

Lecture 3: Convolutional Neural Networks

Michael Cochez

Deep Learning

dlvu.github.io



Welcome to lecture 3 of the deep learning course. Today we will be talking about convolutional networks which are networks responsible for several recent breakthroughs in deep learning.

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

2



In the first part we will talk about where convolutional architectures are needed.

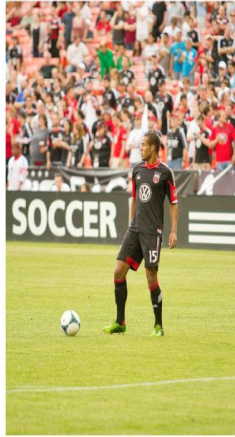
In the second part I will be talking about one dimensional convolutional networks and explain the basic concepts.

In part three we'll scale up to more dimensions (3D, 4D, etc.) and discuss some of the theory.

In part four we will look at some example architectures

PART ONE: INTRODUCTION





a soccer player is kicking a soccer ball



a street sign on a pole in front of a building



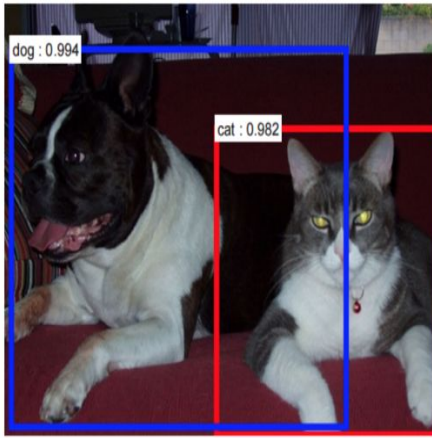
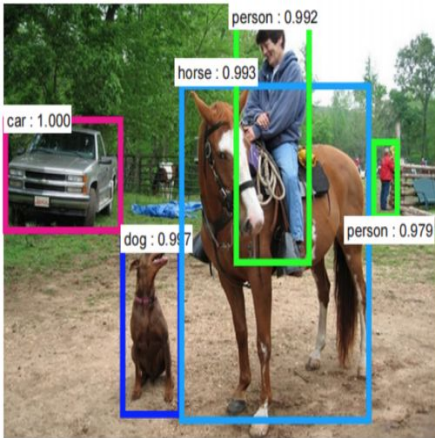
a couple of giraffe standing next to each other

4

source: <https://cs.stanford.edu/people/karpathy/heurtak2/demo.html>



Of the recent breakthroughs in deep learning, the ones which are often visible are those related to images. For example, these are three examples where a deep learning system that includes some convolutional neural networks is able to create a description of an image. So basically the system receives an image and then produces a caption for that image. For example the first one you see a football player and then the system is able to generate a sentence like “a soccer player is kicking a soccer ball”. These kinds of systems receive a lot of training data (in the form of images with their captions) and generate output on images they have never seen before.



A second example is semantic image segmentation. The system is able to determine what certain parts of the picture depict.

INPUT DATA

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

6



When we look back at our basic deep learning knowledge we have a bit of an issue: we have that image coming in but what we need for our deep learning model is a set of features. For tabular data (meaning the typical table) this would be simple but what do we do with more complex data like an image?

INPUT DATA - IMAGES

. Example

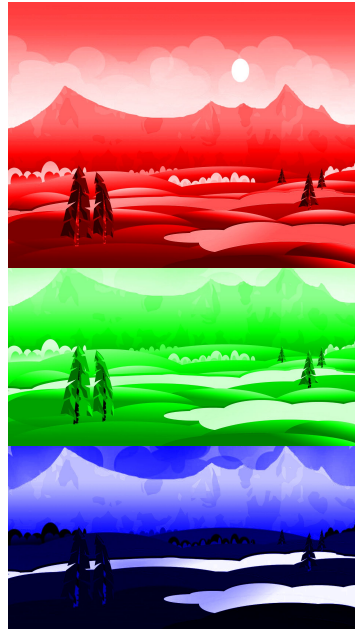


<https://pixabay.com/nl/illustrations/vector-art/beelding/landschap-vector-3833815/>



The first thing we should realize is that an image has a width, has a depth but also has colors.

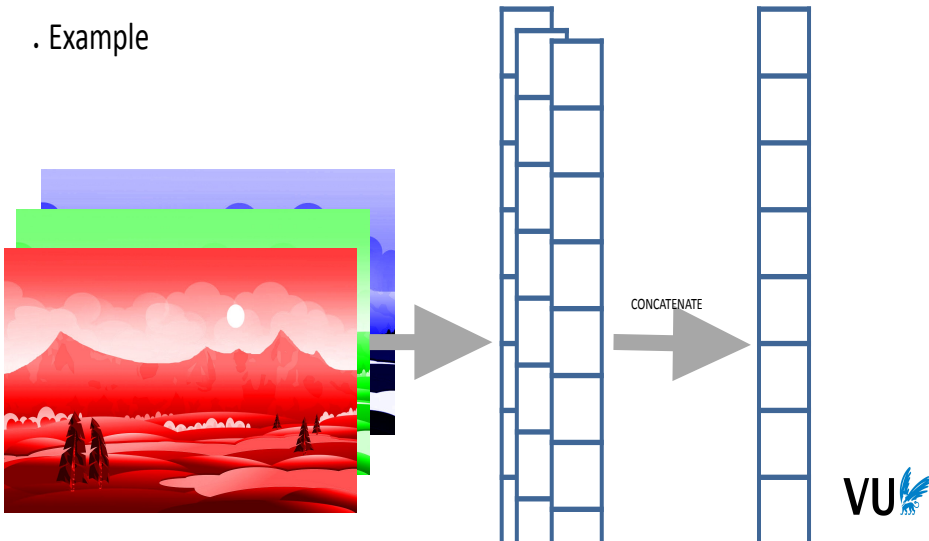
. Example



We can split an image into so-called color-channels. You've probably already heard about RGB (red green blue). We can separate the image such that we have these three channels separate from each other.

INPUT DATA - SOUND, IMAGES, VIDEO - ENCODING

. Example



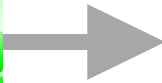
We can then look at each of those colour channels and depict their intensity in the form of a vector. How intense is the intensity of the red colour channel at this pixel? We could then in principle concatenate all these vectors together to form one long vector representing that image.

- . The input dimensions are **very** big
- . One channel of an image of 1920x1080 \approx 2M features
- . 1 second of sound at 44kHz = 44k features
- . A video: frame rate * image features + sound
 - 10 seconds $\Rightarrow 10*(60\text{fps}*3*2\text{M}+44\text{k})$

A problem here is that the input dimensions are very big. That means if we look at this one channel of an image, for example the image in the previous slide, we have two million features. Now we have three channels so that would already be six million. Obviously now one second of sound for example sampled at 44 kilohertz would similarly lead to 44 000 features. For videos, you have sound and a moving image leading to an even greater number of features.

INPUT DATA - SOUND, IMAGES, VIDEO - ENCODING

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



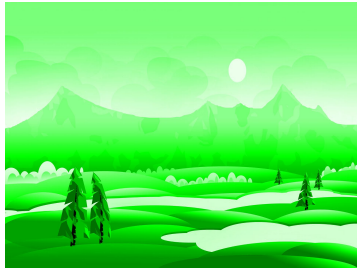
1920x1080 \approx 2M
dimensional vector



Now, the problem is that this very large number of features is also just too big for our normal multi layer perceptron. The example before would lead to a 2 million dimensional vector.

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

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



1920x1080 \approx 2M
dimensional vector

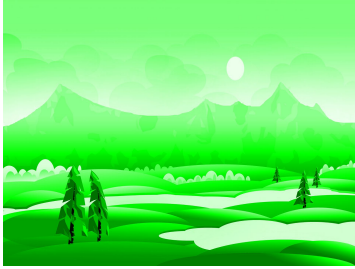
So, you need 2M
weights for just 1
neuron!!!



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.

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

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



1920x1080 \approx 2M
dimensional vector

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

And you want more
than 1



This approach is simply not feasible in practice.

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

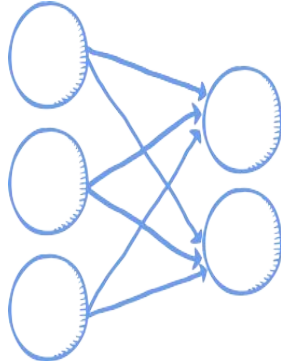
- . The input dimensions are **very** big
- . Too big for an MLP
 - Too many weights
 - Would not converge
 - would not fit in GPU memory
 - . Especially when you also need to keep gradient information



Besides just having too many weights this system would also not converge because you have to train all those weights. Constraints in GPU memory make this infeasible as well.

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



15



One notable observation about networks of this type is that, aside from the size argument, the features within this dataset exhibit a level of interdependence; they possess a certain locality. This implies that when two pixels are adjacent to each other, there is inherent meaning in their proximity. In contrast, when you consider a Multi-Layer Perceptron (MLP), it does not take into account this spatial ordering. Whether the information is presented with one note above another or in a mixed configuration, the MLP does not retain the sequence or order of the notes.

GOAL FOR TODAY

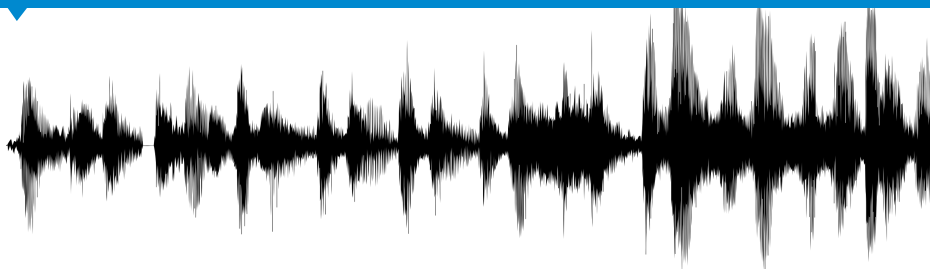
- . We want to be able to do deep learning on this kind of data
- . Steps
 - 1D
 - . Build intuition
 - 2D
 - . Do the same for images

All right, let's recap what we've observed. We've encountered an image and successfully devised a digital representation for it in memory. However, the MLP we initially employed doesn't prove to be a viable option. It lacks scalability and effectiveness. To address this issue and facilitate deep learning for this data type, we'll take a systematic approach. We'll begin by exploring this in 1D, gaining an intuitive understanding. Subsequently, we'll formalize the process and transition to 2D. Our ultimate goal is to apply the same principles to higher dimensions, such as images.

PART TWO-a: conv1D



Welcome to part two of the convolutional neural network lecture. In this part we will be looking at one dimensional convolutions and try to build an intuition for them.



- . 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

18

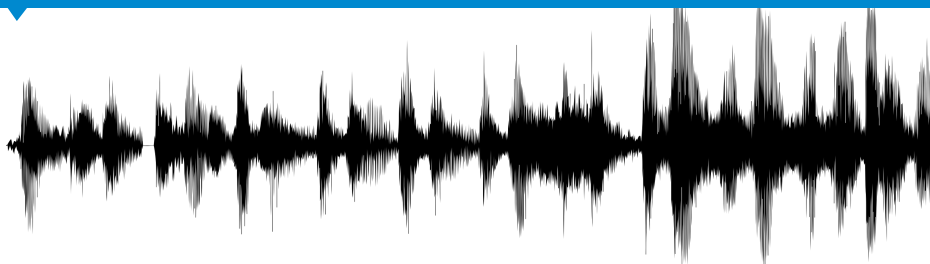


A sound wave is the superposition of sine waves.

With deep learning, we can try to answer hard questions like the ones mentioned here.

The central example we'll consistently refer to in the following slides revolves around a one-dimensional sound wave. Our objective is to harness the power of deep learning to address complex inquiries related to sound. For instance, when presented with a particular sound wave, like the one displayed, we could seek to determine the musical style it embodies. Additionally, we could discern whether the user enjoys this music, rendering the need for a classification model. Essentially, this is a classification problem: given a piece of music, we classify it into clusters.

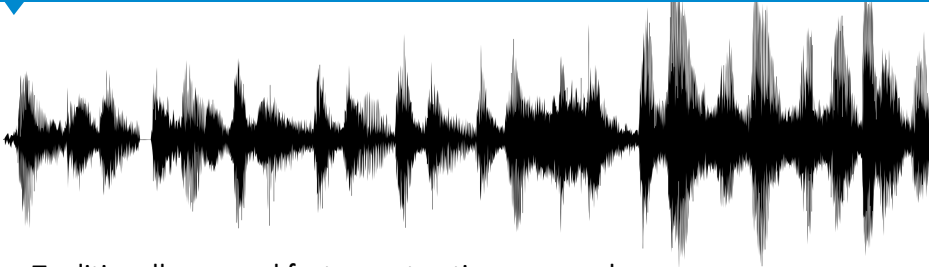
Conversely, there are regression tasks, like discerning the beat or rhythm underlying the music. Furthermore, we can assess the pleasantness of the music, typically quantified on a scale from one to 100.



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

Later, with generative modelling, we will go even further. In addition to the conventional classification and regression tasks, we encounter more challenging endeavors, particularly those falling within the realm of generative tasks. These tasks demand a creative response. For instance, when presented with a sound wave, we might ponder, "What would a pristine version of this audio signal sound like?" Alternatively, we might explore, "What audio composition best complements these lyrics?" Or perhaps, we may contemplate, "How should this sound progress, given a brief excerpt from the song?"

In these instances, the objective isn't limited to delivering a simple categorical label, integer, or floating-point number as an answer. Instead, it involves the creation of a more intricate, generative response – akin to crafting a sandwich, in a metaphorical sense.



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

Traditionally, addressing these challenges entailed the labor-intensive process of manual feature extraction. This approach involved the creation of digital or standard signal processing filters, which were subsequently applied to the audio data. Such filters were employed for tasks like beat detection and the identification of specific, manually engineered patterns.



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

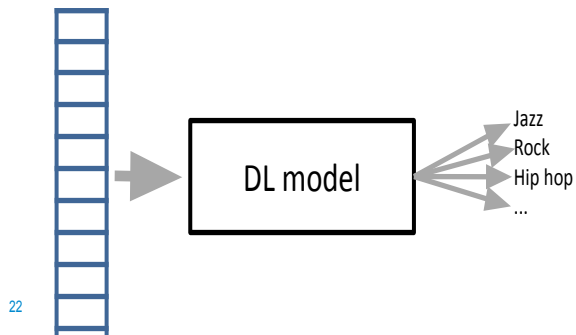
21



However, these approaches are not without their shortcomings. Several challenges emerge in their application. Noise is a significant issue; when the audio signal is contaminated by noise, these methods tend to falter. Additionally, variations in vocal attributes, such as a song performed by a male versus a female, can result in substantially different sounds. Furthermore, scenarios where audio fragments are missing pose a significant challenge, as these filters struggle to perform optimally under such conditions.

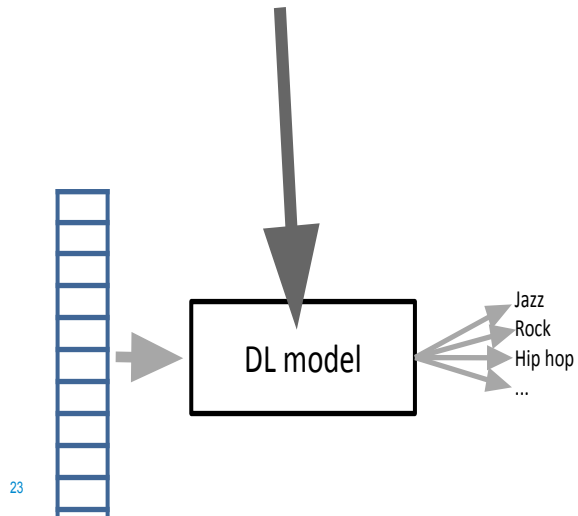
In essence, the fundamental problem lies in the fact that the specific features that need to be extracted from the audio are not predetermined. This uncertainty makes it challenging to identify the appropriate set of features for extraction.

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



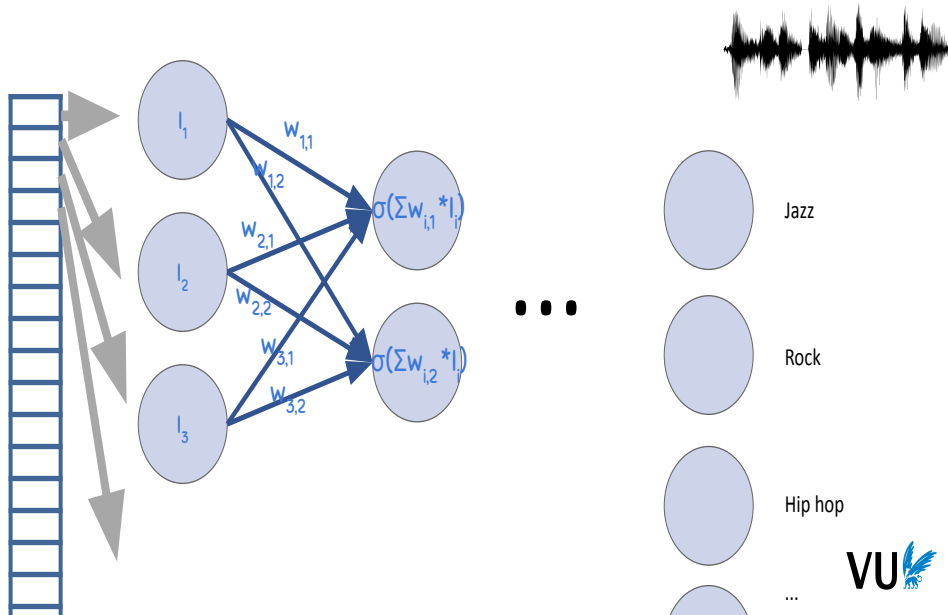
In the context of a deep learning system, we aim to offload the burden of feature extraction onto the model itself. The model is designed to autonomously handle this aspect, and all you need to do is feed it the raw data.

. Let us try to use an MLP



To begin, we'll explore using a Multi-Layer Perceptron (MLP) for this purpose. Our initial focus is on a classification task, specifically the identification of music styles.

1D - SOUND - HOW? - MLP



Weights

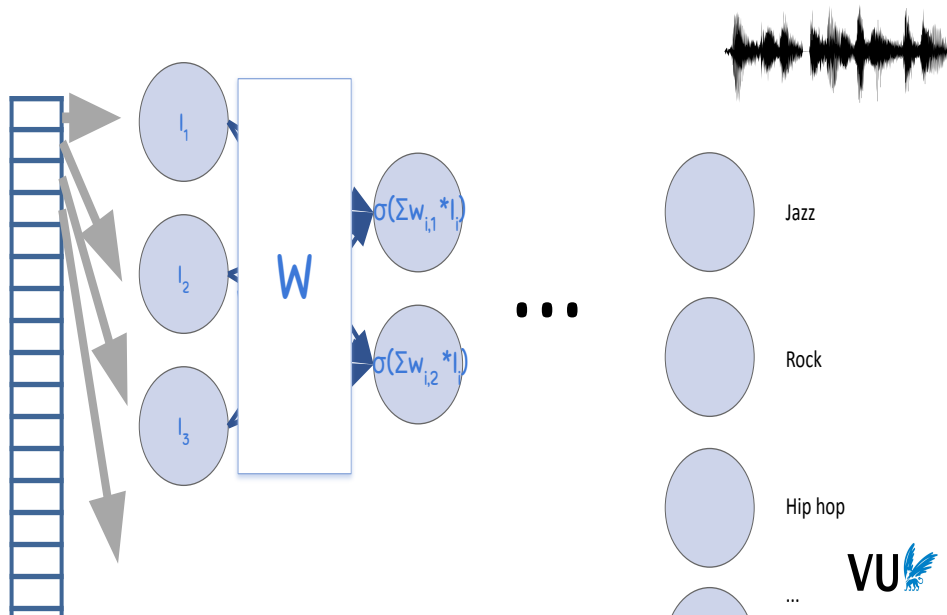
Hidden layers,

Non-linearities

Bias (not in the image)

Backprop

1D - SOUND - HOW? - MLP



Weights

Hidden layers,

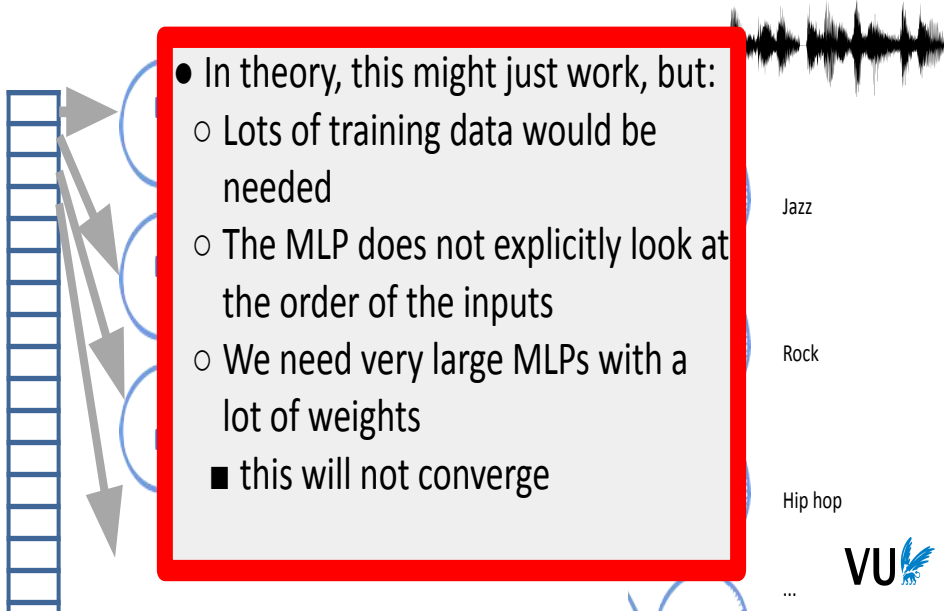
Non-linearities

Bias (not in the image)

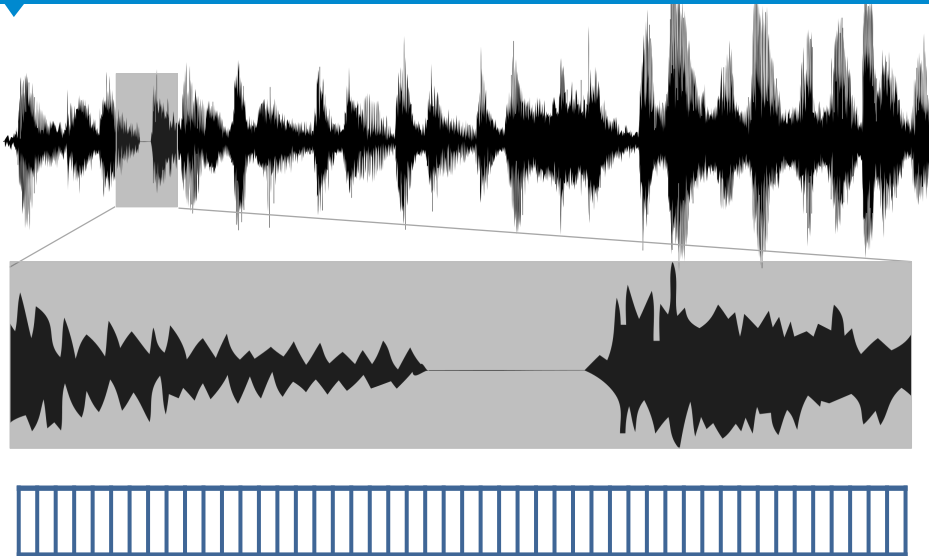
Backprop

So, what would we do in principle or what we do normally is we would have these these neurons, we would feed the information based from the input into this input layer, then we would have the weights on the connections between these these neurons, then we would have a new layers which we basically collect the incoming information summing together and apply non linearity and that way things propagate into the network. I did not explicitly mentioned the bias notes in this in this image, but there are there right so in all the images which are showing about MLPs there is always also a bias note in these in these formulas and indeed images. after things have forward propagated through the network, you can also use the backpropagation basically to improve the classification accuracy of that network. Now, of course, these weights which are here, we can just represent them with one big weight matrix.

1D - SOUND - HOW? - MLP



As previously noted, in theory, this approach may appear feasible, but it is beset by several inherent challenges. Firstly, it necessitates a substantial volume of training data. Furthermore, the Multi-Layer Perceptron (MLP), as mentioned earlier, doesn't account for the order or sequence of input data. Moreover, it was highlighted previously that to make this approach effective, you would require an exceedingly large MLP. This means having a virtually one-to-one connection between neurons and input features, resulting in an impractical abundance of connections. Ultimately, this approach is unlikely to yield desirable results.



A sound wave is the superposition of sine waves.

While in reality, this a sound wave is continuous, it gets sampled at a high rate (44Khz) to make it possible to store on a computer. For or a computer a sound wave is just a sequence of numbers.

For the sake of this lecture we will directly use this wave as features. So, what we have is a one-dimensional vector of numbers.

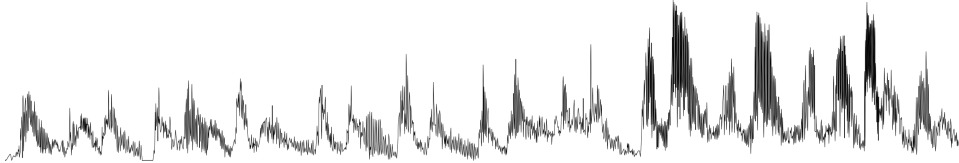
Now, let's let's discuss a bit about how our input data. So what the sound wave basically is, is a superposition of sine waves, that's we have. That's basically words such as signals consists of, and in reality that send your sample if it's continuous, it's a continuous continuous wave. But what we do for getting this into computer is sampling it. So we're sampling is, for example, at a rate of 44 kilohertz is very common. And this makes it possible to store it digitally on a computer. So for a computer, a sound wave is just a sequence of numbers, right, so it is not a wave anymore, it's just a sequence of numbers. And for the sake of this lecture, we will just directly use this wave as the feature. So what we have is basically a one dimensional vector of numbers. Now, let's let's look at how we can do that. So basically, we have that sound wave. Now if we look, if we zoom in, we basically see that the sound wave consists of certain heights, let's say or certain, let's say y axis values,

and we can basically basically sample them say like how high is this peak and measure it and then put it into into the vector.

1D - SOUND - AMPLITUDE



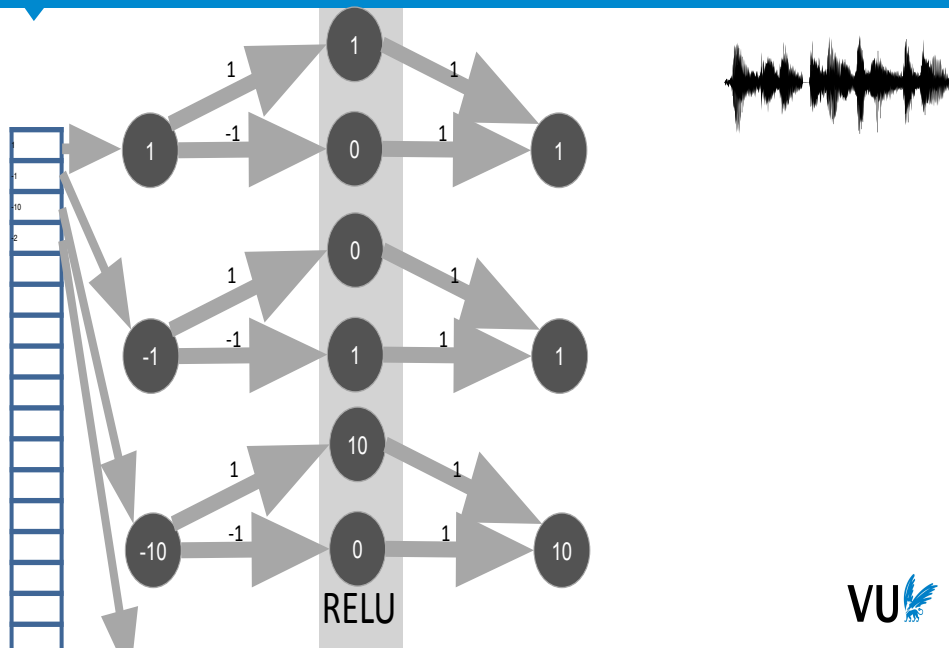
. In the context of the following, we will only use the amplitude of the soundwave:



In the context of the following steps we will only use the amplitude of the soundwave. Note, that this is reasonable, since we can reasonably easily design an MLP that extracts this.

It is a good exercise to pause the video and think how such a network could be created. A possible answer can be found in the slides.

1D - SOUND - FILTERS - AMPLITUDE MLP



VU

In the context of the following steps we will only use the amplitude of the soundwave. Note, that this is reasonable, since we can reasonably easily design an MLP that extracts this.

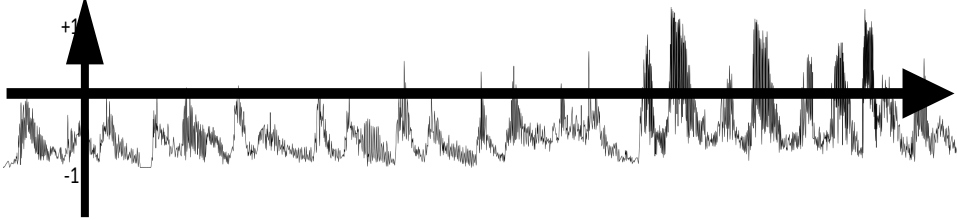
In the example, biases are left out, as we do not even need them.

Note that when we use a CNN as we will introduce later, this network can be represented in a very compact way.

1D - SOUND - AMPLITUDE



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



30



Next, let's explore our following step. Instead of representing this wave in its original form, we'll take a different approach. Essentially, we will gauge the amplitude of the sound wave, measuring its loudness. This choice is quite rational because, in principle, you can design a MLP to perform this transformation. It can take the original sound wave and calculate its amplitude, which is essentially equivalent to computing the absolute value of the waveform.

I encourage you to pause the video for a moment and consider how you might convert the original sound wave into its amplitude. This pause will enhance your comprehension of the subsequent content in this lecture.

In the presentation slides, you'll discover a possible solution for the challenge of deriving the amplitude from the original sound. Moving forward, our next step involves normalizing this amplitude. In essence, we'll adjust it so that its values fall within the range of plus one to minus one, with an average value centered around zero.

1D - SOUND - FILTERS

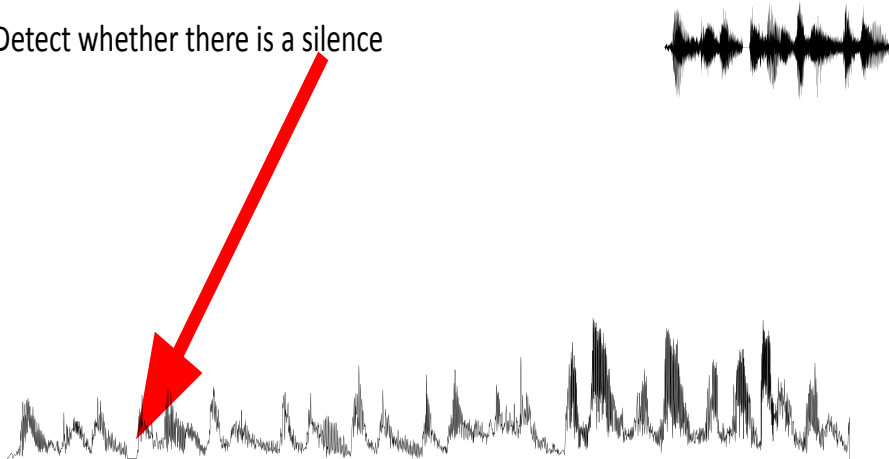
- . The features in the soundwave are not independent
- . Nearby features are more important as far away ones
- . This idea can be used in filters



All right, with that step completed, we now have our vector input. However, when dealing with sound input, as we've previously discussed, and this holds true not just in general but especially for sound data, the features are intricately interconnected. The order in which these features appear is of paramount significance. This ordering plays an important role, signifying that nearby features carry more weight than those located further away. In essence, when you have a particular sound, it's crucial to comprehend it both at a local level and in the context of what was heard shortly before or after in the audio. This concept, emphasizing the significance of nearby information, is often harnessed in what's known as "filtering."

1D - SOUND - FILTERS - EXAMPLE

. Detect whether there is a silence



32

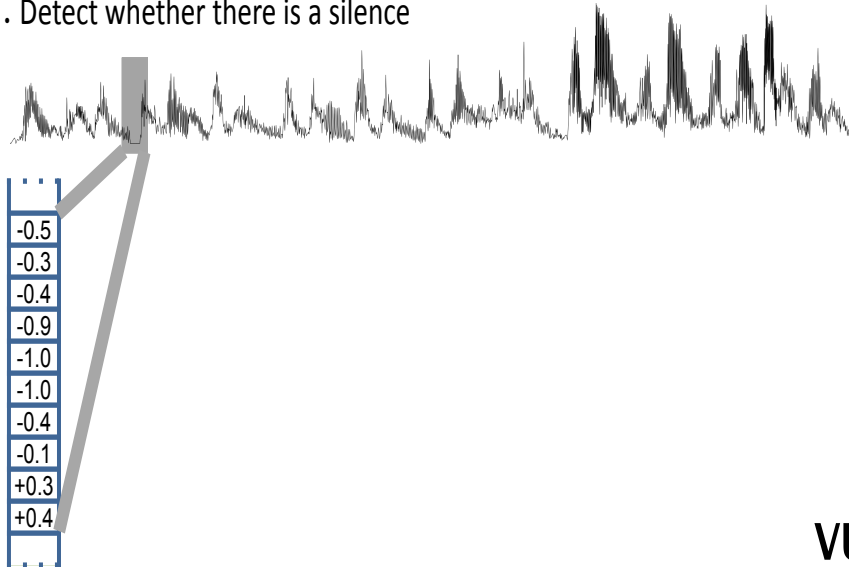


A silence is at points where the amplitude is low for some time.

And so, what we will be doing with filters is we will first look into an example in which we try to find silence in our sound right and. What does it mean in our case? It is basically a time a period in which the amplitude is fairly low, close to zero or exactly zero during that time.

1D - SOUND - FILTERS - EXAMPLE

. Detect whether there is a silence

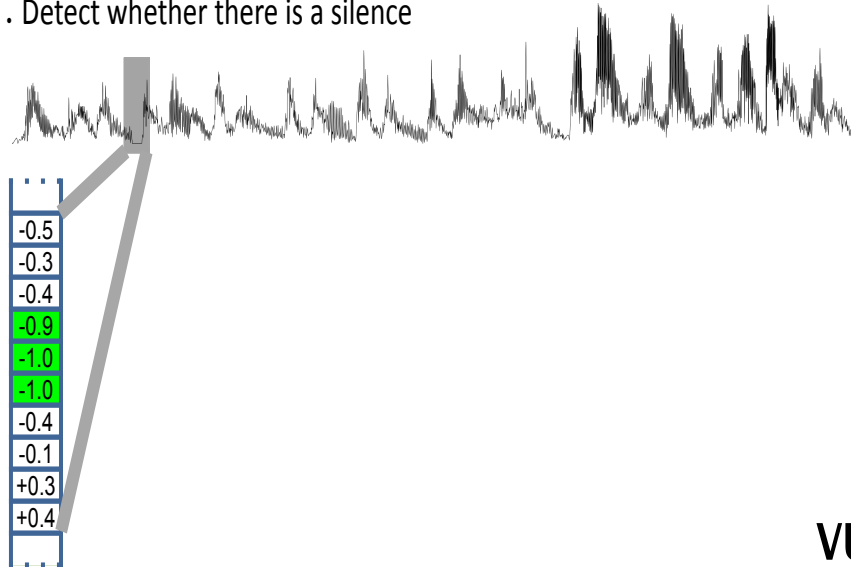


VU

In our case, this means a several consecutive features with a low value.

1D - SOUND - FILTERS - EXAMPLE

. Detect whether there is a silence



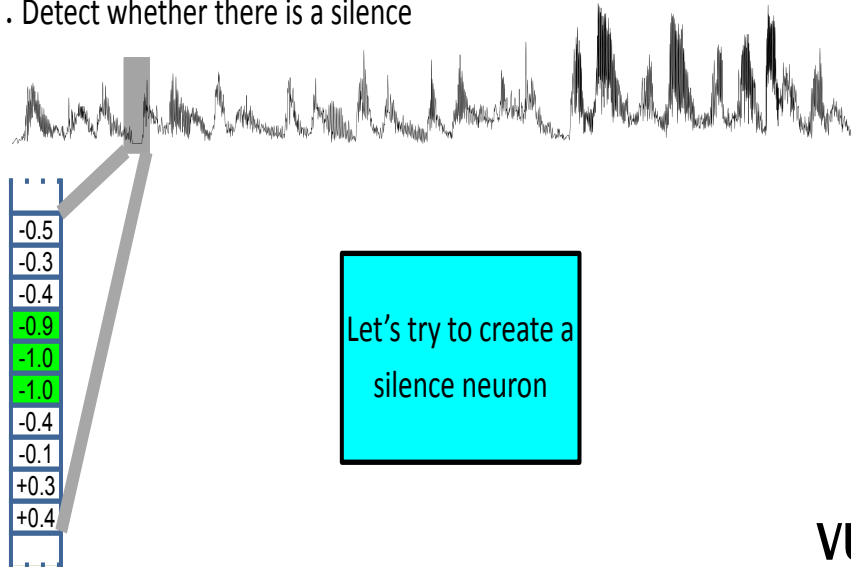
VU

We want to detect where in our sound wave a silence occurs.

If we examine a segment over a specific time interval, as indicated here, and closely inspect the numerical values derived from the original signal's samples, you'll notice that there's a period during which the values remain consistently low for a certain duration.

1D - SOUND - FILTERS - EXAMPLE

. Detect whether there is a silence



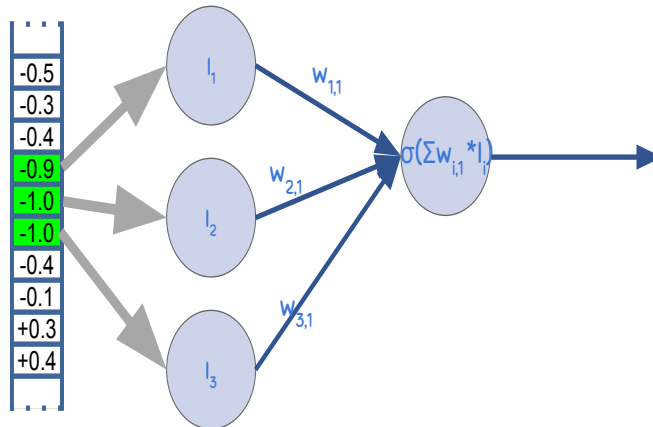
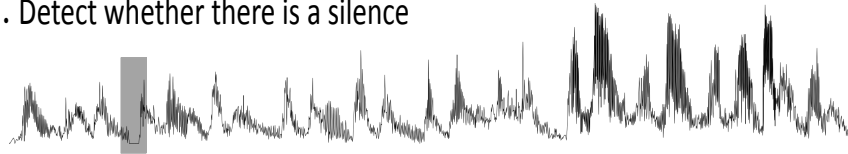
VU

We want to detect where in our sound wave a silence occurs.

In the context of an MLP, we want to create a “silence neuron”. A neuron that fires in case there is a silence.

1D - SOUND - FILTERS - EXAMPLE

. Detect whether there is a silence



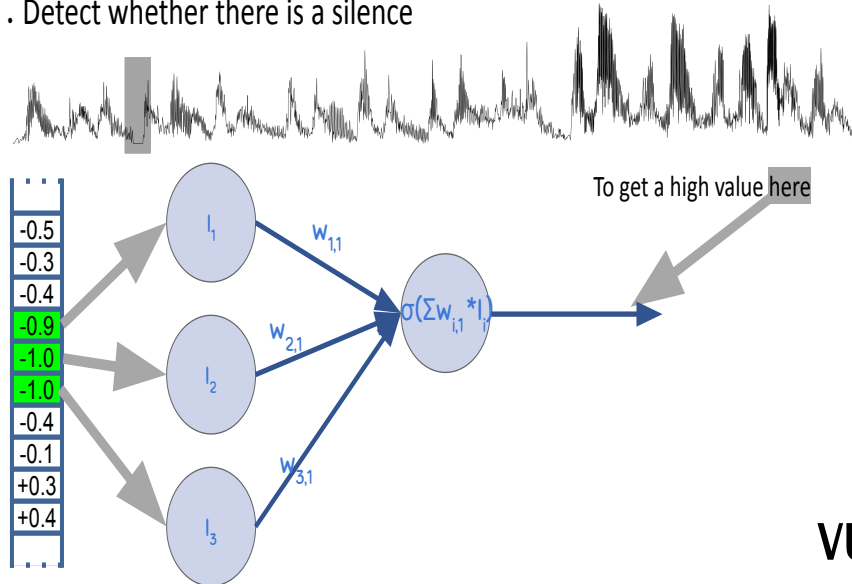
VU

This would be a neuron. Let's work backwards. We want it to have a high value.

So, all parts of the summation need to have a high value (the non-linearity is a monotonic function)

1D - SOUND - FILTERS - EXAMPLE

. Detect whether there is a silence



VU

In our features, this means a several consecutive features with a low value.

We want to detect where in our sound wave a silence occurs.

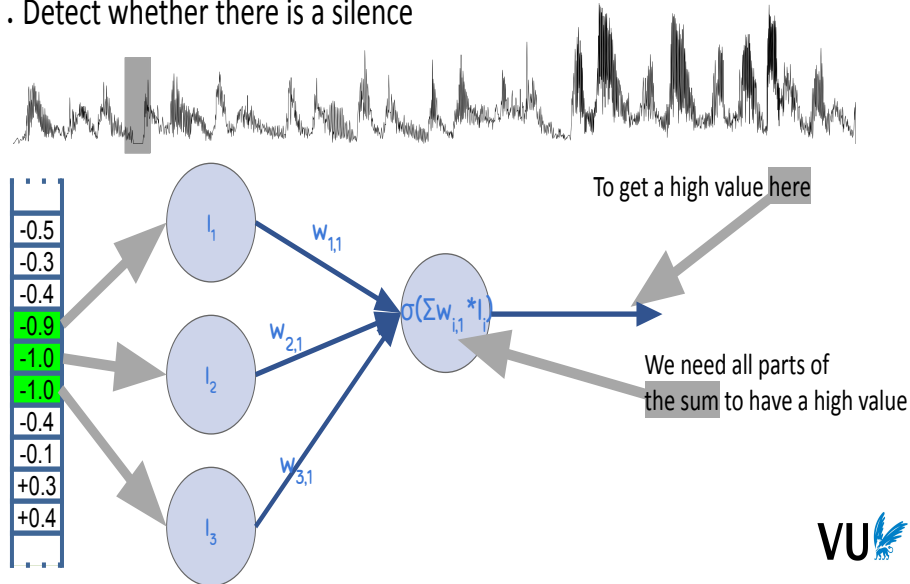
In the context of an MLP, we want to create a “silence neuron”. A neuron that fires in case there is a silence.

This would be a neuron. Let’s work backwards. We want it to have a high value.

So, all parts of the summation need to have a high value (the non-linearity is a monotonic function)

1D - SOUND - FILTERS - EXAMPLE

. Detect whether there is a silence



VU

In our features, this means a several consecutive features with a low value.

We want to detect where in our sound wave a silence occurs.

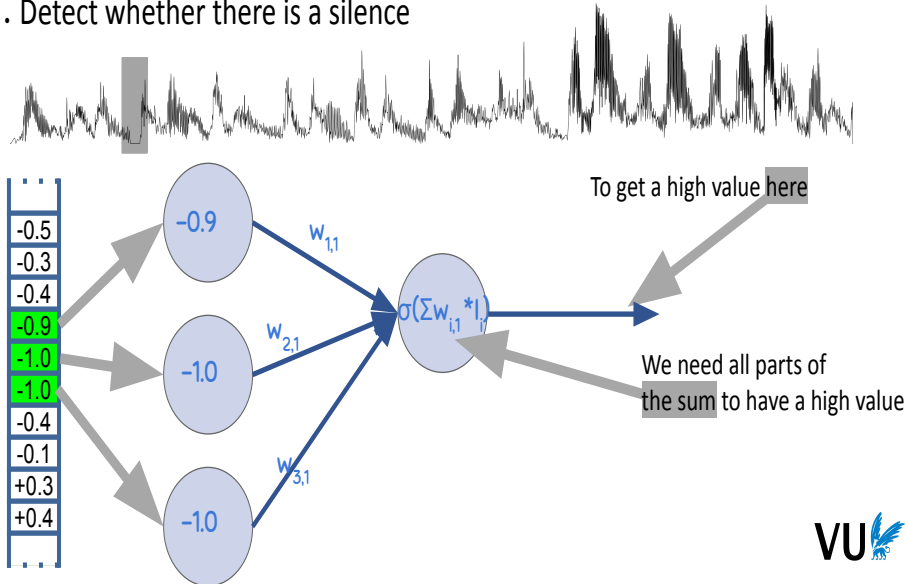
In the context of an MLP, we want to create a “silence neuron”. A neuron that fires in case there is a silence.

This would be a neuron. Let’s work backwards. We want it to have a high value.

So, all parts of the summation need to have a high value (the non-linearity is a monotonic function)

1D - SOUND - FILTERS - EXAMPLE

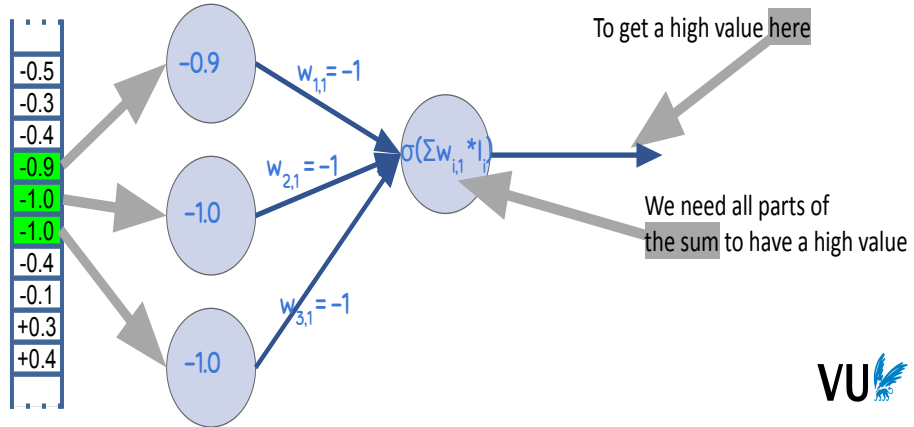
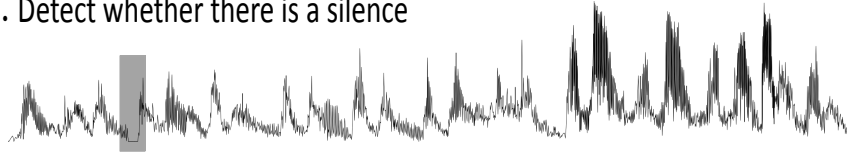
. Detect whether there is a silence



Since the input value are negative numbers, the only option is to also have the weights with negative numbers

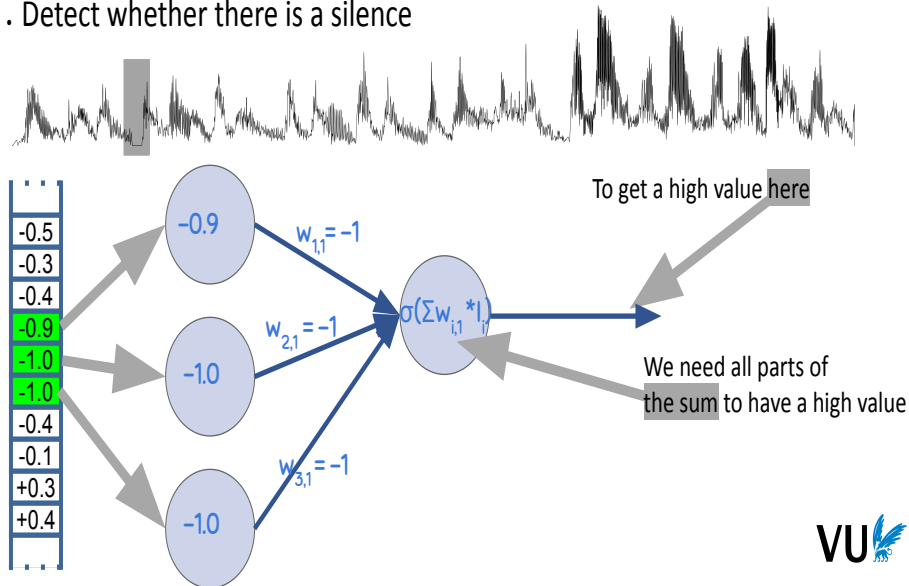
1D - SOUND - FILTERS - EXAMPLE

. Detect whether there is a silence



1D - SOUND - FILTERS - EXAMPLE

. Detect whether there is a silence



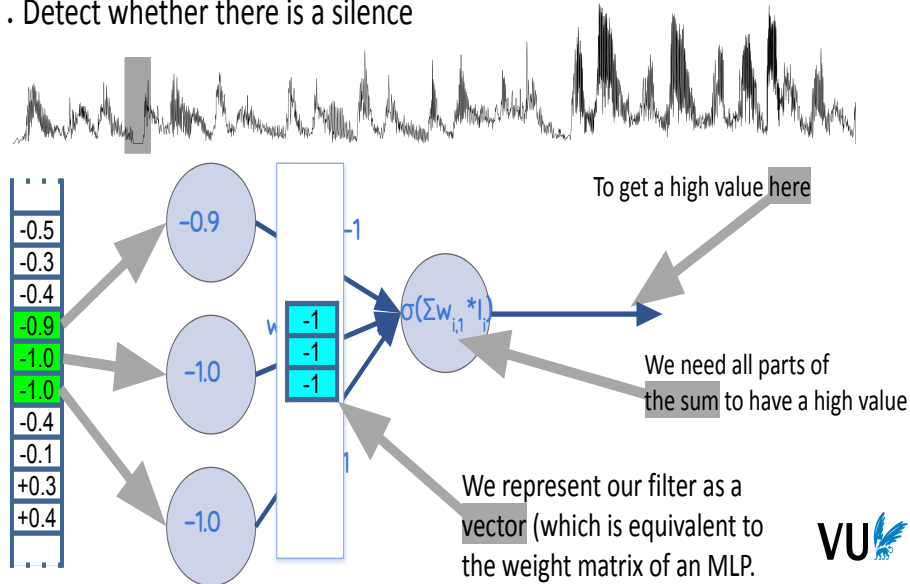
We observe that the weights have a pattern similar to the data we are interested in.

Since we are essentially computing a dot product, the more the weights agree with the pattern we are interested in, the stronger the neuron will be. We will call such patterns - filters - a term which is directly borrowed from the field of signal processing.

animation: 1

1D - SOUND - FILTERS - EXAMPLE

. Detect whether there is a silence



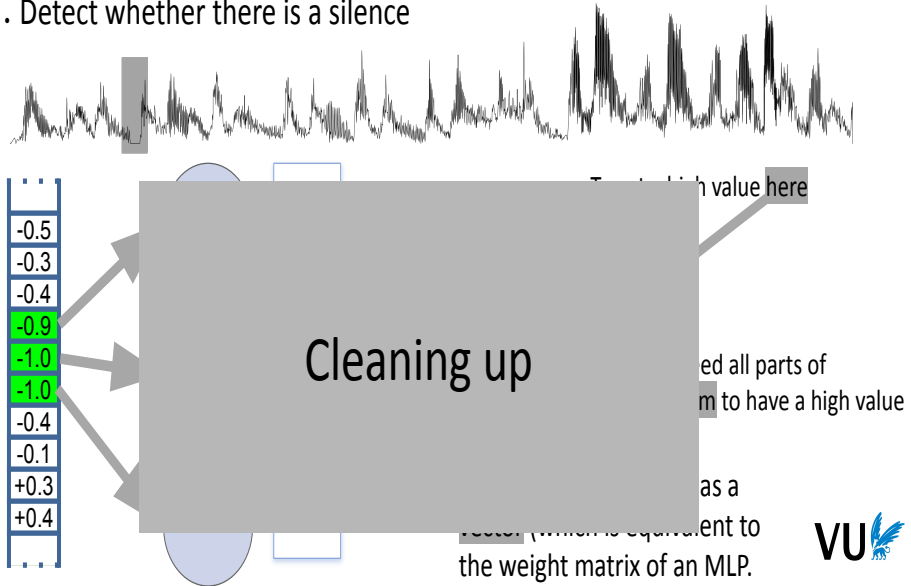
We observe that the weights have a pattern similar to the data we are interested in.

Since we are essentially computing a dot product, the more the weights agree with the pattern we are interested in, the stronger the neuron will be. We will call such patterns - filters - a term which is directly borrowed from the field of signal processing.

animation: 2

1D - SOUND - FILTERS - EXAMPLE

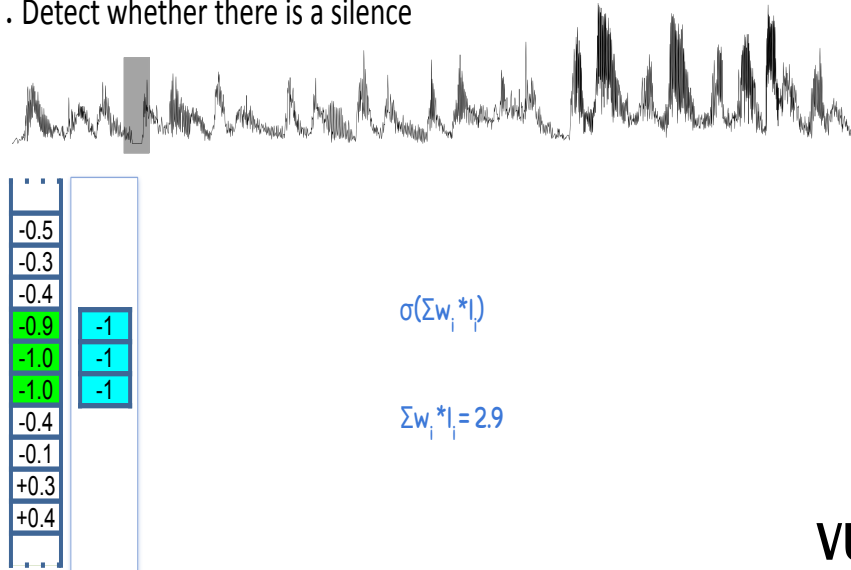
. Detect whether there is a silence



animation: 3

1D - SOUND - FILTERS - CONVOLUTIONS

. Detect whether there is a silence



This image is now simplified to the essential parts, the input to the MLP is in green, the filter in cyan.

This filter is supposed to detect silence, but how does would it perform on other parts?

Now, let's simplify this further because there's a lot of unnecessary detail, and what we'll be left with is something like this: we have our input values, and right beside them, we place our filter. Imagine this as a small MLP connected to these input neurons. It includes a bias, all the necessary connections, and basic computations. Now, in this case, we can compute the output of the neuron depicted here. This output is essentially the result of multiplying these values with each other, followed by summation. At some point, we apply a non-linearity, and the specific nature of that non-linearity remains open-ended, as various types are possible.

A crucial question we need to address is whether this "silence filter" or "noise filter," as we refer to it, truly functions as intended. By "functioning," I mean that it should yield a high value in the presence of silence and a low value when silence is absent. Let's test it out. We'll begin at the beginning of the sound, and you'll notice that we obtain a value of 1.2, significantly lower than what we had seen before. As we

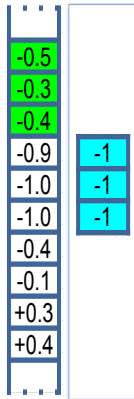
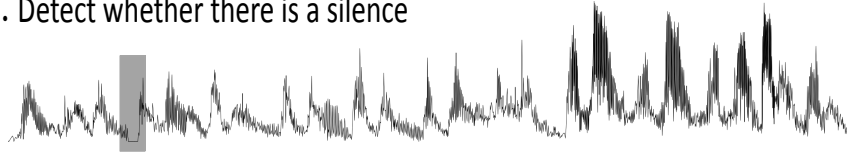
move down the sound, the value increases, reaching 1.6, still relatively low. However, as we approach the period of silence, the value surges, with the highest value, 5, achieved right at the point of silence. After this silent interval, the value begins to decrease again.

Does it give a low signal when there is sound? Let us try...

animation: 1

1D - SOUND - FILTERS - CONVOLUTIONS

. Detect whether there is a silence



$$\sigma(\sum w_i * I_i)$$

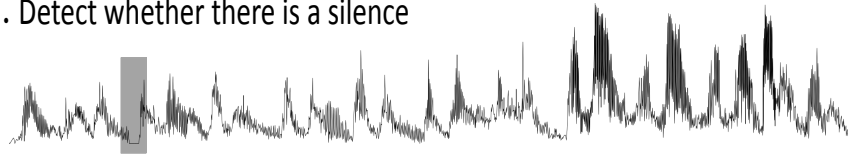
$$\sum w_i * I_i = 1.2$$



animation: 2

1D - SOUND - FILTERS - CONVOLUTIONS

. Detect whether there is a silence



$$\sigma(\sum w_i * I_i)$$

$$\sum w_i * I_i = 1.6$$



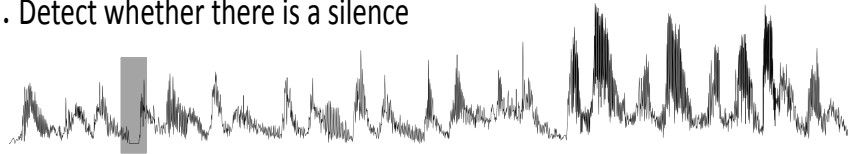
This image is now simplified to the essential parts, the input to the MLP is in green, the filter in cyan.

This filter is supposed to detect silence, but how does would it perform on other parts?

Does it give a low signal when there is sound? Let us try...

1D - SOUND - FILTERS - CONVOLUTIONS

. Detect whether there is a silence



$$\sigma(\sum w_i * I_i)$$

$$\sum w_i * I_i = 2.3$$



This image is now simplified to the essential parts, the input to the MLP is in green, the filter in cyan.

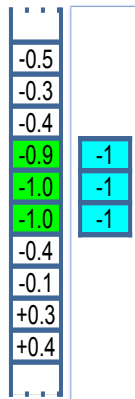
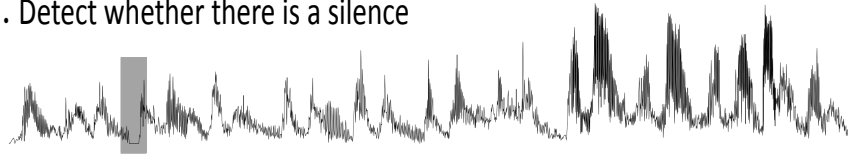
This filter is supposed to detect silence, but how does would it perform on other parts?

Does it give a low signal when there is sound? Let us try...

animation: 1

1D - SOUND - FILTERS - CONVOLUTIONS

. Detect whether there is a silence



$$\sigma(\sum w_i * I_i)$$

$$\sum w_i * I_i = 2.9$$



This image is now simplified to the essential parts, the input to the MLP is in green, the filter in cyan.

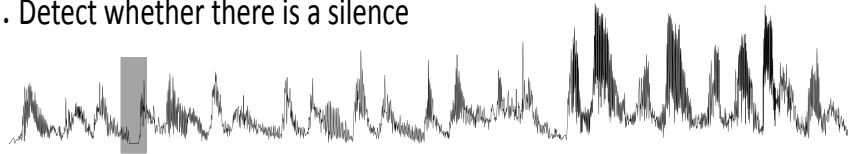
This filter is supposed to detect silence, but how does would it perform on other parts?

Does it give a low signal when there is sound? Let us try...

animation: 2

1D - SOUND - FILTERS - CONVOLUTIONS

. Detect whether there is a silence



$$\sigma(\sum w_i * I_i)$$

$$\sum w_i * I_i = 2.4$$



This image is now simplified to the essential parts, the input to the MLP is in green, the filter in cyan.

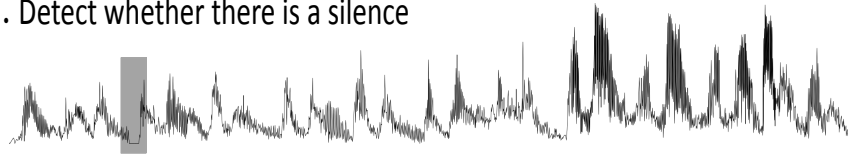
This filter is supposed to detect silence, but how does would it perform on other parts?

Does it give a low signal when there is sound? Let us try...

animation: 3

1D - SOUND - FILTERS - CONVOLUTIONS

. Detect whether there is a silence



$$\sigma(\sum w_i * I_i)$$

$$\sum w_i * I_i = 1.5$$



This image is now simplified to the essential parts, the input to the MLP is in green, the filter in cyan.

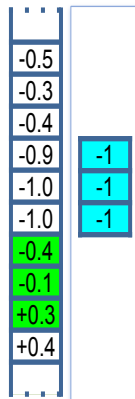
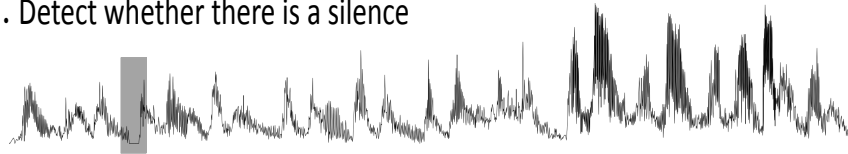
This filter is supposed to detect silence, but how does would it perform on other parts?

Does it give a low signal when there is sound? Let us try...

animation: 4

1D - SOUND - FILTERS - CONVOLUTIONS

. Detect whether there is a silence



$$\sigma(\sum w_i * I_i)$$

$$\sum w_i * I_i = 0.2$$



This image is now simplified to the essential parts, the input to the MLP is in green, the filter in cyan.

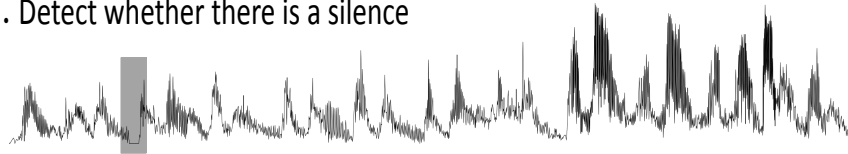
This filter is supposed to detect silence, but how does would it perform on other parts?

Does it give a low signal when there is sound? Let us try...

animation: 5

1D - SOUND - FILTERS - CONVOLUTIONS

. Detect whether there is a silence



$$\sigma(\sum w_i * I_i)$$

$$\sum w_i * I_i = -0.6$$



This image is now simplified to the essential parts, the input to the MLP is in green, the filter in cyan.

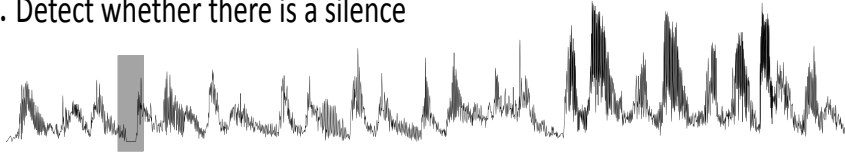
This filter is supposed to detect silence, but how does would it perform on other parts?

Does it give a low signal when there is sound? Let us try...

animation: 6

1D - SOUND - FILTERS - CONVOLUTIONS

- . Detect whether there is a silence



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**

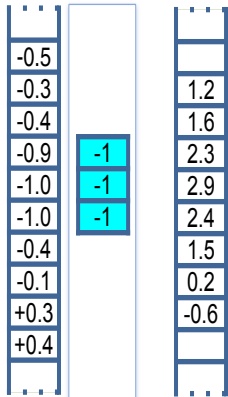
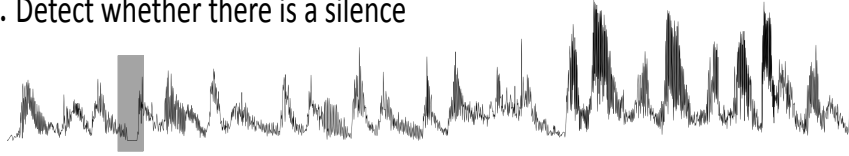


There are two essential observations from what we tried.

1. The filter can act as a feature extractor.
2. The filter can be used at other locations as well and will extract the same feature, but locally.

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**.



There are Three essential observations from what we tried.

1. The filter can act as a feature extractor. The output is itself a vector.
2. The filter can be used at other locations as well and will extract the same feature, but locally. If there is a time shift in the sound, the same shift will be visible in the output vector. This means that the convolution operator is translation equivariant.
3. The convolution is just an MLP, but its connections are localized and weights are tied, meaning that they are kept the same for specific connections.

PART TWO-b: conv1D

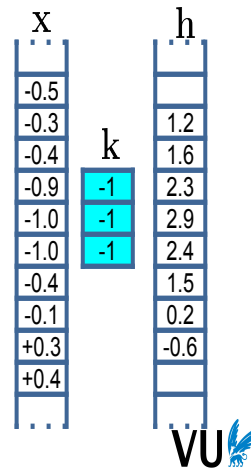


For this part, we will stay within one dimensional convolutions, as we were doing in the previous video.

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:

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



To better grasp how convolutional kernels are learned, we'll delve into a more formal definition. We're still operating within a one-dimensional input sequence. Our input sequence, denoted as \mathbf{x} , is n -dimensional, comprising n floating-point real numbers. In addition, we have a filter, which also possesses a specific length. This length is expressed as $2n+1$, and it's designed to be uneven. In this context, we only support filters with uneven lengths, such as one, three, five, or seven.

Now, what precisely does this convolution operation entail? The formula we see here aims to describe how to compute one of the network's outputs. Let's focus on the output at position T , denoted as $H(T)$. It's essentially a summation of various elements. We've previously encountered the concept of convolution, which entails the summation of input neurons multiplied by their corresponding weights from filter K . To formalize this, we examine the vector \mathbf{x} and identify the part that corresponds to our position T . Then, we traverse m steps both upwards and downwards from this position. In other words, we traverse from $-m$ to m and calculate the dot product of these values with their respective counterparts in filter K . This computation forms the core of the convolution operation. Afterward, we apply the bias, along with our non-linearity. In essence, convolution boils down to

this fundamental operation of multiplication or dot product.

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
- $\mathcal{L}_h = \sum_{t=0}^n \mathcal{L}_h(t)$ 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:

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

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

Always zero,
except when
 $\tau_0 = \tau$



In this context, where we're training a learnable filter, like our silence detector, we employ data samples and backpropagation to update this filter. This entails updating the filter weights based on a specific loss. The loss function, represented here, can vary and depend on factors such as the convolutional response at various locations. For instance, it may align with the cross-correlation in classification tasks, where a better classification results in a lower loss.

To compute the updates to the filter, we need the gradient information to determine in which direction the filter values should be adjusted in each iteration. This requires calculating the derivative of the loss with respect to the filter weights. In this expression, the final term, which pertains to the weight at position τ in the filter, will typically be zero except when τ is equal to τ_0 . τ_0 corresponds to the index of the weight being updated, and it is equal to the specific position where the filter is applied. In other words, these weights are adjacent when the filter operates.

1D CONVOLUTION - FORMAL DEFINITION

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

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

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: $\mathbf{h}(t) = \mathbf{W}(t, :) \cdot \mathbf{x}(t)$)



Another noteworthy observation is that when we calculate the updates for this weight, we take into account all elements in the input sequence, ranging from zero to n. In essence, we consider how each element in the input sequence affects the updates for the position of the filter. This implies that these weights are genuinely shared across the entire input. In contrast, the Multi-Layer Perceptron (MLP) learns independent weights for each position, without this sharing property.

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.



There are several advantages to this approach. Applying filters across the entire dataset makes convolutional neural networks significantly smaller in comparison to Multi-Layer Perceptrons (MLPs). For instance, a feature detector for noise, as demonstrated earlier, involves only three weights. In contrast, designing a comprehensive silence detector with an MLP would necessitate an extensive network to identify silences effectively. This showcases an outstanding parameter efficiency.

Convolutional neural networks also excel at becoming potent pattern recognizers. For example, in the case of detecting silences, a CNN can recognize silences appearing anywhere in the input. In contrast, an MLP would need to learn independent silence patterns for every possible position. This leads to exceptional data efficiency, making the most out of available data. Convolutions can recognize patterns, such as silence, regardless of their specific location, making them versatile. Even if a silence occurs in a position that hasn't been seen before, the convolutional network can adapt effectively, as it's designed to identify the presence of silences anywhere.

In summary, convolutional neural networks offer substantial benefits in terms of generalization improvement. It's worth noting that points two

and three, as indicated in these bullet points, are direct outcomes of the convolution's property of translation equivariance. We'll delve deeper into this concept in the upcoming discussion.

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?

60



So, let's summarize our progress thus far. We've been primarily focused on filters and discussed how they effectively act as an MLP but with the added capability of moving over the input data, generating an output vector. However, we're not only interested in detecting silence; to address the complex questions we've raised, we require more. We need filters to identify various features, such as regions with high noise, specific rhythms, and more. This necessitates multiple filters; a single filter won't suffice.

The next question we need to address is how to handle the data's beginning and end. Our analysis has mainly centered on the middle of the data, but we also need to consider the data's start and finish. Additionally, we've thus far examined a single layer, but it's important to think about the behavior of subsequent layers.


Lastly, we must consider dimension reduction. While applying a filter, we've observed that the output maintains the same dimension as the input. In our network, we need to reduce the dimension to match the number of classes, for example, and further compress it as needed.

1D - CONVOLUTIONS

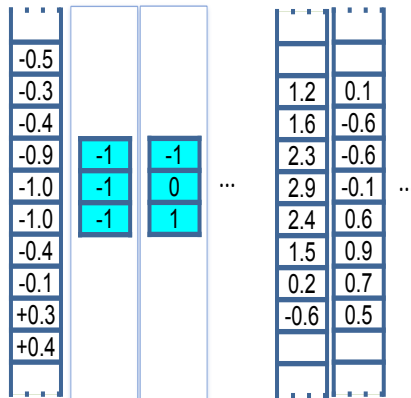
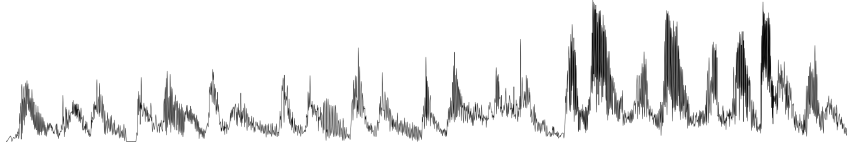
Up till now:

- . 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



- . 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**



We need more as just a silence feature extractor. We are, for example also interested in detecting sudden drops, specific frequencies, periods with sustained sounds, etc.

A first thing we do is adding multiple filters supporting different features. The number of filters is chosen by the network designer.


Now, the output becomes a matrix; **the output volume**. The columns of this volume correspond to each filter. The row in this volume corresponds to a certain group of entries in the input 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?

63



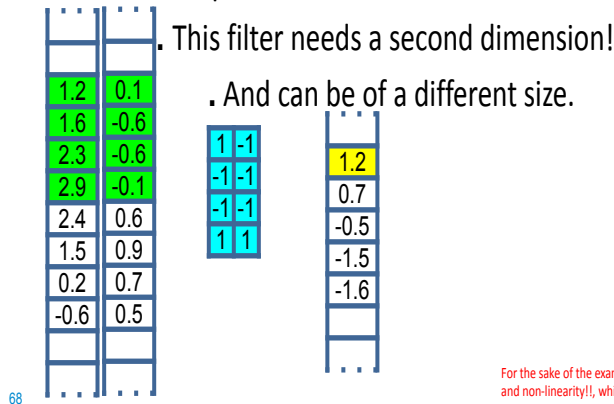
Let's begin with the need for multiple filters. As mentioned, we require numerous filters for feature extraction. The key point is that our design doesn't restrict us to having just one filter; we started with one as an example, but in practice, we can create as many filters as necessary. Whether you need two filters, ten filters, or any number, it's entirely feasible. The main difference is that instead of having a single output factor, you'll obtain multiple output factors. Each filter generates its own factor, and if you have more filters, you'll generate a corresponding number of factors.

This transformation takes us from an initial dimension, denoted as $n \times 1$ or $1 \times n$, and leads to a new dimension, which depends on the number of filters employed. We refer to these outputs as the "output volume." Currently, it's represented as a 2D matrix, but it can expand further in the subsequent steps. So, we're essentially moving from one-dimensional input to a multi-factor output volume.

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!



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



We've established that we can utilize multiple filters, which allows us to transition from a one-dimensional output to a multi-dimensional one in the form of a matrix. The next layer in our network necessitates a different approach; we can't merely replicate the same process. Instead, we redefine our convolutional operators for this layer.

In this new layer, when we apply a convolutional operator, it operates on the data received from our previous multiple filters. Each filter in this layer still resembles a typical filter, but now it has an added dimension. It operates across the entire data, creating a dot product that results in a multi-component output.

This allows us to return to an output volume with just one dimension, but it's important to note that we can also employ multiple filters in this layer, yielding multiple output vectors.

One crucial observation is that the meaning of the filters in these subsequent layers depends on the meaning of the filters in the previous layers. As we progress through the layers, the filters become more intricate and exhibit complex meanings. For instance, if you have a filter like the "silence filter" in the first layer and a "high-frequency noise" filter before it, the filter in this layer combines the meanings of the previous filters. It might represent, for instance, a

period of silence followed by a high-pitched sound.

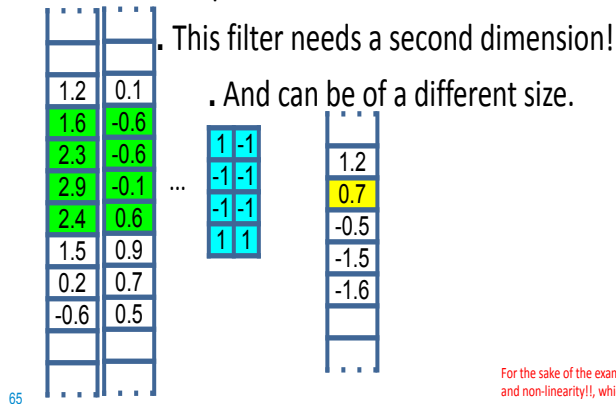
With each layer, the neurons producing the output start to acquire more semantic meaning. This meaning evolves into context; it's not just a high-pitched noise but may signify the presence of a specific beat in music. As you proceed deeper into the network, the features become increasingly refined, and eventually, you might be able to classify music based on the output.

animation: 1

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!



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

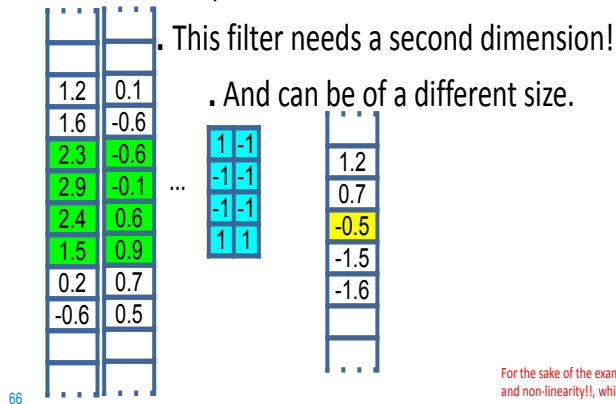


animation: 2

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!



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

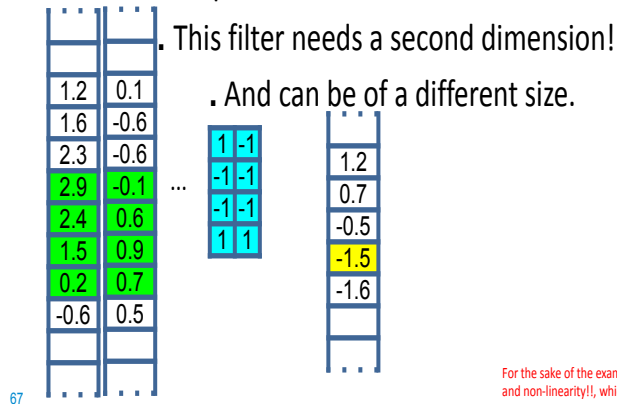


animation: 3

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!



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

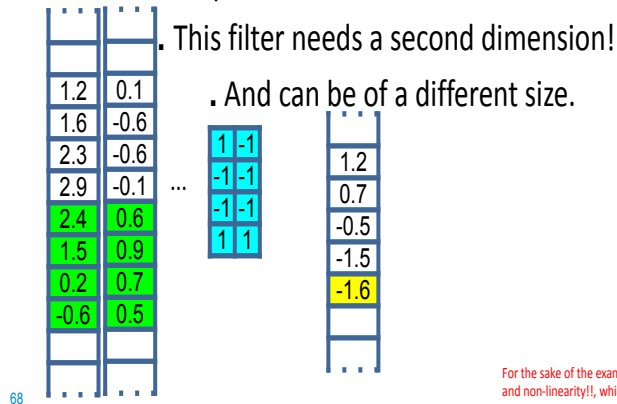


animation: 4

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!



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



animation: 5

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!

...	...
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
...	...

• This filter needs a second dimension!

• And can be of a different size.

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

...
1.2
0.7
-0.5
-1.5
-1.6
...

- The meaning of these filters recursively depends on the meaning of the filters on the layer before. But, overall becomes more complex.

69

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




animation: 6

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?

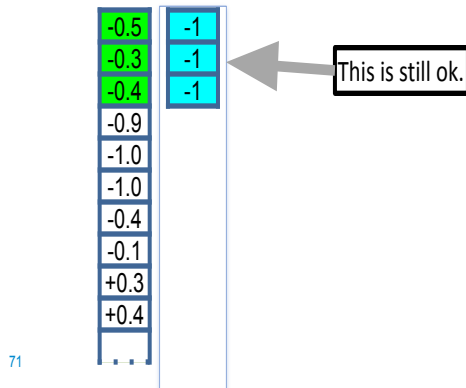
70



Okay, so we have talked about multiple filters and what we can do in the next layers of this convolutional neural network. Now, there is still an issue at the start and end of the data. So let's look at an example.

1D - CONVOLUTIONS - PADDING

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



As we approach the start of our data, a challenge arises near the boundaries. Near the middle of the data, applying convolution works well. However, a problem arises when we move one step further toward the beginning. The problem is that we lack input data for this initial position in our filter.

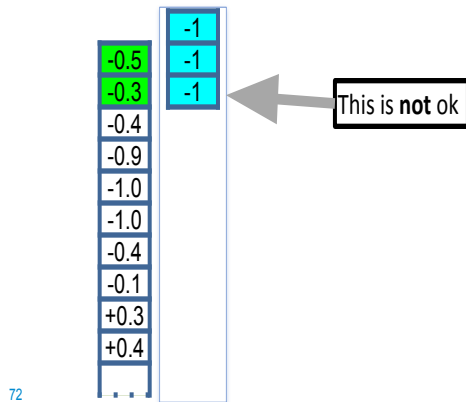
One solution could be to simply ignore the boundaries and start one position lower. However, this approach leads to a reduction in dimension, effectively trimming one value from our data. This means that information near the boundaries is lost.

The more common solution is to pad the boundaries. Padding involves adding extra values that aren't part of the original data and then applying the filter. In the example, zero padding is used at the top, and the convolutional filter is applied. This produces an output, as demonstrated. While padding introduces some artifacts because the exact values are uncertain, it typically yields better results than not padding and reducing the data size.

animation: 1

1D - CONVOLUTIONS - PADDING

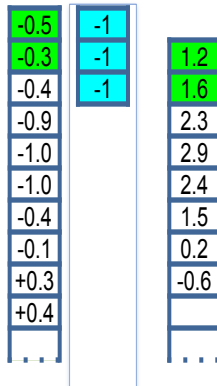
- Near the boundaries of the data, we cannot apply the convolution as we normally do:



animation: 2

1D - CONVOLUTIONS - PADDING

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



73

Solutions:

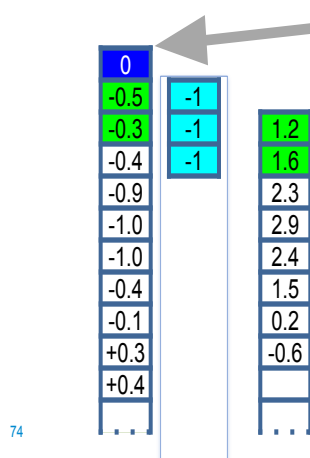
- Ignore the boundaries
 - This also leads to a reduction of dimension!
 - information at the boundaries gets lost



animation: 3

1D - CONVOLUTIONS - PADDING

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



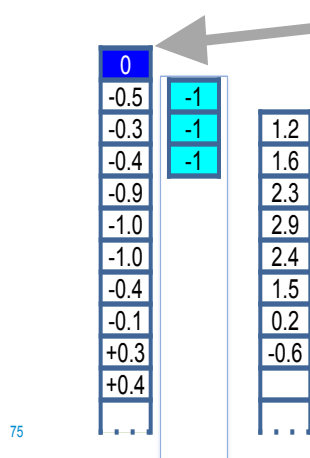
Solutions:

- ignore the boundaries
- Pad the boundaries
 - Add $(\text{filter length}-1)/2$ around the data to preserve the dimension
 - Fill this with a fixed value, often 0

animation: 4

1D - CONVOLUTIONS - PADDING

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



Solutions:

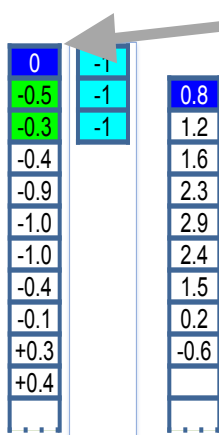
- ignore the boundaries
- Pad the boundaries
 - Add $(\text{filter length}-1)/2$ around the data to preserve the dimension
 - Fill this with a fixed value, often 0



animation: 5

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 $(\text{filter length}-1)/2$ around the data to preserve the dimension
 - Fill this with a fixed value, often 0


animation: 6

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:

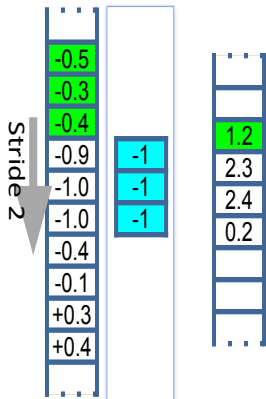
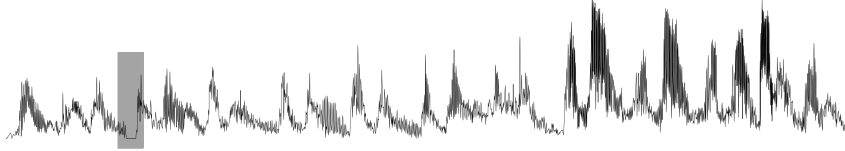
- . We need multiple filters
- . What at the next layer?
- . What do we do at the start and end of the data?
- . How do we get the dimension down? 

77



So we have now seen what to do at the start and the end of our input data. Finally, we have to talk about how we get the dimension down. Here, we want to get the dimension lower in a controlled way.

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**

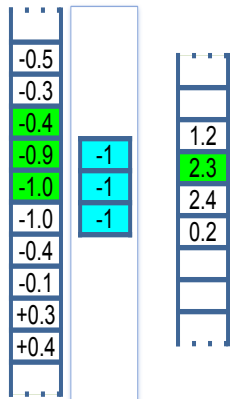
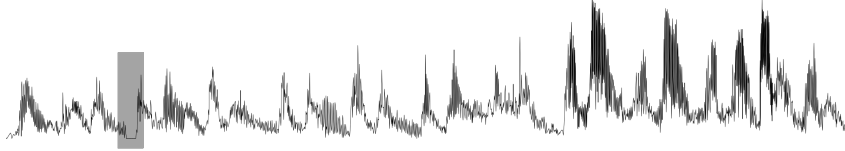


To in the end be able to do a classification, we need to get the dimension down. We do not want it by not padding, as that is destructive for the boundaries of the data.

A first option is to use a different **stride** while convolving the filters. that means, make bigger steps.

animation: 1

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**

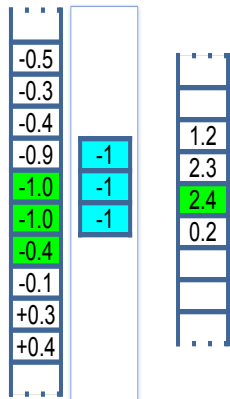
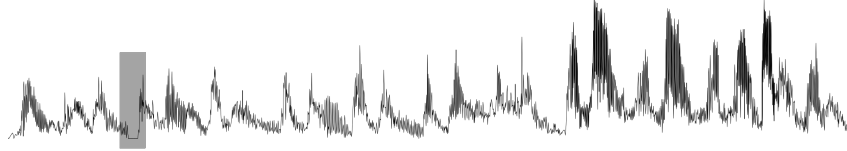


To in the end be able to do a classification, we need to get the dimension down. We do not want it by not padding, as that is destructive for the boundaries of the data.

A first option is to use a different **stride** while convolving the filters. that means, make bigger steps.

animation: 2

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**

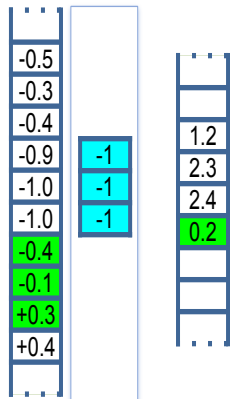
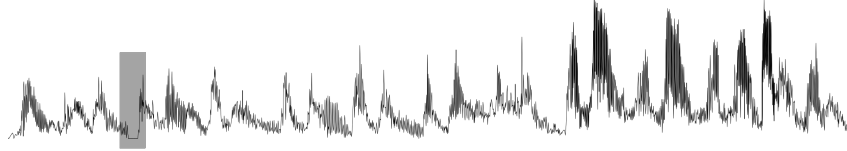


To in the end be able to do a classification, we need to get the dimension down. We do not want it by not padding, as that is destructive for the boundaries of the data.

A first option is to use a different **stride** while convolving the filters. that means, make bigger steps.

animation: 3

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**

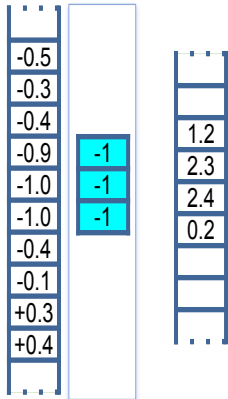
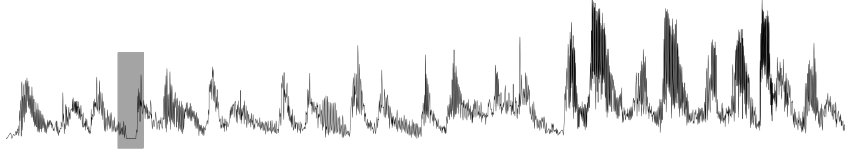


To in the end be able to do a classification, we need to get the dimension down. We do not want it by not padding, as that is destructive for the boundaries of the data.

A first option is to use a different **stride** while convolving the filters. that means, make bigger steps.

animation: 4

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!**

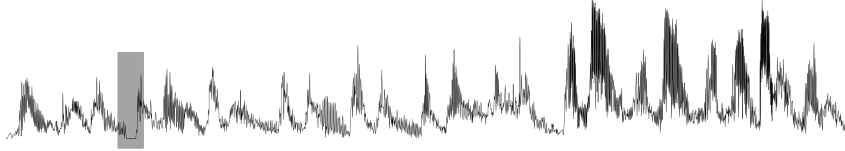


To in the end be able to do a classification, we need to get the dimension down. We do not want it by not padding, as that is destructive for the boundaries of the data.

A first option is to use a different **stride** while convolving the filters. that means, make bigger steps.

This leads to an output volume which has a dimension equal to the input dimension divided by the stride. To make this work, the input dimension must be a multiple of the stride.

1D - CONVOLUTIONS - POOLING



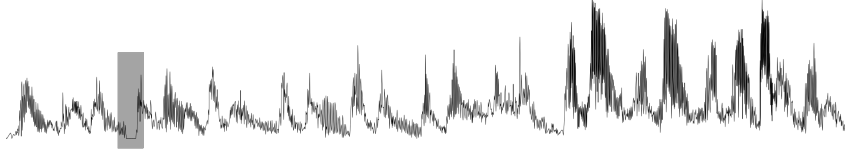
- We need to reduce dimension for the final classification
 - Solution 1: use a larger **stride**
 - Solution 2: use a **pooling layer**



To in the end be able to do a classification, we need to get the dimension down. We do not want it by not padding, as that is destructive for the boundaries of the data.

The second option is the use of a **pooling layer**. To put it simple, a pooling layer takes a set of elements from the previous layer and squeezes that down to one output in a deterministic way. Examples include a **max pooling and average pooling**. The former takes the biggest element of the input and only outputs that. The latter takes the average of the inputs and forwards that.

1D - CONVOLUTIONS - POOLING



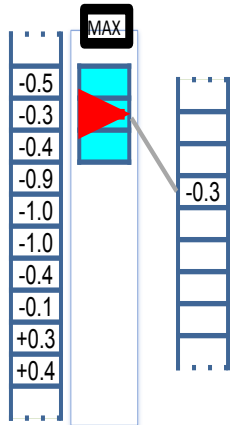
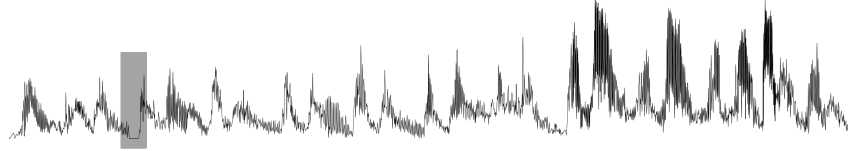
- We need to reduce dimension for the final classification
 - Solution 1: use a larger **stride**
 - Solution 2: use a **pooling layer**
 - Goes over the data similar to a convolution
 - Applies a deterministic function like max or average on the input
 - Usually has stride == pool size



To in the end be able to do a classification, we need to get the dimension down. We do not want it by not padding, as that is destructive for the boundaries of the data.

The second option is the use of a **pooling layer**. To put it simple, a pooling layer takes a set of elements from the previous layer and squeezes that down to one output in a deterministic way. Examples include a **max pooling and average pooling**. The former takes the biggest element of the input and only outputs that. The latter takes the average of the inputs and forwards that.

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



To in the end be able to do a classification, we need to get the dimension down. We do not want it by not padding, as that is destructive for the boundaries of the data.

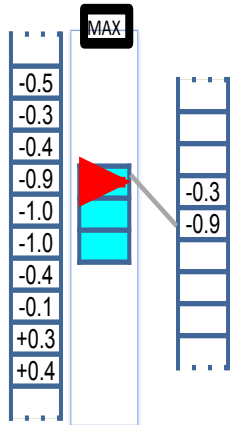
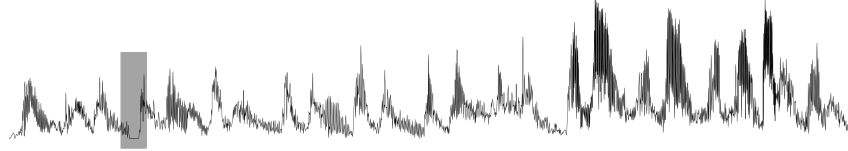
The second option is the use of a **pooling layer**. To put it simple, a pooling layer takes a set of elements from the previous layer and squeezes that down to one output in a deterministic way. Examples include a **max pooling and average pooling**. The former takes the biggest element of the input and only outputs that. The latter takes the average of the inputs and forwards that.

In the example, we look at a **max** pooling layer which looks at 3 elements at a time.; or in other words, the pool size is 3. Its stride is also 3.

Hence, the output dimension will be 3 times lower as the input dimension.

animation: 1

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



To in the end be able to do a classification, we need to get the dimension down. We do not want it by not padding, as that is destructive for the boundaries of the data.

The second option is the use of a **pooling layer**. To put it simple, a pooling layer takes a set of elements from the previous layer and squeezes that down to one output in a deterministic way. Examples include a **max pooling and average pooling**. The former takes the biggest element of the input and only outputs that. The latter takes the average of the inputs and forwards that.

In the example, we look at a **max** pooling layer which looks at 3 elements at a time.; or in other words, the pool size is 3. Its stride is also 3.

Hence, the output dimension will be 3 times lower as the input dimension.

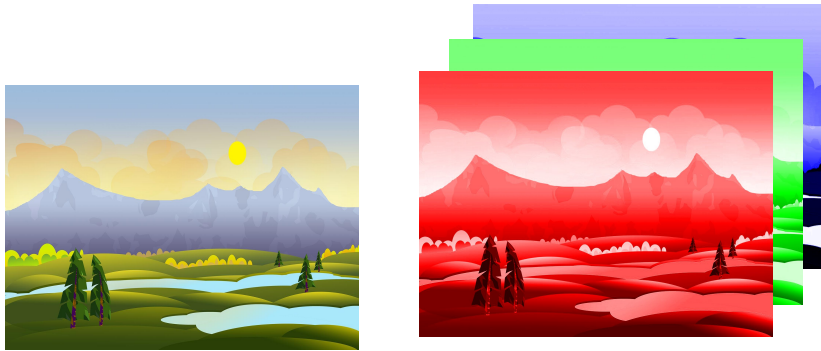
animation: 2

PART THREE: Conv2D, Conv3D, ConvND



In this third part, we aim to expand our understanding beyond one-dimensional convolution. Our goal is to delve into more intricate forms of data processing, such as two-dimensional convolution, three-dimensional convolution, and so forth. The motivation for venturing into higher dimensions is our desire to work with more diverse and intriguing types of data.

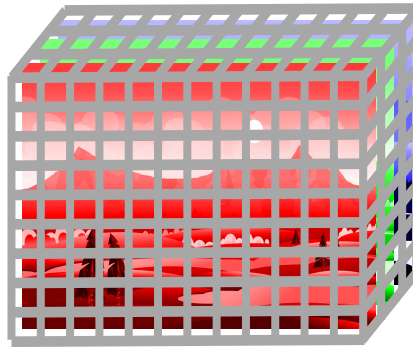
2D - IMAGES - REPRESENTATION



For instance, when we examine images, we've already observed that we move from two dimensions to three dimensions. In the case of images, we typically work with three color channels, often represented as RGB (red, green and blue). One way to conceptualize this is by envisioning an image as a three-dimensional tensor, which accommodates these three channels.

2D - IMAGES - REPRESENTATION - 3D TENSOR

The 2 dimensional color image becomes a 3 dimensional tensor!

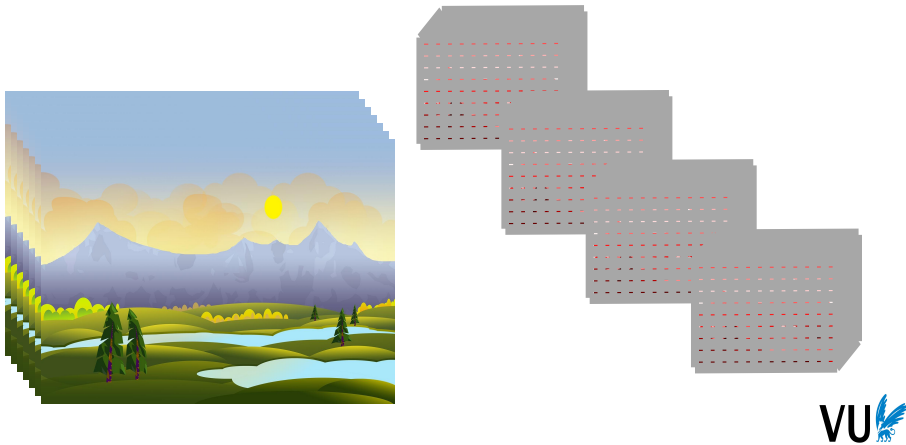


VU 

As we dive into working with this type of data, it becomes apparent that our one-dimensional approach is no longer adequate. The complexity increases, particularly when dealing with video data.

3D - VIDEO - REPRESENTATION - 4D TENSOR

A 3 dimensional color video becomes a 4 dimensional tensor!



A video essentially transforms into a four-dimensional tensor because it comprises a sequence of images. This sequence of images, in turn, can be thought of as a sequence of tensors, culminating in a four-dimensional tensor representation.

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>



In general, we aim to engage with this type of multidimensional data. To begin with, let's consider an example, a five-by-five image—a matrix of pixel intensities. It's essential to note that we don't deal with just one of these matrices; we have multiple color channels to take into account.

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

We start with a 5x5 image

We have 3 channels

`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]`

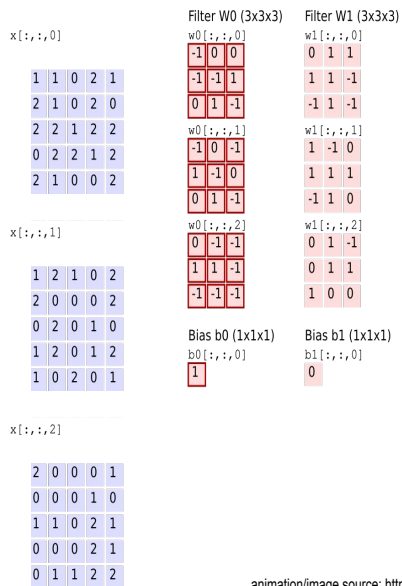
2	0	0	0	1
0	0	0	1	0
1	1	0	2	1
0	0	0	2	1
0	1	1	2	2

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



In this case, as explained in the example, we have three color channels to consider. Next, we move on to defining our filters.

2D - IMAGES - CONVOLUTION - FILTERS



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>



In this specific example, we opt for two filters, but it's important to note that the number of filters can be chosen as needed. For filter W0, the first filter, it exhibits increased dimensions. This depiction illustrates a three-dimensional cube, with dimensions of three by three by three. The depth of this filter, the last dimension, aligns with the number of input channels available. Meanwhile, the width and height of the filter can be selected freely; in this case, it's three by three. However, it could have just as well been five by five, for instance. Additionally, it's worth mentioning that there's an explicit inclusion of a bias node, ensuring that the bias is accounted for.

2D - IMAGES - CONVOLUTION - PADDING

Input Volume (+pad 1) (7x7x3)	Filter W0 (3x3x3)	Filter W1 (3x3x3)
$x[:, :, 0]$ 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	$w0[:, :, 0]$ -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 b_0 (1x1x1) $b_0[:, :, 0]$ 1	$w1[:, :, 0]$ 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 b_1 (1x1x1) $b_1[:, :, 0]$ 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		

We start with a 5x5 image

We have 3 channels

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

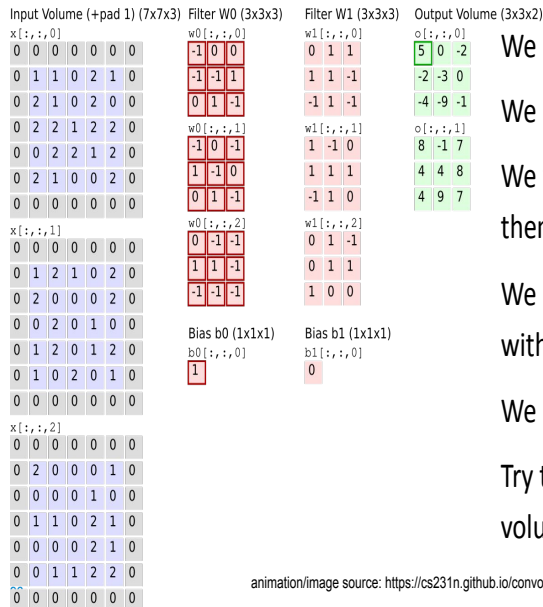
We add padding to solve issues with convolving near the border

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



The challenges related to the boundaries of data persist when dealing with data in two or more dimensions. In two dimensions, such as in this case, we encounter not just the start and end of the data, but also the sides. For instance, when applying convolution at a corner point like this, it becomes evident that we require extra padding on both sides. Therefore, the approach is to add padding around the image, which, in this instance, involves adding one pixel of padding. This augmentation results in an input data size of seven by seven, rather than the original five by five.

2D - IMAGES - CONVOLUTION - OUTPUT VOLUME



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>



Lastly, we must consider the nature of the outputs we're generating. The dimensions of our output depend on both our input size and the filter. In this particular scenario, we've applied a stride of 2. Before delving further into the topic, I recommend taking a brief pause to reflect or attempt to grasp the reasoning behind the resulting output dimensions. Moving forward, we'll explore the precise calculations that lead to these output dimensions.

2D - IMAGES - REPRESENTATION

Input Volume (+pad 1) (7x7x3)	Filter W0 (3x3x3)	Filter W1 (3x3x3)	Output Volume (3x3x2)
$x[:, :, 0]$ 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	$w0[:, :, 0]$ -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	$w1[:, :, 0]$ 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	$o[:, :, 0]$ 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

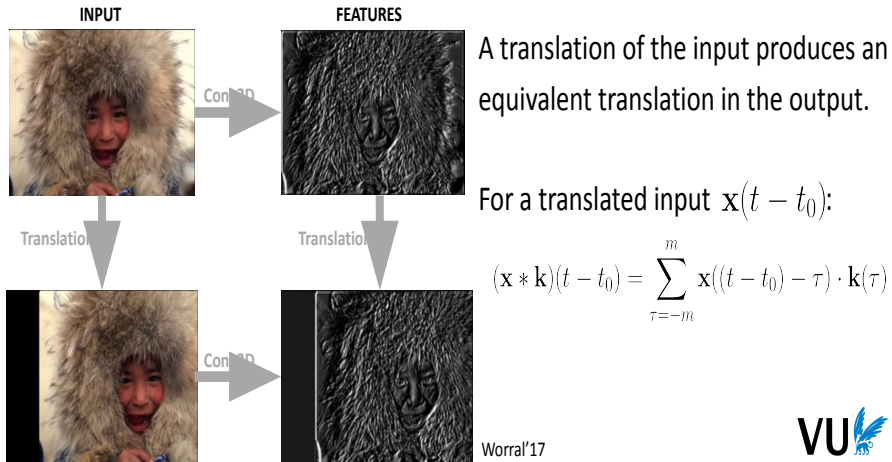
animation/image source: <https://cs231n.github.io/convolutional-networks/#fc>



Let's work on a 2D matrix, i.e., one color channel of an image.

CONVOLUTION - EQUIVARIANCE ANALYSIS

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



Let's delve into a couple of intriguing characteristics. As we mentioned earlier, the concept of equivariance, especially in the context of images, is quite significant. In this example, we have an image on which we apply a 2D convolution, specifically an edge detection filter. This filter helps us detect certain features in the image.

What's particularly interesting about this is the equivariance property of convolutions. When you first translate your input, such as moving the image to the right, and then apply our convolutional filter, you essentially obtain the same output, but it's shifted to the right as well. This property is crucial because it means that if certain features exist in your input, it doesn't matter where they are located. Your filter can detect these features regardless of their position. For instance, if you have a person in an image, it doesn't matter where within the image that person appears; your system or filter will identify it as a person.

This property is already visible in the formal definition. When we include the shift operation by using T_0 in the convolution formula, we can observe that the result remains the same, but it is shifted by the same amount. This emphasizes the equivariance property of convolutions, making them an essential tool for recognizing features across different locations in an image.

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.



Let's discuss an interesting aspect related to the equivariance property. It's fascinating that features can occur anywhere in the data, but for images, there are other transformations we'd like our system to handle, such as rotation and scaling. We want the ability to recognize a person, for example, even if they're upside down, and this is where traditional convolutions have limitations. When you rotate an image, the filter doesn't function as effectively anymore, and it may fail to detect an upside-down person.

This limitation poses a challenge because we need our system to be robust against various transformations, especially in computer vision applications. One approach to address this is training a convolutional neural network with permutations of the data. Essentially, you would take an image of a person and train the system not only with the original image but also with versions that have been rotated by 90 degrees, 180 degrees, and 270 degrees. By exposing the network to these multiple variations of the same data, you can enhance its robustness against rotation without fundamentally changing the system architecture.

However, this approach has some drawbacks, as it requires training the model with many permutations of the same data, which can be

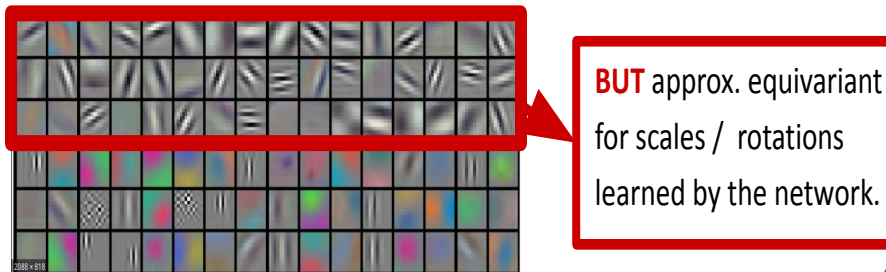
computationally expensive and not very efficient in terms of data utilization.

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.



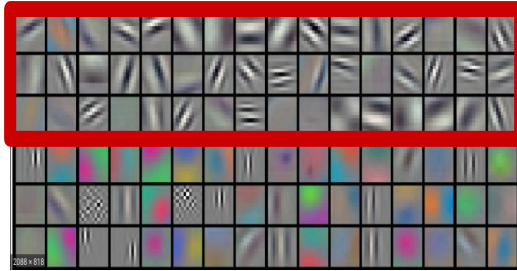
VU

Convolutional neural networks exhibit an interesting property in their filter design, even though they aren't explicitly designed to handle rotations. When we look at the filters in a convolutional network for images, we can observe that these filters represent what the network is detecting in the input data. For instance, some filters detect lines with a particular slope.

What's fascinating is that within the set of filters, you often find filters that are essentially rotations of each other. In other words, filters that detect lines with varying orientations. For instance, you might have a filter that detects lines with a particular slope, and another filter that detects lines with the opposite slope.

This inherent property of the filters means that the convolutional network, while not explicitly designed for handling rotations, can still recognize them to some extent. As you go deeper into the network, you encounter higher-level features that also exhibit this ability to detect rotated versions of the underlying patterns. So, even though rotation robustness is not a primary design consideration, the network's architecture and filter diversity end up providing some level of rotation invariance.

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!**

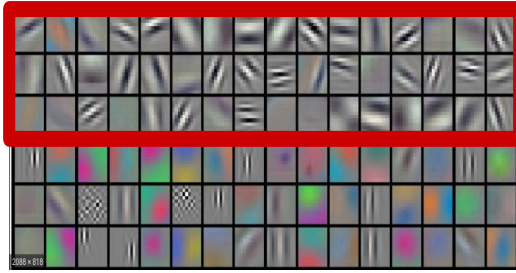


Group convolutions indeed address some of the challenges associated with parameter and data inefficiency in convolutional neural networks. In group convolutions, the idea is to share the pyramids not just for translation but also for other transformations like rotations and scaling. This approach results in extreme parameter sharing across the network.

By doing so, you make the network more efficient and data-efficient because it no longer requires explicit training on various transformations of the same data. It allows the network to generalize better across different transformations without explicitly observing them during training.

Group convolutions are particularly useful when working with data that can have multiple types of transformations or symmetries. They are often used in computer vision tasks to ensure that the network can handle different orientations, scales, and other transformations without the need for extensive data augmentation.

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 ;)



As you mentioned, this parameter-efficient and data-efficient approach is an interesting and effective way to tackle the limitations of standard convolutions, which are mainly translation-invariant. Group convolutions can offer better results and more robustness in situations where multiple transformations need to be considered.

If you have any more specific questions or want to explore real-world examples of convolutional neural networks, please feel free to ask.

PART FOUR: Example of a real world CNN



This part is an example of a real-world convolutional neural network. Specifically, we will be looking at the network called AlexNet, employed for the task of image classification.

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 marked a pivotal moment in deep learning, and it's safe to say that it demonstrated the feasibility of this approach. In 2012, it triggered great developments in deep learning and broader machine learning fields. The primary driver behind AlexNet's remarkable performance, and that of similar architectures around that time, can be attributed to its adept utilization of GPUs for accelerating convolution computations.

To elaborate, consider the intricacies of these convolutions. Each filter necessitates computing numerous dot products with the input image, demanding an immense computational load. This is where GPUs shine; they excel at executing these computations in parallel, thus greatly expediting convolution operations. The parallelization of tasks is executed with notable efficiency.

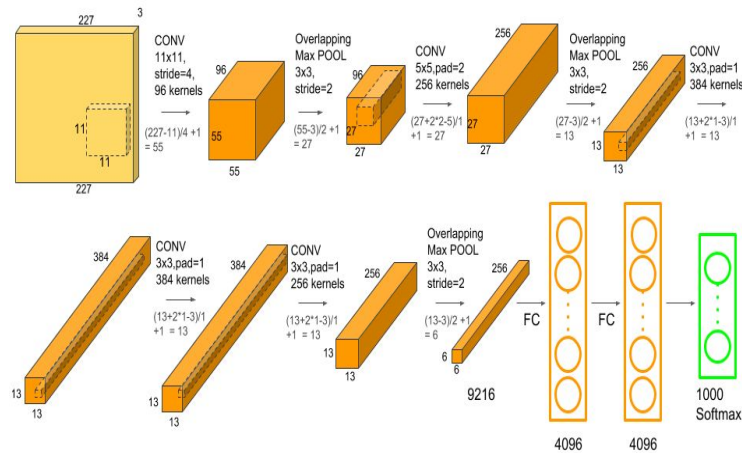
The essence of this work revolved around harnessing the computational power of GPUs available at the time. What's truly astounding is the extent to which it outperformed other systems reliant on manual feature extraction. Achieving a top-five error rate of 15.3 percent on a dataset featuring a thousand categories is quite exceptional. Handling a thousand different categories is a daunting challenge as the probability of achieving correct classifications purely

by chance in such a vast space is exceedingly small. This accomplishment signaled a significant turning point in the industry's acceptance of neural networks for practical applications.

It's noteworthy that this work, although revolutionary, is essentially a modern reiteration of an older concept. This idea harks back to a study from 1989, nearly three decades prior, which initially targeted handwritten digit recognition. While the core technique remains consistent, this reimagining's clever adaptation for GPU acceleration makes it an emblem of technological advancement.

Now, let's delve into an exploration of this network and dissect its architectural components, shall we?

AlexNet



112

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



In this context, the initial input is a 227x227 pixel image with three color channels (RGB). The first layer involves a convolutional operation, similar to what we've discussed earlier. In this case, the convolutional filter is 11x11 in dimension and applies with a stride of four. To calculate the output volume dimensions, a formula subtracts the filter dimensions from the input dimensions and divides it by the stride, then adds one.

This first layer employs 96 different filters or kernels to process the image, each with an 11x11 dimension. The result is the first output volume. Following this is an "overlapping max pool" layer, where the stride is set to two instead of three, creating a slight overlap. This process also reduces the dimensions of the volume.

The network proceeds with another convolutional layer, this time using 5x5 filters with a padding of two and employing 256 of these filters. You can once more apply a similar formula to find the output volume dimensions in this case. This pattern continues with a max pooling layer, another convolutional layer, and so on.

As the network progresses, you'll notice it gradually evolves into a deep network with many filters. Finally, it applies fully connected layers, including two of them, culminating in a softmax function over a

thousand dimensions. This sequence creates a deep neural network architecture.

Exploring the "magic numbers" chosen for the filter dimensions—11, 3, 5, 3—might raise questions. While some aspects remain enigmatic, research is ongoing to gain a better understanding. It is known that smaller filters, such as 3x3 or 5x5, are typically more practical than larger ones. This is because smaller filters are stacked and, when examined recursively, impact a larger region of the image.

Much of the development following the 2012 paper has involved experimenting with different network configurations, seeking incremental improvements.

If we want to attach a neuron to this, so we're going to make first, say, a hidden layer, we need to have a neuron that receives input from the weights. That neuron also needs memory space to store the gradient, 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 in each of these layers. This requires a pretty 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

114



We've just explored the example of AlexNet, a deep architecture that relies on the fundamental concepts we've been discussing. Similar ideas underpin many other networks in this field.

Let's recap today's discussion. We began with an introduction, addressing the necessity of convolutional architectures. We delved into the one-dimensional case, examining why these architectures are indispensable. The primary reason is that using a multi-layer perceptron (MLP), as we've done previously, isn't feasible. An MLP grows too large, becomes impractical to train, and loses the ability to recognize features if they don't occur exactly in the same position as in the training data.

We used sound as an example to understand these concepts better. Subsequently, we extended our understanding to two-dimensional scenarios, applying similar principles like filters, padding, and deeper nested structures. We then scrutinized AlexNet, an example architecture, and mentioned that further developments exist in this field.

The key takeaway from this presentation is the realization that deep learning models can autonomously extract features from data, and these models can be applied across higher dimensions.