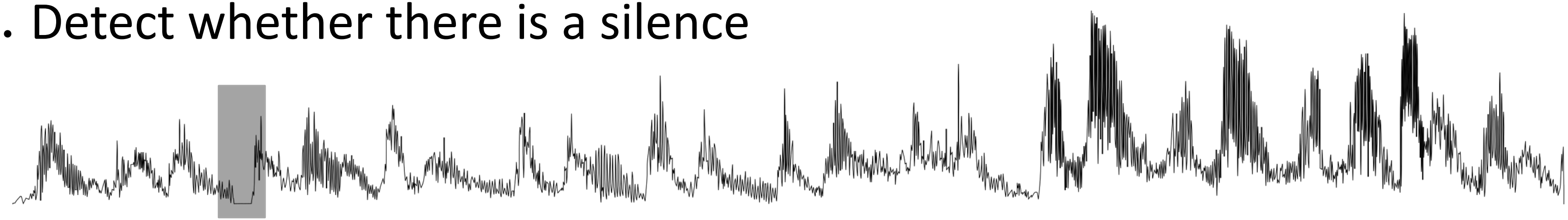
-1
-1
-1

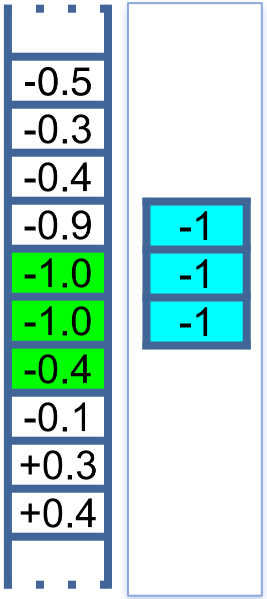
$\sigma(\Sigma w_i * I_i)$

$\Sigma w_i * I_i = 2.9$

1D - SOUND - FILTERS - CONVOLUTIONS

- Detect whether there is a silence





...

-0.5

-0.3

-0.4

-0.9

-1.0

-1.0

-0.4

-0.1

+0.3

+0.4

...

-1

-1

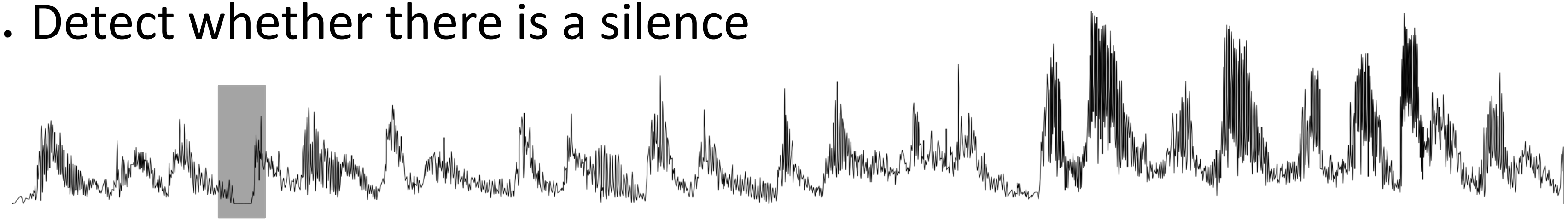
-1

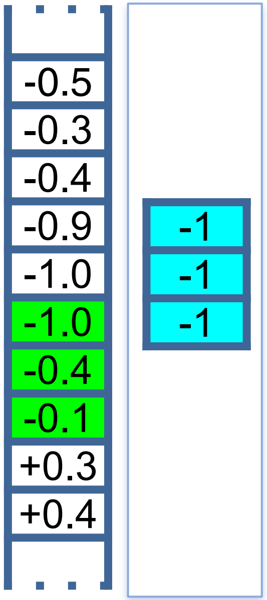
$\sigma(\Sigma w_i * I_i)$

$\Sigma w_i * I_i = 2.4$

1D - SOUND - FILTERS - CONVOLUTIONS

- Detect whether there is a silence



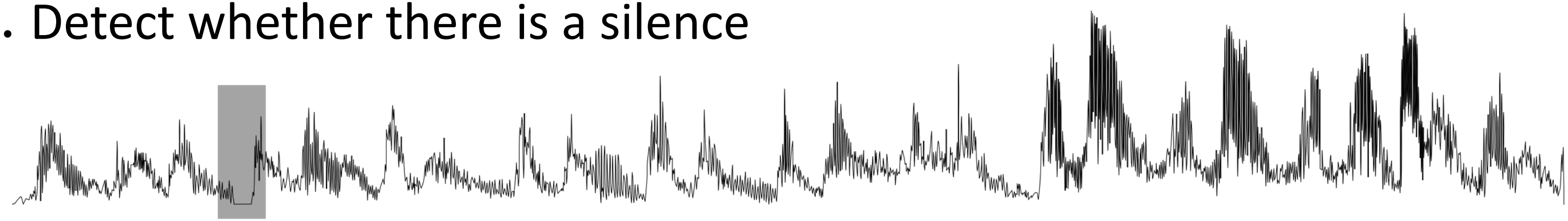


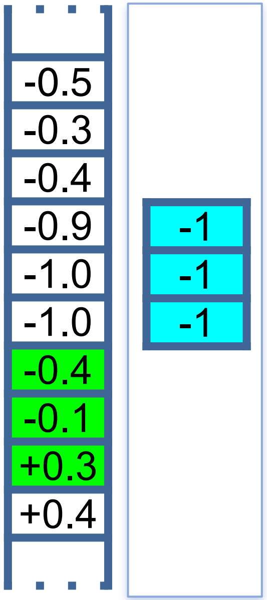
$\sigma(\Sigma w_i * I_i)$

$\Sigma w_i * I_i = 1.5$

1D - SOUND - FILTERS - CONVOLUTIONS

- Detect whether there is a silence





-0.5
-0.3
-0.4
-0.9
-1.0
-1.0
-0.4
-0.1
+0.3
+0.4

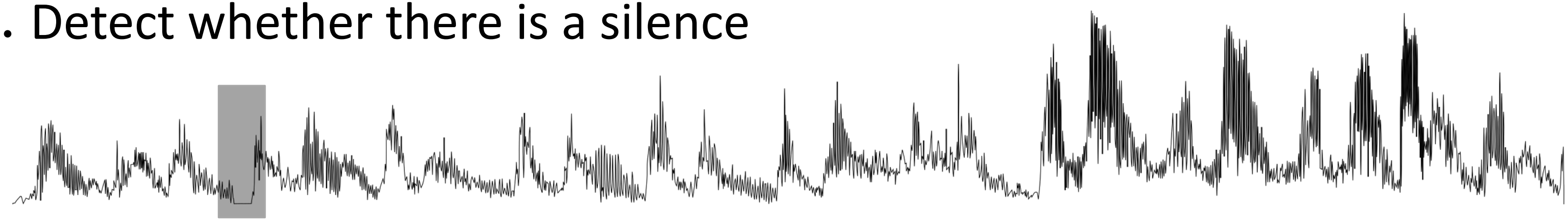
-1
-1
-1

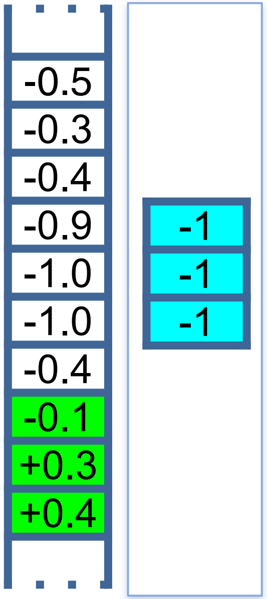
$\sigma(\Sigma w_i * I_i)$

$\Sigma w_i * I_i = 0.2$

1D - SOUND - FILTERS - CONVOLUTIONS

- Detect whether there is a silence





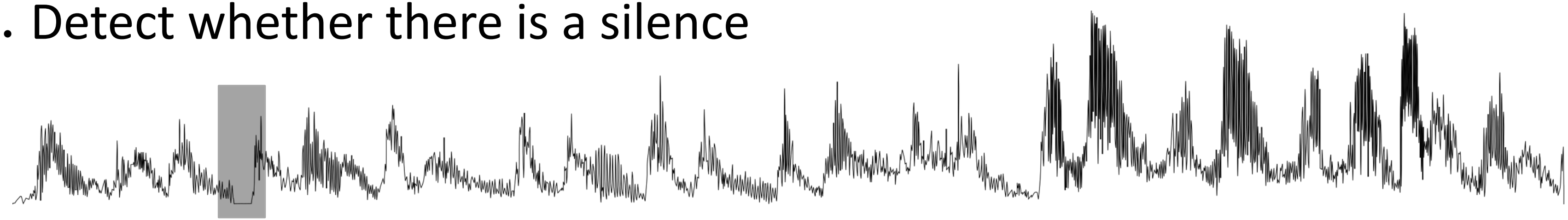
...	
-0.5	
-0.3	
-0.4	
-0.9	-1
-1.0	-1
-1.0	-1
-0.4	
-0.1	
+0.3	
+0.4	
...	

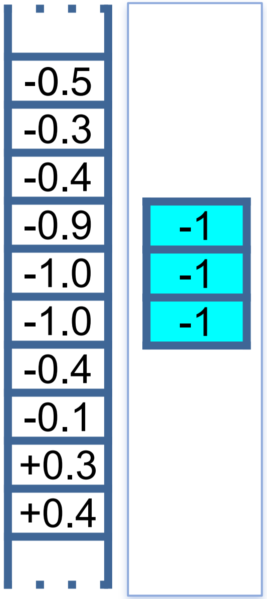
$\sigma(\Sigma w_i * I_i)$

$\Sigma w_i * I_i = -0.6$

1D - SOUND - FILTERS - CONVOLUTIONS

- Detect whether there is a silence





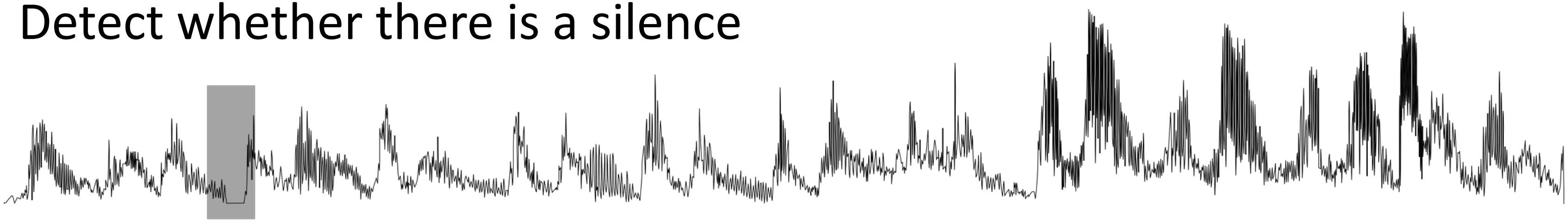
...	
-0.5	
-0.3	
-0.4	
-0.9	-1
-1.0	-1
-1.0	-1
-0.4	
-0.1	
+0.3	
+0.4	
...	

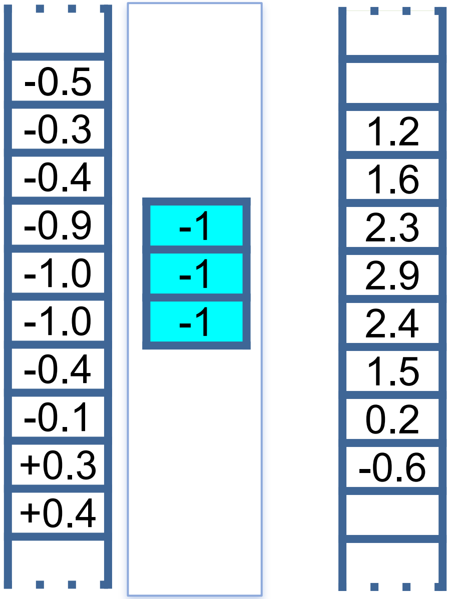
Observations:

- The filter can act as a feature extractor
- The filter can be used at all locations to detect silence
  - We call sliding a filter over all positions **convolving**
    - The operation is called a **convolution operator**

1D - SOUND - FILTERS - CONVOLUTIONS

- Detect whether there is a silence





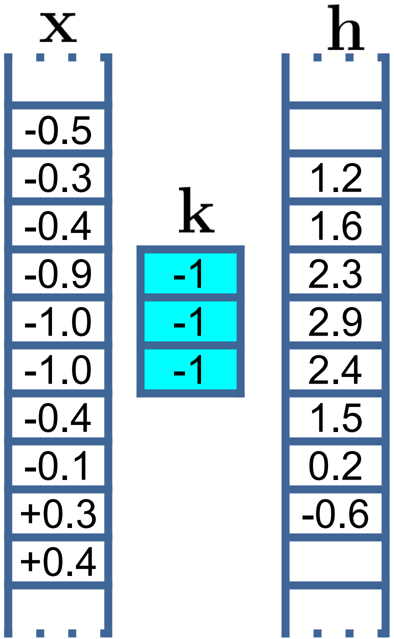
- The output of the convolution operator is itself again a vector!
- A time shift in the input causes the same shift in the output: Convolution is **translation equivariant**
- This operation can be understood as **a small MLP** that wanders across the input signal.
- In practice, the conv. kernels are **learned.**

PART TWO-b: conv1D

1D CONVOLUTION - FORMAL DEFINITION

- To understand how the conv. kernels are learned, we require a formal definition. For a 1D input sequence $\mathbf{x} \in \mathbb{R}^n$ and a filter $\mathbf{k} \in \mathbb{R}^{2m+1}$ the convolution operation is:

$$\mathbf{h}(t) = (\mathbf{x} * \mathbf{k})(t) = \sum_{\tau=-m}^{m} \mathbf{x}(t-\tau) \cdot \mathbf{k}(\tau)$$



$\mathbf{x}$	$\mathbf{k}$	$\mathbf{h}$
...		...
-0.5		
-0.3		1.2
-0.4		1.6
-0.9	-1	2.3
-1.0	-1	2.9
-1.0	-1	2.4
-0.4		1.5
-0.1		0.2
+0.3		-0.6
+0.4		
...		...

1D CONVOLUTION - FORMAL DEFINITION

- How do we **learn** the filter $\mathbf{k}$?
- If $\mathbf{k}$ is learned, we will update the filter weights based on some loss
- $\mathcal{L}_{\mathbf{h}} = \sum_{t=0}^{n} \mathcal{L}_{\mathbf{h}}(t)$ depending on the conv. response at all places, e.g., cross-entropy for classification.
- The gradient utilized to update a kernel weight $\mathbf{k}(\tau_0)$ is given by:

$$\mathbf{h}(t) = (\mathbf{x} * \mathbf{k})(t) = \sum_{\tau=-m}^{m} \mathbf{x}(t-\tau) \cdot \mathbf{k}(\tau)$$

$$\frac{\partial \mathcal{L}_{\mathbf{h}}}{\partial \mathbf{k}(\tau_0)} = \frac{\partial \mathcal{L}_{\mathbf{h}}}{\partial \mathbf{h}} \frac{\partial \mathbf{h}}{\partial \mathbf{k}(\tau_0)} = \sum_{t=0}^{n} \frac{\partial \mathcal{L}_{\mathbf{h}}(t)}{\partial \mathbf{h}(t)} \sum_{\tau=-m}^{m} \mathbf{x}(t-\tau) \cdot \boxed{\frac{\partial \mathbf{k}(\tau)}{\partial \mathbf{k}(\tau_0)}}$$

Always zero, except when tau_0 = tau

1D CONVOLUTION - FORMAL DEFINITION

$$\mathbf{h}(t) = (\mathbf{x} * \mathbf{k})(t) = \sum_{\tau=-m}^{m} \mathbf{x}(t-\tau) \cdot \mathbf{k}(\tau)$$

$$\frac{\partial \mathcal{L}_{\mathbf{h}}}{\partial \mathbf{k}(\tau_0)} = \frac{\partial \mathcal{L}_{\mathbf{h}}}{\partial \mathbf{h}} \frac{\partial \mathbf{h}}{\partial \mathbf{k}(\tau_0)} = \boxed{\sum_{t=0}^{n}} \frac{\partial \mathcal{L}_{\mathbf{h}}(t)}{\partial \mathbf{h}(t)} \sum_{\tau=-m}^{m} \mathbf{x}(t-\tau) \cdot \boxed{\frac{\partial \mathbf{k}(\tau)}{\partial \mathbf{k}(\tau_0)}}$$

Always zero,
except when
\tau_0 == \tau

The update of the weights takes into consideration **ALL ELEMENTS OF THE INPUT SEQUENCE INTO ACCOUNT!**

- The weights are shared across the entire input (MLPs learn independent weights at every position: $\mathbf{h}(t) = \mathbf{W}(t,:) \cdot \mathbf{x}(t)$)

1D CONVOLUTION - FORMAL DEFINITION

- **Advantages:**
- Since weights are shared for every position, Convolutional Networks (CNNs) are much MUCH! **MUCH!** smaller than MLPs. -> **PARAMETER EFFICIENCY**
- Convolutions can learn a powerful pattern recognizers, e.g., for silence, based on “silences” appearing everywhere in the input (MLPs must learn an independent “silence recognizer” for every position) -> **DATA EFFICIENCY**
- Convolutions can recognize a “silence pattern” regardless of where it appears (MLPs must have seen silence at a given position before in order to recognize it) -> **GENERALIZATION IMPROVEMENTS**

IMPORTANT: 2 and 3 are a consequence of convolution being translation equivariant.

1D - CONVOLUTIONS

Up till now:

- We can create filters
  - MLP in disguise
  - We can convolve them over the input which creates an output vector

Coming next:

- We need multiple filters
- What do we do at the start and end of the data?
- What at the next layer?
- How do we get the dimension down?

1D - CONVOLUTIONS

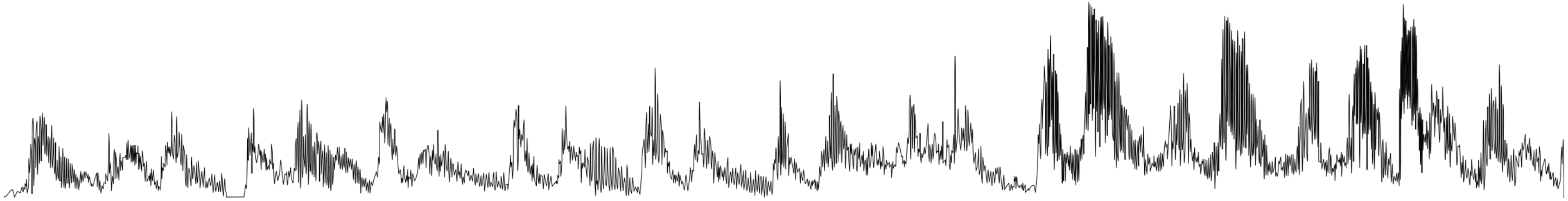
Up till now:

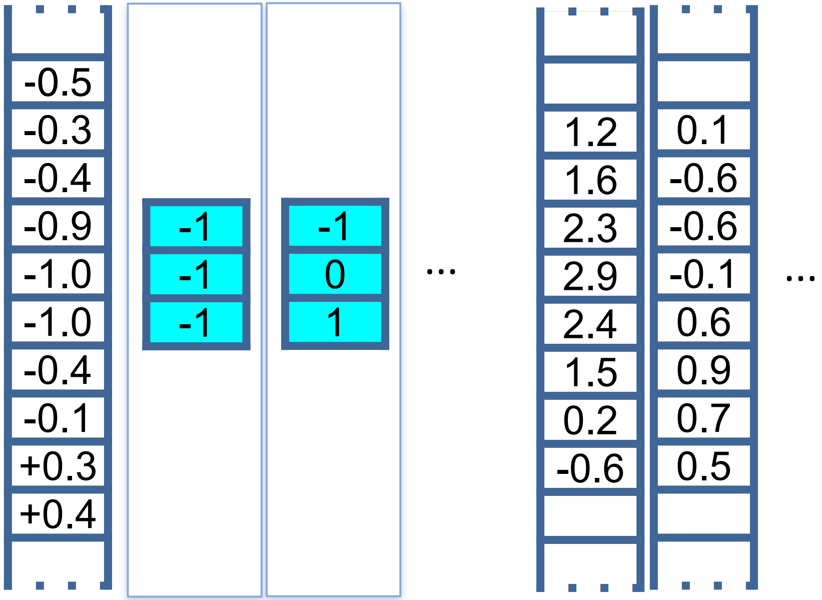
- We can create filters
  - MLP in disguise
  - We can convolve them over the input which creates an output vector

Coming next:

- We need multiple filters
- What at the next layer?
- What do we do at the start and end of the data?
- How do we get the dimension down?

1D - CONVOLUTIONS - MULTIPLE FILTERS





-0.5
-0.3
-0.4
-0.9
-1.0
-1.0
-0.4
-0.1
+0.3
+0.4

-1
-1
-1

-1
0
1

…

1.2
1.6
2.3
2.9
2.4
1.5
0.2
-0.6

0.1
-0.6
-0.6
-0.1
0.6
0.9
0.7
0.5

…

- We need to have all sorts of filters for feature extraction
  - We can have as many filters as we want
  - Now, the output becomes a matrix, called the **output volume**

1D - CONVOLUTIONS

Up till now:

- We can create filters
  - MLP in disguise
  - We can convolve them over the input which creates an output vector

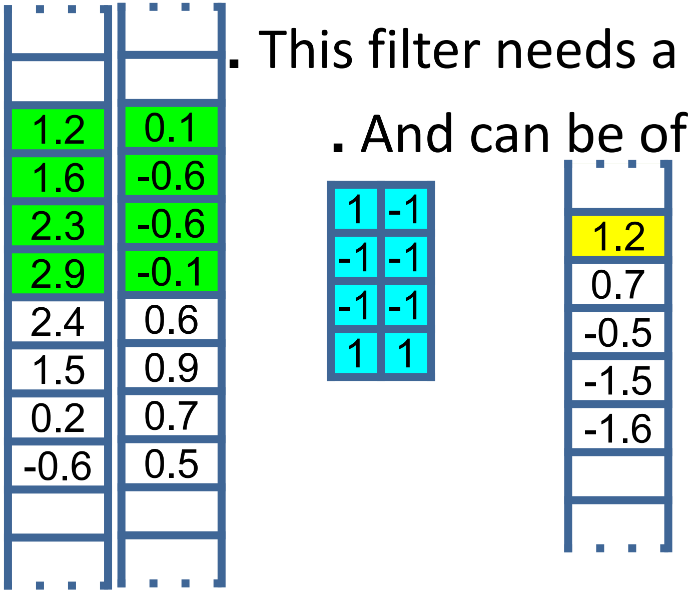
Coming next:

- We need multiple filters
- What at the next layer?
- What do we do at the start and end of the data?
- How do we get the dimension down?

1D - CONVOLUTIONS - FILTERS AT FURTHER LAYERS

What do we do at the next layer?

- We have the output volume of the previous layer and we will just define a convolution operator over that!
  - This filter needs a second dimension!
    - And can be of a different size.



1.2	0.1
1.6	-0.6
2.3	-0.6
2.9	-0.1
2.4	0.6
1.5	0.9
0.2	0.7
-0.6	0.5

1	-1
-1	-1
-1	-1
1	1

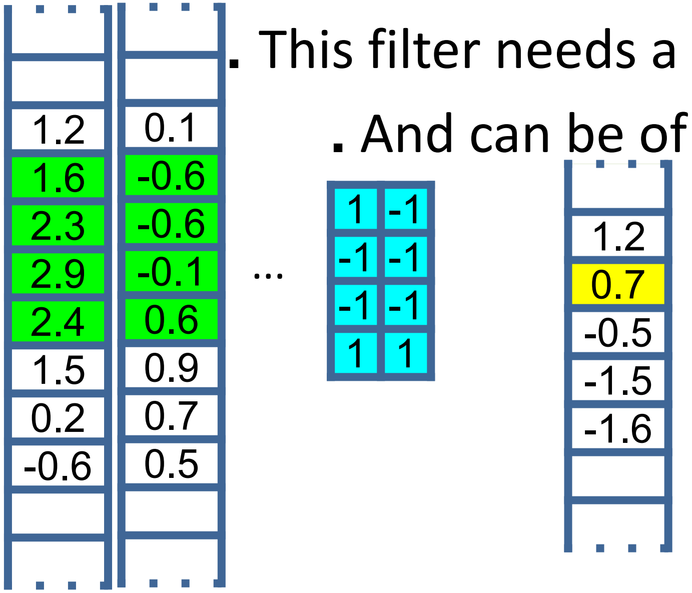
1.2
0.7
-0.5
-1.5
-1.6

For the sake of the example, we did not apply the bias and non-linearity!!, which you would do in practice.

1D - CONVOLUTIONS - FILTERS AT FURTHER LAYERS

What do we do at the next layer?

- We have the output volume of the previous layer and we will just define a convolution operator over that!
  - This filter needs a second dimension!
    - And can be of a different size.



1.2	0.1
1.6	-0.6
2.3	-0.6
2.9	-0.1
2.4	0.6
1.5	0.9
0.2	0.7
-0.6	0.5

…

1	-1
-1	-1
-1	-1
1	1

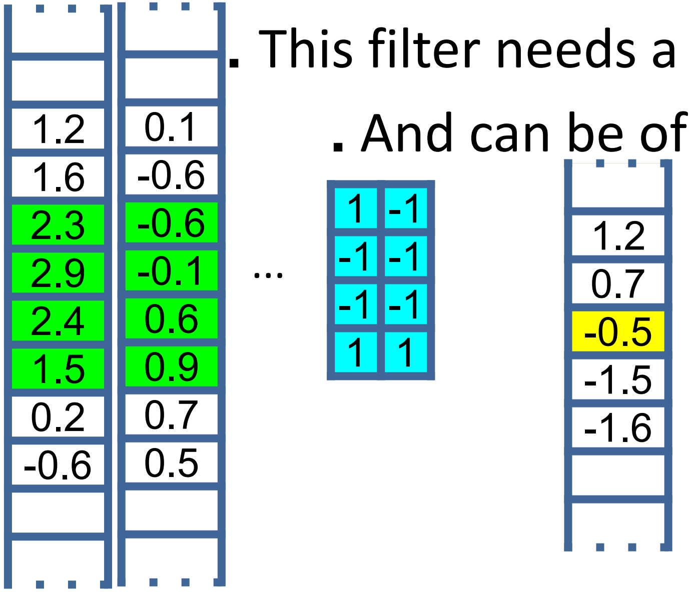
1.2
0.7
-0.5
-1.5
-1.6

For the sake of the example, we did not apply the bias and non-linearity!!, which you would do in practice.

1D - CONVOLUTIONS - FILTERS AT FURTHER LAYERS

What do we do at the next layer?

- We have the output volume of the previous layer and we will just define a convolution operator over that!
  - This filter needs a second dimension!
    - And can be of a different size.



1.2	0.1
1.6	-0.6
2.3	-0.6
2.9	-0.1
2.4	0.6
1.5	0.9
0.2	0.7
-0.6	0.5

…

1	-1
-1	-1
-1	-1
1	1

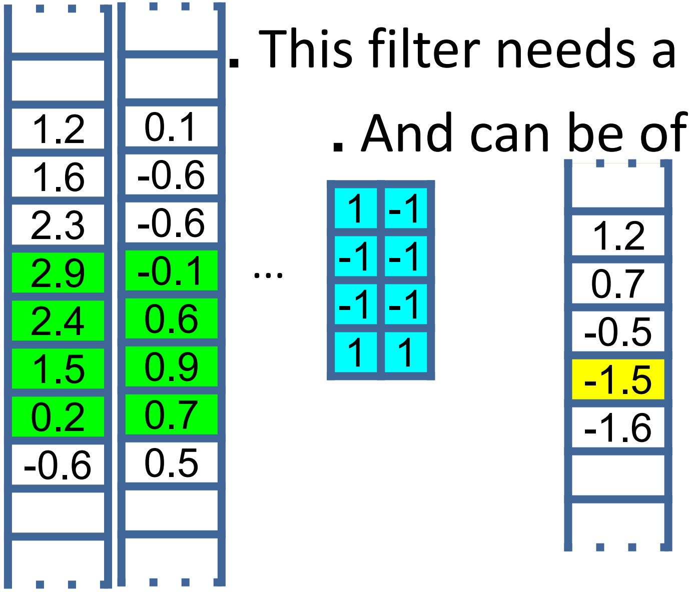
1.2
0.7
-0.5
-1.5
-1.6

For the sake of the example, we did not apply the bias and non-linearity!!, which you would do in practice.

1D - CONVOLUTIONS - FILTERS AT FURTHER LAYERS

What do we do at the next layer?

- We have the output volume of the previous layer and we will just define a convolution operator over that!
  - This filter needs a second dimension!
    - And can be of a different size.



1.2	0.1
1.6	-0.6
2.3	-0.6
2.9	-0.1
2.4	0.6
1.5	0.9
0.2	0.7
-0.6	0.5

…

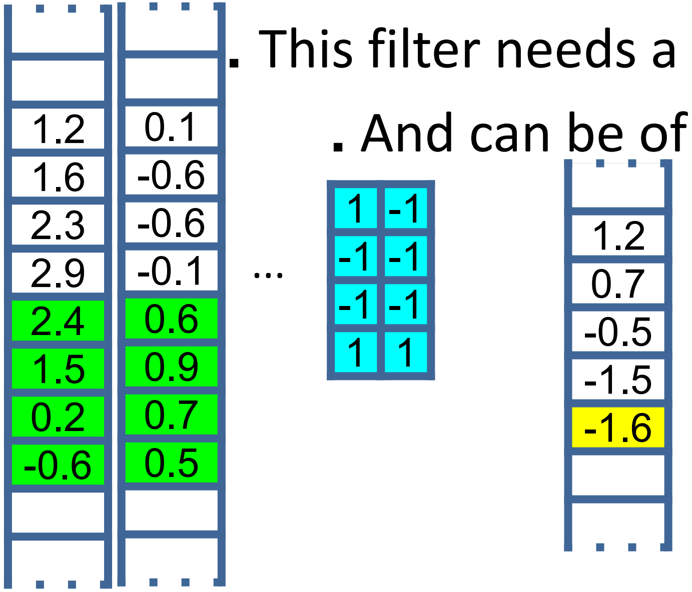
1	-1
-1	-1
-1	-1
1	1

1.2
0.7
-0.5
-1.5
-1.6

For the sake of the example, we did not apply the bias and non-linearity!!, which you would do in practice.

What do we do at the next layer?

- We have the output volume of the previous layer and we will just define a convolution operator over that!
  - This filter needs a second dimension!
    - And can be of a different size.



1.2	0.1
1.6	-0.6
2.3	-0.6
2.9	-0.1
2.4	0.6
1.5	0.9
0.2	0.7
-0.6	0.5

…

1	-1
-1	-1
-1	-1
1	1

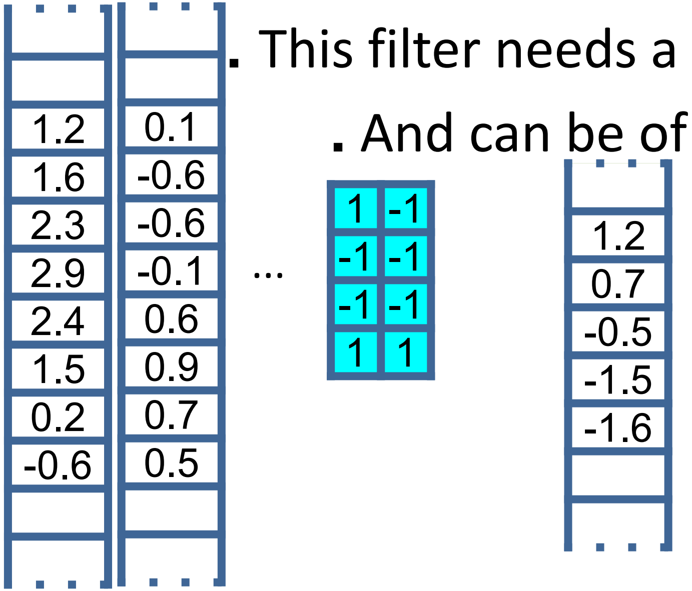
1.2
0.7
-0.5
-1.5
-1.6

For the sake of the example, we did not apply the bias and non-linearity!!, which you would do in practice.

1D - CONVOLUTIONS - FILTERS AT FURTHER LAYERS

What do we do at the next layer?

- We have the output volume of the previous layer and we will just define a convolution operator over that!
  - This filter needs a second dimension!
    - And can be of a different size.
  - The meaning of these filters recursively depends on the meaning of the filters on the layer before. But, overall becomes more complex.



1.2	0.1
1.6	-0.6
2.3	-0.6
2.9	-0.1
2.4	0.6
1.5	0.9
0.2	0.7
-0.6	0.5

1	-1
-1	-1
-1	-1
1	1

1.2
0.7
-0.5
-1.5
-1.6

For the sake of the example, we did not apply the bias and non-linearity!!, which you would do in practice.

1D - CONVOLUTIONS

Up till now:

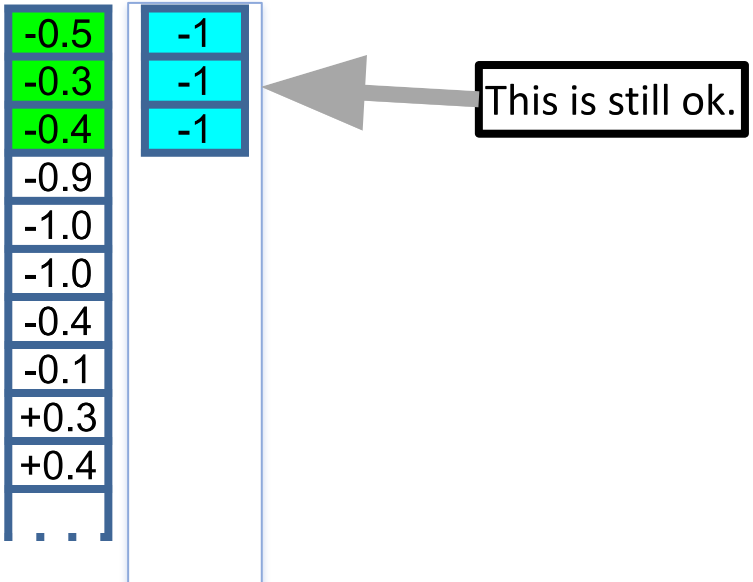
- We can create filters
  - MLP in disguise
  - We can convolve them over the input which creates an output vector

Coming next:

- We need multiple filters
- What at the next layer?
- What do we do at the start and end of the data?
- How do we get the dimension down?

1D - CONVOLUTIONS - PADDING

- Near the boundaries of the data, we cannot apply the convolution as we normally do:



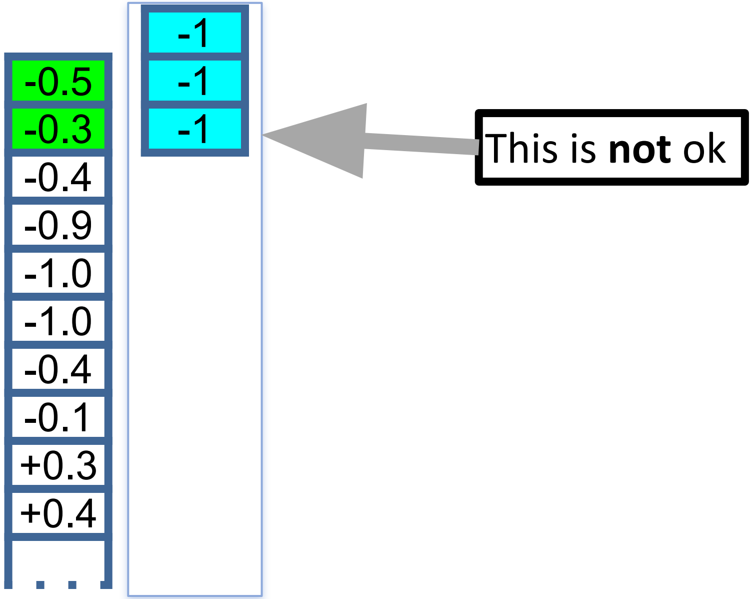
-0.5
-0.3
-0.4
-0.9
-1.0
-1.0
-0.4
-0.1
+0.3
+0.4
. . .

-1
-1
-1

This is still ok.

1D - CONVOLUTIONS - PADDING

- Near the boundaries of the data, we cannot apply the convolution as we normally do:



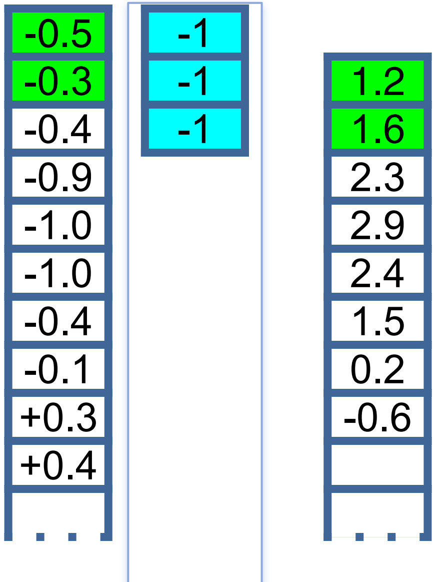
-0.5
-0.3
-0.4
-0.9
-1.0
-1.0
-0.4
-0.1
+0.3
+0.4
. . .

-1
-1
-1

This is **not** ok

1D - CONVOLUTIONS - PADDING

- Near the boundaries of the data, we cannot apply the convolution as we normally do:



-0.5
-0.3
-0.4
-0.9
-1.0
-1.0
-0.4
-0.1
+0.3
+0.4
. . .

-1
-1
-1

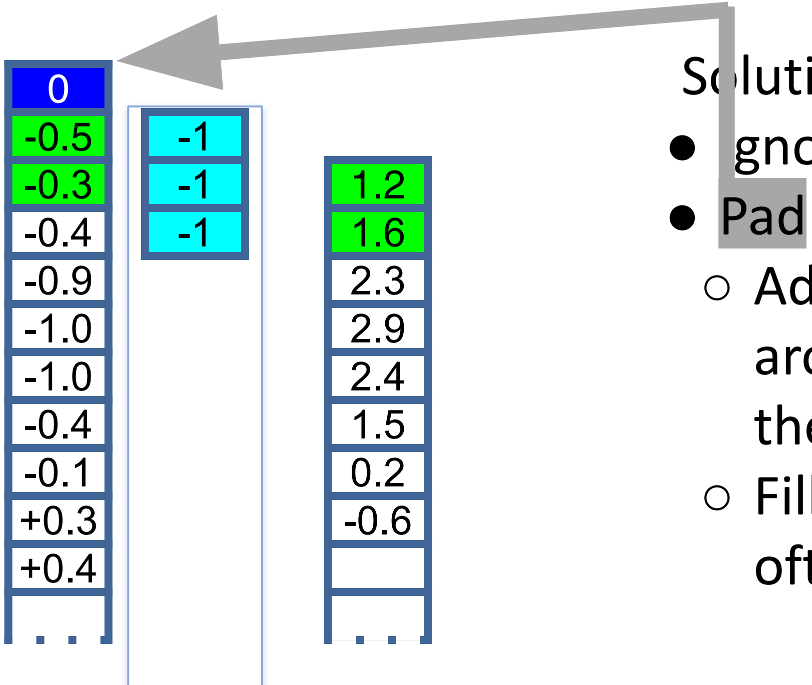
1.2
1.6
2.3
2.9
2.4
1.5
0.2
-0.6
. . .

Solutions:

- Ignore the boundaries
  - This also leads to a reduction of dimension!
  - information at the boundaries gets lost

1D - CONVOLUTIONS - PADDING

- Near the boundaries of the data, we cannot apply the convolution as we normally do:

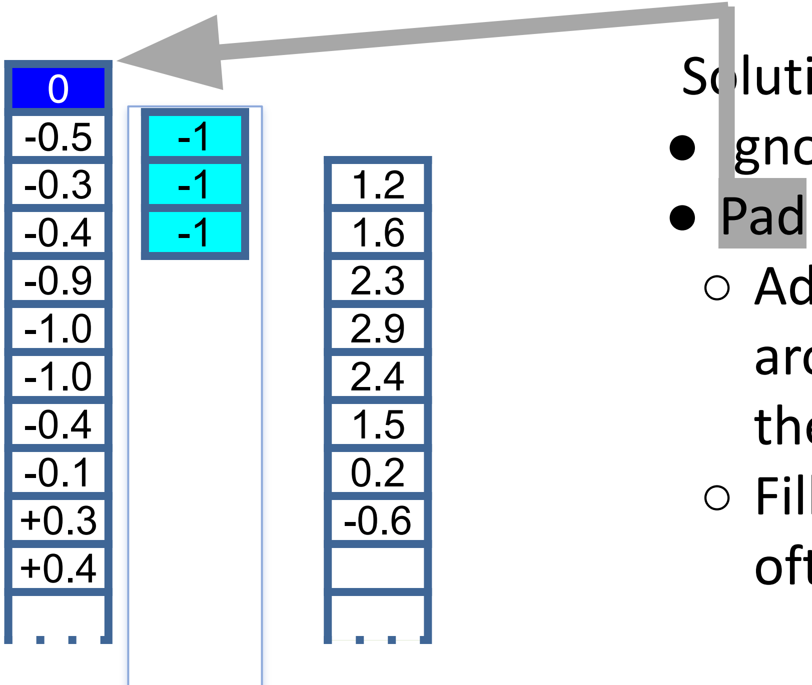


Solutions:

- Ignore the boundaries
- Pad the boundaries
  - Add (filter length-1)/2 around the data to preserve the dimension
  - Fill this with a fixed value, often 0

1D - CONVOLUTIONS - PADDING

- Near the boundaries of the data, we cannot apply the convolution as we normally do:

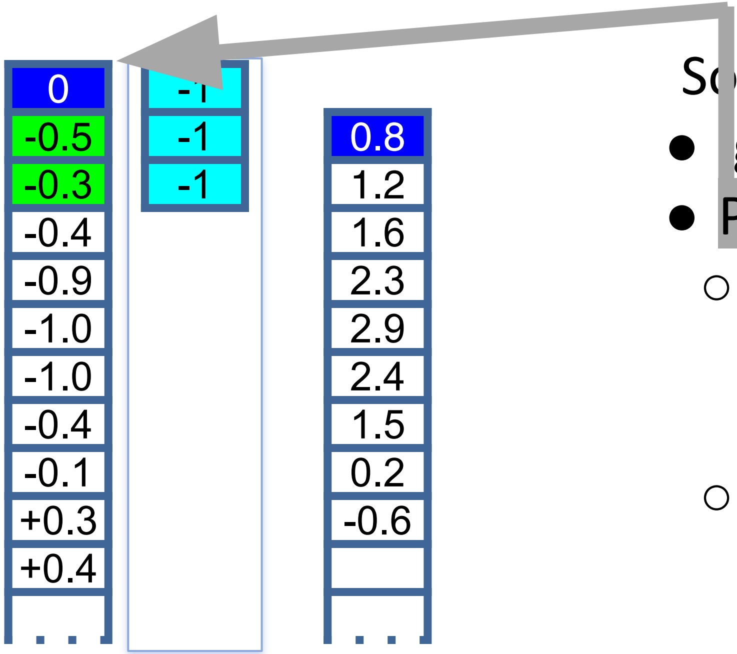


Solutions:

- Ignore the boundaries
- Pad the boundaries
  - Add (filter length-1)/2 around the data to preserve the dimension
  - Fill this with a fixed value, often 0

1D - CONVOLUTIONS - PADDING

- Near the boundaries of the data, we cannot apply the convolution as we normally do:



0	-1	
-0.5	-1	0.8
-0.3	-1	1.2
-0.4		1.6
-0.9		2.3
-1.0		2.9
-1.0		2.4
-0.4		1.5
-0.1		0.2
+0.3		-0.6
+0.4		
. . .		. . .

Solutions:

- Ignore the boundaries
- Pad the boundaries
  - Add (filter length-1)/2 around the data to preserve the dimension
  - Fill this with a fixed value, often 0

1D - CONVOLUTIONS

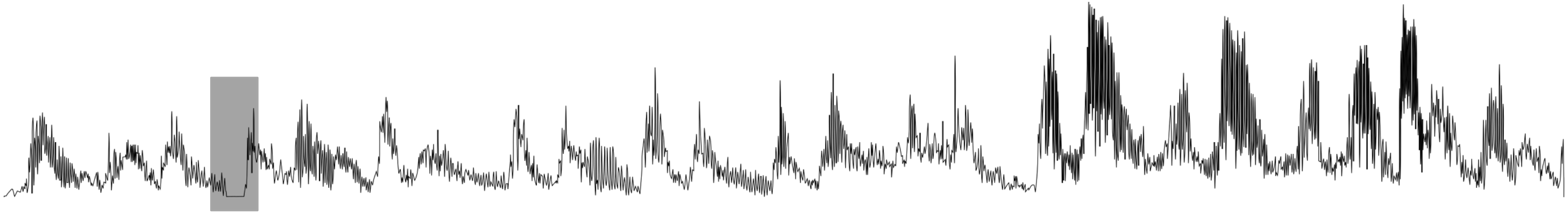
Up till now:

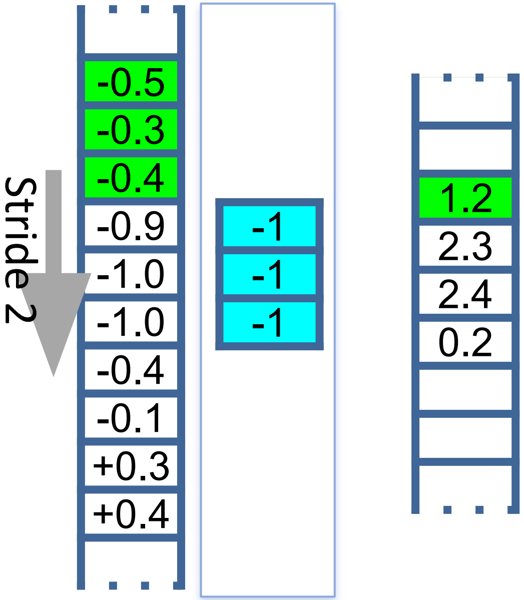
- We can create filters
  - MLP in disguise
  - We can convolve them over the input which creates an output vector

Coming next:

- We need multiple filters
- What at the next layer?
- What do we do at the start and end of the data?
- How do we get the dimension down?

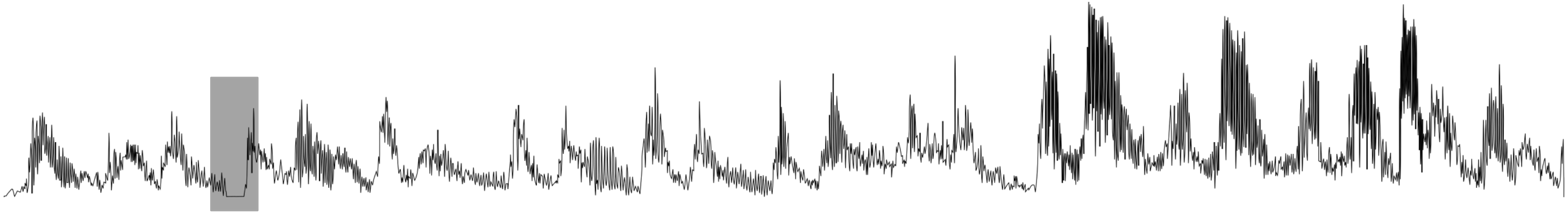
1D - CONVOLUTIONS - STRIDE

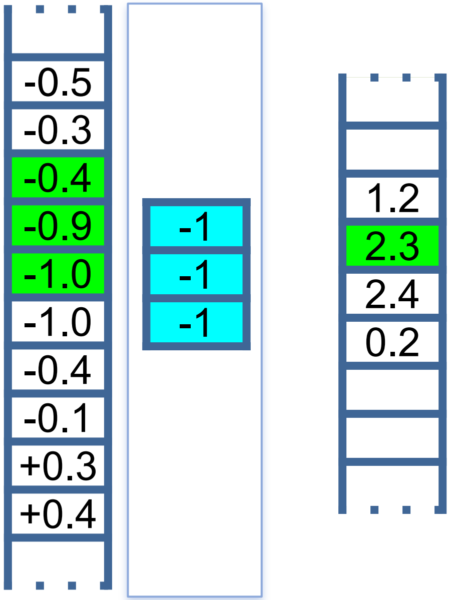




- We need to reduce dimension for the final classification
    - Solution 1: we take larger steps with our filter
        - the size of the step is called the **stride**

1D - CONVOLUTIONS - STRIDE





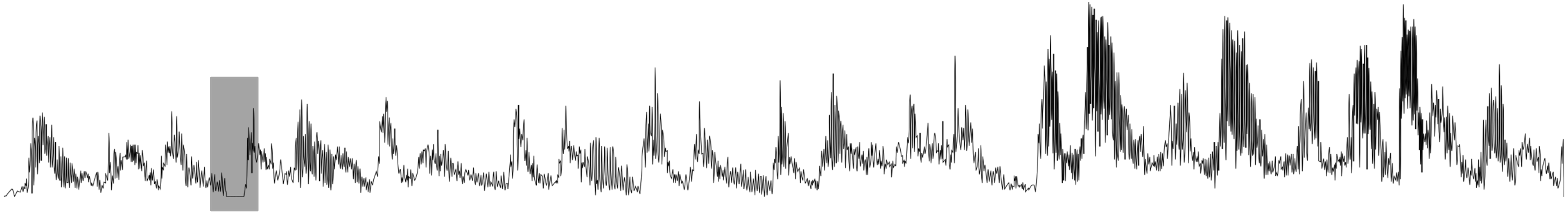
. . .
-0.5
-0.3
-0.4
-0.9
-1.0
-1.0
-0.4
-0.1
+0.3
+0.4
. . .

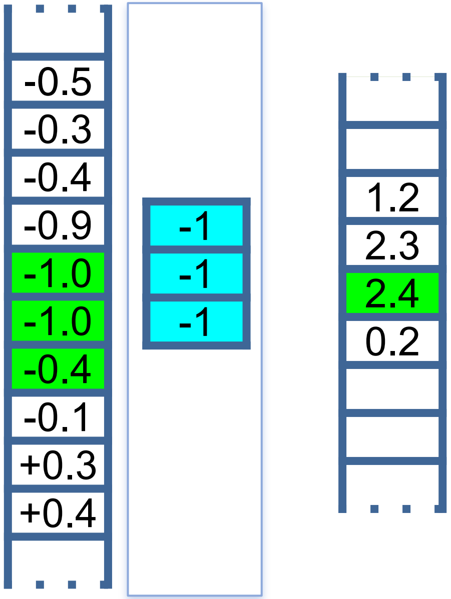
-1
-1
-1

. . .
1.2
2.3
2.4
0.2
. . .

- We need to reduce dimension for the final classification
  - Solution 1: we take larger steps with our filter
    - the size of the step is called the **stride**

1D - CONVOLUTIONS - STRIDE





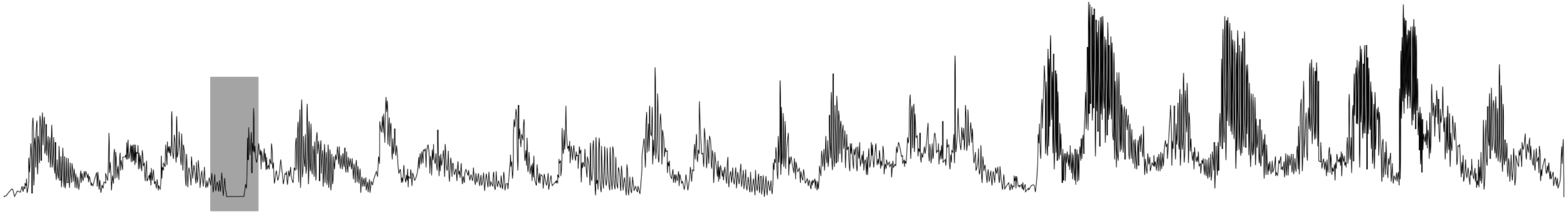
-0.5
-0.3
-0.4
-0.9
-1.0
-1.0
-0.4
-0.1
+0.3
+0.4

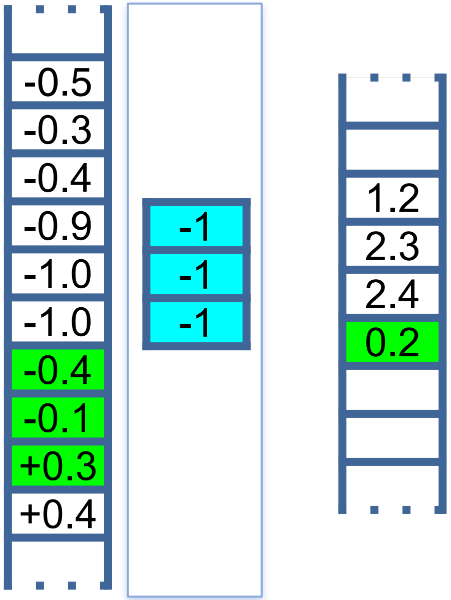
-1
-1
-1

1.2
2.3
2.4
0.2

- We need to reduce dimension for the final classification
  - Solution 1: we take larger steps with our filter
    - the size of the step is called the **stride**

1D - CONVOLUTIONS - STRIDE





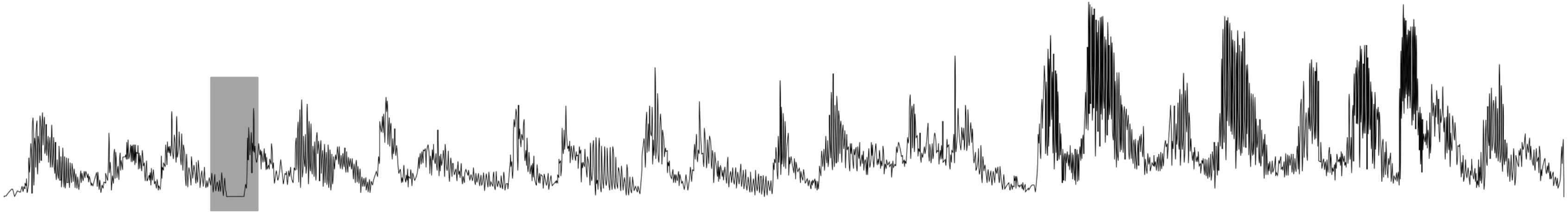
. . .
-0.5
-0.3
-0.4
-0.9
-1.0
-1.0
-0.4
-0.1
+0.3
+0.4
. . .

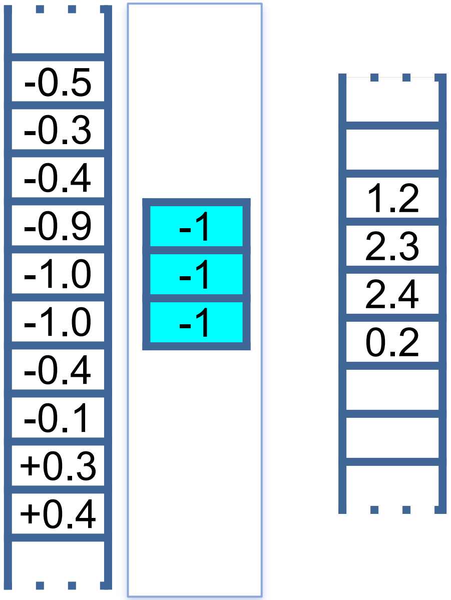
-1
-1
-1

. . .
1.2
2.3
2.4
0.2
. . .

- We need to reduce dimension for the final classification
  - Solution 1: we take larger steps with our filter
    - the size of the step is called the **stride**

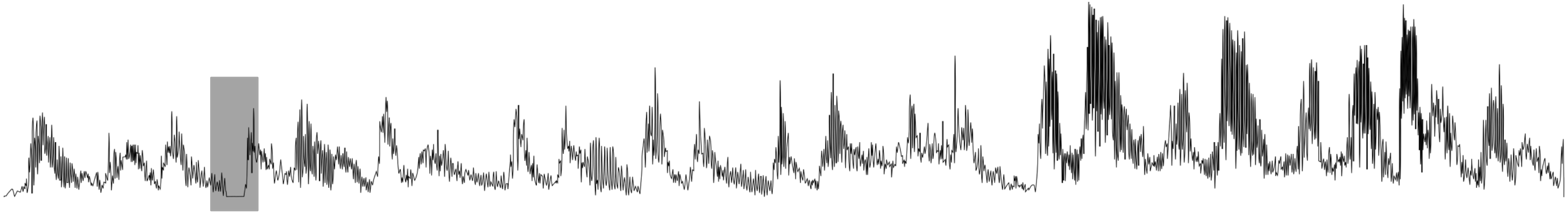
1D - CONVOLUTIONS - STRIDE

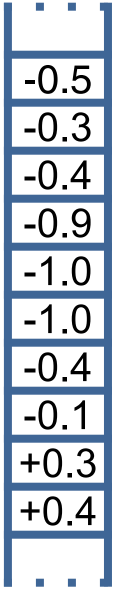




- We need to reduce dimension for the final classification
  - Solution 1: we take larger steps with our filter
    - the size of the step is called the **stride**
    - The dimension reduces with a factor equal to the stride
      - **The input dimension must be a multiple of the stride!**

1D - CONVOLUTIONS - POOLING

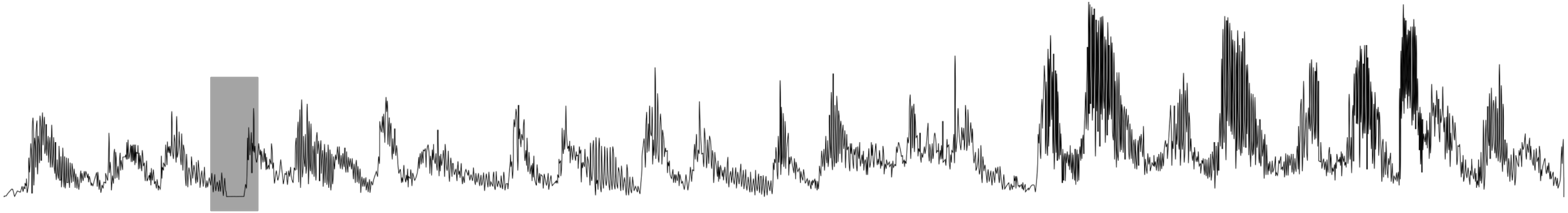


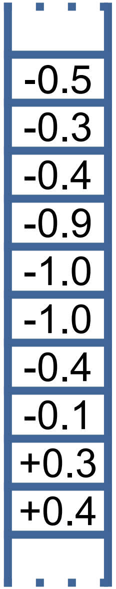


...
-0.5
-0.3
-0.4
-0.9
-1.0
-1.0
-0.4
-0.1
+0.3
+0.4
...

- We need to reduce dimension for the final classification
  - Solution 1: use a larger **stride**
  - Solution 2: use a **pooling layer**

1D - CONVOLUTIONS - POOLING





. . .

-0.5

-0.3

-0.4

-0.9

-1.0

-1.0

-0.4

-0.1

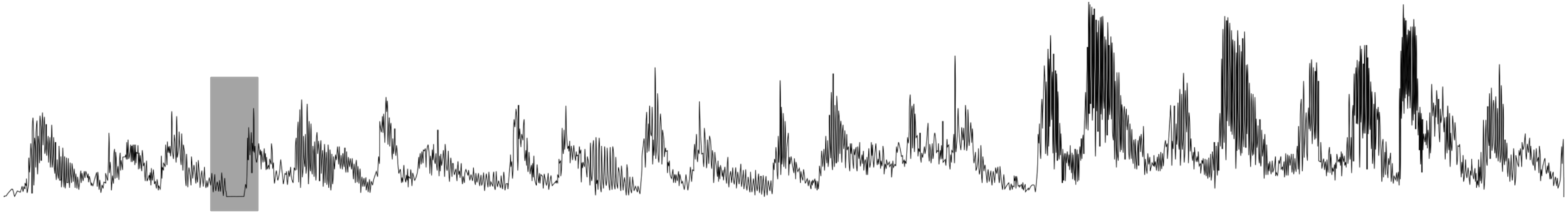
+0.3

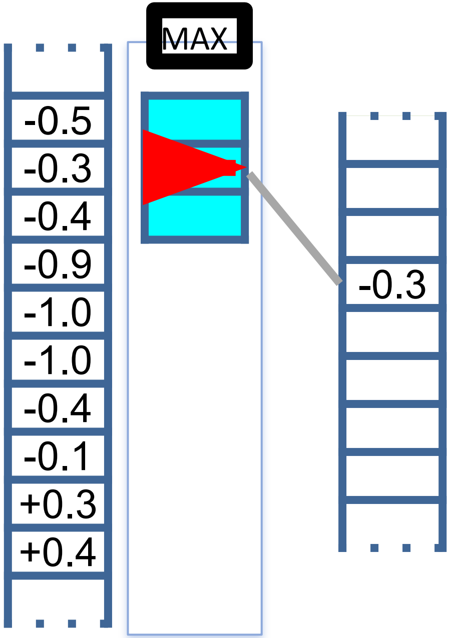
+0.4

. . .

- We need to reduce dimension for the final classification
  - Solution 1: use a larger **stride**
  - Solution 2: use a **pooling layer**
    - Goes over the data similar to a convolution
    - Applies a deterministic function like max or average on the input
    - Usually has stride ==pool size

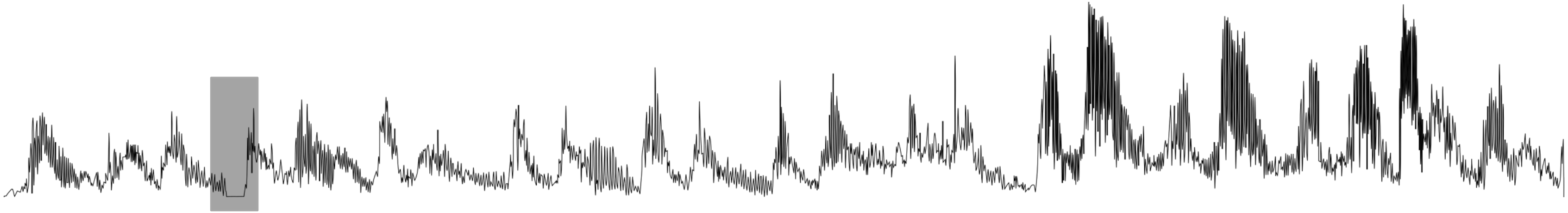
1D - CONVOLUTIONS - POOLING

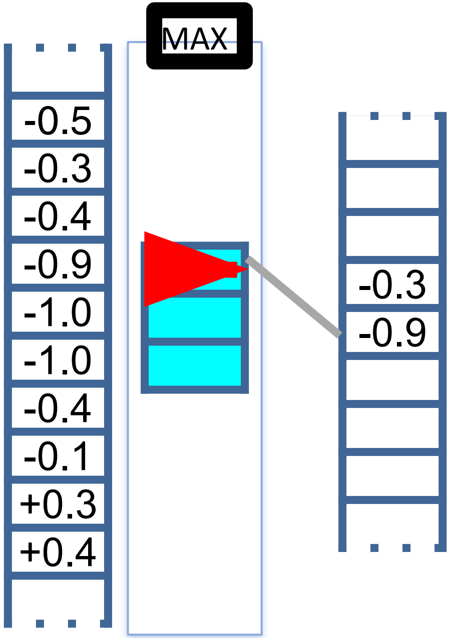




- We need to reduce dimension for the final classification
    - Solution 1: use a larger **stride**
    - Solution 2: use a **pooling layer**
        - Goes over the data similar to a convolution
        - Applies a deterministic function like **max** or average on the input
        - Usually has stride ==pool size

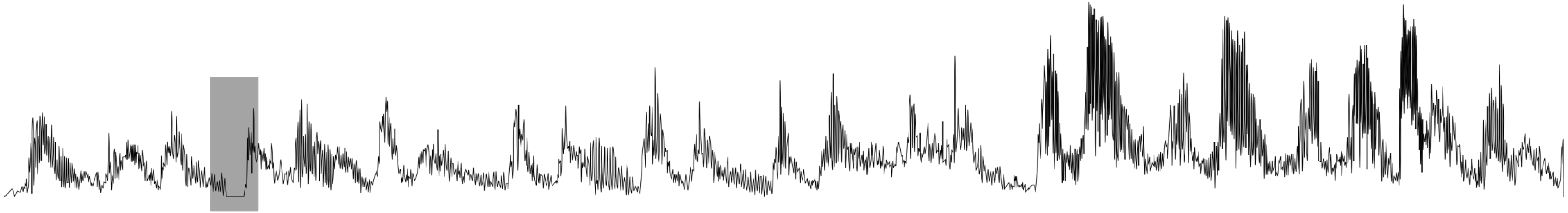
1D - CONVOLUTIONS - POOLING

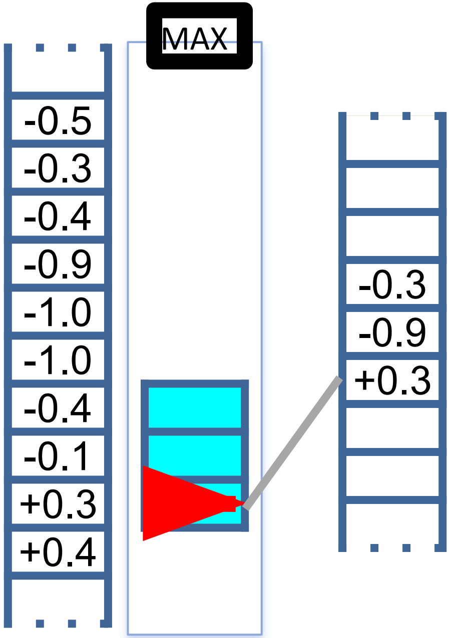




- We need to reduce dimension for the final classification
  - Solution 1: use a larger **stride**
  - Solution 2: use a **pooling layer**
    - Goes over the data similar to a convolution
    - Applies a deterministic function like **max** or average on the input
    - Usually has stride ==pool size

1D - CONVOLUTIONS - POOLING





MAX

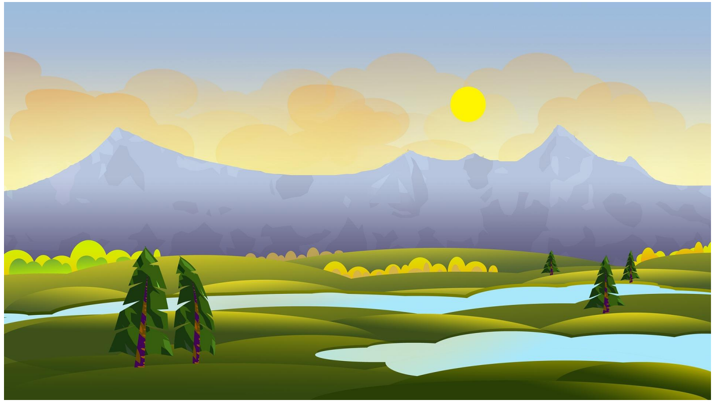
-0.5
-0.3
-0.4
-0.9
-1.0
-1.0
-0.4
-0.1
+0.3
+0.4

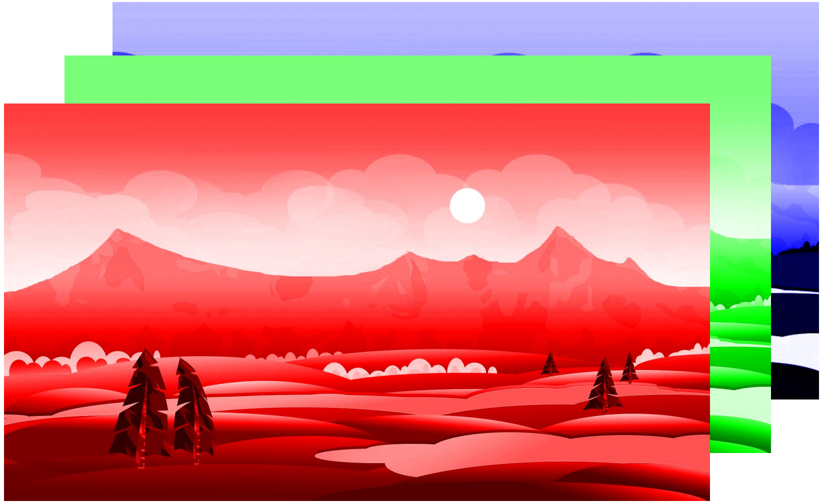
-0.3
-0.9
+0.3

- We need to reduce dimension for the final classification
  - Solution 1: use a larger **stride**
  - Solution 2: use a **pooling layer**
    - Goes over the data similar to a convolution
    - Applies a deterministic function like **max** or average on the input
    - Usually has stride ==pool size

PART THREE: Conv2D, Conv3D, ConvND

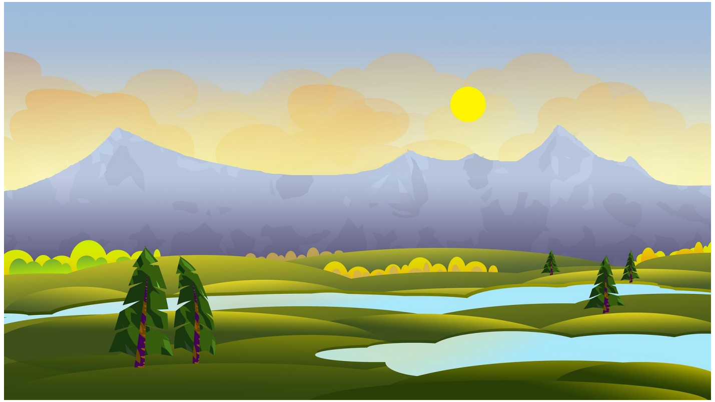
2D - IMAGES - REPRESENTATION

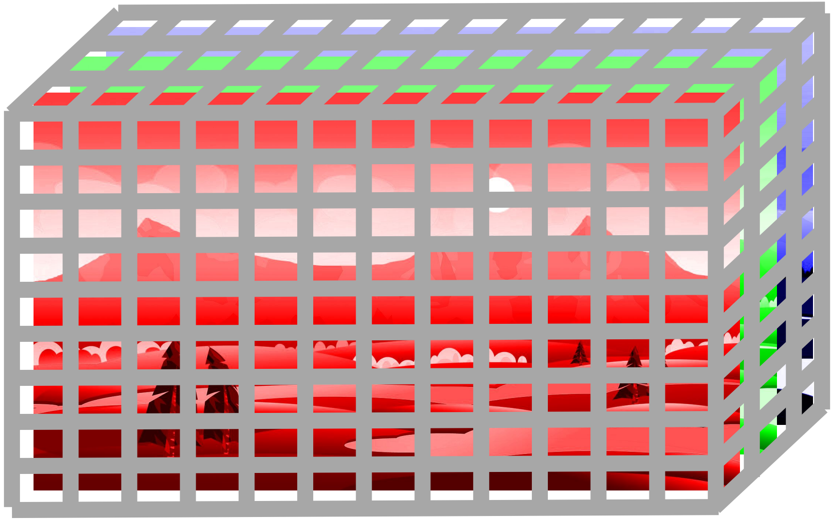




2D - IMAGES - REPRESENTATION - 3D TENSOR

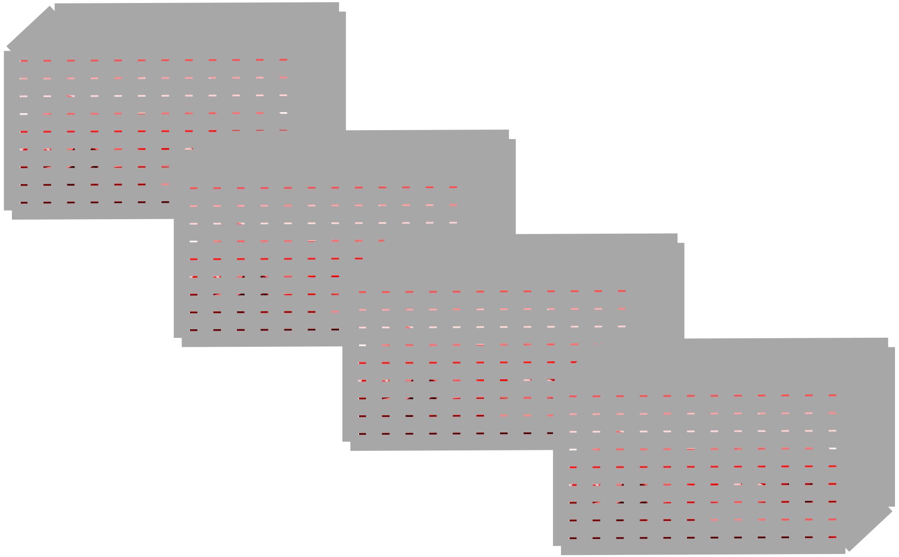
The 2 dimensional color image becomes a 3 dimensional tensor!



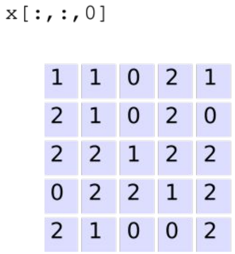


3D - VIDEO - REPRESENTATION - 4D TENSOR

A 3 dimensional color video becomes a 4 dimensional tensor!



2D - IMAGES - CONVOLUTION



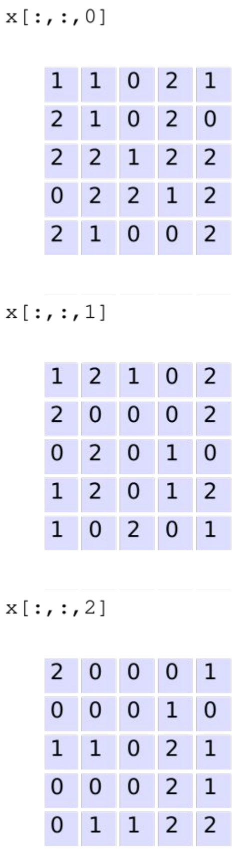
x[:,:,0]

1	1	0	2	1
2	1	0	2	0
2	2	1	2	2
0	2	2	1	2
2	1	0	0	2

We start with a 5x5 image

animation/image source: https://cs231n.github.io/convolutional-networks/#fc

2D - IMAGES - CONVOLUTION - CHANNELS



`x[:,:,0]`

1	1	0	2	1
2	1	0	2	0
2	2	1	2	2
0	2	2	1	2
2	1	0	0	2

`x[:,:,1]`

1	2	1	0	2
2	0	0	0	2
0	2	0	1	0
1	2	0	1	2
1	0	2	0	1

`x[:,:,2]`

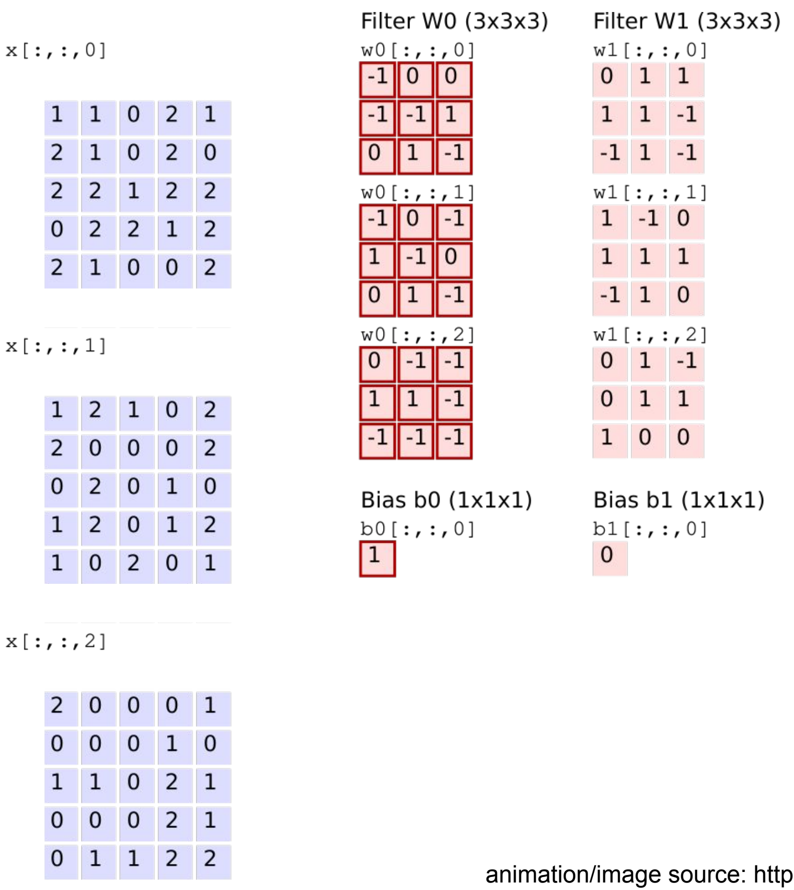
2	0	0	0	1
0	0	0	1	0
1	1	0	2	1
0	0	0	2	1
0	1	1	2	2

We start with a 5x5 image

We have 3 channels

animation/image source: https://cs231n.github.io/convolutional-networks/#fc

2D - IMAGES - CONVOLUTION - FILTERS



x[:,:,0]

1	1	0	2	1
2	1	0	2	0
2	2	1	2	2
0	2	2	1	2
2	1	0	0	2

x[:,:,1]

1	2	1	0	2
2	0	0	0	2
0	2	0	1	0
1	2	0	1	2
1	0	2	0	1

x[:,:,2]

2	0	0	0	1
0	0	0	1	0
1	1	0	2	1
0	0	0	2	1
0	1	1	2	2

Filter W0 (3x3x3)

w0[:,:,0]

-1	0	0
-1	-1	1
0	1	-1

w0[:,:,1]

-1	0	-1
1	-1	0
0	1	-1

w0[:,:,2]

0	-1	-1
1	1	-1
-1	-1	-1

Bias b0 (1x1x1)

b0[:,:,0]

1

Filter W1 (3x3x3)

w1[:,:,0]

0	1	1
1	1	-1
-1	1	-1

w1[:,:,1]

1	-1	0
1	1	1
-1	1	0

w1[:,:,2]

0	1	-1
0	1	1
1	0	0

Bias b1 (1x1x1)

b1[:,:,0]

0

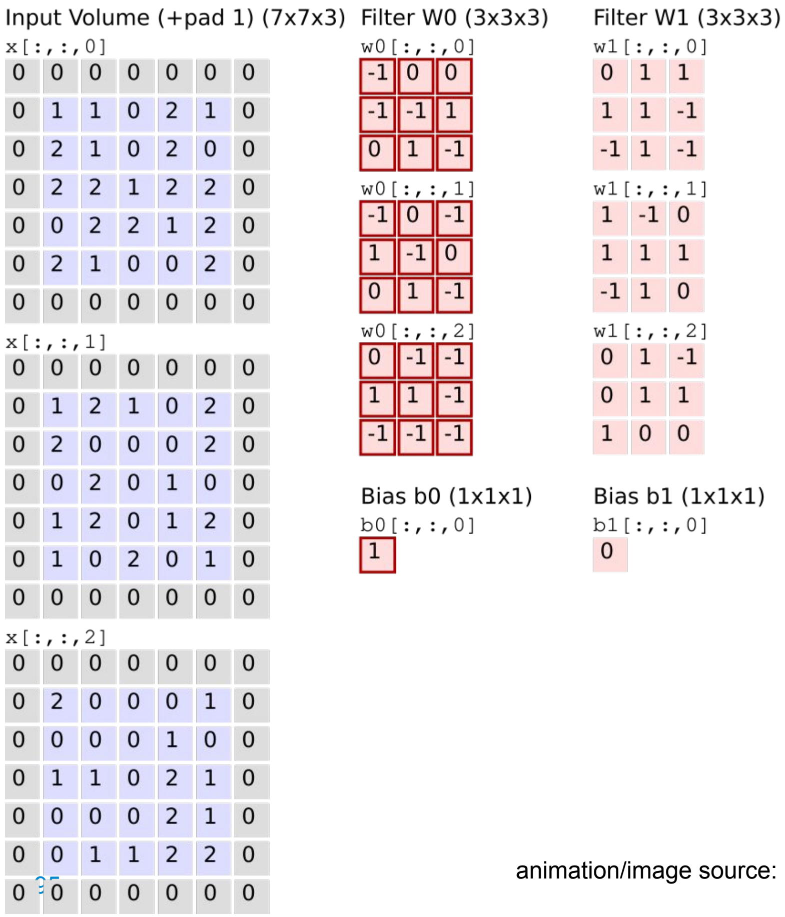
We start with a 5x5 image

We have 3 channels

We want to use 2 filters, these are themselves 3 dimensional

animation/image source: https://cs231n.github.io/convolutional-networks/#fc

2D - IMAGES - CONVOLUTION - PADDING



Input Volume (+pad 1) (7x7x3)

x[:,:,0]

0	0	0	0	0	0	0
0	1	1	0	2	1	0
0	2	1	0	2	0	0
0	2	2	1	2	2	0
0	0	2	2	1	2	0
0	2	1	0	0	2	0
0	0	0	0	0	0	0

x[:,:,1]

0	0	0	0	0	0	0
0	1	2	1	0	2	0
0	2	0	0	0	2	0
0	0	2	0	1	0	0
0	1	2	0	1	2	0
0	1	0	2	0	1	0
0	0	0	0	0	0	0

x[:,:,2]

0	0	0	0	0	0	0
0	2	0	0	0	1	0
0	0	0	0	1	0	0
0	1	1	0	2	1	0
0	0	0	0	2	1	0
0	0	1	1	2	2	0
0	0	0	0	0	0	0

Filter W0 (3x3x3)

w0[:,:,0]

-1	0	0
-1	-1	1
0	1	-1

w0[:,:,1]

-1	0	-1
1	-1	0
0	1	-1

w0[:,:,2]

0	-1	-1
1	1	-1
-1	-1	-1

Bias b0 (1x1x1)

b0[:,:,0]

1

Filter W1 (3x3x3)

w1[:,:,0]

0	1	1
1	1	-1
-1	1	-1

w1[:,:,1]

1	-1	0
1	1	1
-1	1	0

w1[:,:,2]

0	1	-1
0	1	1
1	0	0

Bias b1 (1x1x1)

b1[:,:,0]

0

We start with a 5x5 image

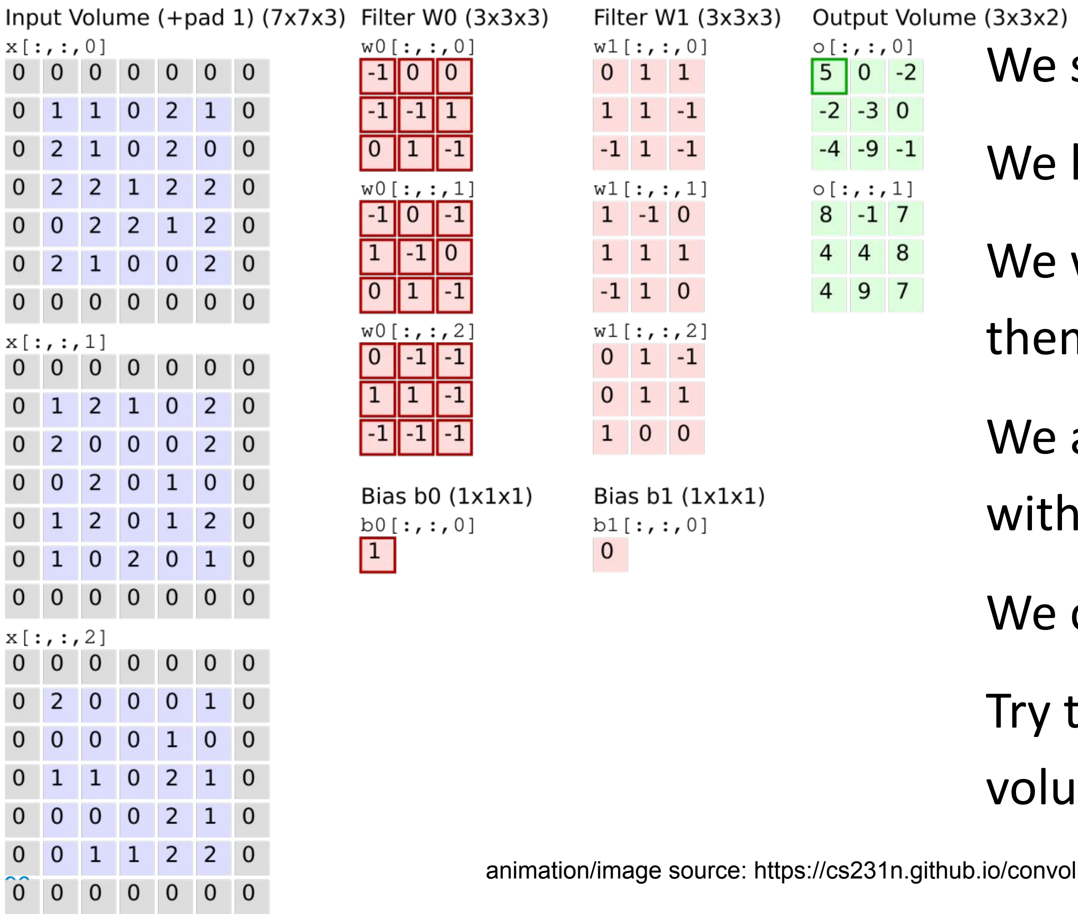
We have 3 channels

We want to use 2 filters, these are themselves 3 dimensional

We add padding to solve issues with convolving near the border

animation/image source: https://cs231n.github.io/convolutional-networks/#fc

2D - IMAGES - CONVOLUTION - OUTPUT VOLUME



Input Volume (+pad 1) (7x7x3)

x[:,:,0]

0	0	0	0	0	0	0
0	1	1	0	2	1	0
0	2	1	0	2	0	0
0	2	2	1	2	2	0
0	0	2	2	1	2	0
0	2	1	0	0	2	0
0	0	0	0	0	0	0

x[:,:,1]

0	0	0	0	0	0	0
0	1	2	1	0	2	0
0	2	0	0	0	2	0
0	0	2	0	1	0	0
0	1	2	0	1	2	0
0	1	0	2	0	1	0
0	0	0	0	0	0	0

x[:,:,2]

0	0	0	0	0	0	0
0	2	0	0	0	1	0
0	0	0	0	1	0	0
0	1	1	0	2	1	0
0	0	0	0	2	1	0
0	0	1	1	2	2	0
0	0	0	0	0	0	0

Filter W0 (3x3x3)

w0[:,:,0]

-1	0	0
-1	-1	1
0	1	-1

w0[:,:,1]

-1	0	-1
1	-1	0
0	1	-1

w0[:,:,2]

0	-1	-1
1	1	-1
-1	-1	-1

Bias b0 (1x1x1)

b0[:,:,0]

1

Filter W1 (3x3x3)

w1[:,:,0]

0	1	1
1	1	-1
-1	1	-1

w1[:,:,1]

1	-1	0
1	1	1
-1	1	0

w1[:,:,2]

0	1	-1
0	1	1
1	0	0

Bias b1 (1x1x1)

b1[:,:,0]

0

Output Volume (3x3x2)

o[:,:,0]

5	0	-2
-2	-3	0
-4	-9	-1

o[:,:,1]

8	-1	7
4	4	8
4	9	7

We start with a 5x5 image

We have 3 channels

We want to use 2 filters, these are themselves 3 dimensional

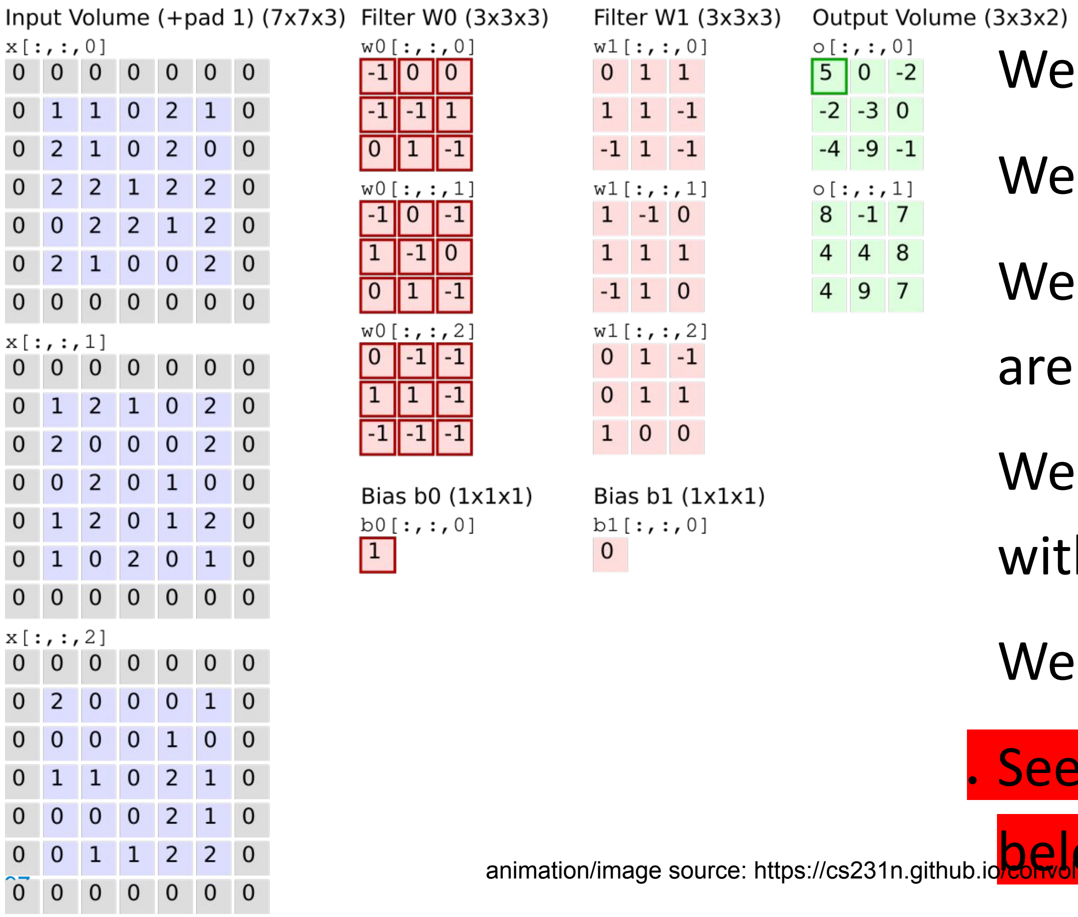
We add padding to solve issues with convolving near the border

We convolve with a stride of 2

Try to understand why the output volume has these dimensions.

animation/image source: https://cs231n.github.io/convolutional-networks/#fc

2D - IMAGES - REPRESENTATION



Input Volume (+pad 1) (7x7x3)

x[:,:,0]

0	0	0	0	0	0	0
0	1	1	0	2	1	0
0	2	1	0	2	0	0
0	2	2	1	2	2	0
0	0	2	2	1	2	0
0	2	1	0	0	2	0
0	0	0	0	0	0	0

x[:,:,1]

0	0	0	0	0	0	0
0	1	2	1	0	2	0
0	2	0	0	0	2	0
0	0	2	0	1	0	0
0	1	2	0	1	2	0
0	1	0	2	0	1	0
0	0	0	0	0	0	0

x[:,:,2]

0	0	0	0	0	0	0
0	2	0	0	0	1	0
0	0	0	0	1	0	0
0	1	1	0	2	1	0
0	0	0	0	2	1	0
0	0	1	1	2	2	0
0	0	0	0	0	0	0

Filter W0 (3x3x3)

w0[:,:,0]

-1	0	0
-1	-1	1
0	1	-1

w0[:,:,1]

-1	0	-1
1	-1	0
0	1	-1

w0[:,:,2]

0	-1	-1
1	1	-1
-1	-1	-1

Bias b0 (1x1x1)

b0[:,:,0]

1

Filter W1 (3x3x3)

w1[:,:,0]

0	1	1
1	1	-1
-1	1	-1

w1[:,:,1]

1	-1	0
1	1	1
-1	1	0

w1[:,:,2]

0	1	-1
0	1	1
1	0	0

Bias b1 (1x1x1)

b1[:,:,0]

0

Output Volume (3x3x2)

o[:,:,0]

5	0	-2
-2	-3	0
-4	-9	-1

o[:,:,1]

8	-1	7
4	4	8
4	9	7

We start with a 5x5 image

We have 3 channels

We want to use 2 filters, these are themselves 3 dimensional

We add padding to solve issues with convolving near the border

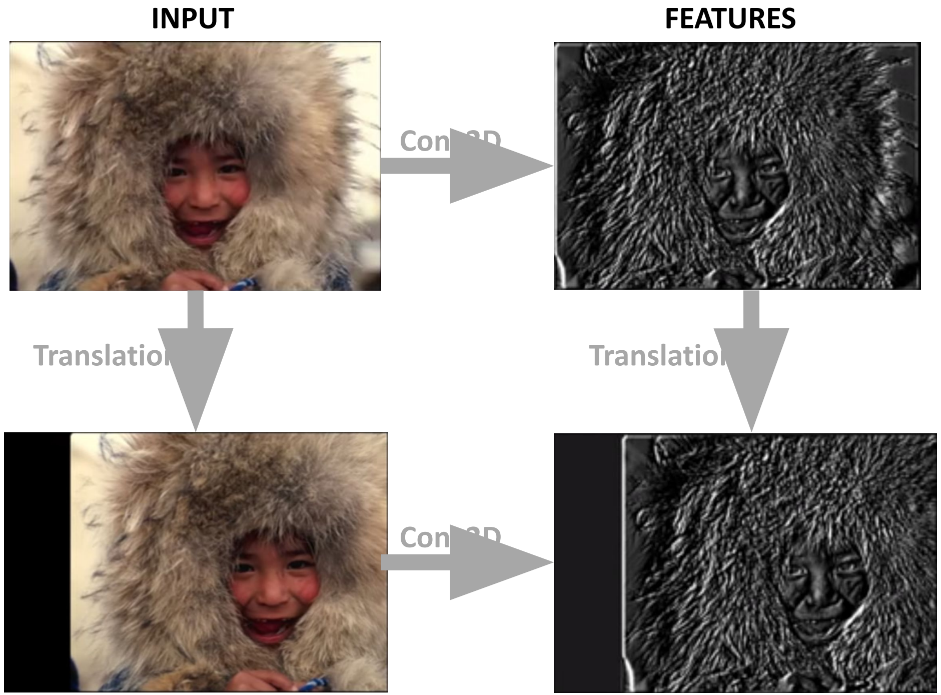
We convolve with a stride of 2

. See the animation on the site below

animation/image source: https://cs231n.github.io/convolutional-networks/#fc

CONVOLUTION - EQUIVARIANCE ANALYSIS

We saw that convolutions are **equivariant** to translations. Naturally it holds for ConvNDs as well:



A translation of the input produces an equivalent translation in the output.

For a translated input $\mathbf{x}(t-t_0)$:

$$(\mathbf{x} * \mathbf{k})(t-t_0) = \sum_{\tau=-m}^{m} \mathbf{x}((t-t_0)-\tau) \cdot \mathbf{k}(\tau)$$

Worral’17

CONVOLUTION - EQUIVARIANCE ANALYSIS

What about other transformations?, e.g., rotation, scaling, …

Are ConvNDs **rotation and scale** equivariant?

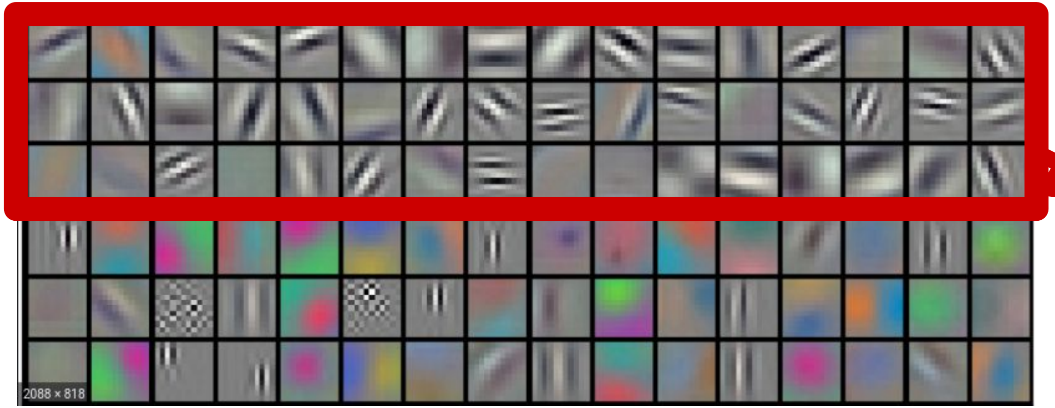
The convolution is **not** a **rotation or scale** equivariant mapping.

CONVOLUTION - EQUIVARIANCE ANALYSIS

What about other transformations?, e.g., rotation, scaling, …

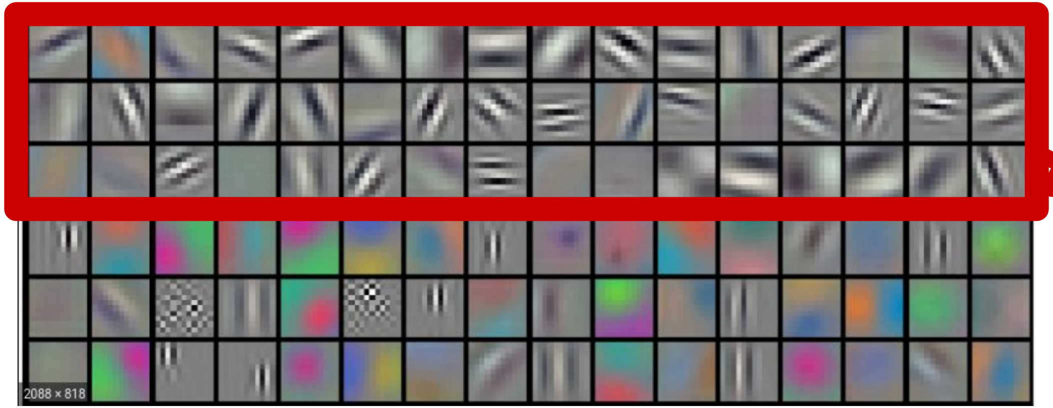
Are ConvNDs **rotation and scale** equivariant?

The convolution is **not** a **rotation or scale** equivariant mapping. But a network can learn rotated / scaled versions of the same filter.



BUT approx. equivariant for scales / rotations learned by the network.

CONVOLUTION - EQUIVARIANCE ANALYSIS

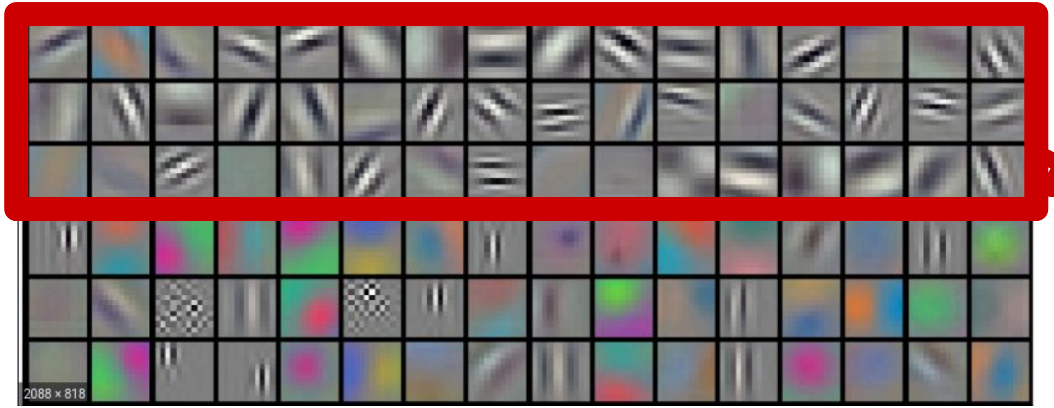


Approx. equivariant for scales / rotations learned by the network.

Problem: Each of these filters are independent weights:

- The network wastes a lot of parameters learning transformed versions of the same -> **Parameter Inefficient!**
- To learn these filters the network must see these transformations in the training set -> **Data Inefficient + No equivariance guarantees!**

CONVOLUTION - EQUIVARIANCE ANALYSIS



Approx. equivariant for scales / rotations learned by the network.

Solution: Group Convolutions:

- Not **just** share parameters for translation but also other transformations!
- Extreme parameter sharing + equivariance guarantees.
- Active field of research. Amsterdam is a big player in this field. Several papers written at the VU, the UvA and Qualcomm AI Research.*

* Let us know if you are interested in writing your thesis in this topic ;)

PART FOUR: Example of a real world CNN

AlexNet (Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." Advances in neural information processing systems. 2012)

Showed the feasibility of deep learning

- Mainly thanks to the use of GPUs for computing convolutions
- Achieved a top-5 error of 15.3% on a dataset with 1000 categories

By some considered as the real start of adoption of neural networks by the industry

Is actually just a variant on an older idea

- LeCun, Y.; Boser, B.; Denker, J. S.; Henderson, D.; Howard, R. E.; Hubbard, W.; Jackel, L. D. (**1989**). "Backpropagation Applied to Handwritten Zip Code Recognition”

AlexNet

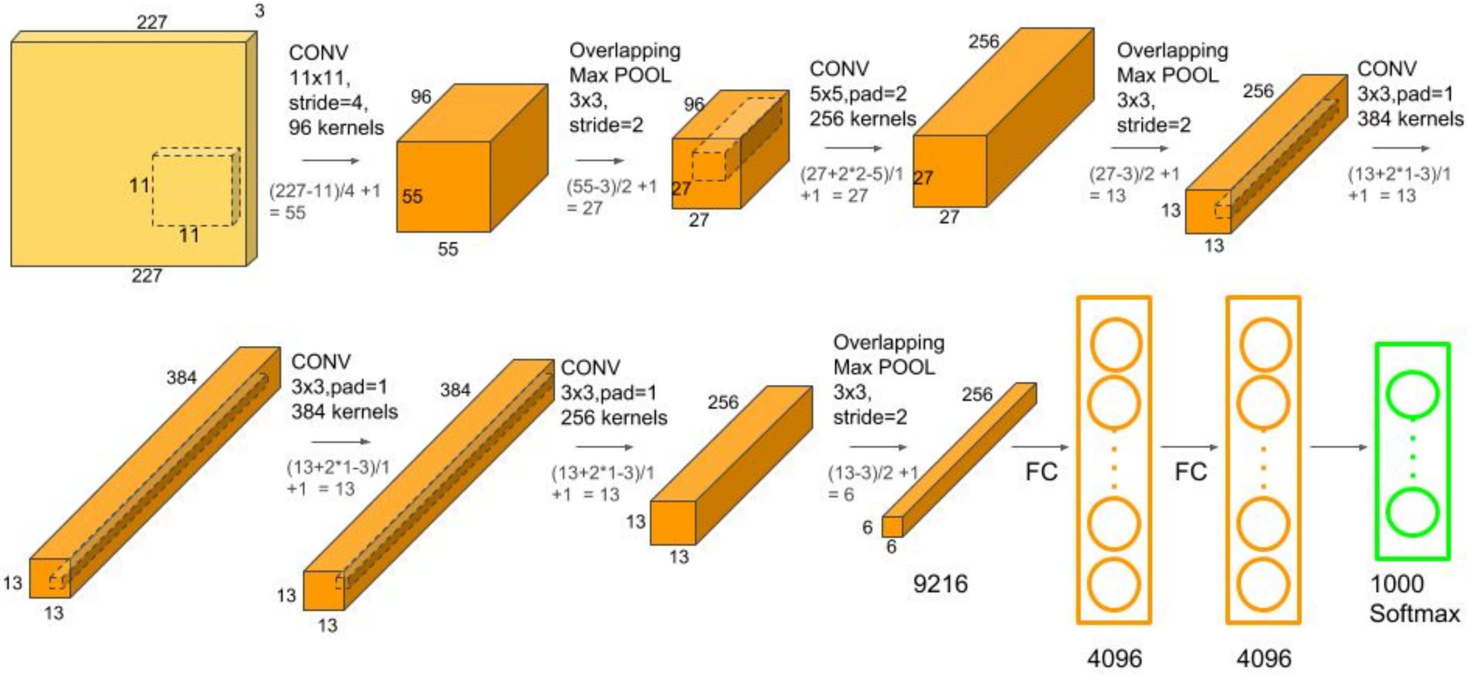


image source: https://www.learnopencv.com/number-of-parameters-and-tensor-sizes-in-convolutional-neural-network/

Summary

If we want to attach a neuron to this, so we're going to make first, say, a hidden layer of our neural network, that neuron also needs 2 million weights. That neuron also needs memory space to store the gradients. And then on top of that, you probably want more than one neuron, you probably want several hidden layers and you want to have more neurons per layer. This requires an extremely large amount of weights.

part 1: Introduction - why are convolutional architectures needed?

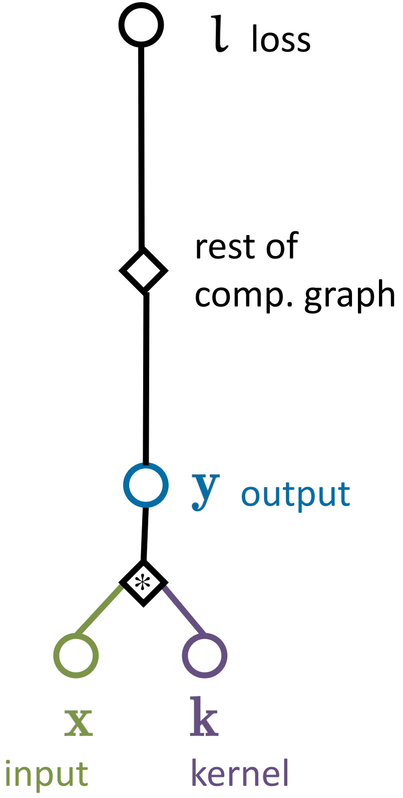
part 2: One-dimensional convolutional neural networks (conv1D)

part 3: Two-dimensions and beyond (conv2D, conv3D, ...)

part 4: Example architecture

PART FIVE*: Backpropagating Convolutions

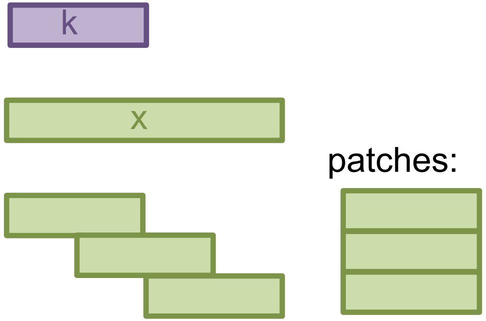
1D CONVOLUTION: FORWARD



definition:

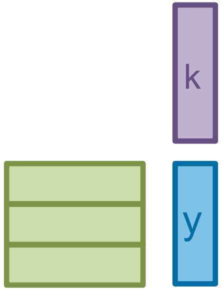
$$\mathbf{y} = \mathbf{x} * \mathbf{k}$$

$$y_t = \sum_{\tau=-m}^{m} x_{t-\tau} k_\tau$$



vectorization:

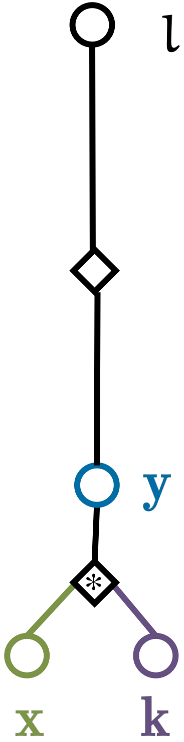
```
patches = get_patches(x, k.size())

y = patches * k[None, :]
```

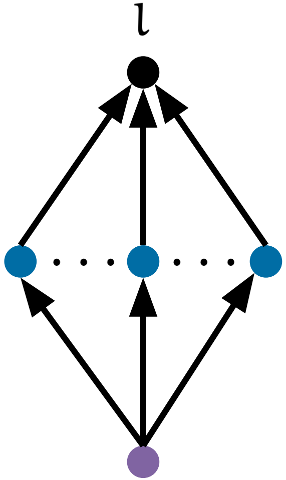


1D CONVOLUTION: BACKWARD

$$\mathbf{y} = \mathbf{x} * \mathbf{k}$$

$$y_t = \sum_{\tau=-m}^{m} x_{t-\tau} k_\tau$$





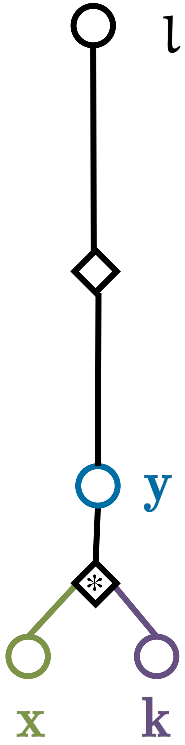
$$k_i^\nabla = \frac{\partial l}{\partial k_i} = \sum_t \frac{\partial l}{\partial y_t} \frac{\partial y_t}{\partial k_i}$$

$$= \sum_t y_t^\nabla \frac{\partial y_t}{\partial k_i} = \sum_t y_t^\nabla \frac{\partial \sum_\tau x_{t-\tau} k_\tau}{\partial k_i}$$

$$= \sum_{t,\tau} y_t^\nabla \frac{\partial x_{t-\tau} k_\tau}{\partial k_i} = \sum_t y_t^\nabla \frac{\partial x_{t-i} k_i}{\partial k_i}$$

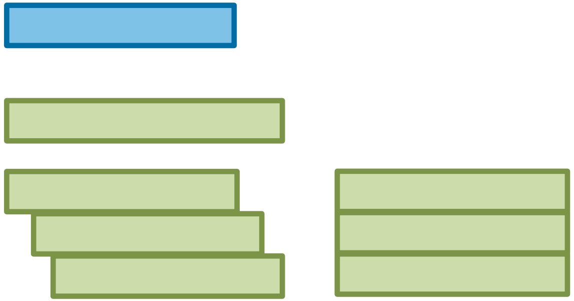
$$= \sum_t y_t^\nabla x_{t-i}$$

1D CONVOLUTION: BACKWARD, VECTORIZATION



$$k_i^\nabla = \sum_t y_t^\nabla x_{t-i}$$

$$\mathbf{k}^\nabla = \begin{pmatrix} \vdots \\ \sum_t y_t^\nabla x_{t-i} \\ \vdots \end{pmatrix}$$



vectorization:

```
patches = get_patches(x, k.size())

k' = patches * y'[None, :]
```