CNNs

June 20, 2025

Convolutional Neural Networks (CNNs)

FH University of Applied Sciences TECHNIKUM WIEN

Lecturer: Sharwin Rezagholi
Authors: Lars Mehnen, Matthias Blaickner, Bernhard Knapp, Sharwin Rezagholi

[Standing at the podium]

Good [morning/afternoon], everyone!

Welcome to today's lecture on **Convolutional Neural Networks**, or simply **CNNs**. This session will guide us through one of the most powerful and widely-used techniques in the field of deep learning—particularly when it comes to handling **images** and **spatial data**.

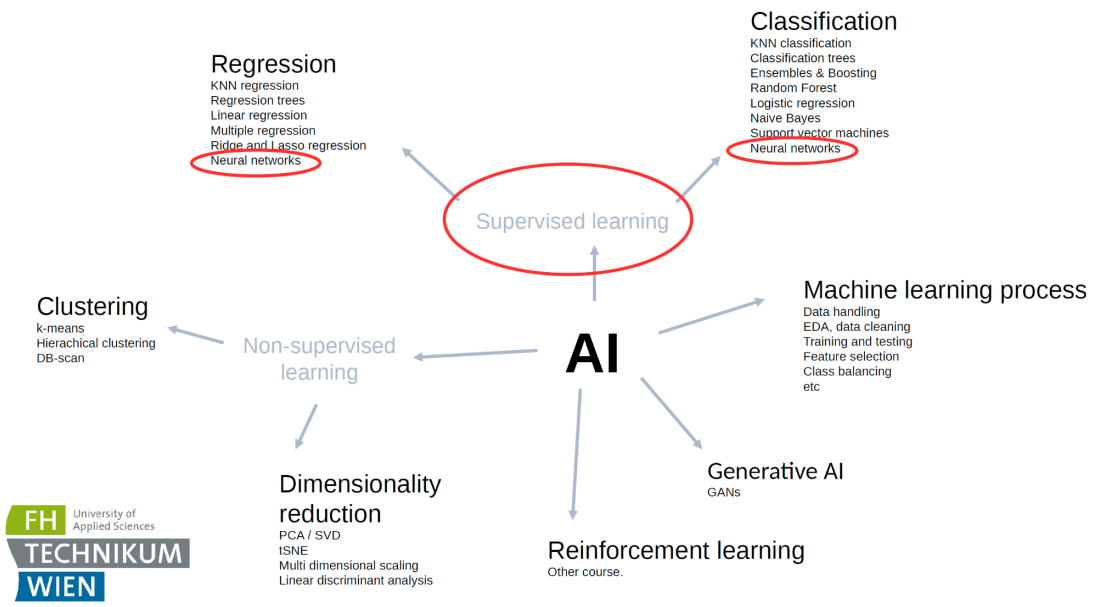
This topic is part of our broader Artificial Intelligence curriculum here at **FH Technikum Wien**, and it builds upon our previous work with traditional neural networks and supervised learning.

CNNs represent a huge step forward in making machines "see" the world—not just react to numbers, but interpret pixels, shapes, and even complex structures in visual inputs.

The content of this presentation is authored by **Lars Mehnen, Matthias Blaickner, Bernhard Knapp, and Sharwin Rezagholi**, who is also your lecturer today.

Let's move on and dive into the origins and motivation behind CNNs.

Next slide, please.



[Presenter voice, pointing at the diagram]

This diagram gives us an excellent **bird's-eye view** of how **Convolutional Neural Networks**—or CNNs—fit into the larger landscape of **Artificial Intelligence**.

At the heart, we have **AI**, branching into various domains:

- **Supervised learning**
- **Unsupervised learning**
- **Reinforcement learning**
- **Generative AI**

CNNs, as a specialized form of **neural networks**, clearly fall under **supervised learning**, where we train models on labeled data—images with known labels like "cat" or "dog", digits like "3" or "8", and so on.

In the supervised realm, neural networks are applied to both:

- **Classification tasks** (like object recognition)
- **Regression tasks** (less common for CNNs, but still possible, e.g. age prediction from face images)

The red ovals in the slide highlight where **neural networks** are typically involved, reinforcing that CNNs are a powerful subclass within this family.

Now, while today we're zooming into CNNs, it's important to understand their **context**—especially their position relative to things like Random Forests, SVMs, or KNN classifiers, which you may already be familiar with.

Let's now see *why* CNNs were needed in the first place. Next slide, please.

[With engaged tone, addressing the audience]

Let's begin with a brief **introduction to CNNs**.

Where did CNNs come from? They originally emerged from the field of **computer vision**—especially for tasks like recognizing handwritten digits. Think of the classic MNIST dataset: grayscale images of digits, 28 by 28 pixels, and the goal is to say whether it's a "3" or a "7".

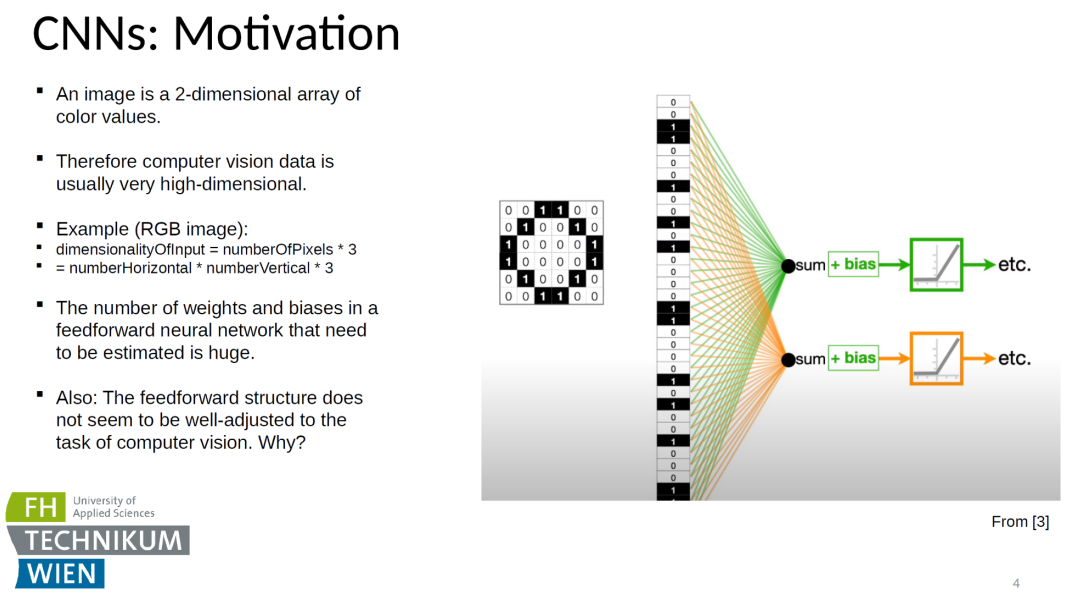
Why were CNNs needed? Because our **human visual system** doesn't process images pixel by pixel—we recognize patterns: edges, shapes, textures. CNNs are inspired by this intuition. Rather than treating all pixels equally (like a fully connected neural network does), CNNs can **focus on meaningful local structures** within the image.

What do CNNs do? They automatically **learn to extract patterns**—such as vertical lines, curves, or textures—at multiple levels of abstraction, helping machines understand visual content in a **hierarchical** way.

Beyond images? Yes! Although CNNs started in computer vision, their architecture also works well in other domains where **spatial locality or structure** exists: audio, time series, and even text (e.g., in sentence classification).

So: CNNs are powerful not because they mimic vision superficially, but because they **learn structured features** efficiently.

Now let's understand the **motivation** in a bit more technical detail. Next slide, please.



[Continuing, pointing at diagram]

Now let's talk **motivation**. Why can't we just use a traditional feedforward neural network to process images?

First: **What is an image?** An image is just a 2D array of **color values**—each pixel holds a value for red, green, and blue. That means an image with $w \times h$ pixels and RGB color has $w \times h \times 3$ features. For a modest 200x200 image, that's **120,000 inputs**!

Second: **High Dimensionality** That's a huge number of inputs! And if we feed this into a dense neural network, it means **millions of weights**—and that's just between the input layer and the first hidden layer.

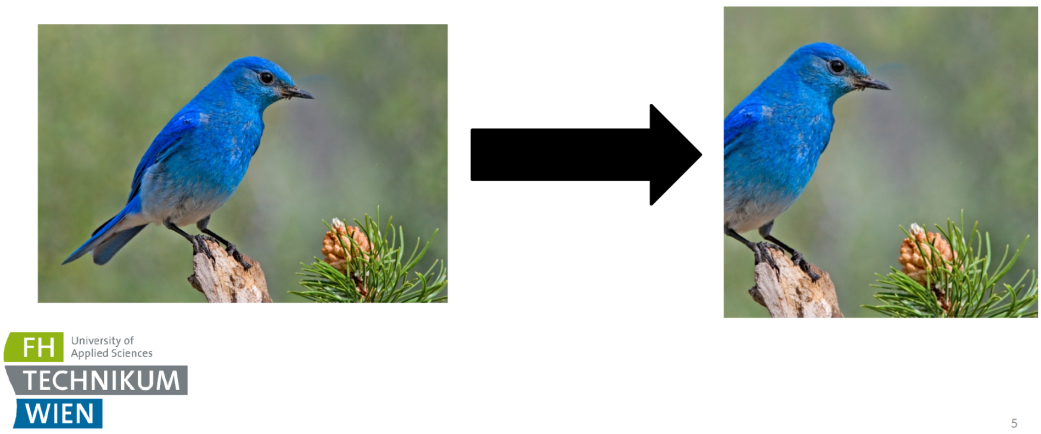
That's not only computationally expensive, but it's also highly prone to **overfitting**, especially if we have limited data.

Third: **Loss of spatial structure** Feedforward networks flatten the image into a single vector. So, the model has no idea that pixel (10,10) is right next to (10,11). It **loses the spatial relationships**—a massive drawback when the position and proximity of features is what defines an object.

This is where CNNs come in: They preserve spatial relationships, and they do **weight sharing** through kernels, dramatically reducing the number of parameters.

Let's now look at how CNNs solve this problem more intuitively.

Next slide, please.



[Smiling at audience, gesturing to the bird images]

Here's a **simple but powerful visual example** of why CNNs are necessary.

On the left, we see an image of a bluebird.

On the right, it's the **same bird**, slightly shifted—maybe it's been translated just a few pixels. For **you and me**, it's *obviously* still a bird. Nothing confusing at all.

But for a **traditional feedforward neural network**? This might as well be a completely different input!

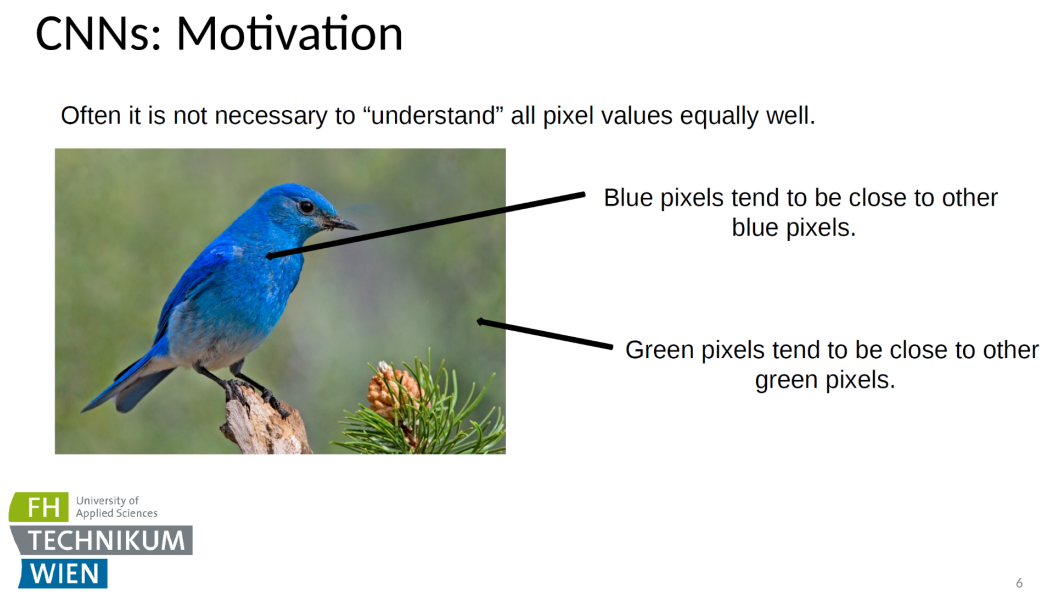
Why? Because feedforward networks treat **each pixel as an independent input**. If the position of a key feature—like the eye or wing—shifts just slightly, its numerical location changes, and the network doesn't recognize it.

CNNs solve this by using **local filters** that scan across the image. These filters don't care where the eye is—they look for *patterns*, not pixel positions.

This gives CNNs something called **translational invariance**—the ability to still recognize an object even if it shifts around a little.

This makes CNNs much more robust and "human-like" in how they interpret images.

Let's now see how they do this by looking at **patterns and neighborhoods** in the next slide.



[Gesturing toward the image of the bird]

Now, here's an even more intuitive reason why CNNs are so effective at handling image data.

Key insight: Not all pixel values need to be understood in isolation.

In natural images, there's **local consistency**. For example:

- **Blue pixels**—like those forming the bird—tend to cluster together.
- **Green pixels**—like the pine needles—also tend to appear in groups.

This is called **spatial correlation**.

CNNs take advantage of this property:

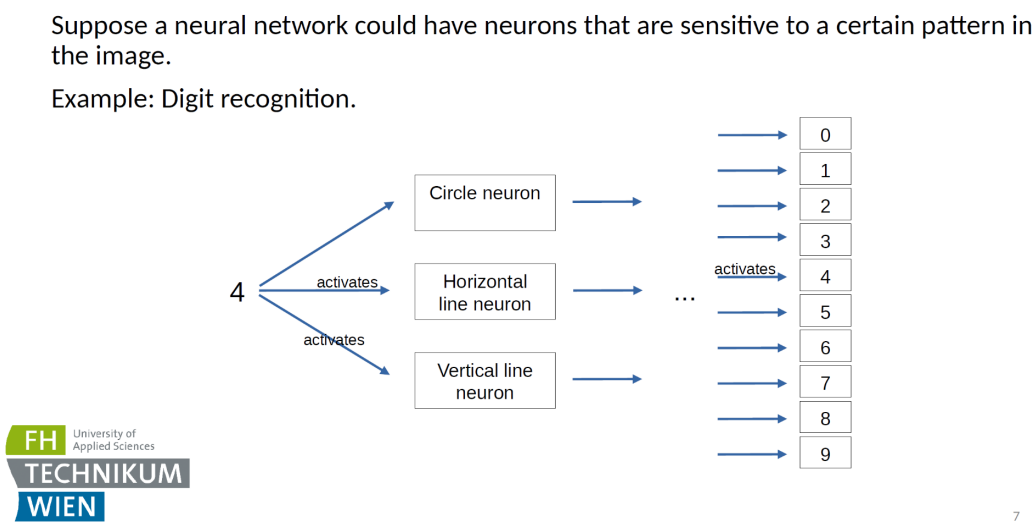
- Instead of analyzing each pixel independently, they **analyze small neighborhoods**—small patches like 3×3 or 5×5 windows—and learn to detect patterns *within* them.
- These learned patterns, or **features**, are then reused across the entire image via **convolutional filters**.

This results in models that:

- Learn faster,
- Use fewer parameters,
- Generalize better,
- And are more robust to minor variations.

So rather than needing to "understand" every single pixel, CNNs ask: *what kind of structure lives in this local region?*

Up next: let's see how CNNs begin to build abstract concepts—like recognizing shapes or lines. Next slide, please.



[Hands raised, walking audience through the idea]

Let's now build the **core intuition** of what CNNs *learn*.

Imagine that a neural network could have **specialized neurons**—each one trained to respond to a certain **visual pattern**:

- A **horizontal line**,
- A **vertical line**,
- Or a **circular shape**.

Let's say we're doing **digit recognition**, and we show the network the digit **4**:

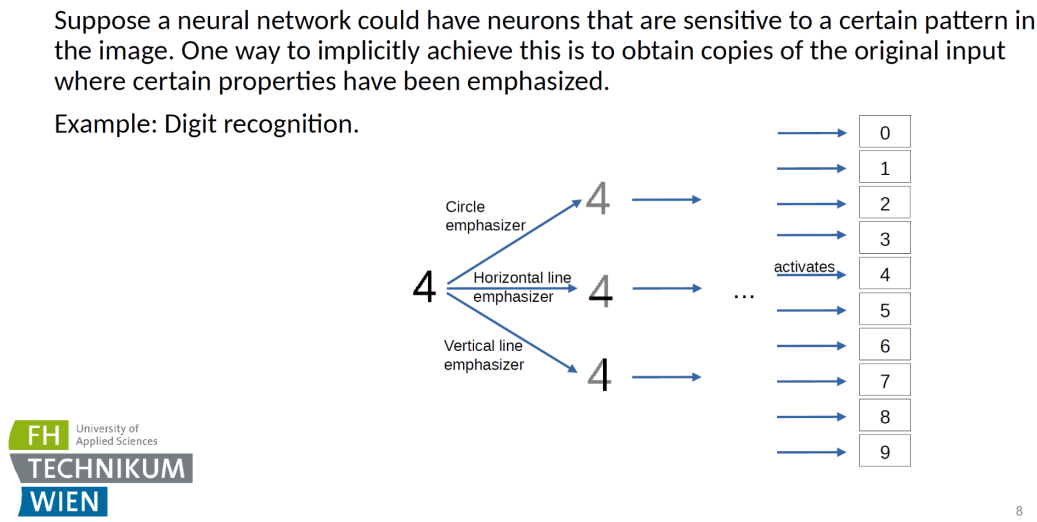
- The **vertical line detector** fires because of the upright stroke.
- The **horizontal line detector** activates because of the top bar.
- Maybe even a partial **circle detector** lights up if there's curvature in the design.

Each of these **pattern-sensitive neurons** becomes a kind of **feature detector**, and their activations **flow into the next layer**, where the network can conclude: *Aha! This is a 4.*

This modularity—building meaning from **basic shapes** to **complex objects**—is a key strength of CNNs.

And the beauty is: CNNs **learn these filters automatically**, without needing to explicitly tell the model "This is a circle."

In the next slide, we'll explore how we can simulate this idea using **image filters or 'emphasizers'**. Ready? Next slide, please.



[With energy, gesturing to the diagram]

Here's how CNNs **build those detectors** we talked about—*without needing us to hand-code them.*

Suppose we have an image of the digit **4**.

Now imagine we apply **three different filters**, also called *emphasizers*:

- One filter emphasizes **circles**,
- Another one emphasizes **horizontal lines**,
- And the third emphasizes **vertical lines**.

These filters act like **visual sieves**, each pulling out a specific type of feature from the original image. The output is not one image, but **multiple transformed copies**, each highlighting a different property of the digit.

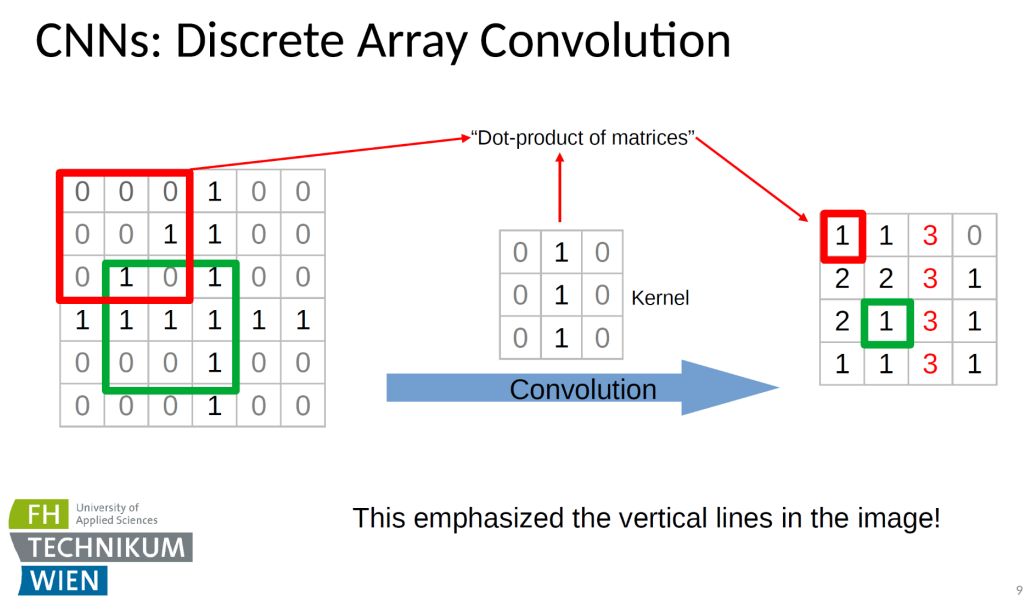
This is exactly how a **convolutional layer** works:

- It applies several **kernels** across the image,
- Each kernel acts like a pattern detector,
- The resulting **feature maps** are stacked together and passed to the next layer.

And here's the kicker: CNNs **learn these kernels** through training. You don't need to define what a "circle" is—the network figures out, "Ah, this shape seems to help me classify 4s, so I'll learn to detect it."

The result? Robust, flexible recognition that works for handwritten digits, cats and dogs, tumors in CT scans, and beyond.

Next, let's formally define how these filters work through **convolution operations**. Slide, please.



[Switching tone to more technical, engaging with pointer]

This is the **core operation** behind CNNs: the **discrete convolution**.

Let me walk you through what's happening here:

On the **left**, we have a **small image** represented as a grid of numbers. These numbers might represent intensities—white, gray, black.

In the **middle**, we have a **kernel**—a small matrix (3×3 in this case). This kernel acts as a **pattern detector**.

In this example, the kernel is designed to detect **vertical lines**. How?

```
[0 1 0]
[0 1 0]
[0 1 0]
```

Only the **center column** contributes to the sum; everything else is ignored.

What happens during convolution?

- We **slide** the kernel across the image, one patch at a time.
- At each location, we compute the **dot product** between the kernel and the overlapping region of the image.
- The result is written into the **output feature map** on the right.

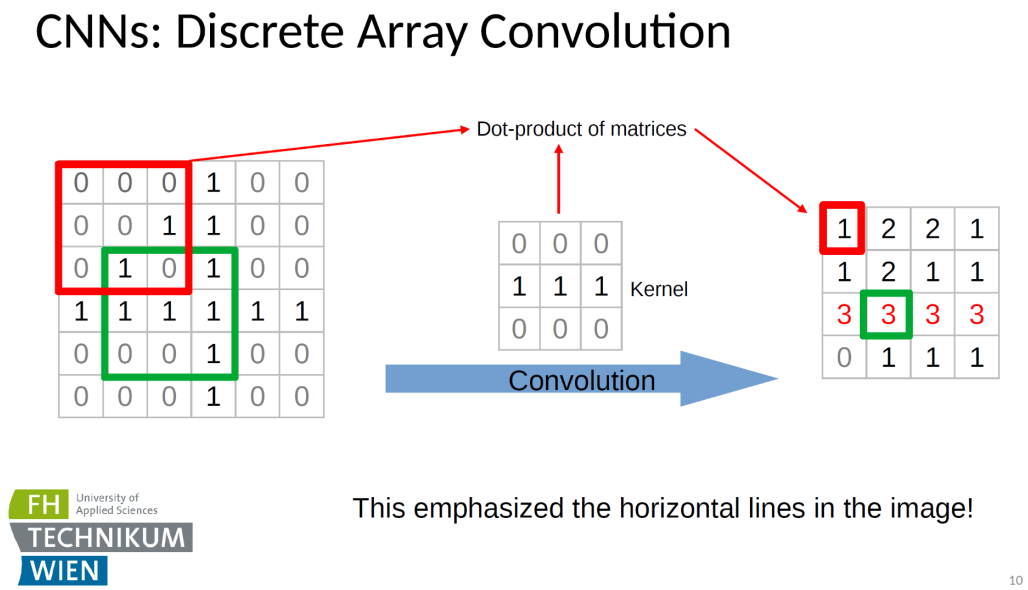
The result?

- Areas of the image that contain **vertical features** produce high values.
- Areas that don't match the pattern produce low or zero values.

This way, the output feature map becomes a **highlighted version** of the image, showing where vertical structures exist.

This is just the beginning—next we'll see how a different kernel emphasizes **horizontal lines** instead.

Next slide, please.



[Stepping to the other side of the screen]

Here's another **convolution** example—this time, we're emphasizing **horizontal lines**.

Let's take a closer look at the **kernel** in the center:

```
[0 0 0]
[1 1 1]
[0 0 0]
```

This kernel will light up **horizontal features** in the image. Why?

- It sums the values in the **middle row** of each patch.
- If there's a strong horizontal line there, the dot product will be large.

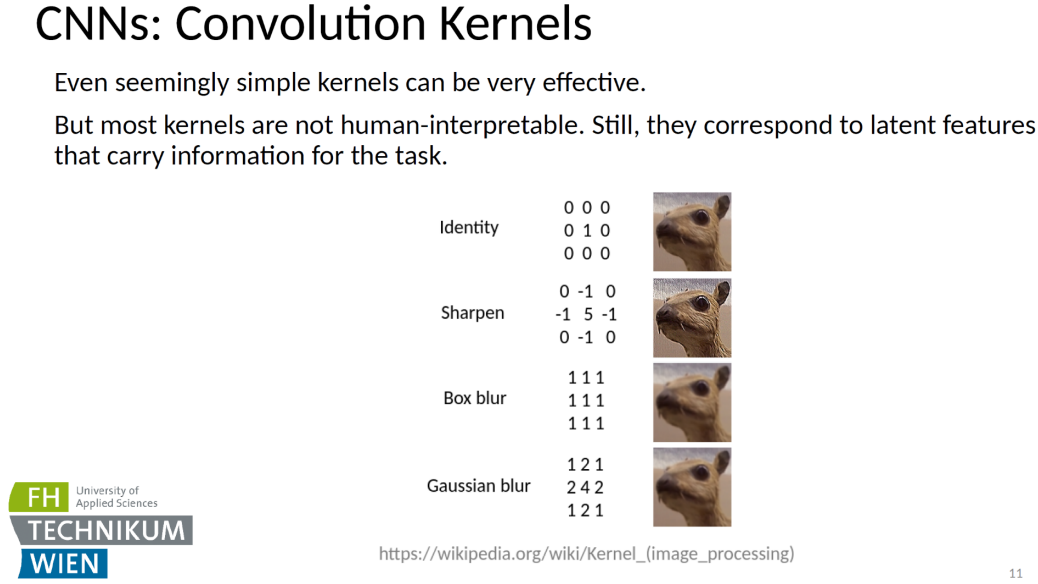
Look at the output feature map on the right:

- You can see **high values (3s)** where the original image had **strong horizontal patterns**.
- The rest of the image shows lower values—meaning the kernel didn't find its desired structure there.

This simple example demonstrates the **directional sensitivity** of different kernels. By combining many such filters, CNNs can detect edges, corners, textures—and eventually entire objects.

That's how CNNs **learn to see**.

Next up, we'll see how **even strange-looking kernels**—ones that humans can't interpret—can still be useful. Slide, please.



[Calmly, with illustrative tone]

Let's now talk about **convolution kernels** in general—and what they *do*.

Kernels are just **small matrices** used to transform an image. You've seen some already—vertical and horizontal edge detectors.

But look here—on the left we have some **classical kernels** from image processing:

- **Identity kernel**: Doesn't change the image.
- **Sharpen kernel**: Highlights edges and details.
- **Box blur** and **Gaussian blur**: These create smoothing effects—useful for noise reduction or softening.

On the right, you can clearly see their effect on a squirrel's face:

- The sharpened version has **crisper lines**.
- The blurred versions look **softer and fuzzier**.

But here's the key point:

> **Most CNN kernels are not human-interpretable.**

Once trained, the learned filters don't necessarily correspond to intuitive things like "circle" or "edge". But that doesn't mean they're random—far from it.

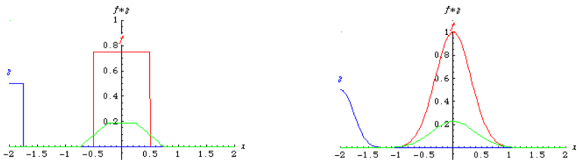
They represent **latent features**—statistical patterns that the network has found **useful for the task**.

The cool thing? CNNs learn to combine these low-level kernels into **increasingly complex and abstract features**: from lines → to textures → to eyes, paws, digits, tumors, logos… and more.

Now, let's formalize how these kernels get embedded into CNNs using convolutional layers. Next slide, please.

Aside: Convolution in Calculus

You may have encountered the convolution of functions with (smoothing) kernel functions in calculus.
This is a common approach to smooth an irregular function, for example to obtain a smooth empirical density in statistics.



FH University of Applied Sciences TECHNIKUM WIEN

[Image source: http://mathworld.wolfram.com/Convolution.html]

12

[Briefly stepping aside with academic tone]

Before we dive deeper into CNN architecture, let's take a **quick detour** for those with a mathematical background.

You may have seen **convolution** before—in **calculus** or **probability theory**.

In those settings, convolution is used to:

- **Smooth functions**,
- Combine distributions,
- Or even compute moving averages.

Here's the general idea:

> You take one function and **slide it across another**, multiplying and integrating as you go. The result is a *blended*, often smoother function.

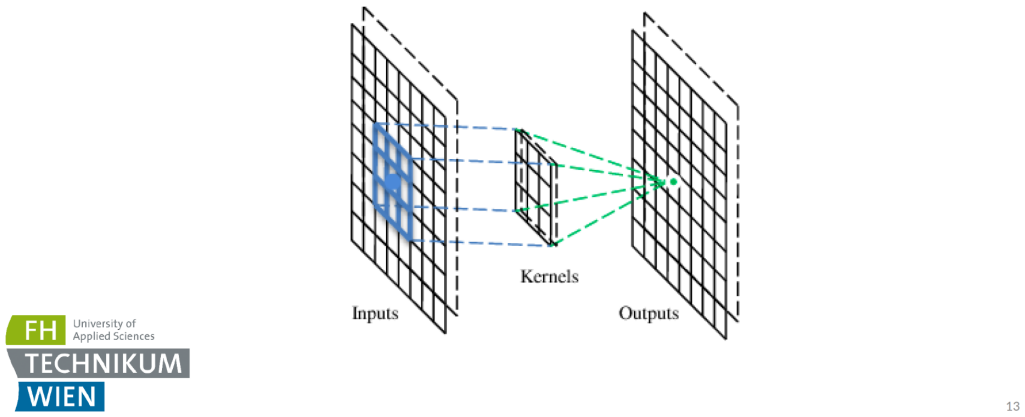
In the graphs here:

- The **blue** curve is the original function.
- The **green** is the kernel (smoothing function).
- The **red** is the **convolution result**.

This is the continuous version of what CNNs do discretely with matrices. And while we use sums instead of integrals in images, the **concept is identical**: combining one pattern with another to produce something more structured or simplified.

So, CNNs aren't just deep learning hacks—they're built on **centuries-old mathematical principles**.

Alright, math interlude over. Let's now look at how these convolutions are built directly into the layers of a CNN. Next slide, please.



[Turning to the architecture diagram]

Here it is—the **building block** of a Convolutional Neural Network: the **convolutional layer**.

A CNN is simply a **neural network that contains these layers**. And what makes them special is this:

> Each convolutional layer learns a **set of kernels**, and each kernel gets **convolved** across the input image to generate an **output feature map**.

In the diagram:

- The **left grid** represents the input image.
- The **blue patch** is the region currently being processed by a **kernel**.
- The **middle matrix** is the kernel sliding over the image.
- The result of that sliding—dot product after dot product—is written into the **output grid** on the right.

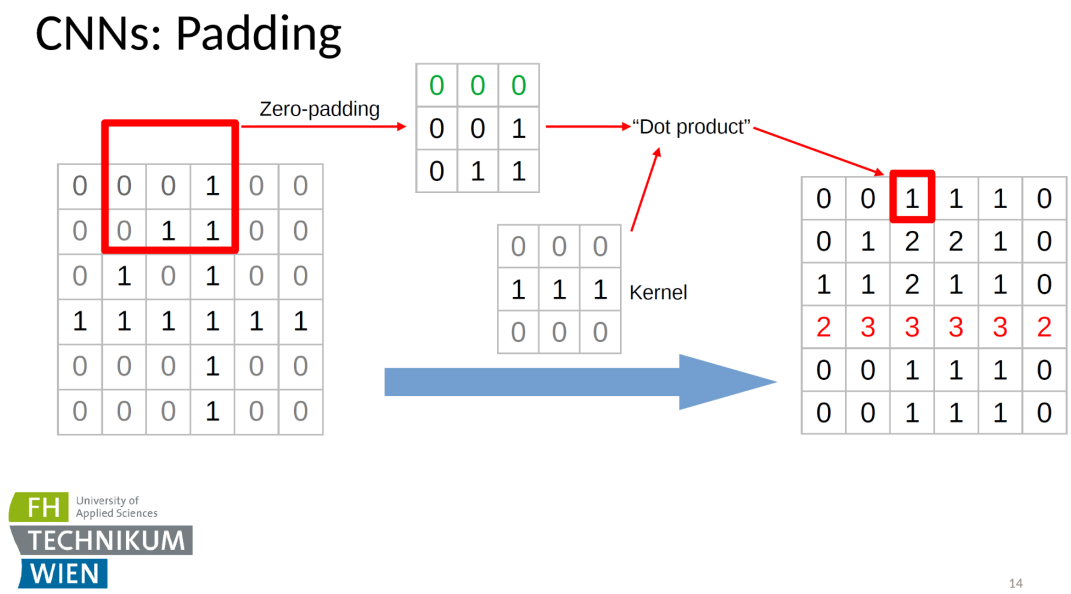
So:

- Each point in the **output** is like a **summary** of a neighborhood in the input.
- The CNN **learns** which patterns are most useful by adjusting the values in those kernels via gradient descent.

And here's the magic: a single convolutional layer doesn't just use one kernel. It uses **dozens**, even **hundreds**—each generating its own feature map, capturing different aspects of the input.

The result is a rich, multi-channel representation of the image—full of **edges**, **shapes**, **textures**, and **latent concepts**.

Let's now move to the next topic—**padding**—and see how CNNs handle borders of the image. Next slide, please.



[Calmly guiding with finger across image]

Now let's address an important technical detail: **padding**.

Question: What happens when our kernel reaches the **edge** of the image?

If we don't pad the image, the kernel can't center itself on border pixels—so the output shrinks with every layer. That's bad news for deep networks.

Solution: Add a border of **zeros** around the image—this is called **zero-padding**.

In the image:

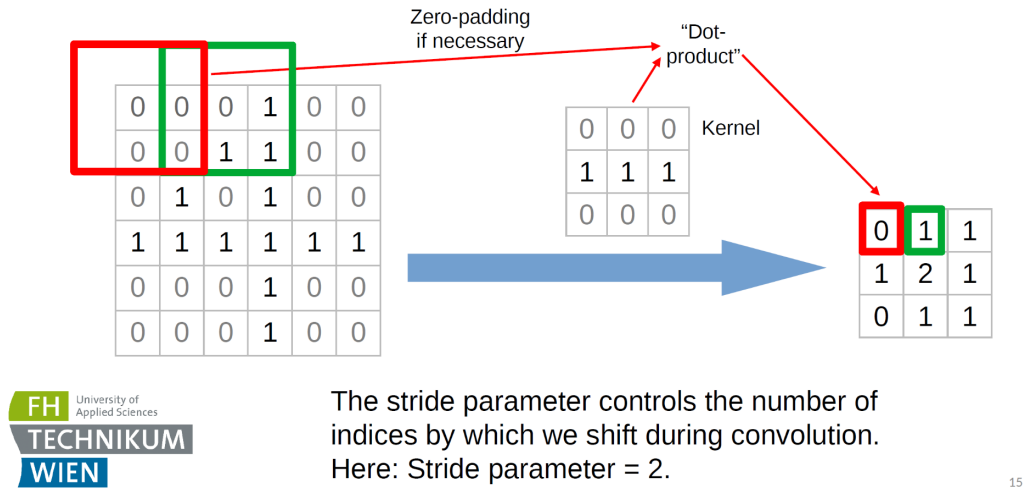
- We see a **3×3 kernel** trying to scan over the corner of an image.
- Without padding, we'd lose valuable edge data.
- With padding, the kernel can still slide over *every* position in the original input.

The result?

- The **output dimensions stay closer to the input** size.
- The model has **full coverage**, even near the borders.
- Padding ensures consistency in **feature map dimensions**, which is essential for building deep architectures.

There are also more advanced padding types—like "same" and "valid" in Keras—but zero-padding is the classic and most intuitive.

Next, we'll see how we can **control how fast we move** the kernel using something called the **stride parameter**. Slide, please.



[Stepping forward with clarity]

Here we introduce the **stride parameter**, which controls how far the kernel moves across the image at each step.

Normally, a kernel slides **one pixel at a time**—this is **stride = 1**.

But what if we set the **stride to 2**? That means the kernel **jumps two pixels** at every move—both horizontally and vertically.

The impact?

- The **output feature map becomes smaller**.
- We reduce the **computational load**.
- It's a form of **downsampling**, similar to max-pooling, but embedded directly in the convolution process.

In the example:

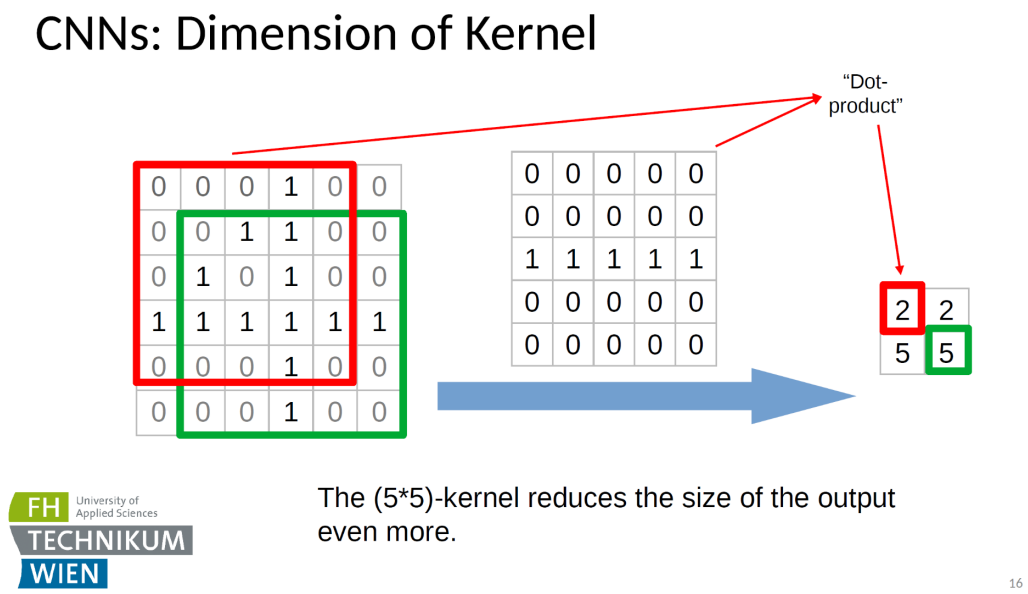
- The kernel is a 3×3 matrix.
- It scans over the image with a stride of 2.
- This skips over intermediate pixels—notice the **gaps** between the red and green patches.

Stride helps us:

- Control resolution,
- Reduce data volume,
- And **build hierarchical representations** more quickly.

But too large a stride? And we might **miss important details**—so it's a design trade-off.

Next up: what if we want to zoom out even further? We increase the **kernel size** itself. Let's see that on the next slide.



[Gesturing to the red and green windows]

Now we explore another important hyperparameter: the **kernel size**, also called **receptive field**.

Up to now, we've mostly used **3×3 kernels**. But in this example, the kernel is **5×5**.

What does this mean?

- It covers a **larger neighborhood** of pixels at once.
- It captures **broader features**—like thick lines, large curves, or textures.
- But it also reduces the **spatial resolution** of the output.

Why does the output shrink more? Because the bigger the kernel, the fewer positions it can slide across the image—unless we use extra padding.

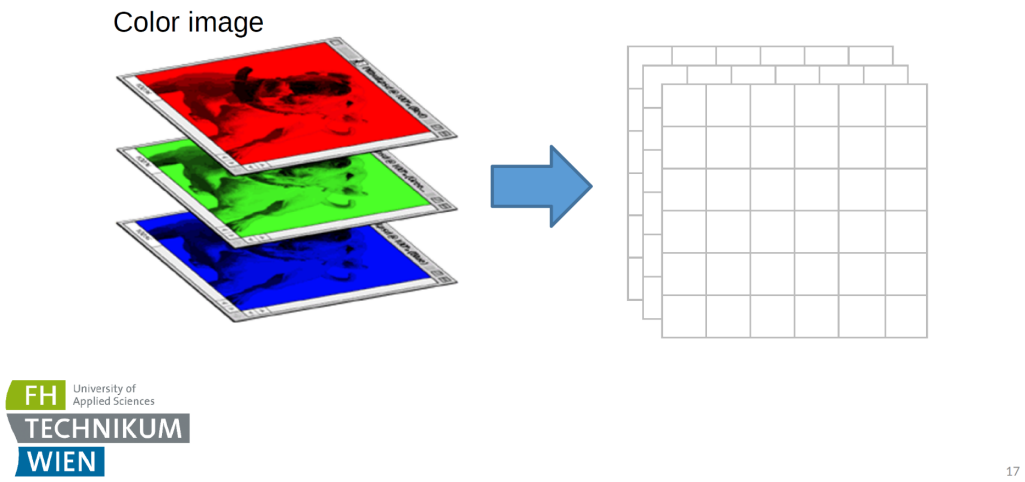
Trade-offs of larger kernels:

- Capture more global context,
- Smooth over noise,
- Fewer positions = lower resolution,
- More parameters to learn.

In practice:

- **3×3** kernels are the most popular—they balance detail and efficiency.
- **5×5** or **7×7** are used occasionally in early layers or for very large inputs.

Let's now shift from grayscale images to something more realistic: **color images** with **multiple channels**—slide, please.



[Back to visual reality, gesturing at RGB layers]

Now let's look at **real images**—in full color.

A color image isn't just one 2D grid—it's actually **three layers**:

- One for **Red**,
- One for **Green**,
- One for **Blue**.

Together, these form an **RGB image**.

So when a CNN receives this input, it doesn't just process one matrix—it gets a **3D volume**: width × height × 3 channels.

Now here's the key point:

> Each **convolutional kernel** also spans across all **input channels**.

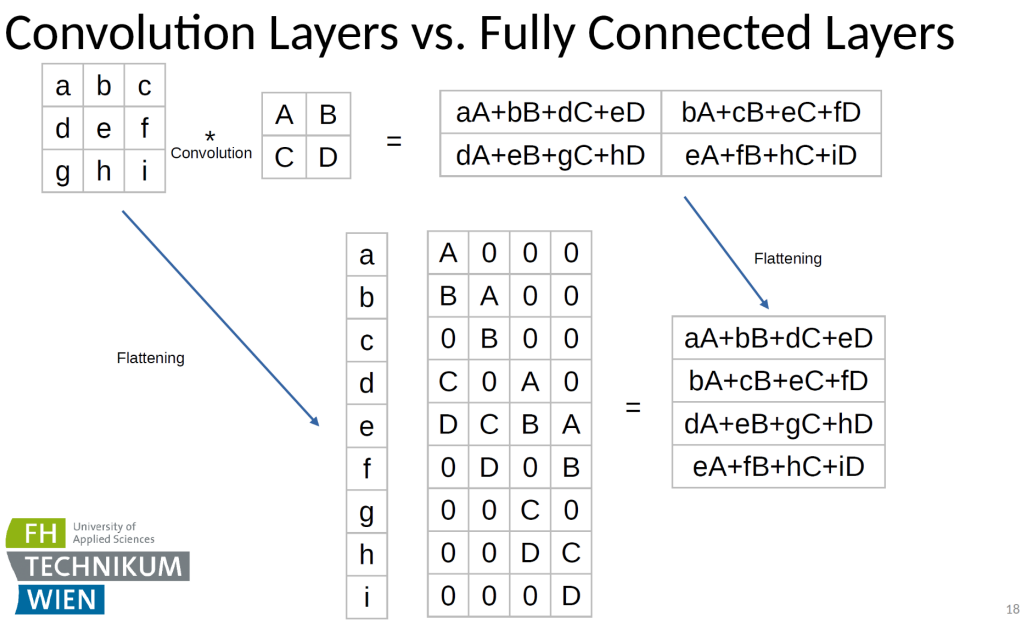
That means:

- If your input has 3 channels (RGB),
- Then your kernel has a shape like **3×3×3**—a small window, but applied *in depth* across all color layers.
- The result is a **single output value**—per spatial location—for that kernel.

Stack multiple kernels? You get **multiple output channels**.

This is how CNNs go from 3-channel inputs → 16-channel feature maps → 32 → 64 → …

Let's now look at a more detailed numerical example of how multi-channel convolution works in practice. Slide, please.



[With a whiteboard-style explanation]

Let's now compare **convolutional layers** and **fully connected (dense) layers**—from a mathematical perspective.

On the **top**, we see a 3×3 input matrix:

```
[a b c]
[d e f]
[g h i]
```

And a 2×2 kernel:

```
[A B]
[C D]
```

When we **convolve** the kernel over the input, we perform a **dot product** at each valid position. The resulting matrix (on the right) contains **weighted sums** of local neighborhoods.

This produces a **sparse, localized connection**:

- Only a subset of weights (those in the kernel) interact with each part of the image.
- We reuse the same kernel everywhere—so **fewer parameters** are needed.

On the **bottom**, we "flatten" everything:

- The input becomes a vector: `[a b c d e f g h i]`
- And the kernel becomes a large matrix of weights with carefully aligned values, simulating the same local dot-products.

This is how a **fully connected layer** would emulate a convolution—but with:

- **Many more parameters**
- **No weight sharing**
- **No locality preserved**

Summary:

- **Convolution layers** are efficient, local, and translation-friendly.
- **Fully connected layers** are brute-force—good for classification at the end, but not suitable for visual feature extraction.

Now let's wrap this part up by explicitly contrasting these two architectures—convolutional vs fully connected—based on bias, variance, and learning efficiency. Next slide, please.

Convolution Layers vs. Fully Connected Layers

- Suppose the input array has size n*n.
- A k*k convolutional layer is equivalent to a certain n^2 to (n-k+1)^2 linear layer.
- The set of such convolutions "is" a (k*k)-dimensional subset of the n^2 (n-k+1)^2-dimensional set of matrices.
- Due to lower-dimensional subspace:
  - More bias
  - Less variance
  - Easier to learn
- Why do you think that convolutional layers work (in computer vision)?

FH University of Applied Sciences TECHNIKUM WIEN

19

[Turning to the theoretical side]

Now let's dive a bit deeper into the **math and reasoning** behind why convolutional layers outperform fully connected layers in **computer vision**.

Assume: Our input is an $n \times n$ image.

A standard **dense layer** would connect each of the n^2 inputs to each of the outputs—massive parameter explosion.

But a **k×k convolutional layer** only looks at small local patches, and uses **the same k×k weights everywhere**.

That means:

- Instead of learning an $n^2 \times n^2$ matrix,
- We're learning a set of **much fewer weights**—the **k×k** kernel(s).

Mathematically:

- A convolutional layer operates in a **k²-dimensional subspace** of all possible linear maps from n^2 to $(n-k+1)^2$.
- So it's more **structured**, more **constrained**.

The upside of this **lower-dimensional space**:

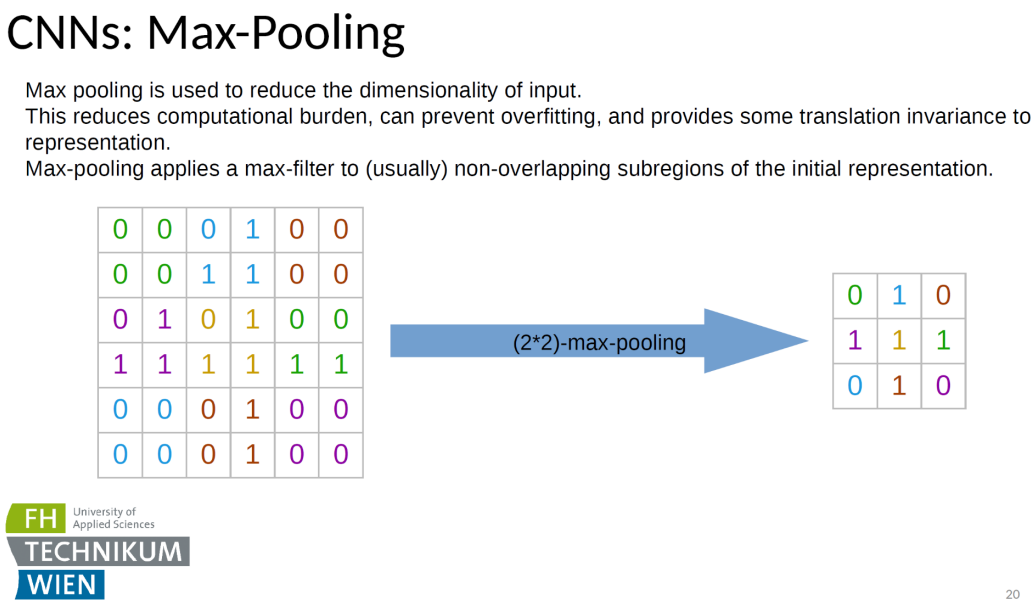
- **More bias** — yes, in the statistical sense, but that's a good thing in small data settings.
- **Less variance** — fewer weights, less risk of overfitting.
- **Easier to learn** — smaller search space = faster convergence.

That's why CNNs shine in vision: they impose **inductive bias** that reflects the **spatial locality and translation invariance** of images.

Final food for thought:

> Why do you think convolutional layers work so well *specifically* in computer vision? (Hint: structure, patterns, and position-independent features!)

Let's now see how we further compress spatial data—and gain translation invariance—through **pooling layers**. Next slide, please.



[Resuming energetically, pointing at arrow]

Let's talk about a powerful yet simple idea: **max-pooling**.

The main goals of max-pooling are:

- **Reduce dimensionality**,
- **Decrease computation**,
- **Introduce translation invariance**, and
- **Prevent overfitting**.

What is it?

> Max-pooling slides a window—typically 2×2—over the input, and keeps only the **maximum value** in each window.

In this example:

- We apply **(2×2)-max-pooling** over the image.

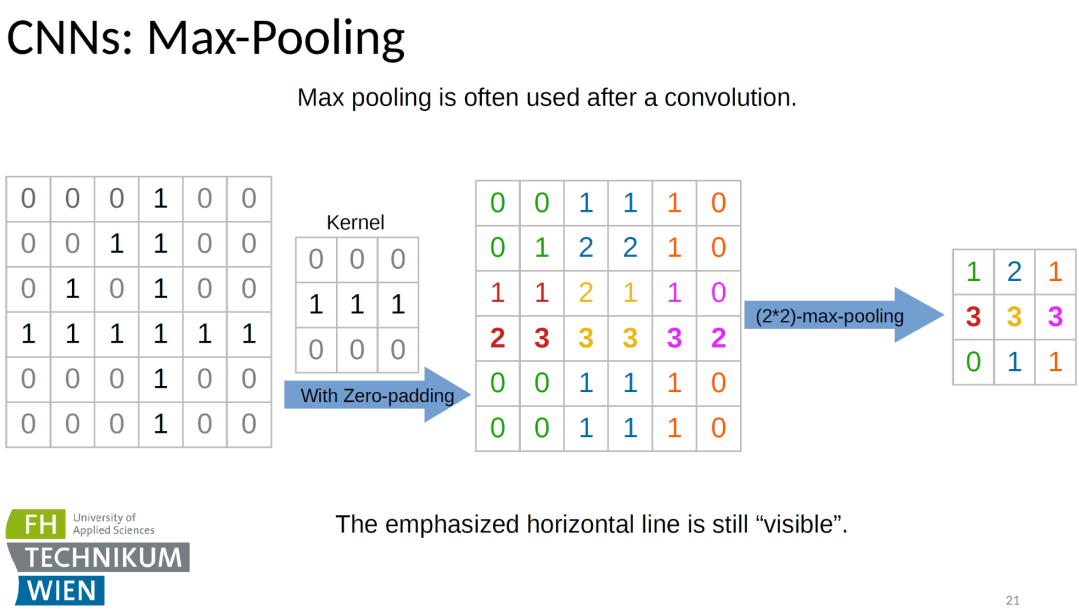
- For each 2×2 block in the input, we take the highest number.
- Result: A **smaller** matrix that still highlights the most **salient features**.

What does this mean in practice?

- A strong edge or pattern remains, even if it **moves slightly**—CNNs become more **tolerant to shifts** in the input.
- We aggressively reduce the **spatial resolution**, allowing deeper layers to operate more efficiently.

Tip: You don't learn anything in max-pooling—it's a **fixed operation**.

Next, we'll look at how max-pooling works together with convolution in a CNN pipeline. Slide, please.



[Confidently walking through the visual pipeline]

Here's a **complete example** of how **convolution and max-pooling** work together.

On the **left**, we begin with a basic input grid.

- We apply a **convolution** using a kernel that emphasizes **horizontal lines**.
- Notice the row of 1s in the kernel—it's tuned to detect horizontal intensity.

The result—shown in the middle—is the **feature map** after convolution.

- High values appear where **horizontal patterns** are strong—like the row with all 1s.

But we're not done yet.

Next, we apply **(2×2) max-pooling**, shown on the **far right**:

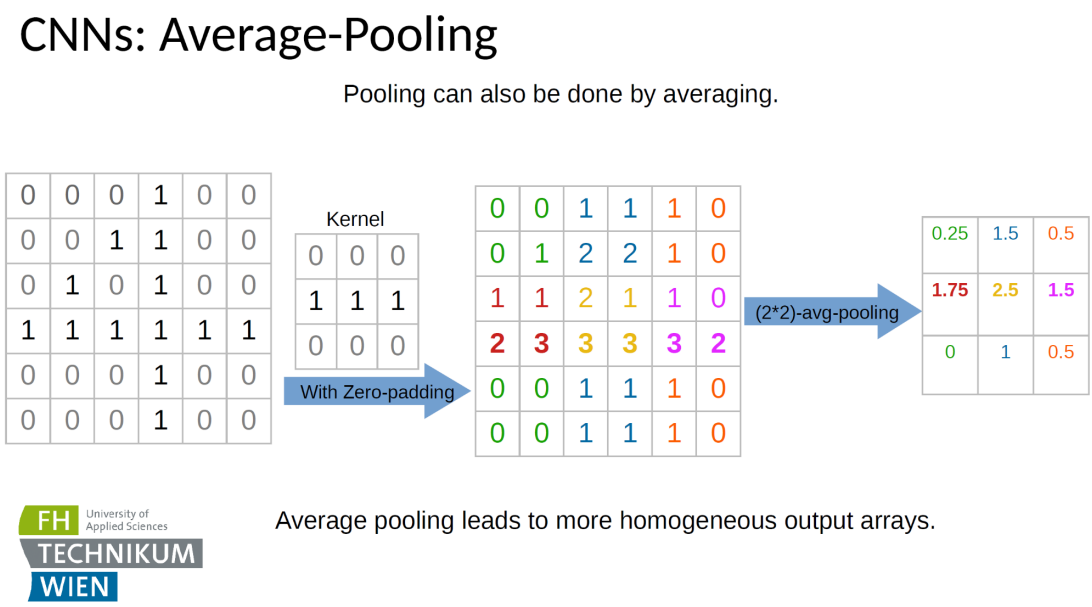
- Each 2×2 block is replaced with its **maximum value**.
- The overall size is reduced, but…
- The **important pattern**—the emphasized horizontal line—is still **visible**.

This is a core idea in CNNs:

> You **compress the data**, but **retain the meaning**.

Pooling not only reduces computation—it keeps the strongest activations and thus the most relevant features.

Next slide: let's compare this with **average pooling** and how it changes the outcome.



[Shifting tone slightly to contrast the methods]

You've seen **max-pooling**—now let's look at its sibling: **average pooling**.

Instead of taking the maximum value in each block, average pooling simply takes the **mean**.

In this example:

- The feature map on the left is **identical** to the one from the max-pooling slide.
- But when we apply **(2×2) avg-pooling**, we compute the **average value** of each 2×2 block.

The result on the right is a **smoother, more homogeneous matrix**.

This method:

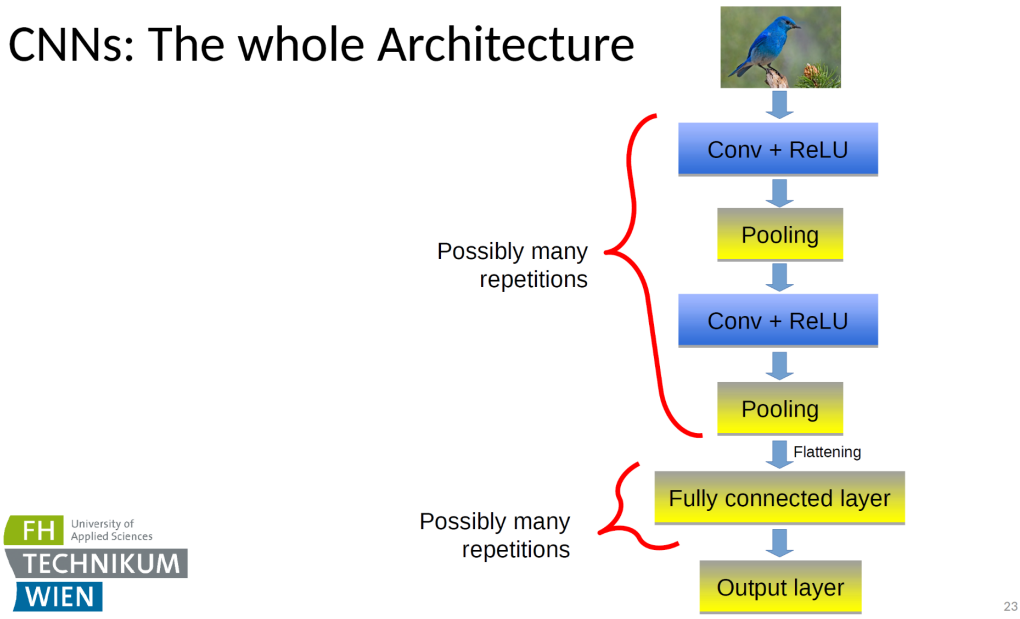
- Keeps more **global information**,
- Is less aggressive at discarding values,
- And can be useful when you want **soft features**—like in tasks involving smooth transitions, attention maps, or regression.

But: It may also **dilute sharp signals**, which are often critical for classification. That's why:

> **Max-pooling is more common in practice**, especially for image recognition tasks.

Still, average pooling has its place in architectures like **ResNet** and **MobileNet**, often near the output layers.

Next, let's see how all these pieces come together into a **full CNN architecture**. Ready? Next slide, please.



[Wrapping things together like a chef presenting a finished dish]

Now we see the **full picture**: this is the **typical architecture** of a Convolutional Neural Network.

It starts with a raw image—like our bluebird friend.

Then we have **three core components** repeated in blocks:

---

1. **Convolution + ReLU**
   - The convolution layer detects local patterns—edges, textures, parts.
   - ReLU (Rectified Linear Unit) adds non-linearity—keeping only positive activations.
2. **Pooling**
   - Typically max-pooling.
   - Downsamples the feature maps, reduces computation, builds invariance.

These steps repeat—layer after layer, the network builds **hierarchies**:

- Early layers see edges.
- Mid layers see shapes and parts.
- Deeper layers understand entire objects.

---

3. **Flattening**
   - After pooling, the feature maps are flattened into a vector.
4. **Fully Connected Layers**

- These dense layers mix information globally.
- They combine all abstract features for final decision-making.

5. Output Layer

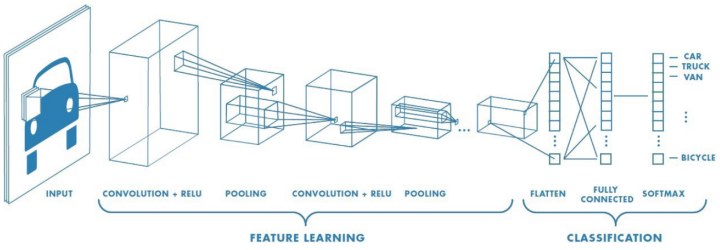
- Often a softmax layer.
- Outputs probabilities over classes—"Is this a bird, a cat, or a dog?"

What's the key?

> Each layer **extracts more abstract and useful features**, and CNNs **learn what to extract automatically**—from raw pixels to predictions.

Up next: we'll talk about what parts of the model actually get learned. Spoiler: it's mostly in the kernels and fully connected layers. Ready for the next slide?

CNNs: Parameters



Feature learning part:

- Values in the kernel matrices

Classification part:

- Weights of artificial neurons
- Biases of artificial neurons

FH University of Applied Sciences TECHNIKUM WIEN

24

[Explaining with laser pointer on the architecture flow]

Let's now zoom in on a key question:

> **What exactly does a CNN learn? Where are the parameters?**

This diagram beautifully splits the network into two major sections:

---

1. **Feature Learning** (left side):

- This is where **convolution** and **pooling** happen.
- The learnable parts here are the **values in the kernel matrices**.
- Each kernel learns to detect a certain feature—like edges, corners, or textures.
- These values are updated via **backpropagation**, just like in regular neural nets.

---

2. **Classification** (right side):

- After flattening, we enter the **fully connected layers**.
- Here we have standard **weights and biases** associated with each artificial neuron.
- These layers mix the high-level features from the convolutional part to form final decisions.

So in summary, CNNs learn:

- **Kernels** in the convolutional layers,
- **Weights and biases** in the fully connected layers.

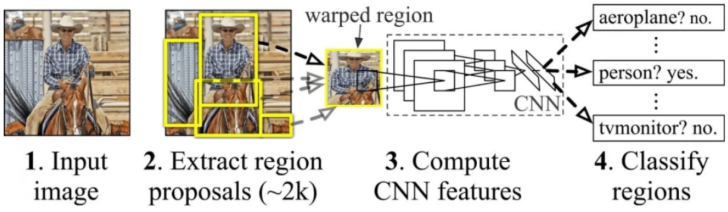
The pooling layers (max or average) do **not** have learnable parameters—they're fixed operations.

It's the combination of these learnable parts that allows CNNs to detect, abstract, and classify with such accuracy.

Let's now shift to some advanced CNN variants—starting with **Region-based CNNs** for object detection. Next slide, please.

Region-based CNNs (RCNNs)

Utilizes bounding boxes across regions, then evaluates convolutional networks independently on regions of interest to identify different objects in one image.



Concept of R-CNN – Region-based Convolutional Networks

https://viso.ai/deep-learning/mask-r-cnn/

FH University of Applied Sciences TECHNIKUM WIEN

25

[Introducing with excitement]

Now we're stepping into the realm of **object detection**—where CNNs not only classify images, but also **find and label multiple objects within a single image**.

This is the world of **Region-based CNNs**, or **RCNNs**.

Here's how it works, step-by-step:

---

1 **Input image**: We start with a raw photo—could be a scene with people, vehicles, animals…

2 **Region proposals**: We generate a set of **candidate regions**—areas that might contain an object. This can be done with algorithms like selective search.

Usually, we get around **2,000 region proposals** per image.

3 **Warp and send through CNN**: Each region is resized (warped) to a fixed size and passed through a **standard CNN** to extract its **feature vector**.

4 **Classification**: The features are then passed to **classifiers**—typically one-vs-all SVMs—to determine what each region contains:

- Person?
- Airplane?
- Dog?

RCNNs combine **localization** (where is the object?) with **recognition** (what is it?).

But… they're slow. Running a CNN on thousands of regions is expensive.

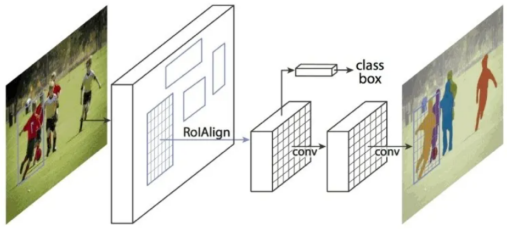
That's why they led to faster variants like **Fast RCNN**, **Faster RCNN**, and ultimately **Mask RCNN**, which we'll see next.

Slide, please: time to look at **segmentation with Mask RCNN**!

Mask RCNNs

Image segmentation is the process of partitioning a digital image into multiple segments (sets of pixels). Mask RCNNs additionally output the object mask.

Example: Organ/tumor segmentation in medical imaging.



Mask R-CNN – The Mask R-CNN Framework for Instance Segmentation

https://viso.ai/deep-learning/mask-r-cnn/

FH University of Applied Sciences TECHNIKUM WIEN

26

[Explaining with enthusiasm and hands illustrating segmentation]

Here we enter the most **precise and pixel-wise** world of computer vision: **Instance segmentation** with **Mask R-CNNs**.

Unlike classic RCNNs, which only draw **bounding boxes**, Mask RCNNs go one step further:

They output a **binary mask**—a pixel-by-pixel map—indicating exactly **which pixels belong to each object**.

Here's the flow:

1 **Input image** We begin with a scene—multiple people, animals, cars…

2 **Region proposals + RoIAlign** Mask RCNN generates **regions of interest (RoIs)** and uses a more precise tool called **RoIAlign** to extract consistent features for each region.

3 **Parallel branches**

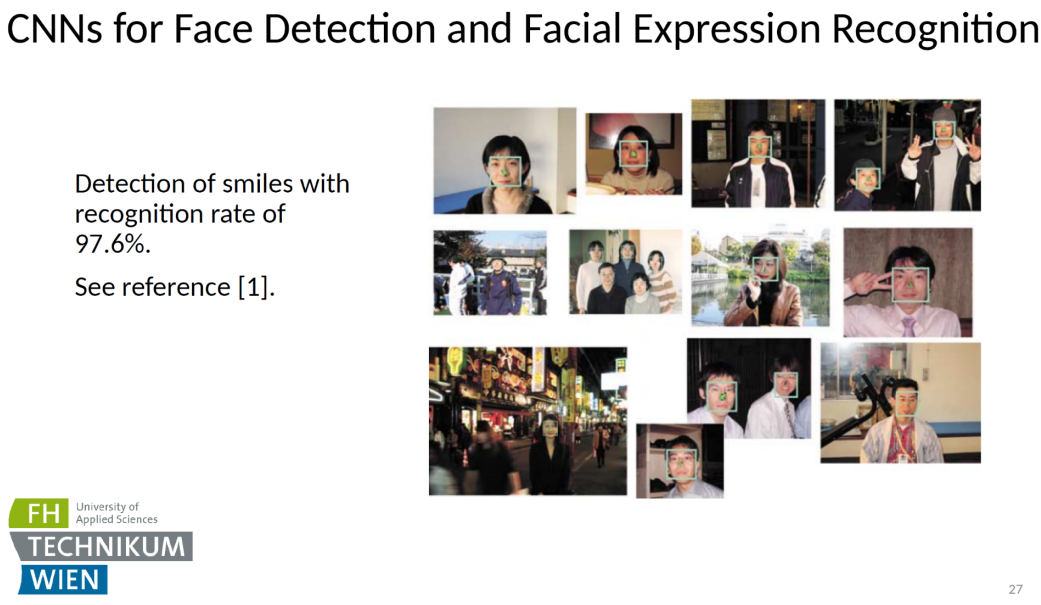
- One branch does classification (what is it?),
- One branch refines bounding boxes (where is it?),
- A **third branch** outputs a **segmentation mask**—a grayscale or binary image that paints the object's silhouette.

The result is what you see on the far right: Each object is **colored separately**, showing that the network not only *sees* the object—it knows exactly **where it begins and ends**.

This technique is widely used in:

- **Medical imaging** (e.g., tumor segmentation),
- **Autonomous driving** (detecting lane lines, pedestrians),
- **Robotics**, **AR**, and more.

Let's now explore a few cool real-world applications—including smiling humans and drug discovery! Next slide, please.



[With an upbeat tone, pointing to smiling faces]

Here we see one of the most intuitive—and impactful—applications of CNNs: **Face detection and facial expression recognition**.

In this case, CNNs were trained to:

1. **Locate faces** in images using bounding boxes.
2. **Classify facial expressions**, such as:
   - Smiling
   - Neutral
   - Angry
   - Surprised

This particular study achieved a smile detection rate of **97.6%**—that's nearly perfect!

How does it work?

- CNNs learn to recognize **facial features**—eyes, eyebrows, mouth contours.
- They detect **subtle patterns of emotion**, even under different lighting or poses.
- The network generalizes well across people and environments.

Applications include:

- Smartphone face unlock,
- Emotion-aware user interfaces,
- Retail or security surveillance,
- Mental health diagnostics.

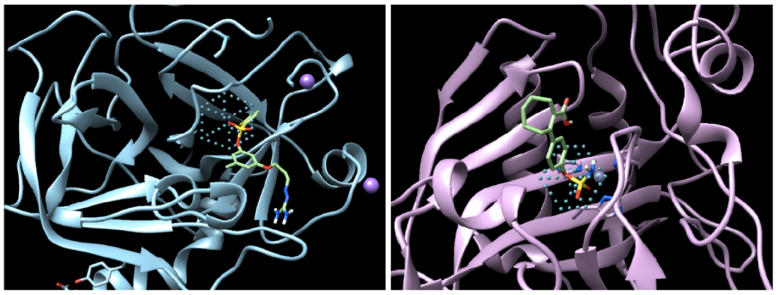
Coming up: a very different but equally powerful use of CNNs—in **drug discovery**. Next slide, please.

CNNs for Drug Discovery

Sulfonyl/sulfonamide detector. A CNN was able to infer a meaningful spatial arrangement of input atom types without any chemical prior knowledge.

Note: This is an application outside of computer vision.

See reference [2].



FH University of Applied Sciences TECHNIKUM WIEN

28

[Impressed tone, highlighting the versatility of CNNs]

And here's something that might surprise you:

CNNs are not limited to images at all.

Welcome to their application in **drug discovery**.

In this case, a CNN was trained to detect **sulfonyl and sulfonamide groups**—chemical structures relevant to pharmacology.

But here's the catch:

- The model had **no prior chemical knowledge**.
- It only had raw **spatial and atom-type data**—essentially a 3D grid of chemical properties.

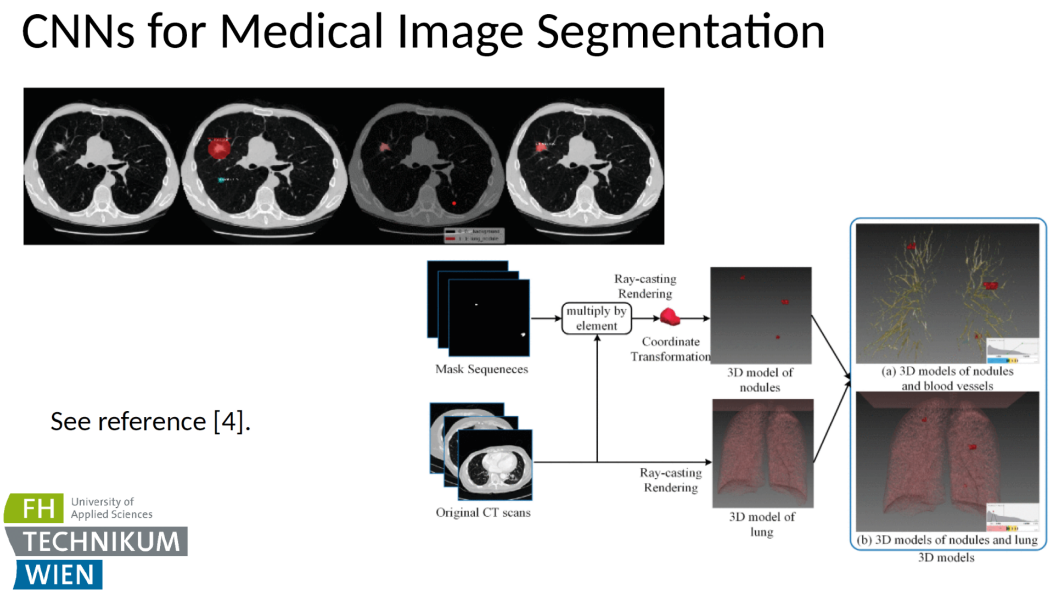
The CNN learned, by itself, to detect **meaningful molecular structures**—just like it would learn to find cat faces or digits.

This is incredibly powerful because it means:

- CNNs can help discover **binding sites**,
- Predict **molecular activity**,
- And suggest **new drug candidates**—all from raw molecular data.

So while CNNs were born in computer vision, they've grown into **scientific discovery tools**—no longer needing images in the traditional sense.

Now let's look at another vital domain where CNNs shine: **medical image segmentation**. Next slide, please.



[In a serious tone, highlighting real-world impact]

CNNs have revolutionized **medical diagnostics**—and this slide shows **why**.

This example focuses on **medical image segmentation**, particularly:

Identifying **lung nodules** in **CT scans** for early detection of diseases like cancer.

Here's what's happening:

1 **Input**: A series of **2D CT slices** (cross-sections of the body) are fed into a CNN.

2 **Mask generation**: The network generates **binary masks** highlighting suspicious regions—possible nodules.

3 **3D reconstruction**: These masks are **stacked** and processed using **ray-casting rendering**, creating **3D visualizations** of:

- The nodules,
- Blood vessels,
- And surrounding tissues.

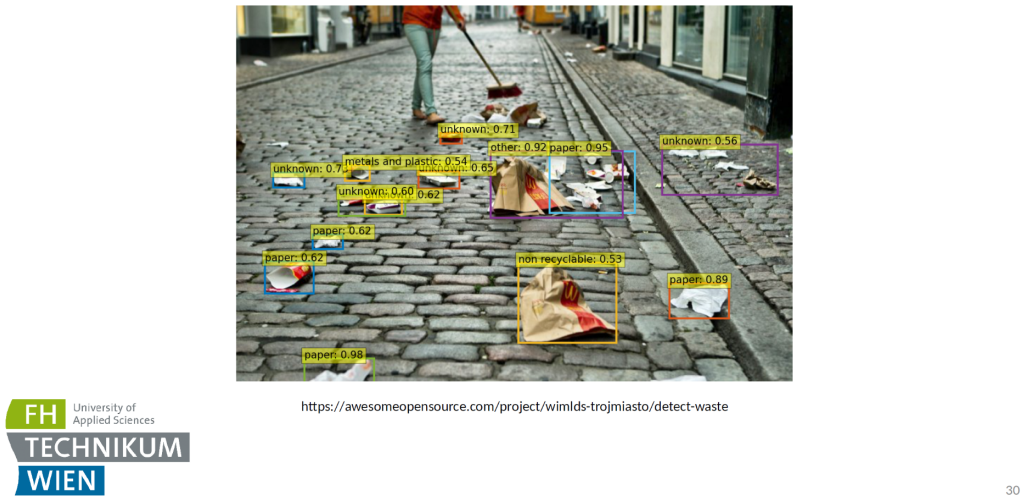
4 **Interpretation**: The output helps **radiologists** and **surgeons** visualize where tumors are—**in 3D**, from raw scans.

This goes far beyond classification. CNNs are:

- **Saving lives** by detecting tumors early,
- **Assisting diagnosis** with precision beyond human eyes,
- And enabling **personalized medicine**.

Techniques like U-Net or 3D CNNs are often used here—tailored to work with volume data.

Next up: a very different domain again—environmental impact through **waste detection**. Slide, please.



[Energetic wrap-up tone, pointing at bounding boxes]

And finally—an application that's both **practical and impactful**: **Waste detection** using CNNs.

In this example, a CNN model has been trained to:

- **Locate** trash on the street,
- **Classify** it into categories like:
  - Paper
  - Metals and plastic
  - Non-recyclable
  - Unknown

Each bounding box shows:

- The **type of waste**,
- The **confidence score**.

This kind of system can:

- Help **automated cleanup robots**,
- Assist in **urban environmental monitoring**,
- Support **recycling initiatives** by guiding sorting systems.

What's amazing is that this doesn't require fancy sensors—just **standard images** and a **good CNN-based detector**.

Even more, projects like this are open-source and community-driven—so anyone can contribute to making cities cleaner.

This shows us just how **versatile CNNs are**: From digit recognition… To tumors… To trash on the street.

That concludes the tour of CNN applications!

Do you want to move on to the **references and assignment** slide now?

References

[1] Matsugu M et al. Subject independent facial expression recognition with robust face detection using a convolutional neural network, Neural Networks 16 (2003) 555–559.

[2] Wallach I et al. AtomNet: A Deep Convolutional Neural Network for Bioactivity Prediction in Structure-based Drug Discovery. arXiv:1510.02855.

[3] StatQuest with Josh Starmer: Neural Networks Part 8: Image Classification with Convolutional Neural Networks. https://www.youtube.com/watch?v=HGwBXDKFk9I

[4] Cai L et al. AtomNet: Mask R-CNN-Based Detection and Segmentation for Pulmonary Nodule 3D Visualization Diagnosis. in IEEE Access, vol. 8, pp. 44400-44409, 2020, doi: 10.1109/ACCESS.2020.2976432.

FH University of Applied Sciences TECHNIKUM WIEN

31